Este artigo tem o objetivo de explicar e orientar o uso da diretiva include e do uso de Makefile em seus projetos. Suponho que você tenha um pouco (não precisa ser muito) de conhecimento em C e saiba usar o compilador gcc sobre o GNU/Linux.

A diretiva include

Vamos começar pela diretiva *include*, depois prosseguiremos com o nosso exemplo (durante todo o artigo iremos abordar um só exemplo). Para programas escritos em C, antes da compilação propriamente dita, há um processo que chamamos de pré-compilação.

O papel das diretivas é instruir o pré-processador (parte do compilador responsável pelo pré-processamento) a realizar determinadas tarefas.

A diretiva *define*, por exemplo, instrui o pré-processador a substituir um texto por outro texto no código-fonte do programa.

A diretiva *include* instrui o pré-processador a incluir num arquivo-fonte o conteúdo de um outro arquivo, geralmente chamado de *header file* (arquivo de cabeçalho). Estes arquivos de cabeçalho tem extensão .h, como em *stdio.h*.

Nosso exemplo constará de 3 arquivos:

main.c;

metodos.h;

e metodos.c.

Convencionalmente, arquivos terminados com extensão .c são arquivos-fonte escritos em C. E, como dito antes, arquivos com extensão .h são arquivos de cabeçalho. Os arquivos de cabeçalhos geralmente contém declarações de funções e variáveis definidas em outros arquivos com extensão .c. Isso é necessário e importante, pois centraliza todas as declarações de um ou mais arquivos fontes em um ou mais arquivos de cabeçalho.

Se quisermos usar os métodos definidos em metodos.c em main.c, devemos colocar uma declaração include no arquivo main.c "apontando" para o arquivo metodos.h. Vamos começar o nosso exemplo. Na sua pasta pessoal crie outra pasta com o nome "projetoC". Agora crie os seguintes arquivos dentro da pasta projetoC:

Arquivo *metodos.c*:

```
#include<stdio.h>

static int global;

void IniciaGlobal()
{
  global =  0;
}

int GetGlobal()
{
   return global;
}

void Metodo1()
{
  printf("metod 1");
  global++;
}

void Metodo2()
{
  printf("metodo 2");
  global++;
}
```

Arquivo *metodos.h*:

```
#ifndef METODOS_H
#define METODOS_H
void IniciaGlobal();
int GetGlobal();
void Metodo1();
void Metodo2();
#endif
```

Arquivo *main.c*:

```
include "metodos.h"
include <stdio.h>

main()
{

    IniciaGlobal();
    Metodo1();
    printf("\n");
    Metodo2();
   printf("\n Número de acessos aos metodos do arquivo metodos.c = %d",GetGlobal());
   return 0;
}
```

Explicando um pouco:

O arquivo main.c contém duas diretivas include. Observe que há duas formas diferentes de incluir um arquivo: usando aspas e usando os símbolos < e >.

O uso de aspas indica ao pré-processador a buscar o arquivo inicialmente dentro do diretório onde main.c está e só depois procurar em outros diretórios .

O uso dos símbolos < e > instrui o pré-processador a buscar o arquivo incluído no diretório include padrão do compilador ou do sistema. No caso do *Linux* esse diretório pode ser /usr/include. No entanto, o padrão ANSI/C deixa a coisa um pouco solta, permitindo assim flexibilidade para os compiladores adaptarem a idéia de acordo com a plataforma onde os programas compilados serão executados.

Na próxima página irei abordar como criar um Makefile para o exemplo dado. A partir do arquivo *Makefile* apresentado, você terá a base para compreender outros mais complexos.

Criando um Makefile para o exemplo apresentado

Na página anterior apresentei um exemplo de como utilizar a diretiva *include*. Num projeto grande, compilar o programa manualmente (ou seja, sem o recurso oferecido pelo *make*) seria realmente desastroso.

O make é um programa que realiza uma séries de operações armazenadas em um arquivo especial, que geralmente chamados de Makefile (este nome é reconhecido por padrão pelo make). Tais operações podem ser a compilação de uma série de arquivos fontes onde haja dependência entre eles. Com a estrutura oferecida pelo Makefile, fica fácil resolver todas as dependências.

Por exemplo, no exemplo da página anterior, a compilação do arquivo main.c exige primeiro a compilação do arquivo metodos.c, pois no arquivo main.c fazemos o uso de funções localizadas naquele arquivo (metodos.c). Usar o make para o exemplo da página anterior talvez não faça muita diferença, mas para projetos maiores as vantagens tornam-se evidente. O make permite ainda que o programa compile apenas os arquivos modificados caso após a primeira compilação fizéssemos alteração em algum arquivo e resolvêssemos recompilar o programa. Para criar o executável manualmente para o exemplo da página anterior, entraríamos na pasta criada (projetoC) e executaríamos os seguintes comandos na ordem em que aparecem:

$ gcc -c metodos.c
$ gcc main.c metodos.o -o exemplo

O resultado seria o arquivo executável exemplo. Para construir um Makefile para o exemplo dado, primeiro vamos entender um pouco da estrutura de um arquivo Makefile. Um arquivo Makefile é dividido em seções. Uma dessas seções são pré-definidas, a seção all (a primeira seção a ser executada). As outras são aquelas que a seção all depende direta ou indiretamente. Estas outras seções tem seus nomes definidas pelo criador do arquivo Makefile, no caso pode ser o próprio programador ou algum programa que gere automaticamente arquivos Makefile. Como ficaria a seção all:

all:[dependência1] [dependência2] ... [dependênciaN]
[TAB][COMANDO1]
[TAB][COMANDO2]
...
[TAB][COMANDON]
[dependência1]: [dependências de 1]
...
[dependência2]: [dependências de 2]
’’’

Onde:

> [TAB] representa uma tabulação (você pressiona a tecla TAB no início da da linha). As dependências são declaradas colocando o nome da dependência (ou regra, como também é chamada) seguido de dois pontos. Caso haja dependência para um seção, após os dois pontos os nomes das dependências devem ser colocados separados apenas por espaços, nenhum outro caractere deve ser colocado entre o nome das regras.

Cada comando numa dependência deve ser posto após uma tabulação logo abaixo da linha onde está o nome da seção. As dependências são também chamadas de regras. O exemplo abaixo mostra o Makefile para o exemplo de programa dado da página anterior (página 1). Este exemplo irá ilustrar as idéias abordadas até aqui.

Exemplo Makefile para o exemplo de programa dado na página anterior: dentro da pasta projetoC cria o seguinte arquivo (nomeado Makefile):

Arquivo *Makefile*:

```
all:METODOS
	gcc main.c metodos.o -o exemplo
METODOS:
	gcc -c metodos.c
```

Agora, dentro da pasta projetoC, execute o comando "make". O resultado será um arquivo compilado denominado exemplo. Para obter informações sobre o compilador *gcc*, no terminal execute o seguinte comando:

$ man gcc

Para mais informações sobre o comando make, consulte a página de manual do make também executando o comando man como segue:

$ man make

Essas duas últimas instruções só são válidas se as páginas de manual das ferramentas mencionadas estiverem instaladas em seu sistema *GNU/Linux*.

Conclusão

Para o exemplo dado, o exemplo de uso de arquivos *Makefile* é muito simples. No entanto, para projetos grandes e sérios, a complexidade é muito grande e torna-se necessário uma ferramenta (diga-se software ou programa) que auxilie na criação dos Makefiles. Num mesmo projeto podem haver vários Makefiles. Makefiles para projetos criados no *KDevelop* são criados automaticamente e à medida que você adiciona arquivos ao projeto. Mas vale à pena ter uma noção geral de como o negócio funciona.

Outra vantagem do uso do *make* que não foi mencionada nas páginas anteriores foi a seguinte: imagina um projeto imenso (um jogo por exemplo) em que o processo de compilação demora horas. Se modificações em um arquivo forçasse a compilação completa do programa, perderíamos muito tempo. Com o uso do make torna-se fácil compilar apenas o que foi modificado e suas dependências (ou regras), reduzindo bastante o tempo de compilação.

Só lembrando que C ainda é uma linguagem muito utilizada para a criação de programas de sistema, como compiladores. Apesar de não ser orientada à objetos, ainda é muito importante estudar a linguagem C (e C++ quando possível). Outras linguagens que eu uso e aconselho são:

C++;

C# (com Mono);

e Java.

IDES Para as linguagens mencionadas:

KDevelop (geralmente é instalada junto com o KDE): C, C++, Java e muitas outras;

Monodevelop: serve para qualquer linguagem suportada pelo Mono, como C#, Java e VB .NET. Site: http://www.monodevelop.com;

Eclipse: com um plugin adequado, qualquer linguagem pode ser usada nesta IDE. Site: http://www.eclipse.org;

X-Develop: suporta C#, Java, C++ e outras linguagens suportadas pelo Mono e/ou .NET. Site: http://www.x-develop.com.