MC202 — Estruturas de Dados
Tutorial: Compilando múltiplos arquivos em C

Mayk Choji
Instituto de Computação, Universidade Estadual de Campinas (UNICAMP)
mayk.choji@students.ic.unicamp.br

13/08/2015

Introdução

O objetivo deste tutorial é explicar o uso do compilador GNU C, `gcc`. Não é objetivo deste material ensinar a linguagem `C` nem explorar todas as opções de compilação, mas sim rever conceitos básicos e dar alguns exemplos de como compilar programas com mais de um arquivo de código-fonte.

Compilando um programa em C

Esta seção descreve as formas de se compilar programas em `C` utilizando o compilador `gcc`. Programas podem ser compilados de um único arquivo fonte ou de múltiplos arquivos, além de poderem usar bibliotecas de sistemas e arquivos de cabeçalho (*headers*).

Compilação refere-se ao processo de converter um programa em sua forma textual, ou *código-fonte*, escrito em uma linguagem como `C` ou `C++`, para *código de máquina*, que é armazenado em um arquivo conhecido como *arquivo executável* ou *arquivo binário*.

Compilando um programa simples

Considere o exemplo clássico do programa *'Hello World'* armazenado em um arquivo chamado `hello.c`.

```c
#include <stdio.h>

int main(void) {
        printf("Hello, world!\n");
        return 0;
}
```

Para compilar este arquivo utilizando o `gcc`, executa-se

```
$ gcc -std=c99 -pedantic -Wall -o hello hello.c
```

O comando acima compila o código em `hello.c` para código de máquina e o armazena em um arquivo executável chamado `hello`. O arquivo de saída é

especificado utilizando a opção `-o`. Se esta opção é omitida, o resultado é escrito em um arquivo padrão chamado `a.out`.
A opção `-Wall` habilita todos os *warnings* mais comumente utilizados. As opções `-std` e `-pedantic` definem o padrão da linguagem `C` utilizado durante a compilação. Durante os laboratórios desta disciplina, é recomendado compilar seus programas com estas opções para que o resultado local reflita o produzido pelo sistema *SuSy*.

Para executar o programa, digita-se o caminho do executável

```
$ ./hello
Hello, world!
```

Dividindo um programa em múltiplos arquivos

Conforme seu programa torna-se maior ou você começa a utilizar códigos de outras pessoas ou terceiros (i.e. bibliotecas), você terá que lidar com um código residindo em múltiplos arquivos. Por exemplo, se você desejar criar uma biblioteca com funções para poder utilizar em vários programas (e.g. *stdio.h, math.h etc.*), a compilação envolverá mais de um arquivo ao mesmo tempo.

Arquivos de cabeçalho — headers

Uma convenção comum em programas em `C` é escrever um *header* (arquivo com sufixo `.h`) para cada arquivo fonte (sufixo `.c`) que você *linka* ao código-fonte principal, ou seja, aquele que contém a função `main`. A ideia é que o arquivo `.c` contenha todo o código e o *header* contenha apenas os **protótipos** das funções, macros, definições de tipos e elementos similares.

Isto é feito para bibliotecas que são providas por terceiros, cuja implementação os autores desejam ocultar, fornecendo, portanto, somente o binário (arquivo compilado) e o *header* com as definições das funções.

Um ponto importante a ser lembrado é que, assim que olhamos para um arquivo de cabeçalho, temos toda a informação sobre quais funções estão definidas no arquivo-fonte e como elas são usadas—i.e. quais seus argumentos de entrada e quais seus argumentos de saída, se existirem. Se quisermos olhar "dentro" dessas funções, podemos procurar no arquivo-fonte. O *header* pode ser visto, então, como uma **interface** entre o código-fonte e o programador.

Exemplo

Nesta seção será apresentado um programa que faz uso de uma biblioteca própria para exibir mensagens na saída padrão e realizar algumas operações matemáticas básicas. Esta biblioteca por si só pode requerer outras bibliotecas locais ou padrões, que devem ser levadas em consideração no momento da compilação.

O programa principal contido no arquivo `operator.c` é dado como segue

```c
#include <stdio.h>
#include "calc.h"

int main(void) {
        greetings(NULL);
        printf("40 + 2 = %d\n", sum(40, 2));
        printf("2.2 ^ 2 = %lf\n", power(2.2, 2));
        greetings("Bye bye!");
        return 0;
}
```

Neste exemplo, incluímos duas bibliotecas:

stdio.h biblioteca padrão para entrada e saída de dados;

calc.h biblioteca local descrita a seguir.

Observe que os nomes das bibliotecas locais vão entre aspas. Neste caso, a biblioteca encontra-se no mesmo diretório que o arquivo principal. Caso a mesma estivesse em um subdiretório chamado `lib`, a inclusão se daria com `"lib/calc.h"`.

Aqui está o arquivo de cabeçalho da nossa biblioteca `calc.h`

```c
#include <stdio.h>
#include <math.h>

#ifndef M_PI
#define M_PI 3.14159265358979323846264338327795
#endif

void greetings(char *msg);

int sum(int a, int b);

double power(double x, int y);
```

Neste arquivo realizamos as chamadas para outras bibliotecas, definimos uma macro/constante (caso ela não esteja definida) e declaramos três protótipos de funções.

A implementação das funções é realizada no arquivo `calc.c`, o qual inclui, em sua primeira linha, o arquivo de cabeçalho previamente criado.

```c
#include "calc.h"

void greetings(char *msg) {
        if (msg == NULL) {
                printf("Welcome!\n");
        }
        else {
                printf("%s\n", msg);
        }
}

int sum(int a, int b) {
        return a + b;
}

double power(double x, int y) {
        return pow(x, y);
}
```

O programa final pode ser compilado executando-se o comando

```
gcc -std=c99 -pedantic -Wall -o operator operator.c calc.c -lm
```

Como pode ser visto, basta adicionar o arquivo `calc.c` na lista de arquivos a ser enviada para o compilador. Aqui é onde o compilador realmente integra o código contido em `calc.c` em nosso programa. Também precisamos da *flag* `-lm`, que instrui o compilador a carregar a biblioteca padrão *math* (uma vez que a diretiva `#include <math.h>` aparece em `calc.h`).

Conforme mencionado anteriormente, com o comando `gcc` descrito, o código de `calc.c` é integrado ao programa principal. Porém, isso nem sempre é verdade. Há casos em que as bibliotecas são carregadas dinamicamente—*shared libraries*—e o código de tais bibliotecas são inseridos uma única vez na memória e utilizados por todos os programas que as utilizam, mas não discutiremos esses casos neste tutorial.

Em resumo, a fim de incluirmos código externo em nosso programa em `C`, precisamos garantir duas condições:

1. O código-fonte `C` externo é passado para o compilador em tempo de compilação;

2. Nosso próprio programa em `C` possui acesso aos protótipos das funções associadas com o código externo.

Compilando arquivos independentemente

Se um programa é armazenado em um único arquivo, então qualquer mudança em uma dada função requer que o programa inteiro seja recompilado para produzir um novo executável. A recompilação de grandes arquivos-fonte pode levar muito tempo.

Quando programas são armazenados em arquivos-fonte independentes, somente os arquivos que sofreram alterações precisam ser recompilados. Nesta abordagem, os arquivos-fonte são compilados separadamente e então *linkados* para formar o programa de fato—um processo de duas etapas. Na primeira etapa, um arquivo é compilado sem criar um executável. O resultado é chamado de **arquivo objeto**, e tem extensão `.o` quando usado o `gcc`.
Na segunda etapa, os objetos são unidos por um programa chamado *linker*. O *linker* combina todos os objetos para criar um único executável.

Um arquivo objeto contém código de máquina onde toda referência a endereço de memória de funções ou variáveis em outros arquivos são deixadas indefinidas. Isto permite aos arquivos-fonte serem compilados sem referência direta uns com os outros. O *linker* preenche estas lacunas de endereços ao produzir o executável.

Criando objetos a partir de arquivos-fonte

A opção `-c` do `gcc` é usada para compilar um arquivo-fonte em um arquivo-objeto. Por exemplo, o seguinte comando compilará o arquivo-fonte `operator.c` para um objeto

```
gcc -std=c99 -pedantic -Wall -c operator.c
```

Isto produz o objeto `operator.o` contendo o código de máquina para a função `main`. Ele contém referências para as funções externas, porém os endereços de memória correspondentes são deixados indefinidos nesta etapa. Ao executar o comando anterior para o arquivo `calc.c`, é gerado o objeto `calc.o`.

Criando executáveis a partir de arquivos-objeto

O passo final para criar um executável é usar o `gcc` para conectar os objetos e preencher as lacunas de endereços de funções externas. Para isso, basta listar esses arquivos na linha de comando

```
gcc -o operator operator.o calc.o -lm
```

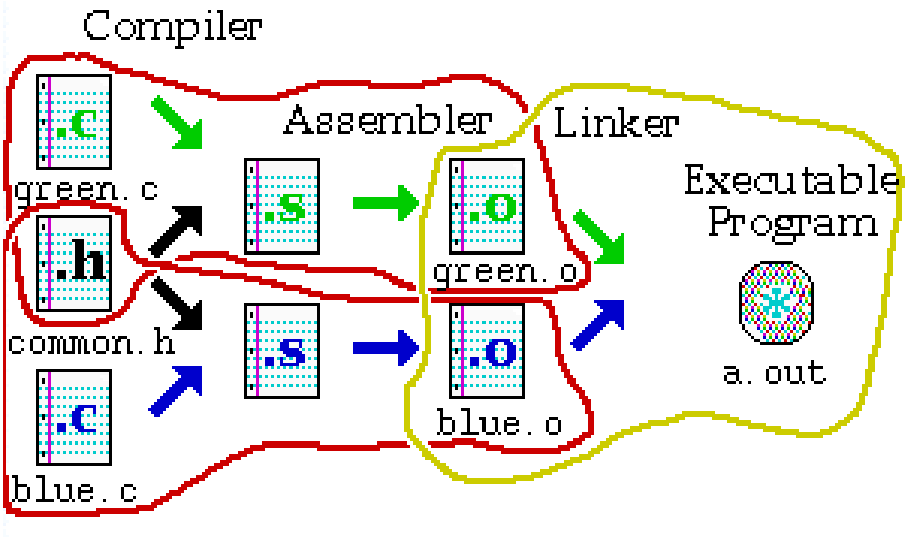
Esta é uma das poucas ocasiões em que não há necessidade de se utilizar a opção `-Wall`, uma vez que os arquivos-fonte individuais já foram compilados para códigos de objetos com sucesso.

Uma vez que o executável é gerado, podemos executar nosso programa

```
$ ./operator
Welcome!
40 + 2 = 42
2.2 ^ 2 = 4.840000
Bye bye!
```

A Figura 1 ilustra o processo descrito anteriormente.
Neste ponto, se um dos arquivos for alterado, basta recompilá-lo isoladamente e então executar o último passo (*link*) para gerar um novo executável. Em grandes projetos, esta abordagem pode economizar muito tempo.

Figura 1: Exemplo de compilação com múltiplos arquivos-fonte [2].



Conclusão

Neste tutorial, foi mostrado como compilar um programa composto por mais de um arquivo-fonte. Em geral, se o programa está dividido em vários arquivos mas estes ainda são pequenos e rápidos para se compilar, a opção mais simples é passar os nomes de todos os arquivos `.c` durante a compilação. Porém, caso algum arquivo seja muito complexo e leve mais tempo para compilar ou seja estável (sofra nenhuma ou poucas alterações), é aconselhável utilizar a segunda abordagem apresentada, ou seja, criar objetos separadamente e, em seguida, o executável.

Por fim, uma sugestão para trabalhar com projetos mais complexos é utilizar `Makefiles`, o que permite automatizar processos de compilação, limpeza de arquivos auxiliares etc.

Referências

[1] Brian Gough. *An Introduction to GCC. for the GNU compilers gcc and g++*. URL: http://www.network-theory.co.uk/docs/gccintro/gccintro_8.html.

[2] Dartmouth College. *Developing and Running Programs in A Linux Environment*. 19 de mar. de 2007. URL: https://www.dartmouth.edu/~rc/classes/softdev_linux/index.html.

[3] Paul Gribble. *Compiling, linking, Makefile, header files*. 2012. URL: http://gribblelab.org/CBootcamp/12_Compiling_linking_Makefile_header_files.html.