**Usando o GCC e o Make para compilar, lincar e criar aplicações C/C++**

O compilador GNU C original foi desenvolvido por Richard Stallman, o fundador do Projeto GNU. Richard Stallman fundou o projeto GNU em 1984 para criar um sistema operacional baseado no Unix completo como software livre, para promover liberdade e cooperação entre usuários de computadores e programadores.

O GCC, antigamente conhecido como “*GNU C Compiler*“, cresceu várias vezes para suportar muitas outras linguagens, como C++, Objective-C, Java, Fortran e Ada. É referenciado atualmente como “*GNU Compiler Collection*“. O site base do GCC é o <http://gcc.gnu.org/>.

O GCC é um componente chave do “*GNU Toolchain*“, para desenvolvimento de aplicações, e sistemas operacionais. O “GNU Toolchain” inclui também:

1. GNU Compiler Collection (GCC): um compilador que suporta diversas linguagens como C/C++, Objective-C e Java.
2. GNU Make: uma ferramenta de automação para compilar e criar aplicações.
3. GNU Binutils: um conjunto de ferramentas binárias, incluindo um lincador e um assembler.
4. GNU Debugger (GDB).
5. GNU Autotools: Um sistema de criação que inclui Autoconf, Autoheader, Automake e Libtool.
6. GNU Bison: um gerador de análises (similar ao *lex* e ao *yacc*).

O GCC é *portável* e roda em muitos plataformas distintas. O GCC (e o GNU Toolchain) está disponível atualmente em todos os Unixes. Eles foram portados também para Windows através dos projetos MinGW e Cygwin. O GCC também é um *cross-compiler*, para produzir executáveis para plataformas diferentes.

As várias versões do GCC são:

* Em 1987, a primeira versão do GCC foi lançada.
* Em 1992, O GCC versão 2 foi lançado com suporte para C++.
* Em 2001, O GCC versão 3 foi lançado incorporando ECGS (Experimental GNU Compiler System), com melhorias na otimização.
* Em 2005, O GCC versão 4 foi lançado. Esta é a versão atual do GCC.

**1.2  Instalando o GCC**

O GCC (GNU Toolchain) é incluído em todos os tipos de Unix. Para o Windows, você pode instalar tanto o MinGW GCC ou Cygwin GCC. Leia o artigo “[How to install Cygwin and MinGW](http://www3.ntu.edu.sg/home/ehchua/programming/howto/Cygwin_HowTo.html)” para saber como.

**MinGW GCC**

O MinGW (Minimalist GNU for Windows) é um porte do *GNU Compiler Collection* (GCC) e do GNU Binutils para uso no Windows. Ele também inclui o MSYS (Minimal System), que é basicamente um aplicativo de terminal.

**Cygwin GCC**

O Cygwin é um ambiente do tipo do Unix e uma interface baseada em linha de comando para o Microsoft Windows. O Cygwin é enorme e inclui a maioria das ferramentas e utilitários do Unix. Ele também inclui os aplicativos de terminal mais usados.

Duas versões do GCC vem instaladas, identificadas como gcc-3.exe e gcc-4.exe (além de g++-3.exe e g++-4.exe). Também é fornecido links simbólicos gcc.exe e g++.exe, que são associados a gcc-4.exe e g++-4.exe, respectivamente. Os links simbólicos funcionam com o aplicativo de terminal Bash, mas infelizmente não funcionam com o CMD, de forma que você precisa usar gcc-4 (ou g++-4).

**Versões**

Você pode exibir a versão do GCC pela opção  --version:

// Cygwin in bash shell

$ **gcc --version**

gcc (GCC) 4.5.3

$ **gcc-3 --version**

gcc-3 (GCC) 3.4.4 (cygming special, gdc 0.12, using dmd 0.125)

// MinGW in CMD shell

> **gcc --version**

gcc (GCC) 4.6.2

> **g++ --version**

gcc (GCC) 4.6.2

Mais detalhes podem ser obtidos através do uso da opção -v, por exemplo,

> **gcc -v**

Using built-in specs.

COLLECT\_GCC=gcc

COLLECT\_LTO\_WRAPPER=d:/mingw/bin/../libexec/gcc/mingw32/4.6.2/lto-wrapper.exe

Target: mingw32

Configured with: ../gcc-4.6.2/configure --enable-languages=c,c++,ada,fortran,objc,obj-c++

--disable-sjlj-exceptions --with-dwarf2 --enable-shared --enable-libgomp

--disable-win32-registry --enable-libstdcxx-debug --enable-version-specific-runtime-libs

--build=mingw32 --prefix=/mingw

Thread model: win32

gcc version 4.6.2 (GCC)

**Ajuda**

Você pode obter um manual de ajuda através da opção --help. Por exemplo:

> **gcc --help**

**Páginas de manual**

Você pode ler as páginas do manual do GCC (ou *man pages)*através do utilitário man:

> **man gcc**

// or

> **man g++**

// Press space key for next page, or 'q' to quit.

A leitura das páginas do manual com o CMD ou Bash pode ser dificil. Você pode gerar um arquivo de texto com o comando:

> **man gcc | col -b > gcc.txt**

O utilitário col é necessário para remover o caractere *backspace*. (No Cygwin, está disponível em “Utils”, pacote “util-linux”).

De forma alternativa, você pode ler as páginas de manual on-line, por exemplo, no site <http://linux.die.net/man/1/gcc>.

No MinGW, as páginas de manual do GCC são mantidas em “share\man\man1\gcc.1“. No Cygwin, elas são mantidas em “usr\share\man\man1“.

**1.3  Começando a usar o GCC**

O compilador GNU C e C++ são, respectivamente, gcc e g++.

**Compilar/Lincar um programa C simples – hello.c**

Abaixo segue um programa Hello-World em C chamado hello.c:

// hello.c

#include <stdio.h>

int main() {

printf("Hello, world!\n");

return 0;

}

|  |
| --- |
|  |

Para compilar o código de hello.c:

> **gcc hello.c**

// Compile and link source file hello.c into executable a.exe

Por padrão, o executável gerado será chamado “a.exe“. Para o Cygwin rodando no CMD, use gcc-4 ou gcc-3, ao invés de .

Para executar o programa:

// Under CMD Shell

> **a**

// Under Bash or Bourne Shell - include the current path (./)

$ **./a**

NOTAS (para o aplicativos de terminal Bash, Bourne e Unixes):

* No terminal Bash ou Bourne, o PATH padrão não inclui o diretório de trabalho atual. Assim, você pode precisar incluir o diretório atual (./) no comando (O Windows inclui o diretório atual no PATH automaticamente, enquanto os Unixes não – você precisa incluir o diretório atual explicitamente).
* Você pode precisar incluir a extensão do arquivo, isto é, “./a.exe“.
* Em alguns Unixes, o arquivo de saída pode ser “a.out” ou simplesmente “a“. Além disso, você pode precisar associar o *executable file-mode* (*x*) ao arquivo “a.out“, através do comando “chmod a+x *filename*” (adiciona o modo executável “+x” para todos os usuário “a+x“).

Para especificar o nome do arquivo de saída, use a opção  -o:

> **gcc -o hello.exe hello.c**

// Compile and link source file hello.c into executable hello.exe

> **hello**

// Execute hello.exe under CMD shell

$ **./hello**

// Execute hello.exe under Bash or Bourne shell, specifying the current path (./)

NOTA para Unixes: Nos diversos Unixes, você pode imitir a extensão de arquivo .exe, e simplesmente nomear o arquivo como hello. Você precisa associar o modo executável ao arquivo através do comando “chmod a+x hello“.

**Mais opções do compilador GCC**

Algumas opções comuns do compilador GCC são:

$ g++ **-Wall -g -o Hello.exe** Hello.cpp

* -o: especifica o nome do arquivo de saída.
* -Wall: imprime todas as mensagens de aviso.
* -g: gera informações de depuração adicionais para uso com o gdb.

**Compilando e lincando separadamente**

O comando acima compila o código fonte em um arquivo objeto e linca com outros arquivos objetos (bibliotecas do sistema) em um executável em um único passo. Você pode separar a compilação e a lincagem dois passos distintos com no exemplo a seguir:

// Compile-only with -c option

> g++ **-c** -Wall -g Hello.cpp

// Link object file(s) into an executable

> g++ -g -o Hello.exe Hello.o

As opções são:

* -c: compila o código fonte em um arquivo objeto “Hello.o“. Por padrão, o objeto tem o mesmo nome do código fonte com a extensão “.o” (não há necessidade de especificar a opção -o). Nenhuma lincagem com outros objetos ou bibliotecas é feita.
* A lincagem é executada quando o arquivo de entrada é um arquivo objeto “.o” (ao invés de código fonte “.cpp” ou “.c“). O GCC usa um programa de lincagem separado (chamado ld.exe) para executar a lincagem.

**Compilando e lincando múltiplos arquivos de código fonte**

Suponha que o seu programa tenha dois arquivos de código fonte: file1.cpp, file2.cpp. Você pode compilar todos os dois com um único comando:

> g++ -o myprog.exe file1.cpp file2.cpp

Porém, normalmente nós compilamos cada código fonte separadamente em um arquivo objeto, e lincamos eles em uma fase posterior. Nesse caso, as mudanças em um arquivo não fazem que seja preciso re-compilar os outros arquivos.

> g++ -c file1.cpp

> g++ -c file2.cpp

> g++ -o myprog.exe file1.o file2.o

**Compilando uma biblioteca compartilhada**

Para compilar e lincar um programa C/C++ como uma biblioteca compartilhada (".dll" no Windows, ".so" nos Unixes), use a opção -shared. Leia o artigo “[Java Native Interface](http://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html)” para um exemplo disso.

**1.4  Processo de compilação do GCC**

[](http://www.klebermota.eti.br/wp-content/GCC_CompilationProcess.png)

O GCC compila um programa C/C++ rm 4 passos como mostrado no diagrama acima. Por exemplo, o comando “gcc -o hello.exe hello.c” é processado da forma a seguir:

1. Pré-processamento: através do Pré-processador do GNU C (cpp.exe), que adiciona os cabeçalhos (#include) e expande as macros (#define).

> **cpp hello.c > hello.i**

O arquivo intermediário resultante  “hello.i” contém o código fonte expandido.

1. Compilação: O compilador compila o código fonte pré-processado em um código montado (*Assembled*) para um processador específico.

> **gcc -S hello.i**

A opção -S indica que deve-se produzir código montado (*Assembled*), ao invés de código objeto. O arquivo resultante é o “hello.s“.

1. Montagem (*Assembly*): O *assembler* (as.exe) converte o código montado (*assembly*) em código de máquina em um arquivo objeto “hello.o“.

> **as -o hello.o hello.s**

1. Lincagem: Finalmente, o lincador – *linker* – (ld.exe) linca o código objeto com a biblioteca apra produzir um arquivo executável “hello.exe“.

> **ld -o hello.exe hello.o ...libraries...**

**Modo detalhado (-v)**

Você pode acompanhar o processo de compilação em detalhes pelo uso da opção -v (verbose). Por exemplo,

> **gcc -v hello.c -o hello.exe**

**Definindo macros (-D)**

Você pode usar a opção -D*name* para definir uma macro, ou -D*name*=*value* para definir uma macro com um valor. O parâmetro *value* deve estar dentro de aspas duplas se possuir espaços em branco.

**1.5  Cabeçalhos (.h), Bibliotecas estáticas (.lib, .a) e Bibliotecas compartilhadas (.dll, .so)**

**Bibliotecas estáticas versus bibliotecas compartilhadas**

Uma biblioteca é uma coleção de arquivos objeto pré-compilados que são lincado ao nosso programa pelo linacador. Exemplos disso são as bibliotecas de funções do sistema como printf() e sqrt().

Existem dois tipos de bibliotecas externas: *bibliotecas estáticas* e *bilbiotecas compartilhadas*.

1. Uma biblioteca estática tem a extensão “.a” (archive) nos diversos Unixes ou “.lib” (library) no Windows. Quando seu programa é lincado com uma biblioteca estática, o código de máquina das funções externas usadas em seu programa é copiado no executável. Uma biblioteca estática pode ser criada pelo programa *archive* (“ar.exe“).
2. Uma biblioteca compartilhadas tem a extensão “.so” (shared objects) nos diversos Unixes ou “.dll” (dynamic link library) no Windows. Quando o seu programa é lincado com uma biblioteca compartilhadas, apenas uma pequena tabela é criada no executável. Antes do executável ser executado, o sistema operacional carrega o código de máquina necessário para as funções externas – um processo conhecido com ligação dinâmica (*dynamic linking*). A ligação dinâmica torna os arquivos menores e economiza espaço em disco, porque uma cópia de uma biblioteca pode ser compartilhada entre vários programas. Além disso, muitos sistemas operacionais permitem que apenas uma cópia de uma biblioteca compartilhada seja usada por vários programa em execução, economizando memória. O código da biblioteca compartilhada pode ser atualizado sem necessidade de recompilar o programa.

Por causa das vantagens da ligação dinâmica, o GCC, por padrão, linca o programa com a biblioteca compartilhada se ela estiver disponível.

Você pode listar o conteúdo de uma biblioteca pelo comando “nm *filename*“.

**Procurando por arquivos de cabeçalho e bibliotecas (-I, -L e -l)**

Quando está compilando o programa, o compilador precisa de arquivos de cabeçalho para compilar o código fonte; o lincador precisa de bibliotecas para resolver referências externas de outros arquivos objetos ou bibliotecas. O compilador e o lincador não encontrão os arquivos de cabeçalhos e bibliotecas a menos que você configure as opções apropriadas, que não óbvias ao usuário iniciante.

Para cada arquivo de cabeçalho usado no código fonte (pelas diretivas #include), o compilador pesquisa o chamados *include-paths* por esses arquivos. Os *include-paths* são especificados pela opção -I*dir* (ou pela variável de ambiente CPATH). Como o nome do arquivo de cabeçalho é conhecido (ex.: iostream.h, stdio.h), o compilador precisa apenas dos diretórios.

O lincador pesquisa os chamados *library-paths* pelas bibliotecas necessárias para lincagem do programa em um executável. O *library-path* é especificado pela opção -L*dir*  ('L' maiúsculo seguido pelo caminho do diretório) (ou pela variável de ambiente LIBRARY\_PATH). Além disso, você precisa especificar o nome da biblioteca. Nos Unixes, a biblioteca lib*xxx*.a é especificada pela opção -l*xxx* (letra 'l' minúscula, sem o prefixo  “lib” e a extensão ".a“). No Windows, fornece o nome completo, como -lxxx.lib. O lincador precisa tanto do diretório quanto do nome da biblioteca.

**Caminhos padrão para Includes, Library e bibliotecas**

Veja a lista de include-paths padrão do seu sistema através do comando “cpp -v” do “GNU C Preprocessor”:

> **cpp -v**

......

#include "..." search starts here:

#include <...> search starts here:

d:\mingw\bin\../lib/gcc/mingw32/4.6.2/include // d:\mingw\lib\gcc\mingw32\4.6.2\include

d:\mingw\bin\../lib/gcc/mingw32/4.6.2/../../../../include // d:\mingw\include

d:\mingw\bin\../lib/gcc/mingw32/4.6.2/include-fixed // d:\mingw\lib\gcc\mingw32\4.6.2\include-fixed

Ative o modo *verbose (-v)*durante a compilação para estudar os *library-paths* (-L) e bibliotecas (-l) usadas em seu sistema:

> **gcc -v -o hello.exe hello.c**

......

-Ld:/mingw/bin/../lib/gcc/mingw32/4.6.2 // d:\mingw\lib\gcc\mingw32\4.6.2

-Ld:/mingw/bin/../lib/gcc // d:\mingw\lib\gcc

-Ld:/mingw/bin/../lib/gcc/mingw32/4.6.2/../../../../mingw32/lib // d:\mingw\mingw32\lib

-Ld:/mingw/bin/../lib/gcc/mingw32/4.6.2/../../.. // d:\mingw\lib

-lmingw32 // libmingw32.a

-lgcc\_eh // libgcc\_eh.a

-lgcc // libgcc.a

-lmoldname

-lmingwex

-lmsvcrt

-ladvapi32

-lshell32

-luser32

-lkernel32

**Eclipse CDT:** No Eclipse CDT, você pode configurar os caminhos de arquivos de cabeçalhos e de bibliotecas além da próprias bibliotecas independentemente clicando com o botão direito no nome do projeto e selecionando ⇒ Properties ⇒ C/C++ General ⇒ Paths and Symbols ⇒ nas abas “Includes”, “Library Paths” e “Libraries”. As configurações serão aplicavéis ao projeto selecionado apenas.

**1.6  Variáveis de ambiente do GCC**

O GCC usa as seguintes variáveis de ambiente:

* PATH: Onde procura por executáveis e bibliotecas compartilhadas (.dll, .so) durante a execução do programa.
* CPATH: Onde procura os *include-paths* para os arquivos de cabeçalho. É pesquisado depois dos caminhos especificados com a opção -I<*dir*>. C\_INCLUDE\_PATH e CPLUS\_INCLUDE\_PATH podem ser usados para especificar os cabeçalhos C e C++ se a linguagem foi indicada no pré-processamento.
* LIBRARY\_PATH: Onde procura os *library-paths* para a lincagem de bibliotecas. É pesquisado depois dos caminhos especificados com a opção  –L<*dir*>.

**1.7  Utilitários para eximar os arquivos compilados**

Para todos os utilitários do GNU, você pode usar “command --help” para listar o menu de ajuda; ou “man command” para exibir o manual.

**Utilitário “file” – Determina o tipo do arquivo**

O utilitário “file” pode ser usado para exibir o tipo dos arquivos objetos e executáveis. Por exemplo,

> **gcc -c hello.c**

> **gcc -o hello.exe hello.o**

> **file hello.o**

hello.o: 80386 COFF executable not stripped - version 30821

> **file hello.exe**

hello.exe: PE32 executable (console) Intel 80386, for MS Windows

**Utilitário “nm” – Lista a tabela de símbolos de arquivos objetos**

O utilitário “nm” lista a tabela de símbolos de arquivos objetos. Por exemplo,

> **nm hello.o**

00000000 b .bss

00000000 d .data

00000000 r .eh\_frame

00000000 r .rdata

00000000 t .text

U \_\_\_main

00000000 T \_main

U \_printf

U \_puts

> **nm hello.exe | grep printf**

00406120 I \_\_imp\_\_printf

0040612c I \_\_imp\_\_vfprintf

00401b28 T \_printf

00401b38 T \_vfprintf

“nm” é usado principalmente para verificar se uma função em particular está definida em um arquivo objeto. Um 'T' na segunda coluna indica que a função está definida, enquanto um 'U' indica que afunção não foi definida e deverá ser processado pelo lincador.

**Utilitário “ldd” – Lista Bibliotecas de lincagem dinâmica**

O utilitário “ldd” examina um executável e exibe uma lista de bibliotecas compartilhadas que são necessárias para o programa rodar. Por exemplo,

> **ldd hello.exe**

ntdll.dll => /cygdrive/c/Windows/SYSTEM32/ntdll.dll (0x77bd0000)

kernel32.dll => /cygdrive/c/Windows/system32/kernel32.dll (0x77600000)

KERNELBASE.dll => /cygdrive/c/Windows/system32/KERNELBASE.dll (0x75fa0000)

msvcrt.dll => /cygdrive/c/Windows/system32/msvcrt.dll (0x763f0000)

**2.  GNU Make**

O utilitário “make” automatiza os aspectos mundanos da criação de executáveis a partir do código fonte. O “make” usa um arquivo chamado de makefile, que contém regras de como criar os excutáveis.

**2.1  Primeiro exemplo de um Makefile**

Vamos começar com um exemplo simples para transformar um programa Hello-World (hello.c) em um executável (hello.exe) através do utilitário make.

// hello.c

#include <stdio.h>

int main() {

printf("Hello, world!\n");

return 0;

}

|  |
| --- |
|  |

Crie o seguinte arquivo com o nome “makefile” (sem nenhuma extensão), que conterá as regras para criar o executável, e salve no mesmo diretório do código fonte. Use “Tab” para identar os comandos (Não use espaços).

all: hello.exe

hello.exe: hello.o

gcc -o hello.exe hello.o

hello.o: hello.c

gcc -c hello.c

clean:

rm hello.o hello.exe

Execute o utilitário “make” com no exemplo a seguir:

> **make**

gcc -c hello.c

gcc -o hello.exe hello.o

Executar make sem argumenos inicia o alvo “all” do makefile. Um Makefile consiste de um conjunto de regras. Uma regra consiste de 3 partes: um alvo (*target*), uma lista de pré-requisitos e um comando, como segue:

*target*: *pre-req-1* *pre-req-2* ...

*command*

O *target* e os *pre-requisites* são separados por dois ponto (:). O *command* precisa ser precedido por um Tab (nunca espaços).

Quando o make tem que executar uma regra, ele primeiro começa pela localização dos arquivos constantes como pré-requisitos. Se algum dos pré-requisitos possuir uma regra associada a ela, o *make*tenta executar estas regras de antemão.

No exemplo acima, a regra “all” possui como pré-requisito “hello.exe“. O make não pode encontrar o arquivo “hello.exe“, então ele procura por uma regra que crie esse arquivo. A regra “hello.exe” possui  pré-requisito “hello.o“. Novamente, esse arquivo não existe, então o make procura por uma regra que o crie. A regra “hello.o” possui como pré-requisito “hello.c“. O make verifica que o arquivo “hello.c” existe e que é mais novo que o *target* (que não existe). O comando “gcc -c hello.c” é executado. A regra “hello.exe” então executa o seu comando “gcc -o hello.exe hello.o“. Finalmente, a regra “all” não faz nada.

Mais importante, se o pré-requisito não for mais novo que o alvo, o comando não será executado. Em outras palavras, o comando será executado apenas se o alvo estiver desatualizado comparado com seus pré-requisitos. Por exemplo, se re-executarmos o comando *make*:

> **make**

make: Nothing to be done for `all'.

Você pode especificar também o alvo que será executado pelo comando make. Por exemplo, o  alvo “clean” remove os arquivos “hello.o” e “hello.exe“. Você pode assim executar executar novamente o make sem nenhum alvo, que é o mesmo que “make all“.

> **make clean**

rm hello.o hello.exe

> **make**

gcc -c hello.c

gcc -o hello.exe hello.o

NOTAS:

* Se o *command* não for precedido por um Tab, você obterá a mensagem de errro “makefile:4: \*\*\* missing separator. Stop.”
* Se existir não existir um arquivo makefile no diretório atualm você obterá a mensagem de erro “make: \*\*\* No targets specified and no makefile found. Stop.”
* O makefile pode ser nomeado como “makefile“, “Makefile” ou “GNUMakefile“, sem extensão de arquivo.

**2.2  Mais sobre Makefiles**

**Comentário e Continuação**

Um comentário é iniciado com um # e terminado com o final da linha. Uma linha longa pode ser quebrada e continuada por várias linhas com uma barra invertida (\).

**Sintaxe das regras**

Uma sintaxe geral para as regras seria:

*target1* [*target2* ...]: [*pre-req-1* *pre-req-2* ...]

[*command1*

*command2*

......]

As regras são organizadas normalmente de forma que as regras gerais venham primeiro. A regra global é chamada frequentemente de “all“, que é a regra padrão para o comando make.

**Alvos falsos (ou Artificiais)**

Um alvo que não representa um arquivo é chamado de Artificial. Por exemplo, o alvo “clean” no exemplo acima, que é apenas uma etiqueta para um comando. Se o alvo é um arquivo, ele será checado em relação seus pré-requisitos em relação a data. Alvos artificiais serão sempre executados. Os alvos artificiais padrão são: all, clean, install.

**Variáveis**

Uma variável começa com um $ e é inclusa entre parênteses (…) ou entre chaves {…}. Variáveis cujo nome tenham apenas um caractere não necessitam de parênteses. Por exemplo, $(CC), $(CC\_FLAGS), $@, $^.

**Variáveis automáticas**

Variáveis automáticas são configuradas pelo *make*após uma regra ser combinada. Algumas desses variáveis são:

* $@: O nome do alvo.
* $\*: O nome do alvo sem a extensão.
* $<: O primeiro dos pré-requisitos.
* $^: Os nomes dos arquivos de todos os pré-requisitos, separados por espaços, descarta duplicatas.
* $+: Similar ao $^, mas inclui duplicatas.
* $?: Os nomes de todos os pré-requisitos que são mais novos do que o alvo, separados por espaços.

Por exemplo, podemos reescrever o makefile anterior da seguinte forma:

all: hello.exe

# $@ matches the target; $< matches the first dependant

hello.exe: hello.o

gcc -o $@ $<

hello.o: hello.c

gcc -c $<

clean:

rm hello.o hello.exe

**Caminhso virtuais – VPATH & vpath**

Você pode usar VPATH (em letras maiúsculas) para especificar o diretório que deve ser pesquisado por dependências e arquivos alvos. Por exemplo,

# Search for dependencies and targets from "src" and "include" directories

# The directories are separated by space

VPATH = src include

Você pode usar também  vpath (em letras minúsculas) para ser mais preciso sobre os tipos de arquivos e o diretório de busca. Por exemplo,

# Search for .c files in "src" directory; .h files in "include" directory

# The pattern matching character '%' matches filename without the extension

vpath %.c src

vpath %.h include

**Regras padronizadas**

Uma regra padronizada, aquela que usa o caractere de reconhecimento de padrões  '%' como nome de arquivos, pode ser aplicada para criar um alvo, se não houver regra explicita. Por exemplo,

# Applicable for create .o object file.

# '%' matches filename.

# $< is the first pre-requsite

# $(COMPILE.c) consists of compiler name and compiler options

# $(OUTPUT\_OPTIONS) could be -o $@

%.o: %.c

$(COMPILE.c) $(OUTPUT\_OPTION) $<

# Applicable for create executable (without extension) from object .o object file

# $^ matches all the pre-requsities (no duplicates)

%: %.o

$(LINK.o) $^ $(LOADLIBES) $(LDLIBS) -o $@

**2.3  Um exemplo de Makefile**

Esse Makefile exemplo foi extraído no Gui de desenvolvimento C/C++ do Eclipse.

# A sample Makefile

# This Makefile demonstrates and explains

# Make Macros, Macro Expansions,

# Rules, Targets, Dependencies, Commands, Goals

# Artificial Targets, Pattern Rule, Dependency Rule.

# Comments start with a # and go to the end of the line.

# Here is a simple Make Macro.

LINK\_TARGET = test\_me.exe

# Here is a Make Macro that uses the backslash to extend to multiple lines.

OBJS = \

Test1.o \

Test2.o \

Main.o

# Here is a Make Macro defined by two Macro Expansions.

# A Macro Expansion may be treated as a textual replacement of the Make Macro.

# Macro Expansions are introduced with $ and enclosed in (parentheses).

REBUILDABLES = $(OBJS) $(LINK\_TARGET)

# Here is a simple Rule (used for "cleaning" your build environment).

# It has a Target named "clean" (left of the colon ":" on the first line),

# no Dependencies (right of the colon),

# and two Commands (indented by tabs on the lines that follow).

# The space before the colon is not required but added here for clarity.

clean :

rm -f $(REBUILDABLES)

echo Clean done

# There are two standard Targets your Makefile should probably have:

# "all" and "clean", because they are often command-line Goals.

# Also, these are both typically Artificial Targets, because they don't typically

# correspond to real files named "all" or "clean".

# The rule for "all" is used to incrementally build your system.

# It does this by expressing a dependency on the results of that system,

# which in turn have their own rules and dependencies.

all : $(LINK\_TARGET)

echo All done

# There is no required order to the list of rules as they appear in the Makefile.

# Make will build its own dependency tree and only execute each rule only once

# its dependencies' rules have been executed successfully.

# Here is a Rule that uses some built-in Make Macros in its command:

# $@ expands to the rule's target, in this case "test\_me.exe".

# $^ expands to the rule's dependencies, in this case the three files

# main.o, test1.o, and test2.o.

$(LINK\_TARGET) : $(OBJS)

g++ -g -o $@ $^

# Here is a Pattern Rule, often used for compile-line.

# It says how to create a file with a .o suffix, given a file with a .cpp suffix.

# The rule's command uses some built-in Make Macros:

# $@ for the pattern-matched target

# $< for the pattern-matched dependency

%.o : %.cpp

g++ -g -o $@ -c $<

# These are Dependency Rules, which are rules without any command.

# Dependency Rules indicate that if any file to the right of the colon changes,

# the target to the left of the colon should be considered out-of-date.

# The commands for making an out-of-date target up-to-date may be found elsewhere

# (in this case, by the Pattern Rule above).

# Dependency Rules are often used to capture header file dependencies.

Main.o : Main.h Test1.h Test2.h

Test1.o : Test1.h Test2.h

Test2.o : Test2.h

# Alternatively to manually capturing dependencies, several automated

# dependency generators exist. Here is one possibility (commented out)...

# %.dep : %.cpp

# g++ -M $(FLAGS) $< > $@

# include $(OBJS:.o=.dep)

**2.4  Breve sumário**

Nesse artigo, foi apresentado os recursos básicos para que você possa ler e entender Makefiles simples para criar aplicações C/C++. O make é normalmente bem complexo, e pode ser considerado uma linguagem de programação a parte.