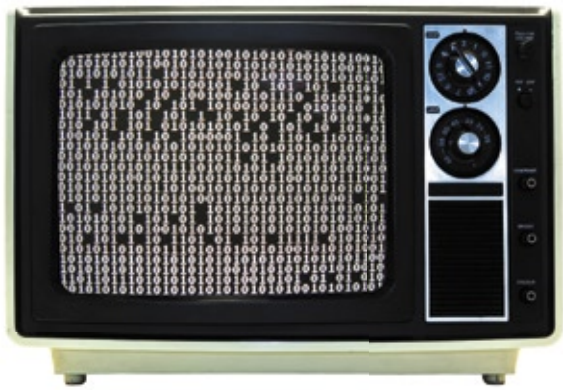
Uma introdução ao sistema de distribuição Cmake

De volta à programação

Configurar a compilação de um programa com as veneráveis Autotools às vezes é mais complexo que o próprio código do programa. Conheça os méritos da alternativa Cmake.

por Alexander Neundorf



Usuários Linux estão acostumados há anos às já tradicionais etapas de compilação e instalação de softwares: `./configure; make; make install`. As responsáveis por isso são as *Autotools*, um conjunto de scripts de *shell*, *m4* e *Perl* que auxiliam a distribuição de softwares de forma portável, que funciona em diversos sabores de Unix.

Todavia, para muitos desenvolvedores, as Autotools são uma grande barreira, em função da grande quantidade de programas disponíveis. De acordo com o que diz um dos desenvolvedores do *Cmake*, Bill Hoffman, ele se apresenta como um sistema de distribuição alternativo para mudar a forma de trabalho de todos os desenvolvedores de software. Como exemplo, cita uma série de projetos que utilizam esse sistema, tanto comerciais quanto livres. O maior e mais conhecido atualmente é o *KDE 4*, composto por milhões de linhas de código[1] e compilável, hoje, em todas as principais plataformas (Linux, *BSD, Mac OS X, Windows® com *Mingw*, assim como compiladores Microsoft). Outros projetos são o *Scribus*, o interpretador *Chicken Scheme*, o jogo de estratégia *Boson*, o *Plplot* e o *fork* do Debian para as *CDR-Tools*. Além disso, o Cmake está em avaliação nos projetos *Wireshark*, *Lya*, *Open Wengo* e *Libgphoto*.

Multi-plataforma

O *Cross-platform Make* (é isso que significa Cmake) está sob a licença BSD, e é desenvolvido pela empresa americana Kitware Inc.[2], especializada em softwares de visualização para medicina. O Cmake nasceu próximo ao *Insight Segmentation and Registration Toolkit*[3] para visualização do corpo humano. A ferramenta reúne a funcionalidade dos diversos componentes das Autotools:

- Geração de *makefiles*;
- Suporte a diversos compiladores em diferentes plataformas;
- Suporte à introspecção de sistemas;
- Suporte à extensão por macros.

A única condição para o emprego do Cmake é um compilador *C++*. Por isso ele também funciona sem problemas, por exemplo, sob o Windows com *MS Visual Studio*. Porém, o Cmake não é apenas um gerador de makefiles; ele gera também dados de entrada para o sistema de distribuição nativo da respectiva plataforma.

Por isso, ele possibilita a geração de Makefiles e projetos do *Kdevelop 3* em Unix, além de arquivos de projeto *Xcode* em Mac OS, makefiles do Windows para *Cygwin*, Mingw, *Msys*, *Borland* e *Microsoft Make*, assim como para os *IDEs* da Microsoft a partir do Visual Studio 6. Adicionalmente, o Cmake permite, em Windows e Mac OS, que os desenvolvedores continuem trabalhando com

Exemplo 1: Compilação

```
01 ~/src/olamundo $ make
02 Scanning dependencies of target olamundo
03 [100%] Building C object CMakeFiles/ola.dir/main.o
04 Linking C executable olamundo
05 [100%] Built target olamundo
06 ~/src/olamundo $ ./olamundo
07 Ola mundo!
```

os ambientes de desenvolvimento a que estão acostumados, evitando o uso da linha de comando.

Para garantir a segurança, o Cmake oferece suporte à integração de testes de unidade, *nightly builds* e outras ferramentas como *Valgrind*[4] e *KWStyle*[5]. Os resultados são colocados em um servidor central *Dart*[6].

Olá mundo

O Cmake lê arquivos com o nome `CMakeLists.txt` (atenção às letras maiúsculas e minúsculas) e a partir daí gera os makefiles desejados, referentes aos arquivos de projeto. Num exemplo de *Olá mundo*, um arquivo `main.c` contendo o código-fonte deverá ser compilado e linkado. O `CMakeLists.txt` correspondente terá a seguinte aparência:

```
add_executable(olamundo main.c)
```

O comando `add_executable()` significa que `main.c` deverá gerar um executável `olamundo`. Em plataformas Windows, o executável receberá o nome `olamundo.exe` enquanto em Unix, será `olamundo`.

Como argumento, o Cmake espera o diretório onde se encontram o código-fonte e o arquivo `CMakeLists.txt`. Com isso, basta um compilador para realizar o trabalho (**exemplo 1**).

Agora vejamos um projeto mais complexo, composto por uma biblioteca e um programa que a utiliza. Os fontes estão organizados como na **figura 1**. No diretório `myproj/lib/` ficam os arquivos `core.c`, `util.c` e – dependendo da plataforma – `unixtool.c` ou `wintool.c`, além da biblioteca compartilhada `libmyutils.so`. No diretório `myproj/app/` ficam os arquivos `main.cpp` e `process.cpp`, que gerarão o programa `fooapp`.

Na compilação de `fooapp`, o compilador encontra os cabeçalhos `core.h` e `util.h`, e linka o programa `myapp` à biblioteca `libmyutil.so`. Em seguida, o Cmake instala o programa e a biblioteca. Os **exemplos 2** a **4** mostram o conteúdo de cada `CMakeLists.txt`.

O arquivo no diretório raiz refere-se aos subdiretórios `lib/` e `app/`, que devem conter, cada um, um arquivo `CMakeLists.txt`. O Cmake ignora qualquer subdiretório sem referência.

Sintaxe

O comando `set()` (**linha 1** do **exemplo 3**) atribui um valor a uma variável. No exemplo, `libSrcs` é uma lista com os elementos `core.c` e `util.c`. O comando não diferencia maiúsculas e minúsculas, ao contrário das variáveis e argumentos.

O Cmake também usa `if` para controlar o fluxo de dados. Os complementos `else` e `endif` devem incluir a mesma palavra-chave do `if`, o que facilita a leitura do script.

Note que a **linha 4** do **exemplo 3**, ao realizar a compilação em sistemas Unix, une o arquivo `unixtool.c` à lista `libSrcs`. A expressão `${libSercs}` retorna o valor da variável, e então a **linha 4** é transformada em:

```
set(libSrcs core.c util.c
➥unixtool.c)
```

Além disso, no **exemplo 3**, `add_library()` acrescenta ao projeto uma biblioteca compartilhada. A sintaxe é semelhante à de `add_executable()`: primeiro o nome (lógico) da biblioteca, depois os arquivos com os fontes, com o auxílio da variável `libSrcs`. O argumento `SHARED` informa tratar-se de uma biblioteca compartilhada; caso contrário, o Cmake gera uma biblioteca estática.

Assim como em `add_executable`, as bibliotecas recebem as extensões corretas para a plataforma: `.so` em Unix e Linux, `.dylib` em MacOS X e `.dll` em Windows. Além disso, os prefixos e sufixos também são acrescentados automaticamente; nos arquivos do Cmake consta sempre apenas o nome lógico da biblioteca.

Exemplo 2: myproj/CMakeLists.txt

```
01 project(FooProject)
02 add_subdirectory(lib)
03 add_subdirectory(app)
```

Exemplo 3: myproj/lib/CMakeLists.txt

```
01 set(libSrcs core.c util.c)
02
03 if (UNIX)
04   set(libSrcs ${libSrcs} unixtool.c)
05 else (UNIX)
06   set(libSrcs ${libSrcs} wintool.c)
07 endif (UNIX)
08
09 add_library(util SHARED ${libSrcs})
10
11 install(TARGETS util DESTINATION lib)
```

Bibliotecas

O comando `install()` (**exemplo 3**) faz o Cmake gerar rotinas de instalação para o alvo `util` e mover seu resultado (a *Libutil*, no caso) para o diretório `lib/` relativo ao caminho de instalação padrão (no Linux, geralmente `/usr/local/`).

Depois de processar esse `CMakeLists.txt`, o Cmake volta ao CMakeLists.txt que o chamou (**exemplo 2**) e prossegue com o processamento, entrando em `myproj/app/CMakeLists.txt` (**exemplo 4**).

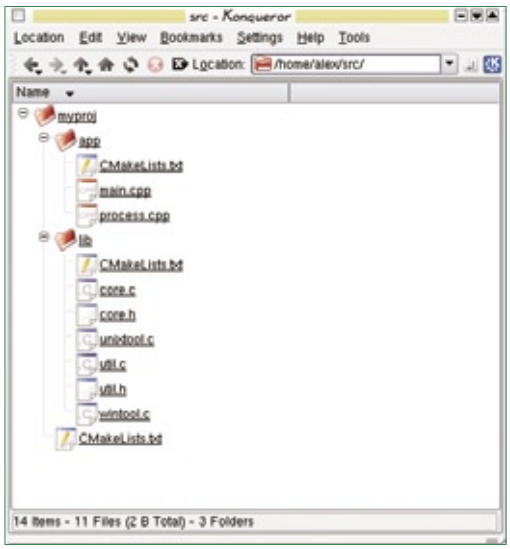


Figura 1 A árvore de diretórios do projeto de exemplo com três arquivos de projeto `CMakeLists.txt`.

Exemplo 4: myproj/app/CMakeLists.txt

```
01 include_directories(${CMAKE_SOURCE_
➥DIR}/lib)
02
03 set(fooappSrcs main.cpp process.
cpp)
04 add_executable(fooapp ${fooappSrcs})
05
06 target_link_libraries(fooapp util)
07
08 install(TARGETS fooapp DESTINATION
➥bin)
```

Exemplo 5: Compilação Out-of-source (1)

```
01 ~/src $ mkdir myproj-build
02 ~/src/ $ cd myproj-build
03 ~/src/myproj-build $ cmake ~/src/myproj
04 -- Check for working C compiler: /usr/bin/gcc
05 -- Check for working C compiler: /usr/bin/gcc -- works
06 -- Check size of void*
07 -- Check size of void* - done
08 -- Check for working CXX compiler: /usr/bin/c++
09 -- Check for working CXX compiler: /usr/bin/c++ -- works
10 -- Configuring done
11 -- Generating done
12 -- Build files have been written to: ~/src/myproj-build
13 ~/src/myproj-build $ make
14 [ 20%] Building C object lib/CMakeFiles/util.dir/core.o
15 [ 40%] Building C object lib/CMakeFiles/util.dir/util.o
16 [ 60%] Building C object lib/CMakeFiles/util.dir/unixtool.o
17 Linking C shared library libutil.so
18 [ 60%] Built target util
19 Scanning dependencies of target fooapp
20 [ 80%] Building CXX object app/CMakeFiles/fooapp.dir/main.o
21 [100%] Building CXX object app/CMakeFiles/fooapp.dir/process.o
22 Linking CXX executable fooapp
23 [100%] Built target fooapp
```

Agora, `add_executable()` gera o aplicativo `fooapp`. No Cmake, o desenvolvedor pode escolher livremente os nomes das variáveis, diferente das Autotools. Esse é um princípio fundamental do Cmake: nada é feito sem que o usuário esteja ciente e não existem arquivos ou nomes de variáveis mágicos.

O comando `target_link_libraries` determina quais bibliotecas são utilizadas pelo alvo `fooapp` (`util`, no caso). O Cmake sabe que se trata dessa biblioteca, e automaticamente se encarrega de:

- Compilar `util` antes de `fooapp`;
- Linkar `fooapp` a `util`;
- Definir `RPATH` para que `fooapp` utilize a biblioteca `util`;
- Na instalação, adaptar `RPATH` ao que o usuário determinar.

In e Out-of-source

O procedimento tradicional de compilação mostrado até agora se chama, no Cmake, de *In-source build* ou "compilação de dentro do código". Entretanto, além dessa forma, o Cmake também suporta a compilação "de fora" do código-fonte, ou *Out-of-source build*, na qual o software não é compilado na árvore do código-fonte, mas sim em qualquer outro diretório da preferência do usuário.

O processo Out-of-source tem mais vantagens: a árvore do código-fonte fica mais limpa e clara e não é ocupada por arquivos-objeto ou outros intermediários. Caso se deseje gerar o projeto inteiro novamente, pode-se simplesmente apagar o diretório de compilação. E no diretório com os códigos-fonte, pode-se adicionar vários diretórios de compilação. Isso pode ser útil, por exemplo, para definir um destino para a compilação final do programa e outro para a versão de depuração.

Para compilar o exemplo acima como Out-of-source (**exemplo 5**), cria-se um diretório de compilação, e lá se inicia o Cmake, com o diretório dos fontes como argumento.

Após a compilação, o comando `make install` instala o software. O `make help` oferece um panorama das opções. No *KDevelop*, por exemplo, o diretório de compilação `myproj-build-2` abriga os arquivos de projeto, e também é controlado pelo Cmake:

```
$ cd myproj-build-2
$ cmake ~/src/myproj -GKDevelop3
-- Check for working C compiler: /
➥usr/bin/gcc
...
-- Build files have been written
➥to: ~/src/myproj-build-2
```

O diretório de compilação contém, entre outros, o arquivo de projeto do KDevelop `FooProject.kdevelop` (**figura 2**). O nome do projeto é dado pelo comando `project()` (**linha 1** do **exemplo 2**).

Variáveis

A variável `CMAKE_SOURCE_DIR` contém o diretório raiz da árvore do código-fonte. Além dessa, o Cmake usa as seguintes variáveis para navegar no código-fonte ou no diretório de compilação:

- `CMAKE_CURRENT_SOURCE_DIR`, que se encontra no diretório atual na árvore do código-fonte;
- `CMAKE_BINARY_DIR`, que contém a raiz do diretório de compilação (análogo a `CMAKE_SOURCE_DIR`);
- `CMAKE_CURRENT_BINARY_DIR`, no subdiretório atual do diretório de compilação (análogo a `CMAKE_CURRENT_SOURCE_DIR`).

É importante sempre saber se estamos trabalhando no diretório do código-fonte ou no diretório de compilação. Em projetos Out-of-source, é necessário atentar para o fato de que caminhos relativos sempre terão como base o atual (`CMAKE_CURRENT_SOURCE_DIR`). Assim, o comando `add_executable(ola main.c)`, por exem-

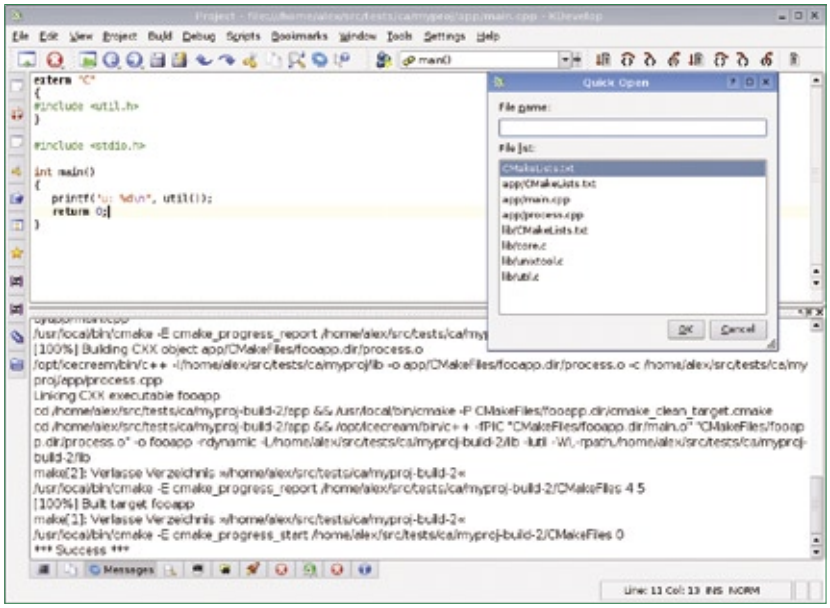


Figura 2 O ambiente de desenvolvimento do *KDevelop* com o projeto gerado pelo *Cmake*.

Exemplo 6: Compilação Out-of-source (2)

```
01 ~/src $ mkdir cmakevars-build
02 ~/src $ cd cmakevars-build
03 ~/src/cmakevars-build $ cmake ../cmakevars
04 -- ~/src/cmakevars ~/src/cmakevars ~/src/cmakevars-build ~/src/
➥cmakevars-build
05 -- ~/src/cmakevars ~/src/cmakevars/misc ~/src/cmakevars-build
➥~/src/cmakevars-build/misc
06 -- ~/src/cmakevars ~/src/cmakevars/misc/sub ~/src/cmakevars-
➥build ~/src/cmakevars-build/misc/sub
07 -- ~/src/cmakevars ~/src/cmakevars/app ~/src/cmakevars-build ~/
➥src/cmakevars-build/app
08 -- Configuring done
09 -- Generating done
10 -- Build files have been written to: ~/src/tests/ca/cmakevars-
➥build
```

Exemplo 7: Compilação In-source

```
01 ~/src/cmakevars $ cmake .
02 -- ~/src/cmakevars ~/src/cmakevars ~/src/cmakevars ~/src/
➥cmakevars
03 -- ~/src/cmakevars ~/src/cmakevars/misc ~/src/cmakevars ~/src/
➥cmakevars/misc
04 -- ~/src/cmakevars ~/src/cmakevars/misc/sub ~/src/cmakevars ~/
➥src/cmakevars/misc/sub
05 -- ~/src/cmakevars ~/src/cmakevars/app ~/src/cmakevars ~/src/
➥cmakevars/app
06 -- Configuring done
07 -- Generating done
08 -- Build files have been written to: ~/src/cmakevars
```

plo, corresponde ao arquivo `${CMAKE_CURRENT_SOURCE_DIR}/main.c`.

Todos os arquivos de dados a serem criados deverão ser gerados no `CMAKE_BINARY_DIR` ou no `CMAKE_CURRENT_BINARY_DIR`. Isso precisa ser feito nos comandos `configure_file()`, `file()` e `add_custom_command()`. Neles, será usado sempre o caminho completo, como `${CMAKE_CURRENT_BINARY_DIR}/generated.h`.

Um pequeno exemplo com apenas quatro arquivos `CMakeLists.txt` (**figura 3**) demonstra a função dessas variáveis. Cada um dos `CMakeLists.txt` exporta as quatro variáveis mencionadas. O conteúdo de `src/cmakevars/CMakeLists.txt`, por exemplo, é:

```
message(STATUS "${CMAKE_SOURCE_
➥DIR} ${CMAKE_CURRENT_SOURCE_DIR}
➥${CMAKE_BINARY_ DIR} ${CMAKE_
➥CURRENT_BINARY_DIR}") add_
➥subdirectory(misc) add_
➥subdirectory(app)
```

Os **exemplos 6** e **7** comparam a saída do Cmake nos usos In-source e Out-of-source. Em Out-of-source, os valores de `BINARY_DIR` são diferentes. O princípio geral é que as variáveis `*_SOURCE_DIR` apontam sempre para a árvore do código-fonte e, por outro, lado `*_BINARY_DIR` aponta sempre para a estrutura do diretório de compilação. As variáveis `CMAKE_CURRENT_*` sempre se referem ao subdiretório atual.

Bibliotecas externas

Os exemplos dados até aqui produziram bibliotecas e programas, mas não utilizaram bibliotecas externas que são sempre necessárias em projetos não triviais. Por exemplo, um software gráfico que trabalhe com fotos *jpeg* utiliza normalmente a biblioteca *Libjpeg* e os cabeçalhos jpeg correspondentes, que costumam estar em diretórios distintos em cada sistema.

Em sistemas diferentes, os cabeçalhos e bibliotecas encontram-se em locais distintos. O Cmake resolve esse problema com o comando `find_package`:

```
find_package(JPEG REQUIRED)
include_directories(${JPEG_
➥INCLUDE_DIR})
add_executable(jpegviewer main.cpp
➥viewer.cpp)
target_link_libraries(jpegviewer
➥${JPEG_LIBRARIES})
```

Exemplo 8: Geração de arquivos

```
01 add_custom_command(OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/generated.h
02 COMMAND cp ${CMAKE_CURRENT_SOURCE_DIR}/generated.h.source ${CMAKE_
➥CURRENT_BINARY_DIR}/generated.h
03 DEPENDS ${CMAKE_CURRENT_SOURCE_DIR}/generated.h.source )
04
05 add_executable(ola main.c ${CMAKE_CURRENT_BINARY_DIR}/generated.h)
```

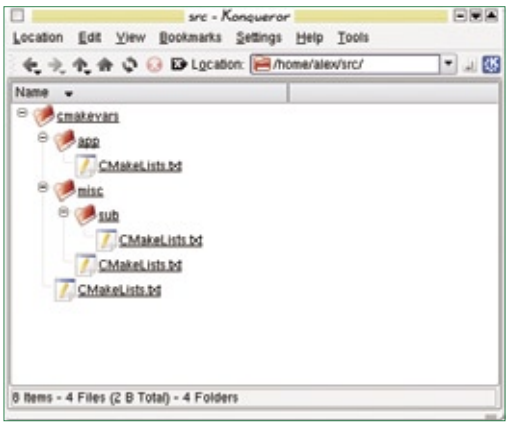


Figura 3 A estrutura de dados de teste das variáveis do *Cmake* possui apenas três arquivos de projeto.

Esse comando faz com que o Cmake localize em seu diretório de módulos de sistema um arquivo chamado `FindJPEG.cmake` e o execute. Para cada pacote que pertença ao projeto, é necessário ter um arquivo de dados como `Find<pacote>.cmake`.

Todos esses arquivos de dados obedecem ao mesmo esquema: tentam localizar os arquivos de biblioteca necessários e guardam os resultados em variáveis. Na maioria das vezes é usada uma variável para os diretórios de inclusão e outra para as bibliotecas.

Exemplo 9: Macros definidas

```
01 macro(print_all_args _greeting)
02   message(STATUS "${_greeting}")
03   foreach(_currentItem ${ARGN})
04     message(STATUS "current item:
➥${_currentItem}")
05   endforeach(_currentItem)
06 endmacro(print_all_args)
07
08 set(fooSrcs main.cpp widget.cpp
➥process.cpp)
09
10 print_all_args("Hallo Welt"
➥${fooSrcs})
```

Os nomes seguem um esquema consistente, como por exemplo `JPEG_INCLUDE_DIR` (ou `FOO_INCLUDES`) e `JPEG_LIBRARIES` (ou `FOO_LIB` e `FOO_LIBS`).

Caso seja encontrada a biblioteca necessária, o `FindJPEG.cmake` atribui valor verdadeiro à variável `JPEG_FOUND`. Com ela, `CMakeLists.txt` pode saber se a libjpeg foi encontrada e, caso necessário, compilar outros códigos-fonte do projeto atual. Se a biblioteca for exigida (palavra-chave `REQUIRED`), o processamento é interrompido com um aviso de erro, caso ela esteja ausente.

O Cmake ainda é capaz de realizar outros testes muito interessantes, como verificar a funcionalidade do cabeçalho `stdio.h`, por exemplo, através da compilação de um pequeno programa de testes. Novamente, o resultado desse tipo de análise é atribuído a uma variável, no estilo `HAVE_STDIO_H`.

Cabeçalhos

O comando `configure_file()` é usado em `CMakeLists.txt` para ler o arquivo `config.h.in`, executar substituições e escrever o conteúdo modificado no arquivo `${CMAKE_CURRENT_BINARY_DIR}/config.h`. A instrução `configure_file()` substitui todas as variáveis do Cmake, como números de versão, por exemplo, por seus valores corretos. Além disso, o script `#cmakedefine` é substituído por `#define` quando a respectiva variável é `TRUE`, e por um `#undef` descomentado se a variável for falsa.

Assim são gerados os cabeçalhos, que descrevem precisamente as respectivas propriedades do sistema, permitindo a criação de programas portáveis. A instrução `check_include_files()` naturalmente não é a única macro disponibilizada pelo Cmake. Existem macros para vários objetivos: examinar o tamanho de variáveis, as opções do compilador e diversos outros fins.

Novos arquivos

Um dos trabalhos consideráveis de um sistema de distribuição é, primeiramente, gerar novos arquivos no início do processo de compilação. Isso é necessário, por exemplo, ao se usar o compilador *Corba IDL*, o compilador de metaobjetos `moc` para *Qt*, ou também com *Flex* e *Bison*. Para demonstrar isso, basta um exemplo trivial (**exemplo 8**), que gera um arquivo (não portável) com o comando `cp`. O comando `add_custom_command()` na **linha 1** é uma instrução que pode gerar o arquivo `generated.h` especificado por `OUTPUT`.

É lógico que os arquivos nunca devem ser criados na árvore de código-fonte, mas sim no diretório de compilação; a variável `CMAKE_CURRENT_BINARY_DIR` cuida disso.

O Cmake não é uma linguagem de programação completa, contudo s.uporta macros para aumentar a portabilidade e a legibilidade. Os **exemplos 9** e **10** mostram isso. As **linhas 1** a **6** (**exemplo 9**) definem uma macro chamada `print_all_args`, que

Exemplo 10: Macros aplicadas

```
01 ~/src/macrotest$ cmake .
02 -- Ola mundo
03 -- current item: main.cpp
04 -- current item: widget.cpp
05 -- current item: process.cpp
06 -- Configuring done
07 -- Generating done
08 -- Build files have been written to: ~/src/macrotest
```

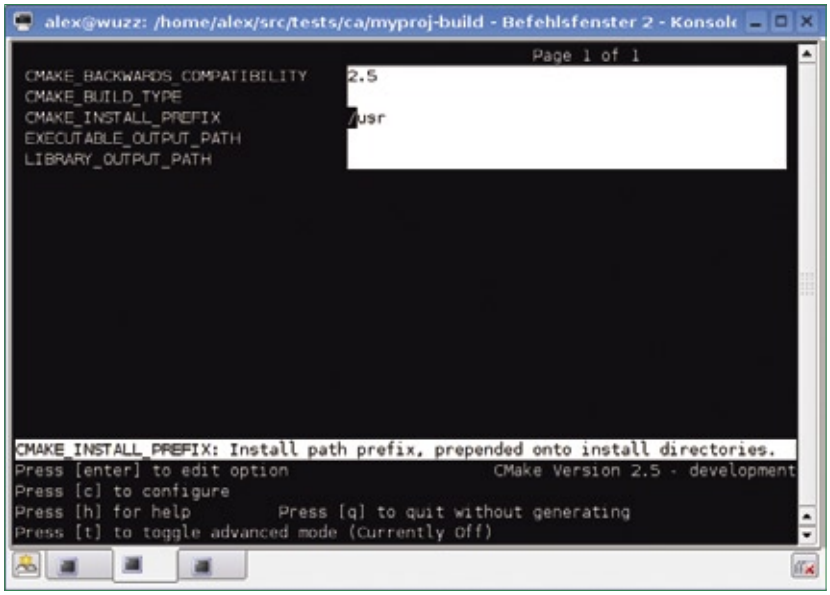


Figura 4 A interface *curses ccmake* editando variáveis de cache. Com ela, é simples alterar a configuração do projeto com *Cmake*.

espera ao menos um argumento `greeting`. Se for chamada com mais argumentos, a variável especial `ARGN` auxilia seu acesso (**linhas 3** a **5**). A lista definida na **linha 8** é passada como argumento na **linha 10**. As macros costumam esconder do desenvolvedor as complexidades, para simplificar o processo e eliminar a preocupação com detalhes.

Conclusão

Este artigo apresentou os aspectos mais fundamentais do Cmake, suficientes para usá-lo em projetos próprios. Não foi abordado, por exemplo, o uso de `RPATH`, o suporte a expressões regulares, a manipulação do cache (**figura 4**) e a execução de programas. Essas funções juntas possibilitam escrever scripts de compilação muito produtivos. No projeto KDE, há muitos exemplos de utilização do Cmake.

No Cmake, os desenvolvedores C e C++ encontram uma ferramenta que realmente permite um controle maior sobre a compilação, ao mesmo tempo garantindo sua simplicidade. Ele é ativamente desenvolvido, e as comunidades de desenvolvedores e usuários cresce rapidamente.

Quem estiver começando agora uma carreira de programador deve considerar seriamente o aprendizado do Cmake, talvez até em detrimento das Autotools. Com isso, será possível compilar seus softwares não apenas no Linux, mas também no Mac OS X e MS Windows. ■

Mais informações

[1] Repositório SVN do KDE: http://websvn.kde.org/trunk/KDE/kdelibs/cmake/modules/

[2] Kitware: http://www.kitware.com

[3] Insight Segmentation and Registration Toolkit: http://www.itk.org

[4] Valgrind: http://www.valgrind.org

[5] KWStyle: http://public.kitware.com/KWStyle

[6] Dashboard do Cmake: http://www.cmake.org/Testing/Dashboard/MostRecentResults-Nightly/Dashboard.html