Como Construir um Programa Multi-arquivo em C Usando gcc (Linux)

Na construção de programas multiarquivos, um editor de programas pode ser utilizado do mesmo modo descrito no documento *Como Construir um Programa Monoarquivo em C Usando gcc*. A única qualidade adicional desejável num editor de texto para programas multiarquivos é que ele permita a edição de vários arquivos simultaneamente e facilite a passagem de um painel de edição para outro. Já o uso do compilador gcc para criação de um executável resultante de um programa multiarquivo requer o entendimento de algumas opções de compilação adicionais.

Compilação e Ligação Conjugadas

Para compilar e fazer as devidas ligações de um programa composto dos arquivos *arquivo-fonte1*, *arquivo-fonte1*, ..., *arquivo-fonteN* invoque o compilador gcc como:

```
gcc arquivo-fonte1 arquivo-fonte2 ... arquivo-fonteN
```

Ou:

```
gcc arquivo-fonte1 arquivo-fonte2 ... arquivo-fonteN -o \
    arquivo-executável
```

A diferença entre estes dois comandos é que, no segundo, o nome do arquivo executável é especificado. Se este não for explicitamente especificado, o nome do executável será `a.out` ou `a.exe`.

Suponha, por exemplo, que seu programa é constituído pelos arquivos `main.c`, `Arq1.c` e `Arq2.c`, e o nome desejado para o arquivo executável seja `MeuProg`. Então, o comando a seguir produz o resultado desejado:

```
gcc main.c Arq1.c Arq2.c -o MeuProg
```

Compilação e Ligação Separadas

No caso de compilação e ligação separadas, é necessário compilar (literalmente) cada arquivo que compõe o programa separadamente, conforme foi visto anteriormente:

```
gcc –c arquivo-fonte1
gcc –c arquivo-fonte2
...
gcc –c arquivo-fonteN
```

Em seguida, invoca-se o *linker* para fazer as ligações e produzir um arquivo executável do seguinte modo:

```
gcc arquivo-objeto1 arquivo-objeto2 ... arquivo-objetoN
```

Ou:

```
gcc arquivo-objeto1 arquivo-objeto2 ... arquivo-objetoN \
    -o arquivo-executável
```

Considere o programa consistindo dos arquivos `main.c`, `Arq1.c` e `Arq2.c`. Usando compilação e ligação separadas, o programa executável seria construído da seguindo os passos descritos abaixo.

Passo 1 – Compilação:

```
gcc –c main.c
gcc –c Arq1.c
gcc –c Arq2.c
```

Como resultado da execução desse comandos, são criados os arquivos objetos `main.o`, `Arq1.o` e `Arq2.o`.

Passo 2 – Ligação:

```
gcc main.o Arq1.o Arq2.o -o MeuProg
```

A vantagem deste método em comparação ao método anterior é que, se apenas um arquivo precisar ser modificado após todos terem sido compilados, você só precisará recompilar esse arquivo. Por exemplo, suponha que, no caso do último exemplo, você execute seu programa e descubra que ele apresenta um erro. Suponha ainda que este erro é resultante de uma instrução equivocada localizada no arquivo `Arq2.c`. Então, após corrigir este arquivo, você precisaria apenas emitir os comandos a seguir para obter uma nova versão do seu programa executável:

```
gcc Arq2.c
gcc main.o Arq1.o Arq2.o -o MeuProg
```

Se você ainda não estiver convencido das vantagem deste último método, suponha que, ao invés de três, seu programa é composto por dezenas de arquivos-fonte... Que tal um programa constituído por 300 arquivos-fonte?

Provendo Informações sobre Bibliotecas e Diretórios

No compilador gcc, pode-se informar o *linker* onde uma dada biblioteca reside utilizando a opção `–l`. O único módulo da biblioteca padrão de C que precisa ser especificado separadamente desta maneira é o módulo math utilizando a opção `–lm`, como em:

```
gcc OperacoesMat.c -lm -o OperacoesMat
```

Suponha que você tem uma biblioteca denominada `Bib`. Então o nome do arquivo que contém o código objeto da biblioteca deve ser `Bib.a` e gcc deve ser invocado usando a opção `–lBib`. O arquivo onde se encontra a biblioteca deve residir num diretório padrão ou então especificado usando a opção `–L`. Por exemplo:

```
gcc MeuProg.c –L~/Bibliotecas -lBib -o MeuProg
```

Se você precisar indicar algum diretório onde o compilador deve procurar arquivos para inclusão, utilize a opção `–I`, que tem o seguinte formato:

`-I`*`dir`*

onde *`dir`* é uma especificação de diretório segundo o sistema operacional em uso.

Make e Arquivos Makefiles

Make é um programa utilitário encontrado em sistemas operacionais da família Unix e distribuído junto com alguns ambientes de desenvolvimento[1]. Na ausência de um ambiente IDE, este utilitário pode facilitar bastante a construção (i.e., compilação e ligação) de programas multiarquivos. O utilitário make é particularmente útil quando utilizado na construção de programas grandes, consistindo de muitos arquivos, pois ele recompila apenas os arquivos que realmente precisam ser recompilados.

Makefile é um arquivo escrito numa linguagem própria que o programa make" entende. Esta seção descreve o utilitário make e ensina como construir arquivos *makefiles* simples.

O Programa Make

Quando o utilitário make" é executado sem informação sobre qual arquivo processar, ele procura um arquivo denominado `Makefile` ou `makefile`:"file no diretório corrente. Se o arquivo a ser processado tiver um outro nome ele precisa ser especificado na chamada de make. Em sua forma mais simples, uma chamada de make usa o seguinte formato:

make -f *nome-do-arquivo-makefile*
alvo

onde *nome-do-arquivo-makefile* é o nome do arquivo *makefile* a ser processado e *alvo* é o alvo (v. abaixo) a ser processado. Tanto o nome do arquivo quanto o alvo são opcionais. Conforme já foi apresentado, se o nome do arquivo não for especificado, o utilitário make procura um arquivo denominado o `Makefile` (ou `makefile`). Se o alvo não for especificado, o programa make considera o primeiro alvo: encontrado no arquivo *makefile* processado, conforme será visto a seguir.

Componente de um Arquivo Makefile

Os principais componentes: de um arquivo *makefile* são **regras** que assumem o seguinte formato:

[1] Alguns sistemas operacionais da família Microsoft Windows possuem um programa similar denominado *nmake*. O utilitário make descrito aqui é aquele distribuído pela organização GNU. Outros programas make funcionam de modo similar, mas cada um apresenta suas próprias peculiaridades. Por exemplo, para o utilitário make GNU o alvo padrão é o primeiro encontrado num arquivo *makefile*, enquanto que outros programas similares consideram o alvo denominado *all* como alvo padrão.

```
alvo:dependências
[TAB]comando1
[TAB]comando2
    ...
[TAB]comandoN
```

onde:

- *alvo* é o alvo: que a regra representa. Usualmente, um alvo consiste de um nome de arquivo resultante do processamento de um programa (por exemplo, um arquivo gerado por um compilador). Um alvo também pode ser o nome dado a uma ação a ser executada (v. abaixo).
- *dependências:*: representam nomes de arquivos ou alvos utilizados em outras regras. Quando há mais de uma dependência, elas devem ser separadas por espaços em branco e, quando não há nenhuma dependência, o espaço reservado para *dependências* deve ser deixado em branco.
- *comando1*, ..., *comandoN* são comandos: do sistema operacional[2] que serão executados quando cada uma das dependências (se existir alguma) for satisfeita. É importante notar que precedendo cada comando deve existir um caractere de tabulação (representado por `[TAB]` no esquema acima). Portanto, se seu editor de texto transforma tabulações em espaços em branco, desabilite esta opção.

Tipicamente, um comando faz parte de uma regra com dependências (ou **pré-requisitos**:) e serve para criar o arquivo que representa o alvo da regra quando algum dos pré-requisitos é alterado.

Conforme foi antecipado acima, os comandos são executados apenas quando todas as respectivas dependências são satisfeitas. Quando uma dependência consiste de um arquivo, ela será satisfeita se a data da última modificação do arquivo for mais recente do que a data da última modificação do alvo. Tipicamente, arquivos objetos são considerados dependentes de arquivos-fonte e estes são considerados dependentes dos arquivos de cabeçalho que eles incluem. Por exemplo, suponha que você tenha um programa multiarquivo contendo os arquivos: `arq1.c`, `arq1.h`, `arq2.c`, `arq2.h` e `main.c`. Suponha ainda que o nome desejado para o executável seja `MeuProg` e que os arquivos de cabeçalho sejam incluídos pelos arquivos de programa de acordo com a tabela a seguir:

Arquivo de programa	**Inclui arquivo de cabeçalho...**
`arq1.c`	`arq1.h`
`arq2.c`	`arq2.h`
`main.c`	`arq1.h` e `arq2.h`

Então, poder-se-ía escrever o seguinte arquivo *makefile* para automatizar a criação do programa executável desejado:

[2] Embora o interesse aqui seja utilizar make para compilação e ligação de programas, pode-se utilizar muitos outros comandos disponíveis num sistema operacional.

```
MeuProg: main.o arq1.o arq2.o
      gcc main.o arq1.o arq2.o –o MeuProg

main.o: main.c arq1.h arq2.h
      gcc –c main.c –o main.o

arq1.o: arq1.c arq1.h
      gcc –c arq1.c –o arq1.o

arq2.o: arq2.c arq2.h
      gcc –c arq2.c –o arq2.o
```

A primeira regra do arquivo *makefile*:

```
MeuProg: main.o arq1.o arq2.o
      gcc main.o arq1.o arq2.o –o MeuProg
```

informa o utilitário make que o alvo principal do arquivo é `MeuProg`. Este alvo tem três dependências: `main.o`, `arq1.o` e `arq2.o`, cada uma das quais é tanto um nome de arquivo resultante de compilação quanto um alvo de regras subseqüentes. O comando associado ao alvo principal será executado se cada um destes arquivos existe e pelo menos um deles é mais recente do que o alvo `MeuProg`.

Considere agora o pré-requisito `main.o` do alvo principal. Se este arquivo não existir, o utilitário make tentará obtê-lo utilizando a regra que tem este pré-requisito como alvo:

```
main.o: main.c arq1.h arq2.h
      gcc –c main.c –o main.o
```

Este alvo tem três dependências: `main.c`, `arq1.h` e `arq2.h`, cada uma das quais é um nome de arquivo-fonte. Se algum deles não for encontrado, o alvo `main.c` não poderá ser criado; conseqüentemente, o comando associado ao alvo principal também não será executado. Por outro lado, se os três arquivos que compõem as dependências do alvo `main.c` existem, o comando:

```
gcc –c main.c –o main.o
```

será executado[3] resultando no arquivo `main.o`. Assim, se as demais dependências da regra associadas ao alvo principal (i.e., `MeuProg`) forem satisfeitas, o comando associado a esta regra será executado.

Agora, suponha que, enquanto avalia a primeira regra, o utilitário make descobre que o arquivo `main.o` existe. Como também existe uma regra que especifica como este arquivo pode ser obtido, o utilitário examinará esta regra para verificar se o arquivo precisa ser reconstruído. Assim, se todos os arquivos que constituem as dependências do alvo `main.o` existirem e algum deles for mais recente do que o arquivo `main.o`, o comando que reconstrói este arquivo será executado.

O mesmo raciocínio empregado acima para a dependência `main.o` do alvo principal aplica-se às demais dependências (i.e., `arq1.o` e `arq2.o`) deste alvo.

Quando uma regra não possui dependências e não representa um nome de arquivo, os comandos correspondentes serão sempre executados. Por exemplo:

[3] Lembre-se que, por enquanto, se está supondo que o arquivo `main.o` não existe.

```
limpa:
      rm -f *.o # Remove todos os arquivo com extensão .o (Unix/Linux)
```

Quando o alvo `limpa` é processado, o respectivo comando é executado independentemente da avaliação de qualquer dependência.

Quando um alvo não é primeiro nem constitui dependência de nenhuma regra, ele só será considerado se for explicitamente especificado na invocação de make, como, por exemplo[4]:

```
% make limpa
```

Algumas vezes, é útil ter um alvo que force uma reconstrução completa do programa. Por exemplo, isto poderia ser feito utilizando o alvo `reconstroi` no arquivo *makefile* seguinte:

```
MeuProg: main.o arq1.o arq2.o
      gcc main.o arq1.o arq2.o –o MeuProg

main.o: main.c arq1.h arq2.h
      gcc –c main.c –o main.o

arq1.o: arq1.c arq1.h
      gcc –c arq1.c –o arq1.o

arq2.o: arq2.c arq2.h
      gcc –c arq2.c –o arq2.o

limpa:
      rm -f *.o core

reconstroi: limpa MeuProg
```

Note que o alvo `reconstroi` não possui nenhum comando associado. Este tipo de alvo é denominado **alvo simbólico**: e deve ter um nome único que não coincida com o nome de qualquer arquivo no diretório corrente.

Quando um comando é executado, ele retorna um valor que indica se sua execução foi bem sucedida ou não. O utilitário make examina este valor e, se ele indicar que a execução do comando não foi bem sucedida, o alvo associado a este comando não será considerado satisfeito. Por exemplo, considerando o último arquivo *makefile* apresentado, se a execução do alvo `limpa` não for bem sucedida (o que ocorreria se não houvesse nenhum arquivo denominado `core`), o alvo `reconstroi` será abandonado, deixando, assim, de considerar a regra associada ao alvo `MeuProg`. Para fazer com que o utilitário make ignore o valor retornado por algum comando, precede-se o nome do comando, cujo status deve ser ignorado, com o sinal de menos. Considerando o último exemplo, isto poderia ser feito do seguinte modo:

```
limpa:
      -rm -f *.o core
```

O que foi exposto até aqui sobre make e *makefiles* é fundamental. Se você já entendeu como o utilitário make funciona, o que será apresentado a seguir apenas incrementa seu conhecimento com o objetivo de facilitar ainda mais a construção de programas multiarquivos utilizando make e

[4] O símbolo % é utilizado para representar o prompt de linha de comando do sistema operacional utilizado.

makefiles. Se você ainda não entendeu como make funciona, releia o que foi apresentado até aqui antes de prosseguir. Caso contrário, os ingredientes que podem ser acrescentados a arquivos *makefiles* não serão de grande valia.

Comentários

Um **comentário**: num arquivo *makefile* é qualquer seqüência de caracteres que segue o símbolo **#** e termina quando encerra a linha que o contém. Por exemplo:

```
# Isto é um comentário de um arquivo makefile
```

Macros

Macros: (ou **variáveis**:) facilitam a alteração de um arquivo *makefile* do mesmo modo que constantes simbólicas facilitam a alteração de programas escritos em C. Uma definição de macro num arquivo *makefile* tem o seguinte formato:

nome-da-macro=valor

Por exemplo:

```
COMPILADOR=gcc
```

Uma macro pode ser expandida no interior de uma regra ou na definição de outra macro utilizando a seguinte sintaxe:

$(*nome-da-macro*)

Por exemplo, considerando a definição da macro `COMPILADOR` acima, a regra a seguir:

```
arq1.o: arq1.c
      $(COMPILADOR) –c arq1.c –o arq1.o
```

após a expansão da macro `COMPILADOR`, tornar-se-ía:

```
arq1.o: arq1.c
      gcc –c arq1.c –o arq1.o
```

Deve-se ressaltar que make importa todas as variáveis de ambiente do sistema operacional, de modo que é possível fazer referência a uma tal variável como se ela fosse uma macro. Por exemplo, a variável de ambiente `PATH` pode ser referenciada com se fosse uma macro assim:

```
$(PATH)
```

Outro ponto importante é que macros podem ser definidas na linha de comando quando o programa make é executado. Por exemplo:

```
% make OPCOES=-std=ansi
```

Este último comando iniciaria o utilitário make e definiria a macro `OPCOES` com o valor `-std=ansi`. Macros definidas na linha de comando têm precedência sobre macros definidas no interior de qualquer arquivo *makefile*.

Cada programa make possui várias macros definidas como padrões. Você pode tomar conhecimento destas macros utilizando o comando:

```
% make -p
```

Modelos de Arquivos Makefile Simples para Programas Multiarquivos

Considere novamente aquele programa multiarquivo hipotético apresentado como primeiro exemplo de *makefile*:modelos de desta seção. O programa consiste dos arquivos-fonte: `arq1.c`, `arq1.h`, `arq2.c`, `arq2.h` e `main.c`; o nome desejado para o executável é `MeuProg`. Então, um arquivo *makefile* apropriado para automatizar o processo de criação do programa executável poderia ser[5]:

```
MeuProg: main.o arq1.o arq2.o
      gcc main.o arq1.o arq2.o –o MeuProg

main.o: main.c arq1.h arq2.h
      gcc –c main.c –o main.o

arq1.o: arq1.c arq1.h
      gcc –c arq1.c –o arq1.o

arq2.o: arq2.c arq2.h
      gcc –c arq2.c –o arq2.o

limpa:
      -rm -f *.o core

reconstroi: limpa MeuProg
```

O alvo `limpa` deste *makefile* é útil pois permite que todos os arquivos objetos sejam apagados simplesmente invocando make conforme foi discutido anteriormente. O alvo `reconstroi` é útil quando se deseja reconstruir (compilar e ligar) incondicionalmente todos os arquivos-fonte do programa. Este alvo também precisa ser especificado explicitamente quando make é invocado:

```
% make reconstroi
```

Utilizando macros pode-se tornar o arquivo *makefile* do último exemplo mais flexível e fácil de alterar:

```
# Compilador utilizado
COMP=gcc

# Opções de compilação (altere, se desejar outras opções)
OPCOES=-c –Wall –std=c99
```

[5] O arquivo `arq1.c` inclui `arq1.h`, o arquivo `arq2.c` inclui `arq2.h` e o arquivo `main.c` inclui `arq1.h` e `arq2.h`. Este arquivo makefile é muito parecido com o primeiro exemplo apresentado nesta seção, mas eles não são exatamente os mesmos.

```
MeuProg: main.o arq1.o arq2.o
	$(COMP) main.o arq1.o arq2.o –o MeuProg

main.o: main.c arq1.h arq2.h
	$(COMP) $(OPCOES) main.c –o main.o

arq1.o: arq1.c arq1.h
	$(COMP) $(OPCOES) arq1.c –o arq1.o

arq2.o: arq2.c arq2.h
	$(COMP) $(OPCOES) arq2.c –o arq2.o

limpa:
	-rm -f *.o core

reconstroi: limpa MeuProg
```

O último exemplo de *makefile* apresentado pode ainda ser melhorado um pouco mais como mostrado a seguir:

```
# Compilador utilizado
COMP=gcc

# Opções de compilação (altere, se desejar outras opções)
OPCOES_COMP=-c –Wall –std=c99

# Opções de ligação (acrescente, se desejar, alguma opção)
OPCOES_LINK=

# Arquivos-fonte (modifique/acrescente)
FONTES=main.c arq1.c arq2.c

# A macro a seguir informa que os arquivos-objeto são
# obtidos a partir dos arquivos-fonte, substituindo
# a extensão .c pela extensão .o
OBJETOS=$(FONTES:.c=.o)

# Nome do arquivo executável (modifique)
EXECUTAVEL=MeuProg

$(EXECUTAVEL): $(OBJETOS)
	$(COMP) $(OPCOES_LINK) $(OBJETOS) -o $@

arq1.o: arq1.c arq1.h
	$(COMP) $(OPCOES_COMP) arq1.c –o arq1.o

arq2.o: arq2.c arq2.h
	$(COMP) $(OPCOES_COMP) arq2.c –o arq2.o

limpa:
	-rm -f *.o core

reconstroi: limpa MeuProg
```

Na primeira regra deste último *makefile*, o símbolo `@` representa uma macro predefinida que resulta no nome do alvo da regra onde esta macro se encontra. Ou seja, na regra:

```
$(EXECUTAVEL): $(OBJETOS)
	$(COMP) $(OPCOES_LINK) $(OBJETOS) -o $@
```

A macro `@` será expandida em `MeuProg`, após a expansão da macro `EXECUTAVEL`.

Existem outras macros predefinidas que facilitam a criação de arquivos *makefiles*, mas uma completa discussão sobre estas macros está além do escopo deste texto. Se você utilizar make com freqüência, compensa encontrar um bom texto sobre make e *makefiles* e aprofundar o estudo.

Bibliografia

- Oliveira, Ulysses de, *Programando em C: Volume I – Fundamentos*, Editora Ciência Moderna, 2007. (**Capítulo 4**)