Mastering CMake

Release 20240719

Ken Martin, Bill Hoffman

(Traduzione: Baldassarre Cesarano)

19 lug 2024

Indice

CAPITOLO 1

Perché CMake?

Se si ha esperienza nella manutenzione del processo di build e di installazione di un pacchetto software, CMake risulterà interessante. CMake è un generatore di sistema di build open source per progetti software che consente agli sviluppatori di specificare i parametri di build in un formato testuale semplice e portatile. Questo file viene poi utilizzato da CMake per generare file di progetto per strumenti di build nativi, inclusi gli ambienti di sviluppo integrato (IDE) come Microsoft Visual Studio o Xcode di Apple, nonché UNIX, Linux, NMake e Ninja. CMake gestisce gli aspetti difficili della creazione di software come build multipiattaforma, introspezione del sistema e build personalizzate dall'utente, in un modo semplice che consente agli utenti di personalizzare facilmente le build per sistemi hardware e software complessi.

Per qualsiasi progetto, e in particolare per quelli cross-piattaforma, è necessario un sistema di build unificato. Molti progetti non basati su CMake vengono forniti con un Makefile UNIX (o Makefile.in) e un'area di lavoro Microsoft Visual Studio. Ciò richiede che gli sviluppatori cerchino costantemente di mantenere entrambi i sistemi di build aggiornati e coerenti tra loro. Per indirizzare sistemi di build aggiuntivi, come Xcode, sono necessarie ancora più copie personalizzate di questi file, creando un problema ancora più grande. Questo problema è aggravato se si tenta di supportare componenti opzionali, come includere il supporto JPEG se libjpeg è disponibile sul sistema. CMake risolve questo problema consolidando queste diverse operazioni in un formato di file semplice e di facile comprensione.

Se ci sono più sviluppatori che lavorano su un progetto o più piattaforme target, il software dovrà essere creato su più di un computer. Data l'ampia gamma di software installati e opzioni personalizzate che sono coinvolte nella configurazione di un computer moderno, è probabile che due computer che eseguono lo stesso sistema operativo siano leggermente diversi. CMake offre molti benefici per ambienti di sviluppo su piattaforma singola e multi-macchina, tra cui:

- La possibilità di cercare automaticamente programmi, librerie e file header che potrebbero essere richiesti dal software in fase di build. Ciò include la possibilità di considerare le variabili di ambiente e le impostazioni del registro di Windows durante la ricerca.

- La capacità di creare una struttura di directory al di fuori dell'albero dei sorgenti. Questa è una funzione utile che si trova su molte piattaforme UNIX; CMake fornisce questa funzionalità anche su Windows. Ciò consente a uno sviluppatore di rimuovere un'intera directory di build senza timore di rimuovere i file sorgenti.
- La capacità di creare comandi complessi e personalizzati per file generati automaticamente come i moc() di Qt o i generatori di wrapper di SWIG. Questi comandi vengono utilizzati per generare nuovi sorgenti durante il processo di build che vengono a loro volta compilati nel software.
- La possibilità di selezionare componenti opzionali al momento della configurazione. Ad esempio, molte delle librerie di VTK sono opzionali e CMake fornisce un modo semplice per gli utenti di selezionare quali librerie sono create.
- La capacità di generare automaticamente aree di lavoro [workspace] e progetti da un semplice file di testo. Questo può risultare molto utile per i sistemi che hanno molti programmi o casi di test, ognuno dei quali richiede un file di progetto separato, in genere un noioso processo manuale da creare utilizzando un IDE.
- La possibilità di passare facilmente tra build statiche e shared. CMake sa come creare librerie e moduli shared su tutte le piattaforme supportate. Vengono gestiti complicati flag del linker specifici della piattaforma e su molti sistemi UNIX sono supportate funzioni avanzate come i path di ricerca per le librerie shared native.
- Generazione automatica di dipendenze di file e supporto per build parallele sulla maggior parte delle piattaforme.

Durante lo sviluppo di software cross-piattaforma, CMake offre una serie di funzionalità aggiuntive:

- La capacità di testare l'ordine dei byte della macchina e altre caratteristiche specifiche dell'hardware.
- Un singolo set di file di configurazione della build che funzionano su tutte le piattaforme. Ciò evita il problema degli sviluppatori che devono mantenere le stesse informazioni in diversi formati all'interno di un progetto.
- Supporto per la creazione di librerie shared su tutte le piattaforme che lo supportano.
- La capacità di configurare i file con informazioni dipendenti dal sistema, come la posizione dei file di dati e altre informazioni. CMake può creare file header che contengono informazioni come path di file di dati e altre informazioni sotto forma di macro `#define`. I flag specifici del sistema possono anche essere inseriti nei file header configurati. Ciò presenta dei vantaggi rispetto alle opzioni della riga di comando `-D` per il compilatore, poiché consente ad altri sistemi di build di utilizzare la libreria compilata di CMake senza dover specificare esattamente le stesse opzioni della riga di comando utilizzate durante la build.

1.1 La Storia di CMake

Dal 1999, CMake è in fase di sviluppo attivo ed è maturato al punto da diventare una soluzione collaudata per un'ampia gamma di problemi di build. Lo sviluppo di CMake è iniziato come parte di Insight Toolkit (ITK), finanziato dalla statunitense «National Library of Medicine». ITK è un grande progetto software che funziona su molte piattaforme e può interagire con molti altri pacchetti software. Per supportarlo, era necessario uno strumento di build potente ma facile da usare. Avendo lavorato in passato con sistemi di build per progetti di grandi dimensioni, gli sviluppatori hanno progettato CMake per soddisfare queste esigenze. Da allora CMake è cresciuto costantemente in popolarità, con molti progetti e sviluppatori che lo adottano per la sua facilità d'uso e flessibilità. L'esempio più eloquente di ciò è l'adozione riuscita di CMake come sistema di build del K Desktop Environment (KDE), probabilmente il più grande progetto opensource esistente.

CMake include anche il supporto per il test del software sotto forma di CTest. Parte del processo di test del software comporta la creazione del software, possibilmente l'installazione e la determinazione di quali parti del software sono appropriate per il sistema corrente. Ciò rende CTest un'estensione logica di CMake poiché dispone già della maggior parte di queste informazioni. Allo stesso modo, CMake contiene CPack, progettato per supportare la distribuzione cross-piattaforma del software. Fornisce un approccio multipiattaforma alla creazione di installazioni native per il software, facendo uso di pacchetti popolari esistenti come WiX, RPM, Cygwin e PackageMaker.

CMake continua a monitorare e supportare gli strumenti di creazione più diffusi non appena diventano disponibili. CMake ha rapidamente fornito supporto per le nuove versioni di Visual Studio di Microsoft e Xcode IDE di Apple. Inoltre, a CMake è stato aggiunto il supporto per il nuovo strumento di compilazione Ninja di Google. Con CMake, una volta scritti i file di input, si ottiene supporto per nuovi compilatori e sistemi di compilazione gratuitamente perché il supporto per essi è integrato nelle nuove versioni di CMake e non è legato alla distribuzione del software. CMake offre anche un supporto continuo per la cross-compilazione con altri sistemi operativi o dispositivi embedded. La maggior parte dei comandi in CMake gestisce correttamente le differenze tra il sistema host e la piattaforma target durante la cross-compilazione.

1.1.1 Perché Non Usare Autoconf?

Prima di sviluppare CMake, i suoi autori usavano il set esistente di strumenti di build disponibili. Autoconf combinato con Automake fornisce alcune delle stesse funzionalità di CMake, ma per utilizzare questi strumenti su una piattaforma Windows è necessaria l'installazione di molti strumenti aggiuntivi che non si trovano nativamente su una macchina Windows. Oltre a richiedere una serie di strumenti, autoconf può essere difficile da usare o estendere e impossibile eseguire alcune attività che risultano facili in CMake. Anche se si ha autoconf e il suo ambiente in esecuzione sul sistema, genera Makefile che costringeranno gli utenti alla riga di comando. CMake, d'altra parte, offre una scelta, consentendo agli sviluppatori di generare file di progetto utilizzabili direttamente dall'IDE a cui sono abituati gli sviluppatori Windows e Xcode.

Sebbene autoconf supporti le opzioni specificate dall'utente, non supporta quelle dipendenti in cui un'opzione dipende da un'altra proprietà o selezione. Ad esempio, in CMake si potrebbe avere un'opzione utente per far dipendere il multithreading dalla prima determinazione se

il sistema dell'utente ha il supporto multithreading. CMake fornisce un'interfaccia utente interattiva, che consente all'utente di vedere facilmente quali opzioni sono disponibili e come impostarle.

Per gli utenti UNIX, CMake fornisce anche la generazione automatica delle dipendenze che non viene gestita direttamente da autoconf. Il semplice formato di input di CMake è anche più facile da leggere e gestire rispetto a una combinazione di file Makefile.in e configure.in. La capacità di CMake di ricordare e concatenare le informazioni sulle dipendenze della libreria non ha equivalenti in autoconf/automake.

1.1.2 Perché non utilizzare JAM, qmake, SCons o ANT?

Altri strumenti come ANT, qmake, SCons e JAM hanno adottato approcci diversi per risolvere questi problemi e ci hanno aiutato a dare forma a CMake. Dei quattro, qmake è il più simile a CMake, sebbene manchi gran parte dell'interrogazione del sistema fornita da CMake. Il formato di input di Qmake è più simile a un Makefile tradizionale. Anche ANT, JAM e SCons sono multipiattaforma sebbene non supportino la generazione di file di progetto nativi. Si staccano dal tradizionale input orientato al Makefile con ANT che usa XML; JAM col proprio linguaggio; e SCons che usa Python. Alcuni di questi strumenti eseguono direttamente il compilatore, invece di lasciare che il processo di build del sistema esegua quell'attività. Molti di questi strumenti richiedono l'installazione di altri come Python o Java prima che funzionino.

1.1.3 Perché non degli Script Autoprodotti?

Alcuni progetti utilizzano linguaggi di scripting esistenti come Perl o Python per configurare i processi di build. Sebbene sia possibile ottenere funzionalità simili con sistemi come questo, l'uso eccessivo di tali strumenti può rendere il processo di build più simile a una «caccia al tesoro» che a un sistema di build semplice da usare. Durante la creazione del pacchetto software, gli utenti sono costretti a trovare e installare la versione 4.3.2 di questo e 3.2.4 di quello prima ancora di poter avviare il processo di build. Per evitare questo problema, è stato deciso che CMake non avrebbe richiesto più strumenti di quelli richiesti dal software che veniva utilizzato per il build. Come minimo, l'utilizzo di CMake richiede un compilatore C, gli strumenti nativi del compilatore e un eseguibile CMake. CMake è stato scritto in C++, richiede solo un compilatore C++ per la compilazione e i binari precompilati sono disponibili per la maggior parte dei sistemi. Generando script autonomamente in genere significa anche che non si produrranno aree di lavoro Xcode o Visual Studio native, limitando le build di Mac e Windows.

1.1.4 Su quali piattaforme funziona CMake?

CMake funziona su un'ampia varietà di piattaforme tra cui Microsoft Windows, Apple Mac OS X e la maggior parte delle piattaforme UNIX o simili. Allo stesso modo, CMake supporta i compilatori più comuni.

1.1.5 Quanto è stabile CMake?

Prima di adottare qualsiasi nuova tecnologia o strumento per un progetto, uno sviluppatore vorrà sapere quanto è ben supportato e popolare. Dall'implementazione iniziale, CMake è cresciuto in popolarità come strumento di build. Sia la community di sviluppatori che quella di utenti continuano a crescere. CMake ha continuato a sviluppare il supporto per nuove tecnologie e strumenti di build non appena si rendono disponibili. Il team di sviluppo di CMake è molto impegnato per la compatibilità con le versioni precedenti. Se CMake può creare il progetto una volta, dovrebbe essere in grado di farlo sempre. Inoltre, poiché CMake è un progetto open source, il codice sorgente è sempre disponibile per un progetto da modificare e correggere secondo necessità.

CAPITOLO 2

Per Iniziare

2.1 Ottenere e Installare CMake sul Computer

Prima di utilizzare CMake, si dovranno installare o creare i binari di CMake. Su molti sistemi, CMake è già installato o è disponibile per l'installazione col tool di gestione pacchetti standard. Cygwin, Debian, FreeBSD, OS X MacPorts, Mac OS X Fink e molti altri hanno tutti distribuzioni di CMake.

Se il sistema non ha un pacchetto CMake, lo si può trovare precompilato per molte architetture nella pagina di Download. Selezionare la versione desiderata e seguire le istruzioni per il download. CMake può essere installato in qualsiasi directory, quindi i privilegi di root non sono richiesti.

Se non si trovano i binari precompilati per il proprio sistema, si può creare CMake dal sorgente. Per creare CMake, c'è bisogno di un compilatore C++ moderno e dei sorgenti dalla pagina di Download di CMake o dalla istanza GitLab di Kitware. Per creare CMake, seguire le istruzioni nel `README.rst` nella root dell'albero dei sorgenti.

2.2 Struttura della Directory

Esistono due directory principali utilizzate da CMake durante la creazione di un progetto: la directory dei sorgenti e la directory dei binari. La directory dei sorgenti è dove si trova il codice sorgente per il progetto. Questo è anche il punto in cui verranno si troveranno i file CMakeLists. La directory dei binari è talvolta indicata come directory di build ed è dove CMake inserirà i file oggetto, le librerie e gli eseguibili risultanti. CMake non scriverà alcun file nella directory dei sorgenti, solo in quella dei binari.

Le build «out-of-source», in cui le directory dei sorgenti e dei binarie sono diverse, sono fortemente volute. Le build «in-source» in cui le directory dei sorgenti e dei binari sono le stesse

sono supportate, ma dovrebbero essere evitate se possibile. Le build «out-of-source» facilitano molto la manutenzione di un albero dei sorgenti pulito e consentono una rapida rimozione di tutti i file generati da una build. Il fatto che l'albero di compilazione sia diverso da quello dei sorgenti semplifica anche il supporto di più build di un singolo albero di sorgenti. Questo è utile quando si vogliono avere più build con diverse opzioni ma solo una copia del codice sorgente.

2.3 Utilizzo di Base di CMake

CMake accetta uno o più file CMakeLists come input e produce file di progetto o Makefile da utilizzare con un'ampia varietà di strumenti di sviluppo nativi.

Il processo tipico di CMake è il seguente:

1. Il progetto è definito in uno o più file CMakeLists
2. CMake configura e genera il progetto
3. Gli utenti creano progetti con il loro strumento di sviluppo nativo preferito

Ogni fase del processo è descritta in dettaglio nelle sezioni seguenti.

2.4 I File CMakeLists

I file CMakeLists (in realtà `CMakeLists.txt` ma di solito si tralascia l'estensione) sono file di semplice testo che contengono la descrizione del progetto nel linguaggio di CMake. Il `cmake-language` è espresso come una serie di commenti, comandi e variabili. Si si potrebbe chiedere perché CMake ha deciso di avere il proprio linguaggio invece di usarne uno esistente come Python, Java o Tcl. Il motivo principale è che gli sviluppatori di CMake non volevano che CMake richiedesse uno strumento aggiuntivo per l'esecuzione. Richiedendo uno di questi altri linguaggi, tutti gli utenti di CMake dovrebbero avere tale linguaggio installato e potenzialmente una versione specifica. Questo è in cima alle estensioni del linguaggio che sarebbero necessarie per eseguire parte del lavoro di CMake, sia per motivi di prestazioni che di capacità.

2.4.1 Lo «Hello World» di CMake

Per iniziare, consideriamo il file CMakeLists più semplice possibile. Per compilare un eseguibile da un file sorgente, il file CMakeLists conterrebbe tre righe:

```
cmake_minimum_required(VERSION 3.20)
project(Hello)
add_executable(Hello Hello.c)
```

La prima riga del file CMakeLists di primo livello dovrebbe essere sempre `cmake_minimum_required`. Ciò consente ai progetti di richiedere una determinata versione di CMake e, inoltre, consente a CMake di essere compatibile con le versioni precedenti.

La riga successiva di qualsiasi file CMakeLists di primo livello dovrebbe essere il comando `project`. Questo comando imposta il nome del progetto e può specificare altre opzioni come il linguaggio o la versione.

Per ogni directory in un progetto in cui il file CMakeLists.txt richiama il comando `project`, CMake genera un Makefile di primo livello o un file di progetto IDE. Il progetto conterrà tutti i target che si trovano nel file CMakeLists.txt e tutte le sottodirectory, come specificato dal comando `add_subdirectory`. Se l'opzione `EXCLUDE_FROM_ALL` è usata nel comando `add_subdirectory`, il progetto generato non apparirà nel Makefile di primo livello o nel file di progetto IDE; questo è utile per generare sottoprogetti che non hanno senso come parte del processo di build principale. Si consideri che un progetto con un numero di esempi potrebbe utilizzare questa funzione per generare i file di build per ogni esempio con un'esecuzione di CMake, ma non avere gli esempi compilati come parte del normale processo di compilazione.

Infine, c'è il comando `add_executable` per aggiungere un eseguibile al progetto utilizzando il file sorgente specificato.

In questo esempio, ci sono due file nella directory dei sorgenti: `CMakeLists.txt` e `Hello.c`.

Le sezioni successive descriveranno come configurare e creare il progetto utilizzando la GUI di CMake e le interfacce a riga di comando.

2.5 Configurare e Generare

Dopo che un file CMakeLists è stato creato, CMake elabora il file di testo e crea le voci in un file di cache. Gli utenti possono modificare il file CMakeLists o specificare i valori della cache con l'interfaccia grafica di CMake o ccmake e riconfigurare. Successivamente, CMake utilizza le voci della cache per generare un progetto nel sistema di build desiderato dall'utente (ad es. Makefile o soluzione Visual Studio).

2.5.1 Esecuzione della GUI di CMake

CMake include un'interfaccia utente basata su Qt utilizzabile sulla maggior parte delle piattaforme, inclusi UNIX, Mac OS X e Windows. La `cmake-gui` è inclusa nel codice sorgente di CMake, ma c'è bisogno di un'installazione di Qt sul sistema per compilarla.

Su Windows, l'eseguibile si chiama `cmake-gui.exe` e dovrebbe trovarsi nel menù Start sotto Programmi. Potrebbe anche esserci un collegamento sul desktop o, se CMake è stato creato dai sorgenti, sarà nella directory di build. Per gli utenti UNIX e Mac, l'eseguibile si chiama `cmake-gui` e si trova dove sono installati gli eseguibili di CMake. Apparirà una GUI simile a quella mostrata in Figura 1. I primi due campi sono il codice sorgente e le directory dei file binari. Consentono di specificare dove si trova il codice sorgente per ciò che si desidera compilare e dove devono essere posizionati i binari risultanti. Si devono impostare per primi questi valori. Se la directory binaria specificata non esiste, verrà creata. Se la directory dei binari è stata configurata in precedenza da CMake, imposterà automaticamente l'albero dei sorgenti.

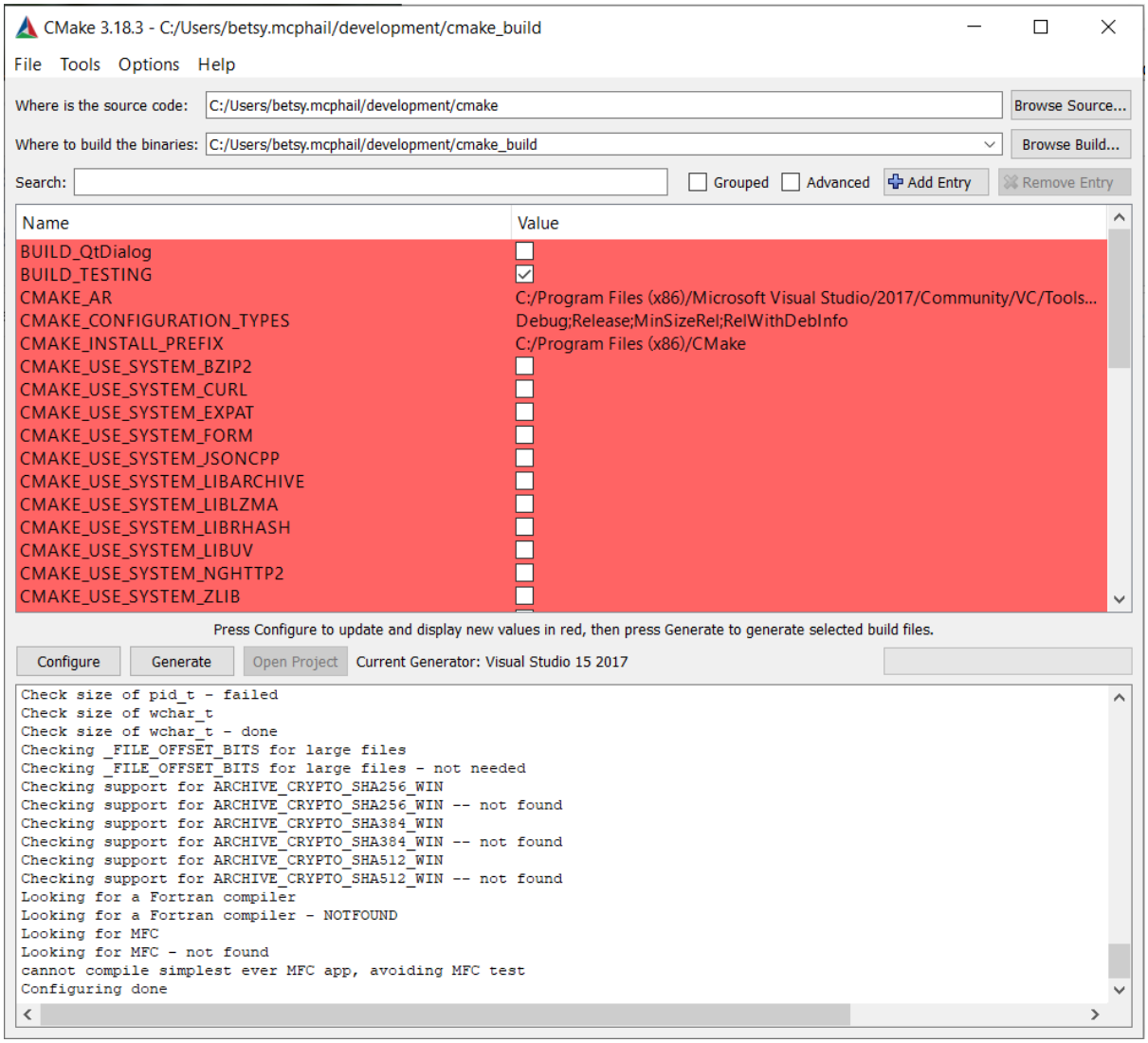


Fig. 1: La GUI CMake basata su Qt

2.5.2 Esecuzione di ccmake con l'Interfaccia di Curses

Sulla maggior parte delle piattaforme UNIX, se la libreria curses è supportata, CMake fornisce un eseguibile chiamato `ccmake`. Questa interfaccia è un'applicazione testuale basata su terminale, molto simile a `cmake-gui`. Per eseguire `ccmake`, cambiare le directory in quella in cui si desidera posizionare i file binari. Poi eseguire `ccmake` con il path della directory dei sorgenti sulla riga di comando. Questo avvierà l'interfaccia testuale come mostrato in Figura 2.

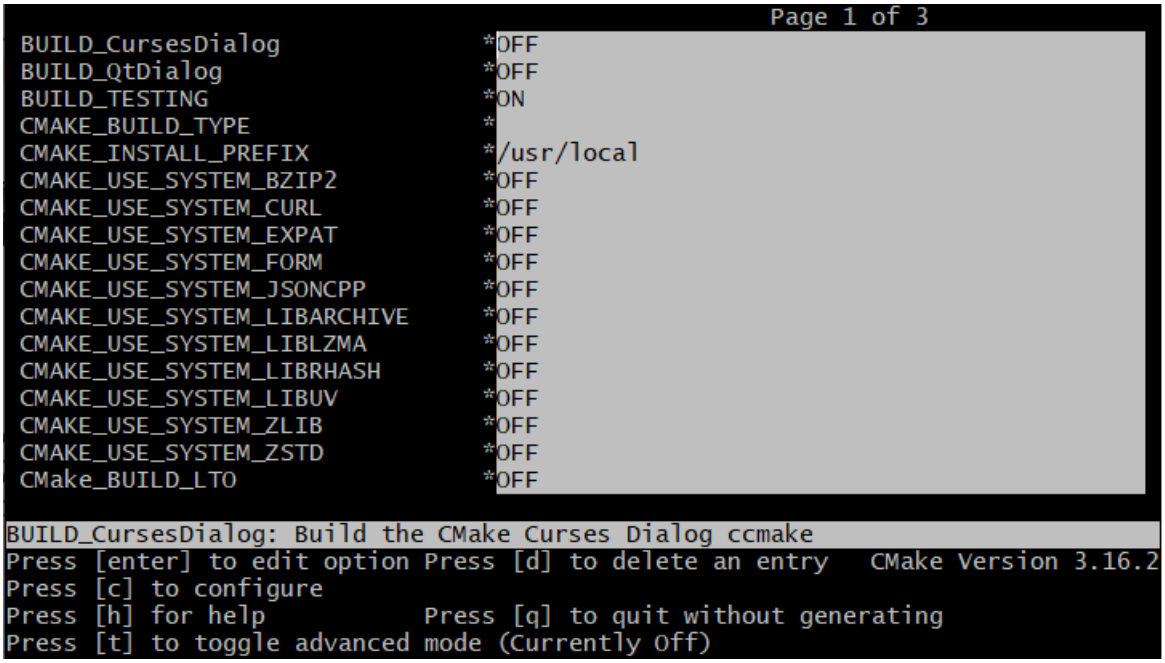


Fig. 2: ccmake in esecuzione su UNIX

Delle brevi istruzioni vengono visualizzate nella parte inferiore della finestra. Premendo il tasto «c», si configurerà il progetto. Si deve sempre configurare dopo aver modificato i valori nella cache. Per modificare i valori, utilizzare i tasti freccia per selezionare le voci della cache e premere il tasto Invio per modificarle. I valori booleani verranno scambiati con il tasto Invio. Una volta impostati tutti i valori, si preme il tasto «g» per generare i Makefile ed uscire. Si può anche premere «h» per l'help, «q» per uscire e «t» per attivare o disattivare la visualizzazione delle voci avanzate della cache.

2.5.3 Esecuzione di CMake da Riga di Comando

Dalla riga di comando, l'eseguibile `cmake` è utilizzabile per generare un sistema di compilazione del progetto. Questo è adatto per progetti con poche o nessuna opzione. Per progetti più grandi come VTK, si consiglia di utilizzare `ccmake` o `cmake-gui`. Per creare un progetto con `cmake`, prima si crea e si modifica la directory in cui si desidera posizionare i file binari. Si esegue `cmake` specificando il path dell'alberatura dei sorgenti e si passa qualsiasi opzione usando il flag `-D`. A differenza di `ccmake`, o di `cmake-gui`, i passaggi di configurazione e generazione sono combinati in uno solo quando si usa l'eseguibile `cmake`.

2.5.4 Specificare il compilatore per CMake

Su alcuni sistemi, si potrebbero avere più di un compilatore tra cui scegliere o il compilatore potrebbe trovarsi in una posizione non standard. In questi casi, si dovrà specificare a CMake dove si trova il compilatore desiderato. Ci sono tre modi per farlo: il generatore può specificare il compilatore; è possibile impostare una variabile d'ambiente; oppure è possibile impostare una voce della cache. Alcuni generatori sono legati a un compilatore specifico; ad esempio, il generatore di Visual Studio 19 utilizza sempre il compilatore Microsoft Visual Studio 19. Per i generatori basati su Makefile, CMake proverà un elenco di compilatori soliti fino a quando non ne troverà uno funzionante.

Gli elenchi possono essere superati con variabili di ambiente impostabili prima dell'esecuzione di CMake. La variabile d'ambiente `CC` specifica il compilatore C, mentre `CXX` specifica quello C++. Si possono specificare i compilatori direttamente sulla riga di comando usando `-DCMAKE_CXX_COMPILER=cl` per esempio. Una volta che `cmake` è stato eseguito e si è scelto un compilatore, se si desidera cambiare il compilatore, si ricomincia daccapo con una directory binaria vuota.

I flag per il compilatore e il linker possono anche essere modificati impostando le variabili di ambiente. L'impostazione `LDFLAGS` inizializzerà i valori della cache per i flag dei link, mentre `CXXFLAGS` e `CFLAGS` inizializzeranno `CMAKE_CXX_FLAGS` e `CMAKE_C_FLAGS` rispettivamente.

2.5.5 Configurazioni della Build

Le configurazioni della build consentono di creare un progetto in modi diversi per il debug, per l'ottimizzazione o per qualsiasi altro set speciale di flag. CMake supporta, per default, le configurazioni Debug, Release, MinSizeRel e RelWithDebInfo. Debug ha i flag di debug di base attivati. Release ha le ottimizzazioni di base attivate. MinSizeRel ha flag che producono il codice oggetto più piccolo, ma non necessariamente quello più veloce. RelWithDebInfo crea una build ottimizzata anche con informazioni di debug.

CMake gestisce le configurazioni in modi leggermente diversi a seconda del generatore utilizzato. Quando possibile, vengono seguite le convenzioni del sistema di build nativo. Ciò significa che le configurazioni influiscono sulla build in modi diversi quando si usano i Makefile rispetto ai file di progetto di Visual Studio.

L'IDE di Visual Studio supporta la nozione di Build Configuration. Un progetto di default in Visual Studio ha in genere configurazioni di Debug e di Release. Dall'IDE è possibile selezionare build Debug e i file verranno compilati con i flag Debug. L'IDE inserisce tutti i file binari nelle directory con il nome della configurazione attiva. Ciò comporta un'ulteriore complessità per i progetti che compilano programmi che devono essere eseguiti come parte del processo di build da comandi personalizzati. Consultare la variabile `CMAKE_CFG_INTDIR` e la sezione dei comandi personalizzati per ulteriori informazioni su come gestire questo problema. La variabile `CMAKE_CONFIGURATION_TYPES` viene utilizzata per indicare a CMake quali configurazioni inserire nell'area di lavoro [workspace].

Con i generatori basati su Makefile, solo una configurazione può essere attiva al momento dell'esecuzione di CMake ed è specificata con la variabile `CMAKE_BUILD_TYPE`. Se la variabile è vuota, non vengono aggiunti flag alla build. Se la variabile è impostata sul nome di una configurazione, le variabili e le regole appropriate (come `CMAKE_CXX_FLAGS_<ConfigName>`)

vengono aggiunte alle righe di compilazione. I makefile non usano sottodirectory di configurazione speciali per i file oggetto. Per creare alberature di debug e release, l'utente deve creare più directory di build utilizzando la funzionalità di build out-of-source di CMake e impostare CMAKE_BUILD_TYPE sulla selezione desiderata per ogni build. Per esempio:

```
# With source code in the directory MyProject
# to build MyProject-debug create that directory, cd into it and
ccmake ../MyProject -DCMAKE_BUILD_TYPE=Debug
# the same idea is used for the release tree MyProject-release
ccmake ../MyProject -DCMAKE_BUILD_TYPE=Release
```

2.6 Building del Progetto

Dopo aver eseguito CMake, il progetto sarà pronto per il building. Se il generatore target è basato su Makefile, si può creare il progetto cambiando la directory nell'albero binario e digitando make (o gmake o nmake a seconda dei casi). Se è stato generato un file per un IDE come Visual Studio, si avviare l'IDE, caricarvi i file di progetto e compilare come si farebbe normalmente.

Un'altra possibilità è quella di usare l'opzione di `cmake --build` dalla riga di comando. Questa opzione è semplicemente una comodità che consente di creare il progetto dalla riga di comando, anche se ciò richiede l'avvio di un IDE.

Questo è tutto ciò che serve per installare ed eseguire CMake per progetti semplici. Nei capitoli seguenti considereremo CMake in modo più dettagliato e spiegheremo come utilizzarlo su progetti software più complessi.

CAPITOLO 3

Scrivere File CMakeLists

Questo capitolo tratterà le basi della scrittura di file CMakeLists efficaci. Tratterà i comandi e i problemi di base necessari per gestire la maggior parte dei progetti. Sebbene CMake sia in grado di gestire progetti estremamente complessi, per la maggior parte dei progetti si scoprirà che i contenuti di questo capitolo diranno tutto ciò che c'è da sapere. CMake è guidato dai file CMakeLists.txt scritti per un progetto software. I file CMakeLists determinano tutto, da quali opzioni presentare agli utenti, a quali file sorgente compilare. Oltre a discutere come scrivere un file CMakeLists, questo capitolo tratterà anche come renderli robusti e manutenibili.

3.1 Modifica dei file CMakeLists

I file CMakeLists possono essere modificati in quasi tutti gli editor di testo. Alcuni editor, come Notepad++, vengono forniti con l'evidenziazione della sintassi CMake e il supporto dell'indentazione. Per editor come Emacs o Vim, CMake include modalità di indentazione e di evidenziazione della sintassi. Questi si trovano nella directory `Auxiliary` della distribuzione sorgente o si possono scaricare dalla pagina di Download di CMake.

All'interno di uno qualsiasi dei generatori supportati (Makefile, Visual Studio, ecc.), se si modifica un file CMakeLists e lo re-builda, ci sono regole che richiameranno automaticamente CMake per aggiornare i file generati (ad es. Makefile o file di progetto) come richiesto. Questo contribuisce a garantire che i file generati siano sempre sincronizzati con i file CMakeLists.

3.2 Il Linguaggio CMake

Il linguaggio CMake è composto da commenti, comandi e variabili.

3.3 Commenti

I commenti iniziano con # e vanno fino alla fine della riga. Vedere il manuale `cmake-language` per maggiori dettagli.

3.4 Variabili

I file CMakeLists utilizzano variabili in modo molto simile a qualsiasi altro linguaggio di programmazione. I nomi delle variabili CMake fanno distinzione tra maiuscole e minuscole e possono contenere solo caratteri alfanumerici e caratteri di sottolineatura.

Un certo numero di variabili utili sono definite automaticamente da CMake e sono discusse nel manuale `cmake-variables`. Tali variabili iniziano con `CMAKE_`. Evitare questa convenzione di denominazione (e, idealmente, stabilirne una propria) per le variabili specifiche del progetto.

Tutte le variabili CMake sono memorizzate internamente come stringhe sebbene a volte possano essere interpretate come altri tipi.

Usare il comando `set` per impostare i valori delle variabili. Nella sua forma più semplice, il primo argomento di `set` è il nome della variabile e il resto degli argomenti sono i valori. Più argomenti di valore vengono inseriti in un elenco separato da punti e virgola e memorizzati nella variabile come stringa. Per esempio:

```
set(Foo "")      # 1 quoted arg -> value is ""
set(Foo a)       # 1 unquoted arg -> value is "a"
set(Foo "a b c") # 1 quoted arg -> value is "a b c"
set(Foo a b c)   # 3 unquoted args -> value is "a;b;c"
```

È possibile fare riferimento alle variabili negli argomenti del comando utilizzando la sintassi `${VAR}` dove `VAR` è il nome della variabile. Se la variabile richiamata non è definita, il riferimento viene sostituito con una stringa vuota; in caso contrario viene sostituito dal valore della variabile. La sostituzione viene eseguita prima dell'espansione degli argomenti non quotati, quindi i valori delle variabili contenenti punti e virgola vengono suddivisi in zero o più argomenti al posto dell'argomento originale non quotato. Per esempio:

```
set(Foo a b c)    # 3 unquoted args -> value is "a;b;c"
command(${Foo})   # unquoted arg replaced by a;b;c
                  # and expands to three arguments
command("${Foo}") # quoted arg value is "a;b;c"
set(Foo "")       # 1 quoted arg -> value is empty string
command(${Foo})   # unquoted arg replaced by empty string
```

(continues on next page)

(continua dalla pagina precedente)

```
                    # and expands to zero arguments
command("${Foo}") # quoted arg value is empty string
```

È possibile accedere direttamente alle variabili di ambiente di sistema e ai valori del registro di Windows in CMake. Per accedere alle variabili di ambiente di sistema, utilizzare la sintassi `$ENV{VAR}`. CMake può anche fare riferimento alle voci di registro in molti comandi utilizzando una sintassi della forma `[HKEY_CURRENT_USER\\Software\\path1\\path2;key]`, dove i percorsi sono costruiti dalla struttura e dalla chiave del registro.

3.4.1 Scope delle Variabili

Le variabili in CMake hanno uno scope leggermente diverso da quello della maggior parte dei linguaggi. Quando si imposta una variabile, essa è visibile al file o alla funzione CMakeLists corrente e ai file CMakeLists di qualsiasi sottodirectory, a qualsiasi funzione o macro richiamata e a qualsiasi file incluso utilizzando il comando `include`. Quando viene elaborata una nuova sottodirectory (o viene chiamata una funzione), viene creato e inizializzato un nuovo scope di variabile con il valore corrente di tutte le variabili nello scope chiamante. Eventuali nuove variabili create nello scope figlio o le modifiche apportate alle variabili esistenti non influiranno sullo scope padre. Si consideri l'esempio seguente:

```
function(foo)
  message(${test}) # test is 1 here
  set(test 2)
  message(${test}) # test is 2 here, but only in this scope
endfunction()

set(test 1)
foo()
message(${test}) # test will still be 1 here
```

A volte, si vorrebbe che una funzione o una sottodirectory imposti una variabile nello scope del suo genitore. Esiste un modo per CMake di restituire un valore da una funzione e può essere fatto utilizzando l'opzione `PARENT_SCOPE` col comando `set`. Possiamo modificare l'esempio precedente in modo che la funzione `foo` cambi il valore di test nello scope del suo genitore in questo modo:

```
function(foo)
  message(${test}) # test is 1 here
  set(test 2 PARENT_SCOPE)
  message(${test}) # test still 1 in this scope
endfunction()

set(test 1)
foo()
message(${test}) # test will now be 2 here
```

Le variabili in CMake sono definite nell'ordine di esecuzione dei comandi `set`.

Si consideri l'esempio seguente:

```
# FOO is undefined

set(FOO 1)
# FOO is now set to 1

set(FOO 0)
# FOO is now set to 0
```

Per comprendere lo scope delle variabili, si consideri questo esempio:

```
set(foo 1)

# process the dir1 subdirectory
add_subdirectory(dir1)

# include and process the commands in file1.cmake
include(file1.cmake)

set(bar 2)
# process the dir2 subdirectory
add_subdirectory(dir2)

# include and process the commands in file2.cmake
include(file2.cmake)
```

In questo esempio, poiché la variabile `foo` è definita all'inizio, verrà definita durante l'elaborazione sia di dir1 che di dir2. Al contrario, `bar` sarà definito solo durante l'elaborazione di dir2. Allo stesso modo, `foo` sarà definito durante l'elaborazione sia di file1.cmake che di file2.cmake, mentre `bar` sarà definito solo durante l'elaborazione di file2.cmake.

3.5 Comandi

Un comando consiste nel nome del comando, parentesi di apertura, argomenti separati da spazi e parentesi di chiusura. Ogni comando viene valutato nell'ordine in cui appare nel file CMakeLists. Consultare il manuale `cmake-commands` per l'elenco completo dei comandi CMake.

CMake non fa più distinzione tra maiuscole e minuscole per i nomi dei comandi, quindi dove si vede `command`, si può usare `COMMAND` o `Command`. È considerata «best practice» utilizzare i comandi in minuscolo. Tutti i «whitespace» (spazi, line feed, tabulazioni) vengono ignorati tranne che per separare gli argomenti. Pertanto, i comandi possono estendersi su più righe purché il nome del comando e la parentesi di apertura si trovino sulla stessa riga.

Gli argomenti dei comandi CMake sono separati da spazi e fanno distinzione tra maiuscole e minuscole. Gli argomenti dei comandi possono essere quotati o non quotati. Un argomento

quotato (tra virgolette) inizia e finisce con un doppio apice (») e rappresenta sempre esattamente un argomento. Eventuali virgolette doppie contenute all'interno del valore devono essere precedute da una barra rovesciata [backslash]. Si consideri l'utilizzo di argomenti tra parentesi per argomenti che richiedono l'escape, consultare il manuale `cmake-language`. Un argomento non quotato inizia con qualsiasi carattere diverso da un doppio apice (successivamente le doppie virgolette sono letterali) e viene automaticamente espanso in zero o più argomenti separando col punto e virgola all'interno del valore. Per esempio:

```
command("")           # 1 quoted argument
command("a b c")      # 1 quoted argument
command("a;b;c")      # 1 quoted argument
command("a" "b" "c") # 3 quoted arguments
command(a b c)        # 3 unquoted arguments
command(a;b;c)        # 1 unquoted argument expands to 3
```

3.5.1 Comandi di Base

Come visto in precedenza, i comandi `set` e `unset` impostano o annullano esplicitamente le variabili. I comandi `string`, `list` e `separate_arguments` offrono comandi di base per manipolare stringhe e liste.

I comandi `add_executable` e `add_library` sono i comandi principali per definire gli eseguibili e le librerie da compilare e quali file sorgenti li comprendono. Per i progetti di Visual Studio, i file sorgenti verranno visualizzati nell'IDE come di consueto, ma i file header utilizzati dal progetto non lo saranno. Per visualizzare i file header, si aggiungerli all'elenco dei file sorgenti per l'eseguibile o la libreria; questo può essere fatto per tutti i generatori. Tutti i generatori che non usano direttamente i file header (come quelli basati su Makefile) semplicemente li ignoreranno.

3.6 Controllo del Flusso

Il linguaggio CMake fornisce tre costrutti per il controllo del flusso per aiutare a organizzare i file CMakeLists e mantenerli gestibili.

- Le istruzioni condizionali (ad es. `if`)
- Costrutti per i cicli (ad es. `foreach` e `while`)
- Definizioni di procedure (ad es. `macro` e `function`)

3.6.1 Istruzioni Condizionali

Per prima cosa considereremo il comando `if`. In molti modi, il comando `if` in CMake è proprio come il comando `if` in qualsiasi altro linguaggio. Valuta la sua espressione e la usa per eseguire il codice nel suo corpo o facoltativamente il codice nella clausola `else`. Per esempio:

```
if(FOO)
  # do something here
else()
  # do something else
endif()
```

CMake supporta anche `elseif` per aiutare a testare in sequenza più condizioni. Per esempio:

```
if(MSVC80)
  # do something here
elseif(MSVC90)
  # do something else
elseif(APPLE)
  # do something else
endif()
```

Il comando `if` documenta le molte condizioni che può testare.

3.6.2 Costrutti per i Cicli

I comandi `foreach` e `while` consentono di gestire attività ripetitive che si verificano in sequenza. Il comando `break` interrompe un ciclo `foreach` o `while` prima che termini normalmente.

Il comando `foreach` consente di eseguire ripetutamente un gruppo di comandi CMake sui membri di un elenco. Si consideri il seguente esempio adattato da VTK

```
foreach(tfile
        TestAnisotropicDiffusion2D
        TestButterworthLowPass
        TestButterworthHighPass
        TestCityBlockDistance
        TestConvolve
        )
  add_test(${tfile}-image ${VTK_EXECUTABLE}
    ${VTK_SOURCE_DIR}/Tests/rtImageTest.tcl
    ${VTK_SOURCE_DIR}/Tests/${tfile}.tcl
    -D ${VTK_DATA_ROOT}
    -V Baseline/Imaging/${tfile}.png
    -A ${VTK_SOURCE_DIR}/Wrapping/Tcl
    )
endforeach()
```

Il primo argomento del comando `foreach` è il nome della variabile che assumerà un valore diverso ad ogni iterazione del ciclo; gli argomenti rimanenti sono l'elenco di valori su cui eseguire il ciclo. In questo esempio, il corpo del ciclo `foreach` è solo un comando CMake, `add_test`. Nel corpo di `foreach`, ogni volta che si fa riferimento alla variabile del ciclo (`tfile` in questo esempio) verrà sostituita con il valore corrente dall'elenco. Nella prima iterazione, le occorrenze di `${tfile}` verranno sostituite con `TestAnisotropicDiffusion2D`. Nella successiva iterazione, `${tfile}` verrà sostituito con `TestButterworthLowPass`. Il ciclo `foreach` continuerà fino a quando tutti gli argomenti non saranno stati elaborati.

Vale la pena ricordare che i cicli `foreach` possono essere nidificati e che la variabile del ciclo viene sostituita prima di qualsiasi altra espansione della variabile. Ciò significa che nel corpo di un ciclo `foreach`, si possono costruire nomi di variabili usando la variabile del ciclo. Nel codice seguente, la variabile di ciclo `tfile` viene espansa e quindi concatenata con `_TEST_RESULT`. Il nuovo nome della variabile viene quindi espanso e testato per vedere se corrisponde a `FAILED`.

```
if(${${tfile}_TEST_RESULT} MATCHES FAILED)
  message("Test ${tfile} failed.")
endif()
```

Il comando `while` fornisce il ciclo basato su una condizione di test. Il formato per l'espressione di test nel comando `while` è lo stesso del comando `if`, come descritto. Si consideri l'esempio seguente, utilizzato da CTest. Da notare che CTest aggiorna internamente il valore di `CTEST_ELAPSED_TIME`.

```
#####################################################
# run paraview and ctest test dashboards for 6 hours
#
while(${CTEST_ELAPSED_TIME} LESS 36000)
  set(START_TIME ${CTEST_ELAPSED_TIME})
  ctest_run_script("dash1_ParaView_vs71continuous.cmake")
  ctest_run_script("dash1_cmake_vs71continuous.cmake")
endwhile()
```

3.6.3 Definizioni delle Procedure

I comandi `macro` e `function` supportano attività ripetitive che possono essere sparse nei file CMakeLists. Una volta definita una macro o una funzione, questa può essere utilizzata da qualsiasi file CMakeLists elaborato dopo la sua definizione.

Una funzione in CMake è molto simile a una funzione in C o C++. Le si possono passare degli argomenti e questi diventano variabili all'interno della funzione. Allo stesso modo, sono definite alcune variabili standard come `ARGC`, `ARGV`, `ARGN` e `ARGV0`, `ARGV1`, ecc. Le chiamate a funzioni hanno uno scope dinamico. All'interno di una funzione ci si trova in un nuovo scope variabile; è come entrare in una sottodirectory usando il comando `add_subdirectory` e trovarsi in un nuovo scope variabile. Tutte le variabili che sono state definite quando la funzione è stata chiamata rimangono definite, ma eventuali modifiche alle variabili o le nuove variabili esistono solo all'interno della funzione. Quando la funzione ritorna, quelle variabili andranno

via. In parole povere: quando si invoca una funzione, viene inserito un nuovo scope variabile; quando ritorna, quello scope variabile viene ripreso.

Il comando `function` definisce una nuova funzione. Il primo argomento è il nome della funzione da definire; tutti gli argomenti aggiuntivi sono parametri formali della funzione.

```
function(DetermineTime _time)
  # pass the result up to whatever invoked this
  set(${_time} "1:23:45" PARENT_SCOPE)
endfunction()

# now use the function we just defined
DetermineTime(current_time)

if(DEFINED current_time)
  message(STATUS "The time is now: ${current_time}")
endif()
```

Notare che in questo esempio, `_time` viene utilizzato per passare il nome della variabile restituita. Il comando `set` viene invocato con il valore di `_time`, che sarà `current_time`. Infine, il comando `set` utilizza l'opzione `PARENT_SCOPE` per impostare la variabile nello scope del chiamante anziché in quello locale.

Le macro vengono definite e chiamate allo stesso modo delle funzioni. Le differenze principali sono che una macro non esegue il push e il pop di un nuovo scope variabile e che gli argomenti di una macro non vengono trattati come variabili ma come stringhe sostituite prima dell'esecuzione. Questo è molto simile alle differenze tra una macro e una funzione in C o C++. Il primo argomento è il nome della macro da creare; tutti gli argomenti aggiuntivi sono parametri formali della macro.

```
# define a simple macro
macro(assert TEST COMMENT)
  if(NOT ${TEST})
    message("Assertion failed: ${COMMENT}")
  endif()
endmacro()

# use the macro
find_library(FOO_LIB foo /usr/local/lib)
assert(${FOO_LIB} "Unable to find library foo")
```

Il semplice esempio precedente crea una macro chiamata `assert`. La macro è definita in due argomenti; il primo è un valore da testare e il secondo è un commento da stampare se il test fallisce. Il corpo della macro è un semplice comando `if` con un comando `message` al suo interno. Il corpo della macro termina quando viene trovato il comando `endmacro`. La macro può essere invocata semplicemente usando il suo nome come se fosse un comando. Nell'esempio precedente, se `FOO_LIB` non è stato trovato, verrà visualizzato un messaggio che indica la condizione di errore.

Il comando `macro` supporta anche la definizione di macro che accettano elenchi variabili di argomenti. Ciò può essere utile se si desidera definire una macro con argomenti facoltativi o firme multiple. Gli argomenti variabili possono essere referenziati usando `ARGC` e `ARGV0`, `ARGV1`, ecc., invece dei parametri formali. `ARGV0` rappresenta il primo argomento della macro; `ARGV1` rappresenta il successivo e così via. È inoltre possibile utilizzare una combinazione di argomenti formali e argomenti variabili, come mostrato nell'esempio seguente.

```
# define a macro that takes at least two arguments
# (the formal arguments) plus an optional third argument
macro(assert TEST COMMENT)
  if(NOT ${TEST})
    message("Assertion failed: ${COMMENT}")

    # if called with three arguments then also write the
    # message to a file specified as the third argument
    if(${ARGC} MATCHES 3)
      file(APPEND ${ARGV2} "Assertion failed: ${COMMENT}")
    endif()

  endif()
endmacro()

# use the macro
find_library(FOO_LIB foo /usr/local/lib)
assert(${FOO_LIB} "Unable to find library foo")
```

In questo esempio, i due argomenti obbligatori sono `TEST` e `COMMENT`. È possibile fare riferimento a questi argomenti obbligatori per nome, come in questo esempio, o facendo riferimento a `ARGV0` e `ARGV1`. Per elaborare gli argomenti come una lista, usare le variabili `ARGV` e `ARGN`. `ARGV` (al contrario di `ARGV0`, `ARGV1`, ecc.) è un elenco di tutti gli argomenti della macro, mentre `ARGN` è un elenco di tutti gli argomenti dopo gli argomenti formali. All'interno della macro, si può utilizzare il comando `foreach` per iterare su `ARGV` o `ARGN` come desiderato.

Il comando `return` ritorna da una funzione, directory o file. Notare che una macro, a differenza di una funzione, viene espansa sul posto e quindi non può gestire `return`.

3.7 Espressioni Regolari

Alcuni comandi CMake, come `if` e `string`, fanno uso di espressioni regolari o possono prendere un'espressione regolare come argomento. Nella sua forma più semplice, un'espressione regolare è una sequenza di caratteri utilizzata per cercare corrispondenze esatte di caratteri. Tuttavia, molte volte la sequenza esatta da trovare è sconosciuta o si desidera solo una corrispondenza all'inizio o alla fine di una stringa. Poiché esistono diverse convenzioni per specificare le espressioni regolari, lo standard di CMake è descritto nella documentazione del comando `string`. La descrizione si basa sulla classe di espressioni regolari open source di Texas Instruments, utilizzata da CMake per l'analisi delle espressioni regolari.

3.8 Comandi Avanzati

Esistono alcuni comandi che possono essere molto utili, ma in genere non vengono utilizzati nella scrittura di file CMakeLists. Questa sezione tratterà alcuni di questi comandi e quando risultano utili.

Innanzitutto, si considera il comando `add_dependencies` che crea una dipendenza tra due target. CMake crea automaticamente dipendenze tra i target quando può determinarle. Ad esempio, CMake creerà automaticamente una dipendenza per un target eseguibile che dipende da un target libreria. Il comando `add_dependencies` viene in genere utilizzato per specificare le dipendenze inter-target in cui almeno uno è un target personalizzato (vedere la sezione *Aggiungere Comandi Custom*).

Anche il comando `include_regular_expression` riguarda le dipendenze. Questo comando controlla l'espressione regolare utilizzata per tracciare [trace] le dipendenze del codice sorgente. Per default, CMake traccerà [trace] tutte le dipendenze per un file sorgente inclusi i file di sistema come `stdio.h`. Se si specifica un'espressione regolare col comando `include_regular_expression`, tale espressione regolare verrà utilizzata per limitare i file header processati. Per esempio; se i file header del progetto software iniziano tutti col prefisso foo (ad es. `fooMain.c fooStruct.h`, ecc.), si può specificare un'espressione regolare di `^foo.*$` per limitare il controllo delle dipendenze ai soli file del progetto.

CAPITOLO 4

La Cache di CMake

La cache di CMake può essere considerata come un file di configurazione. La prima volta che CMake viene eseguito su un progetto, produce un file `CMakeCache.txt` nella directory superiore dell'albero di build. CMake usa questo file per archiviare un set di variabili globali della cache, i cui valori persistono su più esecuzioni all'interno di un albero di build del progetto.

Ci sono alcuni scopi di tale cache. Il primo è memorizzare le selezioni e le scelte dell'utente, in modo che se dovesse eseguire nuovamente CMake non dovrà reinserire tali informazioni. Ad esempio, il comando `option` crea una variabile booleana e la memorizza nella cache.

```
option(USE_JPEG "Do you want to use the jpeg library")
```

La riga precedente crea una variabile chiamata `USE_JPEG` e la inserisce nella cache. In questo modo l'utente può impostare tale variabile dall'interfaccia utente e il suo valore rimarrà nel caso in cui l'utente debba eseguire nuovamente CMake in futuro. Per creare una variabile nella cache, si usano comandi come `option`, `find_file`, o il comando standard `set` con l'opzione `CACHE`.

```
set(USE_JPEG ON CACHE BOOL "include jpeg support?")
```

Quando si usa l'opzione `CACHE`, si può anche fornire il tipo di variabile e una stringa di documentazione. Il tipo della variabile è usato da `cmake-gui` per controllare come la variabile è impostata e visualizzata, ma il valore è sempre memorizzato nel file di cache come una stringa.

Un altro scopo della cache è consentire a CMake stesso di archiviare in modo persistente i valori tra le sue esecuzioni. Queste voci potrebbero non essere visibili o modificabili dall'utente. In genere, questi valori sono variabili dipendenti dal sistema che richiedono a CMake di compilare ed eseguire un programma per determinarne il valore. Una volta che questi valori sono stati determinati, vengono archiviati nella cache per evitare di doverli ricalcolare ogni volta che viene eseguito CMake. CMake generalmente cerca di limitare queste variabili a proprietà che non dovrebbero mai cambiare (come l'ordine dei byte della macchina su cui ci si trova). Se

si modifichi in modo significativo il computer, modificando il sistema operativo o passando a un compilatore diverso, si dovrà eliminare il file cache (e probabilmente tutti i file oggetto, le librerie e gli eseguibili dell'albero binario).

Alcuni progetti sono molto complessi e l'impostazione di un valore nella cache potrebbe far apparire nuove opzioni la prossima volta che la cache viene creata. Ad esempio, VTK supporta l'uso di MPI per eseguire il calcolo distribuito. Ciò richiede che il processo di compilazione determini dove si trovano le librerie MPI e i file header e permetta all'utente di aggiustarne i valori. Ma MPI è disponibile solo se un'altra opzione `VTK_USE_PARALLEL` viene attivata prima in VTK. Quindi, per evitare confusione per le persone che non sanno cosa sia MPI, queste opzioni sono nascoste finché `VTK_USE_PARALLEL` non viene attivato. Quindi CMake mostra l'opzione `VTK_USE_PARALLEL` nell'area della cache, se l'utente la attiva e si riconfigura con CMake, appariranno nuove opzioni per MPI che si possono poi impostare. La regola è continuare il build della cache finché non cambia. Per la maggior parte dei progetti questo avverrà solo una volta. Per alcuni più complicati potrebbe essere due o più.

Si potrebbe essere tentati di modificare direttamente il file della cache o di «inizializzare» un progetto fornendogli un file `CMakeCache.txt` pre-popolato. Questo potrebbe non funzionare e potrebbe causare ulteriori problemi in futuro. Innanzitutto, la sintassi della cache CMake è soggetta a cambiamenti. In secondo luogo, i file di cache contengono path completi che li rendono inadatti allo spostamento tra alberi binari.

Una volta che una variabile è nella cache, il suo valore «cache» normalmente non può essere modificato da un file CMakeLists. Il ragionamento alla base di ciò è che una volta che CMake ha inserito la variabile nella cache con il suo valore iniziale, l'utente può poi modificare quel valore dalla GUI. Se la successiva chiamata di CMake sovrascrivesse la loro modifica al valore `set`, l'utente non sarebbe mai in grado di apportare una modifica che CMake non sovrascriverebbe. Un comando `set(FOO ON CACHE BOOL "doc")` tipicamente farà qualcosa solo quando la cache non contiene la variabile. Una volta che la variabile è nella cache, quel comando non avrà alcun effetto.

Nel raro caso in cui si voglia davvero modificare il valore di una variabile memorizzata nella cache, si usa l'opzione `FORCE` in combinazione con l'opzione `CACHE` nel comando `set`. L'opzione `FORCE` farà sì che il comando `set` sovrascriva e modifichi il valore della cache di una variabile.

Alcuni punti finali dovrebbero essere fatti riguardo alle variabili e alla loro interazione con la cache. Se una variabile è nella cache, può comunque essere sovrascritta in un file CMakeLists usando il comando `set` senza l'opzione `CACHE`. I valori della cache vengono controllati quando una variabile di riferimento non è definita nello scope corrente. Il comando `set` definirà una variabile per lo scope corrente senza modificare il valore nella cache.

```
# assume that FOO is set to ON in the cache

set(FOO OFF)
# sets foo to OFF for processing this CMakeLists file
# and subdirectories; the value in the cache stays ON
```

Le variabili che si trovano nella cache hanno anche una proprietà che indica se sono avanzate o meno. Per default, quando vengono eseguiti `ccmake` o `cmake-gui`, le voci della cache avanzata non vengono visualizzate. In questo modo l'utente può concentrarsi sulle voci della cache che dovrebbe considerare di modificare. Le voci avanzate della cache sono altre opzioni che l'utente

può modificare, ma in genere non lo farà. Non è insolito che un progetto software di grandi dimensioni abbia cinquanta o più opzioni e la proprietà «advanced» consente a un progetto software di suddividerle in opzioni chiave per la maggior parte degli utenti e in opzioni avanzate per utenti esperti. A seconda del progetto, potrebbero non esserci voci di cache non avanzate. Per rendere avanzata una voce della cache, viene utilizzato il comando `mark_as_advanced` con il nome della variabile (ovvero la voce della cache).

In alcuni casi, si potrebbe voler limitare una voce della cache a un insieme limitato di opzioni predefinite. Lo si può fare impostando la proprietà `STRINGS` sulla voce della cache. Il seguente codice CMakeLists lo illustra creando una proprietà denominata `CRYPTOBACKEND` come al solito e quindi impostando la proprietà `STRINGS` su di essa su un insieme di tre opzioni.

```
set(CRYPTOBACKEND "OpenSSL" CACHE STRING
    "Select a cryptography backend")
set_property(CACHE CRYPTOBACKEND PROPERTY STRINGS
             "OpenSSL" "LibTomCrypt" "LibDES")
```

Quando viene eseguito `cmake-gui` e l'utente seleziona la voce della cache `CRYPTOBACKEND`, verrà visualizzato un menù a tendina per selezionare l'opzione voluta.

4.1 Impostazione dei Valori Iniziali per CMake

A volte potrebbe essere necessario impostare le voci della cache senza eseguire una GUI. Questo è comune quando si impostano dashboard notturne o se si creeranno molti alberi di build con gli stessi valori della cache. In questi casi, la cache di CMake può essere inizializzata in due modi diversi. Il primo modo consiste nel passare i valori della cache sulla riga di comando di CMake utilizzando gli argomenti `-DCACHE_VAR:TYPE=VALUE`. Ad esempio, si consideri il seguente script della dashboard notturna per una macchina UNIX:

```
#!/bin/tcsh

cd ${HOME}

# wipe out the old binary tree and then create it again
rm -rf Foo-Linux
mkdir Foo-Linux
cd Foo-Linux

# run cmake to setup the cache
cmake -DBUILD_TESTING:BOOL=ON <etc...> ../Foo

# generate the dashboard
ctest -D Nightly
```

La stessa idea può essere utilizzata con un file batch su Windows.

Il secondo modo consiste nel creare un file da caricare `cmake` con la sua opzione `-C`. In questo caso, invece di configurare la cache con le opzioni `-D`, viene eseguita tramite un file che viene

analizzato da CMake. La sintassi per questo file è quella standard di CMakeLists, che in genere è una serie di comandi `set` come:

```
# Build the vtkHybrid kit.
set(VTK_USE_HYBRID ON CACHE BOOL "doc string")
```

In alcuni casi potrebbe esserci una cache esistente e si desidera forzare l'impostazione dei valori della cache in un certo modo. Ad esempio, si supponga di voler attivare Hybrid anche se l'utente abbia precedentemente eseguito CMake e lo abbia disattivato. Allora si può fare

```
# Build the vtkHybrid kit always.
set(VTK_USE_HYBRID ON CACHE BOOL "doc" FORCE)
```

Un'altra possibilità è che si desidera impostare e quindi nascondere le opzioni in modo che l'utente non sia tentato di modificarle in seguito. Questo può essere fatto usando il tipo `INTERNAL`. Le variabili di cache `INTERNAL` implicano `FORCE` e non vengono mai mostrate negli editor della cache.

```
# Build the vtkHybrid kit always and don't distract
# the user by showing the option.
set(VTK_USE_HYBRID ON CACHE INTERNAL "doc")
```

CAPITOLO 5

Concetti Chiave

Questo capitolo introduce i concetti chiave di CMake. Quando si inizia a lavorare con CMake, ci si imbatte in una varietà di concetti come target, generatori e comandi. La comprensione di questi concetti fornirà le conoscenze operative necessarie per creare file CMakeLists efficaci. Molti oggetti CMake come target, directory e file sorgenti hanno delle proprietà associate. Una proprietà è una coppia chiave-valore associata a un oggetto specifico. Il modo più generico per accedere alle proprietà è tramite i comandi `set_property` e `get_property`. Questi comandi consentono di impostare o ottenere una proprietà da qualsiasi oggetto in CMake che ne dispone. Consultare il manuale `cmake-properties` per un elenco delle proprietà supportate. Dalla riga di comando è possibile ottenere un elenco completo delle proprietà supportate in CMake eseguendo `cmake` con l'opzione `--help-property-list`.

5.1 Target

Probabilmente l'elemento più importante sono i target. I target rappresentano eseguibili, librerie e utilità create da CMake. Ogni comando `add_library`, `add_executable` e `add_custom_target` crea un target. Ad esempio, il seguente comando creerà un target, denominato «foo», che è una libreria statica, con `foo1.c` e `foo2.c` come file sorgente.

```
add_library(foo STATIC foo1.c foo2.c)
```

Il nome «foo» è ora disponibile per l'uso, come nome di libreria, ovunque nel progetto e CMake saprà come espandere il nome nella libreria quando necessario. Le librerie possono essere dichiarate come un tipo particolare come `STATIC`, `SHARED`, `MODULE`, o senza dichiarazione. `STATIC` indica che la libreria deve essere creata come libreria statica. Allo stesso modo, `SHARED` indica che deve essere creata come libreria condivisa. `MODULE` indica che la libreria deve essere creata in modo che possa essere caricata dinamicamente in un eseguibile. Le librerie dei moduli vengono implementate come librerie condivise su molte piattaforme, ma non tutte. Pertanto,

CMake non consente ad altri target di collegarsi ai moduli. Se nessuna di queste opzioni è specificata, significa che la libreria può essere creata come condivisa o statica. In tal caso, CMake utilizza l'impostazione della variabile BUILD_SHARED_LIBS per determinare se la libreria deve essere SHARED o STATIC. Se non è impostata, per default CMake crea librerie statiche.

Allo stesso modo, gli eseguibili hanno alcune opzioni. Per default, un eseguibile sarà un'applicazione console tradizionale con un punto di ingresso principale. È possibile specificare un'opzione WIN32 per richiedere un punto di ingresso WinMain su sistemi Windows, mantenendo «main» su sistemi non Windows.

Oltre a memorizzarne il tipo, i target tengono traccia anche delle proprietà generali. Tali proprietà possono essere impostate e recuperate usando i comandi set_target_properties e get_target_property, o i comandi più generali set_property e get_property. Una proprietà utile è LINK_FLAGS, che viene utilizzata per specificare flag di link aggiuntivi per un target specifico. I target memorizzano un elenco di librerie a cui si collegano, che vengono impostate utilizzando il comando target_link_libraries. I nomi passati a questo comando possono essere librerie, path completi alle librerie o il nome di una libreria da un comando add_library. I target memorizzano anche le directory dei collegamento da utilizzare durante il link e i comandi personalizzati da eseguire dopo il build.

5.2 Requisiti d'Uso

CMake propagherà anche i «requisiti di utilizzo» dai target della libreria linkata. I requisiti di utilizzo influenzano la compilazione dei sorgenti nel <target>. Sono specificati dalle proprietà definite sui target collegati.

Ad esempio, per specificare le directory di inclusione richieste durante il collegamento a una libreria, è possibile eseguire le seguenti operazioni:

```
add_library(foo foo.cxx)
target_include_directories(foo PUBLIC
                           "${CMAKE_CURRENT_BINARY_DIR}"
                           "${CMAKE_CURRENT_SOURCE_DIR}"
                           )
```

Ora tutto ciò che si collega al target foo avrà automaticamente il binario e il sorgente di foo come directory di inclusione. L'ordine delle directory di inclusione introdotte tramite i «requisiti di utilizzo» corrisponderà all'ordine dei target nella chiamata target_link_libraries.

Per ogni libreria o eseguibile creato da CMake, si tiene traccia di tutte le librerie da cui dipende quel target utilizzando il comando target_link_libraries. Per esempio:

```
add_library(foo foo.cxx)
target_link_libraries(foo bar)

add_executable(foobar foobar.cxx)
target_link_libraries(foobar foo)
```

linkerà le librerie «foo» e «bar» nell'eseguibile «foobar» anche se solo «foo» è stato esplicitamente specificato.

5.3 Specifica delle Librerie Ottimizzate o di Debug con un Target

Sulle piattaforme Windows, agli utenti viene spesso richiesto di linkare librerie di debug con librerie di debug e librerie ottimizzate con librerie ottimizzate. CMake aiuta a soddisfare questo requisito con il comando `target_link_libraries`, che accetta un flag opzionale etichettato come `debug` o `optimized`. Se una libreria è preceduta da `debug` o `optimized`, questa sarà collegata solo col tipo di configurazione appropriato. Per esempio

```
add_executable(foo foo.c)
target_link_libraries(foo debug libdebug optimized libopt)
```

In questo caso, foo sarà collegato a libdebug se è stata selezionata una build di debug o a libopt se è stata selezionata una build ottimizzata.

5.4 Librerie di Oggetti

I progetti di grandi dimensioni spesso organizzano i propri file sorgenti in gruppi, magari in sottodirectory separate, ciascuna delle quali necessita di diverse directory di inclusione e definizioni del preprocessore. Per questo caso d'uso CMake ha sviluppato il concetto di Object Libraries (librerie di oggetti).

Una Object Library è una raccolta di sorgenti compilati in un file oggetto che non è collegato a un file di libreria o trasformato in un archivio. Invece altri target creati da `add_library` o da `add_executable` possono fare riferimento agli oggetti utilizzando un'espressione della forma `$<TARGET_OBJECTS:name>` come sorgente, dove «name» è il target creato dalla chiamata `add_library`. Per esempio:

```
add_library(A OBJECT a.cpp)
add_library(B OBJECT b.cpp)
add_library(Combined $<TARGET_OBJECTS:A> $<TARGET_OBJECTS:B>)
```

includerà i file oggetto A e B in una libreria chiamata Combined. Le librerie di oggetti possono contenere solo sorgenti (e header) che vengono compilati in file oggetto.

5.5 File Sorgenti

La struttura del file sorgenti è per molti versi simile a un target. Memorizza il nome del file, l'estensione e una serie di proprietà generali relative a un file sorgente. Come per i target, si possono impostare e ottenere proprietà usando `set_source_files_properties` e `get_source_file_property`, o le versioni più generiche.

5.6 Directory, Test e Proprietà

Oltre ai target e ai sorgenti, ci si potrebbe trovare a lavorare occasionalmente con altri oggetti come directory e test. Normalmente tali interazioni assumono la forma impostazioni o interrogazioni delle proprietà (ad esempio `set_directory_properties` o `set_tests_properties`) da questi oggetti.

CAPITOLO 6

Policy

Occasionalmente viene apportata una nuova funzionalità o modifica a CMake che non è completamente compatibile con le versioni precedenti. Ciò può creare problemi quando qualcuno tenta di utilizzare un vecchio file CMakeLists con una nuova versione di CMake. Per aiutare sia gli utenti finali che gli sviluppatori a risolvere questi problemi, abbiamo introdotto `cmake-policies`. Le Policy sono un meccanismo per aiutare a migliorare la compatibilità con le versioni precedenti e tenere traccia dei problemi di compatibilità tra le diverse versioni di CMake.

6.1 Obiettivi del Progetto

C'erano quattro obiettivi di progettazione principali per il meccanismo delle Policy CMake:

1. I progetti esistenti devono essere compilati con versioni più recenti di CMake rispetto a quella utilizzata dagli autori del progetto.
    - Gli utenti non dovrebbero aver bisogno di modificare il codice per ottenere la compilazione dei progetti.
    - Possono essere emessi avvertimenti ma i progetti dovrebbero essere buildati.
2. La correttezza delle nuove interfacce o la correzione dei bug nelle vecchie interfacce non dovrebbe essere inibita dai requisiti di compatibilità. Qualsiasi riduzione della correttezza dell'ultima interfaccia non è giusta per i nuovi progetti.
3. Ogni modifica apportata a CMake che potrebbe richiedere modifiche ai file CMakeLists di un progetto deve essere documentata.
    - Ogni modifica deve inoltre avere un identificatore univoco a cui è possibile fare riferimento con warning e messaggi di errore.

    - Il nuovo comportamento è abilitato solo quando il progetto ha in qualche modo indicato di essere supportato.

4. Dobbiamo essere in grado di rimuovere eventualmente il codice che implementa la compatibilità con le versioni vecchie di CMake.
    - Tale rimozione è necessaria per mantenere il codice pulito e per consentire il refactoring interno.
    - Dopo tale rimozione, i tentativi di build dei progetti scritti per versioni vecchie devono fallire con un messaggio.

A tutte le policy in CMake viene assegnato un nome nel formato CMPNNNN dove NNNN è un valore intero. Le policy in genere supportano sia un vecchio comportamento che mantiene la compatibilità con le versioni precedenti di CMake, sia un nuovo comportamento considerato corretto e preferito per l'uso da parte di nuovi progetti. Ogni policy ha una documentazione che dettaglia la motivazione delle modifiche e i comportamenti vecchi e nuovi.

6.2 Impostazione delle Policy

I progetti possono configurare l'impostazione di ciascuna policy per richiedere comportamenti vecchi o nuovi. Quando CMake rileva il codice utente che potrebbe essere interessato da una particolare policy, verifica se il progetto l'ha impostata. Se la policy è stata impostata (su `OLD` o `NEW`), CMake segue il comportamento specificato. Se la policy non è stata impostata, viene utilizzato il vecchio comportamento, ma viene emesso un warning che indica all'autore del progetto di impostare il la policy.

Ci sono un paio di modi per impostare il comportamento di una policy. Il modo più rapido consiste nell'impostare tutte le policy su una versione che corrisponda a quella di rilascio di CMake in cui è stato scritto il progetto. L'impostazione della versione della policy richiede il nuovo comportamento per tutte le policy introdotte nella versione corrispondente di CMake o precedente. Le policy introdotte nelle versioni successive sono contrassegnati come «not set» per produrre messaggi appropriati. versione della policy viene impostata utilizzando del comando `cmake_policy` la firma `VERSION`. Ad esempio, il codice

```
cmake_policy(VERSION 3.20)
```

richiederà il nuovo comportamento per tutte le policy introdotte in CMake 3.20 o nelle versioni precedenti.

Il comando `cmake_minimum_required` richiederà una versione minima di CMake e chiamerà `cmake_policy`. Un progetto dovrebbe sempre iniziare con le righe

```
cmake_minimum_required(VERSION 3.20)
project(MyProject)
# ...code using CMake 3.20 policies
```

Questo indica che la persona che esegue CMake deve avere almeno la versione 3.20. Se si sta eseguendo una versione precedente di CMake, verrà visualizzato un messaggio di errore che informa che il progetto richiede almeno la versione specificata di CMake.

Ovviamente, si dovrebbe sostituire «3.20» con la versione di CMake in cui si sta attualmente scrivendo. Si può anche impostare ciascuna policy individualmente; questo a volte è utile per gli autori di progetti che desiderano convertire in modo incrementale i propri progetti per utilizzare un nuovo comportamento o silenziare i warning sulla dipendenza da un vecchio comportamento. Nel comando `cmake_policy` si può usare l'opzione `SET` per richiedere esplicitamente un comportamento vecchio o nuovo per una particolare policy.

Ad esempio, CMake 2.6 ha introdotto la policy `CMP0002`, che richiede che tutti i nomi di target logici siano univoci a livello globale (i nomi di target duplicati in precedenza funzionavano a volte per caso, ma non venivano diagnosticati). I progetti che utilizzano nomi di target duplicati e che funzionano accidentalmente riceveranno degli avvisi che fanno riferimento alla policy. I warning possono essere silenziati con il codice

```
cmake_policy(SET CMP0002 OLD)
```

che dice esplicitamente a CMake di utilizzare il vecchio comportamento per la policy (accettando tacitamente nomi di duplicati dei target). Un'altra opzione è quella di usare il codice

```
cmake_policy(SET CMP0002 NEW)
```

per dire esplicitamente a CMake di utilizzare un nuovo comportamento e produrre un errore quando viene creato un target duplicato. Una volta aggiunto al progetto, non verrà compilato fino a quando l'autore non rimuove eventuali nomi duplicati di target.

Quando viene rilasciata una nuova versione di CMake, vengono introdotte nuove policy che possono ancora creare vecchi progetti, perché per default non richiedono il comportamento `NEW` per nessuna delle nuove policy. Quando si avvia un nuovo progetto, si dovrebbe sempre specificare la versione più recente di CMake da supportare col comando `cmake_minimum_required`. Ciò assicurerà che il progetto sia scritto per funzionare utilizzando le policy di quella versione di CMake e non utilizzando alcun comportamento precedente. Se non è impostata alcuna versione della policy, CMake avviserà e assumerà una versione della policy 2.4. Ciò consente ai progetti esistenti che non specificano `cmake_minimum_required` di essere compilati come avrebbero fatto con CMake 2.4.

6.3 Lo Stack della Policy

Le impostazioni della policy vengono definite utilizzando uno stack. Un nuovo livello dello stack viene spinto [pop] quando si entra in una nuova sottodirectory del progetto (con `add_subdirectory`) e ripreso [pop] quando si esce da esso. Pertanto, l'impostazione di una policy in una directory di un progetto non influirà sulle directory padre o di pari livello, ma influirà sulle sottodirectory.

Ciò è utile quando un progetto contiene sottoprogetti che vengono mantenuti separatamente ma creati all'interno dell'albero. Il file CMakeLists di primo livello in un progetto può essere

```
cmake_policy(VERSION 2.6)
project(MyProject)
```

(continues on next page)

(continua dalla pagina precedente)

```
add_subdirectory(OtherProject)
# ... code requiring new behavior as of CMake 2.6 ...
```

mentre il file `OtherProject/CMakeLists.txt` contiene

```
cmake_policy(VERSION 2.4)
projectS(OtherProject)
# ... code that builds with CMake 2.4 ...
```

Ciò consente di aggiornare un progetto a CMake 2.6 mentre i sottoprogetti, i moduli e i file inclusi continuano a essere compilati con CMake 2.4 fino a quando i manutentori non li aggiornano.

Il codice utente può utilizzare il comando `cmake_policy` per eseguire il push e il pop dei propri livelli di stack purché ogni push sia associato a un pop. Questo è utile quando si richiede temporaneamente un comportamento diverso per una piccola sezione di codice. Ad esempio, policy `CMP0003` rimuove i link alle directory extra che venivano incluse quando veniva utilizzato un nuovo comportamento. Quando si aggiorna in modo incrementale un progetto, potrebbe essere difficile creare un target particolare con i target rimanenti OK. Il codice

```
cmake_policy(PUSH)
cmake_policy(SET CMP0003 OLD) # use old-style link for now
add_executable(myexe ...)
cmake_policy(POP)
```

silenzierà il warning e utilizzerà il vecchio comportamento per quel target. È possibile ottenere un elenco di policy e l'help su policy specifiche eseguendo CMake dalla riga di comando in questo modo

```
cmake --help-command cmake_policy
cmake --help-policies
cmake --help-policy CMP0003
```

6.4 Aggiornamento di un Progetto per una Nuova Versione di CMake

Quando una versione di CMake introduce nuove policy, potrebbe generare dei warning per alcuni progetti esistenti. Tali warning indicano che potrebbero essere necessarie modifiche a un progetto per gestire le nuove policy. Sebbene le vecchie versioni di un progetto possano continuare a essere compilate con i warning, l'albero di sviluppo del progetto dovrebbe essere aggiornato per tenere conto delle nuove policy. Esistono due approcci per l'aggiornamento di un albero: one-shot e incrementale. La questione di quale sia più facile dipende dalle dimensioni del progetto e da quali nuove policy producono warning.

6.4.1 L'Approccio One-Shot

L'approccio più semplice all'aggiornamento di un progetto per una nuova versione di CMake consiste semplicemente nel modificare la versione della policy impostata all'inizio del progetto. Poi, si prova a compilare con la nuova versione di CMake per risolvere i problemi. Ad esempio, per aggiornare un progetto da compilare con CMake 3.20, si potrebbe scrivere

```
cmake_minimum_required(VERSION 3.20)
```

all'inizio del file CMakeLists di primo livello. Ciò indica a CMake di utilizzare il nuovo comportamento per ogni policy introdotta in CMake 3.20 e versioni precedenti. Durante la build di questo progetto con CMake 3.20, non verranno prodotti warning relativi alle policy perché sa che non sono stati introdotte policy nelle versioni successive. Tuttavia, se il progetto dipendeva dal vecchio comportamento della policy, potrebbe non essere compilato poiché CMake ora utilizza il nuovo comportamento senza warning. Spetta all'autore del progetto che ha aggiunto la riga della versione della policy risolvere questi problemi.

6.4.2 L'Approccio Incrementale

Un altro approccio all'aggiornamento di un progetto per una nuova versione di CMake consiste nel gestire ogni warning uno per uno. Un vantaggio di questo approccio è che il progetto continuerà a compilarsi durante tutto il processo, quindi le modifiche possono essere apportate in modo incrementale.

Quando CMake incontra una situazione in cui deve sapere se utilizzare il vecchio o il nuovo comportamento per una policy, controlla se il progetto ha impostato la policy. Se la policy è impostata, CMake usa tacitamente il comportamento corrispondente. Se la policy non è impostata, CMake utilizza il vecchio comportamento ma avverte l'autore che la policy non è impostata.

In molti casi, un warning indicherà l'esatta riga di codice nei file CMakeLists che l'ha causato. In alcuni casi, la situazione non può essere diagnosticata fino a quando CMake non genera le regole del sistema di build nativo per il progetto, pertanto il warning non includerà informazioni esplicite del contesto. In questi casi, CMake proverà a fornire alcune informazioni su dove potrebbe essere necessario modificare il codice. La documentazione per queste policy della «generation-time» dovrebbe indicare il punto nel codice del progetto in cui la policy dovrebbe essere impostata per avere effetto.

Per aggiornare in modo incrementale un progetto, è necessario affrontare un warning alla volta. Possono verificarsi diversi casi, come descritto di seguito.

6.4.3 Tacitare un Warning Quando il Codice è Corretto

Molti warning di policy possono essere prodotti semplicemente perché il progetto non ha impostato la policy anche se il progetto potrebbe funzionare correttamente con il nuovo comportamento (non c'è modo per CMake di conoscere la differenza). Per un warning su alcuni criteri, CMP<NNNN>, si può controllare se questo è il caso aggiungendo

```
cmake_policy(SET CMP<NNNN> NEW)
```

all'inizio del progetto e cercando di compilarlo. Se il progetto viene compilato correttamente con il nuovo comportamento, si passa al warning della policy successivo. Se il progetto non viene compilato correttamente, potrebbe applicarsi uno degli altri casi.

6.4.4 Tacitare un Warning Senza Aggiornare il Codice

Gli utenti possono sopprimere tutte le istanze di un warning CMP<NNNN> aggiungendo

```
cmake_policy(SET CMP<NNNN> OLD)
```

all'inizio del progetto. Tuttavia, incoraggiamo gli autori del progetto ad aggiornare il codice in modo che funzioni con il nuovo comportamento per tutte le policy. Ciò è particolarmente importante perché le versioni di CMake nel (lontano) futuro potrebbero rimuovere il supporto per i vecchi comportamenti e produrre un errore per i progetti che li richiedono (che indica all'utente di ottenere una versione precedente di CMake per creare il progetto).

6.4.5 Tacitare i Warning Aggiornando il Codice

Quando un progetto non funziona correttamente con i NUOVI comportamenti per una policy, il codice deve essere aggiornato. Per gestire un warning per alcuni criteri CMP<NNNN>, si aggiunge

```
cmake_policy(SET CMP<NNNN> NEW)
```

all'inizio del progetto e poi si corregge il codice in modo che funzioni con il NUOVO comportamento.

Se si verificano molte istanze del warning, correggerle tutte contemporaneamente potrebbe essere troppo difficile: invece, uno sviluppatore può correggerle una alla volta utilizzando le firme PUSH/POP del comando cmake_policy:

```
cmake_policy(PUSH)
cmake_policy(SET CMP<NNNN> NEW)
# ... code updated for new policy behavior ...
cmake_policy(POP)
```

Ciò richiederà il nuovo comportamento per una piccola area di codice che sia stata corretta. Altre istanze del warning sulle policy potrebbero ancora essere visualizzate e devono essere risolte separatamente.

6.4.6 Aggiornamento della Versione della Policy del Progetto

Dopo aver risolto tutti i warning relativi alle policy e ottenuto che il progetto venga compilato in modo pulito con la nuova versione di CMake, rimane un passaggio. La versione della policy impostata all'inizio del progetto dovrebbe ora essere aggiornata in modo che corrisponda alla nuova versione di CMake, proprio come nell'approccio one-shot precedente. Ad esempio, dopo aver aggiornato un progetto per creare in modo pulito con CMake 3.20, gli utenti possono aggiornare la parte iniziale del progetto con la riga

```
cmake_minimum_required(VERSION 3.20)
```

Questo imposterà tutti i criteri introdotti in CMake 3.20 o versioni precedenti per utilizzare il nuovo comportamento. Gli utenti, poi, possono esaminare il resto del codice e rimuovere le chiamate che utilizzano il comando `cmake_policy` per richiedere il nuovo comportamento in modo incrementale. Il risultato finale dovrebbe essere lo stesso dell'approccio one-shot, ma potrebbe essere raggiunto passo dopo passo.

6.4.7 Supporto di Versioni Multiple di CMake

Alcuni progetti potrebbero voler supportare contemporaneamente alcune versioni di CMake. L'obiettivo è compilare con una versione precedente, lavorando anche con versioni più recenti senza warning. Per supportare sia CMake 2.4 che 2.6, è possibile scrivere codice come

```
cmake_minimum_required(VERSION 2.4)
if(COMMAND cmake_policy)
  # policy settings ...
  cmake_policy(SET CMP0003 NEW)
endif()
```

Questo imposterà le policy per la compilazione con CMake 2.6 e per ignorarli per CMake 2.4. Per supportare sia CMake 2.6 che alcune policy di CMake 2.8, è possibile scrivere codice come:

```
cmake_minimum_required(VERSION 2.6)
if(POLICY CMP1234)
  # policies not known to CMake 2.6 ...
  cmake_policy(SET CMP1234 NEW)
endif()
```

Questo imposterà le policy per la compilazione con CMake 2.8 e per ignorarli per CMake 2.6. Se è noto che il progetto viene compilato con le nuove policy di CMake 2.6 e CMake 2.8, gli utenti possono scrivere:

```
cmake_minimum_required(VERSION 2.6)
if (NOT ${CMAKE_VERSION} VERSION_LESS 2.8)
   cmake_policy(VERSION 2.8)
endif()
```

Controllo delle Versioni di CMake

CMake è un programma in evoluzione e man mano che vengono rilasciate nuove versioni, vengono introdotte nuove funzionalità o comandi. Di conseguenza, potrebbero esserci casi in cui si potrebbe voler utilizzare un comando che si trova in una versione corrente di CMake ma non nelle versioni precedenti. Ci sono un paio di modi per gestirlo; un'opzione è usare il comando `if` per controllare se esiste un nuovo comando. Per esempio:

```
# test if the command exists
if(COMMAND some_new_command)
  # use the command
  some_new_command( ARGS...)
endif()
```

In alternativa, è possibile testare la versione effettiva di CMake in esecuzione valutando la variabile `CMAKE_VERSION`:

```
# look for newer versions of CMake
if(${CMAKE_VERSION} VERSION_GREATER 3.20)
  # do something special here
endif()
```

Infine, alcune nuove versioni di CMake potrebbero non supportare più alcuni comportamenti utilizzati (sebbene cerchiamo di evitarlo). In questi casi, si utilizzano le politiche di CMake, come discusso nel manuale `cmake-policies`.

CAPITOLO 7

Moduli

7.1 Utilizzo dei Moduli

Il riutilizzo del codice è una tecnica preziosa nello sviluppo del software e CMake è stato progettato per supportarlo. Consentire ai file CMakeLists di utilizzare moduli riutilizzabili consente all'intera comunità di condividere sezioni di codice riutilizzabili. Per CMake, queste sezioni sono chiamate `cmake-modules` e si trovano nella sottodirectory Modules dell'installazione.

La posizione di un modulo può essere specificata utilizzando il path completo del file del modulo o lasciando che CMake lo trovi da solo. CMake cercherà i moduli nelle directory specificate da `CMAKE_MODULE_PATH`; se non riesce a trovarlo lì, cercherà nella sottodirectory Modules. In questo modo i progetti possono sovrascrivere i moduli forniti da CMake e personalizzarli in base alle proprie esigenze. I moduli possono essere suddivisi in alcune categorie principali:

7.1.1 Moduli Find

Questi moduli supportano il comando `find_package` per determinare la posizione degli elementi software, come i file header o le librerie, che appartengono a un determinato pacchetto. Non si devono includere direttamente. Si usa il comando `find_package`. Ogni modulo viene fornito con una documentazione che descrive il pacchetto che trova e le variabili in cui fornisce i risultati.

7.1.2 Moduli Utility

I moduli utility sono semplicemente sezioni di comandi CMake inseriti in un file; possono quindi essere inclusi in altri file CMakeLists utilizzando il comando `include`. Ad esempio, i seguenti comandi includeranno il modulo `CheckTypeSize` di CMake e poi utilizzeranno la macro che definisce.

```
include(CheckTypeSize)
check_type_size(long SIZEOF_LONG)
```

Questi moduli testano il sistema per fornire informazioni sulla piattaforma o sul compilatore target, ad esempio la dimensione di un float o il supporto per gli stream ANSI C++. Molti di questi moduli hanno nomi preceduti da `Test` o `Check`, come `TestBigEndian` e `CheckTypeSize`. Alcuni di loro cercano di compilare il codice per determinare il risultato corretto. In questi casi, il codice sorgente è tipicamente denominato come il modulo, ma con un'estensione `.c` o `.cxx`. I moduli di utility forniscono anche utili macro e funzioni implementate nel linguaggio CMake e destinate a casi d'uso specifici e comuni. Consultare la documentazione di ciascun modulo per i dettagli.

CAPITOLO 8

Installazione di File

Il software viene in genere installato in una directory separata dalle directory dei sorgenti e da quelle di build. Ciò consente di distribuirlo in una forma pulita e isola gli utenti dai dettagli del processo di build. CMake fornisce il comando `install` per specificare come deve essere installato un progetto. Questo comando viene richiamato da un progetto nel file CMakeLists e indica a CMake come generare gli script di installazione. Gli script vengono eseguiti al momento dell'installazione per fare l'effettiva installazione dei file. Per i generatori di Makefile (UNIX, NMake, MinGW, ecc.), l'utente esegue semplicemente `make install` (o `nmake install`) e lo strumento make invocherà il modulo di installazione di CMake. Con i sistemi basati su GUI (Visual Studio, Xcode, ecc.), l'utente builda semplicemente il target chiamato `INSTALL`.

Ogni chiamata al comando `install` definisce alcune regole di installazione. All'interno di un file CMakeLists (directory sorgente), queste regole verranno valutate nell'ordine in cui vengono richiamati i comandi corrispondenti. L'ordine tra directory multiple è cambiato in CMake 3.14.

Il comando `install` ha diverse firme progettate per casi d'uso di installazione comuni. Una particolare invocazione del comando specifica la firma come primo argomento. Le firme [signature] sono `TARGETS`, `FILES` o `PROGRAMS`, `DIRECTORY`, `SCRIPT`, `CODE` e `EXPORT`.

install(TARGETS ...)
: Installa i file binari corrispondenti ai target compilati all'interno del progetto.

install(FILES ...)
: Installa file per uso generico, generalmente utilizzata per file header, documentazione e file di dati richiesti dal software.

install(PROGRAMS ...)
: Installa i file eseguibili non compilati dal progetto, come gli script della shell. Questo argomento è identico a `install(FILES)` tranne per il fatto che i permessi di default del file installato includono il bit eseguibile.

install(DIRECTORY ...)
Questo argomento installa un intero albero di directory. Può essere utilizzato per installare directory con risorse, come icone e immagini.

install(SCRIPT ...)
Specifica un file di script CMake fornito dall'utente da eseguire durante l'installazione. Questo è in genere utilizzato per definire le azioni pre-installazione o post-installazione per altre regole.

install(CODE ...)
Specifica il codice CMake fornito dall'utente da eseguire durante l'installazione. È simile a `install (SCRIPT)` ma il codice viene fornito in linea nella chiamata come stringa.

install(EXPORT ...)
Genera e installa un file CMake contenente il codice per importare i target dall'albero di installazione in un altro progetto.

Le firme `TARGETS`, `FILES`, `PROGRAMS` e `DIRECTORY` hanno tutte lo scopo di creare regole di installazione per i file. I target, i file o le directory da installare sono elencati immediatamente dopo l'argomento del nome della firma. Ulteriori dettagli possono essere specificati utilizzando argomenti di parole chiave seguiti dai valori corrispondenti. Gli argomenti delle parole chiave forniti dalla maggior parte delle firme sono i seguenti.

DESTINATION
Questo argomento specifica la posizione in cui la regola di installazione inserirà i file e deve essere seguito da un path di directory che indica la posizione. Se la directory viene specificata come percorso completo, verrà valutata al momento dell'installazione come path assoluto. Se la directory è specificata come percorso relativo, verrà valutata al momento dell'installazione rispetto al prefisso di installazione. Il prefisso può essere impostato dall'utente tramite la variabile cache `CMAKE_INSTALL_PREFIX`. CMake fornisce un'impostazione di default specifica per la piattaforma: `/usr/local` su UNIX e «<SystemDrive>/`Program Files`/<ProjectName>» su Windows, dove SystemDrive è sulla falsariga di `C:` e ProjectName è il nome dato al comando `project` più in alto.

PERMISSIONS
Questo argomento specifica le autorizzazioni da impostare sui file installati. Questa opzione è necessaria solo per sovrascrivere le autorizzazioni di default selezionate da una particolare firma del comando `install`. I permessi validi sono `OWNER_READ`, `OWNER_WRITE`, `OWNER_EXECUTE`, `GROUP_READ`, `GROUP_WRITE`, `GROUP_EXECUTE`, `WORLD_READ`, `WORLD_WRITE`, `WORLD_EXECUTE`, `SETUID` e `SETGID`. Alcune piattaforme non supportano tutte queste autorizzazioni; su queste tali nomi di autorizzazione vengono ignorati.

CONFIGURATIONS
Questo argomento specifica un elenco di configurazioni di build per le quali si applica una regola di installazione (Debug, Release e così via). Per i generatori di Makefile, la configurazione della build è specificata dalla variabile della cache `CMAKE_BUILD_TYPE`. Per i generatori di Visual Studio e Xcode, la configurazione viene selezionata quando viene creato il target `install`. Una regola di installazione verrà valutata solo se la configurazione dell'installazione corrente corrisponde a una voce nell'elenco fornito a questo

argomento. Il confronto dei nomi di configurazione non fa distinzione tra maiuscole e minuscole.

COMPONENT
Questo argomento specifica il componente dell'installazione per il quale si applica la regola di installazione. Alcuni progetti suddividono le loro installazioni in più componenti per packaging separati. Ad esempio, un progetto può definire un componente `Runtime` che contiene i file necessari per eseguire un tool; un componente `Development` contenente i file necessari per creare estensioni per il tool; e un componente `Documentation` contenente le pagine del manuale e altri file di aiuto. Il progetto può quindi impacchettare ciascun componente separatamente per la distribuzione installando un solo componente alla volta. Per default, tutti i componenti vengono installati. L'installazione «component-specific» è una funzionalità avanzata destinata all'uso da parte dei manutentori dei pacchetti. Richiede l'invocazione manuale degli script di installazione con un argomento che definisce la variabile `COMPONENT` per denominare il componente desiderato. Si noti che i nomi dei componenti non sono definiti da CMake. Ogni progetto può definire il proprio insieme di componenti.

OPTIONAL
Questo argomento specifica che non è un errore se il file di input da installare non esiste. Se il file di input esiste, verrà installato come richiesto. Se non esiste, verrà tacitamente non installato.

8.1 Installazione dei Target

I progetti in genere installano parte della libreria e dei file eseguibili creati durante il processo di build. Il comando `install` fornisce la firma `TARGETS` per questo scopo.

La parola chiave `TARGETS` viene immediatamente seguita da un elenco dei target creati utilizzando `add_executable` o `add_library` e che devono essere installati. Verranno installati uno o più file corrispondenti a ciascun target.

I file installati con questa firma possono essere suddivisi in categorie come `ARCHIVE`, `LIBRARY` o `RUNTIME`. Tali categorie sono progettate per raggruppare i file target in base alla destinazione di installazione tipica. Gli argomenti delle parole chiave corrispondenti sono facoltativi, ma se presenti, specifica che gli altri argomenti che li seguono si applicano solo ai file target di quel tipo. I file target sono classificati come segue:

executables - `RUNTIME`
Creato da `add_executable` (.exe su Windows, senza estensione su UNIX)

loadable modules - `LIBRARY`
Creato da `add_library` con l'opzione `MODULE` (.dll su Windows, .so su UNIX)

shared libraries - `LIBRARY`
Creato da `add_library` con l'opzione `SHARED` su piattaforme simili a UNIX (.so sulla maggior parte degli UNIX, .dylib su Mac)

dynamic-link libraries - `RUNTIME`
Creato da `add_library` con l'opzione `SHARED` su piattaforme Windows (.dll)

import libraries - ARCHIVE
: Un file linkabile creato da una libreria dinamica che esporta simboli (.lib sulla maggior parte di Windows, .dll.a su Cygwin e MinGW).

static libraries - ARCHIVE
: Creato da `add_library` con l'opzione `STATIC` (.lib su Windows, .a su UNIX, Cygwin e MinGW)

Si considera un progetto che definisce un eseguibile, `myExecutable`, che si linka a una libreria shared `mySharedLib`. Fornisce inoltre una libreria statica `myStaticLib` e un modulo plugin per l'eseguibile chiamato `myPlugin` che si collega anche alla libreria shared. L'eseguibile, la libreria statica e il file plugin possono essere installati individualmente utilizzando i comandi

```
install(TARGETS myExecutable DESTINATION bin)
install(TARGETS myStaticLib DESTINATION lib/myproject)
install(TARGETS myPlugin DESTINATION lib)
```

L'eseguibile non sarà in grado di essere eseguito dalla posizione di installazione fino a quando non sarà installata anche la libreria shared a cui si collega. L'installazione della libreria richiede un po" più di cura per supportare tutte le piattaforme. Deve essere installata in una posizione cercata dal linker dinamico su ogni piattaforma. Su piattaforme UNIX-like, la libreria è generalmente installata in `lib`, mentre su Windows dovrebbe essere posizionata accanto all'eseguibile in `bin`. Un'ulteriore sfida è che la libreria di importazione associata alla libreria shared su Windows dovrebbe essere trattata come libreria statica e installata in `lib/myproject`. In altre parole, abbiamo tre diversi tipi di file creati con un unico nome di target che devono essere installati in tre diverse destinazioni! Fortunatamente, questo problema può essere risolto utilizzando la categoria di argomenti delle parole chiavi. La libreria shared può essere installata utilizzando il comando:

```
install(TARGETS mySharedLib
        RUNTIME DESTINATION bin
        LIBRARY DESTINATION lib
        ARCHIVE DESTINATION lib/myproject)
```

Questo dice a CMake che il file di `RUNTIME` (.dll) deve essere installato in `bin`, il file `LIBRARY` (.so) deve essere installato in `lib` e il file `ARCHIVE` (.lib) dovrebbe essere installato in `lib/myproject`. Su UNIX, verrà installato il file `LIBRARY`; su Windows, verranno installati i file `RUNTIME` e `ARCHIVE`.

Se il progetto di esempio precedente deve essere impacchettato in componenti separati di runtime e di sviluppo, dobbiamo assegnare il componente appropriato a ciascun file target installato. L'eseguibile, la libreria shared e il plug-in sono necessari per eseguire l'applicazione, quindi appartengono a un componente `Runtime`. Nel frattempo, la libreria di importazione (corrispondente alla libreria shared su Windows) e la libreria statica sono necessarie solo per sviluppare estensioni all'applicazione, e quindi appartengono a un componente `Development`.

Le assegnazioni dei componenti possono essere specificate aggiungendo l'argomento `COMPONENT` a ciascuno dei comandi precedenti. È inoltre possibile combinare tutte le regole di installazione in un'unica chiamata di comando, che equivale a tutti i comandi precedenti con l'aggiunta dei componenti. I file generati da ciascun target vengono installati utilizzando la regola per la loro categoria.

```
install(TARGETS myExecutable mySharedLib myStaticLib myPlugin
        RUNTIME DESTINATION bin           COMPONENT Runtime
        LIBRARY DESTINATION lib           COMPONENT Runtime
        ARCHIVE DESTINATION lib/myproject COMPONENT Development)
```

Si può specificare NAMELINK_ONLY o NAMELINK_SKIP come opzione di LIBRARY. Su alcune piattaforme, una libreria shared versionata ha un link simbolico come

```
lib<name>.so -> lib<name>.so.1
```

dove lib<name>.so.1 è il «soname» della libreria, e lib<name>.so è un «namelink» che aiuta i linker a trovare la libreria quando viene dato -l<name>. L'opzione NAMELINK_ONLY comporta l'installazione del solo namelink quando è installata una libreria target. L'opzione NAMELINK_SKIP provoca l'installazione di file di libreria diversi dal namelink quando viene installato un target di libreria. Quando non viene fornita nessuna opzione, vengono installate entrambe le parti. Sulle piattaforme in cui le librerie shared con versione non hanno collegamenti ai nomi o quando una libreria non è dotata di versione, l'opzione NAMELINK_SKIP installa la libreria e l'opzione NAMELINK_ONLY non installa nulla. Consultare le proprietà dei target VERSION e SOVERSION per i dettagli sulla creazione di librerie shared e con una versione.

8.2 Installazione di File

I progetti possono installare file diversi da quelli creati con add_executable o con add_library, come file header o documentazione. L'installazione generica dei file viene specificata utilizzando la firma FILES.

La parola chiave FILES è immediatamente seguita da un elenco di file da installare. I path relativi vengono valutati rispetto alla directory dei sorgenti corrente. I file verranno installati nella directory DESTINATION specificata. Ad esempio, il comando

```
install(FILES my-api.h ${CMAKE_CURRENT_BINARY_DIR}/my-config.h
        DESTINATION include)
```

installa il file my-api.h dall'albero dei sorgenti e il file my-config.h dall'albero di build nella directory include sotto il prefisso di installazione. Per default, ai file installati vengono assegnate le autorizzazioni OWNER_WRITE, OWNER_READ, GROUP_READ e WORLD_READ, ma questo potrebbe essere sovrascritto specificando l'opzione PERMISSIONS. Si considerino i casi in cui gli utenti vorrebbero installare un file di configurazione globale su un sistema UNIX leggibile solo dal suo proprietario (come root). Lo si fa con questo comando

```
install(FILES my-rc DESTINATION /etc
        PERMISSIONS OWNER_WRITE OWNER_READ)
```

che installa il file my-rc con i permessi di lettura/scrittura del proprietario nel path assoluto /etc.

L'argomento RENAME specifica un nome per un file installato che potrebbe essere diverso dal file originale. La ridenominazione è consentita solo quando un singolo file viene installato dal comando. Ad esempio, il comando

```
install(FILES version.h DESTINATION include RENAME my-version.h)
```

installerà il file version.h dalla directory dei sorgenti in include/my-version.h sotto il prefisso di installazione.

8.3 Installazione dei Programmi

I progetti possono anche installare programmi di supporto, come script di shell o script Python che non sono effettivamente compilati come target. Questi possono essere installati con la firma FILES utilizzando l'opzione PERMISSIONS per aggiungere il permesso di esecuzione. Tuttavia, questo caso è abbastanza comune da giustificare un'interfaccia più semplice. CMake fornisce la firma PROGRAMS per questo scopo.

La parola chiave PROGRAMS è immediatamente seguita da un elenco di script da installare. Questo comando è identico alla firma FILES, tranne per il fatto che le autorizzazioni di default includono anche OWNER_EXECUTE, GROUP_EXECUTE e WORLD_EXECUTE. Ad esempio, possiamo installare uno script di utilità Python col comando

```
install(PROGRAMS my-util.py DESTINATION bin)
```

che installa my-util.py nella directory bin sotto il prefisso di installazione e gli fornisce i permessi di proprietario, gruppo, lettura ed esecuzione universali, più i permessi di scrittura per il proprietario.

8.4 Installazione di Directory

I progetti possono anche fornire un'intera directory piena di file di risorse, come icone o documentazione html. È possibile installare un'intera directory utilizzando la firma DIRECTORY.

La parola chiave DIRECTORY è immediatamente seguita da un elenco di directory da installare. I path relativi vengono valutati rispetto alla directory dei sorgenti corrente. Ogni directory denominata viene installata nella directory di destinazione. L'ultimo componente di ogni nome di directory di input viene aggiunto alla directory di destinazione quando tale directory viene copiata. Ad esempio, il comando

```
install(DIRECTORY data/icons DESTINATION share/myproject)
```

installerà la directory data/icons dall'albero dei sorgenti in share/myproject/icons sotto il prefisso di installazione. Una barra finale lascerà vuoto l'ultimo componente e installerà il contenuto della directory di input nella destinazione. Il comando

```
install(DIRECTORY doc/html/ DESTINATION doc/myproject)
```

installa il contenuto di `doc/html` dalla directory di origine in `doc/myproject` sotto il prefisso di installazione. Se non vengono forniti nomi di directory di input, come in

```
install(DIRECTORY DESTINATION share/myproject/user)
```

la directory di destinazione verrà creata ma non verrà installato nulla al suo interno.

I file installati dalla firma `DIRECTORY` hanno le stesse autorizzazioni di default della firma `FILES`. Le directory installate dalla firma `DIRECTORY` hanno le stesse autorizzazioni di default della firma `PROGRAMS`. Le opzioni `FILE_PERMISSIONS` e `DIRECTORY_PERMISSIONS` possono essere utilizzate per sovrascrivere queste impostazioni di default. Si consideri un caso in cui una directory piena di script di shell di esempio deve essere installata in una directory che sia scrivibile sia dal proprietario che dal gruppo. Possiamo usare il comando

```
install(DIRECTORY data/scripts DESTINATION share/myproject
        FILE_PERMISSIONS
          OWNER_READ OWNER_EXECUTE OWNER_WRITE
          GROUP_READ GROUP_EXECUTE
          WORLD_READ WORLD_EXECUTE
        DIRECTORY_PERMISSIONS
          OWNER_READ OWNER_EXECUTE OWNER_WRITE
          GROUP_READ GROUP_EXECUTE GROUP_WRITE
          WORLD_READ WORLD_EXECUTE
        )
```

che installa la directory `data/scripts` in `share/myproject/scripts` e imposta i permessi desiderati. In alcuni casi, una directory di input completamente preparata creata dal progetto potrebbe avere già impostate le autorizzazioni desiderate. L'opzione `USE_SOURCE_PERMISSIONS` indica a CMake di utilizzare i permessi di file e directory dalla directory di input durante l'installazione. Se nell'esempio precedente la directory di input fosse già stata predisposta con i permessi corretti, potrebbe essere stato utilizzato invece il seguente comando:

```
install(DIRECTORY data/scripts DESTINATION share/myproject
        USE_SOURCE_PERMISSIONS)
```

Se la directory di input da installare è sotto la gestione dei sorgenti, potrebbero esserci sottodirectory aggiuntive nell'input che non si desiderano installare. Potrebbero esserci anche file specifici che non devono essere installati o che devono essere installati con autorizzazioni diverse, mentre la maggior parte dei file ottiene i valori di default. Le opzioni `PATTERN` e `REGEX` possono essere utilizzate per questo scopo. Un'opzione `PATTERN` è seguita prima da un «glob pattern» e poi da un'opzione `EXCLUDE` o `PERMISSIONS`. Un'opzione `REGEX` è seguita prima da un'espressione regolare e poi da `EXCLUDE` o `PERMISSIONS`. L'opzione `EXCLUDE` salta l'installazione di quei file o directory che corrispondono al modello o all'espressione precedente, mentre l'opzione `PERMISSIONS` assegna loro autorizzazioni specifiche.

Ogni file e directory di input viene testato rispetto al modello o all'espressione regolare come path completo con barre. Un pattern corrisponderà solo ai nomi completi di file o directory

che si trovano alla fine del path completo, mentre un'espressione regolare può corrispondere a qualsiasi parte. Ad esempio, il pattern `foo*` corrisponderà a `.../foo.txt` ma non a `.../myfoo.txt` né a `.../foo/bar.txt`; tuttavia, l'espressione regolare `foo` corrisponderà a tutti.

Tornando all'esempio precedente di installazione di una directory di icone, si consideri il caso in cui la directory di input è gestita da git e contiene anche alcuni file di testo extra che non vogliamo installare. Il comando

```
install(DIRECTORY data/icons DESTINATION share/myproject
        PATTERN ".git" EXCLUDE
        PATTERN "*.txt" EXCLUDE)
```

installa la directory delle icone ignorando qualsiasi directory .git o file di testo contenuto. Il comando equivalente che utilizza l'opzione `REGEX` è

```
install(DIRECTORY data/icons DESTINATION share/myproject
        REGEX "/.git$" EXCLUDE
        REGEX "/[^/]*.txt$" EXCLUDE)
```

che utilizza "/" e "$" per vincolare la corrispondenza allo stesso modo dei pattern. Si consideri un caso simile in cui la directory di input contiene script di shell e file di testo che desideriamo installare con autorizzazioni diverse rispetto agli altri file. Il comando

```
install(DIRECTORY data/other/ DESTINATION share/myproject
        PATTERN ".git" EXCLUDE
        PATTERN "*.txt"
          PERMISSIONS OWNER_READ OWNER_WRITE
        PATTERN "*.sh"
          PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE)
```

installerà il contenuto di `data/other` dalla directory dei sorgenti a `share/myproject` ignorando le directory .git e fornendo autorizzazioni specifiche ai file `.txt` e `.sh`.

8.5 Installazione di Script

Potrebbe essere necessario che le installazioni del progetto eseguano attività diverse dal semplice inserimento di file nell'albero di installazione. I pacchetti di terze parti possono fornire i propri meccanismi per la registrazione di nuovi plug-in che devono essere richiamati durante l'installazione del progetto. La firma `SCRIPT` viene fornita a questo scopo.

La parola chiave `SCRIPT` è immediatamente seguita dal nome di uno script CMake. CMake eseguirà lo script durante l'installazione. Se il nome del file fornito è un path relativo, verrà valutato rispetto alla directory dei sorgenti corrente. Un semplice caso d'uso è la stampa di un messaggio durante l'installazione. Per prima cosa scriviamo un file `message.cmake` contenente il codice

```
message("Installing My Project")
```

e poi fare riferimento a questo script col comando:

```
install(SCRIPT message.cmake)
```

Gli script di installazione personalizzati non vengono eseguiti durante l'elaborazione del file CMakeLists principale; vengono eseguiti durante il processo di installazione stesso. Le variabili e le macro definite nel codice contenente la chiamata `install (SCRIPT)` non saranno accessibili dallo script. Tuttavia, ci sono alcune variabili definite durante l'esecuzione dello script utilizzabili per ottenere informazioni sull'installazione. La variabile `CMAKE_INSTALL_PREFIX` è impostata sul prefisso di installazione effettivo. Questa può essere diversa dal valore della variabile di cache corrispondente, poiché gli script di installazione possono essere eseguiti da uno strumento di pacchettizzazione che utilizza un prefisso diverso. Una variabile di ambiente `ENV{DESTDIR}` può essere impostata dall'utente o dal tool di pacchettizzazione. Il suo valore viene anteposto al prefisso di installazione e ai percorsi di installazione assoluti per determinare la posizione in cui sono installati i file. Per fare riferimento a un path di installazione su disco, lo script custom può utilizzare `$ENV{DESTDIR}${CMAKE_INSTALL_PREFIX}` come parte superiore del path. La variabile `CMAKE_INSTALL_CONFIG_NAME` è impostata sul nome della configurazione di build attualmente installata (Debug, Release, ecc.). Durante l'installazione specifica del componente, la variabile `CMAKE_INSTALL_COMPONENT` è impostata sul nome del componente corrente.

8.6 Installazione del Codice

Gli script di installazione personalizzati, semplici come il messaggio precedente, vengono creati più facilmente con il codice dello script inserito in linea nella chiamata al comando `install`. La firma `CODE` viene fornita a questo scopo.

La parola chiave `CODE` è immediatamente seguita da una stringa contenente il codice da inserire nello script di installazione. È possibile creare un messaggio al momento dell'installazione utilizzando il comando

```
install(CODE "MESSAGE(\"Installing My Project\")")
```

che ha lo stesso effetto dello script `message.cmake` ma contiene il codice in linea.

8.7 Installazione di Librerie Shared Prerequisite

Gli eseguibili vengono spesso creati utilizzando librerie shared come elementi costitutivi. Quando si installa un eseguibile di questo tipo, è necessario installare anche le sue librerie shared prerequisite, chiamate «prerequisiti» perché l'eseguibile richiede la loro presenza per essere caricato ed eseguito correttamente. Le tre principali sorgenti delle librerie shared sono il sistema operativo stesso, i prodotti della build del proprio progetto e le librerie di terze parti appartenenti a un progetto esterno. Si può fare affidamento su quelli del sistema operativo per essere presenti senza installare nulla: sono sulla piattaforma di base in cui viene eseguito l'eseguibile. I prodotti della build nel progetto presumibilmente hanno la regola di build `add_library` nei file CMakeLists, quindi dovrebbe essere semplice creare regole di installazione di CMake per loro. Sono le librerie di terze parti che spesso diventano un elemento di manutenzione elevata

quando ce ne sono parecchie o quando si passa da una versione all'altra del progetto di terze parti. Le librerie si possono aggiungere, il codice può essere riorganizzato e le stesse librerie shared di terze parti potrebbero effettivamente avere prerequisiti aggiuntivi che non sono ovvi a prima vista.

CMake fornisce un modulo, `BundleUtilities` per semplificare la gestione delle librerie shared richieste. Questo modulo fornisce la funzione `fixup_bundle` per copiare e correggere librerie shared prerequisite utilizzando posizioni ben definite relative all'eseguibile. Per le applicazioni bundle Mac, si racchiudono le librerie all'interno del bundle, correggendole con `install_name_tool` per creare un'unità autonoma. Su Windows, si copiano le librerie nella stessa directory con l'eseguibile poiché gli eseguibili cercheranno nelle proprie directory le DLL richieste.

La funzione `fixup_bundle` aiuta a creare alberi di installazione rilocabili. Gli utenti Mac apprezzano le applicazioni bundle autonome: si possono trascinare ovunque, fare doppio clic su di esse e funzionano ancora. Non si basano su nulla di installato in una determinata posizione diversa dal sistema operativo stesso. Allo stesso modo, gli utenti Windows senza privilegi amministrativi apprezzano un albero di installazione rilocabile in cui un eseguibile e tutte le DLL richieste sono installate nella stessa directory, in modo che funzioni indipendentemente da dove lo si installa. Si possono persino spostare le cose dopo averle installate e funzionerà comunque.

Per utilizzare `fixup_bundle`, si installa prima uno dei target eseguibili. Poi, si configura uno script CMake richiamabile al momento dell'installazione. All'interno dello script configurato di CMake, semplicemente si `include BundleUtilities` e si chiama la funzione `fixup_bundle` con gli argomenti appropriati.

In CMakeLists.txt

```
install(TARGETS myExecutable DESTINATION bin)

# To install, for example, MSVC runtime libraries:
include(InstallRequiredSystemLibraries)

# To install other/non-system 3rd party required libraries:
configure_file(
  ${CMAKE_CURRENT_SOURCE_DIR}/FixBundle.cmake.in
  ${CMAKE_CURRENT_BINARY_DIR}/FixBundle.cmake
  @ONLY
  )

install(SCRIPT ${CMAKE_CURRENT_BINARY_DIR}/FixBundle.cmake)
```

In FixBundle.cmake.in:

```
include(BundleUtilities)

# Set bundle to the full path name of the executable already
# existing in the install tree:
set(bundle
   "${CMAKE_INSTALL_PREFIX}/myExecutable@CMAKE_EXECUTABLE_SUFFIX@")
```

(continues on next page)

(continua dalla pagina precedente)

```
# Set other_libs to a list of full path names to additional
# libraries that cannot be reached by dependency analysis.
# (Dynamically loaded PlugIns, for example.)
set(other_libs "")

# Set dirs to a list of directories where prerequisite libraries
# may be found:
set(dirs
   "@CMAKE_RUNTIME_OUTPUT_DIRECTORY@"
   "@CMAKE_LIBRARY_OUTPUT_DIRECTORY@"
   )

fixup_bundle("${bundle}" "${other_libs}" "${dirs}")
```

Si è responsabili di verificare di disporre dell'autorizzazione per copiare e distribuire le librerie shared prerequisite per l'eseguibile. Alcune librerie potrebbero avere licenze software restrittive che vietano di fare copie alla `fixup_bundle`.

CAPITOLO 9

Ispezione del Sistema

Questo capitolo descriverà come utilizzare CMake per ispezionare l'ambiente del sistema in cui viene creato il software. Questo è un fattore critico nella creazione di applicazioni o librerie multipiattaforma. Descrive come trovare e utilizzare i file header e le librerie installati dal sistema e dall'utente. Copre anche alcune delle funzionalità più avanzate di CMake, inclusi i comandi `try_compile` e `try_run`. Questi comandi sono strumenti estremamente potenti per determinare le capacità del sistema e del compilatore che ospita il software.

9.1 Utilizzo di File Header e Librerie

Molti programmi C e C++ dipendono da librerie esterne; tuttavia, quando si tratta degli aspetti pratici della compilazione e del link di un progetto, sfruttare le librerie esistenti può risultare difficile sia per gli sviluppatori che per gli utenti. I problemi in genere si manifestano non appena il software viene costruito su un sistema diverso da quello su cui è stato sviluppato. I presupposti relativi alla posizione delle librerie e dei file header diventano palesi quando non sono installati nella stessa posizione sul nuovo computer e il sistema di build non è in grado di trovarli. CMake ha molte funzionalità per aiutare gli sviluppatori nell'integrazione di librerie software esterne in un progetto.

I comandi CMake più rilevanti per questo tipo di integrazione sono `find_file`, `find_library`, `find_path`, `find_program` e `find_package`. Per la maggior parte delle librerie C e C++, una combinazione di `find_library` e `find_path` sarà sufficiente per compilare e linkare una libreria installata. Il comando `find_library` è utilizzabile per individuare o consentire a un utente di individuare una libreria e `find_path` può essere utilizzato per trovare il path di un file include rappresentativo del progetto. Ad esempio, volendo linkare la libreria tiff, si possono utilizzare i seguenti comandi nel file CMakeLists.txt

```
# find libtiff, looking in some standard places
find_library(TIFF_LIBRARY
             NAMES tiff tiff2
             PATHS /usr/local/lib /usr/lib
            )

# find tiff.h looking in some standard places
find_path(TIFF_INCLUDES tiff.h
           /usr/local/include
           /usr/include
          )

add_executable(mytiff mytiff.c )

target_link_libraries(mytiff ${TIFF_LIBRARY})

target_include_directories(mytiff ${TIFF_INCLUDES})
```

Il primo comando utilizzato è `find_library`, che in questo caso cercherà una libreria con il nome tiff o tiff2. Il comando `find_library` richiede solo il nome di base della libreria senza prefissi o suffissi specifici della piattaforma, come .lib e .dll. I prefissi e i suffissi appropriati per il sistema che esegue CMake verranno aggiunti automaticamente al nome della libreria quando CMake la cerca. Tutti i comandi `FIND_*` cercheranno nella variabile d'ambiente `PATH`. Inoltre, i comandi consentono di specificare ulteriori path di ricerca come argomenti da elencare dopo l'argomento marcatore `PATHS`. Oltre a supportare i path standard, le voci di registro di Windows e le variabili di ambiente possono essere utilizzate per creare path di ricerca. La sintassi per le voci di registro è la seguente:

```
[HKEY_CURRENT_USER\\Software\\Kitware\\Path;Build1]
```

Poiché il software può essere installato in molti luoghi diversi, è impossibile per CMake trovare la libreria ogni volta, ma la maggior parte delle installazioni standard dovrebbe essere coperta. I comandi `find_*` creano automaticamente una variabile di cache in modo che gli utenti possano sovrascrivere o specificare la posizione dalla GUI di CMake. In questo modo, se CMake non è in grado di individuare i file che sta cercando, gli utenti avranno comunque la possibilità di specificarli. Se CMake non trova un file, il valore viene impostato su `VAR-NOTFOUND`; questo valore indica a CMake che dovrebbe continuare a cercare ogni volta che viene eseguito il passaggio di configurazione di CMake. Notare che nelle istruzioni if, i valori di `VAR-NOTFOUND` saranno valutati come falsi.

Il comando successivo utilizzato è `find_path`, un comando generico che, in questo esempio, viene utilizzato per individuare un file header dalla libreria. I file header e le librerie sono spesso installati in posizioni diverse ed entrambe le posizioni sono necessarie per compilare e linkare i programmi che li utilizzano. L'uso di `find_path` è simile a `find_library`, sebbene supporti solo un nome, un elenco di path di ricerca.

Il resto del file CMakeLists può utilizzare le variabili create dai comandi `find_*`. Le variabili possono essere utilizzate senza controllare che i valori siano validi, poiché CMake stamperà un messaggio di errore che notifica all'utente se una qualsiasi delle variabili richieste non è stata

impostata. L'utente può poi impostare i valori della cache e riconfigurare finché il messaggio non scompare. Facoltativamente, un file CMakeLists potrebbe utilizzare il comando `if` per utilizzare librerie o opzioni alternative per creare il progetto se non è possibile trovare la libreria.

Dall'esempio sopra vede come l'uso dei comandi `find_*` può aiutare il software a compilare su una varietà di sistemi. Vale la pena notare che i comandi `find_*` cercano una corrispondenza che inizia con il primo argomento e il primo path, quindi quando si elencano path e nomi di librerie, si devono elencare prima i path e nomi preferiti. Se ci sono più versioni di una libreria e si preferisce tiff a tiff2, devono essere elencate in tale ordine.

9.2 Proprietà di Sistema

Sebbene sia una pratica comune nel codice C e C++ aggiungere codice specifico della piattaforma all'interno delle direttive `ifdef` del preprocessore, per la massima portabilità questo dovrebbe essere evitato. Il software non dovrebbe essere adattato a piattaforme specifiche con `ifdefs`, ma piuttosto su un sistema canonico costituito da un insieme di funzionalità. La codifica per sistemi specifici rende il software meno portabile, perché i sistemi e le funzionalità che supportano cambiano nel tempo e persino da sistema a sistema. Una funzionalità che potrebbe non aver funzionato su una piattaforma in passato potrebbe essere una funzionalità necessaria in futuro. I seguenti frammenti di codice illustrano la differenza tra la codifica in un sistema canonico e un sistema specifico:

```
// coding to a feature
#ifdef HAS_FOOBAR_CALL
  foobar();
#else
  myfoobar();
#endif

// coding to specific platforms
#if defined(SUN) && defined(HPUX) && !defined(GNUC)
  foobar();
#else
  myfoobar();
#endif
```

Il problema con il secondo approccio è che il codice dovrà essere modificato per ogni nuova piattaforma su cui viene compilato il software. Ad esempio, una versione futura di SUN potrebbe non avere più la chiamata foobar. Utilizzando l'approccio `HAS_FOOBAR_CALL`, il software funzionerà fintanto che `HAS_FOOBAR_CALL` è definito correttamente, ed è qui che CMake può aiutare. CMake può essere utilizzato per definire `HAS_FOOBAR_CALL` correttamente e automaticamente utilizzando i comandi `try_compile` e `try_run`. Questi comandi possono essere utilizzati per compilare ed eseguire piccoli programmi di test durante la fase di configurazione di CMake. I programmi di test verranno inviati al compilatore che verrà utilizzato per buildare il progetto e, se si verificano errori, la funzionalità può essere disabilitata. Questi comandi richiedono la scrittura di un piccolo programma C o C++ per testare la funzionalità. Ad esempio, per verificare se la chiamata `foobar` è presente sul sistema, si prova a compilare un semplice

programma che utilizza `foobar`. Prima si scrive il programmino di test (testNeedFoobar.c in questo esempio) e poi si aggiungono le chiamate CMake al file CMakeLists per provare a compilare quel codice. Se la compilazione funziona allora `HAS_FOOBAR_CALL` sarà impostato su true.

```
// --- testNeedFoobar.c -----

#include <foobar.h>
main()
{
  foobar();
}
```

```
# --- testNeedFoobar.cmake ---

try_compile (HAS_FOOBAR_CALL
  ${CMAKE_BINARY_DIR}
  ${PROJECT_SOURCE_DIR}/testNeedFoobar.c
  )
```

Ora che `HAS_FOOBAR_CALL` è impostato correttamente in CMake, lo si può utilizzare nel codice sorgente tramite il comando `target_compile_definitions`. In alternativa, è possibile configurare un file header. Questo è discusso ulteriormente nella sezione chiamata *Come Configurare un file Header*.

A volte la compilazione di un programma di test non è sufficiente. In alcuni casi, si potrebbe effettivamente voler compilare ed eseguire un programma per ottenerne l'output. Un buon esempio di ciò è testare l'ordine dei byte di una macchina. L'esempio seguente mostra come scrivere un piccolo programma che CMake compilerà ed eseguirà per determinare l'ordine dei byte di una macchina.

```
// ---- TestByteOrder.c ------

int main () {
  /* Are we most significant byte first or last */
  union
  {
    long l;
    char c[sizeof (long)];
  } u;
  u.l = 1;
  exit (u.c[sizeof (long) - 1] == 1);
}
```

```
# ----- TestByteOrder.cmake-----

try_run(RUN_RESULT_VAR
  COMPILE_RESULT_VAR
```

(continues on next page)

(continua dalla pagina precedente)

```
  ${CMAKE_BINARY_DIR}
  ${PROJECT_SOURCE_DIR}/Modules/TestByteOrder.c
  OUTPUT_VARIABLE OUTPUT
  )
```

Il risultato restituito dell'esecuzione andrà in RUN_RESULT_VAR, il risultato della compilazione andrà in COMPILE_RESULT_VAR e qualsiasi output dell'esecuzione andrà in OUTPUT. Si possono utilizzare queste variabili per segnalare informazioni di debug agli utenti del progetto.

Per piccoli programmi di test, il comando file con l'opzione WRITE può essere utilizzato per creare il file sorgente dal file CMakeLists. L'esempio seguente verifica che il compilatore C possa essere eseguito.

```
file(WRITE
  ${CMAKE_BINARY_DIR}/CMakeTmp/testCCompiler.c
  "int main(){return 0;}"
  )

try_compile(CMAKE_C_COMPILER_WORKS
  ${CMAKE_BINARY_DIR}
  ${CMAKE_BINARY_DIR}/CMakeTmp/testCCompiler.c
  OUTPUT_VARIABLE OUTPUT
  )
```

Per operazioni try_compile e try_run più avanzate, potrebbe essere desiderabile passare i flag al compilatore o a CMake. Entrambi i comandi supportano gli argomenti facoltativi CMAKE_FLAGS e COMPILE_DEFINITIONS. CMAKE_FLAGS può essere utilizzato per passare i flag -DVAR:TYPE=VALUE a CMake. Il valore di COMPILE_DEFINITIONS viene passato direttamente alla riga di comando del compilatore.

Sono disponibili diversi moduli try-run e try-compile predefiniti in CMake cmake-modules(7), alcuni dei quali sono elencati di seguito. Questi moduli consentono di eseguire alcuni controlli comuni senza dover creare un file sorgente per ogni test. Molti di questi moduli esamineranno il valore corrente delle variabili CMAKE_REQUIRED_FLAGS e CMAKE_REQUIRED_LIBRARIES per aggiungere ulteriori flag di compilazione o link di librerie al test.

CheckIncludeFile
: Fornisce una macro che verifica la presenza di un file di inclusione su un sistema prendendo due argomenti, il primo è il file di inclusione da cercare e il secondo è la variabile in cui archiviare il risultato. CFlag aggiuntivi possono essere passati come terzo argomento o impostando CMAKE_REQUIRED_FLAGS.

CheckIncludeFileCXX
: Fornisce una macro che verifica la presenza di un file di inclusione in un programma C++ prendendo due argomenti, il primo è il file di inclusione da cercare e il secondo è la variabile in cui memorizzare il risultato. CFlag aggiuntivi possono essere passati come terzo argomento.

CheckIncludeFiles
: Fornisce una macro che verifica se i file header specificati possono essere inclusi insieme. Questa macro utilizza CMAKE_REQUIRED_FLAGS se è impostata ed è utile quando un file header che interessa controllare dipende dall'inclusione precedente di un altro file header.

CheckLibraryExists
: Fornisce una macro che controlla se esiste una libreria prendendo quattro argomenti, il primo dei quali è il nome della libreria da controllare; il secondo è il nome di una funzione che dovrebbe essere in quella libreria; il terzo argomento è la posizione in cui dovrebbe essere trovata la libreria; e il quarto argomento è una variabile in cui memorizzare il risultato. Questa macro utilizza CMAKE_REQUIRED_FLAGS e CMAKE_REQUIRED_LIBRARIES se sono impostate.

CheckSymbolExists
: Fornisce una macro che verifica se un simbolo è definito in un file header prendendo tre argomenti, il primo dei quali è il simbolo da cercare; il secondo argomento è un elenco di file header da provare a includere; e il terzo argomento è dove è memorizzato il risultato. Questa macro utilizza CMAKE_REQUIRED_FLAGS e CMAKE_REQUIRED_LIBRARIES se sono impostate.

CheckTypeSize
: Fornisce una macro che determina la dimensione in byte di un tipo di variabile prendendo due argomenti con il primo argomento che rappresenta il tipo da valutare e il secondo argomento dove viene archiviato il risultato. Sia CMAKE_REQUIRED_FLAGS e CMAKE_REQUIRED_LIBRARIES vengono utilizzati se impostati.

CheckVariableExists
: Fornisce una macro che controlla se esiste una variabile globale prendendo due argomenti, il primo è la variabile da cercare e il secondo è la variabile in cui memorizzare il risultato. Questa macro creerà il prototipo della variabile e poi tenterà di utilizzarla. Se il programma di test viene compilato, la variabile esiste. Funziona solo per le variabili C. Questa macro utilizza CMAKE_REQUIRED_FLAGS e CMAKE_REQUIRED_LIBRARIES se sono impostate.

Si consideri l'esempio seguente che mostra una varietà di questi moduli utilizzati per calcolare le proprietà della piattaforma. All'inizio dell'esempio vengono caricati quattro moduli da CMake. Il resto dell'esempio utilizza le macro definite in quei moduli per testare rispettivamente i file header, le librerie, i simboli e le dimensioni dei tipi.

```
# Include all the necessary files for macros
include(CheckIncludeFiles)
include(CheckLibraryExists)
include(CheckSymbolExists)
include(CheckTypeSize)

# Check for header files
set(INCLUDES "")
check_include_files("${INCLUDES};winsock.h" HAVE_WINSOCK_H)

if(HAVE_WINSOCK_H)
```

(continues on next page)

(continua dalla pagina precedente)

```
  set(INCLUDES ${INCLUDES} winsock.h)
endif()

check_include_files("${INCLUDES};io.h" HAVE_IO_H)
if (HAVE_IO_H)
  set(INCLUDES ${INCLUDES} io.h)
endif()

# Check for all needed libraries
set(LIBS "")
check_library_exists("dl;${LIBS}" dlopen "" HAVE_LIBDL)
if(HAVE_LIBDL)
  set(LIBS ${LIBS} dl)
endif()

check_library_exists("ucb;${LIBS}" gethostname "" HAVE_LIBUCB)
if(HAVE_LIBUCB)
  set(LIBS ${LIBS} ucb)
endif()

# Add the libraries we found to the libraries to use when
# looking for symbols with the check_symbol_exists macro
set(CMAKE_REQUIRED_LIBRARIES ${LIBS})

# Check for some functions that are used
check_symbol_exists(socket "${INCLUDES}" HAVE_SOCKET)
check_symbol_exists(poll "${INCLUDES}" HAVE_POLL)

# Various type sizes
check_type_size(int SIZEOF_INT)
check_type_size(size_t SIZEOF_SIZE_T)
```

9.3 Come Passare i Parametri a una Compilazione

Una volta determinate le caratteristiche del sistema, è il momento di configurare il software in base a quanto trovato. Esistono due modi comuni per passare queste informazioni al compilatore: sulla riga di compilazione o utilizzando header preconfigurato. Il primo modo consiste nel passare le definizioni sulla riga di compilazione. Una definizione del preprocessore può essere passata al compilatore da un file CMakeLists col comando `target_compile_definitions`. Ad esempio, una pratica comune nel codice C è avere la possibilità di compilare selettivamente istruzioni di debug in/out.

```
#ifdef DEBUG_BUILD
  printf("the value of v is %d", v);
#endif
```

Una variabile CMake potrebbe essere utilizzata per attivare o disattivare le build di debug utilizzando il comando `option`:

```
option(DEBUG_BUILD
       "Build with extra debug print messages.")

if(DEBUG_BUILD)
  target_compile_definitions(mytarget PUBLIC DEBUG_BUILD)
endif()
```

Un altro esempio potrebbe essere quello di comunicare al compilatore il risultato del precedente test `HAS_FOOBAR_CALL` discusso in precedenza in questo capitolo. Lo si può fare in questo modo:

```
if (HAS_FOOBAR_CALL)
  target_compile_definitions(mytarget PUBLIC HAS_FOOBAR_CALL)
endif()
```

9.4 Come Configurare un file Header

Il secondo approccio per passare le definizioni al codice sorgente consiste nel configurare un file di header. Il file di intestazione includerà tutte le macro `#define` necessarie per la build del progetto. Per configurare un file con CMake, viene utilizzato il comando `configure_file`. Questo comando richiede un file di input che viene analizzato da CMake per produrre un file di output con tutte le variabili espanse o sostituite. Ci sono tre modi per specificare una variabile in un file di input per `configure_file`.

```
#cmakedefine VARIABLE
```

Se VARIABLE è true, allora il risultato sarà:

```
#define VARIABLE
```

Se VARIABLE è false, allora il risultato sarà:

```
/* #undef VARIABLE */
```

Quando si scrive un file da configurare, si prende in considerazione l'utilizzo di `@VARIABLE@` invece di `${VARIABLE}` per le variabili che dovrebbero essere espanse da CMake. Poiché la sintassi `${}` è comunemente usata da altri linguaggi, gli utenti possono dire al comando `configure_file` di espandere solo le variabili usando la sintassi `@var@` passando l'opzione `@ONLY` al comando; questo è utile se si sta configurando uno script che può contenere strin-

ghe `${var}` da preservare. Questo è importante perché CMake sostituirà tutte le occorrenze di `${var}` con la stringa vuota se `var` non è definita in CMake.

L'esempio seguente configura un file .h per un progetto che contiene variabili del preprocessore. La prima definizione indica se la chiamata `FOOBAR` esiste nella libreria e la successiva contiene il path dell'albero di build.

```
# ---- CMakeLists.txt file-----

# Configure a file from the source tree
# called projectConfigure.h.in and put
# the resulting configured file in the build
# tree and call it projectConfigure.h
configure_file(
   ${PROJECT_SOURCE_DIR}/projectConfigure.h.in
   ${PROJECT_BINARY_DIR}/projectConfigure.h
   @ONLY
   )
```

```
// -----projectConfigure.h.in file------
/* define a variable to tell the code if the */
/* foobar call is available on this system */
#cmakedefine HAS_FOOBAR_CALL

/* define a variable with the path to the */
/* build directory  */
#define PROJECT_BINARY_DIR "@PROJECT_BINARY_DIR@"
```

È importante configurare i file nell'albero binario, non nell'albero dei sorgenti. Un singolo albero di sorgenti può essere condiviso da più alberi di build o di piattaforme. Configurando i file nell'albero binario, le differenze tra build o tra piattaforme verranno mantenute isolate nell'albero della build e non danneggeranno altre build. Ciò significa che si devono includere la directory dell'albero di build in cui è stato configurato il file header nell'elenco delle directory di inclusione del progetto utilizzando il comando `target_include_directories`.

CAPITOLO 10

Ricerca dei Pacchetti

Molti progetti software forniscono strumenti e librerie intesi come elementi costitutivi per altri progetti e applicazioni. I progetti CMake che dipendono da pacchetti esterni individuano le loro dipendenze utilizzando il comando `find_package`. Una tipica invocazione è della forma:

```
find_package(<Package> [version])
```

dove `<Package>` è il nome del pacchetto da trovare e `[version]` è una richiesta di versione facoltativa (nella forma `major[.minor.[patch]]`). La nozione del comando "package" è distinta da quella di CPack, che ha lo scopo di creare distribuzioni e programmi di installazione di sorgenti e binari.

Il comando opera in due modalità: modalità `Module` e modalità `Config`. In modalità Module, il comando cerca un `find module`: un file chiamato `Find<Package>.cmake`. Cerca prima in `CMAKE_MODULE_PATH` e poi nell'installazione di CMake. Se viene trovato un modulo "find", viene caricato per cercare i singoli componenti del pacchetto. I moduli Find contengono una conoscenza specifica del pacchetto delle librerie e di altri file che si aspettano di trovare e usano internamente comandi come `find_library` per individuarli. CMake fornisce moduli di ricerca per molti pacchetti comuni; vedere il manuale `cmake-modules(7)`.

La modalità Config di `find_package` fornisce una potente alternativa attraverso la cooperazione col pacchetto da trovare. Entra in questa modalità dopo aver fallito nell'individuare un modulo "find" o quando esplicitamente richiesto dal chiamante. In modalità Config il comando cerca un `file di configurazione del pacchetto`: un file chiamato `<Package>Config.cmake` o `<package>-config.cmake` fornito dal pacchetto da trovare. Dato il nome di un pacchetto, il comando `find_package` sa come cercare approfonditamente all'interno dei prefissi di installazione per locazioni come:

```
<prefix>/lib/<package>/<package>-config.cmake
```

(consultare la documentazione del comando find_package per un elenco completo delle locazioni). CMake crea una voce della cache chiamata <Package>_DIR per memorizzare la posizione trovata o consentire all'utente di impostarla. Dat che un file di configurazione del pacchetto viene fornito con una sua installazione, sa esattamente dove trovare tutto ciò che viene fornito. Una volta che il comando find_package ha individuato il file, fornisce le posizioni dei componenti del pacchetto senza ulteriori ricerche.

L'opzione [version] chiede a find_package di individuare una particolare versione del pacchetto. In modalità Modulo, il comando passa la richiesta al modulo find. In modalità Config il comando cerca accanto a ciascun file di configurazione del pacchetto candidato un file di versione del pacchetto: un file denominato <Package>ConfigVersion.cmake o <package>-config-<version>.cmake. Il file della versione viene caricato per verificare se la versione del pacchetto è una corrispondenza accettabile per quella richiesta (vedere la documentazione di find_package per la specifica dell'API del file della versione). Se il file della versione dichiara la compatibilità, il file di configurazione viene accettato, altrimenti ignorato. Questo approccio consente a ciascun progetto di definire le proprie regole per la compatibilità delle versioni.

10.1 Moduli Find Nativi

CMake ha molti moduli predefiniti che si trovano nella sottodirectory Modules di CMake. I moduli possono trovare molti pacchetti software comuni. Consultare il manuale cmake-modules(7) per un elenco dettagliato.

Ciascun modulo Find<XX>.cmake definisce un insieme di variabili che consentiranno a un progetto di utilizzare il pacchetto software una volta trovato. Queste variabili iniziano tutte con il nome del software trovato <XX>. Con CMake abbiamo provato a stabilire una convenzione per nominare queste variabili, ma si devono leggere i commenti all'inizio del modulo per una risposta definitiva. Le seguenti variabili sono utilizzate per convenzione quando necessario:

<XX>_INCLUDE_DIRS
: Dove trovare i file header del pacchetto, in genere <XX>.h, ecc.

<XX>_LIBRARIES
: Le librerie a cui linkare per utilizzare <XX>. Queste includono path completi.

<XX>_DEFINITIONS
: Definizioni del preprocessore da utilizzare durante la compilazione del codice che utilizza <XX>.

<XX>_EXECUTABLE
: Dove trovare il tool <XX> che fa parte del pacchetto.

<XX>_<YY>_EXECUTABLE
: Dove trovare il tool <YY> fornito con <XX>.

<XX>_ROOT_DIR
: Dove trovare la directory di base dell'installazione di <XX>. Ciò è utile per pacchetti di grandi dimensioni in cui si desidera fare riferimento a molti file relativi a una directory di base (o root) comune.

<XX>_VERSION_<YY>
: Se true, è stata trovata la versione <YY> del pacchetto. Gli autori dei moduli find dovrebbero assicurarsi che al massimo uno di questi non sia mai vero. Per esempio TCL_VERSION_84.

<XX>_<YY>_FOUND
: Se false, la parte facoltativa <YY> del pacchetto <XX> non è disponibile.

<XX>_FOUND
: Impostato su false o indefinito se non abbiamo trovato o non vogliamo utilizzare <XX>.

Non tutte le variabili sono presenti in ciascuno dei file `FindXX.cmake`. Tuttavia, nella maggior parte dei casi, `<XX>_FOUND` dovrebbe esistere. Se `<XX>` è una libreria, si devono definire anche `<XX>_LIBRARIES` e `<XX>_INCLUDE_DIR`.

I moduli possono essere inclusi in un progetto sia col comando `include` che con `find_package`.

```
find_package(OpenGL)
```

è equivalente a:

```
include(${CMAKE_ROOT}/Modules/FindOpenGL.cmake)
```

e

```
include(FindOpenGL)
```

Se il progetto viene convertito in CMake per il suo sistema di build, `find_package` funzionerà ancora se il pacchetto fornisce un file `<XX>Config.cmake`. Come creare un pacchetto CMake è descritto più avanti in questo capitolo.

10.2 Creazione dei File di Configurazione del Pacchetto CMake

I progetti devono fornire i file di configurazione del pacchetto in modo che le applicazioni esterne possano trovarli. Si consideri un semplice progetto «Gromit» che fornisce un eseguibile per generare il codice sorgente e una libreria a cui il codice generato deve linkarsi. Il file `CMakeLists.txt` potrebbe iniziare con:

```
cmake_minimum_required(VERSION 3.20)
project(Gromit C)
set(version 1.0)

# Create library and executable.
add_library(gromit STATIC gromit.c gromit.h)
add_executable(gromit-gen gromit-gen.c)
```

Per installare Gromit ed esportare i suoi target per l'utilizzo da parte di progetti esterni, si aggiunge il codice:

```
# Install and export the targets.
install(FILES gromit.h DESTINATION include/gromit-${version})
install(TARGETS gromit gromit-gen
        DESTINATION lib/gromit-${version}
        EXPORT gromit-targets)
install(EXPORT gromit-targets
        DESTINATION lib/gromit-${version})
```

Infine, Gromit deve fornire un file di configurazione del pacchetto nel suo albero di installazione in modo che i progetti esterni possano individuarlo con `find_package`:

```
# Create and install package configuration and version files.
configure_file(
   ${Gromit_SOURCE_DIR}/pkg/gromit-config.cmake.in
   ${Gromit_BINARY_DIR}/pkg/gromit-config.cmake @ONLY)

configure_file(
   ${Gromit_SOURCE_DIR}/gromit-config-version.cmake.in
   ${Gromit_BINARY_DIR}/gromit-config-version.cmake @ONLY)

install(FILES ${Gromit_BINARY_DIR}/pkg/gromit-config.cmake
         ${Gromit_BINARY_DIR}/gromit-config-version.cmake
         DESTINATION lib/gromit-${version})
```

Questo codice configura e installa il file di configurazione del pacchetto e un corrispondente file di versione. Il file di input della configurazione del pacchetto `gromit-config.cmake.in` contiene il codice:

```
# Compute installation prefix relative to this file.
get_filename_component(_dir "${CMAKE_CURRENT_LIST_FILE}" PATH)
get_filename_component(_prefix "${_dir}/../.." ABSOLUTE)

# Import the targets.
include("${_prefix}/lib/gromit-@version@/gromit-targets.cmake")

# Report other information.
set(gromit_INCLUDE_DIRS "${_prefix}/include/gromit-@version@")
```

Dopo l'installazione, il file di configurazione del pacchetto configurato `gromit-config.cmake` conosce le posizioni degli altri file installati rispetto a se stesso. Il file della versione corrispondente del pacchetto è configurato dal suo file di input `gromit-config-version.cmake.in`, che contiene codice come:

```
set(PACKAGE_VERSION "@version@")
if(NOT "${PACKAGE_FIND_VERSION}" VERSION_GREATER "@version@")
```

(continues on next page)

(continua dalla pagina precedente)

```
  set(PACKAGE_VERSION_COMPATIBLE 1) # compatible with older
  if("${PACKAGE_FIND_VERSION}" VERSION_EQUAL "@version@")
    set(PACKAGE_VERSION_EXACT 1) # exact match for this version
  endif()
endif()
```

Un'applicazione che utilizza il pacchetto Gromit potrebbe creare un file CMake simile al seguente:

```
cmake_minimum_required(VERSION 3.20)
project(MyProject C)

find_package(gromit 1.0 REQUIRED)
include_directories(${gromit_INCLUDE_DIRS})
# run imported executable
add_custom_command(OUTPUT generated.c
                   COMMAND gromit-gen generated.c)
add_executable(myexe generated.c)
target_link_libraries(myexe gromit) # link to imported library
```

La chiamata a `find_package` individua un'installazione di Gromit o termina con un messaggio di errore se non viene trovato (a causa di `REQUIRED`). Dopo che il comando ha avuto successo, il file di configurazione del pacchetto Gromit `gromit-config.cmake` è stato caricato, quindi i target Gromit sono stati importati e sono state definite variabili come `gromit_INCLUDE_DIRS`.

L'esempio precedente crea un file di configurazione del pacchetto e lo inserisce nell'albero `install`. Si può anche creare un file di configurazione del pacchetto nell'albero `build` per consentire alle applicazioni di utilizzare il progetto senza installazione. Per fare ciò, si estende il file CMake di Gromit con il codice:

```
# Make project usable from build tree.
export(TARGETS gromit gromit-gen FILE gromit-targets.cmake)
configure_file(${Gromit_SOURCE_DIR}/gromit-config.cmake.in
               ${Gromit_BINARY_DIR}/gromit-config.cmake @ONLY)
```

Questa chiamata a `configure_file` utilizza un file di input diverso, `gromit-config.cmake.in`, contenente:

```
# Import the targets.
include("@Gromit_BINARY_DIR@/gromit-targets.cmake")

# Report other information.
set(gromit_INCLUDE_DIRS "@Gromit_SOURCE_DIR@")
```

Il file di configurazione del pacchetto `gromit-config.cmake` posizionato nell'albero di build fornisce a un progetto esterno le stesse informazioni di quelle nell'albero di installazione, ma fa riferimento ai file nell'albero dei sorgenti e di compilazione. Condivide un file di versione del pacchetto identico `gromit-config-version.cmake` che si trova nell'albero di installazione.

10.3 Registro CMake dei Pacchetti

CMake fornisce due posizioni centrali per registrare i pacchetti che sono stati compilati o installati ovunque su un sistema: un *User Package Registry* e un *System Package Registry*. Il comando `find_package` cerca nei due registri dei pacchetti come due dei passi di ricerca specificati nella sua documentazione. I registri sono particolarmente utili per aiutare i progetti a trovare i pacchetti in percorsi di installazione non standard o direttamente negli alberi di build dei pacchetti. Un progetto può popolare il registro utente o di sistema (usando i propri mezzi) per fare riferimento alla sua posizione. In entrambi i casi, il pacchetto dovrebbe memorizzare un file di configurazione del pacchetto nella posizione registrata e facoltativamente un file di versione del pacchetto precedentemente descritto in questo capitolo.

Lo *User Package Registry* è memorizzato in una posizione specifica della piattaforma per utente. Su Windows è memorizzato nel registro di Windows sotto una chiave in `HKEY_CURRENT_USER`. Un `<package>` potrebbe apparire sotto la chiave di registro

```
HKEY_CURRENT_USER\Software\Kitware\CMake\Packages\<package>
```

come valore `REG_SZ` con un nome arbitrario che specifica la directory contenente il file di configurazione del pacchetto. Sulle piattaforme UNIX, il registro del pacchetto utente è memorizzato nella directory home dell'utente in `~/.cmake/packages`. Un `<package>` potrebbe apparire sotto la directory

```
~/.cmake/packages/<package>
```

come file con nome arbitrario il cui contenuto specifica la directory contenente il file di configurazione del pacchetto. Il comando `export(PACKAGE)` può essere utilizzato per registrare un albero di build del progetto nello «user package registry». CMake attualmente non fornisce un'interfaccia per aggiungere alberi di installazione al registro dei pacchetti utente; agli installatori deve essere insegnato manualmente a registrare i propri pacchetti, se lo si desidera.

Il *System Package Registry* è archiviato in una posizione specifica per la piattaforma, a livello di sistema. Su Windows è memorizzato nel registro di Windows sotto una chiave in `HKEY_LOCAL_MACHINE`. Un `<package>` potrebbe apparire sotto la chiave di registro

```
HKEY_LOCAL_MACHINE\Software\Kitware\CMake\Packages\<package>
```

come valore `REG_SZ` con un nome arbitrario che specifica la directory contenente il file di configurazione del pacchetto. Non esiste un registro dei pacchetti di sistema su piattaforme diverse da Windows. CMake non fornisce un'interfaccia da aggiungere al registro dei pacchetti di sistema; agli installatori deve essere insegnato manualmente a registrare i propri pacchetti, se lo si desidera.

Le voci di registro dei pacchetti sono di proprietà individuale delle installazioni del progetto a cui fanno riferimento. Un programma di installazione del pacchetto è responsabile dell'aggiunta della propria voce e il programma di disinstallazione corrispondente è responsabile della sua rimozione. Tuttavia, per mantenere puliti i registri, il comando `find_package` rimuove automaticamente le voci di registro dei pacchetti obsoleti che incontra se dispone di autorizzazioni sufficienti. Una voce è considerata obsoleta se fa riferimento a una directory che non esiste o

che non contiene un file di configurazione del pacchetto corrispondente. Questo è particolarmente utile per le voci di registro dei pacchetti utente create dal comando `export(PACKAGE)` per costruire alberi che non hanno eventi di disinstallazione e vengono semplicemente cancellati dagli sviluppatori.

Le voci di registro dei pacchetti possono avere un nome arbitrario. Una semplice convenzione per denominarli consiste nell'utilizzare gli hash di contenuto, poiché sono deterministici ed è improbabile che collidano. Il comando `export(PACKAGE)` utilizza questo approccio. Il nome di una voce che fa riferimento a una directory specifica è semplicemente l'hash del contenuto del path della directory stessa. Ad esempio, un progetto può creare voci di registro del pacchetto come

```
> reg query HKCU\Software\Kitware\CMake\Packages\MyPackage
HKEY_CURRENT_USER\Software\Kitware\CMake\Packages\MyPackage
 45e7d55f13b87179bb12f907c8de6fc4
                            REG_SZ    c:/Users/Me/Work/lib/cmake/
↪MyPackage
 7b4a9844f681c80ce93190d4e3185db9
                            REG_SZ    c:/Users/Me/Work/MyPackage-build
```

su Windows, oppure

```
$ cat ~/.cmake/packages/MyPackage/7d1fb77e07ce59a81bed093bbee945bd
/home/me/work/lib/cmake/MyPackage
$ cat ~/.cmake/packages/MyPackage/f92c1db873a1937f3100706657c63e07
/home/me/work/MyPackage-build
```

su UNIX. Il comando `find_package(MyPackage)` cercherà i path registrati per i file di configurazione del pacchetto. L'ordine di ricerca tra le voci di registro dei pacchetti per un singolo pacchetto non è specificato. Le posizioni registrate possono contenere file di versione del pacchetto per indicare a `find_package` se una posizione specifica è adatta per la versione richiesta.

CAPITOLO 11

Comandi Custom

Spesso il processo di build di un progetto software va oltre la semplice compilazione di librerie e di eseguibili. In molti casi, potrebbero essere necessarie attività aggiuntive durante o dopo il processo di build. Esempi comuni includono: compilazione della documentazione utilizzando un pacchetto di documentazione; generare un file sorgente eseguendo un altro eseguibile; generare file utilizzando strumenti per i quali CMake non ha regole (come lex e yacc); spostare gli eseguibili risultanti; post-elaborazione dell'eseguibile, ecc. CMake supporta queste attività aggiuntive utilizzando i comandi `add_custom_command` e `add_custom_target` commands. Questo capitolo descriverà come utilizzare comandi e obiettivi personalizzati per eseguire attività complesse che CMake non supporta nativamente.

11.1 Comandi Custom Portatili

Prima di entrare nei dettagli su come utilizzare i comandi personalizzati, discuteremo come affrontare alcuni dei loro problemi di portabilità. I comandi personalizzati in genere comportano l'esecuzione di programmi con file come input o output. Anche un semplice comando, come la copia di un file, può essere complicato da eseguire in modo multipiattaforma. Ad esempio, la copia di un file su UNIX viene eseguita con il comando `cp`, mentre su Windows viene eseguita con il comando `copy`. A peggiorare le cose, spesso i nomi dei file cambieranno su piattaforme diverse. Gli eseguibili su Windows terminano con .exe, mentre su UNIX no. Anche tra le implementazioni UNIX ci sono differenze, ad esempio le estensioni utilizzate per le librerie condivise; .so, .sl, .dylib, ecc.

CMake fornisce tre strumenti principali per gestire queste differenze. La prima è l'opzione `-E` (abbreviazione di execute) per `cmake`. Quando all'eseguibile `cmake` viene passata l'opzione `-E`, agisce come un comando di utilità generico multipiattaforma. Gli argomenti che seguono l'opzione `-E` indicano cosa dovrebbe fare `cmake`. Queste opzioni forniscono un modo indipendente dalla piattaforma per eseguire alcune attività comuni tra cui copiare o rimuovere file, confron-

tare e copiare in modo condizionale, cronometrare, creare collegamenti simbolici e altro. È possibile fare riferimento all'eseguibile `cmake` utilizzando la variabile `CMAKE_COMMAND` nei file CMakeLists, come mostreranno i prossimi esempi.

Naturalmente, CMake non limita l'uso di `cmake -E` in tutti i comandi personalizzati. Si può usare qualsiasi comando, anche se è importante considerare i problemi di portabilità. Una pratica comune è quella di usare `find_program` per trovare un eseguibile (Perl, per esempio), e poi usare quell'eseguibile nei comandi personalizzati.

Il secondo strumento fornito da CMake per affrontare i problemi di portabilità è una serie di variabili che descrivono le caratteristiche della piattaforma. Il manuale `cmake-variables(7)` elenca molte variabili utili per i comandi personalizzati che devono fare riferimento a file con nomi dipendenti dalla piattaforma. Questi includono `CMAKE_EXECUTABLE_SUFFIX`, `CMAKE_SHARED_LIBRARY_PREFIX`, ecc. che descrivono le convenzioni di denominazione dei file.

Infine, CMake supporta `generator expressions` nei comandi custom. Si tratta di espressioni che utilizzano la sintassi speciale `$<...>` e non vengono valutate durante l'elaborazione dei file di input CMake, ma vengono invece posticipate fino alla generazione del sistema di build finale. Pertanto, i valori che li sostituiscono conoscono tutti i dettagli del loro contesto di valutazione, inclusa la configurazione di build corrente e tutte le proprietà di build associate a un target.

11.2 Aggiungere Comandi Custom

Ora considereremo la firma per `add_custom_command`. Nella terminologia Makefile, `add_custom_command` aggiunge una regola a un Makefile. Per chi ha più familiarità con Visual Studio, aggiunge un passaggio di compilazione personalizzato a un file. `add_custom_command` ha due firme principali: una per aggiungere un comando personalizzato a un target e una per aggiungere un comando personalizzato per la build di un file.

Il target è il nome di una target CMake (eseguibile, libreria, o custom) a cui si desidera aggiungere il comando custom. È possibile scegliere quando eseguire il comando custom. Si possono specificare tutti i comandi desiderati per un comando personalizzato. Verranno eseguiti nell'ordine specificato.

Consideriamo ora un semplice comando personalizzato per copiare un eseguibile una volta compilato.

```
# first define the executable target as usual
add_executable(Foo bar.c)

# then add the custom command to copy it
add_custom_command(
  TARGET Foo
  POST_BUILD
  COMMAND ${CMAKE_COMMAND}
  ARGS -E copy $<TARGET_FILE:Foo> /testing_department/files
  )
```

Il primo comando in questo esempio è quello standard per la creazione di un eseguibile da un elenco di sorgenti. In questo caso, viene creato un eseguibile chiamato `Foo` dal file sorgente `bar.c`. Poi c'è l'invocazione di `add_custom_command`. Qui l'obiettivo è semplicemente `Foo` e stiamo aggiungendo un comando post build. Il comando da eseguire è `cmake` il cui path completo è specificato nella variabile `CMAKE_COMMAND`. I suoi argomenti sono `-E copy` e le posizioni di origine e destinazione. In questo caso, copierà l'eseguibile `Foo` da dove è stato compilato nella directory `/testing_department/files`. Notare che il parametro `TARGET` accetta un target CMake (`Foo` in questo esempio), ma gli argomenti specificati nel parametro `COMMAND` normalmente richiedono path completi. In questo caso, passiamo a `cmake -E copy`, il path completo dell'eseguibile referenziato tramite l'espressione del generatore `$<TARGET_FILE:...>`.

11.3 Generare un File

Il secondo uso di `add_custom_command` è quello di aggiungere una regola su come creare un file di output. Qui la regola fornita sostituirà qualsiasi regola corrente per la creazione di quel file. Si tenga presente che l'output di `add_custom_command` deve essere utilizzato da un target nello stesso scope. Come discusso in seguito, il comando `add_custom_target` può essere utilizzato per questo.

11.3.1 Utilizzo di un Eseguibile per Creare un File Sorgente

A volte un progetto software crea un eseguibile che viene poi utilizzato per generare un file sorgente, che viene utilizzato per creare altri eseguibili o librerie. Questo può sembrare un caso strano, ma si verifica abbastanza frequentemente. Un esempio è il processo di build per la libreria TIFF, che crea un eseguibile per generare un file sorgente contenente informazioni specifiche del sistema. Questo file viene poi utilizzato come file sorgente nella creazione della libreria TIFF principale. Un altro esempio è il Visualization Toolkit (VTK), che crea un eseguibile chiamato `vtkWrapTcl` che racchiude le classi C++ nel Tcl. L'eseguibile viene compilato e utilizzato per creare più file sorgenti per il processo di build.

```
#####################################################
# Test using a compiled program to create a file
#####################################################

# add the executable that will create the file
# build creator executable from creator.cxx
add_executable(creator creator.cxx)

# add the custom command to produce created.c
add_custom_command(
  OUTPUT ${PROJECT_BINARY_DIR}/created.c
  DEPENDS creator
  COMMAND creator ${PROJECT_BINARY_DIR}/created.c
  )
```

(continues on next page)

(continua dalla pagina precedente)

```
# add an executable that uses created.c
add_executable(Foo ${PROJECT_BINARY_DIR}/created.c)
```

La prima parte di questo esempio produce l'eseguibile `creator` dal file sorgente `creator.cxx`. Il comando custom poi imposta una regola per produrre il file sorgente `created.c` eseguendo l'eseguibile `creator`. Il comando custom dipende dal target `creator` e scrive il suo risultato nell'albero di output (`PROJECT_BINARY_DIR`). Infine, viene aggiunto un target eseguibile chiamato `Foo`, creato utilizzando il file sorgente `created.c`. CMake creerà tutte le regole richieste nel Makefile (o nel workspace di Visual Studio) in modo che quando si crea il progetto, l'eseguibile `creator` verrà compilato ed eseguito per creare `created.c`, che verrà poi utilizzato per creare l'eseguibile `Foo`.

11.4 Aggiungere un Comando Custom

Nella discussione finora, i target di CMake hanno generalmente fatto riferimento a eseguibili e librerie. CMake supporta una nozione più generale di target, chiamati target custom (personalizzati), utilizzabili ogni volta che si desidera la nozione di target ma senza che il prodotto finale sia una libreria o un eseguibile. Esempi di target custom includono quelli per creare documentazione, eseguire test o aggiornare pagine web. Per aggiungere un target custom, si usa il comando `add_custom_target`.

Il nome specificato sarà il nome assegnato al target. È possibile utilizzare quel nome per creare in modo specifico tale target con Makefiles (make nome) o Visual Studio (click destro sul target e quindi selezionare Build). Se viene specificato l'argomento facoltativo `ALL`, il target verrà incluso nel target `ALL_BUILD` e verrà compilato automaticamente ogni volta che viene eseguita la build di Makefile o di Project. Il comando e gli argomenti sono facoltativi; se specificati, verranno aggiunti al target come comando di post-build. Per i target custom che eseguiranno solo un comando, questo è tutto ciò di cui c'è bisogno. Target custom più complessi possono dipendere da altri file, in questi casi vengono utilizzati gli argomenti `DEPENDS` per elencare da quali file dipende questo target. Prenderemo in considerazione esempi di entrambi i casi. Per prima cosa, esaminiamo un target custom che non ha dipendenze:

```
add_custom_target(FooJAR ALL
  ${JAR} -cvf "\"${PROJECT_BINARY_DIR}/Foo.jar\""
              "\"${PROJECT_SOURCE_DIR}/Java\""
  )
```

Con la definizione di cui sopra, ogni volta che il target `FooJAR` viene compilato, eseguirà l'Archiver di Java (jar) per creare il file `Foo.jar` dalle classi java nella directory `${PROJECT_SOURCE_DIR}/Java`. In sostanza, questo tipo di target custom consente allo sviluppatore di associare un comando a un target in modo che possa essere comodamente richiamato durante il processo di build. Consideriamo ora un esempio più complesso che modella approssimativamente la generazione di file PDF da LaTeX. In questo caso, il target custom dipende da altri file generati (principalmente i file .pdf del prodotto finale)

```
# Add the rule to build the .dvi file from the .tex
# file. This relies on LATEX being set correctly
#
add_custom_command(
  OUTPUT  ${PROJECT_BINARY_DIR}/doc1.dvi
  DEPENDS ${PROJECT_SOURCE_DIR}/doc1.tex
  COMMAND ${LATEX} ${PROJECT_SOURCE_DIR}/doc1.tex
  )

# Add the rule to produce the .pdf file from the .dvi
# file. This relies on DVIPDF being set correctly
#
add_custom_command(
  OUTPUT  ${PROJECT_BINARY_DIR}/doc1.pdf
  DEPENDS ${PROJECT_BINARY_DIR}/doc1.dvi
  COMMAND ${DVIPDF} ${PROJECT_BINARY_DIR}/doc1.dvi
  )

# finally add the custom target that when invoked
# will cause the generation of the pdf file
#
add_custom_target(TDocument ALL
  DEPENDS ${PROJECT_BINARY_DIR}/doc1.pdf
  )
```

Questo esempio utilizza sia `add_custom_command` che `add_custom_target`. Le due invocazioni `add_custom_command` vengono utilizzate per specificare le regole per la produzione di un file .pdf da un file .tex. In questo caso, ci sono due passaggi e due comandi personalizzati [custom]. Prima viene prodotto un file .dvi dal file .tex eseguendo LaTeX, quindi il file .dvi viene processato per produrre il file .pdf desiderato. Infine, viene aggiunto un target personalizzato chiamato TDocument. Il suo comando fa semplicemente una «echo» di ciò che sta facendo, mentre il vero lavoro viene svolto dai due comandi custom. L'argomento `DEPENDS` imposta una dipendenza tra il target custom e i comandi custom. Quando TDocument viene compilato, cercherà prima di vedere se tutte le sue dipendenze sono state compilate. Se qualcuna non viene compilata, richiamerà i comandi personalizzati appropriati per crearli. Questo esempio può essere abbreviato combinando i due comandi in un unico comando custom, come mostrato nell'esempio seguente:

```
# Add the rule to build the .pdf file from the .tex
# file. This relies on LATEX and DVIPDF being set correctly
#
add_custom_command(
  OUTPUT  ${PROJECT_BINARY_DIR}/doc1.pdf
  DEPENDS ${PROJECT_SOURCE_DIR}/doc1.tex
  COMMAND ${LATEX}  ${PROJECT_SOURCE_DIR}/doc1.tex
  COMMAND ${DVIPDF} ${PROJECT_BINARY_DIR}/doc1.dvi
  )
```

(continues on next page)

(continua dalla pagina precedente)

```
# finally add the custom target that when invoked
# will cause the generation of the pdf file

add_custom_target(TDocument ALL
  DEPENDS ${PROJECT_BINARY_DIR}/doc1.pdf
  )
```

Si consideri ora un caso in cui la documentazione sia costituita da più file. L'esempio precedente può essere modificato per gestire molti file utilizzando un elenco di input e un ciclo `foreach`. Per esempio

```
# set the list of documents to process
set(DOCS doc1 doc2 doc3)

# add the custom commands for each document
foreach(DOC ${DOCS})
  add_custom_command(
    OUTPUT  ${PROJECT_BINARY_DIR}/${DOC}.pdf
    DEPENDS ${PROJECT_SOURCE_DIR}/${DOC}.tex
    COMMAND ${LATEX} ${PROJECT_SOURCE_DIR}/${DOC}.tex
    COMMAND ${DVIPDF} ${PROJECT_BINARY_DIR}/${DOC}.dvi
    )

  # build a list of all the results
  list(APPEND DOC_RESULTS ${PROJECT_BINARY_DIR}/${DOC}.pdf)
endforeach()

# finally add the custom target that when invoked
# will cause the generation of the pdf file
add_custom_target(TDocument ALL
  DEPENDS ${DOC_RESULTS}
  )
```

In questo esempio, la build del target custom `TDocument` provocherà la generazione di tutti i file .pdf specificati. Aggiungere un nuovo documento all'elenco è consiste semplicemente nell'aggiungere il suo nome alla variabile `DOCS` all'inizio dell'esempio.

11.5 Specifica delle Dipendenze e degli Output

Quando si utilizzano comandi e obiettivi custom, spesso si specificano le dipendenze. Quando si specifica una dipendenza o l'output di un comando custom, si dovrebbe sempre specificare il path completo. Ad esempio, se il comando produce `foo.h` nell'albero binario, il suo output dovrebbe essere qualcosa come `${PROJECT_BINARY_DIR}/foo.h`. CMake proverà a determinare il path corretto per il file se non è specificato; progetti complessi spesso finiscono per utilizzare file sia nell'albero dei sorgenti che in quello di compilazione, questo può eventualmente portare a errori se non vengono specificati i percorsi completi.

Quando si specifica un target come dipendenza, è possibile tralasciare il path completo e l'estensione eseguibile, facendovi riferimento semplicemente col suo nome. Si consideri la specifica dei target del generatore come una dipendenza di `add_custom_command` nell'esempio precedente in questo capitolo. CMake riconosce `creator` come corrispondente a un target esistente e gestisce correttamente le dipendenze.

11.6 Quando Non c'è una Regola Per Un Output

Ci sono un paio di casi insoliti che possono sorgere quando si utilizzano comandi custom che richiedono ulteriori spiegazioni. Il primo è un caso in cui un comando (o eseguibile) può creare più output e il secondo è quando è possibile utilizzare più comandi per creare un singolo output.

11.6.1 Un Singolo Comando che Produce più Output

In CMake, un comando custom può produrre più output semplicemente elencando più output dopo la parola chiave `OUTPUT`. CMake creerà le regole corrette per il sistema di build in modo che, indipendentemente dall'output richiesto per un target, verranno eseguite le regole giuste. Se l'eseguibile produce alcuni output ma il processo di build ne utilizza solo uno, si possono semplicemente ignorare gli altri output durante la creazione del comando custom. Supponiamo che l'eseguibile produca un file sorgente utilizzato nel processo di build e anche un log di esecuzione che non viene utilizzato. Il comando custom deve specificare il file sorgente come output e ignorare il fatto che viene generato anche un file di log.

Un altro caso in cui si ha un comando con più output è quando il comando è lo stesso ma gli argomenti cambiano. Questo è effettivamente come avere un comando diverso e ogni caso dovrebbe avere il proprio comando custom. Un esempio di ciò è stato l'esempio della documentazione, in cui è stato aggiunto un comando personalizzato per ogni file .tex. Il comando è lo stesso ma gli argomenti passati cambiano ogni volta.

11.6.2 Avere un Output che può Essere Generato da Diversi Comandi

In rari casi, si potrebbero avere più di un comando che utilizzabile per generare un output. La maggior parte dei sistemi di build, come make e Visual Studio, non lo supporta e allo stesso modo nemmeno CMake. Esistono due approcci comuni che possono essere utilizzati per risolvere questo problema. Se si hanno veramente due comandi diversi che producono lo stesso output e nessun altro output significativo, si può semplicemente sceglierne uno e crearne un comando custom.

Nei casi più complessi ci sono più comandi con più output; Per esempio:

```
Command1 produces foo.h and bar.h
Command2 produces widget.h and bar.h
```

Ci sono alcuni approcci che possono essere utilizzati in questo caso. Si potrebbe considerare di combinare entrambi i comandi e tutti e tre gli output in un singolo comando custom, in modo che ogni volta che è richiesto un output, tutti e tre vengano compilati contemporaneamente. Si possono anche creare tre comandi custom, uno per ogni output univoco. Il comando custom per `foo.h` invocherebbe Command1, mentre quello per `widget.h` invocherebbe Command2. Quando si specifica il comando custom per `bar.h`, si può scegliere Command1 o Command2.

CAPITOLO 12

Conversione di Sistemi Esistenti in CMake

Molte persone, la prima cosa che faranno con CMake è convertire un progetto esistente dall'utilizzo di un vecchio sistema di build per utilizzare CMake. Questo può essere un processo abbastanza semplice, ma ci sono alcuni problemi da considerare. Questa sezione affronterà tali problemi e fornirà alcuni suggerimenti per convertire efficacemente un progetto in CMake. Il primo problema da considerare durante la conversione in CMake è la struttura della directory del progetto.

12.1 Struttura della Directory Del Codice Sorgente

La maggior parte dei piccoli progetti avrà il proprio codice sorgente nella directory di livello superiore o in una directory chiamata `src` o `source`. Anche se tutto il codice sorgente si trova in una sottodirectory, consigliamo vivamente di creare un file CMakeLists per la directory di primo livello. Ci sono due ragioni per questo. Innanzitutto, può creare confusione per alcuni dover eseguire CMake nella sottodirectory del progetto, anziché in quella principale. In secondo luogo, si potrebbe voler installare la documentazione o altri file di supporto da altre directory. Avendo un file CMakeLists al top del progetto, si può usare il comando `add_subdirectory` per scendere nella directory della documentazione dove il suo file CMakeLists può installare la documentazione (si può avere un file CMakeLists per una directory di documentazione senza target o codice sorgente).

Per i progetti che hanno il codice sorgente in più directory, ci sono alcune scelte. Un'opzione utilizzata da molti progetti basati su Makefile consiste nell'avere un singolo Makefile nella directory di primo livello che elenca tutti i file sorgente da compilare nelle loro sottodirectory. Per esempio:

```
SOURCES=\
  subdir1/foo.cxx \
```

(continues on next page)

(continua dalla pagina precedente)

```
  subdir1/foo2.cxx \
  subdir2/gah.cxx \
  subdir2/bar.cxx
```

Questo approccio funziona altrettanto bene con CMake utilizzando una sintassi simile:

```
set(SOURCES
  subdir1/foo.cxx
  subdir1/foo2.cxx
  subdir1/gah.cxx
  subdir2/bar.cxx
  )
```

Un'altra possibilità è fare in modo che ciascuna sottodirectory crei una libreria o librerie che possano poi essere linkate agli eseguibili. In questi casi, ciascuna sottodirectory definirà il proprio elenco di file sorgenti e aggiungerà i target appropriati. Una terza opzione è una mix delle prime due; ogni sottodirectory può avere un file CMakeLists che elenca i suoi sorgenti, ma il file CMakeLists di primo livello non utilizzerà il comando `add_subdirectory` per entrare nelle sottodirectory. Ma, il file CMakeLists di primo livello utilizzerà il comando `include` per includere ciascuno dei file CMakeLists della sottodirectory. Ad esempio, un file CMakeLists di primo livello potrebbe includere il codice seguente

```
# collect the files for subdir1
include(subdir1/CMakeLists.txt)
foreach(FILE ${FILES})
  set(subdir1Files ${subdir1Files} subdir1/${FILE})
endforeach()

# collect the files for subdir2
include(subdir2/CMakeLists.txt)
foreach(FILE ${FILES})
  set(subdir2Files ${subdir2Files} subdir2/${FILE})
endforeach()

# add the source files to the executable
add_executable(foo ${subdir1Files} ${subdir2Files})
```

Mentre i file CMakeLists nelle sottodirectory potrebbero avere il seguente aspetto:

```
# list the source files for this directory
set(FILES
  foo1.cxx
  foo2.cxx
  )
```

L'approccio è una scelta personale. Per progetti di grandi dimensioni, avere più librerie condivise può sicuramente migliorare i tempi di compilazione quando vengono apportate modifiche.

Per i progetti più piccoli, gli altri due approcci hanno i loro vantaggi. Il suggerimento è quello di scegliere una strategia e seguirla.

12.2 Directory di Build

Il problema successivo da considerare è dove inserire i file oggetto, le librerie e gli eseguibili risultanti. Esistono alcuni approcci diversi e comunemente usati e alcuni funzionano meglio con CMake rispetto ad altri. La strategia consigliata consiste nell'inserire i file binari in un albero separato che abbia la stessa struttura dell'albero dei sorgenti. Ad esempio, se l'albero dei sorgenti fosse simile al seguente:

```
foo/
  subdir1
  subdir2
```

l'albero binario potrebbe essere simile a:

```
foobin/
  subdir1
  subdir2
```

Per alcuni generatori di Windows, come Visual Studio, i file di compilazione vengono effettivamente conservati in una sottodirectory corrispondente alla configurazione selezionata; per esempio. debug, release, ecc.

Se c'è bisogno di supportare più architetture da un albero di sorgenti, consigliamo vivamente una struttura di directory come la seguente:

```
projectfoo/
  foo/
    subdir1
    subdir2
  foo-linux/
    subdir1
    subdir2
  foo-osx/
    subdir1
    subdir2
  foo-solaris/
    subdir1
    subdir2
```

In questo modo, ogni architettura ha la propria directory di build e non interferirà con nessun'altra architettura. Da tenere presente che non solo i file oggetto sono conservati nelle directory binarie, ma anche tutti i file configurati che vengono tipicamente scritti nell'albero binario. Un'altra struttura ad albero che si ritrova principalmente nei progetti UNIX è quella in cui i file binari per diverse architetture sono conservati in sottodirectory dell'albero dei sorgenti (vedere

di seguito). CMake non funziona bene con questa struttura, quindi consigliamo di passare alla struttura ad albero di build separata mostrata sopra.

```
foo/
  subdir1/
    linux
    solaris
    hpux
  subdir2/
    linux
    solaris
    hpux
```

CMake fornisce tre variabili per controllare dove vengono scritti i target binari. Sono le variabili `CMAKE_RUNTIME_OUTPUT_DIRECTORY`, `CMAKE_LIBRARY_OUTPUT_DIRECTORY` e `CMAKE_ARCHIVE_OUTPUT_DIRECTORY`. Queste variabili vengono utilizzate per inizializzare le proprietà delle librerie e degli eseguibili per controllare dove verranno scritte. L'impostazione di questi consente a un progetto di posizionare tutte le librerie e gli eseguibili in un'unica directory. Per i progetti con molte sottodirectory, questo può essere un vero risparmio di tempo. Un'implementazione tipica è mostrata di seguito:

```
# Setup output directories.
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${PROJECT_BINARY_DIR}/bin)
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${PROJECT_BINARY_DIR}/lib)
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${PROJECT_BINARY_DIR}/lib)
```

In questo esempio, tutti i file binari di «runtime» verranno scritti nella sottodirectory `bin` dell'albero binario del progetto, inclusi i file eseguibili su tutte le piattaforme e le DLL su Windows. Altri binari verranno scritti nella directory `lib`. Questo approccio è molto utile per i progetti che utilizzano librerie condivise [shared] (DLL) perché le raccoglie tutte in una directory. Se gli eseguibili vengono inseriti nella stessa directory, possono trovare più facilmente le librerie condivise richieste quando vengono eseguiti su Windows.

Un'ultima nota sulle strutture delle directory: con CMake, è perfettamente accettabile avere un progetto all'interno di un progetto. Ad esempio, all'interno dell'albero dei sorgenti di Visualization Toolkit c'è una directory che contiene una copia completa della libreria di compressione zlib. Nello scrivere il file CMakeLists per quella libreria, usiamo il comando `project` per creare un progetto chiamato VTKZLIB anche se si trova all'interno dell'albero dei sorgenti e del progetto VTK. Ciò non ha alcun impatto reale su VTK, ma ci consente di creare zlib indipendentemente da VTK senza dover modificare il suo file CMakeLists.

12.3 Comandi CMake Utili Durante la Conversione di Progetti

Esistono alcuni comandi CMake che possono semplificare e velocizzare il lavoro di conversione di un progetto esistente. Il comando `file` con l'argomento `GLOB` consente di impostare rapidamente una variabile contenente un elenco di tutti i file che corrispondono all'espressione glob. Per esempio:

```
# collect up the source files
file(GLOB SRC_FILES "*.cxx")

# create the executable
add_executable(foo ${SRC_FILES})
```

imposterà la variabile `SRC_FILES` su un elenco di tutti i file `.cxx` nella directory corrente dei sorgenti. Creerà quindi un eseguibile utilizzando quei sorgenti. Gli sviluppatori di Windows devono essere consapevoli del fatto che le corrispondenze glob fanno distinzione tra maiuscole e minuscole.

12.4 Conversione dei Makefile UNIX

Se il progetto è attualmente basato su Makefile UNIX standard, la loro conversione in CMake dovrebbe essere abbastanza semplice. In sostanza, per ogni directory nel progetto che ha un Makefile, si creerà un file CMakeLists corrispondente. Il modo in cui si gestiscono più Makefile in una directory dipende dalla loro funzione. Se i Makefile aggiuntivi (o i file di tipo Makefile) sono semplicemente inclusi nel Makefile principale, è possibile creare file di input CMake corrispondenti (.cmake) e includerli nel file CMakeLists principale allo stesso modo. Se i diversi Makefile devono essere richiamati sulla riga di comando per situazioni diverse, si prenda in considerazione la creazione di un file CMakeLists principale che utilizzi una logica per scegliere quale `include` in base a un'opzione CMake.

Spesso i Makefile hanno un elenco di file oggetto da compilare. Questi possono essere convertiti in variabili CMake come segue:

```
OBJS= \
  foo1.o \
  foo2.o \
  foo3.o
```

diventa

```
set(SOURCES
  foo1.c
  foo2.c
  foo3.c
)
```

Mentre i file oggetto sono in genere elencati in un Makefile, in CMake l'attenzione è sui sorgenti. Se sono usate istruzioni condizionali nei Makefile, possono essere convertite in comandi `if` CMake. Poiché CMake gestisce la generazione delle dipendenze, è possibile eliminare la maggior parte delle dipendenze o delle regole per generarle. Se si hanno regole per creare librerie o eseguibili, si sostituiscono coi comandi `add_library` o `add_executable`.

Alcuni sistemi di build UNIX (e codice sorgente) fanno un uso massiccio dell'architettura di sistema per determinare quali file compilare o quali flag utilizzare. Tipicamente queste informazioni sono memorizzate in una variabile Makefile chiamata `ARCH` o `UNAME`. La prima scelta in questi casi è sostituire il codice specifico dell'architettura con un test più generico. Con alcuni pacchetti software, c'è troppo codice specifico dell'architettura perché tale modifica sia ragionevole, oppure si potrebbe voler prendere decisioni basate sull'architettura per altri motivi. In questi casi, si possono utilizzare le variabili `CMAKE_SYSTEM_NAME` e `CMAKE_SYSTEM_VERSION`. Forniscono informazioni abbastanza dettagliate sul sistema operativo e sulla versione del computer host.

12.5 Conversione di Progetti Basati su Autoconf

I progetti basati su Autoconf consistono principalmente di tre parti chiave. La prima è il file configure.in che guida il processo. La seconda è Makefile.in che diventerà il Makefile risultante, e il terzo pezzo sono i restanti file configurati che risultano dall'esecuzione di configure. Nella conversione di un progetto basato su autoconf in CMake, si inizia con i file configure.in e Makefile.in.

Il file Makefile.in può essere convertito nella sintassi CMake come spiegato nella sezione precedente sulla conversione dei Makefile UNIX. Fatto ciò, si converte il file configure.in nella sintassi CMake. La maggior parte delle funzioni (macro) in autoconf hanno comandi corrispondenti in CMake. Di seguito è riportata una breve tabella di alcune delle conversioni di base:

AC_ARG_WITH
: Usa il comando `option`.

AC_CHECK_HEADER
: Usa la macro `check_include_file` dal modulo `CheckIncludeFile`.

AC_MSG_CHECKING
: Usa il comando `message` con l'argomento `STATUS`.

AC_SUBST
: Fatto automaticamente quando si utilizza il comando `configure_file`.

AC_CHECK_LIB
: Usa la macro `check_libary_exists` dal modulo `CheckLibraryExists`.

AC_CONFIG_SUBDIRS
: Usa il comando `add_subdirectory`.

AC_OUTPUT
: Usa il comando `configure_file`.

AC_TRY_COMPILE
: Usa il comando `try_compile`.

Se lo script di configurazione esegue compilazioni di test utilizzando `AC_TRY_COMPILE`, si può utilizzare lo stesso codice per CMake. O lo si inserisce direttamente nel file CMakeLists se è breve, o preferibilmente lo si mette in un file di codice sorgente per il progetto. In genere inseriamo tali file in una sottodirectory CMake per progetti di grandi dimensioni che richiedono tali test.

Dove ci si affida ad autoconf per configurare i file, si può usare il comando `configure_file` di CMake. L'approccio di base è lo stesso e in genere chiamiamo i file di input da configurare con un'estensione `.in` proprio come fa autoconf. Questo comando sostituisce qualsiasi variabile nel file di input indicato come `${VAR}` o `@VAR@` con i relativi valori determinati da CMake. Se una variabile non è definita, verrà sostituita con niente. Facoltativamente, solo le variabili nella forma `@VAR@` verranno sostituite e quelle `${VAR}` verranno ignorate. Questo è utile per configurare i file per i linguaggi che usano `${VAR}` come sintassi per valutare le variabili. Si possono anche definire condizionalmente le variabili usando il preprocessore C usando `#cmakedefine VAR`. Se la variabile è definita allora `configure_file` convertirà `#cmakedefine` in un `#define`; se non è definito, diventerà un `#undef` commentato. Per esempio:

```
/* what byte order is this system */
#cmakedefine CMAKE_WORDS_BIGENDIAN

/* what size is an INT */
#cmakedefine SIZEOF_INT @SIZEOF_INT@
```

12.6 Conversione di Workspace Basati su Windows

Per convertire una solution di Visual Studio in CMake sono necessari alcuni passaggi. Per prima cosa si dovrà creare un file CMakeLists nella parte superiore della directory del codice sorgente. Come sempre, questo file dovrebbe iniziare col comando `cmake_minimum_required` e `project` che definisce il nome del progetto CMake. Questo diventerà il nome della solution di Visual Studio risultante. Quindi, si aggiungono tutti i file sorgente nelle variabili CMake. Per progetti di grandi dimensioni che dispongono di più directory, creare un file CMakeLists in ciascuna directory come descritto nella sezione sulle strutture delle directory dei sorgenti all'inizio di questo capitolo. Si aggiungeranno poi le librerie e gli eseguibili usando `add_library` e `add_executable`. Per default, `add_executable` presuppone che l'eseguibile sia un'applicazione console. L'aggiunta dell'argomento `WIN32` a `add_executable` indica che si tratta di un'applicazione Windows (che utilizza WinMain invece di main).

Ci sono alcune caratteristiche interessanti che Visual Studio supporta e di cui CMake può trarre vantaggio. Uno è il supporto per la navigazione tra le classi. In genere in CMake, solo i file sorgenti vengono aggiunti a un target, non i file header. Se si aggiungono i file header a un target, questi verranno visualizzati nel workspace e si potranno sfogliare come al solito. Visual Studio supporta anche la nozione di gruppi di file. Per default, CMake crea gruppi per file sorgenti e file header. Usando il comando `source_group`, si possono creare i propri gruppi e assegnare loro dei file. Se si hanno passaggi di build personalizzati nel workspace, questi possono essere

aggiunti ai file CMakeLists utilizzando il comando `add_custom_command`. I target custom (Utility Target) in Visual Studio possono essere aggiunti con il comando `add_custom_target`.

CAPITOLO 13

Cross Compilazione Con CMake

Cross-compilare un software significa che questo viene compilato su un sistema, ma è destinato a funzionare su un sistema diverso. Il sistema utilizzato per buildare il software sarà chiamato «build host» e il sistema per il quale il software è costruito sarà chiamato «sistema target» o «piattaforma target». Il sistema target di solito esegue un sistema operativo diverso (o nessuno) e/o viene eseguito su hardware diverso. Un caso d'uso tipico è nello sviluppo di software per dispositivi embedded come switch di rete, telefoni cellulari o unità di controllo per motori. In questi casi, la piattaforma target non dispone o non è in grado di eseguire l'ambiente di sviluppo software richiesto.

La Cross-compilazione è completamente supportata da CMake, dalla quella da Linux a Windows; cross-compilazione per supercomputer, fino alla cross-compilazione per piccoli dispositivi embedded senza sistema operativo (OS).

La cross-compilazione ha diverse conseguenze per CMake:

- CMake non è in grado di rilevare automaticamente la piattaforma target.
- CMake non riesce a trovare librerie e header nelle directory di sistema di default.
- Gli eseguibili creati durante la cross-compilazione non possono essere eseguiti.

Il supporto cross-compilazione non significa che tutti i progetti basati su CMake possono essere compilati magicamente in modo incrociato (alcuni lo sono), ma che CMake separa le informazioni sulla piattaforma di build e sulla piattaforma target e fornisce all'utente meccanismi per risolvere problemi di cross-compilazione senza requisiti aggiuntivi come l'esecuzione di macchine virtuali, ecc.

Per supportare la cross-compilazione per un progetto specifico, è necessario comunicare a CMake la piattaforma target tramite un file toolchain. Potrebbe essere necessario modificare CMakeLists.txt, è consapevole che la piattaforma di build potrebbe avere proprietà diverse rispetto a quella target e deve gestire le istanze in cui un eseguibile compilato tenta di essere eseguito sull'host di build.

13.1 I File Toolchain

Per utilizzare CMake per la cross-compilazione, è necessario creare un file CMake che descriva la piattaforma target, chiamato «file di toolchain», questo file dice a CMake tutto ciò che deve sapere sulla piattaforma target. Ecco un esempio che utilizza il cross-compilatore MinGW per Windows sotto Linux; i contenuti verranno spiegati riga per riga in seguito.

```
# the name of the target operating system
set(CMAKE_SYSTEM_NAME Windows)

# which compilers to use for C and C++
set(CMAKE_C_COMPILER   i586-mingw32msvc-gcc)
set(CMAKE_CXX_COMPILER i586-mingw32msvc-g++)

# where is the target environment located
set(CMAKE_FIND_ROOT_PATH  /usr/i586-mingw32msvc
    /home/alex/mingw-install)

# adjust the default behavior of the FIND_XXX() commands:
# search programs in the host environment
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)

# search headers and libraries in the target environment
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

Supponendo che questo file sia salvato con il nome TC-mingw.cmake nella home directory, si istruisce CMake a utilizzare questo file impostando la variabile `CMAKE_TOOLCHAIN_FILE`:

```
~/src$ cd build
~/src/build$ cmake -DCMAKE_TOOLCHAIN_FILE=~/TC-mingw.cmake ..
...
```

`CMAKE_TOOLCHAIN_FILE` deve essere specificato solo durante l'esecuzione iniziale di CMake; successivamente, i risultati vengono riutilizzati dalla cache di CMake. Non è necessario scrivere un file di toolchain separato per ogni software che si desidera creare. I file toolchain sono per piattaforma target; cioè se si stanno creando diversi pacchetti software per la stessa piattaforma target, si deve solo scrivere un file toolchain che possa essere utilizzato per tutti i pacchetti.

Cosa significano le impostazioni nel file toolchain? Li esamineremo una per una. Poiché CMake non è in grado di indovinare il sistema operativo o l'hardware target, è necessario impostare le seguenti variabili CMake:

`CMAKE_SYSTEM_NAME`
: Questa variabile è obbligatoria; imposta il nome del sistema target, ovvero allo stesso valore che `CMAKE_SYSTEM_NAME` avrebbe se CMake fosse eseguito sul sistema target. Esempi tipici sono «Linux» e «Windows». Viene utilizzato per costruire i nomi dei file della piattaforma come Linux.cmake o Windows-gcc.cmake. Se il target è un sistema embedded senza sistema operativo, si imposta `CMAKE_SYSTEM_NAME` su «Generic». La

preimpostazione di CMAKE_SYSTEM_NAME in questo modo, invece di essere rilevata, fa sì che CMake consideri automaticamente la build una cross-compilazione e la variabile CMake CMAKE_CROSSCOMPILING verrà impostata su TRUE. CMAKE_CROSSCOMPILING è la variabile che dovrebbe essere testata nei file CMake per determinare se la build corrente è una build cross-compilata o meno.

CMAKE_SYSTEM_VERSION
: Imposta la versione del sistema target.

CMAKE_SYSTEM_PROCESSOR
: Questa variabile è facoltativa; imposta il nome del processore o dell'hardware del sistema target. Viene utilizzato in CMake per uno scopo, per caricare il file ${CMAKE_SYSTEM_NAME}-COMPILER_ID-${CMAKE_SYSTEM_PROCESSOR}.cmake. Questo file può essere utilizzato per modificare impostazioni come i flag del compilatore per il target. Si deve impostare questa variabile solo se si sta usando un cross-compilatore in cui ogni target necessita di impostazioni di build speciali. Il valore può essere scelto liberamente, quindi potrebbe essere, ad esempio, i386, IntelPXA255 o MyControlBoardRev42.

Nel codice CMake, le variabili CMAKE_SYSTEM_XXX descrivono sempre la piattaforma target. Lo stesso vale per le variabili brevi WIN32, UNIX, APPLE. Queste variabili possono essere utilizzate per testare le proprietà del target. Se è necessario testare il sistema host di build, esiste un insieme corrispondente di variabili: CMAKE_HOST_SYSTEM, CMAKE_HOST_SYSTEM_NAME, CMAKE_HOST_SYSTEM_VERSION, CMAKE_HOST_SYSTEM_PROCESSOR; e anche le forme abbreviate: CMAKE_HOST_WIN32, CMAKE_HOST_UNIX e CMAKE_HOST_APPLE.

Poiché CMake non può indovinare il sistema target, non può indovinare quale compilatore dovrebbe usare. L'impostazione delle seguenti variabili definisce quali compilatori utilizzare per il sistema target.

CMAKE_C_COMPILER
: Questa specifica l'eseguibile del compilatore C come path completo o solo come nome file. Se viene specificato con il path completo, questo sarà preferito durante la ricerca del compilatore C++ e degli altri tool (binutils, linker, ecc.). Se il compilatore è un cross-compilatore GNU con un nome prefisso (ad esempio «arm-elf-gcc»), CMake lo rileverà e troverà automaticamente il compilatore C++ corrispondente (ad esempio «arm-elf-c++»). Il compilatore può anche essere impostato tramite la variabile d'ambiente CC. L'impostazione di CMAKE_C_COMPILER direttamente in un file di toolchain ha il vantaggio che le informazioni sul sistema target sono completamente contenute in questo file e non dipendono dalle variabili di ambiente.

CMAKE_CXX_COMPILER
: Questa specifica l'eseguibile del compilatore C come path completo o solo come nome file. Viene gestito allo stesso modo di CMAKE_C_COMPILER. Se la toolchain è una toolchain GNU, dovrebbe essere sufficiente impostare solo CMAKE_C_COMPILER; CMake dovrebbe trovare automaticamente il compilatore C++ corrispondente. Come per CMAKE_C_COMPILER, anche per C++ il compilatore può essere impostato tramite la variabile d'ambiente CXX.

13.2 Ricerca di Librerie Esterne, Programmi e Altri File

La maggior parte dei progetti non banali fa uso di librerie o tool esterni. CMake offre i comandi `find_program`, `find_library`, `find_file`, `find_path` e `find_package` per questo scopo. Cercano questi file nel file system in luoghi comuni e restituiscono i risultati. `find_package` è un po" diverso in quanto in realtà non cerca se stesso, ma esegue i moduli `Find<*>.cmake`, che a loro volta chiamano i comandi `find_program`, `find_library`, `find_file` e `find_path`.

Durante la cross-compilazione, questi comandi diventano più complicati. Ad esempio, durante la cross-compilazione su Windows su un sistema Linux, ottenere `/usr/lib/libjpeg.so` come risultato del comando `find_package(JPEG)` sarebbe inutile, poiché questa sarebbe la libreria JPEG per il sistema host e non il sistema target. In alcuni casi, si desidera trovare file destinati alla piattaforma target; in altri casi si vorranno trovare i file per l'host di build. Le seguenti variabili sono progettate per dare la flessibilità di modificare il modo in cui funzionano i tipici comandi di ricerca in CMake, in modo da poter trovare sia l'host di build che i file target, se necessario.

La toolchain verrà fornita con il proprio set di librerie e header per la piattaforma target, che di solito sono installate con un prefisso comune. È una buona idea impostare una directory in cui verrà installato tutto il software creato per la piattaforma target, in modo che i pacchetti software non vengano confusi con le librerie fornite con la toolchain.

Il comando `find_program` viene in genere utilizzato per trovare un programma che verrà eseguito durante la compilazione, quindi questo dovrebbe comunque cercare nel file system host e non nell'ambiente della piattaforma target. `find_library` viene normalmente utilizzato per trovare una libreria che viene poi utilizzata per scopi di collegamento, quindi questo comando dovrebbe cercare solo nell'ambiente target. Per `find_path` e `find_file`, non è così ovvio; in molti casi, vengono utilizzati per cercare header, quindi per default dovrebbero cercare solo nell'ambiente target. Le seguenti variabili CMake possono essere impostate per regolare il comportamento dei comandi find per la cross-compilazione.

`CMAKE_FIND_ROOT_PATH`
: Questo è un elenco delle directory che contengono l'ambiente target. Ognuna delle directory elencate qui verrà anteposta a ciascuna delle directory di ricerca di ogni comando find. Supponendo che l'ambiente target sia installato in `/opt/eldk/ppc_74xx` e che l'installazione per quella piattaforma target vada in `~/install-eldk-ppc74xx`, si imposta `CMAKE_FIND_ROOT_PATH` con queste due directory. Poi `find_library(JPEG_LIB jpeg)` cercherà in `/opt/eldk/ppc_74xx/lib`, `/opt/eldk/ppc_74xx/usr/lib`, `~/install-eldk-ppc74xx/lib`, `~/install-eldk-ppc74xx/usr/lib` e dovrebbe risultare in `/opt/eldk/ppc_74xx/usr/lib/libjpeg.so`.

  Per default, `CMAKE_FIND_ROOT_PATH` è vuota. Se impostata, verranno cercate prima le directory con il prefisso del percorso fornito in `CMAKE_FIND_ROOT_PATH`, poi verranno cercate le versioni senza prefisso delle stesse directory.

  Impostando questa variabile, in pratica si sta aggiungendo un nuovo set di prefissi di ricerca a tutti i comandi di ricerca in CMake, ma per alcuni comandi di ricerca si potrebbe non voler cercare nelle directory target o dell'host. Si può controllare come funziona ogni invocazione del comando find passando una delle tre seguenti opzioni `NO_CMAKE_FIND_ROOT_PATH`, `ONLY_CMAKE_FIND_ROOT_PATH` o

CMAKE_FIND_ROOT_PATH_BOTH quando lo si chiama. Si può anche controllare come funzionano i comandi find usando le seguenti tre variabili.

CMAKE_FIND_ROOT_PATH_MODE_PROGRAM
Questo imposta il comportamento di default per il comando find_program. Può essere impostato su NEVER, ONLY o BOTH. Quando impostato su NEVER, CMAKE_FIND_ROOT_PATH non verrà utilizzato per le chiamate find_program tranne dove è abilitato esplicitamente. Se impostato su ONLY, solo le directory di ricerca con i prefissi provenienti da CMAKE_FIND_ROOT_PATH saranno usate da find_program. Il default è BOTH, il che significa che verranno cercate prima le directory con prefisso e poi quelle senza prefisso.

Nella maggior parte dei casi, find_program viene utilizzato per cercare un eseguibile che verrà poi eseguito, ad es. utilizzando execute_process o add_custom_command. Quindi nella maggior parte dei casi è richiesto un eseguibile dall'host di build, quindi è preferibile impostare CMAKE_FIND_ROOT_PATH_MODE_PROGRAM a NEVER.

CMAKE_FIND_ROOT_PATH_MODE_LIBRARY
Questo è lo stesso di sopra, ma per il comando find_library. Nella maggior parte dei casi questo viene utilizzato per trovare una libreria che verrà poi utilizzata per il link, quindi è necessaria una libreria per il target. Nella maggior parte dei casi, dovrebbe essere impostato su ONLY.

CMAKE_FIND_ROOT_PATH_MODE_INCLUDE
Questo è lo stesso di sopra e utilizzato sia per find_path che per find_file. Nella maggior parte dei casi, viene utilizzato per trovare le directory include, quindi la ricerca deve essere eseguita nell'ambiente target. Nella maggior parte dei casi, dovrebbe essere impostato su ONLY. Se c'è anche bisogno di trovare file nel file system dell'host di build (ad esempio alcuni file di dati che verranno elaborati durante la build); potrebbe essere necessario modificare il comportamento per quelle chiamate a find_path o a find_file utilizzando le opzioni NO_CMAKE_FIND_ROOT_PATH, ONLY_CMAKE_FIND_ROOT_PATH e CMAKE_FIND_ROOT_PATH_BOTH.

Con un file di toolchain impostato come descritto, CMake ora sa come gestire la piattaforma target e il cross-compilatore. Ora dovremmo essere in grado di creare software per la piattaforma target. Per progetti complessi, ci sono più problemi che devono essere risolti.

13.3 Ispezione del Sistema

La maggior parte dei progetti software portabili dispone di una serie di test di ispezione del sistema per determinarne le proprietà (del target). Il modo più semplice per verificare una funzionalità di sistema con CMake è testare le variabili. A tale scopo, CMake fornisce le variabili UNIX, WIN32 e APPLE. Durante la cross-compilazione, queste variabili si applicano alla piattaforma target, per testare la piattaforma host di build ci sono variabili corrispondenti CMAKE_HOST_UNIX, CMAKE_HOST_WIN32 e CMAKE_HOST_APPLE.

Se questa granularità è troppo grossolana, si possono testare le variabili CMAKE_SYSTEM_NAME, CMAKE_SYSTEM, CMAKE_SYSTEM_VERSION e CMAKE_SYSTEM_PROCESSOR, insieme alle loro controparti CMAKE_HOST_SYSTEM_NAME, CMAKE_HOST_SYSTEM,

CMAKE_HOST_SYSTEM_VERSION e CMAKE_HOST_SYSTEM_PROCESSOR, che contengono le stesse informazioni, ma per l'host di build e non per il sistema target.

```
if(CMAKE_SYSTEM MATCHES Windows)
   message(STATUS "Target system is Windows")
endif()

if(CMAKE_HOST_SYSTEM MATCHES Linux)
   message(STATUS "Build host runs Linux")
endif()
```

13.3.1 Utilizzo dei Controlli di Compilazione

In CMake, ci sono macro come check_include_files e check_c_source_runs che vengono utilizzate per testare le proprietà della piattaforma. La maggior parte di queste macro utilizza internamente il comando try_compile o il try_run. Il comando try_compile funziona come previsto durante la cross-compilazione; tenta di compilare il pezzo di codice con la toolchain di cross-compilazione, che darà il risultato atteso.

Tutti i test che utilizzano try_run non funzioneranno poiché gli eseguibili creati normalmente non possono essere eseguiti sull'host di build. In alcuni casi, ciò potrebbe essere possibile (ad esempio utilizzando macchine virtuali, layer di emulazione come Wine o interfacce al target effettivo) poiché CMake non dipende da tali meccanismi. La dipendenza dagli emulatori durante il processo di build introdurrebbe una nuova serie di potenziali problemi; possono avere una visione diversa del file system, utilizzare altre terminazioni di riga, richiedere hardware o software speciali, ecc.

Se try_run viene invocato durante la cross-compilazione, proverà prima a compilare il software, che funzionerà allo stesso modo di quando non si esegue la cross-compilazione. Se ciò riesce, controllerà la variabile CMAKE_CROSSCOMPILING per determinare se l'eseguibile risultante può essere eseguito o meno. In caso contrario, creerà due variabili di cache, che dovranno essere impostate dall'utente o tramite la cache di CMake. Supponiamo che il comando assomigli a questo:

```
try_run(SHARED_LIBRARY_PATH_TYPE
        SHARED_LIBRARY_PATH_INFO_COMPILED
        ${PROJECT_BINARY_DIR}/CMakeTmp
        ${PROJECT_SOURCE_DIR}/CMake/SharedLibraryPathInfo.cxx
        OUTPUT_VARIABLE OUTPUT
        ARGS "LDPATH"
        )
```

In questo esempio, il file sorgente SharedLibraryPathInfo.cxx verrà compilato e, se l'operazione riesce, l'eseguibile risultante dovrebbe essere eseguito. La variabile SHARED_LIBRARY_PATH_INFO_COMPILED verrà impostata sul risultato della compilazione, ovvero TRUE o FALSE. CMake creerà una variabile cache SHARED_LIBRARY_PATH_TYPE e la preimposta su PLEASE_FILL_OUT-FAILED_TO_RUN. Questa variabile deve essere impostata su quello che sarebbe stato il codice di uscita dell'eseguibile se fosse stato eseguito sul target.

Inoltre, CMake creerà una variabile cache SHARED_LIBRARY_PATH_TYPE__TRYRUN_OUTPUT preimpostandola su PLEASE_FILL_OUT-NOTFOUND. Questa variabile deve essere impostata sull'output che l'eseguibile stampa su stdout e stderr se è stato eseguito sul target. Questa variabile viene creata solo se il comando try_run è stato utilizzato con l'argomento RUN_OUTPUT_VARIABLE o con OUTPUT_VARIABLE. Si devono inserire i valori appropriati per queste variabili. Per aiutare con questo, CMake fa del suo meglio per dare informazioni utili. Per fare questo CMake crea un file ${CMAKE_BINARY_DIR}/TryRunResults.cmake, di cui si può vedere un esempio qui:

```
# SHARED_LIBRARY_PATH_TYPE
#   indicates whether the executable would have been able to run
#   on its target platform. If so, set SHARED_LIBRARY_PATH_TYPE
#   to the exit code (in many cases 0 for success), otherwise
#   enter "FAILED_TO_RUN".
# SHARED_LIBRARY_PATH_TYPE__TRYRUN_OUTPUT
#   contains the text the executable would have printed on
#   stdout and stderr. If the executable would not have been
#   able to run, set SHARED_LIBRARY_PATH_TYPE__TRYRUN_OUTPUT
#   empty. Otherwise check if the output is evaluated by the
#   calling CMake code. If so, check what the source file would
#   have printed when called with the given arguments.
# The SHARED_LIBRARY_PATH_INFO_COMPILED variable holds the build
# result for this TRY_RUN().
#
# Source file: ~/src/SharedLibraryPathInfo.cxx
# Executable : ~/build/cmTryCompileExec-SHARED_LIBRARY_PATH_TYPE
# Run arguments:  LDPATH
#    Called from: [1]   ~/src/CMakeLists.cmake

set(SHARED_LIBRARY_PATH_TYPE
    "0"
    CACHE STRING "Result from TRY_RUN" FORCE)

set(SHARED_LIBRARY_PATH_TYPE__TRYRUN_OUTPUT
    ""
    CACHE STRING "Output from TRY_RUN" FORCE)
```

Si possono trovare tutte le variabili che CMake non è stato in grado di determinare, da quale file CMake sono state chiamate, il file sorgente, gli argomenti per l'eseguibile e il path dell'eseguibile. CMake copierà anche gli eseguibili nella directory di build; hanno i nomi cmTryCompileExec-<nome della variabile>, ad es. in questo caso cmTryCompileExec-SHARED_LIBRARY_PATH_TYPE. Si può quindi provare a eseguire questo eseguibile manualmente sulla piattaforma target effettiva e controllare i risultati.

Una volta ottenuti questi risultati, devono essere inseriti nella cache di CMake. Questo può essere fatto usando ccmake o cmake-gui, e modificando le variabili direttamente nella cache. Non è possibile riutilizzare queste modifiche in un'altra directory di build o se CMakeCache.txt viene rimosso.

L'approccio consigliato consiste nell'usare il file TryRunResults.cmake creato da CMake. Lo si dovrebbe copiare in una posizione sicura (ovvero dove non verrà rimosso se la directory di build viene eliminata) e assegnargli un nome utile, ad es. TryRunResults-MyProject-eldk-ppc.cmake. Il contenuto di questo file deve essere modificato in modo che i comandi set impostino le variabili richieste sui valori appropriati per il sistema target. Questo file può quindi essere utilizzato per precaricare la cache di CMake utilizzando l'opzione -C di cmake:

```
src/build/ $ cmake -C ~/TryRunResults-MyProject-eldk-ppc.cmake .
```

Non è necessario utilizzare nuovamente le altre opzioni di CMake poiché ora sono nella cache. In questo modo si può utilizzare MyProjectTryRunResults-eldk-ppc.cmake in più alberi di build e può essere distribuito col progetto in modo che sia più facile per altri utenti cross-compilarlo.

13.4 Esecuzione di Eseguibili Creati nel Progetto

In alcuni casi è necessario che durante una build venga richiamato un eseguibile compilato in precedenza nella stessa build; questo di solito è il caso di generatori di codice e strumenti simili. Questo non funziona durante la cross-compilazione, poiché gli eseguibili sono creati per la piattaforma target e non possono essere eseguiti sull'host di build (senza l'uso di macchine virtuali, layer di compatibilità, emulatori, ecc.). Con CMake, questi programmi vengono creati utilizzando `add_executable`, ed eseguiti con `add_custom_command` o con `add_custom_target`. Le seguenti tre opzioni possono essere utilizzate per supportare questi eseguibili con CMake. La vecchia versione del codice CMake potrebbe essere simile a questa

```
add_executable(mygen gen.c)
get_target_property(mygenLocation mygen LOCATION)
add_custom_command(
  OUTPUT "${CMAKE_CURRENT_BINARY_DIR}/generated.h"
  COMMAND ${mygenLocation}
       -o "${CMAKE_CURRENT_BINARY_DIR}/generated.h" )
```

Ora mostreremo come questo file può essere modificato in modo che funzioni durante la cross-compilazione. L'idea di base è che l'eseguibile viene compilato solo quando si esegue una compilazione nativa per l'host di build e viene poi esportato come target eseguibile in un file di script CMake. Questo file viene quindi incluso durante la cross-compilazione e verrà caricato il target eseguibile di mygen. Verrà creato un target importato con lo stesso nome del target originale. `add_custom_command` riconosce i nomi dei target come eseguibili, quindi per il comando in `add_custom_command`, può essere utilizzato semplicemente il nome del target; non è necessario utilizzare la proprietà LOCATION per ottenere il path dell'eseguibile:

```
if(CMAKE_CROSSCOMPILING)
   find_package(MyGen)
else()
   add_executable(mygen gen.c)
   export(TARGETS mygen FILE
          "${CMAKE_BINARY_DIR}/MyGenConfig.cmake")
```

(continues on next page)

(continua dalla pagina precedente)

```
endif()

add_custom_command(
  OUTPUT "${CMAKE_CURRENT_BINARY_DIR}/generated.h"
  COMMAND mygen -o "${CMAKE_CURRENT_BINARY_DIR}/generated.h"
  )
```

Con CMakeLists.txt modificato in questo modo, il progetto può essere cross-compilato. Innanzitutto, è necessario eseguire una build nativa per creare l'eseguibile mygen necessario. Successivamente, si può iniziare la cross-compilazione. La directory di build della build nativa deve essere assegnata a quella della cross-compilazione come la posizione del pacchetto MyGen, in modo che find_package(MyGen) possa trovarlo:

```
mkdir build-native; cd build-native
cmake ..
make
cd ..
mkdir build-cross; cd build-cross
cmake -DCMAKE_TOOLCHAIN_FILE=MyToolchain.cmake \
      -DMyGen_DIR=~/src/build-native/ ..
make
```

13.5 Cross-Compilare Hello World

Ora iniziamo effettivamente con la cross-compilazione. Il primo passo consiste nell'installare una toolchain di cross-compilazione. Se questo è già installato, si può saltare il paragrafo successivo.

Esistono molti approcci e progetti diversi che si occupano di cross-compilazione per Linux, che vanno dai progetti di software libero che lavorano su PDA basati su Linux ai fornitori commerciali di Linux embedded. La maggior parte di questi progetti ha il proprio modo di buildare e utilizzare la rispettiva toolchain. Ciascuna di queste toolchain può essere utilizzata con CMake; l'unico requisito è che funzioni nel normale file system e non preveda un ambiente «sandboxed», come ad esempio il progetto Scratchbox.

Una toolchain di facile utilizzo con un ambiente target relativamente completo è l'Embedded Linux Development Toolkit (http://www.denx.de/wiki/DULG/ELDK). Supporta ARM, PowerPC e MIPS come piattaforme target. ELDK è scaricabile da ftp://ftp.sunet.se/pub/Linux/distributions/eldk/. Il modo più semplice è quello di scaricare gli ISO, montarli e poi installarli:

```
mkdir mount-iso/
sudo mount -tiso9660 mips-2007-01-21.iso mount-iso/ -o loop
cd mount-iso/
./install -d /home/alex/eldk-mips/
...
```

(continues on next page)

(continua dalla pagina precedente)

```
Preparing...          ########################################### [100
↪%]
   1:appWeb-mips_4KCle ########################################### [100
↪%]
Done
ls /opt/eldk-mips/
bin  eldk_init  etc  mips_4KC  mips_4KCle  usr  var  version
```

ELDK (e altre toolchain) possono essere installate ovunque, nella home directory o in tutto il sistema se ci sono più utenti che ci lavorano. In questo esempio, la toolchain si troverà in /home/alex/eldk-mips/usr/bin/ e l'ambiente target in /home/alex/eldk-mips/mips_4KC/.

Ora che è installata una toolchain per la cross-compilazione incrociata, CMake deve essere configurato per utilizzarla. Come già descritto, questo viene fatto creando un file di toolchain per CMake. In questo esempio, il file della toolchain ha questo aspetto:

```
# the name of the target operating system
set(CMAKE_SYSTEM_NAME Linux)

# which C and C++ compiler to use
set(CMAKE_C_COMPILER /home/alex/eldk-mips/usr/bin/mips_4KC-gcc)
set(CMAKE_CXX_COMPILER
    /home/alex/eldk-mips/usr/bin/mips_4KC-g++)

# location of the target environment
set(CMAKE_FIND_ROOT_PATH /home/alex/eldk-mips/mips_4KC
                         /home/alex/eldk-mips-extra-install )

# adjust the default behavior of the FIND_XXX() commands:
# search for headers and libraries in the target environment,
# search for programs in the host environment
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

I file della toolchain possono trovarsi ovunque, ma è una buona idea metterli in una posizione centralizzata in modo che possano essere riutilizzati in più progetti. Salveremo questo file come ~/Toolchains/Toolchain-eldk-mips4K.cmake. Le variabili sopra menzionate sono impostate qui: `CMAKE_SYSTEM_NAME`, i compilatori C/C++ e `CMAKE_FIND_ROOT_PATH` per specificare dove si trovano le librerie e gli header per l'ambiente target. Anche le modalità di ricerca sono impostate in modo che le librerie e gli header vengano cercati solo nell'ambiente target, mentre i programmi vengono cercati solo nell'ambiente host. Ora cross-compileremo il progetto hello world del Capitolo 2

```
project(Hello)
add_executable(Hello Hello.c)
```

Si esegue CMake, questa volta dicendogli di usare il file toolchain precedente:

```
mkdir Hello-eldk-mips
cd Hello-eldk-mips
cmake -DCMAKE_TOOLCHAIN_FILE=~/Toolchains/Toolchain-eldk-mips4K.cmake .
 ↪.
make VERBOSE=1
```

Questo dovrebbe dare un eseguibile per la piattaforma target. Grazie all'opzione VERBOSE=1, si dovrebbe vedere che viene utilizzato il cross-compilatore. Ora renderemo l'esempio un po" più sofisticato aggiungendo l'ispezione del sistema e le regole di installazione. Eseguiremo la build e installeremo una libreria shared denominata Tools, poi creeremo l'applicazione Hello che si linka alla libreria Tools.

```
include(CheckIncludeFiles)
check_include_files(stdio.h HAVE_STDIO_H)

set(VERSION_MAJOR 2)
set(VERSION_MINOR 6)
set(VERSION_PATCH 0)

configure_file(config.h.in ${CMAKE_BINARY_DIR}/config.h)

add_library(Tools SHARED tools.cxx)
set_target_properties(Tools PROPERTIES
    VERSION ${VERSION_MAJOR}.${VERSION_MINOR}.${VERSION_PATCH}
    SOVERSION ${VERSION_MAJOR})

install(FILES tools.h DESTINATION include)
install(TARGETS Tools DESTINATION lib)
```

Non c'è differenza in un normale CMakeLists.txt; non sono richiesti prerequisiti speciali per la cross-compilazione. CMakeLists.txt verifica che l'header stdio.h sia disponibile e imposta il numero di versione per la libreria Tools. Questi vengono configurati in config.h, che viene poi utilizzato in tools.cxx. Il numero di versione viene utilizzato anche per impostare quello della libreria Tools. La libreria e gli header sono installati rispettivamente in ${CMAKE_INSTALL_PREFIX}/lib e in ${CMAKE_INSTALL_PREFIX}/include. L'esecuzione di CMake dà questo risultato:

```
mkdir build-eldk-mips
cd build-eldk-mips
cmake -DCMAKE_TOOLCHAIN_FILE=~/Toolchains/Toolchain-eldk-mips4K.cmake \
      -DCMAKE_INSTALL_PREFIX=~/eldk-mips-extra-install ..
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /home/alex/eldk-mips/usr/bin/mips_4KC-
 ↪gcc
-- Check for working C compiler:
   /home/alex/eldk-mips/usr/bin/mips_4KC-gcc -- works
```

(continues on next page)

(continua dalla pagina precedente)

```
-- Check size of void*
-- Check size of void* - done
-- Check for working CXX compiler: /home/alex/eldk-mips/usr/bin/mips_
↪4KC-g++
-- Check for working CXX compiler:
   /home/alex/eldk-mips/usr/bin/mips_4KC-g++ -- works
-- Looking for include files HAVE_STDIO_H
-- Looking for include files HAVE_STDIO_H - found
-- Configuring done
-- Generating done
-- Build files have been written to: /home/alex/src/tests/Tools/build-
↪mips
make install
Scanning dependencies of target Tools
[100%] Building CXX object CMakeFiles/Tools.dir/tools.o
Linking CXX shared library libTools.so
[100%] Built target Tools
Install the project...
-- Install configuration: ""
-- Installing /home/alex/eldk-mips-extra-install/include/tools.h
-- Installing /home/alex/eldk-mips-extra-install/lib/libTools.so
```

Come si può vedere nell'output precedente, CMake ha rilevato il compilatore corretto, ha trovato l'header stdio.h per la piattaforma target e ha generato correttamente i Makefile. È stato richiamato il comando make, che ha poi compilato e installato correttamente la libreria nella directory di installazione specificata. Ora possiamo buildare un eseguibile che usa la libreria Tools ed esegue un'ispezione del sistema

```
project(HelloTools)

find_package(ZLIB REQUIRED)

find_library(TOOLS_LIBRARY Tools)
find_path(TOOLS_INCLUDE_DIR tools.h)

if(NOT TOOLS_LIBRARY OR NOT TOOLS_INCLUDE_DIR)
  message FATAL_ERROR "Tools library not found")
endif()

set(CMAKE_INCLUDE_CURRENT_DIR TRUE)
set(CMAKE_INCLUDE_DIRECTORIES_PROJECT_BEFORE TRUE)
include_directories("${TOOLS_INCLUDE_DIR}"
                    "${ZLIB_INCLUDE_DIR}")

add_executable(HelloTools main.cpp)
target_link_libraries(HelloTools ${TOOLS_LIBRARY}
```

(continues on next page)

(continua dalla pagina precedente)

```
                      ${ZLIB_LIBRARIES})
set_target_properties(HelloTools PROPERTIES
                      INSTALL_RPATH_USE_LINK_PATH TRUE)

install(TARGETS HelloTools DESTINATION bin)
```

La build funziona allo stesso modo della libreria; il file toolchain deve essere utilizzato e poi dovrebbe funzionare:

```
cmake -DCMAKE_TOOLCHAIN_FILE=~/Toolchains/Toolchain-eldk-mips4K.cmake \
      -DCMAKE_INSTALL_PREFIX=~/eldk-mips-extra-install ..
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /home/alex/denx-mips/usr/bin/mips_4KC-
 ↪gcc
-- Check for working C compiler:
   /home/alex/denx-mips/usr/bin/mips_4KC-gcc -- works
-- Check size of void*
-- Check size of void* - done
-- Check for working CXX compiler: /home/alex/denx-mips/usr/bin/mips_
 ↪4KC-g++
-- Check for working CXX compiler:
   /home/alex/denx-mips/usr/bin/mips_4KC-g++ -- works
-- Found ZLIB: /home/alex/denx-mips/mips_4KC/usr/lib/libz.so
-- Found Tools library: /home/alex/denx-mips-extra-install/lib/
 ↪libTools.so
-- Configuring done
-- Generating done
-- Build files have been written to:
   /home/alex/src/tests/HelloTools/build-eldk-mips
make
[100%] Building CXX object CMakeFiles/HelloTools.dir/main.o
Linking CXX executable HelloTools
[100%] Built target HelloTools
```

Ovviamente CMake ha trovato lo zlib corretto e anche libTools.so, che era stato installato nel passaggio precedente.

13.6 Cross-Compilazione per un Microcontroller

CMake può essere utilizzato per molto di più della cross-compilazione su target con sistemi operativi, è anche possibile utilizzarlo in fase di sviluppo per dispositivi «deeply-embedded» con piccoli microcontrollori e nessun sistema operativo. Ad esempio, utilizzeremo Small Devices C Compiler (http://sdcc.sourceforge.net), che funziona su Windows, Linux e Mac OS X e supporta microcontrollori a 8 e 16 bit. Per gestire la build, useremo MS NMake sotto Windows. Come prima, il primo passo è scrivere un file di toolchain in modo che CMake conosca la piattaforma target. CMakeLists.txt dovrebbe essere simile al seguente

```
set(CMAKE_SYSTEM_NAME Generic)
set(CMAKE_C_COMPILER "c:/Program Files/SDCC/bin/sdcc.exe")
```

Il nome del sistema per i target che non dispongono di un sistema operativo, «Generic», deve essere utilizzato come `CMAKE_SYSTEM_NAME`. Il file della piattaforma CMake per «Generic» non imposta alcuna funzionalità specifica. Tutto ciò che presuppone è che la piattaforma target non supporti le librerie shared, quindi tutte le proprietà dipenderanno dal compilatore e da `CMAKE_SYSTEM_PROCESSOR`. Il file toolchain sopra non imposta le variabili relative a FIND. Finché nessuno dei comandi find viene utilizzato nei comandi CMake, va bene. In molti progetti per piccoli microcontrollori, sarà così. CMakeLists.txt dovrebbe essere simile al seguente

```
project(Blink C)

add_library(blink blink.c)

add_executable(hello main.c)
target_link_libraries(hello blink)
```

Non ci sono grandi differenze in altri file CMakeLists.txt. Un punto importante è che il linguaggio «C» è abilitato esplicitamente usando il comando `project`. Se ciò non viene fatto, CMake proverà anche ad abilitare il supporto per C++, che fallirà poiché sdcc supporta solo C. L'esecuzione di CMake e la creazione del progetto dovrebbero funzionare come al solito:

```
cmake -G"NMake Makefiles" \
      -DCMAKE_TOOLCHAIN_FILE=c:/Toolchains/Toolchain-sdcc.cmake ..
-- The C compiler identification is SDCC
-- Check for working C compiler: c:/program files/sdcc/bin/sdcc.exe
-- Check for working C compiler: c:/program files/sdcc/bin/sdcc.exe --␣
↪works
-- Check size of void*
-- Check size of void* - done
-- Configuring done
-- Generating done
-- Build files have been written to: C:/src/tests/blink/build

nmake
Microsoft (R) Program Maintenance Utility Version 7.10.3077
```

(continues on next page)

(continua dalla pagina precedente)

```
Copyright (C) Microsoft Corporation.  All rights reserved.

Scanning dependencies of target blink
[ 50%] Building C object CMakeFiles/blink.dir/blink.rel
Linking C static library blink.lib
[ 50%] Built target blink
Scanning dependencies of target hello
[100%] Building C object CMakeFiles/hello.dir/main.rel
Linking C executable hello.ihx
[100%] Built target hello
```

Questo era un semplice esempio usando NMake con sdcc con le impostazioni di default di sdcc. Naturalmente, sono possibili layout di progetto più sofisticati. Per questo tipo di progetto, è anche una buona idea impostare una directory di installazione in cui è possibile installare librerie riutilizzabili, quindi è più facile utilizzarle in più progetti. Normalmente è necessario scegliere la piattaforma target corretta per sdcc; non tutti usano i8051, che è il default per sdcc. Il modo consigliato per farlo è impostando CMAKE_SYSTEM_PROCESSOR.

In questo modo CMake cercherà e caricherà il file della piattaforma Platform/Generic-SDCC-C-${CMAKE_SYSTEM_PROCESSOR}.cmake. Poiché ciò accade, subito prima di caricare Platform/Generic-SDCC-C.cmake, può essere utilizzato per impostare i flag del compilatore e del linker per l'hardware e il progetto target specifici. Pertanto, è necessario un file di toolchain leggermente più complesso

```
get_filename_component(_ownDir
                       "${CMAKE_CURRENT_LIST_FILE}" PATH)
set(CMAKE_MODULE_PATH "${_ownDir}/Modules" ${CMAKE_MODULE_PATH})

set(CMAKE_SYSTEM_NAME Generic)
set(CMAKE_C_COMPILER "c:/Program Files/SDCC/bin/sdcc.exe")
set(CMAKE_SYSTEM_PROCESSOR "Test_DS80C400_Rev_1")

# here is the target environment located
set(CMAKE_FIND_ROOT_PATH  "c:/Program Files/SDCC"
                          "c:/ds80c400-install" )

# adjust the default behavior of the FIND_XXX() commands:
# search for headers and libraries in the target environment
# search for programs in the host environment
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

Questo file di toolchain contiene alcune nuove impostazioni; si tratta anche del file di toolchain più complicato di cui si dovrebbe mai aver bisogno. CMAKE_SYSTEM_PROCESSOR è impostato su Test_DS80C400_Rev_1, un identificatore per l'hardware target specifico. Ciò ha l'effetto che CMake proverà a caricare Platform/Generic-SDCC-C-Test_DS80C400_Rev_1.cmake. Poiché questo file non esiste nella directory del modulo di sistema CMake, la variabile CMa-

ke CMAKE_MODULE_PATH deve essere aggiustata in modo che questo file possa essere trovato. Se questo file della toolchain viene salvato in c:/Toolchains/sdcc-ds400.cmake, il file specifico dell'hardware deve essere salvato in c:/Toolchains/Modules/Platform/. Un esempio di ciò è mostrato di seguito:

```
set(CMAKE_C_FLAGS_INIT "-mds390 --use-accelerator")
set(CMAKE_EXE_LINKER_FLAGS_INIT "")
```

Questo selezionerà DS80C390 come piattaforma target e aggiungerà l'argomento –use-accelerator ai flag di compilazione di default. In questo esempio è stato utilizzato il generatore «NMake Makefiles». Allo stesso modo sono disponibili, ad es., il generatore e.g. «MinGW Makefiles» potrebbe essere utilizzato per un make GNU da MinGW, o un'altra versione Windows di GNU make. È richiesta almeno la versione 3.78 o il generatore «Unix Makefiles» sotto UNIX. Inoltre, è possibile utilizzare qualsiasi generatore di progetti IDE basato su Makefile; per esempio. il generatore Eclipse, CodeBlocks o KDevelop3.

13.7 Cross-Compilazione di un Progetto Complesso - VTK

La Build di un progetto complesso è un processo in più fasi. Complesso in questo caso significa che il progetto utilizza test che eseguono eseguibili e che crea eseguibili utilizzati successivamente nella build per generare codice (o qualcosa di simile). Uno di questi progetti è VTK, che utilizza diversi test try_run e crea diversi generatori di codice. Quando si esegue CMake sul progetto, ogni comando try_run produrrà un messaggio di errore; alla fine ci sarà un file TryRunResults.cmake nella directory di build. È necessario esaminare tutte le voci di questo file e inserire i valori appropriati. Se non si è certi della correttezza del risultato, si può anche provare a eseguire i binari di test sulla piattaforma target reale, dove vengono salvati nella directory binaria.

VTK contiene diversi generatori di codice, uno dei quali è ProcessShader. Questi generatori di codice vengono aggiunti utilizzando add_executable; get_target_property(LOCATION) viene utilizzato per ottenere le posizioni dei file binari risultanti, che vengono poi utilizzati nei comandi add_custom_command o add_custom_target commands. Poiché gli eseguibili cross-compilati non possono essere eseguiti durante la compilazione, le chiamate add_executable sono circondate dai comandi if (NOT CMAKE_CROSSCOMPILING) e gli eseguibili target vengono importati nel progetto col comando add_executable con l'opzione IMPORTED. Queste istruzioni di importazione si trovano nel file VTKCompileToolsConfig.cmake, che non deve essere creato manualmente, ma viene creato da una build nativa di VTK.

Per eseguire la cross-compilazione di VTK, è necessario:

- Installare una toolchain e creare un file toolchain per CMake.
- Eseguire una build di VTK nativamente per l'host di build.
- Eseguire CMake per la piattaforma target.
- Completare TryRunResults.cmake.

- Usa il file `VTKCompileToolsConfig.cmake` dalla build nativa.
- Eseguire la build.

Quindi, per prima cosa, si crea un VTK nativo per l'host di build utilizzando la procedura standard.

```
cd VTK
mkdir build-native; cd build-native
ccmake ..
make
```

Tutte le opzioni richieste devono essere abilitate usando ccmake; per esempio. se c'è bisogno del wrapping di Python per la piattaforma target, si deve abilitare il wrapping Python in `build-native/`. Una volta terminata questa build, ci sarà un file `VTKCompileToolsConfig.cmake` in `build-native/`. Se ciò ha avuto successo, possiamo continuare a cross-compilare il progetto, in questo esempio per un supercomputer IBM BlueGene.

```
cd VTK
mkdir build-bgl-gcc
cd build-bgl-gcc
cmake -DCMAKE_TOOLCHAIN_FILE=~/Toolchains/Toolchain-BlueGeneL-gcc.
↪cmake \
      -DVTKCompileTools_DIR=~/VTK/build-native/ ..
```

Questo finirà con un messaggio di errore per ogni `try_run` e un file `TryRunResults.cmake`, da completare come descritto precedentemente. Si dovrebbe salvare il file in una posizione sicura, altrimenti verrà sovrascritto alla prossima esecuzione di CMake.

```
cp TryRunResults.cmake ../TryRunResults-VTK-BlueGeneL-gcc.cmake
ccmake -C ../TryRunResults-VTK-BlueGeneL-gcc.cmake .
...
make
```

Alla seconda esecuzione di ccmake, tutti gli altri argomenti possono essere saltati perché ora sono nella cache. È possibile indirizzare CMake alla directory di build che contiene un CMakeCache.txt, poi CMake capirà che questa è la directory di build.

13.8 Alcuni Suggerimenti e Trucchi

13.8.1 Gestione dei test try_run

Per semplificare la cross-compilazione del progetto, si cercano di evitare i test `try_run` usando invece altri metodi per testare qualcosa. Se non si possono evitare i test `try_run`, provare a utilizzare solo il codice di uscita dall'esecuzione e non l'output del processo. In questo modo non sarà necessario impostare sia il codice output che le variabili stdout e stderr per il test `try_run` durante la cross-compilazione. Ciò consente di omettere le opzioni `OUTPUT_VARIABLE` o `RUN_OUTPUT_VARIABLE` per `try_run`.

Facendolo, si crea e si completa un file `TryRunResults.cmake` corretto per la piattaforma target, si potrebbe considerare di aggiungere questo file ai sorgenti del progetto, in modo che possa essere riutilizzato da altri. Questi file sono per target, per toolchain.

13.8.2 Problemi relativi alla piattaforma e alla toolchain target

Se la toolchain non è in grado di creare un semplice programma senza argomenti speciali, come ad es. un file di script del linker o un file di layout di memoria, i test inizialmente eseguiti da CMake falliranno. Per farlo funzionare, il modulo CMake CMakeForceCompiler offre le seguenti macro:

```
CMAKE_FORCE_SYSTEM(name version processor),
CMAKE_FORCE_C_COMPILER(compiler compiler_id sizeof_void_p)
CMAKE_FORCE_CXX_COMPILER(compiler compiler_id).
```

Queste macro possono essere utilizzate in un file di toolchain in modo che le variabili richieste vengano preimpostate e i test CMake evitati.

13.8.3 Gestione di RPATH sotto UNIX

Per le build native, CMake crea eseguibili e librerie per default con RPATH. Nell'albero di build, RPATH è impostato in modo che gli eseguibili possano essere eseguiti dall'albero di build; cioè i punti RPATH nell'albero di build. Durante l'installazione del progetto, CMake linka nuovamente gli eseguibili, questa volta con l'RPATH per l'albero di installazione, che è vuoto per default.

Durante la cross-compilazione probabilmente si vorrà impostare la gestione di RPATH in modo diverso. Poiché l'eseguibile non può essere lanciato sull'host di build, ha più senso compilarlo con l'installazione RPATH fin dall'inizio. Esistono diverse variabili CMake e proprietà del target per regolare la gestione di RPATH.

```
set(CMAKE_BUILD_WITH_INSTALL_RPATH TRUE)
set(CMAKE_INSTALL_RPATH "<whatever you need>")
```

Con queste due impostazioni, i target verranno compilati con l'RPATH di installazione anziché con l'RPATH di build, il che evita la necessità di linkarli nuovamente durante l'installazione. Se non c'è bisogno del supporto RPATH nel progetto, non c'è bisogno di impostare `CMAKE_INSTALL_RPATH`; è vuota per default.

L'impostazione di `CMAKE_INSTALL_RPATH_USE_LINK_PATH` a `TRUE` è utile per le build native, poiché raccoglie automaticamente l'RPATH da tutte le librerie a cui si linka un target. Per la cross-compilazione dovrebbe essere lasciato all'impostazione di default `FALSE`, perché sul target l'RPATH generato automaticamente sarà errato nella maggior parte dei casi; probabilmente avrà un layout di file system diverso rispetto all'host di build.

CAPITOLO 14

Il Packaging Con CPack

CPack è uno strumento di packaging software potente, facile da usare e multipiattaforma distribuito con CMake. Utilizza il concetto di generatori di CMake per astrarre la generazione di pacchetti su piattaforme specifiche. Può essere utilizzato con o senza CMake, ma potrebbe dipendere da alcuni software installati sul sistema. Utilizzando un semplice file di configurazione o utilizzando un modulo CMake, l'autore di un progetto può impacchettare un progetto complesso in un semplice programma di installazione. Questo capitolo descriverà come applicare CPack a un progetto CMake.

14.1 Nozioni di base su CPack

Gli utenti del software potrebbero non sempre volere o essere in grado di creare il software per installarlo. Il software potrebbe essere a codice chiuso o la compilazione potrebbe richiedere molto tempo oppure, nel caso di un'applicazione per l'utente finale, gli utenti potrebbero non disporre delle competenze o degli strumenti per creare l'applicazione. In questi casi, è necessario un modo per creare il software su una macchina e quindi spostare l'albero di installazione su un'altra macchina. Il modo più semplice per farlo è utilizzare la variabile d'ambiente DESTDIR per installare il software in una posizione temporanea, poi tarare o comprimere quella directory e spostarla su un'altra macchina. Tuttavia, l'approccio DESTDIR non è all'altezza su Windows, semplicemente perché i path in genere iniziano con una lettera di unità (C:/) e non è possibile aggiungere semplicemente un prefisso a un path completo con un altro e ottenere un nome di path valido. Un altro approccio più potente consiste nell'utilizzare CPack, incluso in CMake.

CPack è un tool incluso con CMake, può essere utilizzato per creare programmi di installazione e pacchetti per progetti. CPack può creare due tipi base di pacchetti, sorgente e binario. CPack funziona più o meno allo stesso modo di CMake per la creazione di software. Non mira a sostituire i tool di packaging nativi, piuttosto fornisce un'unica interfaccia a una varietà di strumenti. Attualmente CPack supporta la creazione di programmi di installazione Windows utilizzando

il programma di installazione NullSoft NSIS o WiX, il tool Mac OS X PackageMaker, OS X Drag and Drop, OS X X11 Drag and Drop, pacchetti Cygwin Setup, pacchetti Debian, RPM, .tar.gz, .sh (file .tar.gz autoestraenti) e file compressi .zip. L'implementazione di CPack funziona in modo simile a CMake. Per ogni tipo di strumento di creazione pacchetti supportato, esiste un generatore CPack scritto in C++ utilizzato per eseguire il tool nativo e creare il pacchetto. Per semplici pacchetti basati su tar, CPack include una versione di libreria di tar e non richiede l'installazione di tar sul sistema. Per molti degli altri programmi di installazione, gli strumenti nativi devono essere presenti affinché CPack funzioni.

Con i pacchetti dei sorgenti, CPack crea una copia dell'albero dei sorgenti e crea un file zip o tar. Per i pacchetti binari, l'uso di CPack è legato ai comandi di installazione che funzionano correttamente per un progetto. Quando si impostano i comandi di installazione, il primo passo consiste nell'assicurarsi che i file vadano nella struttura di directory corretta con le autorizzazioni corrette. Il passaggio successivo consiste nell'assicurarsi che il software sia riposizionabile e possa essere eseguito in un albero installato. Ciò potrebbe richiedere la modifica del software stesso e ci sono molte tecniche per farlo per ambienti diversi che vanno oltre lo scopo di questo libro. Fondamentalmente, gli eseguibili dovrebbero essere in grado di trovare dati o altri file utilizzando path relativi alla posizione in cui sono installati. CPack installa il software in una directory temporanea e copia l'albero di installazione nel formato del tool di packaging nativo. Una volta che i comandi di installazione sono stati aggiunti a un progetto, l'abilitazione di CPack nel caso più semplice viene eseguita includendo il file CPack.cmake nel progetto.

14.1.1 Un Esempio Semplice

Il progetto CPack più semplice sarebbe simile a questo

```
cmake_minimum_required(VERSION 3.20)
project(CoolStuff)
add_executable(coolstuff coolstuff.cxx)
install(TARGETS coolstuff RUNTIME DESTINATION bin)
include(CPack)
```

Nel progetto CoolStuff, un eseguibile viene creato e installato nella directory bin. Quindi il file CPack viene incluso dal progetto. A questo punto il progetto CoolStuff avrà CPack abilitato. Per eseguire CPack per CoolStuff, si deve prima creare il progetto come si farebbe con qualsiasi altro progetto CMake. CPack aggiunge diversi target al progetto generato. Questi target nei Makefile sono package e package_source e PACKAGE in Visual Studio e Xcode. Ad esempio, per creare un pacchetto sorgente e binario per CoolStuff utilizzando un generatore di Makefile, si eseguiranno i seguenti comandi:

```
mkdir build
cd build
cmake ../CoolStuff
make
make package
make package_source
```

Questo creerebbe un file zip di sorgenti chiamato `CoolStuff-0.1.1-Source.zip`, in installer NSIS chiamato `CoolStuff-0.1.1-win32.exe`, e un file zip binario `CoolStuff-0.1.1-win32.zip`. La stessa cosa potrebbe essere fatta usando la riga di comando CPack.

```
cd build
cpack --config CPackConfig.cmake
cpack --config CPackSourceConfig.cmake
```

14.1.2 Cosa Succede Quando Viene Incluso CPack.cmake?

Quando viene eseguito il comando `include(CPack)`, viene incluso il file `CPack.cmake` nel progetto. Per default userà il comando `configure_file` per creare `CPackConfig.cmake` e `CPackSourceConfig.cmake` nell'albero binario del progetto. Questi file contengono una serie di comandi set che impostano le variabili da utilizzare quando CPack viene eseguito durante la fase di impacchettamento. I nomi dei file configurati dal file `CPack.cmake` possono essere personalizzati con queste due variabili; `CPACK_OUTPUT_CONFIG_FILE` il cui valore predefinito è `CPackConfig.cmake` e `CPACK_SOURCE_OUTPUT_CONFIG_FILE` il cui valore predefinito è `CPackSourceConfig.cmake`.

Il sorgente di questi file si trova in `Templates/CPackConfig.cmake.in`. Questo file contiene alcuni commenti e una singola variabile impostata da `CPack.cmake`. Il file contiene questa riga di codice CMake:

```
@_CPACK_OTHER_VARIABLES_@
```

Se il progetto contiene il file `CPackConfig.cmake.in` nel livello superiore dell'albero dei sorgenti, tale file verrà utilizzato al posto di quello nella directory Templates. Se il progetto contiene il file `CPackSourceConfig.cmake.in`, quel file verrà utilizzato per la creazione di `CPackSourceConfig.cmake`.

I file di configurazione creati da `CPack.cmake` conterranno tutte le variabili che iniziano con «CPACK_" nel progetto corrente. Questo viene fatto usando il comando

```
get_cmake_property(res VARIABLES)
```

Il comando precedente ottiene tutte le variabili definite per il progetto CMake corrente. Parte del codice CMake cerca quindi tutte le variabili che iniziano con «CPACK_" e ogni variabile trovata viene configurata nei due file di configurazione come codice CMake. Ad esempio, se avessimo una variabile impostata come questa nel progetto CMake

```
set(CPACK_PACKAGE_NAME "CoolStuff")
```

`CPackConfig.cmake` e `CPackSourceConfig.cmake` avrebbero la stessa cosa al loro interno:

```
set(CPACK_PACKAGE_NAME "CoolStuff")
```

È importante tenere presente che CPack viene eseguito dopo CMake sul progetto. CPack utilizza lo stesso parser di CMake, ma non avrà gli stessi valori di variabile del progetto CMake. Avrà

solo le variabili che iniziano con CPACK_, e queste variabili saranno configurate in un file da CMake. Ciò può causare alcuni errori e confusione se i valori delle variabili utilizzano caratteri di escape. Poiché vengono analizzati due volte dal linguaggio CMake, avranno bisogno del doppio del livello di escape. Ad esempio, se avessimo quanto segue nel progetto CMake:

```
set(CPACK_PACKAGE_VENDOR "Cool \"Company\"")
```

I file CPack risultanti avrebbero questo:

```
set(CPACK_PACKAGE_VENDOR "Cool "Company"")
```

Non sarebbe esattamente quello voluto o che ci si aspetterebbe. In effetti, semplicemente non funzionerebbe. Per aggirare questo problema, ci sono due soluzioni. La prima è aggiungere un ulteriore livello di escape al comando set originale in questo modo:

```
set(CPACK_PACKAGE_VENDOR "Cool \\\"Company\\\"")
```

Ciò risulterebbe nel comando `set` corretto che sarebbe simile a questo:

```
set(CPACK_PACKAGE_VENDOR "Cool \"Company\"")
```

La seconda soluzione al problema dell'escape è utilizzare un file di configurazione del progetto CPack, spiegato nella sezione successiva.

14.1.3 Aggiunta di Opzioni CPack Custom

Per evitare il problema dell'escape, è possibile specificare un file di configurazione CPack specifico per il progetto. Questo file verrà caricato da CPack dopo il caricamento di `CPackConfig.cmake` o di `CPackSourceConfig.cmake` e `CPACK_GENERATOR` verrà impostata sul generatore CPack in funzione. Le variabili impostate in questo file richiedono solo un livello di escape CMake. Questo file può essere configurato o meno e contiene il normale codice CMake. Nell'esempio sopra, si può spostare `CPACK_FOOBAR` in un file `MyCPackOptions.cmake.in` e configurarlo nell'albero di build del progetto. Poi impostare il path del file di configurazione del progetto in questo modo:

```
configure_file ("${PROJECT_SOURCE_DIR}/MyCPackOptions.cmake.in"
                "${PROJECT_BINARY_DIR}/MyCPackOptions.cmake"
                @ONLY)
set (CPACK_PROJECT_CONFIG_FILE
     "${PROJECT_BINARY_DIR}/MyCPackOptions.cmake")
```

Dove `MyCPackOptions.cmake.in` contiene:

```
set(CPACK_PACKAGE_VENDOR "Cool \"Company\"")
```

La variabile `CPACK_PROJECT_CONFIG_FILE` dovrebbe contenere il path completo del file di configurazione CPack per il progetto, come mostrato nell'esempio precedente. Ciò ha l'ulteriore vantaggio che il codice CMake può contenere istruzioni if basate sul valore di `CPACK_GENERATOR`, in modo che i valori specifici del packager possano essere impostati per

un progetto. Ad esempio, il progetto CMake imposta l'icona per il programma di installazione in questo file:

```
set (CPACK_NSIS_MUI_ICON
     "@CMake_SOURCE_DIR@/Utilities/Release\\CMakeLogo.ico")
```

Si noti che il path ha barre ad eccezione dell'ultima parte che ha un carattere di escape come separatore di path. Al momento della stesura di questo libro, NSIS aveva bisogno che l'ultima parte del path avesse una barra in stile Windows. Se non lo si fa, si potrebbe ricevere il seguente errore:

```
File: ".../Release/CMakeLogo.ico" -> no files found.
Usage: File [/nonfatal] [/a] ([/r] [/x filespec [...]]
       filespec [...] | /oname=outfile one_file_only)
```

14.1.4 Opzioni Aggiunte da CPack

Oltre a creare i due file di configurazione, `CPack.cmake` aggiungerà alcune opzioni avanzate al progetto. Le opzioni aggiunte dipendono dall'ambiente e dal sistema operativo su cui è in esecuzione CMake e controllano i pacchetti di default creati da CPack. Queste opzioni sono nel formato `CPACK_<CPack Generator Name>`, dove i nomi dei generatori disponibili su ogni piattaforma possono essere trovati nella seguente tabella:

Windows	Cygwin	Linux/UNIX	Mac OS X
NSIS	CYGWIN_BINARY	DEB	PACKAGEMAKER
ZIP	SOURCE_CYGWIN	RPM	DRAGNDROP
SOURCE_ZIP		STGZ	BUNDLE
		TBZ2	OSXX11
		TGZ	
		TZ	
		SOURCE_TGZ	
		SOURCE_TZ	

L'attivazione o la disattivazione di queste opzioni influisce sui pacchetti creati durante l'esecuzione di CPack senza opzioni. Se l'opzione è disattivata nel file CMakeCache.txt per il progetto, si può comunque creare quel tipo di pacchetto specificando l'opzione -G nella riga di comando CPack.

14.2 Package dei Sorgenti CPack

I pacchetti di sorgenti in CPack copiano semplicemente l'intero albero dei sorgenti per un progetto in un file e non vengono utilizzate regole di installazione come nel caso dei pacchetti binari. Si dovrebbero effettuare build al di fuori dei sorgenti per evitare che materiale binario extra inquini il pacchetto sorgente. Se ci sono file o directory nell'albero dei sorgenti che non sono desiderati nel pacchetto dei sorgenti, si può usare la variabile CPACK_SOURCE_IGNORE_FILES per escludere delle cose dal pacchetto. Questa variabile contiene un elenco di espressioni regolari. Qualsiasi file o directory che corrisponda a un'espressione regolare in questo elenco verrà escluso dalle fonti. L'impostazione di default è la seguente:

```
"/CVS/;/\\.svn/;\\.swp$;\\.#;/#"
```

Ci sono molti livelli di escape utilizzati nel valore di default poiché questa variabile viene analizzata da CMake una volta e da CPack di nuovo. È importante rendersi conto che l'albero dei sorgenti non utilizzerà alcun comando di installazione, copierà semplicemente l'intero albero dei sorgenti meno i file che gli vengono detti di ignorare nel pacchetto. Per evitare i molteplici livelli di escape, il file a cui fa riferimento CPACK_PROJECT_CONFIG_FILE dovrebbe essere utilizzato per impostare questa variabile. L'espressione è un'espressione regolare e non un'istruzione con caratteri jolly, vedere il Capitolo 4 per ulteriori informazioni sulle espressioni regolari di CMake.

14.3 Comandi CPack dell'Installer

Poiché i pacchetti binari richiedono che CPack interagisca con le regole di installazione del progetto da impacchettare, questa sezione coprirà alcune delle opzioni fornite da CPack per interagire con le regole di installazione di un progetto. CPack può funzionare con gli script di installazione di CMake o con comandi di installazione esterni.

14.3.1 Comandi Install CPack e CMake

Nella maggior parte dei progetti CMake, l'utilizzo delle regole di installazione di CMake sarà sufficiente per creare il pacchetto desiderato. Per default CPack eseguirà la regola di installazione per il progetto corrente. Tuttavia, con un progetto più complicato, si possono specificare sottoprogetti e directory di installazione con la variabile CPACK_INSTALL_CMAKE_PROJECTS. Questa variabile dovrebbe contenere quadruple di directory di installazione, nome del progetto di installazione, la componente dell'installazione e le sottodirectory di installazione. Ad esempio, per un progetto con un sottoprogetto chiamato MySub compilato in una directory chiamata SubProject e per installare tutti i suoi componenti, si avrebbe:

```
SET(CPACK_INSTALL_CMAKE_PROJECTS  "SubProject;MySub;ALL;/")
```

14.3.2 CPack e DESTDIR

Per default CPack non usa l'opzione DESTDIR durante la fase di installazione. Imposta invece CMAKE_INSTALL_PREFIX sul path completo della directory temporanea utilizzata da CPack per preparare il pacchetto di installazione. Questo si può cambiare impostando CPACK_SET_DESTDIR su on. Se l'opzione CPACK_SET_DESTDIR è on, CPack utilizzerà il valore della cache del progetto per CPACK_INSTALL_PREFIX e imposta DESTDIR sul valore temporaneo dell'area di staging. Ciò consente l'installazione di path assoluti nella directory temporanea. I path relativi vengono installati in DESTDIR/${project's CMAKE_INSTALL_PREFIX} dove DESTDIR è impostata sull'area di staging temporanea.

Come notato in precedenza, l'approccio DESTDIR non funziona quando le regole di installazione fanno riferimento ai file tramite path completi di Windows che iniziano con lettere di unità (C:/).

Quando si esegue un'installazione diversa da DESTDIR per il pacchetto, che è il default, tutti i path assoluti vengono installati nelle directory assolute e non nel pacchetto. Pertanto, i progetti che non utilizzano l'opzione DESTDIR non devono utilizzare path assoluti nelle regole di installazione. Al contrario, i progetti che utilizzano path assoluti devono utilizzare l'opzione DESTDIR.

Un'altra variabile può essere utilizzata per controllare il path di root in cui sono installati i progetti, la CPACK_PACKAGING_INSTALL_PREFIX. Per default molti dei generatori installano nella directory /usr. Tale variabile può essere utilizzata per cambiarla in qualsiasi directory, incluso solo /.

14.3.3 CPack e altre directory installate

È possibile eseguire altre regole di installazione se il progetto non è basato su CMake. Questo può essere fatto usando le variabili CPACK_INSTALL_COMMANDS e CPACK_INSTALLED_DIRECTORIES. CPACK_INSTALL_COMMANDS sono comandi che verranno eseguiti durante la fase di installazione del pacchetto. CPACK_INSTALLED_DIRECTORIES dovrebbe contenere coppie di directory e sottodirectory. La sottodirectory può essere "." da installare nella directory di primo livello dell'installazione. I file in ciascuna directory verranno copiati nella sottodirectory corrispondente della directory di staging di CPack e impacchettati con il resto dei file.

14.4 CPack per Windows Installer NSIS

Per creare programmi di installazione basati su procedure guidate in stile Windows, CPack utilizza NSIS (NullSoft Scriptable Install System). Ulteriori informazioni su NSIS sono disponibili nella home page di NSIS: http://nsis.sourceforge.net/ NSIS è un potente strumento con un linguaggio di scripting utilizzato per creare programmi di installazione professionali di Windows. Per creare programmi di installazione di Windows con CPack, c'è bisogno che NSIS sia installato sul computer.

CPack utilizza file template configurati per controllare NSIS. Ci sono due file configurati da CPack durante la creazione di un programma di installazione NSIS. Entrambi i file si trovano

nella directory Modules di CMake. `Modules/NSIS.template.in` è il template per lo script NSIS e `Modules/NSIS.InstallOptions.ini.in` è il template per la moderna interfaccia utente o MUI utilizzata da NSIS. Il file delle opzioni di installazione contiene le informazioni sulle pagine utilizzate nella procedura guidata (wizard) di installazione. Questa sezione descriverà come configurare CPack per creare una procedura guidata di installazione NSIS.

14.4.1 Variabili CPack Usate da CMake per NSIS

Questa sezione contiene schermate acquisite dalla procedura guidata di installazione di CMake NSIS. Per ogni parte dell'installer modificabile o controllabile da CPack, vengono fornite le variabili e i valori utilizzati.

La prima cosa che un utente vedrà del programma di installazione in Windows è l'icona dell'eseguibile del programma di installazione stesso. Per default l'installer avrà l'icona Null Soft Installer, come mostrato nella Figura 1 per il programma di installazione CMake 20071023. Questa icona può essere modificata impostando la variabile `CPACK_NSIS_MUI_ICON`. L'installer per 20071025 nella stessa figura mostra l'icona CMake utilizzata per l'installer.staller.

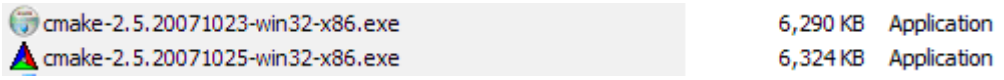


Fig. 1: Icona per l'installer in Windows Explorer

L'ultima cosa che gli utenti vedranno del programma di installazione in Windows è l'icona dell'eseguibile di disinstallazione, come mostrato in Figura 2. Questa opzione può essere impostata con la variabile `CPACK_NSIS_MUI_UNIICON`. Entrambe le icone di installazione e disinstallazione devono avere le stesse dimensioni e lo stesso formato, un file Windows .ico valido utilizzabile da Windows Explorer. Le icone vengono impostate in questo modo:

```
# set the install/uninstall icon used for the installer itself
set (CPACK_NSIS_MUI_ICON
     "${CMake_SOURCE_DIR}/Utilities/Release\\CMakeLogo.ico")
set (CPACK_NSIS_MUI_UNIICON
     "${CMake_SOURCE_DIR}/Utilities/Release\\CMakeLogo.ico")
```

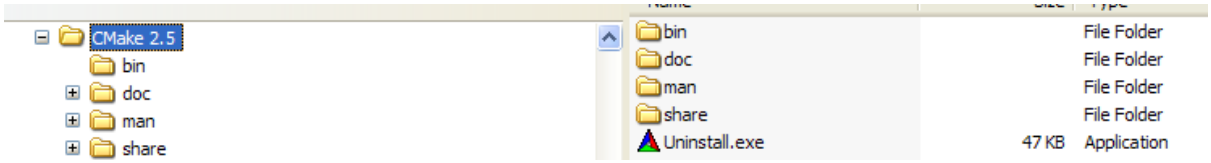


Fig. 2: Icona Uninstall per l'installer NSIS

Su Windows, i programmi possono anche essere rimossi utilizzando il tool «Add or Remove Programs» (Aggiungi o rimuovi programmi) dal pannello di controllo, come mostrato in Figura 3. L'icona per questo dovrebbe essere inclusa in uno degli eseguibili installati. Questo può essere impostato in questo modo:

```
# set the add/remove programs icon using an installed executable
SET(CPACK_NSIS_INSTALLED_ICON_NAME "bin\\cmake-gui.exe")
```

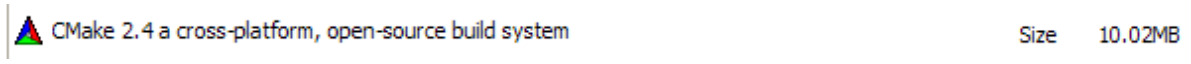


Fig. 3: La Voce «Add or Remove Programs»

Fig. 4: Prima Schermata dell'Installazione Guidata

Quando si esegue il programma di installazione, la prima schermata della procedura guidata sarà simile alla Figura 4. In questa schermata si può controllare il nome del progetto che appare in due punti sullo schermo. Il nome utilizzato per il progetto è controllato dalla variabile CPACK_PACKAGE_INSTALL_DIRECTORY o da CPACK_NSIS_PACKAGE_NAME. In questo esempio, è stato impostato su «CMake 2.5» in questo modo:

```
set (CPACK_PACKAGE_INSTALL_DIRECTORY "CMake
     ${CMake_VERSION_MAJOR}.${CMake_VERSION_MINOR}")
```

o in questo:

```
set (CPACK_NSIS_PACKAGE_NAME "CMake
     ${CMake_VERSION_MAJOR}.${CMake_VERSION_MINOR}")
```

La seconda pagina della procedura guidata di installazione può essere visualizzata in Figura 5. Questa schermata contiene il contratto di licenza e ci sono diverse cose che possono essere configurate. La bitmap del banner a sinistra dell'etichetta «License Agreement» (Contratto di licenza) è controllata dalla variabile CPACK_PACKAGE_ICON in questo modo:

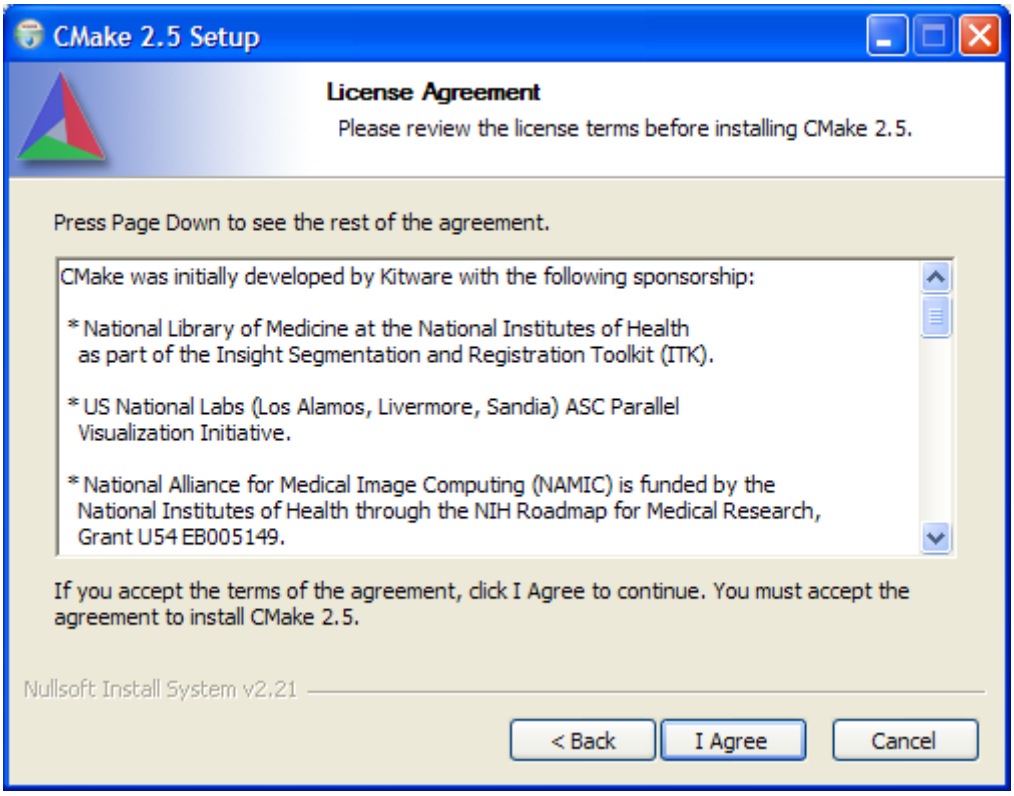


Fig. 5: Seconda Schermata dell'Installazione Guidata

```
set (CPACK_PACKAGE_ICON
     "${CMake_SOURCE_DIR}/Utilities/Release\\CMakeInstall.bmp")
```

CPACK_PACKAGE_INSTALL_DIRECTORY viene nuovamente utilizzata in questa pagina ovunque si veda il testo «CMake 2.5». Il testo del contratto di licenza è impostato sul contenuto del file specificato nella variabile CPACK_RESOURCE_FILE_LICENSE variable. CMake fa quanto segue:

```
set (CPACK_RESOURCE_FILE_LICENSE
     "${CMAKE_CURRENT_SOURCE_DIR}/Copyright.txt")
```

La terza pagina del programma di installazione è visibile nella Figura 6. Questa pagina verrà visualizzata solo se CPACK_NSIS_MODIFY_PATH è impostata su on. Se si seleziona il pulsante Create «name» Desktop Icon e si inseriscono i nomi degli eseguibili nella variabile CPACK_CREATE_DESKTOP_LINKS, verrà creata un'icona sul desktop per quegli eseguibili. Ad esempio, per creare un'icona sul desktop per il programma cmake-gui di CMake, si esegue quanto segue:

```
set (CPACK_CREATE_DESKTOP_LINKS cmake-gui)
```

È possibile creare più collegamenti sul desktop se l'applicazione contiene più di un eseguibile. Il collegamento verrà creato alla voce del menù Start, quindi CPACK_PACKAGE_EXECUTABLES, descritto più avanti in questa sezione, deve contenere anche l'applicazione per poter creare un collegamento sul desktop.

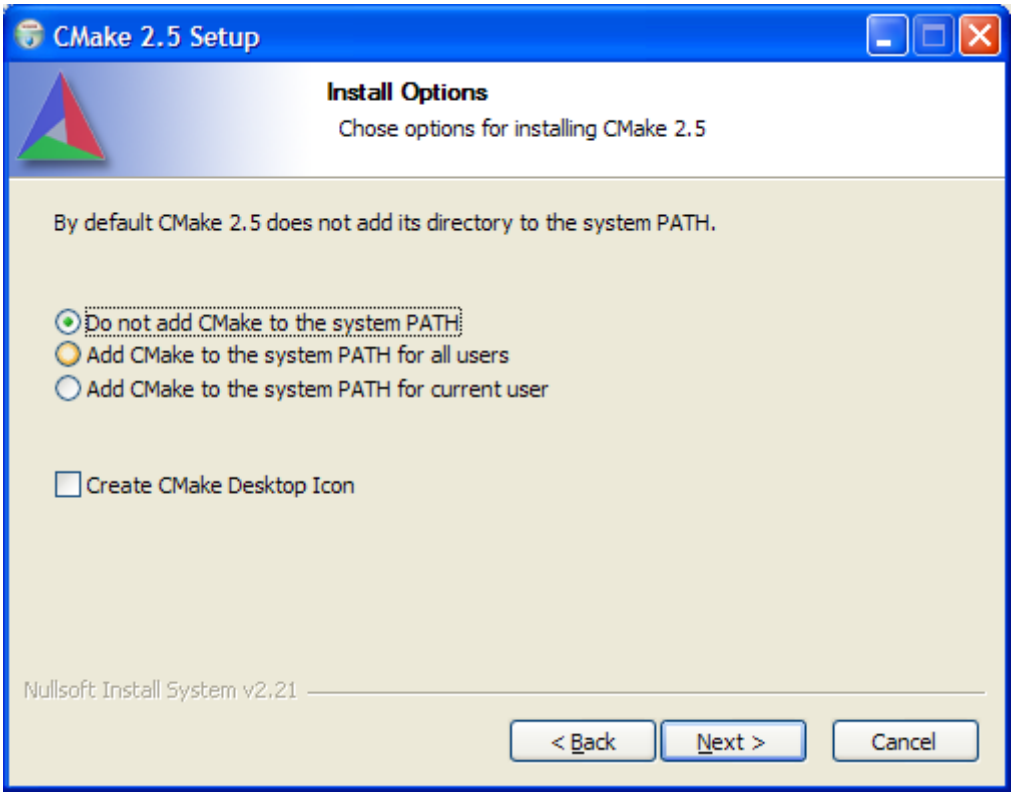


Fig. 6: Terza pagina della procedura guidata di installazione

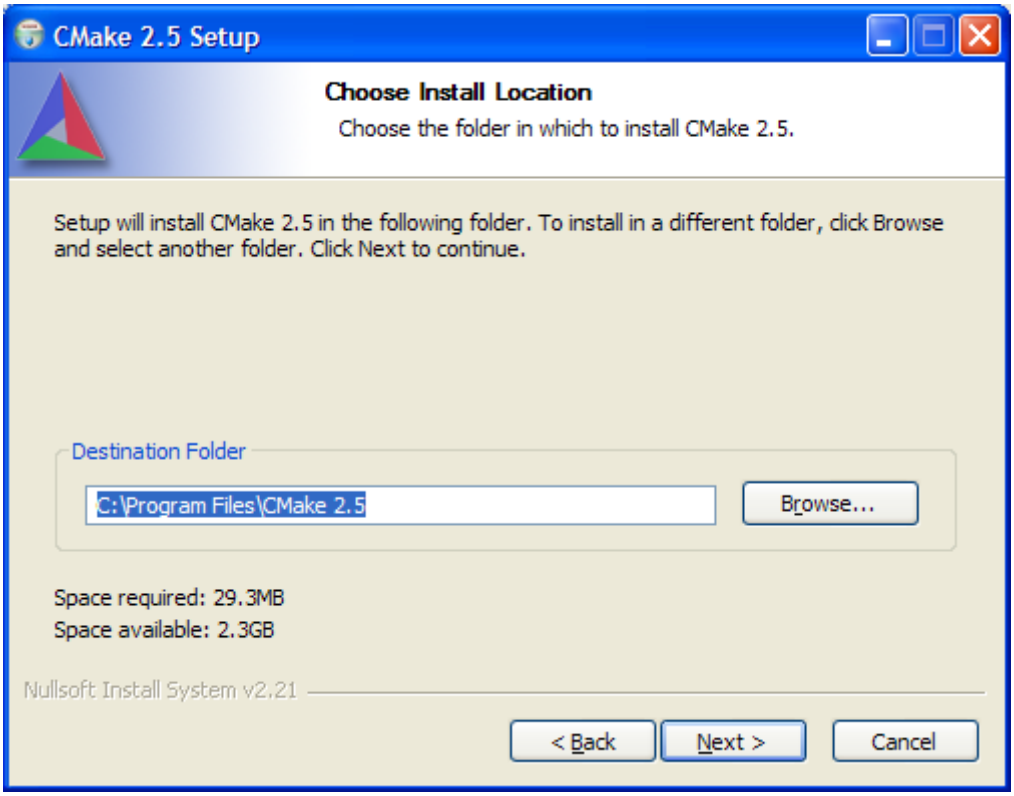


Fig. 7: Quarta pagina della procedura guidata di installazione

La quarta pagina del programma di installazione visualizzata in Figura 7 utilizza la variabile CPACK_PACKAGE_INSTALL_DIRECTORY per specificare la cartella di destinazione di default in Program Files. Il seguente codice CMake è stato utilizzato per impostare tale default:

```
set (CPACK_PACKAGE_INSTALL_DIRECTORY "CMake
     ${CMake_VERSION_MAJOR}.${CMake_VERSION_MINOR}")
```

Le restanti pagine della procedura guidata di installazione non utilizzano variabili CPack aggiuntive e non sono incluse in questa sezione. Un'altra opzione importante che può essere impostata dal generatore NSIS CPack è la chiave di registro utilizzata. Esistono diverse variabili CPack che controllano la chiave di default utilizzata. La chiave è definita nel file NSIS.template.in questo modo:

```
!define MUI_STARTMENUPAGE_REGISTRY_KEY
    "Software\@CPACK_PACKAGE_VENDOR@\@CPACK_PACKAGE_INSTALL_REGISTRY_
↪KEY@"
```

Dove il default di CPACK_PACKAGE_VENDOR è Humanity e quello di CPACK_PACKAGE_INSTALL_REGISTRY_KEY è ${CPACK_PACKAGE_NAME} ${CPACK_PACKAGE_VERSION}

Quindi per CMake 2.5.20071025 la chiave di registro sarebbe simile a questa:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Kitware\CMake 2.5.20071025
```

14.4.2 Creazione di scorciatoie di Windows nel Menù Start

Esistono due variabili che controllano le scorciatoie create nel menù Start di Windows da NSIS. Le variabili contengono elenchi di coppie e devono avere un numero pari di elementi per funzionare correttamente. Il primo è CPACK_PACKAGE_EXECUTABLES, dovrebbe contenere il nome del file eseguibile seguito dal nome del testo del collegamento. Ad esempio, nel caso di CMake, l'eseguibile si chiama cmake-gui, ma il collegamento si chiama «CMake». CMake fa quanto segue per creare quel collegamento:

```
set (CPACK_PACKAGE_EXECUTABLES "cmake-gui" "CMake" )
```

Il secondo è CPACK_NSIS_MENU_LINKS. Questa variabile contiene collegamenti arbitrari all'albero di installazione o a pagine web esterne. Il primo della coppia è sempre il file o la posizione di origine esistente e il secondo è il nome che verrà visualizzato nel menù Start. Per aggiungere un collegamento al file della guida per cmake-gui e un collegamento alla pagina Web di CMake, aggiungere quanto segue:

```
set (CPACK_NSIS_MENU_LINKS
     "doc/cmake-${VERSION_MAJOR}.${VERSION_MINOR}/cmake-gui.html"
     "cmake-gui Help" "http://www.cmake.org" "CMake Web Site")
```

14.4.3 Opzioni Avanzate di NSIS CPack

Oltre alle variabili già discusse, CPack fornisce alcune variabili aggiuntive configurate direttamente nel file di script NSIS. Queste sono utilizzabili per aggiungere frammenti di script NSIS allo script NSIS finale utilizzato per creare il programma di installazione. Sono le seguenti:

CPACK_NSIS_EXTRA_INSTALL_COMMANDS
Comandi aggiuntivi utilizzati durante l'installazione.

CPACK_NSIS_EXTRA_UNINSTALL_COMMANDS
Comandi aggiuntivi utilizzati durante la disinstallazione.

CPACK_NSIS_CREATE_ICONS_EXTRA
Comandi NSIS aggiuntivi nella sezione delle icone dello script.

CPACK_NSIS_DELETE_ICONS_EXTRA
Comandi NSIS aggiuntivi nella sezione di cancellazione delle icone dello script.

Quando si usano queste variabili si dovrebbe fare riferimento alla documentazione NSIS e l'autore dovrebbe guardare il file `NSIS.template.in` per l'esatto posizionamento delle variabili.

14.4.4 Impostazioni delle Associazioni delle Estensioni dei File Con NSIS

Un esempio di una cosa utile che può essere fatta con i comandi di installazione extra è creare associazioni dalle estensioni di file all'applicazione installata. Ad esempio, avendo un'applicazione CoolStuff in grado di aprire file con estensione `.cool`, si devono impostare i seguenti comandi aggiuntivi di installazione e disinstallazione:

```
set (CPACK_NSIS_EXTRA_INSTALL_COMMANDS "
     WriteRegStr HKCR '.cool' '' 'CoolFile'
     WriteRegStr HKCR 'CoolFile' '' 'Cool Stuff File'
     WriteRegStr HKCR 'CoolFile\\shell' '' 'open'
     WriteRegStr HKCR 'CoolFile\\DefaultIcon' \\
                      '' '$INSTDIR\\bin\\coolstuff.exe,0'
     WriteRegStr HKCR 'CoolFile\\shell\\open\\command' \\
                      '' '$INSTDIR\\bin\\coolstuff.exe \"%1\"'
     WriteRegStr HKCR \"CoolFile\\shell\\edit' \\
                      '' 'Edit Cool File'
     WriteRegStr HKCR 'CoolFile\\shell\\edit\\command' \\
                      '' '$INSTDIR\\bin\\coolstuff.exe \"%1\"'
     System::Call \\
       'Shell32::SHChangeNotify(i 0x8000000, i 0, i 0, i 0)'
     ")

set (CPACK_NSIS_EXTRA_UNINSTALL_COMMANDS "
     DeleteRegKey HKCR '.cool'
```

(continues on next page)

(continua dalla pagina precedente)

```
    DeleteRegKey HKCR 'CoolFile'
    ")
```

Questo crea un'associazione di file Windows a tutti i file che terminano con `.cool`, in modo che quando un utente fa doppio clic su un file `.cool`, `coolstuff.exe` viene eseguito con il path completo del file come argomento. Questo imposta anche un'associazione per modificare il file dal menù di scelta rapida di Windows allo stesso programma `coolstuff.exe`. L'icona di Windows Explorer per il file è impostata sull'icona trovata nell'eseguibile `coolstuff.exe`. Quando viene disinstallato, le chiavi di registro vengono rimosse. Poiché le virgolette doppie e i separatori di path di Windows devono essere "escaped", è meglio inserire questo codice in `CPACK_PROJECT_CONFIG_FILE` per il progetto.

```
configure_file(
  ${CoolStuff_SOURCE_DIR}/CoolStuffCPackOptions.cmake.in
  ${CoolStuff_BINARY_DIR}/CoolStuffCPackOptions.cmake @ONLY)

set (CPACK_PROJECT_CONFIG_FILE
  ${CoolStuff_BINARY_DIR}/CoolStuffCPackOptions.cmake)
include (CPack)
```

14.4.5 Installazione delle Microsoft Run Time Libraries

Sebbene non sia strettamente un comando NSIS CPack, se si creano applicazioni su Windows con il compilatore Microsoft, molto probabilmente si dovranno distribuire le librerie di runtime da Microsoft insieme al progetto. In CMake, tutto ciò che si deve fare è quanto segue:

```
include (InstallRequiredSystemLibraries)
```

Questo aggiungerà le librerie di runtime del compilatore come file di installazione che andranno nella directory bin dell'applicazione. Se non si vuole che le librerie vadano nella directory bin, si deve fare questo:

```
set (CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS_SKIP TRUE)
include (InstallRequiredSystemLibraries)
install (PROGRAMS ${CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS}
         DESTINATION mydir)
```

È importante notare che le librerie di runtime devono trovarsi proprio accanto agli eseguibili del pacchetto affinché Windows possa trovarle. Con Visual Studio 2005 e 2008, è necessario installare anche i file manifest affiancati con l'applicazione durante la distribuzione delle librerie di runtime. Volendo impacchettare una versione di debug del software, si dovrà impostare `CMAKE_INSTALL_DEBUG_LIBRARIES` su ON prima dell'inclusione. Da tenere presente, tuttavia, che i termini della licenza potrebbero proibire di ridistribuire le librerie di debug. Ricontrollare i termini di licenza per la versione di Visual Studio utilizzata prima di decidere di impostare `CMAKE_INSTALL_DEBUG_LIBRARIES` su ON. ON.

14.4.6 Supporto per l'Installazione dei Componenti CPack

Per default, i programmi di installazione di CPack considerano tutti i file installati da un progetto come una singola unità monolitica: o viene installato l'intero set di file o non viene installato alcun file. Tuttavia, con molti progetti ha senso suddividere l'installazione in componenti distinti e selezionabili dall'utente. Alcuni utenti potrebbero voler installare solo i tool a riga di comando per un progetto, mentre altri utenti potrebbero desiderare la GUI o i file header.

Questa sezione descrive come configurare CPack per generare programmi di installazione basati su componenti che consentano agli utenti di selezionare il set di componenti del progetto che desiderano installare. Ad esempio, verrà creato un semplice programma di installazione per una libreria che ha tre componenti: un binario di libreria, un'applicazione di esempio e un file header C++. Al termine, i programmi di installazione risultanti per Windows e Mac OS X sono simili a quelli in Figura 8.

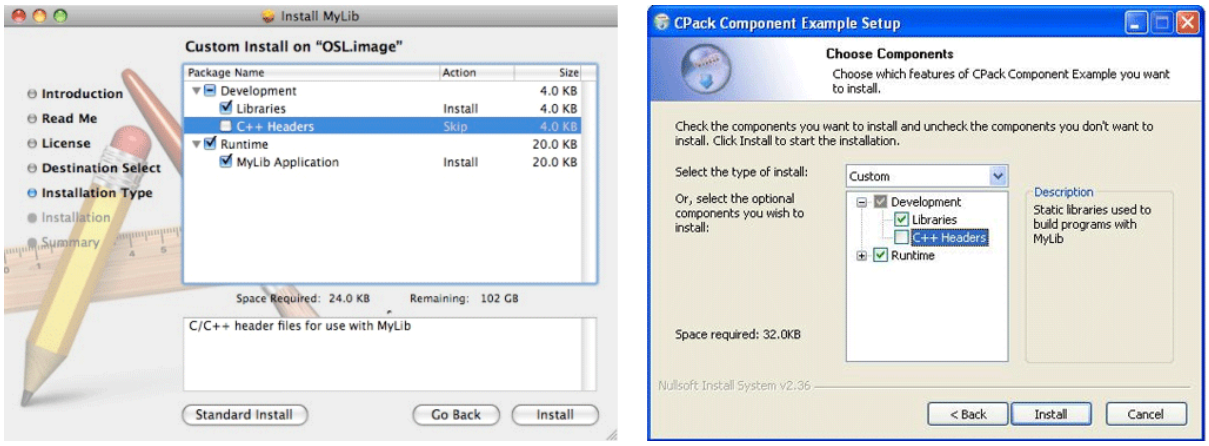


Fig. 8: Installer di componenti per Mac e Windows

Il semplice esempio con cui lavoreremo è il seguente; ha una libreria e un eseguibile. Vengono utilizzati i comandi CPack già trattati.

```
cmake_minimum_required(VERSION 3.20 FATAL_ERROR)
project(MyLib)

add_library(mylib mylib.cpp)

add_executable(mylibapp mylibapp.cpp)
target_link_libraries(mylibapp mylib)

install(TARGETS mylib ARCHIVE DESTINATION lib)
install(TARGETS mylibapp RUNTIME DESTINATION bin)
install(FILES mylib.h DESTINATION include)
# add CPack to project
set(CPACK_PACKAGE_NAME "MyLib")
set(CPACK_PACKAGE_VENDOR "CMake.org")
set(CPACK_PACKAGE_DESCRIPTION_SUMMARY
    "MyLib - CPack Component Installation Example")
```

(continues on next page)

(continua dalla pagina precedente)

```
set(CPACK_PACKAGE_VERSION "1.0.0")
set(CPACK_PACKAGE_VERSION_MAJOR "1")
set(CPACK_PACKAGE_VERSION_MINOR "0")
set(CPACK_PACKAGE_VERSION_PATCH "0")
set(CPACK_PACKAGE_INSTALL_DIRECTORY "CPack Component Example")

# This must always be after all CPACK\_\* variables are defined
include(CPack)
```

Specifica dei Componenti

Il primo passo nella creazione di un'installazione basata su componenti consiste nell'identificare l'insieme di componenti installabili. In questo esempio verranno creati tre componenti: il file binario della libreria, l'applicazione e il file header. Questa decisione è arbitraria e specifica del progetto, ma si devono identificare i componenti che corrispondono alle unità di funzionalità importanti per l'utente, piuttosto che basare i componenti sulla struttura interna del programma.

Per ciascuno di questi componenti, dobbiamo identificare a quale componente appartiene ciascuno dei file installati. Per ogni comando `install` in CMakeLists.txt, si aggiunge un argomento `COMPONENT` appropriato che indichi a quale componente saranno associati i file installati:

```
install(TARGETS mylib
  ARCHIVE
  DESTINATION lib
  COMPONENT libraries)
install(TARGETS mylibapp
  RUNTIME
  DESTINATION bin
  COMPONENT applications)
install(FILES mylib.h
  DESTINATION include
  COMPONENT headers)
```

Si noti che l'argomento `COMPONENT` del comando `install` non è nuovo; ha fatto parte di CMake `install` utilizza uno qualsiasi dei comandi di installazione precedenti (`install_targets`, `install_files`, ecc.), si dovranno convertirli in comandi `install` per poter utilizzare i componenti.

Il passo successivo consiste nel notificare a CPack i nomi di tutti i componenti del progetto chiamando la funzione `cpack_add_component` per ciascun componente del pacchetto:

```
cpack_add_component(applications)
cpack_add_component(libraries)
cpack_add_component(headers)
```

A questo punto si può creare un programma di installazione basato su componenti con CPack che consentirà di installare in modo indipendente le applicazioni, le librerie e gli header di MyLib. I programmi di installazione di Windows e Mac OS X saranno simili a quelli mostrati in Figura 9.

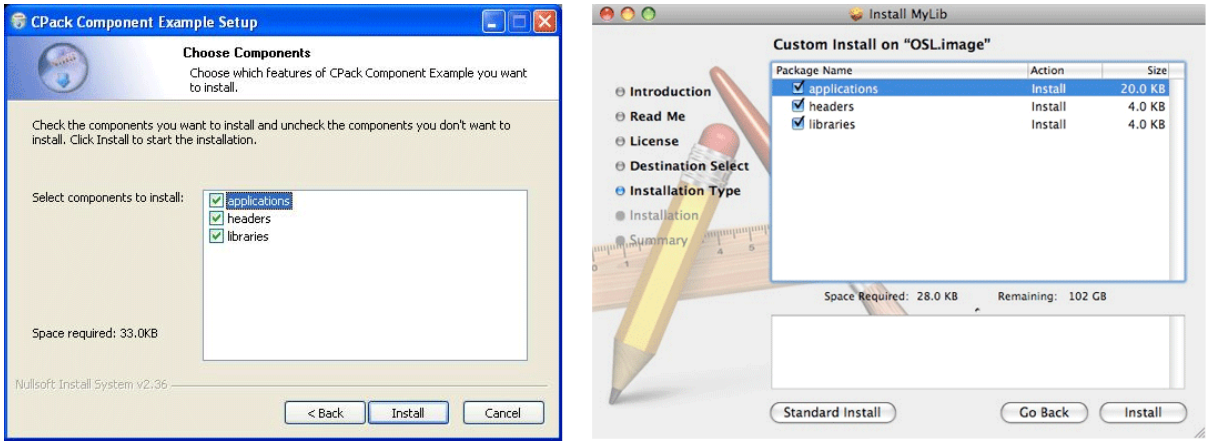


Fig. 9: Prima pagina del programma di installazione dei componenti di Windows e Mac OS X

Denominazione dei Componenti

A questo punto, si può notare che i nomi dei componenti effettivi nel programma di installazione non sono molto descrittivi: dicono semplicemente «applicazioni», «librerie» o «header», come specificato nei nomi dei componenti. Questi nomi possono essere migliorati utilizzando l'opzione DISPLAY_NAME in `cpack_add_component function`:

```
cpack_add_component(applications DISPLAY_NAME
"MyLib Application")
cpack_add_component(libraries DISPLAY_NAME "Libraries")
cpack_add_component(headers DISPLAY_NAME "C++ Headers")
```

Qualsiasi macro con prefisso `CPACK_COMPONENT_${COMPNAME}`, dove `${COMPNAME}` è il nome maiuscolo di un componente, è utilizzato per impostare una particolare proprietà di quel componente nel programma di installazione. Qui, impostiamo la proprietà `DISPLAY_NAME` di ciascuno dei nostri componenti in modo da ottenere nomi leggibili. Questi nomi verranno elencati nella casella di selezione anziché i nomi dei componenti interni «applications», «libraries», «headers»,

Aggiungere le Descrizioni dei Componenti

Esistono diverse altre proprietà associate ai componenti, inclusa la possibilità di rendere un componente nascosto, richiesto o disabilitato per default, che forniscono ulteriori informazioni descrittive. Di particolare rilievo è la proprietà `DESCRIPTION`, che fornisce del testo descrittivo per il componente. Questo testo descrittivo verrà visualizzato in una casella «description» separata nel programma di installazione e verrà aggiornato quando il mouse dell'utente passa sopra il nome del componente corrispondente (Windows) o quando l'utente fa clic su un componente (Mac OS X). Di seguito aggiungeremo una descrizione per ciascuno dei nostri componenti:

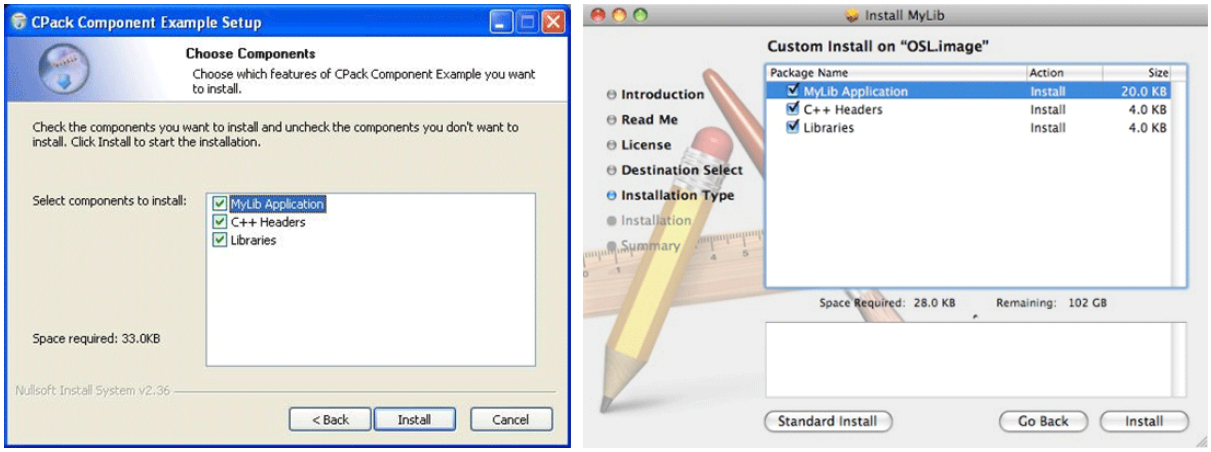


Fig. 10: Programmi di installazione per Windows e Mac OS X con i nomi dei componenti

```
cpack_add_component(applications DISPLAY_NAME "MyLib Application"
  DESCRIPTION
  "An extremely useful application that makes use of MyLib"
  )
cpack_add_component(libraries DISPLAY_NAME "Libraries"
  DESCRIPTION
  "Static libraries used to build programs with MyLib"
  )
cpack_add_component(headers DISPLAY_NAME "C++ Headers"
  DESCRIPTION "C/C++ header files for use with MyLib"
  )
```

Generalmente, le descrizioni dovrebbero fornire all'utente informazioni sufficienti per decidere se installare il componente, ma non dovrebbero esse stesse essere più lunghe di qualche riga (la casella «Descrizione» nei programmi di installazione tende ad essere piccola). La Figura 11 mostra la visualizzazione della descrizione per i programmi di installazione di Windows e Mac OS X.

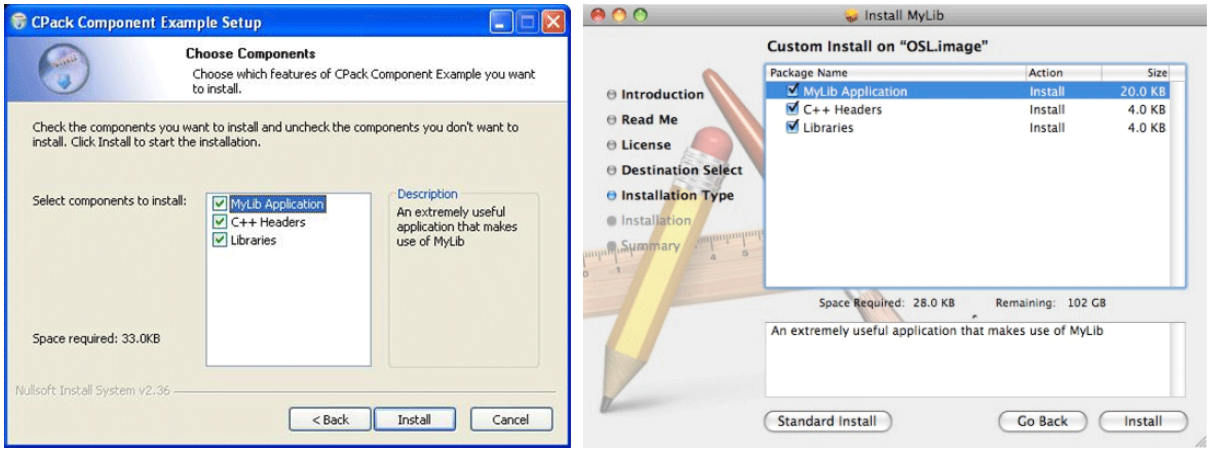


Fig. 11: Installer di componenti con descrizioni

Interdipendenze dei Componenti

Nella maggior parte dei progetti i vari componenti non sono completamente indipendenti. Ad esempio, un componente dell'applicazione può dipendere dalle librerie shared in un altro componente per essere eseguito correttamente, in modo tale che l'installazione del componente dell'applicazione senza le corrispondenti librerie condivise risulterebbe in un'installazione inutilizzabile. CPack consente di esprimere le dipendenze tra i componenti, in modo che un componente venga installato solo se sono installati anche tutti gli altri componenti da cui dipende.

Per illustrare le dipendenze dei componenti inseriremo una semplice restrizione sul nostro programma di installazione basato sui componenti. Poiché non forniamo il codice sorgente nel nostro programma di installazione, i file header C++ che distribuiamo possono essere effettivamente utilizzati solo se l'utente installa anche la libreria binaria per linkare il proprio programma. Pertanto, il componente «headers» dipende dalla disponibilità del componente «libraries». Possiamo esprimere questa nozione impostando la proprietà `DEPENDS` per il componente `HEADERS` come:

```
cpack_add_component(headers DISPLAY_NAME "C++ Headers"
  DESCRIPTION
  "C/C++ header files for use with MyLib"
  DEPENDS libraries
  )
```

La proprietà `DEPENDS` per un componente è in realtà un elenco, in quanto tale componente può dipendere da molti altri componenti. Esprimendo tutte le dipendenze dei componenti in questo modo, è possibile garantire che gli utenti non siano in grado di selezionare un insieme incompleto di componenti al momento dell'installazione.

Raggruppamento di Componenti

Quando il numero di componenti nel progetto aumenta, potrebbe essere necessario fornire un'organizzazione aggiuntiva per l'elenco dei componenti. Per aiutare con questa organizzazione, CPack include la nozione di gruppi di componenti. Un gruppo di componenti è semplicemente un modo per fornire un nome a un gruppo di componenti correlati. All'interno dell'interfaccia utente un gruppo di componenti ha il proprio nome e sotto quel gruppo ci sono i nomi di tutti i componenti in quel gruppo. Gli utenti avranno la possibilità di (de)selezionare l'installazione di tutti i componenti nel gruppo con un solo clic o di espandere il gruppo per selezionare singoli componenti.

Amplieremo il nostro esempio categorizzando i suoi tre componenti, «applications», «libraries» e «headers», nei gruppi «Runtime» e «Development». Possiamo inserire un componente in un gruppo utilizzando l'opzione `GROUP` della funzione `cpack_add_component` come segue:

```
cpack_add_component(applications
 DISPLAY_NAME "MyLib Application"
 DESCRIPTION
  "An extremely useful application that makes use of MyLib"
```

(continues on next page)

(continua dalla pagina precedente)

```
  GROUP Runtime)
cpack_add_component(libraries
  DISPLAY_NAME "Libraries"
  DESCRIPTION
  "Static libraries used to build programs with MyLib"
  GROUP Development)
cpack_add_component(headers
  DISPLAY_NAME "C++ Headers"
  DESCRIPTION "C/C++ header files for use with MyLib"
  GROUP Development
  DEPENDS libraries
  )
```

Come i componenti, i gruppi di componenti hanno varie proprietà che possono essere personalizzate, tra cui `DISPLAY_NAME` e `DESCRIPTION`. Ad esempio, il codice seguente aggiunge una descrizione espansa al gruppo «Development»:

```
cpack_add_component_group(Development
 EXPANDED
 DESCRIPTION
"All of the tools you'll ever need to develop software")
```

Dopo aver personalizzato i gruppi di componenti a proprio piacimento, ricostruire l'installer binario per vedere la nuova organizzazione: l'applicazione MyLib verrà visualizzata sotto il nuovo gruppo «Runtime», mentre la libreria MyLib e l'intestazione C++ verranno visualizzate sotto il nuovo gruppo «Development». È possibile attivare/disattivare facilmente tutti i componenti all'interno di un gruppo utilizzando la GUI dell'installatore. Questo può essere visto in Figura 12.

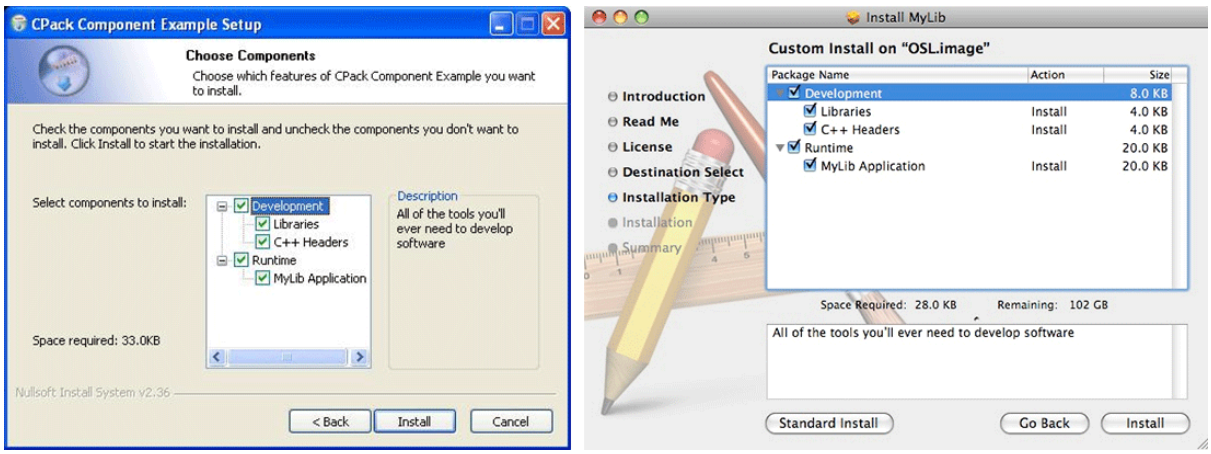


Fig. 12: Raggruppamento dei componenti

Tipi di Installazione (solo NSIS)

Quando un progetto contiene un numero elevato di componenti, è normale che un programma di installazione di Windows fornisca set di componenti preselezionati in base alle esigenze specifiche dell'utente. Ad esempio, un utente che desidera sviluppare software per una libreria vorrà un set di componenti, mentre un utente finale potrebbe utilizzare un set completamente diverso. CPack supporta questa nozione di set di componenti preselezionati tramite i tipi di installazione. Un tipo di installazione è semplicemente un insieme di componenti. Quando l'utente seleziona un tipo di installazione, viene selezionato esattamente quel set di componenti. Quindi l'utente può personalizzare ulteriormente la propria installazione come desiderato. Attualmente questo è supportato solo dal generatore NSIS.

Per il nostro semplice esempio, creeremo due tipi di installazione: un tipo di installazione «Full» che contiene tutti i componenti e un tipo di installazione «Developer» che include solo le librerie e gli header. Per fare questo usiamo la funzione `cpack_add_install_type` per aggiungere i tipi.

```
cpack_add_install_type(Full DISPLAY_NAME "Everything")
cpack_add_install_type(Developer)
```

Successivamente, impostiamo la proprietà `INSTALL_TYPES` di ogni componente per indicare quali tipi di installazione includeranno quel componente. Questo viene fatto con l'opzione INSTALL_TYPES della funzione `cpack_add_component`.

```
 cpack_add_component(libraries DISPLAY_NAME "Libraries"
  DESCRIPTION
   "Static libraries used to build programs with MyLib"
  GROUP Development
  INSTALL_TYPES Developer Full)
cpack_add_component(applications
  DISPLAY_NAME "MyLib Application"
  DESCRIPTION
   "An extremely useful application that makes use of MyLib"
  GROUP Runtime
  INSTALL_TYPES Full)
cpack_add_component(headers
  DISPLAY_NAME "C++ Headers"
  DESCRIPTION "C/C++ header files for use with MyLib"
  GROUP Development
  DEPENDS libraries
  INSTALL_TYPES Developer Full)
```

I componenti possono essere elencati in qualsiasi numero di tipi di installazione. Se si re-builda il programma di installazione di Windows, la pagina dei componenti conterrà una «combo box» che consente di selezionare il tipo di installazione, e quindi il set di componenti corrispondente, come mostrato nella Figura 13.

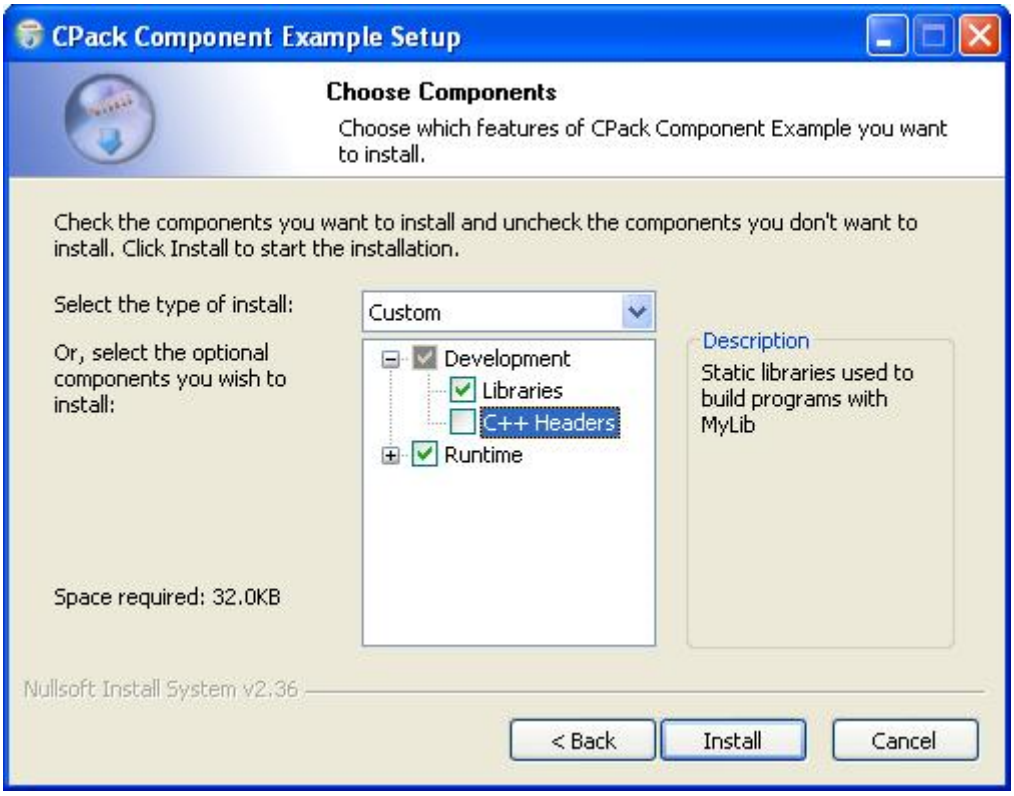


Fig. 13: Tipi di installazioni NSIS

Variabili che controllano i componenti CPack

Le funzioni `cpack_add_install_type`, `cpack_add_component_group` e `cpack_add_component` impostano semplicemente le variabili `CPACK_`. Tali variabili sono descritte nel seguente elenco:

CPACK_COMPONENTS_ALL
: Questo è un elenco contenente i nomi di tutti i componenti che dovrebbero essere installati da CPack. La presenza di questa macro indica che CPack dovrebbe creare un programma di installazione basato su componenti. I file associati a componenti non elencati qui o comandi di installazione non associati a componenti non verranno installati.

CPACK_COMPONENT_${COMPNAME}_DISPLAY_NAME
: Il nome visualizzato del componente ${COMPNAME}, utilizzato nei programmi di installazione grafici per visualizzare il nome del componente. Questo valore può essere qualsiasi stringa.

CPACK_COMPONENT_${COMPNAME}_DESCRIPTION
: Una descrizione estesa del componente ${COMPNAME}, utilizzata negli installer grafici per fornire all'utente ulteriori informazioni sul componente. Le descrizioni possono estendersi su più righe utilizzando «n» come separatore di riga.

CPACK_COMPONENT_${COMPNAME}_HIDDEN
: Un flag che indica che questo componente sarà nascosto nel programma di installazione grafico e quindi non potrà essere selezionato o installato. Disponibile solo con NSIS.

CPACK_COMPONENT_${COMPNAME}_REQUIRED
Un flag che indica che questo componente è obbligatorio e quindi sarà sempre installato. Sarà visibile nell'installer grafico ma non può essere deselezionato.

CPACK_COMPONENT_${COMPNAME}_DISABLED
Un flag che indica che questo componente deve essere disabilitato (deselezionato) per default. L'utente è libero di selezionare questo componente per l'installazione.

CPACK_COMPONENT_${COMPNAME}_DEPENDS
Elenca i componenti da cui dipende questo componente. Se questo componente è selezionato, è necessario selezionare anche ciascuno dei componenti elencati.

CPACK_COMPONENT_${COMPNAME}_GROUP
Denomina un gruppo di componenti di cui fa parte questo componente. Se non fornito, il componente sarà un componente autonomo, non farà parte di alcun gruppo di componenti.

CPACK_COMPONENT_${COMPNAME}_INSTALL_TYPES
Elenca i tipi di installazione di cui fa parte questo componente. Quando viene selezionato uno di questi tipi di installazione, questo componente verrà selezionato automaticamente. Disponibile solo con NSIS.

CPACK_COMPONENT_GROUP_${GROUPNAME}_DISPLAY_NAME
Il nome visualizzato del gruppo di componenti ${GROUPNAME}, utilizzato nei programmi di installazione grafici per visualizzare il nome del gruppo di componenti. Questo valore può essere qualsiasi stringa.

CPACK_COMPONENT_GROUP_${GROUPNAME}_DESCRIPTION
Una descrizione estesa del gruppo di componenti ${GROUPNAME}, utilizzata negli installatori grafici per fornire all'utente informazioni aggiuntive sui componenti contenuti in questo gruppo. Le descrizioni possono estendersi su più righe utilizzando «n» come separatore di riga.

CPACK_COMPONENT_GROUP_${GROUPNAME}_BOLD_TITLE
Un flag che indica se il titolo del gruppo deve essere in grassetto. Disponibile solo con NSIS.

CPACK_COMPONENT_GROUP_${GROUPNAME}_EXPANDED
Un flag che indica se il gruppo deve iniziare «espanso», mostrandone i componenti. Altrimenti verrà mostrato solo il nome del gruppo stesso finché l'utente non fa clic sul gruppo. Disponibile solo con NSIS.

CPACK_INSTALL_TYPE_${INSTNAME}_DISPLAY_NAME
Il nome visualizzato del tipo di installazione. Questo valore può essere qualsiasi stringa.

14.5 CPack per l'installazione di Cygwin

Cygwin (http://www.cygwin.com/) è un ambiente simile a Linux per Windows che consiste in una DLL di runtime e una raccolta di tool. Per aggiungere tool al cygwin ufficiale, viene utilizzato il programma di installazione di cygwin. Lo strumento di installazione ha layout molto specifici per gli alberi sorgenti e binari che devono essere inclusi. CPack può creare i file tar sorgenti e binari e comprimerli correttamente in modo che possano essere caricati sui siti mirror di cygwin. Ovviamente si deve far accettare il pacchetto dalla comunità cygwin prima che ciò avvenga. Poiché il layout del pacchetto è più restrittivo rispetto ad altri strumenti di creazione di pacchetti, potrebbe essere necessario modificare alcune delle opzioni di installazione per il progetto.

Il programma di installazione di Cygwin richiede che tutti i file siano installati in `/usr/bin`, `/usr/share/package-version`, `/usr/share/man` e `/usr/share/doc/package-version`. Il generatore cygwin CPack aggiungerà automaticamente `/usr` alla directory di installazione per il progetto. Il progetto deve installare cose in `share` e `bin`, e CPack aggiungerà automaticamente il prefisso `/usr`.

Cygwin richiede inoltre di fornire uno script di shell che utilizzabile per creare il pacchetto dai sorgenti. Eventuali patch specifiche di cygwin necessarie per il pacchetto devono essere fornite in un ulteriore file diff. Il comando configure_file di CMake può essere utilizzato per creare entrambi questi file per un progetto. Poiché CMake è un pacchetto cygwin, il codice CMake utilizzato per configurare CMake per i generatori cygwin CPack è il seguente

```
set (CPACK_PACKAGE_NAME CMake)

# setup the name of the package for cygwin
set (CPACK_PACKAGE_FILE_NAME
    "${CPACK_PACKAGE_NAME}-${CMake_VERSION}")

# the source has the same name as the binary
set (CPACK_SOURCE_PACKAGE_FILE_NAME ${CPACK_PACKAGE_FILE_NAME})

# Create a cygwin version number in case there are changes
# for cygwin that are not reflected upstream in CMake
set (CPACK_CYGWIN_PATCH_NUMBER 1)

# if we are on cygwin and have cpack, then force the
# doc, data and man dirs to conform to cygwin style directories
set (CMAKE_DOC_DIR "/share/doc/${CPACK_PACKAGE_FILE_NAME}")
set (CMAKE_DATA_DIR "/share/${CPACK_PACKAGE_FILE_NAME}")
set (CMAKE_MAN_DIR "/share/man")

# These files are required by the cmCPackCygwinSourceGenerator and
# the files put into the release tar files.
set (CPACK_CYGWIN_BUILD_SCRIPT
    "${CMake_BINARY_DIR}/@CPACK_PACKAGE_FILE_NAME@-
       @CPACK_CYGWIN_PATCH_NUMBER@.sh")
```

(continues on next page)

(continua dalla pagina precedente)

```
set (CPACK_CYGWIN_PATCH_FILE
     "${CMake_BINARY_DIR}/@CPACK_PACKAGE_FILE_NAME@-
        @CPACK_CYGWIN_PATCH_NUMBER@.patch")

# include the sub directory for cygwin releases
include (Utilities/Release/Cygwin/CMakeLists.txt)

# when packaging source make sure to exclude the .build directory
set (CPACK_SOURCE_IGNORE_FILES
 "/CVS/" "/\\\\.build/" "/\\\\.svn/" "\\\\.swp$" "\\\\.#" "/#" "~$")
```

`Utilities/Release/Cygwin/CMakeLists.txt`:

```
# create the setup.hint file for cygwin
configure_file (
 "${CMake_SOURCE_DIR}/Utilities/Release/Cygwin/cygwin-setup.hint.in"
 "${CMake_BINARY_DIR}/setup.hint")

configure_file (
 "${CMake_SOURCE_DIR}/Utilities/Release/Cygwin/README.cygwin.in"
 "${CMake_BINARY_DIR}/Docs/@CPACK_PACKAGE_FILE_NAME@-
    @CPACK_CYGWIN_PATCH_NUMBER@.README")

install_files (/share/doc/Cygwin FILES
   ${CMake_BINARY_DIR}/Docs/@CPACK_PACKAGE_FILE_NAME@-
       @CPACK_CYGWIN_PATCH_NUMBER@.README)

# create the shell script that can build the project
configure_file (
"${CMake_SOURCE_DIR}/Utilities/Release/Cygwin/cygwin-package.sh.in"
  ${CPACK_CYGWIN_BUILD_SCRIPT})

# Create the patch required for cygwin for the project
configure_file (
"${CMake_SOURCE_DIR}/Utilities/Release/Cygwin/cygwin-patch.diff.in"
  ${CPACK_CYGWIN_PATCH_FILE})
```

Il file `Utilities/Release/Cygwin/cygwin-package.sh.in` si trova nell'albero dei sorgenti di CMake. È uno script di shell utilizzabile per ricreare il pacchetto cygwin dal sorgente. Per altri progetti, esiste uno script template di installazione che si trova in `Templates/cygwin-package.sh.in`. Questo script dovrebbe essere in grado di configurare e impacchettare qualsiasi progetto CPack basato su Cygwin ed è richiesto per tutti i pacchetti Cygwin ufficiali.

Un altro file importante per i binari di Cygwin è `share/doc/Cygwin/package-version.README`. Questo file dovrebbe contenere le informazioni richieste da cygwin sul progetto. Nel caso di CMake, il file è configurato in modo da poter contenere le informazioni sulla versione corretta. Ad esempio, parte di quel file per CMake ha questo aspetto:

```
Build instructions:
  unpack CMake-2.5.20071029-1-src.tar.bz2
    if you use setup to install this src package, it will be
         unpacked under /usr/src automatically
  cd /usr/src
  ./CMake-2.5.20071029-1.sh all
This will create:
  /usr/src/CMake-2.5.20071029.tar.bz2
  /usr/src/CMake-2.5.20071029-1-src.tar.bz2
```

14.6 CPack per Mac OS X PackageMaker

Sul sistema operativo Apple Mac OS X, CPack offre la possibilità di utilizzare il tool di sistema PackageMaker. Questa sezione mostrerà le schermate di installazione dell'applicazione CMake che gli utenti vedranno durante l'installazione del pacchetto CMake su OS X. Le variabili CPack impostate per modificare il testo nel programma di installazione verranno fornite per ciascuna schermata del programma di installazione.

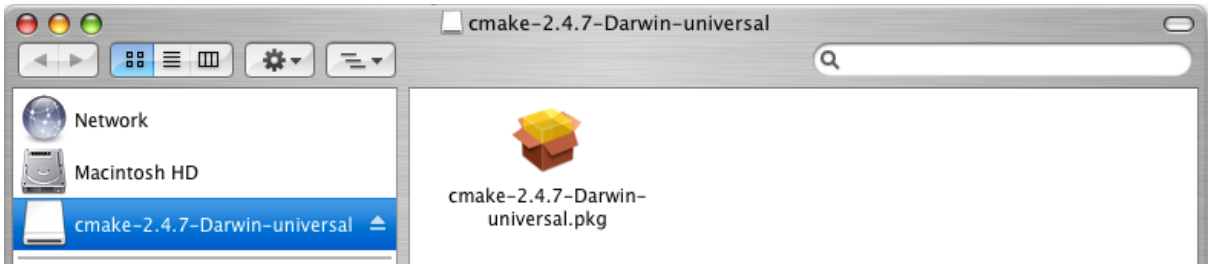


Fig. 14: Pacchetto Mac all'interno di .dmg

La Figura 14 mostra il file `.pkg` che si trova nell'immagine del disco `.dmg` creata dal creatore di pacchetti CPack per Mac OS X. Il nome di questo file è controllato dalla variabile `CPACK_PACKAGE_FILE_NAME`. Se questo non è impostato, CPack utilizzerà un nome di default basato sul nome del pacchetto e sulle impostazioni della versione.

Quando viene eseguito il file `.pkg`, la procedura guidata del pacchetto si avvia con la schermata visualizzata in Figura 15. Il testo in questa finestra è controllato dal file indicato dalla variabile `CPACK_RESOURCE_FILE_WELCOME`.

La figura sopra mostra la sezione Readme della procedura guidata del pacchetto. Il testo di questa finestra è personalizzato utilizzando la variabile `CPACK_RESOURCE_FILE_README`. Dovrebbe contenere un path al file contenente il testo che dovrebbe essere visualizzato su questa schermata.

Questa figura contiene il testo della licenza per il pacchetto. Gli utenti devono accettare la licenza affinché il processo di installazione continui. Il testo della licenza proviene dal file indicato dalla variabile `CPACK_RESOURCE_FILE_LICENSE`.

Le altre schermate nel processo di installazione non sono personalizzabili da CPack. Per modificare le funzionalità più avanzate di questo programma di installazione, ci sono due model-

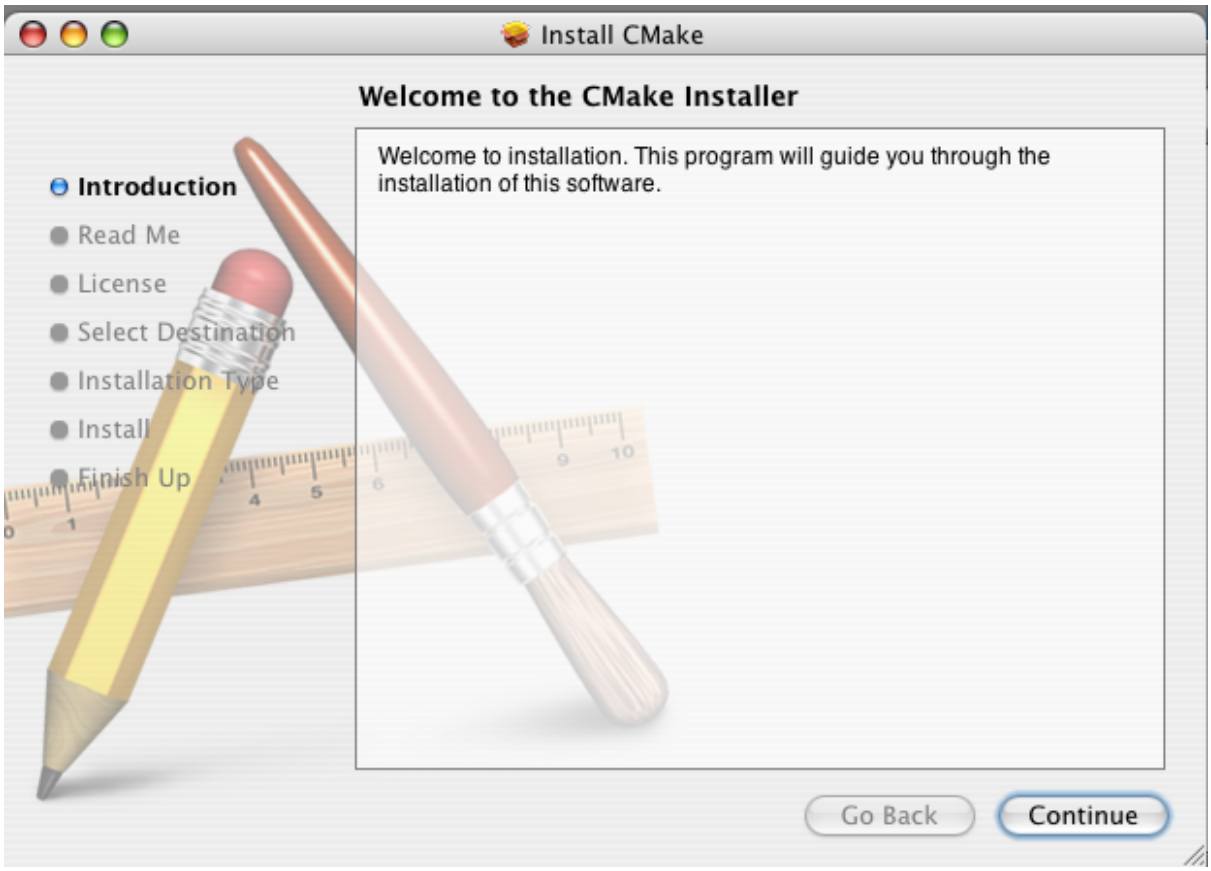


Fig. 15: Schermata introduttiva Mac PackageMaker

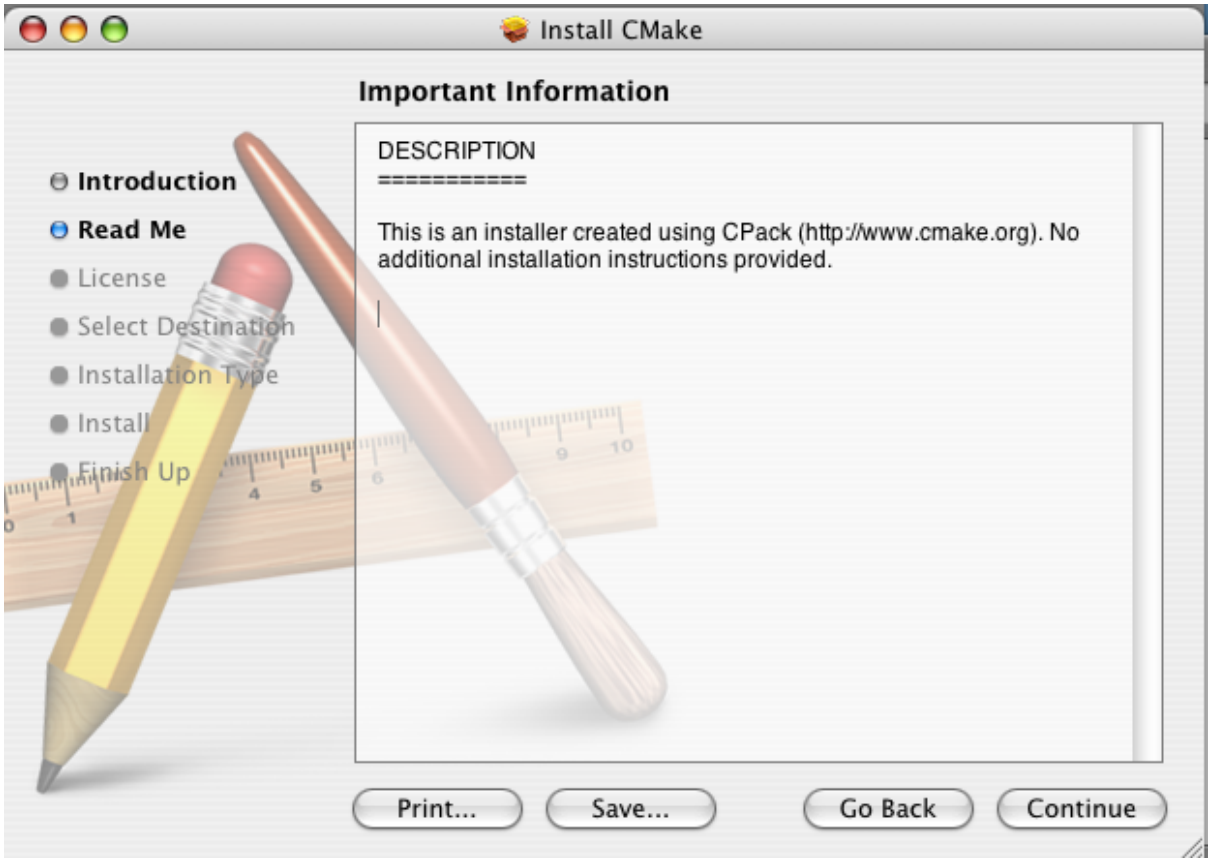


Fig. 16: La sezione Readme della procedura guidata per i pacchetti Mac

li CPack modificabili, `Modules/CPack.Info.plist.in` e `Modules/CPack.Description.plist.in`. Questi file possono essere sostituiti utilizzando la variabile `CMAKE_MODULE_PATH` per puntare a una directory nel progetto contenente una copia modificata di uno o entrambi.

14.7 CPack per Mac OS X «Drag and Drop»

CPack supporta anche la creazione di un programma di installazione Drag and Drop per Mac. In questo caso viene creata un'immagine .dmg. L'immagine contiene sia un link simbolico alla directory /Applications sia una copia dell'albero di installazione del progetto. In questo caso è meglio utilizzare un pacchetto di applicazioni Mac o una singola cartella contenente l'installazione rilocabile come unica destinazione di installazione per il progetto. La variabile `CPACK_PACKAGE_EXECUTABLES` viene utilizzata per puntare al bundle dell'applicazione per il progetto.

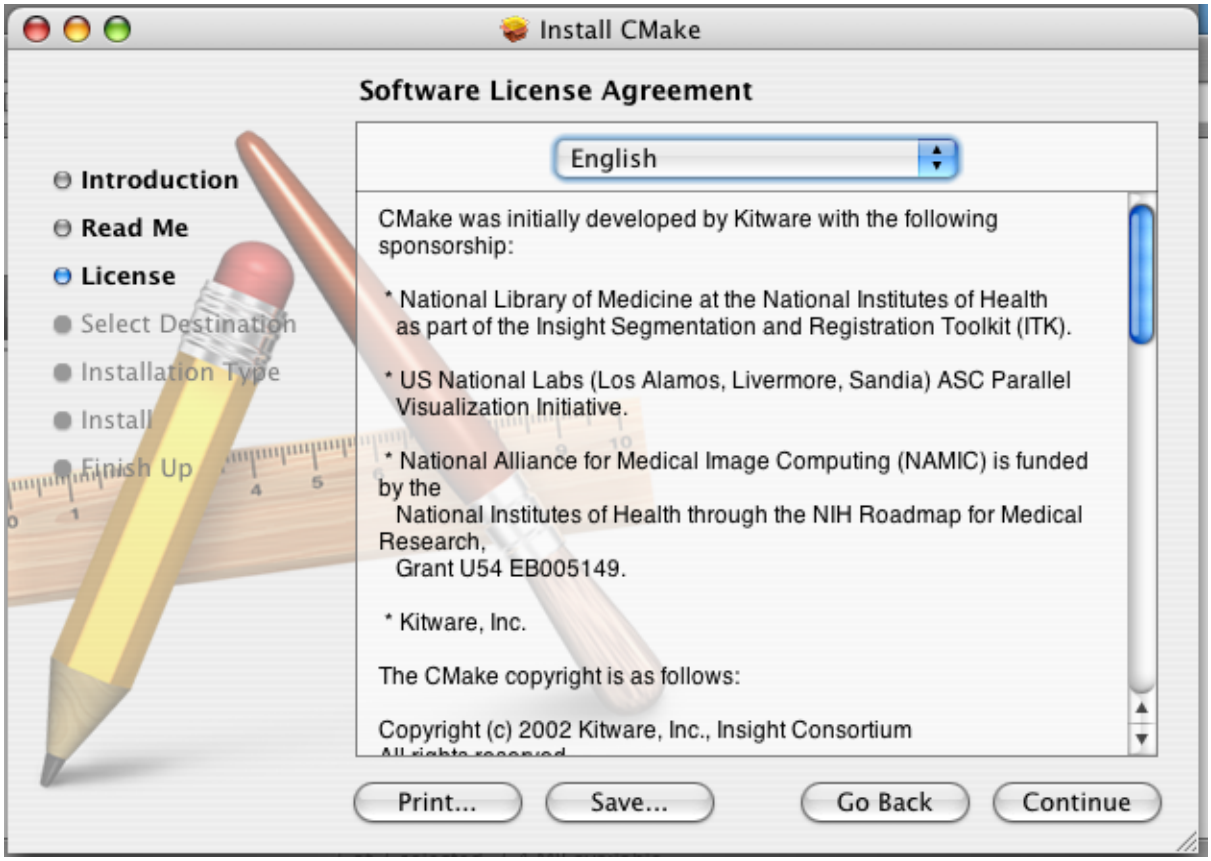


Fig. 17: Schermata della licenza Mac packager

14.8 CPack per applicazioni Mac OS X X11

CPack include anche un generatore di creatori di pacchetti OS X X11. Questo può essere utilizzato per impacchettare applicazioni basate su X11, oltre a farle agire più come applicazioni OS X native racchiudendole con uno script che consentirà agli utenti di eseguirle come farebbero con qualsiasi applicazione OS X nativa. Proprio come il generatore OS X PackageMaker, il generatore OS X X11 crea un file immagine `.dmg`. In questo esempio, un'applicazione X11 chiamata KWPolygonalObjectViewerExample è inclusa nel pacchetto con il generatore OS X X11 CPack.

Questa figura mostra l'immagine del disco creata. In questo caso `CPACK_PACKAGE_NAME` è stato impostato su KWPolygonalObjectViewerExample e le informazioni sulla versione sono state lasciate con il default di CPack di 0.1.1. La variabile `CPACK_PACKAGE_EXECUTABLES` è stata impostata sulla coppia KWPolygonalObjectViewerExample e KWPolygonalObjectViewerExample, l'applicazione X11 installata si chiama KWPolygonalObjectViewerExample.

La figura sopra mostra ciò che un utente vedrebbe dopo aver fatto clic sul file `.dmg` creato da CPack. Mac OS X sta montando questa immagine come disco

Questa figura mostra l'immagine del disco montata. Conterrà un link simbolico alla directory /Applications per il sistema e conterrà un pacchetto di applicazioni per ogni eseguibile trovato in `CPACK_PACKAGE_EXECUTABLES`. Gli utenti possono quindi trascinare e rilasciare le applicazioni nella cartella Applications come mostrato nella figura seguente.

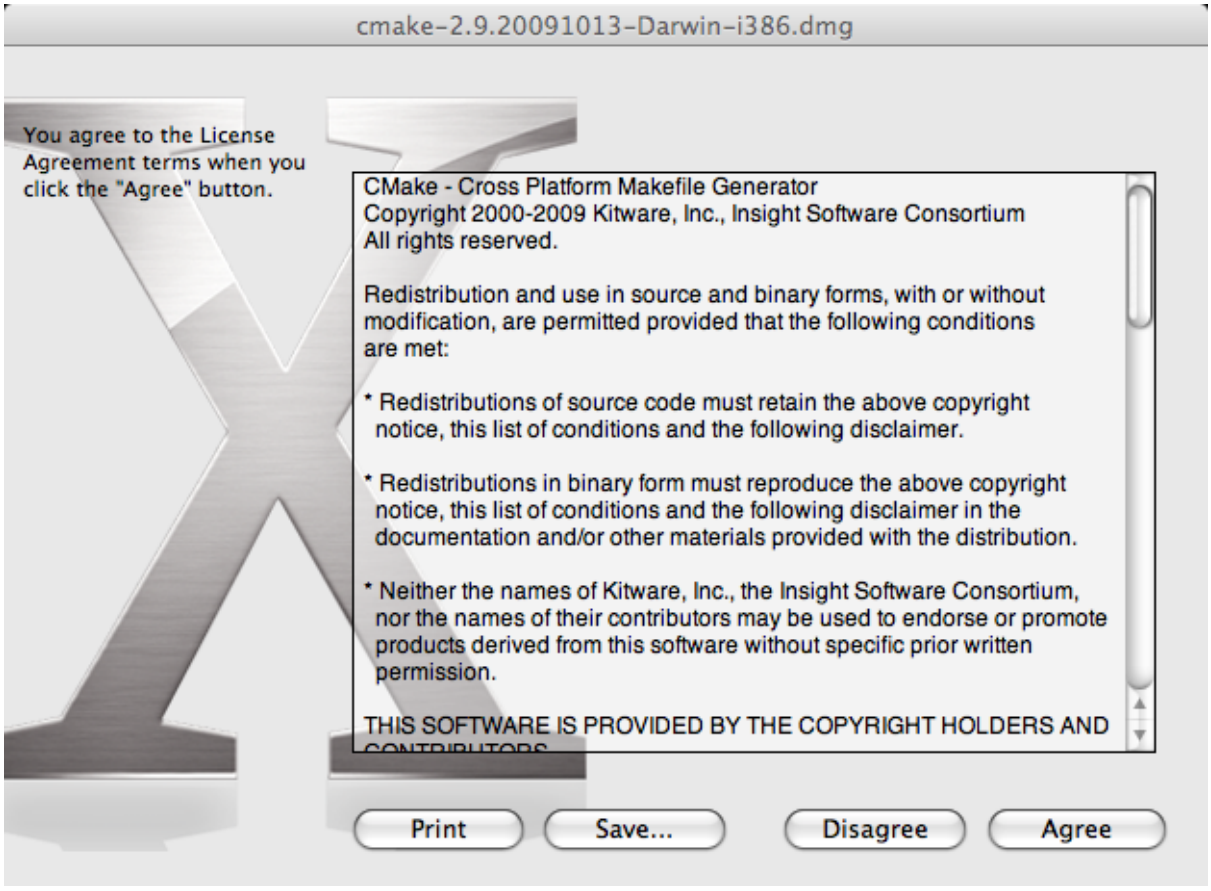


Fig. 18: Finestra di dialogo della licenza Drag and Drop

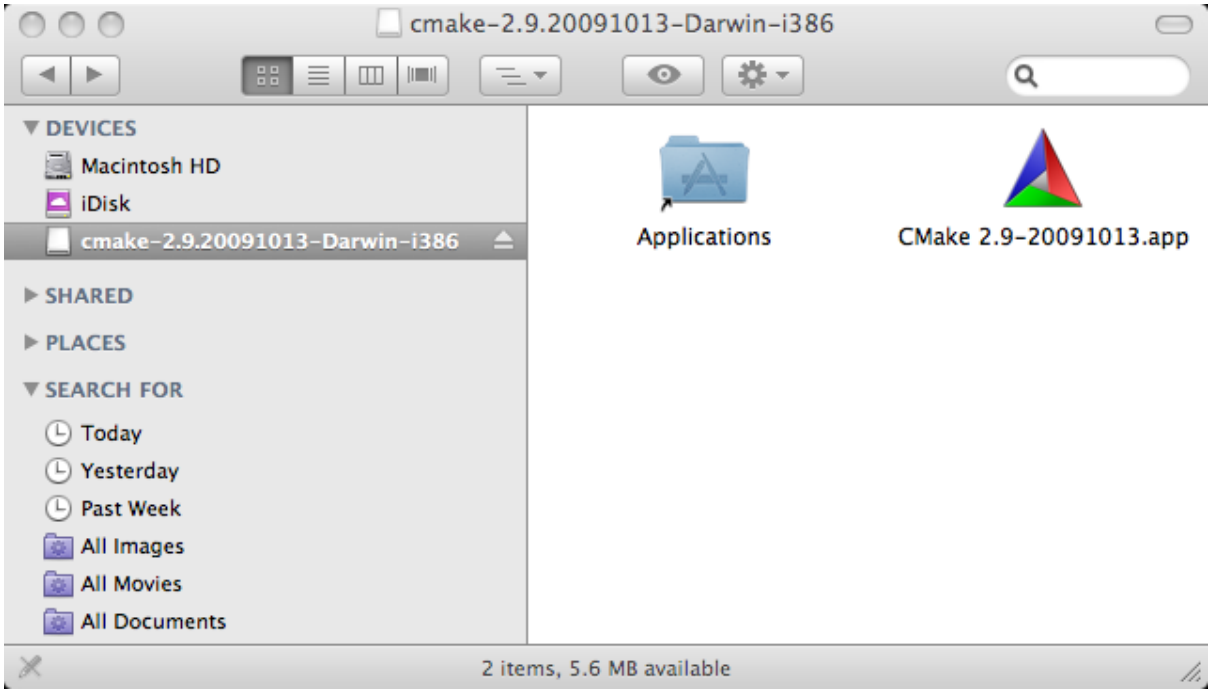


Fig. 19: Cartelle Drag and Drop risultanti

Fig. 20: Immagine del disco del pacchetto Mac OS X X11

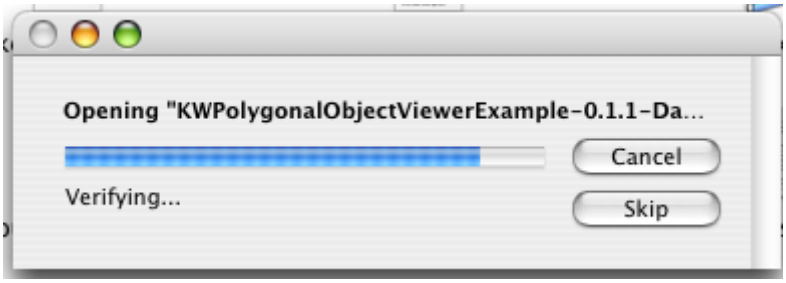


Fig. 21: Apertura dell'immagine disco OS X X11

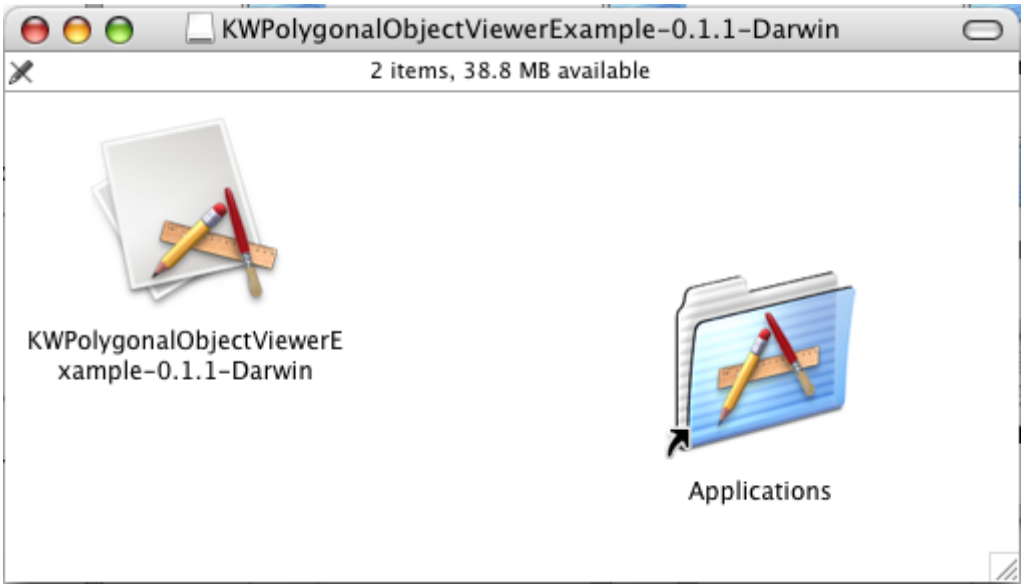


Fig. 22: L'immagine .dmg montata

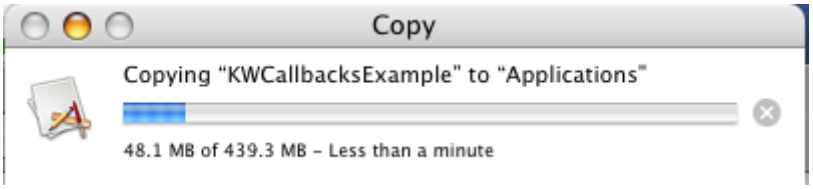


Fig. 23: Drag and drop dell'applicazione in Applications

CPack fornisce effettivamente un eseguibile basato su C++ che può eseguire un'applicazione X11 tramite il linguaggio di scripting Apple. Il bundle dell'applicazione installato eseguirà l'applicazione di inoltro quando l'utente fa doppio clic su KWPolygonalObjectViewerExample. Questo script farà in modo che il server X11 sia avviato. Lo script che viene eseguito si trova in `CMake/Modules/CPack.RuntimeScript.in`. Il sorgente per il programma C++ di avvio degli script si trova in `Source/CPack/OSXScriptLauncher.cxx`.

14.9 CPack per i pacchetti Debian

Un pacchetto Debian `.deb` è semplicemente un archivio «ar». CPack include il codice per l'ar in stile BSD richiesto dai pacchetti Debian. Il packager Debian usa l'insieme standard di variabili CPack per inizializzare un insieme di variabili specifiche di Debian. Questi possono essere sovrascritte in `CPACK_PROJECT_CONFIG_FILE`; il nome del generatore è «DEB». Le variabili utilizzate dal generatore DEB sono le seguenti:

CPACK_DEBIAN_PACKAGE_NAME

il default è il minuscolo di `CPACK_PACKAGE_NAME`.

CPACK_DEBIAN_PACKAGE_ARCHITECTURE

il default è `i386`.

CPACK_DEBIAN_PACKAGE_DEPENDS

Questo deve essere impostato su altri pacchetti da cui dipende questo pacchetto e, se vuoto, viene emesso un warning.

CPACK_DEBIAN_PACKAGE_MAINTAINER

il default è il valore di `CPACK_PACKAGE_CONTACT`

CPACK_DEBIAN_PACKAGE_DESCRIPTION

il default è il valore di `CPACK_PACKAGE_DESCRIPTION_SUMMARY`

CPACK_DEBIAN_PACKAGE_SECTION

il default è `devl`

CPACK_DEBIAN_PACKAGE_PRIORITY

il default è `optional`

14.10 CPack per RPM

CPack ha il supporto per la creazione di file RPM Linux. Il nome del generatore come impostato in `CPACK_GENERATOR` è «RPM». La funzionalità del pacchetto RPM richiede che rpmbuild sia installato sulla macchina e si trovi in PATH. Il packager RPM utilizza il set standard di variabili CPack per inizializzare variabili specifiche di RPM. Le variabili specifiche RPM sono le seguenti:

CPACK_RPM_PACKAGE_SUMMARY

il default è il valore di `CPACK_PACKAGE_DESCRIPTION_SUMMARY`

CPACK_RPM_PACKAGE_NAME

il default è il minuscolo di `CPACK_PACKAGE_NAME`

CPACK_RPM_PACKAGE_VERSION

il default è il valore di `CPACK_PACKAGE_VERSION`.

CPACK_RPM_PACKAGE_ARCHITECTURE

il default è `i386`.

CPACK_RPM_PACKAGE_RELEASE

il default è `1`. Questa è la versione del file RPM, non quella del software che impacchettato.

CPACK_RPM_PACKAGE_GROUP

il default è `none`.

CPACK_RPM_PACKAGE_VENDOR

il default è il valore di `CPACK_PACKAGE_VENDOR`

14.11 I File CPack

Esistono numerosi file utilizzati da CPack che possono essere utili per saperne di più su come funziona CPack e quali opzioni è possibile impostare. Questi file possono essere utilizzati anche come punto di partenza per altri generatori per CPack. Questi file si trovano principalmente nelle directory Modules e Templates di CMake e in genere iniziano con il prefisso CPack. A partire dalla versione 2.8.8, si può anche fare riferimento a `cpack --help-variable-list` e a `cpack --help-variable` per il set completo di variabili `CPACK_*` documentate.

CAPITOLO 15

Test con CMake e CTest

Il test è uno strumento essenziale per la produzione e la manutenzione di software robusto e valido. Questo capitolo esaminerà gli strumenti che fanno parte di CMake per supportare il test del software. Inizieremo con una breve discussione sugli approcci ai test, quindi discuteremo su come aggiungere test al progetto utilizzando CMake.

I test per un pacchetto software possono assumere diverse forme. Al livello più elementare ci sono gli «smoke test», come quello che verifica semplicemente che il software venga compilato. Anche se questo può sembrare un test semplice, con l'ampia varietà di piattaforme e configurazioni disponibili, gli «smoke test» rilevano più problemi di qualsiasi altro tipo di test. Un'altra forma di «smoke test» consiste nel verificare che un test venga eseguito senza arresti anomali. Questo può risultare utile per le situazioni in cui lo sviluppatore non vuole dedicare tempo alla creazione di test più complessi, ma è disposto a eseguirne alcuni semplici. Il più delle volte questi semplici test possono essere piccoli programmi di esempio. Eseguendoli si verifica non solo che la compilazione sia andata a buon fine, ma che tutte le librerie condivise richieste possano essere caricate (per i progetti che le utilizzano) e che almeno una parte del codice possa essere eseguita senza crash.

Andare oltre gli «smoke test» di base porta a test più specifici come quelli di regressione, black-box e white-box. Ognuno di questi ha i suoi punti di forza. Il test di regressione verifica che i risultati di un test non cambino nel tempo o tra le piattaforme. Questo è molto utile se eseguito frequentemente, in quanto fornisce un rapido controllo che il comportamento e i risultati del software non sono cambiati. Quando un test di regressione fallisce, una rapida occhiata alle recenti modifiche al codice può in genere identificare il colpevole. Sfortunatamente, i test di regressione in genere richiedono uno sforzo maggiore per la creazione rispetto ad altri test.

I test white- e black- box si riferiscono a test scritti per testare unità di codice (a vari livelli di integrazione), con e senza conoscenza di come tali unità sono rispettivamente implementate. Il test white-box è progettato per evidenziare potenziali punti di errore nel codice sapendo come quel codice è stato scritto e quindi i suoi punti deboli. Come per i test di regressione, questo può richiedere uno sforzo notevole per creare buoni test. I test black-box in genere sanno poco o nulla

sull'implementazione del software oltre alla sua API pubblica. I test black-box possono fornire molta copertura del codice senza troppi sforzi nello sviluppo dei test. Ciò è particolarmente vero per le librerie di software orientato agli oggetti in cui le API sono ben definite. È possibile scrivere un test black-box per eseguire e richiamare una serie di metodi tipici su tutte le classi del software.

L'ultimo tipo di test che discuteremo è quello di conformità agli standard del software. Mentre gli altri tipi di test che discussi sono incentrati sulla determinazione del corretto funzionamento del codice, quelli di conformità tentano di determinare se il codice aderisce agli standard di codifica del progetto. Questo potrebbe essere un controllo per verificare che tutte le classi abbiano implementato un metodo chiave o che tutte le funzioni abbiano un prefisso comune. Le opzioni per questo tipo di test sono illimitate e ci sono diversi modi per eseguire tale test. Esistono strumenti di analisi del software utilizzabili o potrebbero essere scritti programmi di test specializzati (forse script Python ecc.). Il punto chiave da comprendere è che i test non devono necessariamente comportare l'esecuzione di certe parti del software. I test potrebbero eseguire qualche altro strumento sul codice sorgente stesso.

Esistono diversi motivi per cui è utile avere il supporto per i test, integrato nel processo di build. In primo luogo, i progetti complessi possono avere una serie di opzioni di configurazione o dipendenze dalla piattaforma. Il sistema di build sa quali opzioni possono essere abilitate e può quindi abilitare i test appropriati per tali opzioni. Ad esempio, il Visualization Toolkit (VTK) include il supporto per una libreria di elaborazione parallela denominata MPI. Se VTK viene creato con il supporto MPI, vengono abilitati test aggiuntivi che utilizzano MPI e verificano che il codice specifico di MPI in VTK funzioni come previsto. In secondo luogo, il sistema di build sa dove verranno posizionati gli eseguibili e dispone di strumenti per trovare gli altri eseguibili richiesti (come perl, python ecc.). La terza ragione è che con i Makefile UNIX è normale avere un target test nel Makefile in modo che gli sviluppatori possano digitare make test e far eseguire i test. Affinché funzioni, il sistema di build deve avere una certa conoscenza del processo di test.

15.1 In Che Modo CMake Facilita i Test?

CMake facilita il test del software tramite speciali comandi di test e l'eseguibile `CTest`. Innanzitutto, discuteremo i principali comandi di test in CMake. Per aggiungere test a un progetto basato su CMake, semplicemente `include(CTest)` e si usa il comando `add_test`. Il comando `add_test` ha una sintassi semplice come questa:

```
add_test(NAME TestName COMMAND ExecutableToRun arg1 arg2 ...)
```

Il primo argomento è semplicemente un nome di stringa per il test. Questo è il nome che verrà visualizzato dai programmi di test. Il secondo argomento è l'eseguibile da eseguire. L'eseguibile può essere compilato come parte del progetto o può essere un eseguibile standalone come python, perl, ecc. Gli argomenti rimanenti verranno passati all'eseguibile in esecuzione. Un tipico esempio di test col comando `add_test` sarebbe simile a questo:

```
add_executable(TestInstantiator TestInstantiator.cxx)
target_link_libraries(TestInstantiator vtkCommon)
```

(continues on next page)

(continua dalla pagina precedente)

```
add_test(NAME TestInstantiator
         COMMAND TestInstantiator)
```

Il comando `add_test` viene solitamente inserito nel file CMakeLists per la directory che contiene il test. Per progetti di grandi dimensioni, potrebbero esserci più file CMakeLists con comandi `add_test` al loro interno. Una volta che i comandi `add_test` sono presenti nel progetto, l'utente può eseguire i test richiamando il target «test» di Makefile o il target `RUN_TESTS` di Visual Studio o Xcode. Un esempio di esecuzione sui test CMake utilizzando il generatore di Makefile su Linux potrebbe essere:

```
$ make test
Running tests...
Test project
     Start 2: kwsys.testEncode
 1/20 Test  #2: kwsys.testEncode ..........   Passed    0.02 sec
     Start 3: kwsys.testTerminal
 2/20 Test  #3: kwsys.testTerminal ........   Passed    0.02 sec
     Start 4: kwsys.testAutoPtr
 3/20 Test  #4: kwsys.testAutoPtr .........   Passed    0.02 sec
```

15.2 Ulteriori proprietà dei Test

Per default, un test viene superato se tutte le seguenti condizioni sono vere:

- L'eseguibile da testare è stato trovato
- Il test è stato eseguito senza eccezioni
- Il test è terminato con codice di ritorno 0

Detto questo, tali comportamenti possono essere modificati usando il comando `set_property`:

```
set_property(TEST test_name
             PROPERTY prop1 value1 value2 ...)
```

Questo comando imposterà proprietà aggiuntive per i test specificati. Esempi di proprietà sono:

ENVIRONMENT
: Specifica le variabili di ambiente che devono essere definite per l'esecuzione di un test. Se impostato su un elenco di variabili di ambiente e valori nel formato `MYVAR=value`, tali variabili di ambiente verranno definite durante l'esecuzione del test. L'ambiente viene ripristinato allo stato precedente al termine del test.

LABELS
: Specifica un elenco di etichette di testo associate a un test. Queste etichette possono essere utilizzate per raggruppare i test in base a ciò che testano. Ad esempio, è possibile aggiungere un'etichetta MPI a tutti i test che utilizzano il codice MPI.

WILL_FAIL
Se questa opzione è impostata su true, allora il test passerà se il codice di ritorno non è 0, e fallirà se lo è. Questo inverte la terza condizione dei requisiti di superamento del test.

PASS_REGULAR_EXPRESSION
Se questa opzione è specificata, l'output del test viene confrontato con l'espressione regolare fornita (si può passare anche un elenco di espressioni regolari). Se nessuna delle espressioni regolari corrisponde, il test fallirà. Se almeno uno di essi corrisponde, il test passerà.

FAIL_REGULAR_EXPRESSION
Se questa opzione è specificata, l'output del test viene confrontato con l'espressione regolare fornita (si può passare anche un elenco di espressioni regolari). Se nessuna delle espressioni regolari corrisponde, il test passerà. Se almeno uno di essi corrisponde, il test fallirà.

Se sono specificati sia PASS_REGULAR_EXPRESSION che FAIL_REGULAR_EXPRESSION, allora ha la precedenza FAIL_REGULAR_EXPRESSION. L'esempio seguente illustra l'utilizzo di PASS_REGULAR_EXPRESSION e di FAIL_REGULAR_EXPRESSION:

```
add_test (NAME outputTest COMMAND outputTest)

set (passRegex "^Test passed" "^All ok")
set (failRegex "Error" "Fail")

set_property (TEST outputTest
              PROPERTY PASS_REGULAR_EXPRESSION "${passRegex}")
set_property (TEST outputTest
              PROPERTY FAIL_REGULAR_EXPRESSION "${failRegex}")
```

15.3 Test Con CTest

Quando si eseguono i test dall'ambiente di build, ciò che realmente accade è che tale ambiente esegue CTest. CTest è un eseguibile fornito con CMake; gestisce l'esecuzione dei test per progetto. Mentre CTest funziona bene con CMake, non è necessario utilizzare CMake per utilizzare CTest. Il file di input principale per CTest si chiama CTestTestfile.cmake. Questo file verrà creato in ogni directory elaborata da CMake (in genere ogni directory con un file CMakeLists). La sintassi di CTestTestfile.cmake è come quella di CMake, con un sottoinsieme dei comandi disponibili. Se CMake viene utilizzato per generare file di test, verranno elencate tutte le sottodirectory che devono essere elaborate così come qualsiasi chiamata add_test. Le sottodirectory sono quelle aggiunte dai comandi add_subdirectory. CTest può quindi analizzare questi file per determinare quali test eseguire. Un esempio di tale file è mostrato di seguito:

```
# CMake generated Testfile for
# Source directory: C:/CMake
# Build directory: C:/CMakeBin
#
```

(continues on next page)

(continua dalla pagina precedente)

```
# This file includes the relevant testing commands required
# for testing this directory and lists subdirectories to
# be tested as well.

add_test (SystemInformationNew ...)

add_subdirectory (Source/kwsys)
add_subdirectory (Utilities/cmzlib)
...
```

Quando CTest analizza i file `CTestTestfile.cmake`, ne estrae l'elenco dei test. Questi test verranno eseguiti e per ogni test CTest visualizzerà il nome del test e il suo stato. Si consideri il seguente output di esempio:

```
$ ctest
Test project C:/CMake-build26
        Start 1: SystemInformationNew
 1/21 Test  #1: SystemInformationNew ......   Passed    5.78 sec
        Start 2: kwsys.testEncode
 2/21 Test  #2: kwsys.testEncode ..........   Passed    0.02 sec
        Start 3: kwsys.testTerminal
 3/21 Test  #3: kwsys.testTerminal ........   Passed    0.00 sec
        Start 4: kwsys.testAutoPtr
 4/21 Test  #4: kwsys.testAutoPtr .........   Passed    0.02 sec
        Start 5: kwsys.testHashSTL
 5/21 Test  #5: kwsys.testHashSTL .........   Passed    0.02 sec
...
100% tests passed, 0 tests failed out of 21
Total Test time (real) =  59.22 sec
```

CTest viene eseguito dall'interno dell'albero di build. Eseguirà tutti i test trovati nella directory corrente e in tutte le sottodirectory elencate in `CTestTestfile.cmake`. Per ogni test eseguito, CTest riporterà se il test è stato superato e quanto tempo è stato necessario per eseguirlo.

L'eseguibile CTest include alcune utili opzioni della riga di comando per semplificare un po" i test. Inizieremo esaminando le opzioni che si userebbero normalmente dalla riga di comando.

```
-R <regex>            Run tests matching regular expression
-E <regex>            Exclude tests matching regular expression
-L <regex>            Run tests with labels matching the regex
-LE <regex>           Run tests with labels not matching regexp
-C <config>           Choose the configuration to test
-V,--verbose          Enable verbose output from tests.
-N,--show-only        Disable actual execution of tests.
-I [Start,End,Stride,test#,test#|Test file]
                      Run specific tests by range and number.
-H                                    Display a help message
```

L'opzione `-R` è probabilmente la più usata. Permette di specificare un'espressione regolare; verranno eseguiti solo i test con nomi corrispondenti all'espressione regolare. L'utilizzo dell'opzione `-R` con il nome (o parte del nome) di un test è un modo rapido per eseguire un singolo test. L'opzione `-E` è simile, tranne per il fatto che esclude tutti i test che corrispondono all'espressione regolare. Le opzioni `-L` e `-LE` sono simili a `-R` e `-E`, tranne per il fatto che si applicano alle etichette impostate utilizzando il comando `set_property` descritto in precedenza. L'opzione `-C` è principalmente per le build con IDE in cui si potrebbero avere più configurazioni, come Release e Debug nello stesso albero. L'argomento che segue `-C` determina quale configurazione verrà testata. L'argomento `-V` è utile quando si cerca di determinare perché un test fallisce. Con `-V`, CTest stamperà la riga di comando utilizzata per eseguire il test, nonché qualsiasi output del test stesso. L'opzione `-V` può essere utilizzata con qualsiasi invocazione di CTest per fornire un output più dettagliato. L'opzione `-N` è utile per vedere quali test verrebbero eseguiti da CTest senza eseguirli effettivamente.

Eseguire i test e assicurarsi che passino tutti prima di confermare qualsiasi modifica al software è un modo infallibile per migliorare la qualità del software e il processo di sviluppo. Sfortunatamente, per progetti di grandi dimensioni il numero di test e il tempo necessario per eseguirli possono essere proibitivi. In queste situazioni può essere utilizzata l'opzione `-I` di CTest. L'opzione `-I` consente di specificare in modo flessibile un sottoinsieme dei test da eseguire. Ad esempio, la seguente chiamata di CTest verrà eseguita ogni settimo test.

```
ctest -I ,,7
```

Anche se questo non è buono come eseguire tutti i test, è meglio che non eseguirne nessuno e potrebbe essere una soluzione più pratica per molti sviluppatori. Si noti che se gli argomenti start e end non sono specificati, come in questo esempio, per default verranno utilizzati il primo e l'ultimo test. In un altro esempio, si supponga di voler sempre eseguire alcuni test più un sottoinsieme degli altri. In questo caso si possono aggiungere esplicitamente quei test alla fine degli argomenti per `-I`. Per esempio:

```
ctest -I ,,5,1,2,3,10
```

eseguirà i test 1, 2, 3 e 10, più ogni quinto test. Si possono passare tutti i numeri di test desiderati dopo l'argomento di Stride.

15.4 Utilizzo di CTest per Pilotare Test Complessi

A volte per testare correttamente un progetto è necessario compilare effettivamente il codice durante la fase di test. Ci sono diverse ragioni per questo. Innanzitutto, se i programmi di test vengono compilati come parte del progetto principale, possono finire per occupare una quantità significativa del tempo di building. Inoltre, se un test fallisce, la build principale non dovrebbe fallire. Infine, i progetti IDE possono diventare rapidamente troppo grandi per essere caricati e utilizzati. Il comando CTest supporta un gruppo di opzioni della riga di comando che ne consentono l'utilizzo come eseguibile di test da eseguire. Se utilizzato come eseguibile di test, CTest può eseguire CMake, eseguire il passo della compilazione e infine eseguire un test compilato. Esamineremo ora le opzioni della riga di comando di CTest che supportano la creazione e l'esecuzione di test.

```
--build-and-test  src_directory build_directory
Run cmake on the given source directory using the specified build␣
↪directory.
--test-command        Name of the program to run.
--build-target        Specify a specific target to build.
--build-nocmake       Run the build without running cmake first.
--build-run-dir       Specify directory to run programs from.
--build-two-config    Run cmake twice before the build.
--build-exe-dir       Specify the directory for the executable.
--build-generator     Specify the generator to use.
--build-project       Specify the name of the project to build.
--build-makeprogram   Specify the make program to use.
--build-noclean       Skip the make clean step.
--build-options       Add extra options to the build step.
```

Ad esempio, si consideri il seguente comando `add_test` preso dal file CMakeLists.txt di CMake stesso. Esso mostra come CTest è utilizzabile sia per compilare che per eseguire un test.

```
add_test(simple ${CMAKE_CTEST_COMMAND}
   --build-and-test "${CMAKE_SOURCE_DIR}/Tests/Simple"
                    "${CMAKE_BINARY_DIR}/Tests/Simple"
   --build-generator ${CMAKE_GENERATOR}
   --build-makeprogram ${CMAKE_MAKE_PROGRAM}
   --build-project Simple
   --test-command simple)
```

In questo esempio, al comando `add_test` viene prima passato il nome del test, «simple». Dopo il nome del test, viene specificato il comando da eseguire. In questo caso, il comando di test da eseguire è CTest. Il comando CTest viene referenziato tramite la variabile `CMAKE_CTEST_COMMAND`. Questa variabile è sempre impostata da CMake sul comando CTest proveniente dall'installazione di CMake utilizzata per creare il progetto. Successivamente, vengono specificate le directory dei sorgenti e quella dei binari. Le opzioni successive a CTest sono le `--build-generator` e `--build-makeprogram`. Questi vengono specificate utilizzando le variabili CMake `CMAKE_MAKE_PROGRAM` e `CMAKE_GENERATOR`. Sia `CMAKE_MAKE_PROGRAM` che `CMAKE_GENERATOR` sono definiti da CMake. Questo è un passaggio importante in quanto garantisce che per la creazione del test venga utilizzato lo stesso generatore utilizzato per la creazione del progetto stesso. L'opzione `--build-project` viene passata `Simple`, che corrisponde al comando `project` utilizzato nel test Simple. L'argomento finale è `--test-command` che indica a CTest il comando da eseguire dopo una compilazione riuscita e dovrebbe essere il nome dell'eseguibile che verrà compilato dal test.

15.5 Gestire un Gran Numero di Test

Quando esiste un numero elevato di test in un singolo progetto, è ingombrante disporre di singoli eseguibili per ciascun test. Detto questo, allo sviluppatore del progetto non dovrebbe essere richiesto di creare test con l'analisi [parsing] di argomenti complessi. Questo è il motivo per cui CMake fornisce un comodo comando per la creazione di un programma di test driver. Questo comando si chiama `create_test_sourcelist`. Un test driver è un programma che collega insieme molti piccoli test in un singolo eseguibile. Ciò è utile quando si creano eseguibili statici con librerie di grandi dimensioni per ridurre la dimensione totale richiesta. La firma per `create_test_sourcelist` è la seguente:

```
create_test_sourcelist (SourceListName
                        DriverName
                        test1 test2 test3
                        EXTRA_INCLUDE include.h
                        FUNCTION function
                        )
```

Il primo argomento è la variabile che conterrà l'elenco dei sorgenti che dovranno essere compilati per rendere eseguibile il test. DriverName è il nome del programma del test driver (ad esempio il nome dell'eseguibile risultante). Il resto degli argomenti è costituito da un elenco dei file sorgenti del test. Ogni sorgente del test dovrebbe contenere una funzione che abbia lo stesso nome del file senza estensione (`foo.cxx` dovrebbe avere `int foo(argc, argv);`). L'eseguibile risultante sarà in grado di richiamare ciascuno dei test per nome sulla riga di comando. Gli argomenti `EXTRA_INCLUDE` e `FUNCTION` supportano un'ulteriore personalizzazione del programma del test driver. Consideriamo il seguente frammento di file CMakeLists per vedere come questo comando può essere utilizzato:

```
# create the testing file and list of tests
set (TestToRun
  ObjectFactory.cxx
  otherArrays.cxx
  otherEmptyCell.cxx
  TestSmartPointer.cxx
  SystemInformation.cxx
  ...
)
create_test_sourcelist (Tests CommonCxxTests.cxx ${TestToRun})

# add the executable
add_executable (CommonCxxTests ${Tests})

# Add all the ADD_TEST for each test
foreach (test ${TestsToRun})
  get_filename_component (TName ${test} NAME_WE)
  add_test (NAME ${TName} COMMAND CommonCxxTests ${TName})
endforeach ()
```

Il comando `create_test_sourcelist` viene richiamato per creare un test driver. In questo caso crea e scrive `CommonCxxTests.cxx` nell'albero binario del progetto, usando il resto degli argomenti per determinarne il contenuto. Successivamente, il comando `add_executable` viene utilizzato per aggiungere l'eseguibile alla build. Quindi viene creata una nuova variabile chiamata `TestsToRun` con un valore iniziale delle fonti richieste per il test driver. Il comando `remove` viene utilizzato per rimuovere il programma driver stesso dall'elenco. Quindi, viene utilizzato un comando `foreach` per eseguire il ciclo sui sorgenti rimanenti. Per ogni sorgente, il suo nome di file senza estensione viene estratto e inserito nella variabile `TName`, poi viene aggiunto un nuovo test per `TName`. Il risultato finale è che per ogni sorgente in `create_test_sourcelist` viene chiamato un comando `add_test` con il nome del test. Man mano che vengono aggiunti più test al comando `create_test_sourcelist`, il ciclo `foreach` chiamerà automaticamente `add_test` per ognuno.

15.6 Gestione dei Dati di Test

Oltre a gestire un gran numero di test, CMake contiene un sistema per la gestione dei dati dei test. È incapsulato in un modulo CMake `ExternalData`, scarica dati di grandi dimensioni in base alle necessità, conserva le informazioni sulla versione e consente l'archiviazione distribuita.

Il design di `ExternalData` segue quello dei sistemi di controllo della versione distribuiti che utilizzano identificatori di file basati su hash e archivi di oggetti, ma sfrutta anche la presenza di un sistema di build basato sulle dipendenze. La figura seguente illustra l'approccio. Le strutture dei sorgenti contengono «collegamenti ai contenuti» leggeri, che fanno riferimento ai dati nell'archiviazione remota tramite hash del loro contenuto. Il modulo `ExternalData` produce regole di build per scaricare i dati negli archivi locali e farvi riferimento da alberi di build tramite link simbolici (copie su Windows).

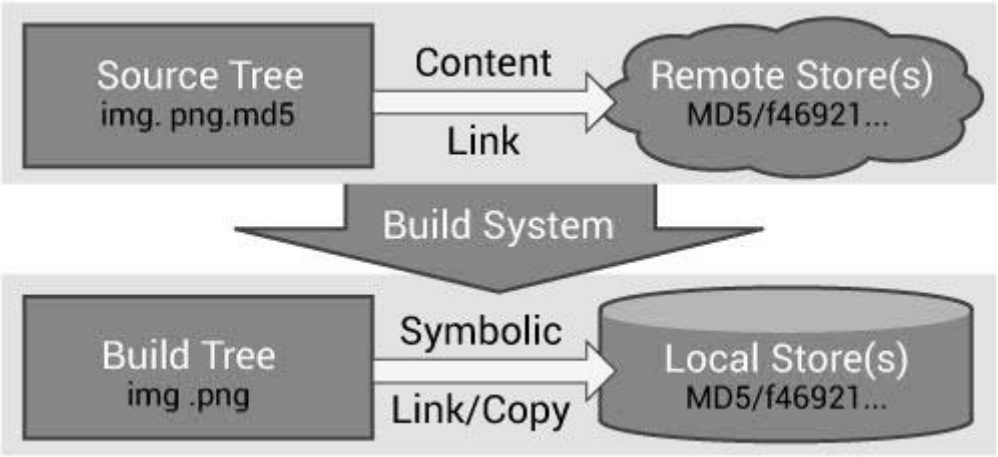


Fig. 1: Diagramma di flusso del modulo ExternalData

Un link al contenuto è un piccolo file di semplice testo contenente un hash dei dati reali. Il suo nome è lo stesso del suo file di dati, con un'estensione aggiuntiva che identifica l'algoritmo di hash, ad es. img.png.md5. I link ai contenuti occupano sempre la stessa (piccola) quantità di spazio nell'albero dei sorgenti, indipendentemente dalla dimensione reale dei dati. I file di configurazione CMake CMakeLists.txt fanno riferimento ai dati utilizzando una sintassi DATA{}

all'interno delle chiamate all'API del modulo ExternalData. Ad esempio, DATA{img.png} dice al modulo ExternalData di rendere disponibile img.png nell'albero di build anche se nell'albero dei sorgenti appare solo un link di contenuto img.png.md5.

Il modulo ExternalData implementa un sistema flessibile per impedire la duplicazione del recupero e dell'archiviazione dei contenuti. Gli oggetti vengono recuperati da un elenco di posizioni locali e remote (possibilmente ridondanti) specificate nella configurazione ExternalData CMake come un elenco di «URL templates». L'unico requisito dei sistemi di archiviazione remota è la capacità di recuperare da un URL che individua il contenuto attraverso la specifica dell'algoritmo hash e del valore hash. I file system locali o in rete, un server FTP Apache o un server Midas, ad esempio, hanno tutti questa capacità. Ogni template di URL ha segnaposto %(algo) e %(hash) per ExternalData da sostituire con i valori di un link al contenuto.

Un archivio oggetti locale persistente può memorizzare nella cache il contenuto scaricato per condividerlo tra gli alberi di build impostando la variabile CMake di configurazione della build ExternalData_OBJECT_STORES. Ciò è utile per de-duplicare il contenuto per più alberi di build. Risolve anche un'importante preoccupazione pragmatica in un contesto di test di regressione; quando molte macchine avviano contemporaneamente una build notturna del dashboard, possono utilizzare il loro archivio oggetti locale invece di sovraccaricare i server di dati e inondare il traffico di rete.

Il recupero è integrato con un sistema di build basato sulle dipendenze, quindi le risorse vengono recuperate solo quando necessario. Ad esempio, se il sistema viene utilizzato per recuperare dati di test e BUILD_TESTING è disattivato, i dati non vengono recuperati inutilmente. Quando l'albero dei sorgenti viene aggiornato e un link al contenuto cambia, il sistema di build recupera i nuovi dati secondo necessità.

Poiché tutti i riferimenti che lasciano l'albero dei sorgenti passano attraverso gli hash, non dipendono da alcuno stato esterno. Gli archivi di oggetti remoti e locali possono essere riposizionati senza invalidare i link ai contenuti nelle versioni precedenti del codice sorgente. I link al contenuto all'interno di un albero di origine possono essere riposizionati o rinominati senza modificare gli archivi oggetti. In un albero di sorgenti possono esistere link ai contenuti duplicati, ma il download avverrà solo una volta. Più versioni di dati con lo stesso nome del file dell'albero dei sorgenti nella cronologia di un progetto vengono identificate in modo univoco negli archivi degli oggetti.

I sistemi basati su hash consentono l'utilizzo di connessioni non attendibili a risorse remote perché il contenuto scaricato viene verificato dopo il recupero. La configurazione dell'elenco dei modelli di URL migliora l'affidabilità consentendo più risorse di archiviazione remote ridondanti. Le risorse di archiviazione possono anche cambiare nel tempo in base alle necessità. Se l'archiviazione remota di un progetto si sposta nel tempo, è sempre possibile creare versioni precedenti del codice sorgente modificando i modelli di URL configurati per l'albero di build o popolando manualmente un archivio oggetti locale.

Una semplice applicazione del modulo ExternalData ha il seguente aspetto:

```
include(ExternalData)
set(midas "http://midas.kitware.com/MyProject")
```

(continues on next page)

(continua dalla pagina precedente)

```
# Add standard remote object stores to user's
# configuration.
list(APPEND ExternalData_URL_TEMPLATES
 "${midas}?algorithm=%(algo)&hash=%(hash)"
 "ftp://myproject.org/files/%(algo)/%(hash)"
 )
# Add a test referencing data.
ExternalData_Add_Test(MyProjectData
 NAME SmoothingTest
 COMMAND SmoothingExe DATA{Input/Image.png}
                      SmoothedImage.png
 )
# Add a build target to populate the real data.
ExternalData_Add_Target(MyProjectData)
```

La funzione `ExternalData_Add_Test` è un wrapper del comando `add_test` di CMake. L'albero dei sorgenti viene analizzato per un link al contenuto Input/Image.png.md5 contenente l'hash MD5 dei dati. Dopo aver verificato l'archivio oggetti locale, viene effettuata una richiesta in sequenza a ciascun URL nell'elenco `ExternalData_URL_TEMPLATES` con l'hash dei dati. Una volta trovato, viene creato un link simbolico nell'albero di compilazione. Il percorso DATA{Input/Image.png} si espanderà nel percorso dell'albero di build nella riga di comando del test. I dati vengono recuperati quando viene creato il target MyProjectData.

CAPITOLO 16

CDash

Man mano che le esigenze dei test del progetto crescono, la gestione dei risultati può diventare travolgente. Ciò è particolarmente vero per i progetti che vengono testati ogni notte su diverse piattaforme. In questi casi, si consiglia di utilizzare una dashboard di test per riepilogare i risultati del test.

Un dashboard di test riassume i risultati di molti test su molte piattaforme e i suoi collegamenti ipertestuali consentono di approfondire rapidamente ulteriori livelli di dettaglio. L'eseguibile CTest include il supporto per la produzione di dashboard di test. Se eseguito con le opzioni corrette, CTest produrrà un output basato su XML registrando i risultati della build e del test e inviandoli a un server dashboard. Il server dashboard esegue un pacchetto software open source chiamato CDash. CDash raccoglie i risultati XML e da essi produce pagine Web HTML.

Prima di discutere su come utilizzare CTest per produrre una dashboard, consideriamo le parti principali di una dashboard di test. Ogni notte a un'ora specificata, il server della dashboard aprirà una nuova dashboard, quindi ogni giorno c'è una nuova pagina web che mostra i risultati dei test per quel periodo di ventiquattro ore. Ci sono link nella pagina principale che consentono di navigare rapidamente in giorni diversi. Guardando la pagina principale di un progetto (come la dashboard di CMake su www.cmake.org), si vedrà che è divisa in alcuni componenti principali. Nella parte superiore ci sono una serie di collegamenti che consentono di passare a dashboard precedenti, nonché link a pagine di progetto come bug tracker, documentazione, ecc.

Sotto, si trovano gruppi di risultati. In genere i gruppi includono Nightly, Experimental, Continuous, Coverage e Dynamic Analysis. La categoria in cui verrà inserita una voce della dashboard dipende da come è stata generata. Le più semplici sono voci Experimental che rappresentano i risultati della dashboard per la copia corrente di qualcuno del codice sorgente del progetto. Con una dashboard sperimentale, non è garantito che il codice sorgente sia aggiornato. Al contrario, una voce della dashboard Nightly è quella in cui CTest tenta di aggiornare il codice sorgente a una data e un'ora specifiche. L'aspettativa è che tutte le voci del dashboard nightly per un determinato giorno siano basate sullo stesso codice sorgente.

Una voce nelle dashboard Continuous viene eseguita ogni volta che vengono archiviati nuovi file. A seconda della frequenza con cui i nuovi file vengono controllati, la dashboard di un singolo giorno potrebbe avere molte voci. Le dashboard Continuous sono particolarmente utili per i progetti multipiattaforma in cui un problema può presentarsi solo su alcune piattaforme. In questi casi uno sviluppatore può eseguire il commit di una modifica che funziona per loro sulla propria piattaforma e poi un'altra piattaforma che esegue una build continua potrebbe rilevare l'errore, consentendo allo sviluppatore di correggere il problema prontamente.

Le dashboard Dynamic Analysis e Coverage sono progettate per testare la sicurezza della memoria e la copertura del codice di un progetto. Una voce nella dashboard Dynamic Analysis è quella in cui tutti i test vengono eseguiti con un programma per la memoria riguardo l'accesso e il controllo dei leak abilitato. Eventuali errori o warning risultanti vengono analizzati, riepilogati e visualizzati. Questo è importante per verificare che il software non stia perdendo memoria [leak] o leggendo dalla memoria non inizializzata. Le voci della dashboard Coverage sono simili in quanto vengono eseguiti tutti i test, ma poiché sono le righe di codice in esecuzione, vengono monitorate. Quando tutti i test sono stati eseguiti, viene prodotto e visualizzato sulla dashboard un elenco di quante volte ciascuna riga di codice è stata eseguita.

16.1 Aggiungere il Supporto CDash Dashboard a un Progetto

Il modo più semplice per iniziare a utilizzare CDash è registrare un account e creare un nuovo progetto su https://my.cdash.org.

Se si preferisce installare il proprio server CDash, seguire una di queste guide:

- Installation guide
- Docker instructions

16.2 Configurazione del Client

Per supportare le dashboard nel progetto si deve includere il modulo `CTest` come segue.

```
# Include CDash dashboard testing module
include(CTest)
```

Il modulo CTest leggerà quindi le impostazioni dal file `CTestConfig.cmake` creato o scaricato da CDash. Se sono state aggiunte chiamate al comando `add_test` al progetto, creare una voce della dashboard è semplice come eseguire:

```
ctest -D Experimental
```

L'opzione `-D` indica a CTest di creare una voce della dashboard. L'argomento successivo indica il tipo di voce della dashboard da creare. La creazione di una voce della dashboard comporta alcuni passaggi che possono essere eseguiti in modo indipendente o come un unico comando. In questo esempio, l'argomento Experimental farà in modo che CTest esegua una serie di passaggi

diversi come un unico comando. Di seguito sono riepilogate le diverse fasi della creazione di una voce della dashboard.

Start
: Prepara una nuova voce della dashboard. Questo crea una sottodirectory `Testing` nella directory di build. La sottodirectory `Testing` conterrà una sottodirectory per i risultati della dashboard con un nome che corrisponde all'ora della dashboard. La sottodirectory `Testing` conterrà anche una sottodirectory per i risultati temporanei dei test chiamata `Temporary`.

Update
: Esegue un aggiornamento del controllo del codice sorgente (in genere utilizzato per esecuzioni notturne o "continuous"). Attualmente CTest supporta Concurrent Versions System (CVS), Subversion, Git, Mercurial e Bazaar.

Configure
: Esegue CMake sul progetto per assicurare che i Makefile o i file di progetto siano aggiornati.

Build
: Build del software utilizzando il generatore specificato.

Test
: Esegue tutti i test e registra i risultati.

MemoryCheck
: Esegue i controlli della memoria usando Purify o valgrind.

Coverage
: Raccoglie informazioni sulla copertura del codice sorgente utilizzando gcov o Bullseye.

Submit
: Invia i risultati del test come voce della dashboard al server.

Ciascuno di questi passaggi può essere eseguito indipendentemente per una voce Nightly o Experimental utilizzando la seguente sintassi:

```
ctest -D NightlyStart
ctest -D NightlyBuild
ctest -D NightlyCoverage -D NightlySubmit
```

oppure

```
ctest -D ExperimentalStart
ctest -D ExperimentalConfigure
ctest -D ExperimentalCoverage -D ExperimentalSubmit
```

In alternativa, si possono utilizzare le scorciatoie che eseguono le combinazioni più comuni tutte in una volta. Le [shortcut] che CTest ha definito includono:

ctest -D Experimental
: esegue i comandi start, configure, build, test, coverage e submit.

ctest -D Nightly
esegue i comandi start, update, configure, build, test, coverage e submit.

ctest -D Continuous
esegue i comandi start, update, configure, build, test, coverage e submit.

ctest -D MemoryCheck
esegue i comandi start, configure, build, memorycheck, coverage e submit.

Quando si configura per la prima volta una dashboard è spesso utile combinare l'opzione `-D` con la `-V`. Ciò consentirà di vedere l'output di tutte le diverse fasi del processo della dashboard. Allo stesso modo, CTest mantiene i file di log nella directory `Testing/Temporary` che crea nell'albero binario. Lì ci sono i file di log per l'esecuzione più recente della dashboard. Anche i risultati della dashboard (file XML) vengono archiviati nella directory `Testing`.

16.3 Personalizzazione di una Dashboard per un Progetto

CTest ha alcune opzioni che possono essere utilizzate per controllare come elabora un progetto. Se, quando CTest esegue una dashboard, trova i file `CTestCustom.ctest` nell'albero binario, caricherà questi file e utilizzerà le relative impostazioni per controllarne il comportamento. La sintassi di un file CTestCustom è la stessa della normale sintassi di CMake. Detto questo, in questo file vengono normalmente utilizzati solo i comandi set. Questi comandi specificano le proprietà che CTest prenderà in considerazione durante l'esecuzione del test.

16.3.1 Impostazioni degli Invii della Dashboard

Alcune delle impostazioni di base della dashboard sono fornite nel file scaricato da CDash. È possibile modificare questi valori iniziali e fornire valori aggiuntivi, se lo si desidera. Il primo valore impostato è l'ora di inizio della nightly. Questo è il tempo che le dashboard di tutto il mondo utilizzeranno per controllare la loro copia del codice sorgente notturno. Questo tempo controlla anche il modo in cui gli invii della dashboard verranno raggruppati. Tutti gli invii dall'ora di inizio del nightly fino all'ora di inizio del nightly successivo saranno inclusi nello stesso «giorno».

```
# Dashboard is opened for submissions for a 24 hour period
# starting at the specified NIGHTLY_START_TIME. Time is
# specified in 24 hour format.
set (CTEST_NIGHTLY_START_TIME "01:00:00 UTC")
```

Il prossimo gruppo di impostazioni controlla dove inviare i risultati del test. Questa è la posizione del server CDash.

```
# CDash server to submit results (used by client)
set (CTEST_DROP_METHOD http)
set (CTEST_DROP_SITE "my.cdash.org")
```

(continues on next page)

(continua dalla pagina precedente)

```
set (CTEST_DROP_LOCATION "/submit.php?project=KensTest")
set (CTEST_DROP_SITE_CDASH TRUE)
```

CTEST_DROP_SITE specifica la posizione del server CDash. I risultati di build e test generati dai client CDash vengono inviati a questa posizione. CTEST_DROP_LOCATION è la directory o l'URL HTTP sul server in cui i client CDash lasciano i report di build e test. CTEST_DROP_SITE_CDASH specifica che il server corrente è CDash, il che impedisce a CTest di tentare di «trigger»-are l'invio (ciò viene comunque fatto se questa variabile non è impostata per consentire la retrocompatibilità con Dart e Dart 2).

Attualmente CDash supporta solo il metodo di invio drop HTTP; tuttavia CTest supporta altri tipi di invio. CTEST_DROP_METHOD specifica il metodo utilizzato per inviare i risultati dei test. L'impostazione più comune per questo sarà HTTP che utilizza il protocollo HTTP (Hyper Text Transfer Protocol) per trasferire i dati del test al server. Altri metodi di rilascio sono supportati per casi speciali come FTP e SCP. Nell'esempio seguente, i client che inviano i propri risultati utilizzando il protocollo HTTP utilizzano un indirizzo Web come sito di destinazione. Se l'invio avviene tramite FTP, questa posizione è relativa a dove CTEST_DROP_SITE_USER effettuerà l'accesso per default. CTEST_DROP_SITE_USER specifica il nome utente FTP che il client utilizzerà sul server. Per gli invii FTP questo utente sarà tipicamente «anonymous». Tuttavia, è possibile utilizzare qualsiasi nome utente in grado di comunicare con il server. Per i server FTP che richiedono una password, può essere memorizzata nella variabile CTEST_DROP_SITE_PASSWORD. CTEST_DROP_SITE_MODE (non utilizzata in questo esempio) è una variabile facoltativa utilizzabile per specificare la modalità FTP. La maggior parte dei server FTP gestirà la modalità passiva di default, ma si può impostare esplicitamente la modalità su attiva se il server non lo fa.

CTest può anche essere eseguito da dietro un firewall. Se il firewall consente il traffico FTP o HTTP, non sono necessarie impostazioni aggiuntive. Se il firewall richiede un proxy FTP/HTTP o utilizza un proxy di tipo SOCKS4 o SOCKS5, è necessario impostare alcune variabili di ambiente. HTTP_PROXY e FTP_PROXY specificano i server che gestiscono le richieste di proxy HTTP e FTP. HTTP_PROXY_PORT e FTP_PROXY_PORT specificano la porta su cui risiedono i proxy HTTP e FTP. HTTP_PROXY_TYPE specifica il tipo di proxy HTTP utilizzato. I tre diversi tipi di proxy supportati sono quelli di default, che includono un proxy HTTP/FTP generico, «SOCKS4» e «SOCKS5», che specificano i proxy compatibili con SOCKS4 e SOCKS5.

16.3.2 Filtraggio di Errori e Warning

Per default, CTest dispone di un elenco di espressioni regolari a cui corrisponde per trovare gli errori e gli avvisi dall'output del processo di build. Queste impostazioni si possono sovrascrivere nei file `CTestCustom.ctest` utilizzando diverse variabili come mostrato di seguito.

```
set (CTEST_CUSTOM_WARNING_MATCH
  ${CTEST_CUSTOM_WARNING_MATCH}
  "{standard input}:[0-9][0-9]*: Warning: "
  )
```

(continues on next page)

(continua dalla pagina precedente)

```
set (CTEST_CUSTOM_WARNING_EXCEPTION
  ${CTEST_CUSTOM_WARNING_EXCEPTION}
  "tk8.4.5/[^/]+/[^/]+.c[:\"]"
  "xtree.[0-9]+. : warning C4702: unreachable code"
  "warning LNK4221"
  "variable .var_args[2]*. is used before its value is set"
  "jobserver unavailable"
  )
```

Un'altra caratteristica utile dei file CTestCustom è che è possibile utilizzarlo per limitare i test eseguiti per le dashboard di controllo della memoria. Il controllo della memoria con purify o valgrind è un processo ad alta intensità di CPU che può richiedere venti ore per una dashboard che normalmente richiede un'ora. Per alleviare questo problema, CTest consente di escludere alcuni dei test dal processo di controllo della memoria come segue:

```
set (CTEST_CUSTOM_MEMCHECK_IGNORE
     ${CTEST_CUSTOM_MEMCHECK_IGNORE}
  TestSetGet
  otherPrint-ParaView
  Example-vtkLocal
  Example-vtkMy
  )
```

Il formato per escludere i test è semplicemente un elenco di nomi di test come specificato quando i test sono stati aggiunti nel file CMakeLists con `add_test`.

Oltre alle impostazioni mostrate, come `CTEST_CUSTOM_WARNING_MATCH`, `CTEST_CUSTOM_WARNING_EXCEPTION` e `CTEST_CUSTOM_MEMCHECK_IGNORE`, CTest controlla anche diverse altre variabili.

CTEST_CUSTOM_ERROR_MATCH
: Espressioni regolari aggiuntive per considerare una riga di build come una riga di errore

CTEST_CUSTOM_ERROR_EXCEPTION
: Espressioni regolari aggiuntive per considerare una riga di build come una riga di errore

CTEST_CUSTOM_WARNING_MATCH
: Espressioni regolari aggiuntive per considerare una riga di build come una riga di warning

CTEST_CUSTOM_WARNING_EXCEPTION
: Espressioni regolari aggiuntive per considerare una riga di build come una riga di warning

CTEST_CUSTOM_MAXIMUM_NUMBER_OF_ERRORS
: Numero massimo di errori prima che CTest smetta di segnalare gli errori (default 50)

CTEST_CUSTOM_MAXIMUM_NUMBER_OF_WARNINGS
: Numero massimo di warning prima che CTest smetta di segnalarli (default 50)

CTEST_CUSTOM_COVERAGE_EXCLUDE
: Espressioni regolari per i file da escludere dall'analisi della «coverage»

CTEST_CUSTOM_PRE_MEMCHECK
Elenco dei comandi da eseguire prima di eseguire il controllo della memoria

CTEST_CUSTOM_POST_MEMCHECK
Elenco dei comandi da eseguire dopo aver eseguito il controllo della memoria

CTEST_CUSTOM_MEMCHECK_IGNORE
Elenco dei test da escludere dalla fase di verifica della memoria

CTEST_CUSTOM_PRE_TEST
Elenco dei comandi da eseguire prima di eseguire il test

CTEST_CUSTOM_POST_TEST
Elenco dei comandi da eseguire dopo aver eseguito il test

CTEST_CUSTOM_TESTS_IGNORE
Elenco dei test da escludere dalla fase di test

CTEST_CUSTOM_MAXIMUM_PASSED_TEST_OUTPUT_SIZE
Dimensione massima dell'output del test per il test superato (default 1k)

CTEST_CUSTOM_MAXIMUM_FAILED_TEST_OUTPUT_SIZE
Dimensione massima dell'output del test per il test fallito (default 300k)

I comandi specificati in `CTEST_CUSTOM_PRE_TEST` e `CTEST_CUSTOM_POST_TEST`, oltre a quelli equivalenti per il controllo della memoria, vengono eseguiti una volta per esecuzione di CTest. Questi comandi possono essere utilizzati, ad esempio, se tutti i test richiedono l'esecuzione di una configurazione iniziale e di una pulizia finale.

16.3.3 Aggiungere Note a una Dashboard

CTest e CDash supportano l'aggiunta di file di note a un invio di dashboard. Questi appariranno sulla dashboard come un'icona cliccabile che si collega al testo di tutti i file. Per aggiungere le note, si chiama CTest con l'opzione -A seguita da un elenco di nomi di file separati da punto e virgola. Il contenuto di questi file verrà inviato come note per la dashboard. Per esempio:

```
ctest -D Continuous -A C:/MyNotes.txt;C:/OtherNotes.txt
```

Un altro modo per inviare note con una dashboard è copiare o scrivere le note come file in una directory Notes nella directory `Testing` dell'albero binario. Anche tutti i file trovati lì quando CTest invia una dashboard verranno caricati come note.

16.4 Script di CTest

Questa sezione descrive come scrivere script CTest basati su comandi che consentono al manutentore di avere un controllo granulare sui singoli passaggi di una dashboard.

Il manutentore della dashboard ha accesso alle singole funzioni di comando CTest, come `ctest_configure` e `ctest_build`. Eseguendo queste funzioni individualmente, l'utente può sviluppare in modo flessibile schemi di test personalizzati. Ecco un esempio di uno script CTest

```
cmake_minimum_required(VERSION 3.20)

set(CTEST_SITE          "andoria.kitware")
set(CTEST_BUILD_NAME    "Linux-g++")
set(CTEST_NOTES_FILES
    "${CTEST_SCRIPT_DIRECTORY}/${CTEST_SCRIPT_NAME}")

set(CTEST_DASHBOARD_ROOT   "$ENV{HOME}/Dashboards/My Tests")
set(CTEST_SOURCE_DIRECTORY "${CTEST_DASHBOARD_ROOT}/CMake")
set(CTEST_BINARY_DIRECTORY "${CTEST_DASHBOARD_ROOT}/CMake-gcc ")

set(CTEST_UPDATE_COMMAND    "/usr/bin/cvs")
set(CTEST_CONFIGURE_COMMAND
    "\"${CTEST_SOURCE_DIRECTORY}/bootstrap\"")
set(CTEST_BUILD_COMMAND     "/usr/bin/make -j 2")


ctest_empty_binary_directory(${CTEST_BINARY_DIRECTORY})

ctest_start(Nightly)
ctest_update(SOURCE "${CTEST_SOURCE_DIRECTORY}")
ctest_configure(BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_build(BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_test(BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_submit()
```

Il primo blocco contiene le variabili relative all'invio.

```
set(CTEST_SITE              "andoria.kitware")
set(CTEST_BUILD_NAME        "Linux-g++")
set(CTEST_NOTES_FILES
    "${CTEST_SCRIPT_DIRECTORY}/${CTEST_SCRIPT_NAME}")
```

Queste variabili vengono utilizzate per identificare il sistema una volta che invia i risultati alla dashboard. `CTEST_NOTES_FILES` è un elenco di file che devono essere inviati come le note dell'invio della dashboard. Questa variabile corrisponde al flag -A di `ctest`.

Il secondo blocco descrive le informazioni che le funzioni CTest utilizzeranno per eseguire le attività:

```
set(CTEST_DASHBOARD_ROOT   "$ENV{HOME}/Dashboards/My Tests")
set(CTEST_SOURCE_DIRECTORY "${CTEST_DASHBOARD_ROOT}/CMake")
set(CTEST_BINARY_DIRECTORY "${CTEST_DASHBOARD_ROOT}/CMake-gcc ")
set(CTEST_UPDATE_COMMAND    "/usr/bin/cvs")
set(CTEST_CONFIGURE_COMMAND
   "\"${CTEST_SOURCE_DIRECTORY}/bootstrap\"")
set(CTEST_BUILD_COMMAND     "/usr/bin/make -j 2")
```

CTEST_UPDATE_COMMAND è il path del comando utilizzato per aggiornare la directory dei sorgenti dal repository. Attualmente CTest supporta Concurrent Versions System (CVS), Subversion, Git, Mercurial e Bazaar.

Entrambi i gestori configure e build supportano due modalità Una modalità consiste nel fornire il comando completo che verrà richiamato durante quella fase. Questo è progettato per supportare progetti che non utilizzano CMake come strumento di configurazione o build. In questo caso, si devono specificare le righe di comando complete per configurare e creare il progetto impostando rispettivamente le variabili CTEST_CONFIGURE_COMMAND e CTEST_BUILD_COMMAND.

Per la fase di build si devono anche impostare le variabili CTEST_PROJECT_NAME e CTEST_BUILD_CONFIGURATION, per specificare come eseguire la build del progetto. In questo caso CTEST_PROJECT_NAME corrisponderà al comando project del file CMakeLists di primo livello. CTEST_BUILD_CONFIGURATION deve essere uno tra Release, Debug, MinSizeRel o RelWithDebInfo. Inoltre, CTEST_BUILD_FLAGS può essere fornito come suggerimento per il comando build. Un esempio di test per un progetto basato su CMake potrebbe essere:

```
set(CTEST_PROJECT_NAME "Grommit")
set(CTEST_BUILD_CONFIGURATION "Debug")
```

Il blocco finale esegue il test e l'invio effettivi:

```
ctest_empty_binary_directory(${CTEST_BINARY_DIRECTORY})

ctest_start(Nightly)

ctest_update(SOURCE
             "${CTEST_SOURCE_DIRECTORY}" RETURN_VALUE res)
ctest_configure(BUILD
                "${CTEST_BINARY_DIRECTORY}" RETURN_VALUE res)
ctest_build(BUILD "${CTEST_BINARY_DIRECTORY}" RETURN_VALUE res)
ctest_test(BUILD "${CTEST_BINARY_DIRECTORY}" RETURN_VALUE res)
ctest_submit(RETURN_VALUE res)
```

Il comando ctest_empty_binary_directory svuota la directory e tutte le sottodirectory. Si noti che questo comando ha una misura di sicurezza incorporata, ovvero rimuoverà la directory solo se è presente un file CMakeCache.txt nella directory di livello superiore. Questo aveva lo scopo di impedire a CTest di rimuovere erroneamente una directory non di build.

Il resto del blocco contiene le chiamate alle effettive funzioni CTest. Ciascuno di essi corrisponde a un'opzione CTest -D. Ad esempio, invece di:

```
ctest -D ExperimentalBuild
```

lo script conterrebbe:

```
ctest_start(Experimental)
ctest_build(BUILD "${CTEST_BINARY_DIRECTORY}" RETURN_VALUE res)
```

Ogni passaggio restituisce un valore di ritorno, che indica se il passaggio ha avuto esito positivo. Ad esempio, il valore restituito della fase di aggiornamento può essere utilizzato in una dashboard continua per determinare se il resto della dashboard deve essere eseguito.

Esaminiamo uno script CTest più avanzato. Questo script guida il test di un'applicazione chiamata Slicer. Slicer utilizza CMake internamente, ma guida il processo di build attraverso una serie di script Tcl. Uno dei problemi di questo approccio è che non supporta build «out-of-source». Inoltre, su Windows alcuni moduli sono precompilati, quindi devono essere copiati nella directory di build. Per testare un progetto del genere, useremmo uno script come questo:

```
cmake_minimum_required(VERSION 3.20)

# set the dashboard specific variables -- name and notes
set(CTEST_SITE              "dash11.kitware")
set(CTEST_BUILD_NAME        "Win32-VS71")
set(CTEST_NOTES_FILES
     "${CTEST_SCRIPT_DIRECTORY}/${CTEST_SCRIPT_NAME}")

# do not let any single test run for more than 1500 seconds
set(CTEST_TIMEOUT "1500")

# set the source and binary directories
set(CTEST_SOURCE_DIRECTORY  "C:/Dashboards/MyTests/slicer2")
set(CTEST_BINARY_DIRECTORY  "${CTEST_SOURCE_DIRECTORY}-build")

set (SLICER_SUPPORT
     "//Dash11/Shared/Support/SlicerSupport/Lib")
set (TCLSH   "${SLICER_SUPPORT}/win32/bin/tclsh84.exe")
# set the complete update, configure and build commands
set (CTEST_UPDATE_COMMAND
     "C:/Program Files/TortoiseCVS/cvs.exe")
set (CTEST_CONFIGURE_COMMAND
    "\"${TCLSH}\"
     \"${CTEST_BINARY_DIRECTORY}/Scripts/genlib.tcl\"")
set (CTEST_BUILD_COMMAND
    "\"${TCLSH}\"
     \"${CTEST_BINARY_DIRECTORY}/Scripts/cmaker.tcl\"")

# clear out the binary tree
file (WRITE "${CTEST_BINARY_DIRECTORY}/CMakeCache.txt"
      "// Dummy cache just so that ctest will wipe binary dir")
ctest_empty_binary_directory (${CTEST_BINARY_DIRECTORY})

# special variables for the Slicer build process
set (ENV{MSVC6}              "0")
set (ENV{GENERATOR}          "Visual Studio 7 .NET 2003")
set (ENV{MAKE}               "devenv.exe ")
set (ENV{COMPILER_PATH}
```

(continues on next page)

(continua dalla pagina precedente)

```
    "C:/Program Files/Microsoft Visual Studio .NET
2003/Common7/Vc7/bin")
set (ENV{CVS}                "${CTEST_UPDATE_COMMAND}")

# start and update the dashboard
ctest_start (Nightly)
ctest_update (SOURCE "${CTEST_SOURCE_DIRECTORY}")

# define a macro to copy a directory
macro (COPY_DIR srcdir destdir)
  exec_program ("${CMAKE_EXECUTABLE_NAME}" ARGS
                "-E copy_directory \"${srcdir}\" \"${destdir}\"")
endmacro ()

# Slicer does not support out of source builds so we
# first copy the source directory to the binary directory
# and then build it
copy_dir ("${CTEST_SOURCE_DIRECTORY}"
          "${CTEST_BINARY_DIRECTORY}")


# copy support libraries that slicer needs into the binary tree
copy_dir ("${SLICER_SUPPORT}"
          "${CTEST_BINARY_DIRECTORY}/Lib")

# finally do the configure, build, test and submit steps
ctest_configure (BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_build (BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_test (BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_submit ()
```

Con lo scripting CTest esteso abbiamo il pieno controllo del flusso, quindi possiamo eseguire comandi arbitrari in qualsiasi momento. Ad esempio, dopo aver eseguito un aggiornamento del progetto, lo script copia l'albero dei sorgenti nella directory di build. Ciò gli consente di eseguire una build «out-of-source».

16.4.1 Ruoli del progetto: CDash supporta tre livelli di ruolo per gli utenti

- Gli utenti normali sono utenti regolari con accesso in lettura e/o scrittura al repository del codice del progetto.
- I manutentori del sito sono responsabili degli invii periodici a CDash.
- Gli amministratori del progetto hanno privilegi riservati per amministrare il progetto in CDash.

I primi due livelli possono essere definiti dagli utenti stessi. L'accesso come amministratore del progetto deve essere concesso da un altro amministratore del progetto o da un amministratore del server CDash.

16.4.2 Backup degli Invii

CDash esegue il backup di tutti gli invii XML in arrivo e li inserisce nella directory `backup` per default. L'intervallo di tempo di default è di 48 ore. L'intervallo di tempo può essere modificato in `config.local.php` come segue:

```
$CDASH_BACKUP_TIMEFRAME=72;
```

Se i progetti sono privati, si consiglia di impostare la directory di backup al di fuori di quella principale di apache per assicurarsi che nessuno possa accedere ai file XML o di aggiungere le seguenti righe al file .htaccess nella directory di backup:

```
<Files *>
order allow,deny
deny from all
</Files>
```

Si noti che la directory di backup viene svuotata solo quando arriva un nuovo invio. Se necessario, CDash può anche importare build dalla directory di backup.

16.4.3 Gruppi di Build

Le build possono essere organizzate per gruppi. In CDash, tre gruppi vengono definiti automaticamente e non possono essere rimossi: `Nightly`, `Continuous` e `Experimental`. Questi gruppi sono gli stessi imposti da CTest. Ogni gruppo ha una descrizione associata che viene visualizzata quando si fa clic sul nome del gruppo nella dashboard principale.

Per default, una build appartiene al gruppo associato al tipo di build definito da CTest, ovvero una build notturna andrà nella sezione Nightly. CDash abbina una build in base al nome, al sito e al tipo di build. Ad esempio, una build notturna denominata «Linux-gcc-4.3» dal sito «midworld.kitware» verrà spostata nella sezione Nightly a meno che non sia presente una regola su «Linux-gcc-4.3»-«midworld.kitware»-«Nightly». Esistono due modi per spostare una build in un determinato gruppo definendo una regola: Global Move e Single Move.

La single move consente di modificare solo una particolare build

Se si è effettuato l'accesso come amministratore del progetto, viene visualizzata una piccola icona di cartella accanto a ciascuna build nella pagina principale della dashboard. Facendo clic sull'icona vengono visualizzate alcune opzioni per ciascuna build. In particolare, gli amministratori del progetto possono contrassegnare una build come prevista, spostare una build in un gruppo specifico o eliminare una build bacata.

Build previste [expected]: Gli amministratori del progetto possono contrassegnare determinate build come previste. Ciò significa che le build devono essere inviate quotidianamente. Questo consente di verificare rapidamente se una build non è stata inviata nella dashboard odierna o di valutare rapidamente da quanto tempo manca la build facendo clic sull'icona delle informazioni sulla dashboard principale.

Se una build «expected» non è stata inviata il giorno precedente e l'opzione «Email Build Missing» è selezionata per il progetto, verrà inviata un'e-mail al manutentore del sito e all'amministratore del progetto per avvisarli (vedere la sezione Siti per maggiori informazioni).

16.4.4 Email

CDash invia e-mail agli sviluppatori e agli amministratori di progetto quando si verifica un errore per una determinata build. La configurazione della funzione email si trova in tre posizioni: il file `config.local.php`, la sezione di amministrazione del progetto e la sezione dei gruppi del progetto.

In `config.local.php`, sono definite due variabili per specificare l'indirizzo e-mail da cui viene inviata l'e-mail e l'indirizzo di risposta. Si noti che il server SMTP non può essere definito nella versione corrente di CDash, si presume che sulla macchina sia in esecuzione un server di posta locale.

```
$CDASH_EMAIL_FROM = 'admin@mywebsite.com';
$CDASH_EMAIL_REPLY = 'noreply@mywebsite.com';
```

Nella sezione di configurazione della posta elettronica del progetto, è possibile regolare diversi parametri per controllare la funzione di posta elettronica.

Nella sezione di amministrazione «build groups» di un progetto, un amministratore può decidere se inviare le email a un gruppo specifico o se inviare solo un'email di riepilogo. L'e-mail di riepilogo viene inviata per un determinato gruppo quando almeno una build non riesce nel giorno corrente.

16.4.5 Siti

CDash si riferisce a un sito come a una singola macchina che invia almeno una build a un determinato progetto. Un sito potrebbe inviare più build (ad esempio notturne e continue) a più progetti archiviati in CDash.

Per visualizzare la descrizione del sito, fare clic sul nome del sito dalla pagina principale della dashboard per un progetto. La descrizione di un sito include informazioni riguardanti il tipo e la velocità del processore, nonché la quantità di memoria disponibile sulla macchina in questione. La descrizione di un sito viene inviata automaticamente da CTest, tuttavia in alcuni casi potrebbe essere necessario modificarla manualmente. Inoltre, se la macchina viene aggiornata, cioè la memoria viene aggiornata; CDash tiene traccia della cronologia della descrizione, consentendo agli utenti di confrontare le prestazioni prima e dopo l'aggiornamento.

I siti di solito appartengono a un manutentore, responsabile degli invii a CDash. È importante che i manutentori del sito vengano avvisati quando un sito non viene inviato poiché potrebbe essere correlato a un problema di configurazione.

Una volta rivendicato un sito, il suo manutentore riceverà una e-mail se il computer client non invia per un motivo sconosciuto, supponendo che il sito debba essere inviato ogni notte. Inoltre, il sito comparirà nella sezione «My Sites» del profilo del manutentore, facilitando una rapida verifica dello stato del sito.

Un'altra caratteristica della pagina del sito è il grafico a torta che mostra il carico della macchina. Supponendo che un sito invii a più progetti, di solito è utile sapere se la macchina ha spazio per altri invii a CDash. Il grafico a torta fornisce una panoramica del tempo di invio della macchina per ciascun progetto.

16.4.6 Grafici

CDash disegna attualmente tre tipi di grafici. I grafici sono generati dinamicamente dai record del database e sono interattivi.

Il grafico del tempo di build mostra il tempo necessario per buildare un progetto nel tempo.

I grafici del tempo del test mostrano il tempo necessario per eseguire un test specifico e il suo stato (superato/fallito) nel tempo.

16.4.7 Aggiunta di note a una build

In alcuni casi, è utile informare altri sviluppatori che qualcuno sta esaminando gli errori di una build.

16.4.8 Logging

CDash supporta un meccanismo di log interno che usa la funzione PHP error_log(). Tutti gli errori SQL critici vengono loggati. Per default, il file di log di CDash si trova nella directory di backup con il nome `cdash.log`. La posizione del file di log può essere modificata cambiando la variabile nel file di configurazione `config.local.php`.

```
$CDASH_BACKUP_DIRECTORY='/var/temp/cdashbackup/log';
```

È possibile accedere al file di log direttamente da CDash se il file si trova nella posizione standard.

È disponibile un meccanismo di rotazione del log che consente agli utenti di limitare il file di log corrente a una determinata dimensione.

16.4.9 Test dei Tempi

CDash supporta i controlli sulla durata dei test. CDash mantiene, nel database, la media ponderata corrente della media e della deviazione standard per il tempo impiegato da ciascun test per essere eseguito. Per mantenere il calcolo il più efficiente possibile viene utilizzata la seguente formula, che coinvolge solo la build precedente.

```
// alpha is the current "window" for the computation
// By default, alpha is 0.3
newMean = (1-alpha)*oldMean + alpha*currentTime

newSD = sqrt((1-alpha)*SD*SD +
  alpha*(currentTime-newMean)*(currentTime-newMean)
```

Un test è definito come temporizzazione fallita in base alla seguente logica:

```
if previousSD < thresholdSD then previousSD = thresholdSD
if currentTime > previousMean+multiplier*previousSD then fail
```

16.4.10 Supporto Mobile

Dal momento che CDash è scritto utilizzando layer di template tramite XSLT, sviluppare nuovi layout è semplice come aggiungere nuovi template di rendering. A titolo dimostrativo, viene fornito un template Web per iPhone con la versione corrente di CDash.

```
http://mycdashserver/CDash/iphone
```

La pagina principale mostra un elenco dei progetti pubblici ospitati sul server. Facendo clic sul nome di un progetto viene caricata la sua dashboard corrente. Allo stesso modo, clickando su una determinata build vengono visualizzate informazioni più dettagliate su di essa. Al momento della stesura di questo documento, la possibilità di effettuare il login e di accedere alle sezioni private di CDash non è supportata con questo layout.

16.4.11 Backup di CDash

Tutti i dati (tranne i log) utilizzati da CDash sono archiviati nel suo database. È importante eseguire regolarmente il backup del database, soprattutto prima di eseguire un aggiornamento di CDash. Ci sono un paio di modi per eseguire il backup di un database MySQL. Il più semplice è utilizzare il comando mysqldump:

```
mysqldump -r cdashbackup.sql cdash
```

Se si usano esclusivamente tabelle MyISAM, si può copiare la directory CDash nella directory dei dati MySQL. Si noti che è necessario effettuare lo shutdown di MySQL prima di eseguire la copia in modo che nessun file possa essere modificato durante la copia. Analogamente a MySQL, PostGreSQL ha un'utilità pg_dump:

```
pg_dump -U posgreSQL_user cdash > cdashbackup.sql
```

16.4.12 Aggiornamento di CDash

Quando viene rilasciata una nuova versione di CDash o si decide di aggiornare dal repository, CDash avviserà in prima pagina se il database corrente deve essere aggiornato. Quando si esegue l'aggiornamento a una nuova versione di rilascio, è necessario eseguire i seguenti passaggi:

1. Eseguire il backup del database SQL (vedere la sezione precedente).
2. Eseguire il backup dei file di configurazione `config.local.php` (o `config.php`).
3. Sostituisci l'attuale directory cdash con l'ultima versione e copiare `config.local.php` nella directory di cdash.
4. Naviga nel browser fino alla pagina CDash. (ad esempio http://localhost/CDash).
5. Annotare il numero di versione sulla pagina principale, dovrebbe corrispondere a quella a cui si sta effettuando l'aggiornamento.
6. Potrebbe apparire il seguente messaggio: «The current database schema doesn't match the version of CDash you are running, upgrade your database structure in the Administration panel of CDash». Questo è un utile promemoria per eseguire i seguenti passaggi.
7. Accedere a CDash come amministratore.
8. Nella sezione "Administration", clickare su "[CDash Maintenance]".
9. Clickare su "Upgrade CDash": questo processo potrebbe richiedere del tempo a seconda delle dimensioni del database (non chiudere il browser).
    - I messaggi di avanzamento possono essere visualizzati mentre CDash esegue l'aggiornamento.
    - Se il processo di aggiornamento richiede troppo tempo, si può controllare nel file `backup/cdash.log` per vedere dove il processo impiega molto tempo e/o fallisce.

- È stato segnalato che su alcuni sistemi l'icona rotante non si trasforma mai in un segno di spunta. Controllare `cdash.log` per la stringa «Upgrade done.» se si ritiene che l'aggiornamento richieda troppo tempo.
- Su un database da 50 GB l'aggiornamento potrebbe richiedere fino a 2 ore.

10. Alcuni browser Web potrebbero avere problemi durante l'aggiornamento (con alcune variabili javascript non passate correttamente), in tal caso è possibile eseguire singoli aggiornamenti. Ad esempio, l'aggiornamento da CDash 1-2 a 1-4:

```
http://mywebsite.com/CDash/backwardCompatibilityTools.php?upgrade-
↪1-4=1
```

16.4.13 Manutenzione di CDash

Manutenzione del database: si consiglia di eseguire regolarmente l'ottimizzazione del database (re-indicizzazione, purging, ecc.) per mantenere un database stabile. MySQL ha un'utilità chiamata `mysqlcheck` e PostgreSQL ha diverse utilità come `vacuumdb`.

Eliminazione di build con date errate: alcune build potrebbero essere inviate a CDash con la data errata, o perché la data nel file XML non è corretta o perché il fuso orario non è stato riconosciuto da CDash (principalmente da PHP). Queste build non verranno visualizzate in nessuna dashboard perché l'ora di inizio è bacata. Per rimuovere queste build:

1. Accedere a CDash come amministratore.
2. Clickare su [CDash maintenance] nella sezione administration.
3. Clickare su "Delete builds with wrong start date".

Ricalcolare i tempi del test: se è stato appena aggiornato CDash si potrebbero notare che gli invii correnti mostrano un numero elevato di test falliti a causa di un tempo errato. Questo perché CDash non ha abbastanza punti campione per calcolare la media e la deviazione standard per ogni test, in particolare la deviazione standard potrebbe essere molto piccola (probabilmente zero per i primi campioni). Si deve disattivare «enable test timing» per circa una settimana, o fino a quando non si ricevono un numero sufficiente di invii di build e CDash ha calcolato una media approssimativa e una deviazione standard per ogni tempo di test.

L'altra opzione è quella di forzare CDash a calcolare la media e la deviazione standard per ogni test degli ultimi giorni. Si tenga presente che questo processo potrebbe richiedere molto tempo, a seconda del numero di test e progetti coinvolti. Per ricalcolare i tempi del test:

1. Accedere a CDash come amministratore.
2. Clickare su [CDash maintenance] nella sezione administration.
3. Specificare il numero di giorni (il default è 4) per ricalcolare i tempi del test.
4. Clickare su «Compute test timing». Al termine del processo, la nuova media, la deviazione standard e lo stato devono essere aggiornati per i test inviati durante questo periodo.

Rimozione automatica della build

Per mantenere il database a dimensioni ragionevoli, CDash può eliminare automaticamente le vecchie build. Attualmente ci sono due modi per impostare la rimozione automatica delle build: senza un cronjob, si modifica `config.local.php` e si aggiunge/modifica la riga seguente

```
$CDASH_AUTOREMOVE_BUILDS='1';
```

CDash rimuoverà automaticamente le build al primo invio della giornata. Si noti che la rimozione delle build potrebbe aggiungere un carico aggiuntivo al database o rallentare l'attuale processo di invio se il database è grande e il numero di invii è elevato. Se possibile utilizzare un cronjob, lo strumento della riga di comando PHP può essere utilizzato per attivare le rimozioni di build in un momento opportuno. Ad esempio, rimuovendo le build per tutti i progetti ogni domenica alle 6:00:

```
0 6 * * 0 php5 /var/www/CDash/autoRemoveBuilds.php all
```

Si noti che il parametro "all" può essere modificato in un nome di progetto specifico per eliminare le build da un singolo progetto.

Schema XML di CDash

I parser XML in CDash possono essere facilmente estesi per supportare nuove funzionalità. Gli attuali schemi XML generati da CTest e le loro funzionalità descritte nel libro si trovano in:

```
http://public.kitware.com/Wiki/CDash:XML
```

16.4.14 Sottoprogetti

CDash supporta la suddivisione dei progetti in sottoprogetti. Alcuni dei sottoprogetti possono a loro volta dipendere da altri sottoprogetti. Un tipico progetto reale è costituito da librerie, eseguibili, test suite, documentazione, pagine Web e programmi di installazione. Organizzare il progetto in sottoprogetti ben definiti e presentare i risultati delle build notturne su un dashboard CDash può aiutare a identificare dove si trovano i problemi a diversi livelli di granularità.

Un progetto con sottoprogetti ha una vista diversa per la sua pagina CDash di primo livello rispetto a un progetto senza. Contiene una riga di riepilogo per il progetto nel suo insieme, quindi una riga di riepilogo per ogni sottoprogetto.

Organizzazione e definizione di sottoprogetti

Per aggiungere l'organizzazione del sottoprogetto al progetto, si deve: (1) definire i sottoprogetti per CDash, in modo che sappia come visualizzarli correttamente e (2) utilizzare gli script di build con CTest per inviare le build del sottoprogetto. Potrebbe anche essere necessaria una (ri)organizzazione dei file CMakeLists.txt del progetto per consentire la creazione del progetto per sottoprogetti.

Esistono due modi per definire i sottoprogetti e le loro dipendenze: in modo interattivo nella GUI di CDash quando si accede come amministratore del progetto o inviando un file `Project.xml` che descrive i sottoprogetti e le dipendenze.

Aggiunta Interattiva di Sottoprogetti

In qualità di amministratore del progetto, apparirà un pulsante «Manage subprojects» per ciascuno dei progetti nella pagina My CDash. Clickando sul pulsante Manage Subprojects si apre la pagina di gestione dei sottoprogetti, dove si possono aggiungere nuovi sottoprogetti o stabilire dipendenze tra sottoprogetti esistenti per qualsiasi progetto di cui si è amministratore.

Aggiunta automatica di sottoprogetti

Un altro modo per definire i sottoprogetti CDash e le loro dipendenze è quello di inviare un file «Project.xml» insieme ai normali spediti da CTest quando invia una build a CDash. Per definire gli stessi due sottoprogetti dell'esempio interattivo sopra (Exes e Libs) con la stessa dipendenza (gli Exes dipendono da Libs), il file `Project.xml` dovrebbe avere il seguente esempio:

```
<Project name="Tutorial">
  <SubProject name="Libs"></SubProject>
  <SubProject name="Exes">
    <Dependency name="Libs">
  </SubProject>
</Project>
```

Una volta che il file `Project.xml` è stato scritto o generato, può essere inviato a CDash da uno script ctest -S usando l'argomento FILES al comando `ctest_submit`, oppure direttamente dalla riga di comando `ctest` in un albero di build configurato per l'invio della dashboard.

Dall'interno di uno script ctest -S:

```
ctest_submit(FILES "${CTEST_BINARY_DIRECTORY}/Project.xml")
```

Dalla riga di comando:

```
cd ../Project-build
ctest --extra-submit Project.xml
```

CDash aggiungerà automaticamente sottoprogetti e dipendenze in base al file `Project.xml`. CDash rimuoverà anche eventuali sottoprogetti o dipendenze non definiti nel file `Project.xml`. Inoltre, se lo stesso `Project.xml` viene inviato più volte, il secondo e i successivi invii

non avranno alcun effetto osservabile: il primo invio aggiunge/modifica i dati, il secondo e i successivi invii inviano gli stessi dati, quindi non sono necessarie modifiche. CDash tiene traccia delle modifiche alle definizioni dei sottoprogetti nel tempo per consentire l'evoluzione dei progetti. Se si visualizzano le dashboard da una data passata, CDash presenterà le viste del progetto/sottoprogetto in base alle definizioni del sottoprogetto in vigore in quella data.

16.4.15 Uso di ctest_submit con PARTS e FILES

Il comando `ctest_submit` supporta gli argomenti `PARTS` e `FILES`. Con `PARTS`, si può inviare qualsiasi sottoinsieme dei file xml con ogni chiamata `ctest_submit`. Per default, tutte le parti disponibili vengono inviate con qualsiasi chiamata a `ctest_submit`. Uno script può attendere il completamento di tutte le fasi della dashboard e poi chiamare `ctest_submit` una volta per inviare i risultati di tutte le fasi alla fine dell'esecuzione. In alternativa, uno script può chiamare `ctest_submit` con `PARTS` per eseguire invii parziali di sottoinsiemi dei risultati. Ad esempio, si possono inviare i risultati di configurazione dopo `ctest_configure`, i risultati di build dopo `ctest_build` e i risultati di test dopo `ctest_test`. Ciò consente di pubblicare informazioni man mano che le build progrediscono.

Con `FILES`, si possono inviare file XML arbitrari a CDash. Oltre ai file XML dei risultati di build standard inviati da CTest, CDash gestisce anche un file `Project.xml` che descrive sottoprogetti e dipendenze. Di seguito è riportato un esempio di uno script di dashboard che contiene una singola chiamata `ctest_submit` nell'ultima riga

```
ctest_start(Experimental)
ctest_update(SOURCE "${CTEST_SOURCE_DIRECTORY}")
ctest_configure(BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_build(BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_test(BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_submit()
```

Gli invii possono avvenire in modo incrementale, con ciascuna parte dell'invio inviata in modo frammentario non appena diventa disponibile:

```
ctest_start(Experimental)
ctest_update(SOURCE "${CTEST_SOURCE_DIRECTORY}")
ctest_configure(BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_submit(PARTS Update Configure Notes)

ctest_build(BUILD "${CTEST_BINARY_DIRECTORY}" APPEND)
ctest_submit(PARTS Build)

ctest_test(BUILD "${CTEST_BINARY_DIRECTORY}")
ctest_submit(PARTS Test)
```

L'invio incrementale per parti significa che si possono ispezionare i risultati della fase di configurazione in tempo reale sulla dashboard di CDash mentre la build è ancora in corso. Allo stesso modo, si possono controllare i risultati della fase di build dal vivo mentre i test sono ancora in esecuzione. Quando si invia per parti, è importante utilizzare la parola chiave `APPEND` nel

comando ctest_build. Se non si usa APPEND, CDash cancellerà qualsiasi build esistente con lo stesso nome della build, nome del sito e [stamp] della build quando riceve il file Build.xml.

16.4.16 Suddivisione del Progetto in Più Sottoprogetti

Un'invocazione di ctest_build che compila tutto, seguita da un'invocazione di ctest_test che verifica tutto è sufficiente per un progetto che non ha sottoprogetti, ma per inviare i risultati in base al sottoprogetto a CDash, sarà necessario apportare alcune modifiche al progetto e agli script di test. Per il progetto si deve identificare quali target fanno parte di quali sottoprogetti. Se si organizzano i file CMakeLists in modo tale da avere un target da creare per ogni sottoprogetto e si può derivare (o cercare) il nome di tale target in base al nome del sottoprogetto, rivedere lo script per separarlo in più configurazioni/build/test più piccoli dovrebbero essere relativamente semplice. Per fare ciò, si possono modificare i file CMakeLists in vari modi a seconda delle esigenze. Le modifiche più comuni sono elencate di seguito.

Modifiche a CMakeLists.txt

- Chiamare i target con lo stesso nome dei sottoprogetti, basare i nomi dei target sui nomi dei sottoprogetti o fornire un meccanismo di ricerca per mappare dal nome del sottoprogetto al nome del target.
- Possibilmente aggiungere target custom per aggregare target esistenti in sottoprogetti, usando add_dependencies per dire da quali target esistenti dipende il target custom.
- Aggiungere la proprietà LABELS col nome del sottoprogetto come valore ai target.
- Aggiungere la proprietà LABELS col nome del sottoprogetto come valore ai test.

Successivamente, è necessario modificare gli script CTest che eseguono le dashboard. Per suddividere la build monolitica di grandi dimensioni in build di sottoprogetti più piccole, si può utilizzare un ciclo foreach nello script del driver CTest. Per scorrere i sottoprogetti, CDash fornisce una variabile denominata CTEST_PROJECT_SUBPROJECTS in CTestConfig.cmake. Dato l'esempio sopra, CDash produce una variabile come questa:

```
set(CTEST_PROJECT_SUBPROJECTS Libs Exes)
```

CDash ordina gli elementi in questo elenco in modo tale che i sottoprogetti indipendenti (che non dipendono da altri sottoprogetti) siano i primi, seguiti dai sottoprogetti che dipendono solo dai sottoprogetti indipendenti e quindi dai sottoprogetti che dipendono da questi. La stessa logica continua finché tutti i sottoprogetti non vengono elencati esattamente una volta in questo elenco in un ordine che abbia senso per la loro creazione in sequenza, uno dopo l'altro.

Per facilitare la build solo dei target associati a un sottoprogetto, si usa la variabile chiamata CTEST_BUILD_TARGET per dire a ctest_build cosa buildare. Per facilitare l'esecuzione dei soli test associati a un sottoprogetto, si assegna la proprietà test LABELS ai test e si usa l'argomento INCLUDE_LABEL per ctest_test.

Modifiche allo script del driver ctest

- Scorrere i sottoprogetti in ordine di dipendenza (dall'indipendente al più dipendente...).
- Impostare le proprietà globali SubProject e Label: CTest utilizza queste proprietà per inviare i risultati al sottoprogetto corretto sul server CDash.
- Creare il/i target per questo sottoprogetto: calcolare il nome del target da compilare dal nome del sottoprogetto, impostare `CTEST_BUILD_TARGET`, chiamare `ctest_build`.
- Eseguire i test per questo sottoprogetto utilizzando gli argomenti `INCLUDE` o `INCLUDE_LABEL` in `ctest_test`.
- Usare `ctest_submit` con l'argomento `PARTS` per inviare risultati parziali man mano che vengono completati.

Per illustrare ciò, l'esempio seguente mostra le modifiche necessarie per suddividere una build in parti più piccole. Si supponga che il nome del sottoprogetto sia lo stesso del nome del target richiesto per creare i componenti del sottoprogetto. Ad esempio, ecco uno snippet da CMakeLists.txt, nell'ipotetico progetto Tutorial. Le uniche aggiunte necessarie (poiché i nomi dei target sono gli stessi dei nomi dei sottoprogetti) sono le chiamate a `set_property` per ogni target e ogni test.

```
# "Libs" is the library name (therefore a target name) and
# the subproject name
add_library (Libs ...)
set_property (TARGET Libs PROPERTY LABELS Libs)
add_test (LibsTest1 ...)
add_test (LibsTest2 ...)
set_property (TEST LibsTest1 LibsTest2 PROPERTY LABELS Libs)

# "Exes" is the executable name (therefore a target name)
# and the subproject name
add_executable (Exes ...)
target_link_libraries (Exes Libs)
set_property (TARGET Exes PROPERTY LABELS Exes)
add_test (ExesTest1 ...)
add_test (ExesTest2 ...)
set_property (TEST ExesTest1 ExesTest2 PROPERTY LABELS Exes)
```

Ecco un esempio di come potrebbe apparire lo script del driver CTest prima e dopo l'organizzazione di questo progetto in sottoprogetti. Prima delle modifiche

```
ctest_start (Experimental)
ctest_update (SOURCE "${CTEST_SOURCE_DIRECTORY}")
ctest_configure (BUILD "${CTEST_BINARY_DIRECTORY}")
# builds *all* targets: Libs and Exes
ctest_build (BUILD "${CTEST_BINARY_DIRECTORY}")
# runs *all* tests
ctest_test (BUILD "${CTEST_BINARY_DIRECTORY}")
```

(continues on next page)

(continua dalla pagina precedente)

```
# submits everything all at once at the end
ctest_submit ()
```

Dopo le modifiche:

```
ctest_start (Experimental)
ctest_update (SOURCE "${CTEST_SOURCE_DIRECTORY}")
ctest_submit (PARTS Update Notes)

# to get CTEST_PROJECT_SUBPROJECTS definition:
include ("${CTEST_SOURCE_DIRECTORY}/CTestConfig.cmake")

foreach (subproject ${CTEST_PROJECT_SUBPROJECTS})
  set_property (GLOBAL PROPERTY SubProject ${subproject})
  set_property (GLOBAL PROPERTY Label ${subproject})

  ctest_configure (BUILD "${CTEST_BINARY_DIRECTORY}")
  ctest_submit (PARTS Configure)

  set (CTEST_BUILD_TARGET "${subproject}")
  ctest_build (BUILD "${CTEST_BINARY_DIRECTORY}" APPEND)
    # builds target ${CTEST_BUILD_TARGET}
  ctest_submit (PARTS Build)

  ctest_test (BUILD "${CTEST_BINARY_DIRECTORY}"
    INCLUDE_LABEL "${subproject}"
  )
# runs only tests that have a LABELS property matching
# "${subproject}"
  ctest_submit (PARTS Test)
endforeach ()
```

In alcuni progetti, potrebbe essere necessario più di un passo `ctest_build` per buildare tutti i pezzi del sottoprogetto. Ad esempio, in Trilinos, ogni sottoprogetto compila il `${subproject}_libs target`, poi compila il target all per compilare tutti gli eseguibili configurati nella suite di test. Inoltre configurano le dipendenze in modo tale che solo gli eseguibili che devono essere compilati per i pacchetti attualmente configurati vengano compilati quando viene compilato il target all.

Normalmente, se si inviano più file `Build.xml` a CDash con lo stesso [stamp] della build, eliminerà la voce esistente e aggiungerà la nuova voce al suo posto. Nel caso in cui siano richiesti più passaggi `ctest_build`, ciascuno con la propria chiamata `ctest_submit(PARTS Build)`, utilizzare l'argomento chiave `APPEND` in tutte le chiamate `ctest_build` che risultano raggruppate. Il flag `APPEND` indica a CDash di accumulare i risultati di più invii e di visualizzare l'aggregazione di tutti in una riga sulla dashboard. Dal punto di vista di CDash, più chiamate a `ctest_build` (con lo [stamp] di build e sottoprogetto e `APPEND` attivato) risultano in una singola build di CDash.

Adottare alcuni di questi suggerimenti e tecniche nel progetto basato su CMake:

- `LABELS` è una proprietà CMake/CTest che si applica a file sorgenti, target e test. Le etichette vengono inviate a CDash all'interno dei file xml risultanti.
- Usare `ctest_submit(PARTS ...)` per effettuare invii incrementali. I risultati sono disponibili per la visualizzazione prima sulle dashboard. Non dimenticare di usare `APPEND` nelle chiamate a `ctest_build` quando si invia per parti.
- Usare `INCLUDE_LABEL` con `ctest_test` per eseguire solo i test con etichette che corrispondono all'espressione regolare.
- Usare `CTEST_BUILD_TARGET` per la build dei sottoprogetti uno alla volta, inviando le dashboard dei sottoprogetti man mano.

CAPITOLO 17

Tutorial di CMake

17.1 Introduzione

I tutorial su CMake fornisce una guida passo passo che copre i problemi comuni del sistema di build che CMake aiuta a risolvere. Vedere come i vari argomenti lavorano insieme in un progetto di esempio può essere molto utile. La documentazione del tutorial e il codice sorgente degli esempi si trovano nella directory `Help/guide/tutorial` dell'albero del codice sorgente di CMake. Ogni passaggio ha la propria sottodirectory contenente il codice utilizzabile come punto di partenza. Gli esempi del tutorial sono progressivi in modo che ogni passaggio fornisca la soluzione completa per il passaggio precedente.

17.2 Un Primo Punto di Partenza (Step 1)

Il progetto più elementare è un eseguibile creato da file di codice sorgente. Per progetti semplici, tutto ciò che serve è un file `CMakeLists.txt`. Questo sarà il punto di partenza del tutorial. Creare un file `CMakeLists.txt` nella directory `Step1` che assomigli a:

```
cmake_minimum_required(VERSION 3.10)

# set the project name
project(Tutorial)

# add the executable
add_executable(Tutorial tutorial.cxx)
```

Notare che questo esempio utilizza comandi in minuscolo nel file `CMakeLists.txt`. I comandi maiuscoli, minuscoli e misti sono supportati da CMake. Il codice sorgente per `tutorial.cxx` è fornito nella directory `Step1` ed è utilizzabile per calcolare la radice quadrata di un numero.

17.2.1 Aggiunta di un Numero di Versione e di un File di Header di Configurazione

La prima funzionalità che aggiungeremo è fornire all'eseguibile e al progetto un numero di versione. Sebbene potremmo farlo esclusivamente nel codice sorgente, l'utilizzo di `CMakeLists.txt` offre maggiore flessibilità.

Innanzitutto, si modifica il file `CMakeLists.txt` per utilizzare il comando `project` per impostare il nome del progetto e il numero della versione.

```
cmake_minimum_required(VERSION 3.10)

# set the project name and version
project(Tutorial VERSION 1.0)
```

Poi, si configura un file header per passare il numero di versione al codice sorgente:

```
configure_file(TutorialConfig.h.in TutorialConfig.h)
```

Poiché il file configurato verrà scritto nell'albero binario, dobbiamo aggiungere quella directory all'elenco dei percorsi in cui cercare i file include. Si aggiungono le seguenti righe alla fine del file `CMakeLists.txt`:

```
target_include_directories(Tutorial PUBLIC
                           "${PROJECT_BINARY_DIR}"
                           )
```

Utilizzando un editor, si crea `TutorialConfig.h.in` nella directory sorgente con il seguente contenuto:

```
// the configured options and settings for Tutorial
#define Tutorial_VERSION_MAJOR @Tutorial_VERSION_MAJOR@
#define Tutorial_VERSION_MINOR @Tutorial_VERSION_MINOR@
```

Quando CMake configura questo file header, i valori per `@Tutorial_VERSION_MAJOR@` e `@Tutorial_VERSION_MINOR@` verranno rimpiazzati.

Successivamente si modifica `tutorial.cxx` per includere il file header configurato , `TutorialConfig.h`.

Infine, stampiamo il nome dell'eseguibile e il numero della versione aggiornando `tutorial.cxx` come segue:

```
  if (argc < 2) {
    // report version
```

(continues on next page)

(continua dalla pagina precedente)

```
    std::cout << argv[0] << " Version " << Tutorial_VERSION_MAJOR << ".
↪"
              << Tutorial_VERSION_MINOR << std::endl;
    std::cout << "Usage: " << argv[0] << " number" << std::endl;
    return 1;
  }
```

17.2.2 Specificare lo standard del C++

Successivamente aggiungiamo alcune funzionalità del C++11 al progetto sostituendo `atof` con `std::stod` in `tutorial.cxx`. Contemporaneamente si rimuove `#include <cstdlib>`.

```
  const double inputValue = std::stod(argv[1]);
```

Dovremo dichiarare esplicitamente nel codice CMake che dovrebbe utilizzare i flag corretti. Il modo più semplice per abilitare il supporto per uno standard C++ specifico in CMake è utilizzare la variabile `CMAKE_CXX_STANDARD`. Per questo tutorial, si imposta la variabile `CMAKE_CXX_STANDARD` nel file `CMakeLists.txt` a 11 e `CMAKE_CXX_STANDARD_REQUIRED` a True. Si devono aggiungere le dichiarazioni `CMAKE_CXX_STANDARD` al di sopra della chiamata a `add_executable`.

```
cmake_minimum_required(VERSION 3.10)

# set the project name and version
project(Tutorial VERSION 1.0)

# specify the C++ standard
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)
```

17.2.3 Build e Test

Lanciare l'eseguibile `cmake` o `cmake-gui` per configurare il progetto e quindi crearlo col tool di build scelto.

Ad esempio, dalla riga di comando potremmo navigare nella directory `Help/guide/tutorial` dell'albero del codice sorgente di CMake e creare una directory di build:

```
mkdir Step1_build
```

Successivamente, si va nella directory di build e si esegue CMake per configurare il progetto e generare un sistema di build nativo:

```
cd Step1_build
cmake ../Step1
```

Poi si chiama quel sistema di build per compilare/linkare effettivamente il progetto:

```
cmake --build .
```

Infine, provare a utilizzare il `Tutorial` appena creato con questi comandi:

```
Tutorial 4294967296
Tutorial 10
Tutorial
```

17.3 Aggiunta di una libreria (Step 2)

Ora aggiungeremo una libreria al progetto Questa libreria conterrà l'implementazione per calcolare la radice quadrata di un numero. L'eseguibile può quindi utilizzare questa libreria invece della funzione radice quadrata standard fornita dal compilatore.

Per questo tutorial inseriremo la libreria in una sottodirectory chiamata `MathFunctions`. Questa directory contiene già un file di header, `MathFunctions.h`, e un file sorgente `mysqrt.cxx`. Il file sorgente ha una funzione chiamata `mysqrt` che fornisce funzionalità simili alla funzione `sqrt` del compilatore.

Si aggiunge il seguente file `CMakeLists.txt` alla directory `MathFunctions`:

```
add_library(MathFunctions mysqrt.cxx)
```

Per utilizzare la nuova libreria aggiungeremo una chiamata `add_subdirectory` nel file di livello superiore `CMakeLists.txt` in modo che la libreria venga creata. Aggiungiamo la nuova libreria all'eseguibile e aggiungiamo `MathFunctions` come directory di inclusione in modo che sia raggiungibile il file header `mysqrt.h`. Le ultime righe del file di livello superiore `CMakeLists.txt` dovrebbero ora assomigliare a:

```
# add the MathFunctions library
add_subdirectory(MathFunctions)

# add the executable
add_executable(Tutorial tutorial.cxx)

target_link_libraries(Tutorial PUBLIC MathFunctions)

# add the binary tree to the search path for include files
# so that we will find TutorialConfig.h
target_include_directories(Tutorial PUBLIC
                           "${PROJECT_BINARY_DIR}"
                           "${PROJECT_SOURCE_DIR}/MathFunctions"
                           )
```

Rendiamo ora facoltativa la libreria MathFunctions. Mentre per il tutorial non ce n'è realmente bisogno, per progetti più grandi questo è un evento comune. Il primo passo consiste nell'aggiungere un'opzione al file `CMakeLists.txt` di livello superiore.

```
option(USE_MYMATH "Use tutorial provided math implementation" ON)

# configure a header file to pass some of the CMake settings
# to the source code
configure_file(TutorialConfig.h.in TutorialConfig.h)
```

Questa opzione verrà visualizzata in `cmake-gui` e `ccmake` con un valore predefinito di ON modificabile dall'utente. Questa impostazione verrà archiviata nella cache in modo che l'utente non debba impostare il valore ogni volta che esegue CMake su una directory di build.

La modifica successiva consiste nel rendere condizionale la creazione e il link della libreria MathFunctions. Per fare ciò modifichiamo la fine del file di livello superiore `CMakeLists.txt` in modo che assomigli al seguente:

```
if(USE_MYMATH)
  add_subdirectory(MathFunctions)
  list(APPEND EXTRA_LIBS MathFunctions)
  list(APPEND EXTRA_INCLUDES "${PROJECT_SOURCE_DIR}/MathFunctions")
endif()

# add the executable
add_executable(Tutorial tutorial.cxx)

target_link_libraries(Tutorial PUBLIC ${EXTRA_LIBS})

# add the binary tree to the search path for include files
# so that we will find TutorialConfig.h
target_include_directories(Tutorial PUBLIC
                           "${PROJECT_BINARY_DIR}"
                           ${EXTRA_INCLUDES}
                           )
```

Da notare l'uso della variabile `EXTRA_LIBS` per raccogliere eventuali librerie opzionali da linkare successivamente all'eseguibile. La variabile `EXTRA_INCLUDES` viene utilizzata in modo simile per i file header opzionali. Questo è un approccio classico quando si ha a che fare con molti componenti opzionali, tratteremo l'approccio moderno nel prossimo step.

Le modifiche corrispondenti al codice sorgente sono abbastanza semplici. Per prima cosa, in `tutorial.cxx`, si include l'header `MathFunctions.h` se ne abbiamo bisogno:

```
#ifdef USE_MYMATH
#  include "MathFunctions.h"
#endif
```

Quindi, nello stesso file, si fa in modo che `USE_MYMATH` controlli quale funzione di radice quadrata utilizzare:

```
#ifdef USE_MYMATH
  const double outputValue = mysqrt(inputValue);
#else
  const double outputValue = sqrt(inputValue);
#endif
```

Dato che il codice sorgente ora richiede `USE_MYMATH` possiamo aggiungerlo a `TutorialConfig.h.in` con la seguente riga:

```
#cmakedefine USE_MYMATH
```

Esercizio: Perché è importante configurare `TutorialConfig.h.in` dopo l'opzione per `USE_MYMATH`? Cosa accadrebbe se invertissimo le due cose?

Eseguire l'eseguibile `cmake` o `cmake-gui` per configurare il progetto e quindi crearlo con lo strumento di build scelto. Eseguire poi il Tutorial buildato.

Ora aggiorniamo il valore di `USE_MYMATH`. Il modo più semplice è usare `cmake-gui` o `ccmake` se si sta nel terminale. Oppure, in alternativa, per modificare l'opzione da riga di comando, provare:

```
cmake ../Step2 -DUSE_MYMATH=OFF
```

Re-buildare ed eseguire nuovamente il tutorial.

Quale funzione dà risultati migliori, sqrt o mysqrt?

17.4 Aggiunta dei Requisiti di Utilizzo per la Libreria (Step 3)

I requisiti di utilizzo consentono un controllo molto migliore sul link di una libreria o di un eseguibile e sulla riga di include, fornendo allo stesso tempo un maggiore controllo sulla proprietà transitiva dei target all'interno di CMake. I comandi principali che sfruttano i requisiti di utilizzo sono:

- `target_compile_definitions`
- `target_compile_options`
- `target_include_directories`
- `target_link_libraries`

Effettuiamo il refactoring del codice da *Aggiunta di una libreria (Step 2)* per utilizzare il moderno approccio CMake dei requisiti di utilizzo. Per prima cosa affermiamo che chiunque effettui il link a MathFunctions deve includere la directory corrente dei sorgenti, mentre lo stesso MathFunctions non lo fa. Quindi questo può diventare un requisito di utilizzo dell'`INTERFACCIA`.

§Da ricordare che INTERFACCIA significa cose che i «consumer» richiedono ma il produttore no. Aggiungere le seguenti righe alla fine di MathFunctions/CMakeLists.txt:

```
target_include_directories(MathFunctions
          INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}
          )
```

Ora che abbiamo specificato i requisiti di utilizzo per MathFunctions possiamo rimuovere in sicurezza gli usi della variabile EXTRA_INCLUDES dal CMakeLists.txt del livello superiore, qui:

```
if(USE_MYMATH)
  add_subdirectory(MathFunctions)
  list(APPEND EXTRA_LIBS MathFunctions)
endif()
```

E qui:

```
target_include_directories(Tutorial PUBLIC
                           "${PROJECT_BINARY_DIR}"
                           )
```

Fatto ciò, eseguire cmake o cmake-gui per configurare il progetto e quindi crearlo con la build scelta o utilizzando cmake --build . dalla directory build.

17.5 Installazione e Test (Step 4)

Ora possiamo iniziare ad aggiungere regole di installazione e testare il supporto al progetto.

17.5.1 Regole di Installazione

Le regole di installazione sono abbastanza semplici: per MathFunctions vogliamo installare la libreria e il file header e per l'applicazione vogliamo installare l'eader eseguibile e configurato.

Quindi alla fine di MathFunctions/CMakeLists.txt aggiungiamo:

```
install(TARGETS MathFunctions DESTINATION lib)
install(FILES MathFunctions.h DESTINATION include)
```

E alla fine del CMakeLists.txt di livello superiore aggiungiamo:

```
install(TARGETS Tutorial DESTINATION bin)
install(FILES "${PROJECT_BINARY_DIR}/TutorialConfig.h"
  DESTINATION include
  )
```

Questo è tutto ciò che serve per creare un'installazione locale di base del tutorial.

Ora eseguire `cmake` o `cmake-gui` per configurare il progetto e quindi crearlo col tool di build scelto.

Quindi esegui lo step di installazione utilizzando l'opzione `install` del comando `cmake` (introdotto nella versione 3.15, le versioni precedenti di CMake devono utilizzare `make install`) dalla riga di comando. Per gli strumenti multi-configurazione, non dimenticare di utilizzare l'argomento `--config` per specificare la configurazione. Se si utilizza un IDE, creare semplicemente il target `INSTALL`. Questo passaggio installerà i file header, le librerie e gli eseguibili appropriati. Per esempio:

```
cmake --install .
```

La variabile CMake `CMAKE_INSTALL_PREFIX` viene utilizzata per determinare la root in cui verranno installati i file. Se si usa il comando `cmake --install`, il prefisso di installazione può essere sovrascritto tramite l'argomento `--prefix`. Per esempio:

```
cmake --install . --prefix "/home/myuser/installdir"
```

Passare alla directory di installazione e verificare che il Tutorial installato venga eseguito.

17.5.2 Supporto per i Test

Successivamente testiamo l'applicazione. Alla fine del file `CMakeLists.txt` di livello superiore possiamo abilitare il test e quindi aggiungere una serie di test di base per verificare che l'applicazione funzioni correttamente.

```
enable_testing()

# does the application run
add_test(NAME Runs COMMAND Tutorial 25)

# does the usage message work?
add_test(NAME Usage COMMAND Tutorial)
set_tests_properties(Usage
  PROPERTIES PASS_REGULAR_EXPRESSION "Usage:.*number"
  )

# define a function to simplify adding tests
function(do_test target arg result)
  add_test(NAME Comp${arg} COMMAND ${target} ${arg})
  set_tests_properties(Comp${arg}
    PROPERTIES PASS_REGULAR_EXPRESSION ${result}
    )
endfunction(do_test)

# do a bunch of result based tests
do_test(Tutorial 4 "4 is 2")
do_test(Tutorial 9 "9 is 3")
```

(continues on next page)

(continua dalla pagina precedente)

```
do_test(Tutorial 5 "5 is 2.236")
do_test(Tutorial 7 "7 is 2.645")
do_test(Tutorial 25 "25 is 5")
do_test(Tutorial -25 "-25 is [-nan|nan|0]")
do_test(Tutorial 0.0001 "0.0001 is 0.01")
```

Il primo test verifica semplicemente che l'applicazione venga eseguita, non subisca segfault o si blocchi in altro modo e abbia un valore di ritorno pari a zero. Questa è la forma base di un test CTest.

Il test successivo utilizza la proprietà di test `PASS_REGULAR_EXPRESSION` per verificare che l'output del test contenga determinate stringhe. In questo caso, verificare che il messaggio di utilizzo venga stampato quando viene fornito un numero errato di argomenti.

Infine, abbiamo una funzione chiamata `do_test` che esegue l'applicazione e verifica che la radice quadrata calcolata sia corretta per un dato input. Per ogni invocazione di `do_test`, un altro test viene aggiunto al progetto con un nome, input e risultati attesi in base agli argomenti passati.

Ri-eseguire la build dell'applicazione e passare alla directory binaria e lanciare l'eseguibile di `ctest`: `ctest -N` e `ctest -VV`. Per i generatori multi-config (ad esempio Visual Studio), è necessario specificare il tipo di configurazione. Per eseguire test in modalità Debug, ad esempio, utilizzare `ctest -C Debug -VV` dalla directory build (non dalla sottodirectory Debug!). In alternativa, creare il target `RUN_TESTS` dall'IDE.

17.6 Aggiungere l'Introspezione del Sistema (Step 5)

Consideriamo di aggiungere del codice al progetto che dipende dalle funzionalità che la piattaforma target potrebbe non avere. Per questo esempio, aggiungeremo del codice che dipende dal fatto che la piattaforma target abbia o meno le funzioni `log` e `exp`. Naturalmente quasi tutte le piattaforme hanno queste funzioni, ma per questo tutorial supponiamo che non siano comuni.

Se la piattaforma ha `log` e `exp` li useremo per calcolare la radice quadrata nella funzione `mysqrt`. Per prima cosa testiamo la disponibilità di queste funzioni utilizzando il modulo `CheckSymbolExists` in `MathFunctions/CMakeLists.txt`. Su alcune piattaforme dovremo linkare la libreria m. Se inizialmente non vengono trovati `log` e `exp`, richiede la libreria m e riprova.

```
include(CheckSymbolExists)
check_symbol_exists(log "math.h" HAVE_LOG)
check_symbol_exists(exp "math.h" HAVE_EXP)
if(NOT (HAVE_LOG AND HAVE_EXP))
  unset(HAVE_LOG CACHE)
  unset(HAVE_EXP CACHE)
  set(CMAKE_REQUIRED_LIBRARIES "m")
  check_symbol_exists(log "math.h" HAVE_LOG)
  check_symbol_exists(exp "math.h" HAVE_EXP)
```

(continues on next page)

(continua dalla pagina precedente)

```
  if(HAVE_LOG AND HAVE_EXP)
    target_link_libraries(MathFunctions PRIVATE m)
  endif()
endif()
```

Se disponibile, si usa `target_compile_definitions` per specificare `HAVE_LOG` and `HAVE_EXP` come definizioni di compilazione `PRIVATE`.

```
if(HAVE_LOG AND HAVE_EXP)
  target_compile_definitions(MathFunctions
                             PRIVATE "HAVE_LOG" "HAVE_EXP")
endif()
```

Se nel sistema sono disponibili `log` e `exp`, li utilizzeremo per calcolare la radice quadrata nella funzione `mysqrt`. Si aggiunge il seguente codice alla funzione `mysqrt` in `MathFunctions/mysqrt.cxx` (da non dimenticare `#endif` prima di restituire il risultato!):

```
#if defined(HAVE_LOG) && defined(HAVE_EXP)
  double result = exp(log(x) * 0.5);
  std::cout << "Computing sqrt of " << x << " to be " << result
            << " using log and exp" << std::endl;
#else
  double result = x;
```

Dovremo anche modificare `mysqrt.cxx` per includere `cmath`.

```
#include <cmath>
```

Lanciare l'eseguibile `cmake` o `cmake-gui` per configurare il progetto, poi lo si crea col strumento di creazione stool di build e si esegue il tutorial.

Quale funzione dà risultati migliori ora, sqrt o mysqrt?

17.7 Aggiunta di un Comando Personalizzato e di un File Generato (Step 6)

Supponiamo, ai fini di questo tutorial, di decidere di non voler mai utilizzare le funzioni `log` e `exp` della piattaforma e di voler invece generare una tabella di valori precalcolati da utilizzare in `mysqrt`. In questa sezione creeremo la tabella come parte del processo di build, poi compileremo la tabella nell'applicazione.

Innanzitutto, rimuoviamo il controllo per le funzioni `log` e `exp` in `MathFunctions/CMakeLists.txt`. Quindi si rimuove il controllo per `HAVE_LOG` e `HAVE_EXP` da `mysqrt.cxx`. Allo stesso tempo, possiamo rimuovere `#include <cmath>`.

Nella sottodirectory `MathFunctions` è stato fornito un nuovo file sorgente denominato `MakeTable.cxx` per generare la tabella.

Dall'esame del file, si vede che la tabella viene prodotta come codice C++ valido e che il nome del file di output viene passato come argomento.

Il passo successivo è quello di aggiungere i comandi appropriati al file `MathFunctions/CMakeLists.txt` per creare l'eseguibile MakeTable per poi eseguirlo come parte del processo di build. Per eseguire questa operazione sono necessari alcuni comandi.

Innanzitutto, nella parte superiore di `MathFunctions/CMakeLists.txt`, viene aggiunto l'eseguibile per `MakeTable` così come verrebbe aggiunto qualsiasi altro eseguibile.

```
add_executable(MakeTable MakeTable.cxx)
```

Quindi aggiungiamo un comando personalizzato che specifica come produrre `Table.h` eseguendo MakeTable.

```
add_custom_command(
  OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/Table.h
  COMMAND MakeTable ${CMAKE_CURRENT_BINARY_DIR}/Table.h
  DEPENDS MakeTable
  )
```

Successivamente dobbiamo far sapere a CMake che `mysqrt.cxx` dipende dal file generato `Table.h`. Questo viene fatto aggiungendo il file `Table.h` generato all'elenco dei sorgenti per la libreria MathFunctions.

```
add_library(MathFunctions
            mysqrt.cxx
            ${CMAKE_CURRENT_BINARY_DIR}/Table.h
            )
```

Dobbiamo anche aggiungere la directory binaria corrente all'elenco delle directory di inclusione in modo che `Table.h` sia raggiungibile da `mysqrt.cxx`.

```
target_include_directories(MathFunctions
           INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}
           PRIVATE ${CMAKE_CURRENT_BINARY_DIR}
           )
```

Ora utilizziamo la tabella generata. Innanzitutto, si modifica `mysqrt.cxx` per includere `Table.h`. Successivamente, possiamo riscrivere la funzione mysqrt per utilizzare la tabella:

```
double mysqrt(double x)
{
  if (x <= 0) {
    return 0;
  }

  // use the table to help find an initial value
  double result = x;
  if (x >= 1 && x < 10) {
```

(continues on next page)

(continua dalla pagina precedente)

```
    std::cout << "Use the table to help find an initial value " <<␣
↪std::endl;
    result = sqrtTable[static_cast<int>(x)];
  }

  // do ten iterations
  for (int i = 0; i < 10; ++i) {
    if (result <= 0) {
      result = 0.1;
    }
    double delta = x - (result * result);
    result = result + 0.5 * delta / result;
    std::cout << "Computing sqrt of " << x << " to be " << result <<␣
↪std::endl;
  }

  return result;
}
```

Eseguire l'eseguibile `cmake` o `cmake-gui` per configurare il progetto e quindi crearlo con lo strumento di build scelto.

Con la build di questo progetto, verrà prima creato l'eseguibile `MakeTable`. Verrà poi eseguito `MakeTable` per produrre `Table.h`. Infine, si compilerà `mysqrt.cxx` che include `Table.h` per produrre la libreria MathFunctions.

Lanciare l'eseguibile del tutorial e verificare che utilizzi la tabella.

17.8 Creazione di un Programma di Installazione (Step 7)

Supponiamo ora di voler distribuire il nostro progetto ad altre persone in modo che possano usarlo. Vogliamo fornire distribuzioni sia binarie che sorgenti su una varietà di piattaforme. Questo è un po" diverso dall'installazione fatta in precedenza in *Installazione e Test (Step 4)* , dove stavamo installando i binari creati dal codice sorgente. In questo esempio creeremo pacchetti di installazione che supportano installazioni binarie e funzionalità di gestione dei pacchetti. Per raggiungere questo obiettivo utilizzeremo CPack per creare programmi di installazione specifici della piattaforma. Nello specifico dobbiamo aggiungere alcune righe alla fine del nostro file `CMakeLists.txt` di livello superiore.

```
include(InstallRequiredSystemLibraries)
set(CPACK_RESOURCE_FILE_LICENSE "${CMAKE_CURRENT_SOURCE_DIR}/License.
↪txt")
set(CPACK_PACKAGE_VERSION_MAJOR "${Tutorial_VERSION_MAJOR}")
```

(continues on next page)

(continua dalla pagina precedente)

```
set(CPACK_PACKAGE_VERSION_MINOR "${Tutorial_VERSION_MINOR}")
include(CPack)
```

Questo è tutto. Iniziamo includendo `InstallRequiredSystemLibraries`. Questo modulo includerà tutte le librerie runtime necessarie al progetto per la piattaforma corrente. Successivamente impostiamo alcune variabili CPack in cui abbiamo archiviato le informazioni sulla licenza e sulla versione per questo progetto. Le informazioni sulla versione sono state impostate in precedenza in questo tutorial e il file `license.txt` è stato incluso nella directory dei sorgenti di livello superiore per questo step.

Infine includiamo `CPack module` che utilizzerà queste variabili e alcune altre proprietà del sistema corrente per impostare un programma di installazione.

Il passo successivo è creare il progetto nel solito modo e poi lanciare l'eseguibile `cpack`. Per creare una distribuzione binaria, dalla directory binaria eseguire:

```
cpack
```

Per specificare il generatore, utilizzare l'opzione `-G`. Per build multi-configurazione, utilizzare `-C` per specificare la configurazione. Per esempio:

```
cpack -G ZIP -C Debug
```

Per creare una distribuzione sorgente si deve digitare:

```
cpack --config CPackSourceConfig.cmake
```

In alternativa, eseguire `make package` o click destro sul target `Package` e `Build Project` dall'IDE.

Eseguire il programma di installazione trovato nella directory binaria. Lanciare poi l'eseguibile installato e verificare che funzioni.

17.9 Aggiungere il Supporto per una Dashboard (Step 8)

Aggiungere il supporto per inviare i risultati dei nostri test a una dashboard è semplice. Abbiamo già definito una serie di test per il nostro progetto in *Supporto per i Test*. Ora non ci resta che eseguire questi test e inviarli a una dashboard. Per aggiungere il supporto per la dashboard includiamo il modulo `CTest` nel `CMakeLists.txt` di livello superiore.

Sostituire:

```
# enable testing
enable_testing()
```

Con:

```
# enable dashboard scripting
include(CTest)
```

Il modulo CTest chiamerà automaticamente enable_testing(), quindi possiamo rimuoverlo dai file CMake.

Dovremo anche creare un file CTestConfig.cmake nella directory di primo livello in cui possiamo specificare il nome del progetto e dove inviare la dashboard.

```
set(CTEST_PROJECT_NAME "CMakeTutorial")
set(CTEST_NIGHTLY_START_TIME "00:00:00 EST")

set(CTEST_DROP_METHOD "http")
set(CTEST_DROP_SITE "my.cdash.org")
set(CTEST_DROP_LOCATION "/submit.php?project=CMakeTutorial")
set(CTEST_DROP_SITE_CDASH TRUE)
```

ctest leggerà questo file quando verrà eseguito. Per creare una semplice dashboard si può lanciare l'eseguibile cmake o cmake-gui per configurare il progetto, ma senza farne la build ancora. Cambiare invece la directory nell'albero binario ed eseguire:

    ctest [-VV] -D Experimental

Ricordarsi che per i generatori multi-config (es. Visual Studio), è necessario specificare il tipo di configurazione:

```
ctest [-VV] -C Debug -D Experimental
```

Oppure, da un IDE, si esegue la build Experimental del target.

L'eseguibile ctest creerà e testerà il progetto e invierà i risultati alla dashboard pubblica di Kitware: https://my.cdash.org/index.php?project=CMakeTutorial.

17.10 Mix di Static e Shared (Step 9)

In questa sezione mostreremo come la variabile BUILD_SHARED_LIBS è utilizzabile per controllare il comportamento di default di add_library e consentire il controllo su come le librerie senza un tipo esplicito (STATIC, SHARED, MODULE o OBJECT) vengono costruite.

Per ottenere ciò dobbiamo aggiungere BUILD_SHARED_LIBS al CMakeLists.txt di livello superiore. Usiamo il comando option poiché consente agli utenti di selezionare facoltativamente se il valore deve essere ON o OFF.

Successivamente effettueremo il refactoring di MathFunctions per farla diventare una vera libreria che incapsuli l'utilizzo di mysqrt o sqrt, invece di richiedere al codice chiamante di eseguire questa logica. Ciò significherà anche che USE_MYMATH non controllerà la build di MathFunctions, ma controllerà invece il comportamento di questa libreria.

Il primo passo è aggiornare la sezione iniziale del file CMakeLists.txt di livello superiore in modo che assomigli a:

```
cmake_minimum_required(VERSION 3.10)

# set the project name and version
project(Tutorial VERSION 1.0)

# specify the C++ standard
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# control where the static and shared libraries are built so that on␣
↪windows
# we don't need to tinker with the path to run the executable
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY "${PROJECT_BINARY_DIR}")
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY "${PROJECT_BINARY_DIR}")
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${PROJECT_BINARY_DIR}")

option(BUILD_SHARED_LIBS "Build using shared libraries" ON)

# configure a header file to pass the version number only
configure_file(TutorialConfig.h.in TutorialConfig.h)

# add the MathFunctions library
add_subdirectory(MathFunctions)

# add the executable
add_executable(Tutorial tutorial.cxx)
target_link_libraries(Tutorial PUBLIC MathFunctions)
```

Ora che abbiamo fatto in modo che MathFunctions venga sempre utilizzata, ne dovremo aggiornare la logica. Quindi, in `MathFunctions/CMakeLists.txt` dobbiamo creare una SqrtLibrary che verrà creata e installata in modo condizionale quando `USE_MYMATH` è abilitato. Ora, poiché questo è un tutorial, richiederemo esplicitamente che SqrtLibrary venga creata staticamente.

Il risultato finale è che `MathFunctions/CMakeLists.txt` dovrebbe assomigliare a:

```
# add the library that runs
add_library(MathFunctions MathFunctions.cxx)

# state that anybody linking to us needs to include the current source␣
↪dir
# to find MathFunctions.h, while we don't.
target_include_directories(MathFunctions
                           INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}
                           )

# should we use our own math functions
```

(continues on next page)

(continua dalla pagina precedente)

```
option(USE_MYMATH "Use tutorial provided math implementation" ON)
if(USE_MYMATH)

  target_compile_definitions(MathFunctions PRIVATE "USE_MYMATH")

  # first we add the executable that generates the table
  add_executable(MakeTable MakeTable.cxx)

  # add the command to generate the source code
  add_custom_command(
    OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/Table.h
    COMMAND MakeTable ${CMAKE_CURRENT_BINARY_DIR}/Table.h
    DEPENDS MakeTable
    )

  # library that just does sqrt
  add_library(SqrtLibrary STATIC
              mysqrt.cxx
              ${CMAKE_CURRENT_BINARY_DIR}/Table.h
              )

  # state that we depend on our binary dir to find Table.h
  target_include_directories(SqrtLibrary PRIVATE
                             ${CMAKE_CURRENT_BINARY_DIR}
                             )

  target_link_libraries(MathFunctions PRIVATE SqrtLibrary)
endif()

# define the symbol stating we are using the declspec(dllexport) when
# building on windows
target_compile_definitions(MathFunctions PRIVATE "EXPORTING_MYMATH")

# install rules
set(installable_libs MathFunctions)
if(TARGET SqrtLibrary)
  list(APPEND installable_libs SqrtLibrary)
endif()
install(TARGETS ${installable_libs} DESTINATION lib)
install(FILES MathFunctions.h DESTINATION include)
```

Successivamente, si aggiorna `MathFunctions/mysqrt.cxx` per utilizzare i namespace `mathfunctions` e `detail`:

```
#include <iostream>
```

(continues on next page)

(continua dalla pagina precedente)

```
#include "MathFunctions.h"

// include the generated table
#include "Table.h"

namespace mathfunctions {
namespace detail {
// a hack square root calculation using simple operations
double mysqrt(double x)
{
  if (x <= 0) {
    return 0;
  }

  // use the table to help find an initial value
  double result = x;
  if (x >= 1 && x < 10) {
    std::cout << "Use the table to help find an initial value " <<␣
↪std::endl;
    result = sqrtTable[static_cast<int>(x)];
  }

  // do ten iterations
  for (int i = 0; i < 10; ++i) {
    if (result <= 0) {
      result = 0.1;
    }
    double delta = x - (result * result);
    result = result + 0.5 * delta / result;
    std::cout << "Computing sqrt of " << x << " to be " << result <<␣
↪std::endl;
  }

  return result;
}
}
}
```

Dobbiamo anche apportare alcune modifiche a `tutorial.cxx`, in modo che non utilizzi più `USE_MYMATH`:

1. Include sempre `MathFunctions.h`
2. Utilizza sempre `mathfunctions::sqrt`
3. Non include cmath

Infine, si aggiorn `MathFunctions/MathFunctions.h` per utilizzare le definizioni di esportazione delle dll:

```
#if defined(_WIN32)
#  if defined(EXPORTING_MYMATH)
#    define DECLSPEC __declspec(dllexport)
#  else
#    define DECLSPEC __declspec(dllimport)
#  endif
#else // non windows
#  define DECLSPEC
#endif

namespace mathfunctions {
double DECLSPEC sqrt(double x);
}
```

A questo punto, buildando il tutto, si noterà che il link fallisce poiché stiamo combinando una libreria statica senza codice indipendente dalla posizione con una libreria che ha codice indipendente dalla posizione. La soluzione a questo problema è impostare esplicitamente la proprietà del target `POSITION_INDEPENDENT_CODE` di SqrtLibrary su True, indipendentemente dal tipo di build.

```
  # state that SqrtLibrary need PIC when the default is shared␣
↪libraries
  set_target_properties(SqrtLibrary PROPERTIES
                        POSITION_INDEPENDENT_CODE ${BUILD_SHARED_LIBS}
                        )

  target_link_libraries(MathFunctions PRIVATE SqrtLibrary)
```

Esercizio: Abbiamo modificato `MathFunctions.h` per utilizzare le definizioni di esportazione delle dll. Utilizzando la documentazione di CMake si può trovare un modulo di supporto per semplificarlo?

17.11 Aggiungere di Espressioni del Generatore (Step 10)

`Generator expressions` vengono valutate durante la generazione del sistema di build per produrre informazioni specifiche per ciascuna configurazione di build.

`Generator expressions` sono consentite nel contesto di molte proprietà di target, come `LINK_LIBRARIES`, `INCLUDE_DIRECTORIES`, `COMPILE_DEFINITIONS` e altre. Sono anche utilizzabili quando si usano comandi per popolare tali proprietà, come `target_link_libraries`, `target_include_directories`, `target_compile_definitions` e altri.

`Generator expressions` possono essere utilizzate per abilitare il link condizionale, le definizioni condizionali utilizzate durante la compilazione, le directory di inclusione condizionale e altro ancora. Le condizioni possono essere basate sulla configurazione della build, sul-

le proprietà del target, sulle informazioni della piattaforma o su qualsiasi altra informazione interrogabile.

Esistono diversi tipi di generator expressions comprese le espressioni Logiche, Informative e di Output.

Le espressioni logiche vengono utilizzate per creare un output condizionale. Le espressioni di base sono le espressioni 0 e 1. $<0:...> restituisce una stringa vuota e <1:...> restituisce il contenuto di «...». Possono anche essere annidate.

Un utilizzo comune delle generator expressions è quello di aggiungere in modo condizionale flag del compilatore, come quelli per i livelli del linguaggio o i warning. Un pattern interessante è quello di associare queste informazioni a un target INTERFACE che consente a queste informazioni di propagarsi. Iniziamo costruendo un target INTERFACE e specificando il livello dello standard C++ richiesto di 11 invece di utilizzare CMAKE_CXX_STANDARD.

Quindi il seguente codice:

```
# specify the C++ standard
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)
```

Verrebbe sostituito con:

```
add_library(tutorial_compiler_flags INTERFACE)
target_compile_features(tutorial_compiler_flags INTERFACE cxx_std_11)
```

Successivamente aggiungiamo i flag di warning del compilatore desiderati che vogliamo per il progetto. Poiché i flag di warning variano in base al compilatore, utilizziamo l'espressione generatrice COMPILE_LANG_AND_ID per controllare quali flag applicare dato un linguaggio e un set di ID del compilatore come mostrato di seguito:

```
set(gcc_like_cxx "$<COMPILE_LANG_AND_ID:CXX,ARMClang,AppleClang,Clang,
↪GNU>")
set(msvc_cxx "$<COMPILE_LANG_AND_ID:CXX,MSVC>")
target_compile_options(tutorial_compiler_flags INTERFACE
  "$<${gcc_like_cxx}:$<BUILD_INTERFACE:-Wall;-Wextra;-Wshadow;-
↪Wformat=2;-Wunused>>"
  "$<${msvc_cxx}:$<BUILD_INTERFACE:-W3>>"
)
```

Osservando questo vediamo che i flag di warning sono incapsulati all'interno di una condizione BUILD_INTERFACE. Questo viene fatto in modo che i consumer (consumatori) del progetto installato non ereditino i flag di warning.

Esercizio: Modificare MathFunctions/CMakeLists.txt in modo che tutti i target abbiano una chiamata target_link_libraries a tutorial_compiler_flags.

17.12 Aggiungere la Configurazione di Esportazione (Step 11)

In *Installazione e Test (Step 4)* del tutorial abbiamo aggiunto la possibilità per CMake di installare la libreria e gli header del progetto. In *Creazione di un Programma di Installazione (Step 7)* abbiamo aggiunto la possibilità di impacchettare queste informazioni in modo che possano essere distribuite ad altri.

Il passaggio successivo è quello di aggiungere le informazioni necessarie in modo che altri progetti CMake possano utilizzare il nostro progetto, sia da una directory di build, da un'installazione locale o da un pacchetto.

Il primo passo è aggiornare i comandi `install(TARGETS)` per specificare non solo una `DESTINATION` ma anche una `EXPORT`. La parola chiave `EXPORT` genera e installa un file CMake contenente il codice per importare tutti i target elencati nel comando install dall'albero di installazione. Quindi andiamo avanti con un `EXPORT` esplicito della libreria MathFunctions aggiornando il comando `install` in `MathFunctions/CMakeLists.txt` in modo che assomigli a:

```
set(installable_libs MathFunctions tutorial_compiler_flags)
if(TARGET SqrtLibrary)
  list(APPEND installable_libs SqrtLibrary)
endif()
install(TARGETS ${installable_libs}
        DESTINATION lib
        EXPORT MathFunctionsTargets)
install(FILES MathFunctions.h DESTINATION include)
```

Ora che abbiamo esportato MathFunctions, dobbiamo anche installare esplicitamente il file `MathFunctionsTargets.cmake` generato. Questo viene fatto aggiungendo quanto segue in fondo al file `CMakeLists.txt` di livello superiore:

```
install(EXPORT MathFunctionsTargets
  FILE MathFunctionsTargets.cmake
  DESTINATION lib/cmake/MathFunctions
)
```

A questo punto dovrebbe provare ad eseguire CMake. Se tutto è configurato correttamente, CMake genererà un errore simile a:

```
Target "MathFunctions" INTERFACE_INCLUDE_DIRECTORIES property contains
path:

  "/Users/robert/Documents/CMakeClass/Tutorial/Step11/MathFunctions"

which is prefixed in the source directory.
```

Ciò che CMake sta cercando di dire è che nella generazione delle informazioni di esportazione ci sarà un percorso che è intrinsecamente legato alla macchina corrente e non sarà valido su altre

macchine. La soluzione a questo è aggiornare `target_include_directories` di MathFunctions per comprendere che necessita di posizioni `INTERFACE` diverse quando viene utilizzato dalla directory di build e da un pacchetto/installazione. Ciò significa convertire la chiamata `target_include_directories` per MathFunctions in modo che assomigli a:

```
target_include_directories(MathFunctions
                           INTERFACE
                            $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_
↪DIR}>
                            $<INSTALL_INTERFACE:include>
                           )
```

Una volta aggiornato, possiamo eseguire nuovamente CMake e verificare che non emetta più dei warning.

A questo punto, abbiamo CMake che impacchetta correttamente le informazioni del target richieste, ma dovremo comunque generare un `MathFunctionsConfig.cmake` in modo che il comando CMake `find_package` possa trovare il progetto. Andiamo quindi avanti e aggiungiamo un nuovo file al livello più alto del progetto chiamato `Config.cmake.in` col seguente contenuto:

```
@PACKAGE_INIT@

include ( "${CMAKE_CURRENT_LIST_DIR}/MathFunctionsTargets.cmake" )
```

Quindi, per configurare e installare correttamente il file, si aggiunge quanto segue in fondo al file `CMakeLists.txt` di livello superiore:

```
install(EXPORT MathFunctionsTargets
  FILE MathFunctionsTargets.cmake
  DESTINATION lib/cmake/MathFunctions
)

include(CMakePackageConfigHelpers)
# generate the config file that is includes the exports
configure_package_config_file(${CMAKE_CURRENT_SOURCE_DIR}/Config.cmake.
↪in
  "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfig.cmake"
  INSTALL_DESTINATION "lib/cmake/example"
  NO_SET_AND_CHECK_MACRO
  NO_CHECK_REQUIRED_COMPONENTS_MACRO
  )
# generate the version file for the config file
write_basic_package_version_file(
  "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfigVersion.cmake"
  VERSION "${Tutorial_VERSION_MAJOR}.${Tutorial_VERSION_MINOR}"
  COMPATIBILITY AnyNewerVersion
)
```

(continues on next page)

(continua dalla pagina precedente)

```
# install the configuration file
install(FILES
  ${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfig.cmake
  DESTINATION lib/cmake/MathFunctions
  )
```

A questo punto, abbiamo generato una configurazione CMake rilocabile per il progetto utilizzabile dopo che il progetto è stato installato o inserito in un pacchetto. Se vogliamo che il progetto venga utilizzato anche da una directory di build dobbiamo solo aggiungere quanto segue in fondo al `CMakeLists.txt` del livello superiore:

```
export(EXPORT MathFunctionsTargets
  FILE "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsTargets.cmake"
)
```

Con questa chiamata di esportazione ora generiamo un `Targets.cmake`, consentendo al `MathFunctionsConfig.cmake` configurato nella directory build di essere utilizzato da altri progetti, senza bisogno che sia installato.

17.13 Packaging di Debug e Release (Step 12)

Nota: Questo esempio è valido per generatori a configurazione singola e non funzionerà per generatori a configurazione multipla (ad esempio Visual Studio).

Per default, il modello di CMake prevede che una directory di build contenga solo una singola configurazione, che si tratti di Debug, Release, MinSizeRel o RelWithDebInfo. È possibile, tuttavia, configurare CPack per raggruppare più directory di build e costruire un pacchetto che contenga più configurazioni dello stesso progetto.

Innanzitutto, vogliamo garantire che le build di debug e di release utilizzino nomi diversi per gli eseguibili e le librerie che verranno installate. Usiamo *d* come suffisso per l'eseguibile e le librerie di debug.

Si setta `CMAKE_DEBUG_POSTFIX` vicino all'inizio del file `CMakeLists.txt` di livello superiore:

```
set(CMAKE_DEBUG_POSTFIX d)

add_library(tutorial_compiler_flags INTERFACE)
```

E la proprietà `DEBUG_POSTFIX` sull'eseguibile del tutorial:

```
add_executable(Tutorial tutorial.cxx)
set_target_properties(Tutorial PROPERTIES DEBUG_POSTFIX ${CMAKE_DEBUG_
↪POSTFIX})

target_link_libraries(Tutorial PUBLIC MathFunctions)
```

Aggiungiamo anche la numerazione delle versioni alla libreria MathFunctions. In `MathFunctions/CMakeLists.txt`, si impostano le proprietà `VERSION` e `SOVERSION`:

```
set_property(TARGET MathFunctions PROPERTY VERSION "1.0.0")
set_property(TARGET MathFunctions PROPERTY SOVERSION "1")
```

Dalla directory `Step12`, creare le sottodirectory `debug` e `release`. Il layout sarà simile a:

```
- Step12
   - debug
   - release
```

Ora dobbiamo impostare la build di debug e quella di release. Possiamo usare `CMAKE_BUILD_TYPE` per impostare il tipo di configurazione:

```
cd debug
cmake -DCMAKE_BUILD_TYPE=Debug ..
cmake --build .
cd ../release
cmake -DCMAKE_BUILD_TYPE=Release ..
cmake --build .
```

Ora che sia la build di debug che quella di release sono complete, possiamo utilizzare un file di configurazione personalizzato per comprimere entrambe le build in un'unica versione. Nella directory `Step12`, creare un file chiamato `MultiCPackConfig.cmake`. In questo file, si include innanzitutto il file di configurazione di default creato dall'eseguibile `cmake`.

Successivamente, si utilizza la variabile `CPACK_INSTALL_CMAKE_PROJECTS` per specificare quali progetti installare. In questo caso, vogliamo installare sia debug che release.

```
include("release/CPackConfig.cmake")

set(CPACK_INSTALL_CMAKE_PROJECTS
    "debug;Tutorial;ALL;/"
    "release;Tutorial;ALL;/"
    )
```

Dalla directory `Step12`, si esegue `cpack` specificando il file di configurazione personalizzato con l'opzione `config`:

```
cpack --config MultiCPackConfig.cmake
```

CAPITOLO 18

Guida all'Interazione con l'Utente

18.1 Introduzione

Laddove un pacchetto software fornisce un sistema di build basato su CMake col sorgente del software, il consumatore del software è tenuto a eseguire uno strumento di interazione con l'utente CMake per buildare.

I sistemi di build basati su CMake ben funzionanti non creano alcun output nella directory sorgente, quindi in genere l'utente esegue una compilazione fuori sorgente ed esegue la compilazione lì. Innanzitutto, è necessario istruire CMake affinché generari un sistema di build adatto, poi l'utente richiama un tool di build per elaborare il sistema generato. Il buildsystem generato è specifico per la macchina utilizzata per generarlo e non è ridistribuibile. Ogni «consumer» di un pacchetto software con sorgente è tenuto a utilizzare CMake per generare un sistema di build specifico per il proprio sistema.

I sistemi di build generati dovrebbero generalmente essere trattati come di sola lettura. I file CMake come artefatto primario dovrebbero specificare completamente il sistema di build e non dovrebbero esserci motivi per popolare manualmente le proprietà in un IDE, ad esempio dopo aver generato il sistema di build. CMake riscriverà periodicamente il sistema di build generato, quindi le modifiche apportate dagli utenti verranno sovrascritte.

Le funzionalità e le interfacce utente descritte in questo manuale sono disponibili per tutti i sistemi di build basati su CMake grazie ai file CMake.

Gli strumenti CMake potrebbero segnalare errori all'utente durante l'elaborazione dei file CMake forniti, ad esempio segnalare che il compilatore non è supportato o che il compilatore non supporta un'opzione di compilazione richiesta oppure che non è possibile trovare una dipendenza. Questi errori devono essere risolti dall'utente scegliendo un compilatore diverso, `installing dependencies`, o indicando a CMake dove trovarli, ecc.

18.1.1 Strumento cmake da Riga di Comando

Un uso semplice ma tipico di cmake(1) con una nuova copia del codice sorgente del software è quello di creare una directory di build e invocarvi cmake:

```
$ cd some_software-1.4.2
$ mkdir build
$ cd build
$ cmake .. -DCMAKE_INSTALL_PREFIX=/opt/the/prefix
$ cmake --build .
$ cmake --build . --target install
```

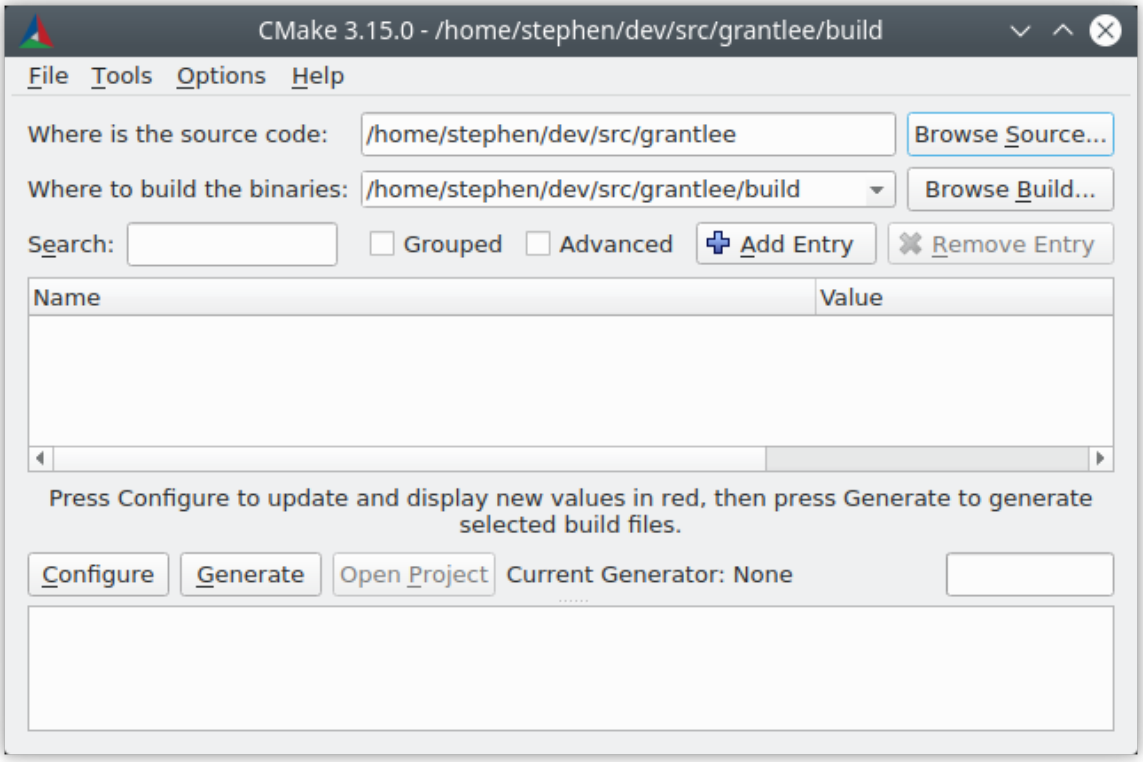
Si consiglia di creare una directory separata rispetto al sorgente perché ciò mantiene la directory dei sorgenti incontaminata, consente di creare una singola sorgente con più toolchain e consente una facile cancellazione degli artefatti di build semplicemente eliminando la directory di build.

Gli strumenti CMake possono segnalare warning destinati al fornitore del software, non all'utente del software. Tali warning terminano con «This warning is for project developers». Gli utenti possono disabilitare tali avvisi passando il flag `-Wno-dev` a cmake(1).

18.1.2 Il tool cmake-gui

Gli utenti più abituati alle interfacce GUI possono utilizzare cmake-gui(1) per invocare CMake e generare un sistema di build.

Le directory sorgente e binaria devono prima essere popolate. Si consiglia sempre di utilizzare directory diverse per il sorgente e la build.



18.2 Generazione di un Buildsystem

Esistono diversi strumenti dell'interfaccia utente utilizzabili per generare un sistema di build dai file CMake. Gli strumenti `ccmake(1)` e `cmake-gui(1)` guidano l'utente attraverso l'impostazione delle varie opzioni necessarie. Il tool `cmake(1)` può essere richiamato per specificare le opzioni sulla riga di comando. Questo manuale descrive le opzioni che possono essere impostate utilizzando uno qualsiasi degli strumenti dell'interfaccia utente, sebbene la modalità di impostazione di un'opzione sia diversa per ciascuno.

18.2.1 Ambiente della riga di comando

Quando si invoca `cmake(1)` con un buildsystem a riga di comando come `Makefiles` o `Ninja`, è necessario utilizzare l'ambiente di build corretto per garantire che gli strumenti siano disponibili. CMake deve essere in grado di trovare il `tool di build` il compilatore, il linker e altri strumenti appropriati secondo necessità.

Sui sistemi Linux, gli strumenti appropriati vengono spesso forniti in posizioni a livello di sistema e possono essere facilmente installati tramite il gestore pacchetti di sistema. È possibile utilizzare anche altre toolchain fornite dall'utente o installate in posizioni non predefinite.

Durante la cross-compilazione, alcune piattaforme potrebbero richiedere l'impostazione di variabili di ambiente o fornire script per impostare l'ambiente.

Visual Studio fornisce più prompt dei comandi e script `vcvarsall.bat` per impostare gli ambienti corretti per i buildsystem a riga di comando. Sebbene non sia strettamente necessario

utilizzare un ambiente a riga di comando corrispondente quando si utilizza un generatore come Visual Studio, farlo non presenta svantaggi.

Quando si utilizza Xcode, è possibile che sia installata più di una versione di Xcode. Quale utilizzare può essere selezionato in diversi modi, ma i metodi più comuni sono:

- Impostazione della versione di default nelle preferenze dell'IDE Xcode.
- Impostazione della versione di default tramite lo strumento a riga di comando `xcode-select`.
- Sostituire la versione di default impostando la variabile di ambiente `DEVELOPER_DIR` durante l'esecuzione di CMake e dello strumento di build.

Per comodità, `cmake-gui(1)` fornisce un editor di variabili d'ambiente.

18.2.2 L'opzione `-G` della riga di comando

CMake sceglie un generatore per default in base alla piattaforma. Di solito, il generatore di default è sufficiente per consentire all'utente di procedere alla build del software.

L'utente può sovrascrivere il generatore di default con l'opzione `-G`:

```
$ cmake .. -G Ninja
```

L'output di `cmake --help` include un elenco dei `generatori` tra cui l'utente può scegliere. Notare che i nomi dei generatori fanno distinzione tra maiuscole e minuscole.

Sui sistemi simili a Unix (incluso Mac OS X), il generatore `Unix Makefiles` viene utilizzato per default. Una variante di quel generatore può essere utilizzata anche su Windows in vari ambienti, come il generatore `NMake Makefiles` e `MinGW Makefiles`. Questi generatori producono una variante di `Makefile` eseguibile con `make`, `gmake`, `nmake` o strumenti simili. Consulta la documentazione del singolo generatore per ulteriori informazioni sugli ambienti e sugli strumenti specifici.

Il generatore `Ninja` è disponibile su tutte le principali piattaforme. `ninja` è uno strumento di build simile nei casi d'uso a `make`, ma con un focus su prestazioni ed efficienza.

Su Windows, è possibile utilizzare `cmake(1)` per generare soluzioni per l'IDE di Visual Studio. Le versioni di Visual Studio possono essere specificate dal nome del prodotto dell'IDE, che include un anno a quattro cifre. Vengono forniti alias per altri significati con cui a volte si fa riferimento alle versioni di Visual Studio, ad esempio due cifre che corrispondono alla versione del prodotto del compilatore VisualC++ o una combinazione dei due:

```
$ cmake .. -G "Visual Studio 2019"
$ cmake .. -G "Visual Studio 16"
$ cmake .. -G "Visual Studio 16 2019"
```

I generatori di Visual Studio possono avere come target architetture diverse. È possibile specificare l'architettura del target utilizzando l'opzione *-A*:

```
cmake .. -G "Visual Studio 2019" -A x64
cmake .. -G "Visual Studio 16" -A ARM
cmake .. -G "Visual Studio 16 2019" -A ARM64
```

Su Apple, il generatore `Xcode` può essere utilizzato per generare file di progetto per l'IDE Xcode.

Alcuni IDE come KDevelop4, QtCreator e CLion hanno il supporto nativo per i sistemi di build basati su CMake. Questi IDE forniscono un'interfaccia utente per selezionare un generatore da utilizzare, tipicamente una scelta tra un generatore basato su `Makefile` o su `Ninja`.

Notare che non è possibile modificare il generatore con `-G` dopo la prima invocazione di CMake. Per modificare il generatore è necessario eliminare la directory di build e avviare la build da zero.

Quando si generano file di progetto e soluzioni di Visual Studio, sono disponibili diverse altre opzioni da utilizzare durante la prima esecuzione di `cmake(1)`.

Il set di tool di Visual Studio può essere specificato con l'opzione `-T`:

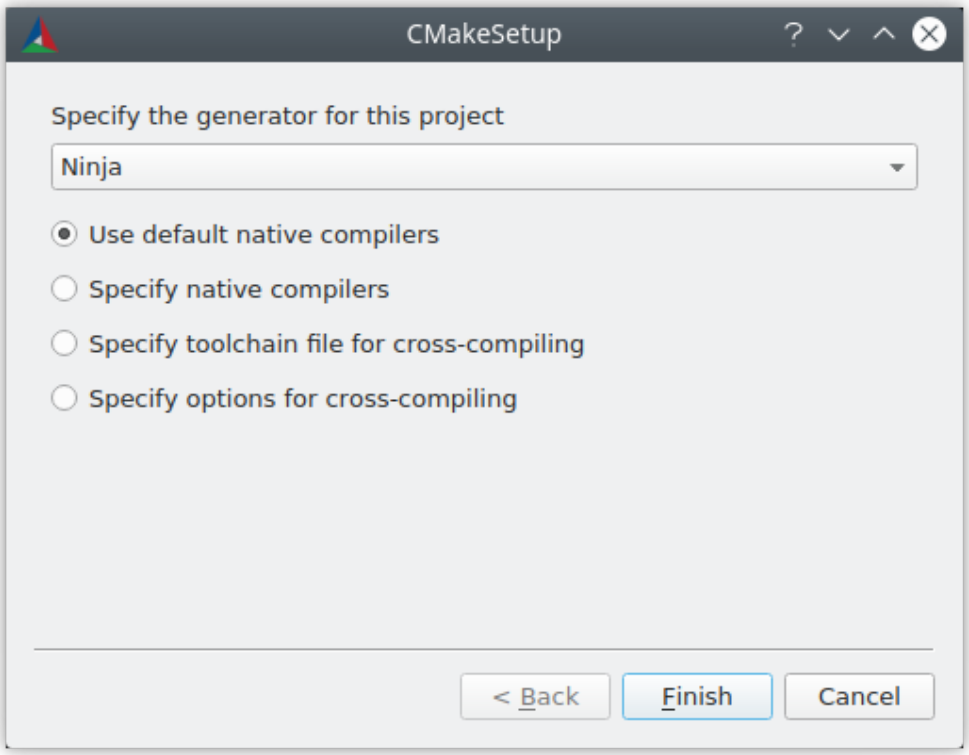
```
$ # Build with the clang-cl toolset
$ cmake.exe .. -G "Visual Studio 16 2019" -A x64 -T ClangCL
$ # Build targeting Windows XP
$ cmake.exe .. -G "Visual Studio 16 2019" -A x64 -T v120_xp
```

Mentre l'opzione `-A` specifica l'architettura _target_, l'opzione `-T` può essere utilizzata per specificare i dettagli della toolchain utilizzata. Ad esempio, è possibile fornire *-Thost=x64* per selezionare la versione a 64 bit degli strumenti host. Di seguito viene illustrato come utilizzare i tool a 64 bit ed eseguire una build per un'architettura target a 64 bit:

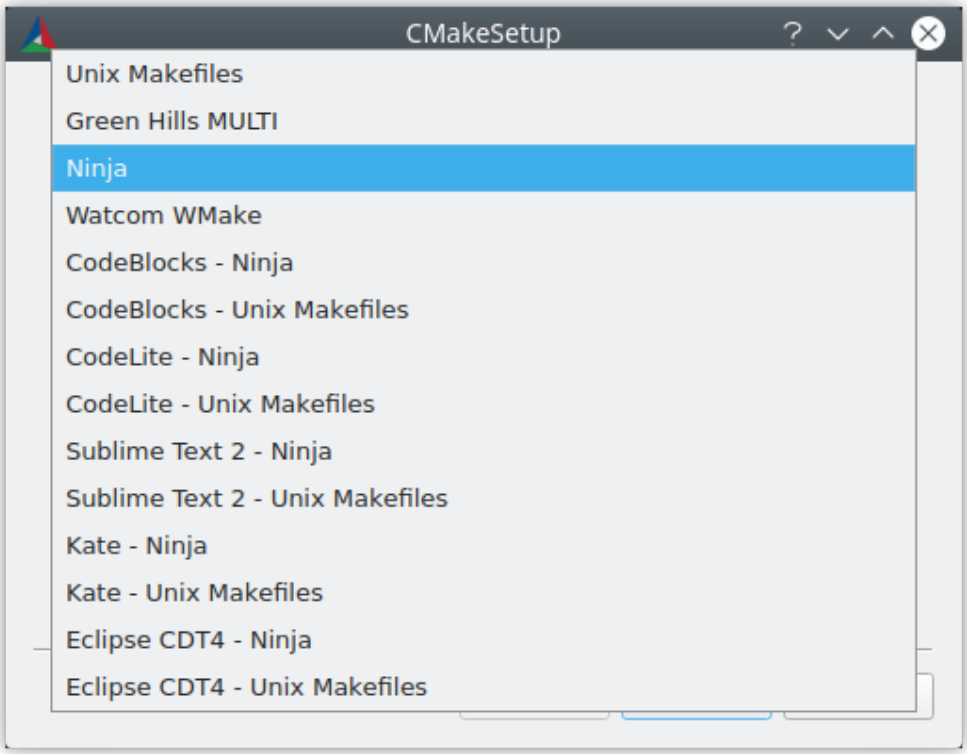
```
$ cmake .. -G "Visual Studio 16 2019" -A x64 -Thost=x64
```

18.2.3 Scelta di un generatore in cmake-gui

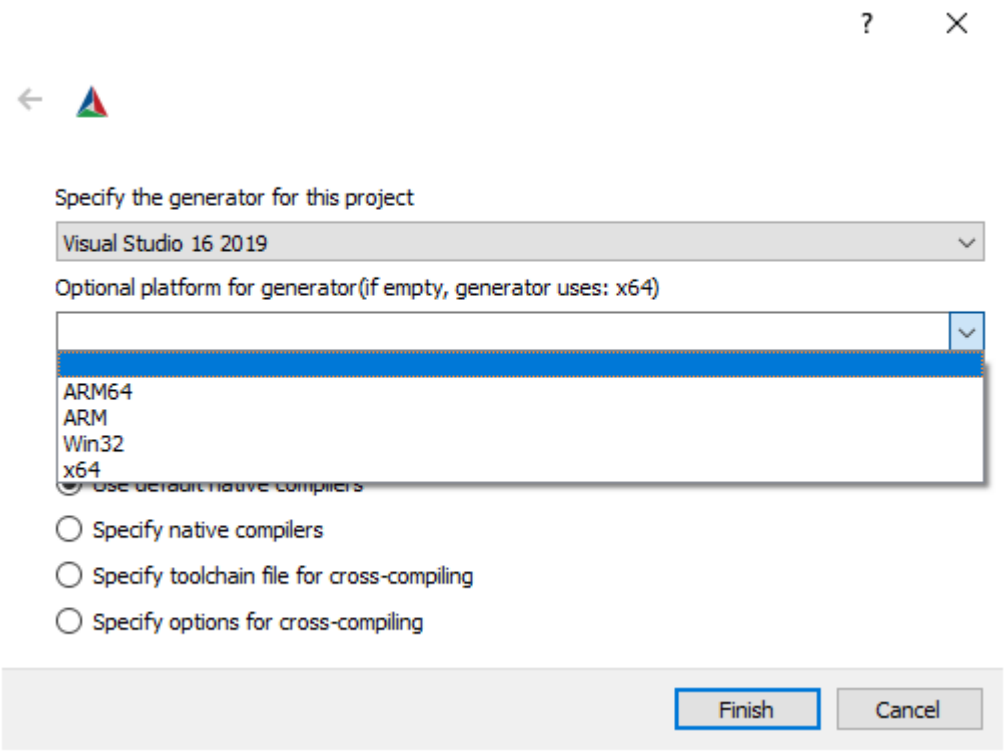
Il pulsante «Configure» attiva una nuova finestra di dialogo per selezionare il generatore CMake da utilizzare.



Tutti i generatori disponibili sulla riga di comando sono disponibili anche in `cmake-gui(1)`.



Quando si sceglie un generatore di Visual Studio, sono disponibili ulteriori opzioni per impostare un'architettura per la quale generare.



18.3 Impostazione delle Variabili di Build

I progetti software spesso richiedono l'impostazione di variabili sulla riga di comando quando si richiama CMake. Alcune delle variabili CMake più comunemente utilizzate sono elencate nella tabella seguente:

Variabile	Significato
CMAKE_PRI	Percorso per cercare `pacchetti dipendenti`
CMAKE_MOI	Percorso per cercare moduli CMake aggiuntivi
CMAKE_BUI	Configurazione di build, come `Debug` o `Release`, determinando i flag di debug/ottimizzazione. Ciò è rilevante solo per sistemi di compilazione a configurazione singola come `Makefile` e `Ninja`. I sistemi di build multiconfigurazione come quelli per Visual Studio e Xcode ignorano questa impostazione.
CMAKE_INS	Locazione in cui installare il software col target della build `install`
CMAKE_TOO	File contenente dati di cross-compilazione come `toolchains e sysroots`.
BUILD_SHA	Se creare librerie shared anziché statiche per i comandi `add_library` utilizzati senza un tipo
CMAKE_EXI	Genera un file `compile_commands.json` da utilizzare con strumenti basati su clang

Potrebbero essere disponibili altre variabili specifiche del progetto per controllare le build, come l'abilitazione o la disabilitazione dei componenti del progetto.

Non esiste alcuna convenzione fornita da CMake su come tali variabili vengono denominate tra i diversi sistemi di build forniti, tranne che le variabili con il prefisso `CMAKE_` di solito si

riferiscono alle opzioni fornite da CMake stesso e non dovrebbero essere utilizzate in opzioni di terze parti, che dovrebbero utilizzare invece il proprio prefisso. Il tool cmake-gui(1) può visualizzare le opzioni in gruppi definiti dal loro prefisso, quindi è logico che terze parti si assicurino di utilizzare un prefisso auto-coerente.

18.3.1 Impostazione delle variabili sulla riga di comando

Le variabili CMake possono essere impostate sulla riga di comando durante la creazione della build iniziale:

```
$ mkdir build
$ cd build
$ cmake .. -G Ninja -DCMAKE_BUILD_TYPE=Debug
```

o successivamente in una ulteriore invocazione di cmake(1):

```
$ cd build
$ cmake . -DCMAKE_BUILD_TYPE=Debug
```

Il flag -U può essere utilizzato per annullare l'impostazione delle variabili sulla riga di comando di cmake(1):

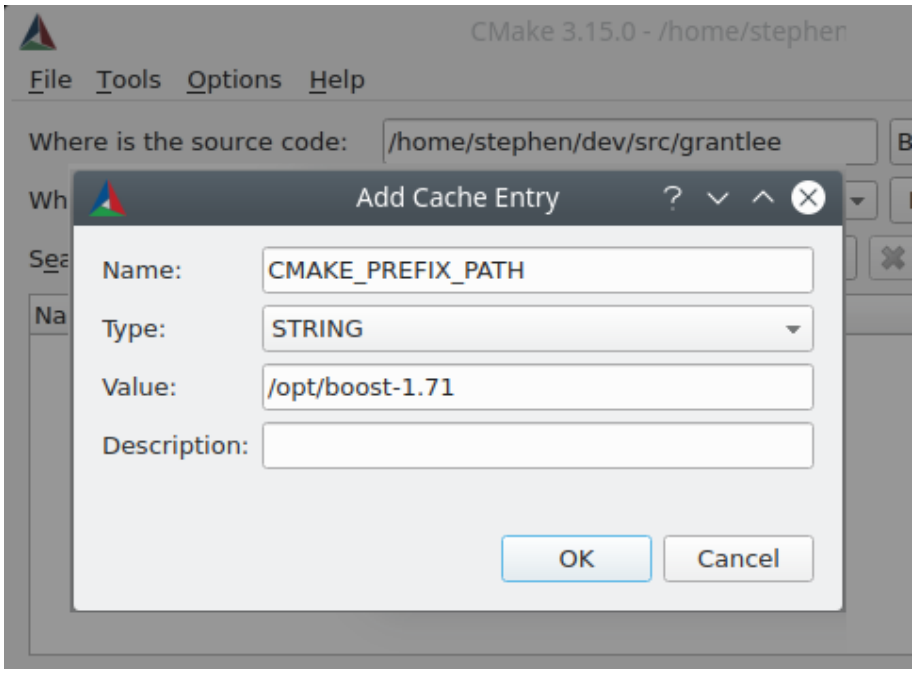
```
$ cd build
$ cmake . -UMyPackage_DIR
```

Un buildsystem CMake inizialmente creato sulla riga di comando può essere modificato utilizzando cmake-gui(1) e viceversa.

Il tool cmake(1) consente di specificare un file da utilizzare per popolare la cache iniziale utilizzando l'opzione -C. Ciò può essere utile per semplificare comandi e script che richiedono ripetutamente le stesse voci della cache.

18.3.2 Impostazione delle variabili con cmake-gui

Le variabili possono essere impostate in cmake-gui utilizzando il pulsante «Add Entry». Ciò attiva una nuova finestra di dialogo per impostare il valore della variabile.



La vista principale dell'interfaccia utente di `cmake-gui(1)` può essere utilizzata per modificare le variabili esistenti.

18.3.3 La Cache di CMake

Quando CMake viene eseguito, è necessario trovare le posizioni di compilatori, tool e dipendenze. Deve anche essere in grado di rigenerare in modo coerente un sistema di build per utilizzare gli stessi flag di compilazione/link e percorsi per le dipendenze. Tali parametri devono inoltre essere configurabili dall'utente perché sono percorsi e opzioni specifici del sistema dell'utente.

Quando viene eseguito per la prima volta, CMake genera un file `CMakeCache.txt` nella directory di build contenente coppie chiave-valore per tali artefatti. Il file della cache può essere visualizzato o modificato dall'utente eseguendo `cmake-gui(1)` o `ccmake(1)`. Gli strumenti forniscono un'interfaccia interattiva per riconfigurare il software e rigenerare il sistema di build, come necessario dopo aver modificato i valori memorizzati nella cache. A ciascuna voce della cache può essere associato un breve testo di aiuto che viene visualizzato nei tool dell'interfaccia utente.

Le voci della cache possono anche avere un tipo per indicare come dovrebbero essere presentate nell'interfaccia utente. Ad esempio, una voce della cache di tipo `BOOL` può essere modificata da una casella di controllo in un'interfaccia utente, una `STRING` può essere modificata in un campo di testo e un `FILEPATH` pur essendo simile a `STRING` dovrebbe anche fornire un modo per individuare i percorsi del filesystem utilizzando una finestra di dialogo per i file. Una voce di tipo `STRING` potrebbe consentire un elenco ristretto di valori che vengono poi forniti in un menù a discesa nell'interfaccia utente `cmake-gui(1)` (vedere la sezione della proprietà della cache `STRINGS`).

I file CMake forniti con un pacchetto software possono anche definire opzioni di attivazione/disattivazione booleane utilizzando il comando `option`. Il comando crea una voce nella cache che ha un testo di aiuto e un valore di default. Tali voci della cache sono in genere specifiche del software fornito e influiscono sulla configurazione della build, ad esempio se vengono creati test ed esempi, se compilare con le eccezioni abilitate, ecc.

18.4 Preimpostazioni

CMake riconosce un file, `CMakePresets.json`, e la sua controparte specifica dell'utente, `CMakeUserPresets.json`, per il salvataggio delle preimpostazioni per le configurazione di uso comune. Queste preimpostazioni possono impostare la directory di build, il generatore, le variabili della cache, le variabili di ambiente e altre opzioni della riga di comando. Tutte queste opzioni possono essere sovrascritte dall'utente. I dettagli completi del formato `CMakePresets.json` sono elencati nel manuale `cmake-presets(7)`.

18.4.1 Utilizzo delle preimpostazioni sulla riga di comando

Quando si utilizza lo strumento da riga di comando `cmake(1)`, è possibile richiamare un preset utilizzando l'opzione `--preset`. Se viene specificato `--preset`, il generatore e la directory di build non sono necessari, ma possono essere specificati per sovrascriverli. Ad esempio, se si ha il seguente file `CMakePresets.json`:

```
{
  "version": 1,
  "configurePresets": [
    {
      "name": "ninja-release",
      "binaryDir": "${sourceDir}/build/${presetName}",
      "generator": "Ninja",
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Release"
      }
    }
  ]
}
```

e si esegue quanto segue:

```
cmake -S /path/to/source --preset=ninja-release
```

Questo genererà una directory di build in `/path/to/source/build/ninja-release` col generatore `Ninja` generator e con `CMAKE_BUILD_TYPE` impostato su `Release`.

Per vedere l'elenco dei preset disponibili, si può eseguire:

```
cmake -S /path/to/source --list-presets
```

Questo elencherà le preimpostazioni disponibili in `/path/to/source/CMakePresets.json` e in `/path/to/source/CMakeUsersPresets.json` senza generare un albero di build.

18.4.2 Utilizzo dei preset in cmake-gui

Se un progetto ha dei preset disponibili, tramite `CMakePresets.json` o `CMakeUserPresets.json`, l'elenco dei preset apparirà in un menù a discesa in `cmake-gui(1)` tra la directory dei sorgenti e quella binaria. La scelta di una preimpostazione imposta la directory binaria, il generatore, le variabili di ambiente e le variabili della cache, ma tutte queste opzioni possono essere sovrascritte dopo aver selezionato una preimpostazione.

18.5 Invocare il Buildsystem

Dopo aver generato il sistema di build, è possibile creare il software richiamando il particolare tool. Nel caso dei generatori IDE, ciò può comportare il caricamento del file di progetto generato nell'IDE per richiamare la build.

CMake è a conoscenza dello strumento specifico necessario per invocare una build quindi, in generale, per creare un sistema di build o un progetto dalla riga di comando dopo la generazione, è possibile invocare il seguente comando nella directory di build:

```
$ cmake --build .
```

Il flag `--build` abilita una particolare modalità operativa per `cmake(1)`. Invoca il comando `CMAKE_MAKE_PROGRAM` associato al `generatore` o allo strumento di build configurato dall'utente.

La modalità `--build` accetta anche il parametro `--target` per specificare un particolare target da compilare, ad esempio una particolare libreria, un eseguibile o un target personalizzato, o un particolare target speciale come `install`:

```
$ cmake --build . --target myexe
```

La modalità `--build` accetta anche un parametro `--config` nel caso di generatori multi-config per specificare quale particolare configurazione buildare:

```
$ cmake --build . --target myexe --config Release
```

L'opzione `--config` non ha effetto se il generatore genera un sistema di build specifico per una configurazione scelta quando si invoca cmake con la variabile `CMAKE_BUILD_TYPE`.

Alcuni sistemi di build omettono i dettagli delle righe di comando invocate durante la build. Il flag `--verbose` può essere utilizzato per visualizzare tali righe di comando:

```
$ cmake --build . --target myexe --verbose
```

La modalità `--build` può anche passare particolari opzioni della riga di comando allo strumento di build sottostante elencandole dopo `--`. Ciò può essere utile per specificare le opzioni

per lo strumento di build, ad esempio per continuare dopo una build non riuscita, in cui CMake non fornisce un'interfaccia utente di alto livello.

Per tutti i generatori, è possibile eseguire lo strumento di build sottostante dopo aver richiamato CMake. Ad esempio, `make` può essere eseguito dopo aver generato con il generatore `Unix Makefiles` per invocare la build, o `ninja` dopo aver generato con il generatore `Ninja` ecc. I sistemi di build IDE di solito forniscono strumenti da riga di comando per creare un progetto che può anche essere invocato.

18.5.1 Selezionare un Target

Ogni eseguibile e libreria descritta nei file CMake è un target della build e il sistema di build può descrivere target personalizzati, sia per uso interno che per consumo da parte dell'utente, ad esempio per creare documentazione.

CMake fornisce alcuni target nativi per tutti i sistemi di build che forniscono file CMake.

`all`
: Il target di default utilizzato dai generatori `Makefile` e `Ninja`. Esegue la build di tutti i target nel buildsystem, eccetto quelli che sono esclusi dalla loro proprietà di target `EXCLUDE_FROM_ALL` o dalla proprietà di directory `EXCLUDE_FROM_ALL`. A questo scopo viene utilizzato il nome `ALL_BUILD` per i generatori Xcode e Visual Studio.

`help`
: Elenca i target disponibili per la build. Questo target è disponibile quando si usa il generatore `Unix Makefiles` o `Ninja` e l'output esatto è specifico del tool.

`clean`
: Elimina i file oggetto creati e altri file di output. I generatori basati su `Makefile` creano un target `pulito` per ogni directory, in modo che una singola directory possa essere pulita. `Ninja` fornisce il proprio sistema granulare `-t clean`.

`test`
: Esegue i test. Questo target è disponibile automaticamente solo se i file CMake forniscono test basati su CTest. Vedere anche *Running Tests*.

`install`
: Installa il software. Questo target è disponibile automaticamente solo se il software definisce le regole di installazione col comando `install`. Vedere anche *Software Installation*.

`package`
: Crea un pacchetto binario. Questo target è disponibile automaticamente solo se i file CMake forniscono pacchetti basati su CPack.

`package_source`
: Creare un pacchetto sorgente Questo target è disponibile automaticamente solo se i file CMake forniscono pacchetti basati su CPack.

Per i sistemi basati su `Makefile`, vengono fornite varianti `/fast` dei target binari creati. Le varianti `/fast` vengono utilizzate per creare il target specificato senza riguardo per le sue dipendenze. Le dipendenze non vengono controllate e non vengono re-buildate se non aggiornate. Il

generatore Ninja è sufficientemente veloce nel controllare che tali dipendenze dei target non siano fornite per quel generatore.

I sistemi basati su Makefile forniscono anche target di build per pre-processare, assemblare e compilare singoli file in una particolare directory.

```
$ make foo.cpp.i
$ make foo.cpp.s
$ make foo.cpp.o
```

L'estensione del file è inclusa nel nome del target perché potrebbe esistere un altro file con lo stesso nome ma con un'estensione diversa. Tuttavia, vengono forniti anche build-target senza estensione del file.

```
$ make foo.i
$ make foo.s
$ make foo.o
```

Nei sistemi di compilazione che contengono foo.c e foo.cpp, la build del target foo.i preprocesserà entrambi i file.

18.5.2 Specificare di un Programma di Build

Il programma richiamato dalla modalità --build è determinato dalla variabile CMAKE_MAKE_PROGRAM. Per la maggior parte dei generatori non è necessario configurare il programma specifico.

Generatore	Programma make di default	Alternative
XCode	xcodebuild	
Unix Makefiles	make	
NMake Makefiles	nmake	jom
NMake Makefiles JOM	jom	nmake
MinGW Makefiles	mingw32-make	
MSYS Makefiles	make	
Ninja	ninja	
Visual Studio	msbuild	
Watcom WMake	wmake	

Il tool jom è in grado di leggere makefile del tipo NMake e di compilarli in parallelo, mentre nmake compila sempre serialmente. Dopo aver generato con il generatore NMake Makefiles un utente può eseguire jom invece di nmake. La modalità --build utilizzerebbe anche jom se la CMAKE_MAKE_PROGRAM fosse impostata su jom durante l'utilizzo del generatore NMake Makefiles, e per comodità, viene fornito il generatore NMake Makefiles JOM per trovare jom nel modo normale e usarlo come CMAKE_MAKE_PROGRAM. Per completezza, nmake è uno strumento alternativo che può elaborare l'output del generatore NMake Makefiles JOM, ma farlo peggiorerebbe le cose.

18.6 Installazione Software

La variabile CMAKE_INSTALL_PREFIX può essere impostata nella cache CMake per specificare dove installare il software. Se il software fornito dispone di regole di installazione, specificate utilizzando il comando install, verranno installati gli artefatti in quel prefisso. Su Windows, il percorso di installazione predefinito corrisponde alla directory di sistema ProgramFiles che potrebbe essere specifica dell'architettura. Sugli host Unix, /usr/local è il percorso di installazione predefinito.

La variabile CMAKE_INSTALL_PREFIX si riferisce sempre al prefisso di installazione sul filesystem target.

Negli scenari di cross-compilazione o di impacchettamento in cui il sysroot è di sola lettura o dove il sysroot dovrebbe comunque rimanere intatto, la variabile CMAKE_STAGING_PREFIX può essere impostata su una posizione per installare effettivamente i file.

I comandi:

```
$ cmake .. -DCMAKE_INSTALL_PREFIX=/usr/local \
  -DCMAKE_SYSROOT=$HOME/root \
  -DCMAKE_STAGING_PREFIX=/tmp/package
$ cmake --build .
$ cmake --build . --target install
```

comportano l'installazione di file in percorsi come /tmp/package/lib/libfoo.so sul computer host. La posizione /usr/local sulla macchina host non è interessata.

Alcuni software forniti potrebbero specificare regole di uninstall, ma CMake non genera tali regole per default.

18.7 Esecuzione di Test

ctest(1) viene fornito con la distribuzione CMake per eseguire i test forniti e riportare i risultati. Il build-target test viene fornito per eseguire tutti i test disponibili, ma ctest(1) consente un controllo granulare su quali test eseguire, come eseguirli e come riportare i risultati. L'esecuzione di ctest(1) nella directory build equivale a eseguire il target test:

```
$ ctest
```

È possibile passare un'espressione regolare per eseguire solo i test che corrispondono all'espressione. Per eseguire solo i test con Qt nel nome:

```
$ ctest -R Qt
```

I test possono essere esclusi anche tramite espressioni regolari. Per eseguire solo i test senza Qt nel nome:

```
$ ctest -E Qt
```

I test possono essere eseguiti in parallelo passando gli argomenti `-j` a `ctest(1)`:

```
$ ctest -R Qt -j8
```

In alternativa, è possibile impostare la variabile d'ambiente `CTEST_PARALLEL_LEVEL` per evitare la necessità di passare `-j`.

Per default `ctest(1)` non stampa l'output dei test. L'argomento della riga di comando `-V` (o `--verbose`) abilita la modalità «verbose» per stampare l'output di tutti i test. L'opzione `--output-on-failure` stampa l'output del test solo per quelli falliti. La variabile d'ambiente `CTEST_OUTPUT_ON_FAILURE` può essere impostata su 1 come alternativa al passaggio dell'opzione `--output-on-failure` a `ctest(1)`.

CAPITOLO 19

Guida all'Uso delle Dipendenze

19.1 Introduzione

Per gli sviluppatori che desiderano utilizzare CMake per usare un pacchetto binario di terze parti, esistono molteplici possibilità su come farlo in modo ottimale, a seconda di quanto sia compatibile con CMake la libreria di terze parti.

I file CMake forniti con un pacchetto software contengono istruzioni per trovare ciascuna dipendenza della build. Alcune dipendenze di build sono facoltative in quanto questa potrebbe avere esito positivo con un set di funzionalità diverso se la dipendenza è mancante e alcune dipendenze sono obbligatorie. CMake cerca posizioni note per ciascuna dipendenza e il software può fornire ulteriori suggerimenti o posizioni a CMake per trovare ciascuna dipendenza.

Se una dipendenza richiesta non viene trovata da `cmake(1)`, la cache viene popolata con una voce che contiene un valore `NOTFOUND`. Questo valore può essere sostituito specificandolo sulla riga di comando o nello strumento `ccmake(1)` oi in `cmake-gui(1)`. Consultare la guida `User Interaction Guide` per ulteriori informazioni sull'impostazione delle voci della cache.

19.2 Librerie che Forniscono Pacchetti di file di configurazione

Il modo più conveniente per una terza parte di fornire file binari di libreria da utilizzare con CMake è fornire Config-file Packages. Questi pacchetti sono file di testo forniti con la libreria che istruiscono CMake su come utilizzare i file binari della libreria e gli header associati, gli strumenti di supporto e le macro CMake fornite dalla libreria.

I file di configurazione di solito si trovano in una directory il cui nome corrisponde al pattern `lib/cmake/<PackageName>`, anche se potrebbero trovarsi invece in altre posizioni. Il

<PackageName> corrisponde all'uso nel codice CMake con il comando find_package come find_package(PackageName REQUIRED).

La directory lib/cmake/<PackageName> conterrà un file denominato <PackageName>Config.cmake o <PackageName>-config.cmake. Questo è il punto di ingresso al pacchetto per CMake. Nella directory potrebbe anche esistere un file facoltativo separato denominato <PackageName>ConfigVersion.cmake. Questo file viene utilizzato da CMake per determinare se la versione del pacchetto di terze parti soddisfa gli usi del comando find_package che specifica i vincoli di versione. È facoltativo specificare una versione quando si usa find_package, anche se è presente un file ConfigVersion.

Se il file Config.cmake viene trovato e la versione specificata facoltativamente è soddisfatta, allora il comando CMake find_package considera il pacchetto da trovare e si presuppone che l'intero pacchetto della libreria sia completo come progettato.

Potrebbero esserci file aggiuntivi che forniscono macro CMake o Imported Targets da utilizzare. CMake non applica alcuna convenzione sui nomi per questi file. Sono correlati al file Config primario tramite l'uso del comando CMake include.

Invocare CMake con l'intento di utilizzare un pacchetto di file binari di terze parti richiede che i comandi cmake find_package riescano a trovare il pacchetto. Se la posizione del pacchetto è in una directory nota a CMake, la chiamata find_package dovrebbe riuscire. Le directory note a cmake sono specifiche della piattaforma. Ad esempio, i pacchetti installati su Linux con un gestore di pacchetti di sistema standard verranno trovati automaticamente nel prefisso /usr. Allo stesso modo, i pacchetti installati in Program Files su Windows verranno trovati automaticamente.

I pacchetti che non vengono trovati automaticamente si trovano in posizioni non prevedibili da CMake come /opt/mylib o $HOME/dev/prefix. Questa è una situazione normale e CMake fornisce agli utenti diversi modi per specificare dove trovare tali librerie.

La variabile CMAKE_PREFIX_PATH può essere *impostata quando si invoca CMake*. Viene trattato come un elenco di percorsi in cui cercare Config-file Packages. Un pacchetto installato in /opt/somepackage tipicamente installerà file di configurazione come /opt/somepackage/lib/cmake/somePackage/SomePackageConfig.cmake. In tal caso, /opt/somepackage dovrebbe essere aggiunto a CMAKE_PREFIX_PATH.

La variabile d'ambiente CMAKE_PREFIX_PATH può anche essere popolata con prefissi per cercare pacchetti. Come la variabile di ambiente PATH, questo è un elenco e deve utilizzare il separatore di voci dell'elenco di variabili di ambiente specifico della piattaforma (: su Unix e ; su Windows).

La variabile CMAKE_PREFIX_PATH risulta comoda nei casi in cui è necessario specificare più prefissi o quando sono disponibili più file binari di pacchetti diversi nello stesso prefisso. I percorsi dei pacchetti possono anche essere specificati impostando variabili corrispondenti a <PackageName>_DIR, come SomePackage_DIR. Notare che questo non è un prefisso ma dovrebbe essere un percorso completo a una directory contenente un file di pacchetto in stile configurazione, come /opt/somepackage/lib/cmake/SomePackage/ nell'esempio precedente.

19.3 Target Importati dai Pacchetti

Un pacchetto di terze parti che fornisce pacchetti di file di configurazione può anche fornire Imported Targets. Questi verranno specificati in file contenenti percorsi di file specifici della configurazione rilevanti per il pacchetto, come versioni di debug e di rilascio delle librerie.

Spesso la documentazione dei pacchetti di terze parti indicherà i nomi dei target importati disponibili dopo un riuscito `find_package` per una libreria. I nomi dei target importati possono essere utilizzati col comando `target_link_libraries`.

Un esempio completo che fa un semplice utilizzo di una libreria di terze parti potrebbe essere simile a:

```
cmake_minimum_required(VERSION 3.10)
project(MyExeProject VERSION 1.0.0)

find_package(SomePackage REQUIRED)
add_executable(MyExe main.cpp)
target_link_libraries(MyExe PRIVATE SomePrefix::LibName)
```

Consultare `cmake-buildsystem(7)` per ulteriori informazioni sullo sviluppo di un sistema di build CMake.

19.3.1 Librerie Senza Pacchetti di File di Configurazione

Le librerie di terze parti che non forniscono pacchetti di file di configurazione possono comunque essere trovate col comando `find_package`, se è disponibile un file `FindSomePackage.cmake`.

Questi pacchetti di file-modulo sono diversi dai pacchetti di config-file in quanto:

1. Non dovrebbero essere forniti da terzi, tranne forse sotto forma di documentazione
2. La disponibilità di un file `Find<PackageName>.cmake` non indica la disponibilità dei binari stessi.
3. CMake non cerca in `CMAKE_PREFIX_PATH` i file `Find<PackageName>.cmake`. Invece CMake cerca tali file nella variabile `CMAKE_MODULE_PATH`. È normale che gli utenti impostino `CMAKE_MODULE_PATH` durante l'esecuzione di CMake ed è comune che i progetti CMake aggiungano a `CMAKE_MODULE_PATH` per consentire l'uso di pacchetti di file di moduli locali.
4. CMake fornisce i file `Find<PackageName>.cmake` per alcuni `pacchetti di terze parti` per comodità nei casi in cui la terza parte non fornisce direttamente i pacchetti di file di configurazione. Questi file rappresentano un onere di manutenzione per CMake, quindi i nuovi moduli «Find» generalmente non vengono più aggiunti a CMake. Le terze parti dovrebbero fornire pacchetti di file di configurazione invece di fare affidamento su un modulo «Find» fornito da CMake.

I pacchetti di file modulo possono anche fornire Imported Targets. Un esempio completo che trova un pacchetto di questo tipo potrebbe essere simile a:

```
cmake_minimum_required(VERSION 3.10)
project(MyExeProject VERSION 1.0.0)

find_package(PNG REQUIRED)

# Add path to a FindSomePackage.cmake file
list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")
find_package(SomePackage REQUIRED)

add_executable(MyExe main.cpp)
target_link_libraries(MyExe PRIVATE
    PNG::PNG
    SomePrefix::LibName
)
```

La variabile `<PackageName>_ROOT` viene cercata anche come prefisso per le chiamate `find_package` utilizzando pacchetti di file modulo come `FindSomePackage`.

CAPITOLO 20

Guida all'Importazione e all'Esportazione

20.1 Introduzione

In questa guida presenteremo il concetto di target `IMPORTED` e mostreremo come importare file eseguibili o librerie esistenti dal disco in un progetto CMake. Mostreremo poi come CMake supporta l'esportazione di target da un progetto basato su CMake e l'importazione in un altro. Infine, mostreremo come creare un pacchetto di un progetto con un file di configurazione per consentire una facile integrazione in altri progetti CMake. Questa guida e il codice sorgente di esempio completo si trovano nella directory `Help/guide/importing-exporting` dell'albero del codice sorgente di CMake.

20.2 Importazione dei Target

I target `IMPORTED` vengono utilizzati per convertire file all'esterno di un progetto CMake in target logici all'interno del progetto. I target `IMPORTED` vengono creati utilizzando l'opzione `IMPORTED` dei comandi `add_executable` e `add_library`. Nessun file di build viene generato per i target `IMPORTED` targets. Una volta importati, i target `IMPORTED` possono essere referenziati come qualsiasi altro target all'interno del progetto e forniscono un riferimento comodo e flessibile a eseguibili e librerie esterne.

Per default, il nome del target `IMPORTED` ha uno «scope» nella directory in cui è stato creato e al di sotto. Possiamo utilizzare l'opzione `GLOBAL` per estendere lo scope in modo che il target sia accessibile globalmente nel sistema di build.

I dettagli sul target `IMPORTED` vengono specificati impostando le proprietà i cui nomi iniziano con `IMPORTED_` e `INTERFACE_`. Ad esempio, `IMPORTED_LOCATION` contiene il percorso completo del target su disco.

20.2.1 Importazione di Eseguibili

Per iniziare, esamineremo un semplice esempio che crea un target eseguibile IMPORTED e poi vi fa riferimento dal comando add_custom_command.

Avremo bisogno di fare qualche configurazione per iniziare. Vogliamo creare un eseguibile che quando viene eseguito crea un file elementare main.cc nella directory corrente. I dettagli di questo progetto non sono importanti. Passare a Help/guide/importing-exporting/MyExe, creare una directory di build, eseguire cmake e creare e installare il progetto.

```
$ cd Help/guide/importing-exporting/MyExe
$ mkdir build
$ cd build
$ cmake ..
$ cmake --build .
$ cmake --install . --prefix <install location>
$ <install location>/myexe
$ ls
[...] main.cc [...]
```

Ora possiamo importare questo eseguibile in un altro progetto CMake. Il codice sorgente di questa sezione è disponibile in Help/guide/importing-exporting/Importing. Nel file CMakeLists, si usa il comando add_executable per creare un nuovo target chiamato myexe. Usare l'opzione IMPORTED per indicare a CMake che questo target fa riferimento a un file eseguibile situato all'esterno del progetto. Non verrà generata alcuna regola per costruirlo e la proprietà del target IMPORTED verrà impostata su true.

```
add_executable(myexe IMPORTED)
```

Successivamente, si imposta la proprietà IMPORTED_LOCATION del target utilizzando il comando set_property. Questo indicherà a CMake la posizione del target sul disco. Potrebbe essere necessario modificare il percorso in <install location> specificato nel passaggio precedente.

```
set_property(TARGET myexe PROPERTY
             IMPORTED_LOCATION "../InstallMyExe/bin/myexe")
```

Ora possiamo fare riferimento a questo target IMPORTED proprio come qualsiasi target creato all'interno del progetto. In questo caso, immaginiamo di voler utilizzare il file sorgente generato nel nostro progetto. Si usa il target IMPORTED nel comando add_custom_command:

```
add_custom_command(OUTPUT main.cc COMMAND myexe)
```

Poiché COMMAND specifica un nome di target dell'eseguibile, verrà automaticamente sostituito dalla posizione dell'eseguibile fornita dalla proprietà IMPORTED_LOCATION di cui sopra.

Infine, si usa l'output di add_custom_command:

```
add_executable(mynewexe main.cc)
```

20.2.2 Importazione di Librerie

In modo simile, è possibile accedere alle librerie di altri progetti tramite target `IMPORTED`.

Nota: Il codice sorgente completo per gli esempi in questa sezione non viene fornito e viene lasciato come esercizio per il lettore.

Nel file CMakeLists, si aggiunge una libreria `IMPORTED` e se ne specifica la posizione sul disco:

```
add_library(foo STATIC IMPORTED)
set_property(TARGET foo PROPERTY
             IMPORTED_LOCATION "/path/to/libfoo.a")
```

Poi si usa la libreria `IMPORTED` all'interno del progetto:

```
add_executable(myexe src1.c src2.c)
target_link_libraries(myexe PRIVATE foo)
```

Su Windows, un file .dll e la relativa libreria di importazione .lib possono essere importati insieme:

```
add_library(bar SHARED IMPORTED)
set_property(TARGET bar PROPERTY
             IMPORTED_LOCATION "c:/path/to/bar.dll")
set_property(TARGET bar PROPERTY
             IMPORTED_IMPLIB "c:/path/to/bar.lib")
add_executable(myexe src1.c src2.c)
target_link_libraries(myexe PRIVATE bar)
```

Una libreria con più configurazioni può essere importata con un unico target:

```
find_library(math_REL NAMES m)
find_library(math_DBG NAMES md)
add_library(math STATIC IMPORTED GLOBAL)
set_target_properties(math PROPERTIES
  IMPORTED_LOCATION "${math_REL}"
  IMPORTED_LOCATION_DEBUG "${math_DBG}"
  IMPORTED_CONFIGURATIONS "RELEASE;DEBUG"
)
add_executable(myexe src1.c src2.c)
target_link_libraries(myexe PRIVATE math)
```

Il sistema di build generato linkerà `myexe` a `m.lib` nella build nella configurazione di release e a `md.lib` nella configurazione di debug.

20.3 Esportazione dei Target

Anche se i target IMPORTED da soli sono utili, richiedono comunque che il progetto che li importa conosca le posizioni dei loro file sul disco. Il vero potere dei target IMPORTED è quando il progetto che ne fornisce i file fornisce anche un file CMake per importarli. Un progetto può essere configurato per produrre le informazioni necessarie in modo che possa essere facilmente utilizzato da altri progetti CMake da una directory di build, da un'installazione locale o quando inserito in un pacchetto.

Nelle restanti sezioni, esamineremo passo dopo passo una serie di progetti di esempio. Il primo progetto creerà e installerà una libreria e la configurazione CMake corrispondente e i file del pacchetto. Il secondo progetto utilizzerà il pacchetto generato.

Iniziamo guardando il progetto MathFunctions nella directory Help/guide/importing-exporting/MathFunctions. Qui abbiamo un file di header MathFunctions.h che dichiara una funzione sqrt:

```
#pragma once

namespace MathFunctions {
double sqrt(double x);
}
```

E un file sorgente corrispondente MathFunctions.cxx:

```
#include "MathFunctions.h"

#include <cmath>

namespace MathFunctions {
double sqrt(double x)
{
  return std::sqrt(x);
}
}
```

Non ci si preoccupi troppo delle specifiche dei file C++, sono pensati solo per essere un semplice esempio che verrà compilato ed eseguito su molti sistemi di build.

Ora possiamo creare un file CMakeLists.txt per il progetto MathFunctions. Si inizia specificando la versione di cmake_minimum_required e il nome del project:

```
cmake_minimum_required(VERSION 3.15)
project(MathFunctions)

# specify the C++ standard
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)
```

Si crea una libreria chiamata MathFunctions col comando add_library:

```
add_library(MathFunctions STATIC MathFunctions.cxx)
```

Poi si usa il comando `target_include_directories` per specificare le directory di inclusione per il target:

```
target_include_directories(MathFunctions
                           PUBLIC
                           "$<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_
↪DIR}>"
                           "$<INSTALL_INTERFACE:include>"
)
```

Dobbiamo dire a CMake che vogliamo utilizzare directory di inclusione diverse a seconda che stiamo costruendo la libreria o utilizzandola da una posizione installata. Se non lo facciamo, quando CMake crea le informazioni di esportazione esporterà un percorso specifico per la directory di build corrente e non sarà valido per altri progetti. Possiamo usare `generator expressions` per specificare che se stiamo «buildando» la libreria deve includere la directory sorgente corrente. Altrimenti, una volta installato, include la directory `include`. Consultare la sezione *Creazione di Pacchetti Rilocabili* per maggiori dettagli.

I comandi `install(TARGETS)` e `install(EXPORT)` lavorano insieme per installare entrambi i target (una libreria nel nostro caso) e un file CMake progettato per semplificare l'importazione dei target in un altro progetto CMake.

Innanzitutto, nel comando `install(TARGETS)` specificheremo il target, il nome `EXPORT` e le destinazioni che indicano a CMake dove installare i target.

```
install(TARGETS MathFunctions
        EXPORT MathFunctionsTargets
        LIBRARY DESTINATION lib
        ARCHIVE DESTINATION lib
        RUNTIME DESTINATION bin
        INCLUDES DESTINATION include
)
```

Qui, l'opzione `EXPORT` dice a CMake di creare un'esportazione chiamata `MathFunctionsTargets`. I target `IMPORTED` generati hanno proprietà appropriate impostate per definire i loro requisiti di utilizzo, come `INTERFACE_INCLUDE_DIRECTORIES`, `INTERFACE_COMPILE_DEFINITIONS` e altre rilevanti proprietà native [built-in] `INTERFACE_`. La variante `INTERFACE` delle proprietà definite dall'utente elencate in `COMPATIBLE_INTERFACE_STRING` e altre Compatible Interface Properties vengono propagate anche ai target `IMPORTED` generati. Ad esempio, in questo caso, il target `IMPORTED` avrà la sua proprietà `INTERFACE_INCLUDE_DIRECTORIES` popolata con la directory specificata dalla proprietà `INCLUDES DESTINATION`. Poiché è stato fornito un percorso relativo, viene trattato come relativo a `CMAKE_INSTALL_PREFIX`.

Notare, *non* abbiamo ancora chiesto a CMake di installare l'export.

Non vogliamo dimenticare di installare il file header `MathFunctions.h` col comando `install(FILES)`. Il file di intestazione dovrebbe essere installato nella directory `include`, come specificato dal comando `target_include_directories` sopra.

```
install(FILES MathFunctions.h DESTINATION include)
```

Ora che la libreria `MathFunctions` e il file header sono installati, dobbiamo anche installare esplicitamente i dettagli di esportazione di `MathFunctionsTargets`. Utilizzare il comando `install(EXPORT)` per esportare i target in `MathFunctionsTargets`, come definito dal comando `install(TARGETS)`.

```
install(EXPORT MathFunctionsTargets
        FILE MathFunctionsTargets.cmake
        NAMESPACE MathFunctions::
        DESTINATION lib/cmake/MathFunctions
)
```

Questo comando genera il file `MathFunctionsTargets.cmake` e provvede ad installarlo in `lib/cmake`. Il file contiene codice utilizzabile dai downstream per importare tutti i target elencati nel comando install dall'albero di installazione.

L'opzione `NAMESPACE` anteporrà `MathFunctions::` ai nomi dei target mentre vengono scritti nel file export. Questa convenzione dei doppi due-punti fornisce a CMake un suggerimento sul fatto che il nome è un target `IMPORTED` quando viene utilizzato dai progetti downstream. In questo modo, CMake può emettere un messaggio diagnostico se il pacchetto che lo fornisce non è stato trovato.

Il file di esportazione generato contiene il codice che crea una libreria `IMPORTED`.

```
# Create imported target MathFunctions::MathFunctions
add_library(MathFunctions::MathFunctions STATIC IMPORTED)

set_target_properties(MathFunctions::MathFunctions PROPERTIES
  INTERFACE_INCLUDE_DIRECTORIES "${_IMPORT_PREFIX}/include"
)
```

Questo codice è molto simile all'esempio che abbiamo creato manualmente nella sezione *Importazione di Librerie*. Notare che `${_IMPORT_PREFIX}` viene calcolato in relazione alla posizione del file.

Un progetto esterno può caricare questo file col comando `include` e fare riferimento alla libreria `MathFunctions` dall'albero di installazione come se fosse creata nel proprio albero. Per esempio:

```
1 include(${INSTALL_PREFIX}/lib/cmake/MathFunctionTargets.cmake)
2 add_executable(myexe src1.c src2.c )
3 target_link_libraries(myexe PRIVATE MathFunctions::MathFunctions)
```

La riga 1 carica il file CMake del target. Anche se abbiamo esportato un solo target, questo file può importare un numero qualsiasi di target. Le loro posizioni vengono calcolate rispetto a quella del file in modo che l'albero di installazione possa essere facilmente spostato. La riga 3

fa riferimento alla libreria importata `MathFunctions`. Il sistema di build risultante linkerà la libreria dalla posizione di installazione.

Gli eseguibili possono anche essere esportati e importati utilizzando lo stesso processo.

Allo stesso nome di esportazione può essere associato un numero qualsiasi di installazioni di target. I nomi di esportazione sono considerati globali, quindi qualsiasi directory può contribuire a un'installazione di un target. Il comando `install(EXPORT)` deve essere chiamato solo una volta per installare un file che fa riferimento a tutti i target. Di seguito è riportato un esempio di come è possibile combinare più esportazioni in un unico file export, anche se si trovano in sottodirectory diverse del progetto.

```
# A/CMakeLists.txt
add_executable(myexe src1.c)
install(TARGETS myexe DESTINATION lib/myproj
        EXPORT myproj-targets)

# B/CMakeLists.txt
add_library(foo STATIC foo1.c)
install(TARGETS foo DESTINATION lib EXPORTS myproj-targets)

# Top CMakeLists.txt
add_subdirectory (A)
add_subdirectory (B)
install(EXPORT myproj-targets DESTINATION lib/myproj)
```

20.3.1 Creazione di Pacchetti

A questo punto, il progetto `MathFunctions` sta esportando le informazioni richieste del target per essere utilizzate da altri progetti. Possiamo rendere questo progetto ancora più semplice da utilizzare per altri progetti generando un file di configurazione in modo che il comando CMake `find_package` possa trovare il nostro progetto.

Per iniziare, dovremo fare alcune aggiunte al file `CMakeLists.txt`. Innanzitutto, si include il modulo `CMakePackageConfigHelpers` per ottenere l'accesso ad alcune funzioni di supporto per la creazione dei file di configurazione.

```
include(CMakePackageConfigHelpers)
```

Poi creeremo un file di configurazione e un file della versione del pacchetto.

Creazione di un File di Configurazione del Pacchetto

Usare il comando `configure_package_config_file` fornito da `CMakePackageConfigHelpers` per generare il file di configurazione del pacchetto. Notare che questo comando dovrebbe essere usato al posto del semplice comando `configure_file`. Aiuta a garantire che il pacchetto risultante sia riposizionabile evitando percorsi codificati nel file di configurazione installato. Il percorso fornito a `INSTALL_DESTINATION` deve essere la destinazione in cui verrà installato il file `MathFunctionsConfig.cmake`. Esamineremo il contenuto del file di configurazione del pacchetto nella prossima sezione.

```
configure_package_config_file(${CMAKE_CURRENT_SOURCE_DIR}/Config.cmake.
↪in
  "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfig.cmake"
  INSTALL_DESTINATION lib/cmake/MathFunctions
)
```

Installare i file di configurazione generati col comando `INSTALL(files)`. Sia `MathFunctionsConfigVersion.cmake` che `MathFunctionsConfig.cmake` sono installati nella stessa posizione, completando il pacchetto.

```
install(FILES
          "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfig.cmake"
          "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfigVersion.cmake
↪"
        DESTINATION lib/cmake/MathFunctions
)
```

Ora dobbiamo creare il file di configurazione del pacchetto stesso. In questo caso, il file `Config.cmake.in` è molto semplice ma sufficiente per consentire ai downstream di utilizzare i target `IMPORTED`.

```
@PACKAGE_INIT@

include("${CMAKE_CURRENT_LIST_DIR}/MathFunctionsTargets.cmake")

check_required_components(MathFunctions)
```

La prima riga del file contiene solo la stringa `@PACKAGE_INIT@`. Questo si espande quando il file viene configurato e consente l'uso di percorsi riposizionabili col prefisso `PACKAGE_`. Fornisce inoltre le macro `set_and_check()` e `check_required_components()`.

La macro helper `check_required_components` garantisce che tutti i componenti richiesti e non opzionali siano stati trovati controllando le variabili `<Package>_<Component>_FOUND` per tutti i componenti richiesti. Questa macro dovrebbe essere richiamata alla fine del file di configurazione del pacchetto anche se il pacchetto non ha alcun componente. In questo modo, CMake può assicurarsi che il progetto downstream non abbia specificato componenti inesistenti. Se `check_required_components` fallisce, la variabile `<Package>_FOUND` viene impostata su FALSE e il pacchetto viene considerato non trovato.

La macro `set_and_check()` dovrebbe essere utilizzata nei file di configurazione invece del normale comando `set()` per impostare directory e posizioni dei file. Se un file o una directory a cui si fa riferimento non esiste, la macro fallirà.

Se delle macro devono essere fornite dal pacchetto `MathFunctions`, dovrebbero trovarsi in un file separato installato nella stessa posizione del file `MathFunctionsConfig.cmake` e incluse da lì.

Tutte le dipendenze richieste di un pacchetto si devono trovare anche nel file di configurazione del pacchetto. Immaginiamo di aver bisogno della libreria `Stats` nel nostro progetto. Nel file CMakeLists, aggiungeremmo:

```
find_package(Stats 2.6.4 REQUIRED)
target_link_libraries(MathFunctions PUBLIC Stats::Types)
```

Dato che il target `Stats::Types` è una dipendenza `PUBLIC` di `MathFunctions`, i downstream devono anche trovare il pacchetto `Stats` e linkarlo alla libreria `Stats::Types`. Per garantire ciò, nel file di configurazione dovrebbe essere presente il pacchetto `Stats`.

```
include(CMakeFindDependencyMacro)
find_dependency(Stats 2.6.4)
```

La macro `find_dependency` dal modulo `CMakeFindDependencyMacro` aiuta a propagare se il pacchetto è `REQUIRED` o `QUIET`, ecc. La macro `find_dependency` imposta anche `MathFunctions_FOUND` a `False` se la dipendenza non viene trovata, insieme a una diagnostica che indica che il pacchetto `MathFunctions` non può essere utilizzato senza il pacchetto `Stats`.

Esercizio: Aggiungere la libreria richiesta al progetto``MathFunctions``.

Creazione di un File di Versione del Pacchetto

Il modulo `CMakePackageConfigHelpers` fornisce il comando `write_basic_package_version_file` per creare un semplice file di versione del pacchetto. Questo file viene letto da CMake quando viene chiamato `find_package` per determinare la compatibilità con la versione richiesta e per impostare alcune variabili specifiche della versione come `<PackageName>_VERSION`, `<PackageName>_VERSION_MAJOR`, `<PackageName>_VERSION_MINOR`, ecc. Consultare la documentazione `cmake-packages` per maggiori dettagli.

```
set(version 3.4.1)

set_property(TARGET MathFunctions PROPERTY VERSION ${version})
set_property(TARGET MathFunctions PROPERTY SOVERSION 3)
set_property(TARGET MathFunctions PROPERTY
  INTERFACE_MathFunctions_MAJOR_VERSION 3)
set_property(TARGET MathFunctions APPEND PROPERTY
  COMPATIBLE_INTERFACE_STRING MathFunctions_MAJOR_VERSION
)
```

(continues on next page)

(continua dalla pagina precedente)

```
# generate the version file for the config file
write_basic_package_version_file(
  "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfigVersion.cmake"
  VERSION "${version}"
  COMPATIBILITY AnyNewerVersion
)
```

Nel nostro esempio, `MathFunctions_MAJOR_VERSION` è definito come una `COMPATIBLE_INTERFACE_STRING`, il che significa che deve essere compatibile con qualsiasi dipendenza. Impostando questa proprietà utente personalizzata in questa versione e nella prossima di `MathFunctions`, `cmake` emetterà una diagnostica se si tenta di utilizzare la versione 3 insieme alla versione 4. I pacchetti possono scegliere di utilizzare tale modello se diverse versioni «major» del pacchetto sono progettate per essere incompatibili.

20.3.2 Esportare i Target dall'albero di Build

In genere, i progetti vengono creati e installati prima di essere utilizzati da un progetto esterno. Tuttavia, in alcuni casi, è preferibile esportare i target direttamente da un albero di build. I target possono quindi essere utilizzati da un progetto esterno che fa riferimento all'albero di build senza che sia coinvolta alcuna installazione. Il comando `export` viene utilizzato per generare un file di export dei target da un albero di build del progetto.

Per usare il progetto di esempio anche da una directory di build dobbiamo solo aggiungere quanto segue a `CMakeLists.txt`:

```
export(EXPORT MathFunctionsTargets
       FILE "${CMAKE_CURRENT_BINARY_DIR}/cmake/MathFunctionsTargets.
↪cmake"
       NAMESPACE MathFunctions::
)
```

Qui utilizziamo il comando `export` per generare gli export dei target per l'albero di build. In questo caso, creeremo un file chiamato `MathFunctionsTargets.cmake` nella subdirectory `cmake` dellla directory di build. Il file generato contiene il codice richiesto per importare il target e può essere caricato da un progetto esterno che conosce l'albero di build del progetto. Questo file è specifico dell'albero di build **non è riposizionabile**.

È possibile creare un file di configurazione del pacchetto e uno di versione adatti per definire un pacchetto per l'albero di build utilizzabile senza installazione. Gli utenti dell'albero di build possono semplicemente assicurarsi che `CMAKE_PREFIX_PATH` contenga la directory di build o impostare `MathFunctions_DIR` a `<build_dir>/MathFunctions` nella cache.

Un esempio di applicazione di questa funzionalità è la creazione di un eseguibile su una piattaforma host durante la cross-compilazione. Il progetto contenente l'eseguibile può essere creato sulla piattaforma host e quindi il progetto cross-compilato per un'altra piattaforma può caricarlo.

20.3.3 Creazione e Installazione di un Pacchetto

A questo punto, abbiamo generato una configurazione CMake rilocabile per il progetto utilizzabile dopo l'installazione del progetto. Proviamo a «buildare» il progetto `MathFunctions`:

```
mkdir MathFunctions_build
cd MathFunctions_build
cmake ../MathFunctions
cmake --build .
```

Nella directory di build, notare che il file `MathFunctionsTargets.cmake` è stato creato nella sottodirectory `cmake`.

Ora si installa il progetto:

```
$ cmake --install . --prefix "/home/myuser/installdir"
```

20.4 Creazione di Pacchetti Rilocabili

I pacchetti creati da `install(EXPORT)` sono progettati per essere riposizionabili, utilizzando percorsi relativi alla posizione del pacchetto stesso. Non devono fare riferimento a path assoluti sulla macchina su cui è creato il pacchetto che non esisteranno sulle macchine su cui il pacchetto potrebbe essere installato.

Quando si definisce l'interfaccia di un target per `EXPORT`, tenere presente che le directory di inclusione devono essere specificate come percorsi relativi alla `CMAKE_INSTALL_PREFIX` ma non devono includere esplicitamente la `CMAKE_INSTALL_PREFIX`:

```
target_include_directories(tgt INTERFACE
  # Wrong, not relocatable:
  $<INSTALL_INTERFACE:${CMAKE_INSTALL_PREFIX}/include/TgtName>
)

target_include_directories(tgt INTERFACE
  # Ok, relocatable:
  $<INSTALL_INTERFACE:include/TgtName>
)
```

L'espressione `$<INSTALL_PREFIX>` del `generator expression` è utilizzabile come segnaposto per il prefisso di installazione senza risultare in un pacchetto non riposizionabile. Ciò è necessario se si utilizzano «generator expression» complesse:

```
target_include_directories(tgt INTERFACE
  # Ok, relocatable:
  $<INSTALL_INTERFACE:$<INSTALL_PREFIX>/include/TgtName>
)
```

Questo vale anche per i percorsi che fanno riferimento a dipendenze esterne. Non è consigliabile popolare le proprietà che possono contenere percorsi, come INTERFACE_INCLUDE_DIRECTORIES e INTERFACE_LINK_LIBRARIES, con percorsi relativi alle dipendenze. Ad esempio, questo codice potrebbe non funzionare bene per un pacchetto rilocabile:

```
target_link_libraries(MathFunctions INTERFACE
  ${Foo_LIBRARIES} ${Bar_LIBRARIES}
  )
target_include_directories(MathFunctions INTERFACE
  "$<INSTALL_INTERFACE:${Foo_INCLUDE_DIRS};${Bar_INCLUDE_DIRS}>"
  )
```

Le variabili di riferimento possono contenere i percorsi assoluti delle librerie e includere directory **come si trovano sulla macchina su cui è stato creato il pacchetto**. Ciò creerebbe un pacchetto con percorsi alle dipendenze «scolpiti» non adatti al riposizionamento.

Idealmente, tali dipendenze dovrebbero essere utilizzate attraverso i propri target IMPORTED che hanno le proprie IMPORTED_LOCATION e proprietà dei requisiti di utilizzo come INTERFACE_INCLUDE_DIRECTORIES popolate in modo appropriato. Questi target importati possono quindi essere utilizzati col comando target_link_libraries per MathFunctions:

```
target_link_libraries(MathFunctions INTERFACE Foo::Foo Bar::Bar)
```

Con questo approccio il pacchetto fa riferimento alle sue dipendenze esterne solo attraverso i nomi di target IMPORTED. Quando un «consumer» utilizza il pacchetto installato, eseguirà i comandi find_package appropriati (tramite la macro find_dependency descritta sopra) per trovare le dipendenze e popolare i target importati con percorsi appropriati sul proprio computer.

20.5 Utilizzo dei File di Configurazione del Pacchetto

Ora siamo pronti per creare un progetto per utilizzare la libreria MathFunctions installata. In questa sezione utilizzeremo il codice sorgente da Help\guide\importing-exporting\Downstream. In questa directory c'è un file sorgente chiamato main.cc che usa la libreria MathFunctions per calcolare la radice quadrata di un dato numero e poi stampa i risultati:

```
// A simple program that outputs the square root of a number
#include <iostream>
#include <string>

#include "MathFunctions.h"

int main(int argc, char* argv[])
{
  if (argc < 2) {
    std::cout << "Usage: " << argv[0] << " number" << std::endl;
    return 1;
```

(continues on next page)

(continua dalla pagina precedente)

```
  }

  // convert input to double
  const double inputValue = std::stod(argv[1]);

  // calculate square root
  const double sqrt = MathFunctions::sqrt(inputValue);
  std::cout << "The square root of " << inputValue << " is " << sqrt
            << std::endl;

  return 0;
}
```

Come prima, inizieremo coi comandi `cmake_minimum_required` e `project` nel file `CMakeLists.txt`. Per questo progetto specificheremo anche lo standard del C++.

```
cmake_minimum_required(VERSION 3.15)
project(Downstream)

# specify the C++ standard
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)
```

Possiamo usare il comando `find_package`:

```
find_package(MathFunctions 3.4.1 EXACT)
```

Creare un eseguibile:

```
add_executable(myexe main.cc)
```

E linkarlo alla libreria `MathFunctions`:

```
target_link_libraries(myexe PRIVATE MathFunctions::MathFunctions)
```

Questo è tutto! Ora proviamo a buildare il progetto `Downstream`.

```
mkdir Downstream_build
cd Downstream_build
cmake ../Downstream
cmake --build .
```

Potrebbe essere apparso un warning durante la configurazione di CMake:

```
CMake Warning at CMakeLists.txt:4 (find_package):
  By not providing "FindMathFunctions.cmake" in CMAKE_MODULE_PATH this
  project has asked CMake to find a package configuration file␣
↪provided by
```

(continues on next page)

(continua dalla pagina precedente)

```
  "MathFunctions", but CMake did not find one.

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

    MathFunctionsConfig.cmake
    mathfunctions-config.cmake

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

Impostare `CMAKE_PREFIX_PATH` sul punto in cui MathFunctions è stato installato in precedenza e riprovare. Assicurarsi che l'eseguibile appena creato venga eseguito come previsto.

20.6 Aggiungere Componenti

Modifichiamo il progetto `MathFunctions` per utilizzare i componenti. Il codice sorgente di questa sezione si trova in `Help\guide\importing-exporting\MathFunctionsComponents`. Il file CMakeLists per questo progetto aggiunge due sottodirectory: `Addition` e `SquareRoot`.

```
cmake_minimum_required(VERSION 3.15)
project(MathFunctionsComponents)

# specify the C++ standard
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

add_subdirectory(Addition)
add_subdirectory(SquareRoot)
```

Generare e installare i file di configurazione e di versione del pacchetto:

```
include(CMakePackageConfigHelpers)

# set version
set(version 3.4.1)

# generate the version file for the config file
```

(continues on next page)

(continua dalla pagina precedente)

```
write_basic_package_version_file(
  "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfigVersion.cmake"
  VERSION "${version}"
  COMPATIBILITY AnyNewerVersion
)

# create config file
configure_package_config_file(${CMAKE_CURRENT_SOURCE_DIR}/Config.cmake.
↪in
  "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfig.cmake"
  INSTALL_DESTINATION lib/cmake/MathFunctions
  NO_CHECK_REQUIRED_COMPONENTS_MACRO
)

# install config files
install(FILES
          "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfig.cmake"
          "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfigVersion.cmake
↪"
        DESTINATION lib/cmake/MathFunctions
)
```

Se vengono specificati `COMPONENTS` quando il downstream utilizza `find_package`, vengono elencati nella variabile `<PackageName>_FIND_COMPONENTS`. Possiamo usare questa variabile per verificare che tutti i componenti target necessari siano inclusi in `Config.cmake.in`. Allo stesso tempo, questa funzione agirà come una macro custom `check_required_components` per garantire che il downstream tenti di utilizzare solo i componenti supportati.

```
@PACKAGE_INIT@

set(_supported_components Addition SquareRoot)

foreach(_comp ${MathFunctions_FIND_COMPONENTS})
  if (NOT _comp IN_LIST _supported_components)
    set(MathFunctions_FOUND False)
    set(MathFunctions_NOT_FOUND_MESSAGE "Unsupported component: ${_
↪comp}")
  endif()
  include("${CMAKE_CURRENT_LIST_DIR}/MathFunctions${_comp}Targets.cmake
↪")
endforeach()
```

Qui, `MathFunctions_NOT_FOUND_MESSAGE` è settato su una diagnosi secondo cui il pacchetto non è stato trovato perché è stato specificato un componente non valido. Questa variabile del messaggio può essere impostata per ogni caso in cui la variabile `_FOUND` è settata a `False` e verrà visualizzata all'utente.

Le directory `Addition` e `SquareRoot` sono simili. Diamo un'occhiata a uno dei file

CMakeLists:

```
# create library
add_library(SquareRoot STATIC SquareRoot.cxx)

add_library(MathFunctions::SquareRoot ALIAS SquareRoot)

# add include directories
target_include_directories(SquareRoot
                           PUBLIC
                           "$<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_
↪DIR}>"
                           "$<INSTALL_INTERFACE:include>"
)

# install the target and create export-set
install(TARGETS SquareRoot
        EXPORT SquareRootTargets
        LIBRARY DESTINATION lib
        ARCHIVE DESTINATION lib
        RUNTIME DESTINATION bin
        INCLUDES DESTINATION include
)

# install header file
install(FILES SquareRoot.h DESTINATION include)

# generate and install export file
install(EXPORT SquareRootTargets
        FILE MathFunctionsSquareRootTargets.cmake
        NAMESPACE MathFunctions::
        DESTINATION lib/cmake/MathFunctions
)
```

Ora possiamo buildare il progetto come descritto nelle sezioni precedenti. Per testare l'utilizzo di questo pacchetto, possiamo utilizzare il progetto in `Help\guide\importing-exporting\DownstreamComponents`. Ci sono due differenze rispetto al precedente progetto `Downstream`. Per prima cosa dobbiamo trovare i componenti del pacchetto. Cambiare la riga `find_package` da:

```
find_package(MathFunctions 3.4.1 EXACT)
```

A:

```
find_package(MathFunctions 3.4 COMPONENTS Addition SquareRoot)
```

e la riga `target_link_libraries` da:

```
target_link_libraries(myexe PRIVATE MathFunctions::MathFunctions)
```

A:

```
target_link_libraries(myexe PRIVATE MathFunctions::Addition␣
↪MathFunctions::SquareRoot)
```

In `main.cc`, sostituire `#include MathFunctions.h` con:

```
#include "Addition.h"
#include "SquareRoot.h"
```

Infine, utilizzare la libreria `Addition`:

```
  const double sum = MathFunctions::add(inputValue, inputValue);
  std::cout << inputValue << " + " << inputValue << " = " << sum <<␣
↪std::endl;
```

Buildare il progetto `Downstream` e verificare che possa trovare e utilizzare i componenti del pacchetto.

CAPITOLO 21

Guida all'integrazione dell'IDE

21.1 Introduzione

Gli ambienti di sviluppo integrati (IDE) potrebbero volersi integrare con CMake per migliorare l'esperienza di sviluppo per gli utenti di CMake. Questo documento illustra le «best practices» consigliate per tale integrazione.

21.2 Bundling (Raggruppamento)

Molti fornitori di IDE vorranno eseguire il bundle di una copia di CMake con il proprio IDE. Gli IDE che raggruppano CMake dovrebbero offrire all'utente la possibilità di utilizzare un'installazione CMake esterna anziché quella in bundle, nel caso in cui tale copia diventi obsoleta e l'utente desideri utilizzare una versione più recente.

Sebbene i fornitori IDE possano essere tentati di raggruppare diverse versioni di CMake con la loro applicazione, tale pratica non è consigliata. CMake offre forti garanzie di compatibilità con le versioni precedenti e non c'è motivo di non utilizzare una versione di CMake più recente di quella richiesta da un progetto, o addirittura la versione più recente. Pertanto, è consigliabile che i fornitori IDE che eseguono il blundle (raggruppamento) di CMake con la propria applicazione includano sempre la versione patch più recente di CMake disponibile al momento del rilascio.

Come suggerimento, gli IDE potrebbero anche fornire una copia del sistema di build Ninja insieme a CMake. Ninja è altamente performante e ben supportato su tutte le piattaforme che supportano CMake. Gli IDE che raggruppano Ninja dovrebbero utilizzare Ninja 1.10 o versione successiva, che contiene le funzionalità necessarie per supportare le build Fortran.

21.3 Preimpostazioni

CMake supporta un formato di file chiamato `CMakePresets.json` e la sua controparte specifica dell'utente, `CMakeUserPresets.json`. Questo file contiene informazioni sulle varie preimpostazioni della configurazione che un utente potrebbe desiderare. Ogni preimpostazione può avere un compilatore diverso, build flag, ecc. I dettagli di questo formato sono spiegati nel manuale `cmake(1)`.

I fornitori di IDE sono incoraggiati a leggere e valutare questo file allo stesso modo di CMake e a presentare all'utente le preimpostazioni elencate nel file. Gli utenti dovrebbero essere in grado di visualizzare (ed eventualmente modificare) le variabili della cache CMake, le variabili di ambiente e le opzioni della riga di comando definite per una determinata preimpostazione. L'IDE dovrebbe poi costruire l'elenco degli argomenti della riga di comando `cmake(1)` appropriati in base a queste impostazioni, anziché utilizzare direttamente l'opzione `--preset=`. L'opzione `--preset=` è intesa solo come un comodo frontend per gli utenti della riga di comando e non dovrebbe essere utilizzata dall'IDE.

Ad esempio, se un preset denominato `ninja` specifica `Ninja` come generatore e `${sourceDir}/build` come directory di build, invece di eseguire:

```
cmake -S /path/to/source --preset=ninja
```

l'IDE dovrebbe invece calcolare le impostazioni del preset `ninja`, e poi eseguire:

```
cmake -S /path/to/source -B /path/to/source/build -G Ninja
```

Sebbene leggere, analizzare e valutare il contenuto di `CMakePresets.json` sia semplice, non è banale. Oltre alla documentazione, i fornitori IDE potrebbero anche voler fare riferimento al codice sorgente CMake e ai casi di test per una migliore comprensione di come implementare il formato. `Questo file` fornisce uno schema JSON leggibile dalla macchina per il formato `CMakePresets.json` che i fornitori IDE potrebbero trovare utile per la convalida e fornire assistenza per la modifica.

21.4 Configurazione

Gli IDE che invocano `cmake(1)` per eseguire il passaggio di configurazione potrebbero voler ricevere informazioni sugli artefatti che la build produrrà, nonché sulle directory di inclusione, sulle definizioni della compilazione, ecc. utilizzate per creare gli artefatti. Tali informazioni possono essere ottenute utilizzando `File API`. La pagina del manuale per l'API File contiene ulteriori informazioni sull'API e su come richiamarla. `La modalità server` è stata rimossa a partire da CMake 3.20 e non deve essere utilizzata su CMake 3.14 o versioni successive.

Gli IDE devono evitare di creare più alberi di compilazione del necessario e crearne di più solo se l'utente desidera passare a un compilatore diverso, usare flag di compilazione diversi, ecc. In particolare, gli IDE NON dovrebbero creare più alberi di build a configurazione singola che abbiano tutti le stesse proprietà ad eccezione di una `CMAKE_BUILD_TYPE` diversa, creando di fatto un ambiente multi-config. Al contrario, per ottenere l'elenco delle configurazioni di build si deve usare il generatore `Ninja Multi-Config`, insieme al `File API`.

Gli IDE non dovrebbero utilizzare i «generatori extra» con i generatori Makefile o Ninja, che generano file di progetto IDE oltre ai file Makefile o Ninja. Si dovrebbe invece usare `File API` per ottenere l'elenco degli artefatti della compilazione.

21.5 Building

Se per generare l'albero di compilazione viene utilizzato un generatore Makefile o Ninja, non è consigliabile invocare direttamente `make` o `ninja`. Si consiglia invece che l'IDE invochi `cmake(1)` con l'argomento `--build`, che a sua volta invocherà lo strumento di compilazione appropriato.

Se viene utilizzato un generatore di progetti IDE, come `Xcode` o uno dei generatori di Visual Studio, e l'IDE comprende il formato di progetto utilizzato, l'IDE dovrebbe leggere il file di progetto e compilarlo nello stesso modo in cui lo farebbe il generatore.

Il `File API` può essere utilizzato per ottenere un elenco di configurazioni di build dall'albero di build e l'IDE dovrebbe presentare questo elenco all'utente per selezionare una configurazione di build.

21.6 Test

`ctest(1)` supporta l'output in formato JSON con informazioni sui test disponibili e sulle configurazioni dei test. Gli IDE che desiderano eseguire CTest dovrebbero ottenere queste informazioni e utilizzarle per presentare all'utente un elenco di test.

Gli IDE non dovrebbero invocare il target `test` del buildsystem generato. Dovrebbero invece invocare direttamente `ctest(1)`.