Universidade de São Paulo

Instituto de Ciências Matemáticas e de Computação

Uma breve introdução à criação de bibliotecas e makefiles em C/C++

Moacir Pereira Ponti Jr.

2011

1 Arquivos de cabeçalho e bibliotecas

É muito interessante criar bibliotecas cujas funções (classes e métodos, no caso de linguagens orientadas a objetos) possam ser reutilizadas em diversos programas. Você pode inclusive distribuir essas bibliotecas, com a opção de "esconder" o código fonte utilizado nas funções. Geralmente ao programar em `C/C++`, já utilizamos muitas bibliotecas com funções prontas como quando incluímos `<stdio.h>` ou `<iostream>`.

Para criarmos nossa própria biblioteca é preciso ter arquivos de cabeçalho (com extensão `'.h'`) e arquivos de biblioteca (com extensão dependente do compilador e sistema operacional, no `gcc` a extensão geralmente utilizada é `'.a'`).

Vamos mostrar um exemplo em `C` de biblioteca contendo uma única função, que receba um valor inteiro e retorne seu fatorial. O arquivo cabeçalho será `funcoes.h`, e o arquivo com o código fonte da função será nomeado `funcoes.c` e tem o seguinte conteúdo:

```
#include "funcoes.h"

int fatorial(int x) {
    int a, fatx=1;
    for (a=x; a>1; a--) {
       fatx = fatx*a;
    }
    return fatx;
}
```

Veja que já incluímos `funcoes.h`. Isso será necessário para o compilador reconhecer a função como sendo parte da biblioteca.

O arquivo cabeçalho irá conter informações apenas da *interface*, tipicamente as assinaturas (ou protótipos) das funções. A extensão `.h` vem da palavra *header* (cabeçalho em inglês). Nesse caso teremos o arquivo cabeçalho, nomeado `funcoes.h`, definido da seguinte forma:

```
#ifndef FUNCOES_H
#define FUNCOES_H

int fatorial(int);

#endif
```

As primeiras linhas (`#ifndef e #define`) tem a função de verificar se o arquivo cabeçalho já foi incluído num projeto, antes de incluí-lo novamente de forma desnecessária.

Temos então dois arquivos: `funcoes.c` e `funcoes.h`. Para utilizá-los, devemos gerar o arquivo objeto da biblioteca:

```
$ gcc -c funcoes.c -o funcoes.o
```

Se desejar criar um arquivo `.a`, que possa ser distribuído, utilize o comando:

```
$ ar -cru libfuncoes.a funcoes.o
```

Os arquivos `.a` são bibliotecas estáticas, que tem a vantagem de poder carregar vários objetos. Nesse caso não faz muita diferença, mas o comando é bastante útil em projetos maiores. Se quiser saber mais sobre o `ar` entre em seu manual digitando `$ man ar`.

Agora podemos copiar todos os arquivos para um diretório separado, por exemplo `./biblioteca`.

1.1 Utilizando a biblioteca

Agora, sempre que for necessário usar funções definidas no arquivo `funcoes.c`, incluimos o arquivo `funcoes.h` no programa que vamos implementar. Abaixo um exemplo de código fonte, que iremos nomear `programa.c`, que utiliza a biblioteca `funcoes`:

```
#include <stdio.h>
#include "funcoes.h"

int main(void) {
    int b;

    printf("Valor para calcular o fatorial: ");
    scanf("%d", b);

    printf("\n O fatorial de %d = %d", b, fatorial(b));

    return 0;
}
```

Repare que não utilizamos os sinais de menor/maior para incluir `funcoes.h`, como na biblioteca `stdio.h`. Eles são usados quando o arquivo cabeçalho estiver instalado num diretório padrão do sistema.

Agora há duas opções de compilação: uma usando o arquivo objeto e outra usando a biblioteca estática

Usando biblioteca estática: é preciso instruir o compilador com as opções de *includes* e edição de ligações (*linker*) para que a biblioteca possa ser incluída no programa executável. No `gcc` isso é feito utilizando:

```
$ gcc programa.c -I./biblioteca -L./biblioteca -lfuncoes -o programa
```

Onde os *flags* significam:

- `-I` inclui diretórios onde existam cabeçalhos utilizados no código fonte.
- `-L` inclui diretórios onde existam bibliotecas estáticas que devem ser incluídas no programa.
- `-lfuncoes` utiliza o arquivo de biblioteca criado, `libfuncoes.a`
- `-o programa` gera como saída o executável `programa`

Usando o arquivo objeto: é preciso instruir o compilador com as opções de *includes* e manualmente indicar onde está o arquivo objeto. No `gcc` isso é feito utilizando:

```
$ gcc programa.c ./biblioteca/funcoes.o -I./biblioteca -o programa
```

Repare que, nesse caso, não utilizamos os flags de diretórios de biblioteca `-L` nem incluímos a biblioteca estática `-lfuncoes`, pois utilizamos o arquivo objeto diretamente.

2 Compilação e 'makefiles'

A compilação e ligação de códigos fonte pode ser uma tarefa complexa quando se utiliza referências a diversas bibliotecas e quando temos muitos arquivos fonte (`.c`, `.cpp`, ...) para juntar à compilação e gerar o programa. A forma mais simples de compilar arquivos e obter um executável utilizando o `gcc` é, por exemplo:

```
$ gcc programa.c biblioteca.c funcoes.c -o programa
```

Repare que, nesse exemplo, para obtermos o programa precisamos de três arquivos de código fonte. Muitas vezes temos também que adicionar o caminho para bibliotecas que usamos no nosso programa, o que aumenta a linha de comando, como por exemplo:

```
$ gcc programa.c biblioteca.c funcoes.c -o programa -L/home/lib
```

Ainda, quando alteramos apenas um dos fontes, geralmente compilamos tudo novamente, de forma manual, para obter o programa desejado. Para minimizar esse esforço, podemos utilizar arquivos "*makefile*" em conjunto com o utilitário `make`.

Makefiles são arquivos com um formato especial que auxiliam na compilação e ligação de projetos. *Make* é um programa especialmente criado para ler esses arquivos, que contém as instruções para tudo o que deve ser feito ("*make*").

Se você executar:

```
$ make
```

esse programa irá procurar por um arquivo chamado `Makefile` no diretório atual, e irá executar utilizando as instruções desse arquivo. Se você tiver mais do que um *makefile* ou seu arquivo possuir um nome diferente, você pode usar o comando:

```
$ make -f MeuMakefile
```

que especifica o arquivo que você quer utilizar para gerar seu programa.

2.1 Arquivos 'Makefile'

Um makefile é um arquivo texto composto basicamente de alvos (*target*), dependências (*dependencies*) e comandos (*system commands*), na forma:

```
target: dependencies
<TAB>system command
```

onde `<TAB>` representa uma tabulação. No arquivo você realmente precisa usar essa tabulação para o programa identificar corretamente os comandos a serem executados.

Podemos definir um arquivo da seguinte forma:

```
all:
<TAB>gcc programa.c biblioteca.c funcoes.c -o programa -L/home/lib
```

e salvá-lo com o nome "Makefile". Para executar:

```
$ make
```

No exemplo acima usamos o alvo padrão, chamado *all*. Esse alvo será executado sem qualquer dependência, ou seja, não precisamos que nenhum outro arquivo exista para executar *all*. Nesse caso, a linha de comando é executada diretamente.

2.2 Dependências

Podemos definir múltiplos alvos, cada um com dependências diferentes, para que o `make` possa executar alvos a partir das dependências, ou então para que possamos escolher qual alvo desejamos executar.

Suponha que tenhamos apenas dois arquivos para compilar, por exemplo, `programa.c` e `funcoes.c`, e queremos gerar o executável `programa`. Podemos definir o `Makefile` da seguinte forma:

```
programa: funcoes.o programa.o
	gcc programa.o funcoes.o -o programa

programa.o: programa.c
	gcc -c programa.c

funcoes.o: funcoes.c
	gcc -c funcoes.c
```

Repare que não usamos mais o indicador `<TAB>`, mas você deverá utilizar uma tabulação para que o arquivo funcione. Agora temos três alvos. O utilitário `make` irá tentar resolver as dependências dentro de cada alvo disponível, executando o comando para gerar cada arquivo necessário. Caso um dos arquivos já exista e não tenha sido modificado, ele não é recompilado.

No exemplo, inicialmente o `make` irá tentar executar o alvo `programa`. Como este possui duas dependências: `programa.o` e `funcoes.o`, será preciso antes executar os respectivos alvos para satisfazer as dependências e compilar o programa.

Supondo que no diretório existam apenas os arquivos `.c`, ao executarmos `make`, a sequência de comandos seria:

```
gcc -c funcoes.c
gcc -c programa.c
gcc programa.o funcoes.o -o programa
```

Ou seja, primeiro é preciso criar o arquivo objeto `funcoes.o`, a seguir o arquivo objeto `programa.o`, para finalmente realizar a ligação e obter o executável. Experimente alterar apenas um dos arquivos e executar `make` para verificar que ele recompila apenas as dependências de arquivos que foram alterados.

Podemos definir dependências com funções específicas como, por exemplo:

```
programa: funcoes.o programa.o
	gcc programa.o funcoes.o -o programa

programa.o: programa.c
	gcc -c programa.c

funcoes.o: funcoes.c
	gcc -c funcoes.c

clean:
	rm -rf programa *.o
```

Veja que adicionamos o alvo `clean` que é responsável por apagar o binário `programa`, todos os objetos , ou seja, todos os arquivos com extensão `.o`. Para que o `make` execute apenas a limpeza, utilizamos:

```
$ make clean
```

Definir esse tipo de dependência é útil quando queremos rapidamente excluir arquivos para recompilar completamente o projeto.

2.3 Variáveis, comentários e detalhes

É possível utilizar variáveis e comentários úteis quando queremos definir caminhos, opções de compilação, entre outros. Por exemplo:

```
# Comentario do makefile
# utilizaremos a variavel CP para definir o compilador a ser utilizado
CP=gcc

# a variavel COPT sera utilizada para adicionar opcoes a compilacao
COPT=-c -Wall

programa: funcoes.o programa.o
	$(CP) programa.o funcoes.o -o programa

programa.o: programa.c
	$(CP) $(COPT) programa.c

funcoes.o: funcoes.c
	$(CP) $(COPT) funcoes.c

clean:
	rm -rf *o programa
```

A variável é criada simplesmente atribuindo um valor. E pode ser usada utilizando o operador cifrão, na forma `$(VARIAVEL)`. Nesse exemplo, poderíamos facilmente trocar o compilador para `g++` ou alterar alguma opção de compilação.

Se uma linha é muito grande, você pode inserir uma barra invertida, seguida imediatamente por uma quebra de linha (ou seja, pressionando ENTER). Um exemplo:

```
CP=gcc
```

```
p1:
      $(CP) p1.o biblioteca.h biblioteca.o funcoes.h funcoes.o \
            lista.h lista.o -o p1
```

2.4 Criando mais do que um executável

Para criar mais do que um executável podemos usar o alvo `all`. Por exemplo, para criar três executáveis `p1`, `p2` e `p3`:

```
all: p1 p2 p3

p1: funcoes.o prog1.o
      gcc prog1.o funcoes.o -o p1

p2: biblioteca.o prog2.o
      gcc prog2.o biblioteca.o -o p1

p3: biblioteca.o funcoes.o prog3.o
      gcc prog3.o biblioteca.o funcoes.o -o p1
```

Nesse caso cada executável tem suas dependências, que devem ser definidas, se necessário no arquivo.

2.5 Gerando um arquivo 'tar'

Muitas vezes precisamos gerar um arquivo compactado com todos os arquivos do projeto. Adicionando um alvo extra podemos gerar por exemplo um arquivo `tar`:

```
tar:
      tar cfv programa.tar funcoes.h funcoes.c biblioteca.h \
              biblioteca.c lista.h lista.c programa.c
```

Nesse caso, assim como no `clean`, não há dependências, sendo o arquivo "`programa.tar`" criado quando chamado o comando:

```
$ make tar
```

2.6 Erros comuns

Os erros geralmente estão relacionados ao uso incorreto da tabulação (`<TAB>`). Infelizmente elas são difíceis de visualizar. Uma dica é entrar no editor, posicionar o cursor no início da linha e mover o cursor para a frente uma vez, se ele pular vários espaços, temos uma tabulação.

Os erros mais comuns são:

1. Esquecer de inserir a tabulação antes do início dos comandos,
2. Inserir uma tabulação no início de uma linha em branco,
3. Não inserir uma quebra de linha logo após uma barra invertida (quando se utiliza esse recurso).

2.7 Compilando um projeto com subdivisão em diretórios

Muitos projetos em `C/C++` utilizam diretórios diferentes para armazenar o código fonte, os objetos e bibliotecas. Geralmente utilizam a estrutura:

```
./projeto
     ./include
     ./obj
     ./lib
     ./src
```

No diretório `include` estão todos os cabeçalhos, no diretório `obj` todos os arquivos objeto, no diretório `lib` as bibliotecas utilizadas e em `src` os fontes. O uso de make auxilia muito nesses casos.

Suponha um projeto em `C++` que irá possuir um único executável `manage`, e que utilize uma biblioteca `Product` que contém a implementação de uma classe de mesmo nome. Assim, temos os seguintes arquivos e diretórios:

```
./projeto
     ./include
          Product.h
     ./obj
     ./lib
     ./src
          Product.cc
          manage.cc
```

Queremos compilar `manage.cc`, gerando um executável `manage` no diretório do projeto e ao mesmo tempo gerar uma biblioteca para `Product`, no diretório `lib`. No final teremos:

```
./projeto
     ./include
          Product.h
     ./obj
          Product.o
     ./lib
          libProduct.a
     ./src
          Product.cc
          manage.cc
     Makefile
     manage
```

Para isso criaremos um Makefile com as seguintes variáveis:

```
CC=g++

LIB=./lib
INCLUDE=./include
SRC=./src
OBJ=./obj

LIBFLAGS = -lProduct
FLAGS = -Wall
```

A primeira variável define o `g++` como compilador, as próximas quatro nos auxiliarão a controlar os diretórios, e a seguir as váriaveis de *flag* irão informar ao compilador quais opções utilizar. Como iremos criar uma biblioteca para `Product`, incluímos a biblioteca como se já estivesse criada em `LIBFLAGS` (que deverá conter todas as bibliotecas estáticas a serem incluídas no projeto). Já em `FLAGS` colocamos opções que desejamos incluir na compilação.

Como o objetivo é gerar um só executável a partir de `manage.cc` e esse depende da existência de uma biblioteca, teremos:

```
manage: Product
      $(CC) $(SRC)/manage.cc $(FLAGS) -I$(INCLUDE) -L$(LIB) $(LIBFLAGS) \
            -o manage
```

```
Product:
      $(CC) -c $(SRC)/Product.cc $(FLAGS) -I$(INCLUDE) -o $(OBJ)/Product.o
      ar -cru $(LIB)/libProduct.a $(OBJ)/Product.o
```

Ao iniciar, o make irá tentar resolver `manage`, que depende de `Product`. O target `Product` primeiro compila o arquivo `Product.cc`, gerando um objeto `Product.o` na pasta `obj`.

A seguir, cria uma biblioteca no diretório `lib`. Finalizada a dependência, `manage.cc` é compilado e um arquivo executável é criado no diretório do projeto.

Opcionalmente, podemos incluir uma opção para limpar o projeto:

```
clean:
      rm manage $(SRC)/*~ $(OBJ)/*o $(LIB)/*a
```

Apaga o executavel, todos os arquivos textos de backup, objetos e a biblioteca.

O arquivo Makefile final fica assim:

```
# compilador
CC=g++

# variaveis com diretorios
LIB=./lib
INCLUDE=./include
SRC=./src
OBJ=./obj

# opcoes de compilacao
LIBFLAGS = -lProduct
FLAGS = -Wall

manage: Product
     $(CC) $(SRC)/manage.cc $(FLAGS) -I$(INCLUDE) -L$(LIB) $(LIBFLAGS) \
           -o manage

Product:
     $(CC) -c $(SRC)/Product.cc $(FLAGS) -I$(INCLUDE) -o $(OBJ)/Product.o
     ar -cru $(LIB)/libProduct.a $(OBJ)/Product.o

clean:
     rm *~ manage $(OBJ)/*o $(LIB)/*a
```