AULA 10

PROGRAMAS EXTENSOS

Nesta aula aprenderemos como dividir e distribuir nossas funções em vários arquivos. Esta possibilidade é uma característica importante da linguagem C, permitindo que o(a) programador(a) possa ter sua própria biblioteca de funções e possa usá-la em conjunto com um ou mais programas.

Os programas que escrevemos até o momento são bem simples e, por isso mesmo pequenos, com poucas linhas de código. No entanto, programas pequenos são uma exceção. À medida que os problemas têm maior complexidade, os programas para solucioná-los têm, em geral, proporcionalmente mais linhas de código. Por exemplo, a versão 2.6.25 do núcleo do sistema operacional LINUX, de abril de 2008, tem mais de nove milhões de linhas de código na linguagem C e seria impraticável mantê-las todas no mesmo arquivo[1]. Nesta aula veremos que um programa na linguagem C consiste de vários arquivos-fontes e também de alguns arquivos-cabeçalhos. aprenderemos a dividir nossos programas em múltiplos arquivos.

Esta aula é baseada na referência [7].

10.1 Arquivos-fontes

Até a última aula, sempre consideramos que um programa na linguagem C consiste de um único arquivo. Na verdade, um programa pode ser dividido em qualquer número de **arquivos-fontes** que, por convenção, têm a extensão `.c`. Cada arquivo-fonte contém uma parte do programa, em geral definições de funções e variáveis. Um dos arquivos-fontes de um programa deve necessariamente conter uma função `main`, que é o ponto de início do programa. Quando dividimos um programa em arquivos, faz sentido colocar funções relacionadas e variáveis em um mesmo arquivo-fonte. A divisão de um programa em arquivos-fontes múltiplos tem vantagens significativas:

- agrupar funções relacionadas e variáveis em um único arquivo ajuda a deixar clara a estrutura do programa;
- cada arquivo-fonte pode ser compilado separadamente, com uma grande economia de tempo se o programa é grande é é modificado muitas vezes;
- funções são mais facilmente re-usadas em outros programas quando agrupadas em arquivos-fontes separados.

[1] O sistema operacional LINUX, por ser livre e de código aberto, permite que você possa consultar seu código fonte, além de modificá-lo a seu gosto.

10.2 Arquivos-cabeçalhos

Quando abordamos, em aulas anteriores, a biblioteca padrão e o pré-processador da linguagem C, estudamos com algum detalhe os arquivos-cabeçalhos. Quando dividimos um programa em vários arquivos-fontes, como uma função definida em um arquivo pode chamar uma outra função definida em outro arquivo? Como dois arquivos podem compartilhar a definição de uma mesma macro ou a definição de um tipo? A diretiva `#include` nos ajuda a responder a essas perguntas, permitindo que essas informações possam ser compartilhadas entre arquivos-fontes.

Muitos programas grandes contêm definições de macros e definições de tipos que necessitam ser compartilhadas por vários arquivos-fontes. Essas definições devem ser mantidas em arquivos-cabeçalhos.

Por exemplo, suponha que estamos escrevendo um programa que usa macros com nomes `LOGIC`, `VERDADEIRO` e `FALSO`. Ao invés de repetir essas macros em cada arquivo-fonte do programa que necessita delas, faz mais sentido colocar as definições em um arquivo-cabeçalho com um nome como `logico.h` tendo as seguintes linhas:

```
#define VERDADEIRO 1
#define FALSO      0
#define LOGIC      int
```

Qualquer arquivo-fonte que necessite dessas definições deve conter simplesmente a linha a seguir:

```
#include "logico.h"
```

Definições de tipos também são comuns em arquivos-cabeçalhos. Por exemplo, ao invés de definir a macro `LOGIC` acima, podemos usar `typedef` para criar um tipo `logic`. Assim, o arquivo `logico.h` terá as seguintes linhas:

```
#define VERDADEIRO 1
#define FALSO      0
typedef logic int;
```

A figura 10.1 mostra um exemplo de dois arquivos-fontes que incluem o arquivo-cabeçalho `logico.h`.

Colocar definições de macros e tipos em um arquivo-cabeçalho tem algumas vantagens. Primeiro, economizamos tempo por não ter de copiar as definições nos arquivos-fontes onde são necessárias. Segundo, o programa torna-se muito mais fácil de modificar, já que a modificação da definição de uma macro ou de um tipo necessita ser feita em um único arquivo-cabeçalho. E terceiro, não temos de nos preocupar com inconsistências em conseqüência de arquivos-fontes contendo definições diferentes da mesma macro ou tipo.

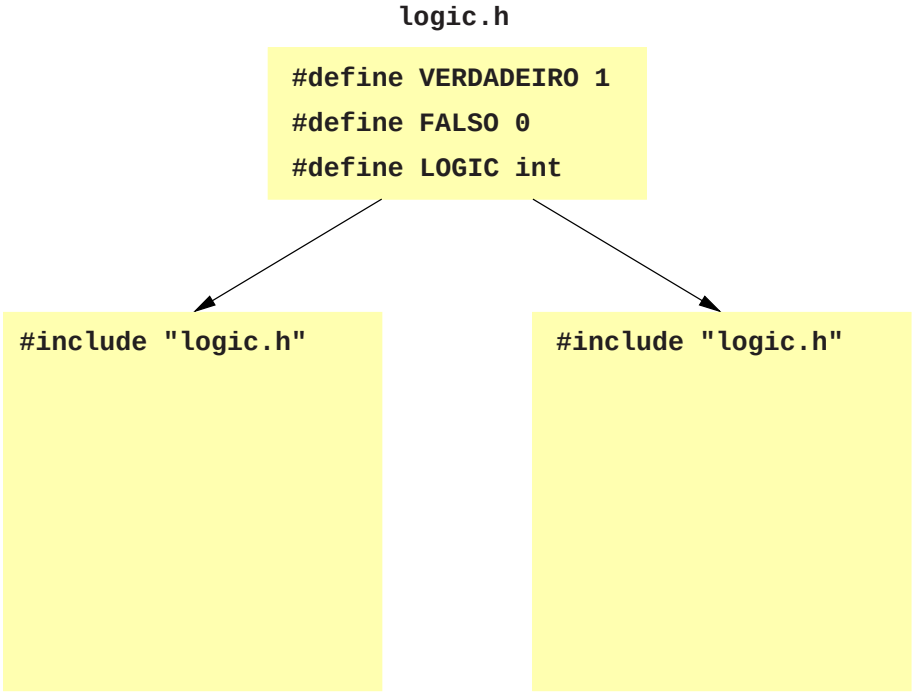

Figura 10.1: Inclusão de um arquivo-cabeçalho em dois arquivos-fontes.

Suponha agora que um arquivo-fonte contém uma chamada a uma função `soma` que está definida em um outro arquivo-fonte, com nome `calculos.c`. Chamar a função `soma` sem sua declaração pode ocasionar erros de execução. Quando chamamos uma função que está definida em outro arquivo, é sempre importante ter certeza que o compilador viu sua declaração, isto é, seu protótipo, antes dessa chamada.

Nosso primeiro impulso é declarar a função `soma` no arquivo-fonte onde ela foi chamada. Isso resolve o problema, mas pode criar imensos problemas de gerenciamento. Suponha que a função é chamada em cinqüenta arquivos-fontes diferentes. Como podemos assegurar que os protótipos das funções `soma` são idênticos em todos esses arquivos? Como podemos garantir que todos esses protótipos correspondem à definição de `soma` em `calculos.c`? Se `soma` deve ser modificada posteriormente, como podemos encontrar todos os arquivos-fontes onde ela é usada? A solução é evidente: coloque o protótipo da função `soma` em um arquivo-cabeçalho e inclua então esse arquivo-cabeçalho em todos os lugares onde `soma` é chamada. Como `soma` é definida em `calculos.c`, um nome natural para esse arquivo-cabeçalho é `calculos.h`. Além de incluir `calculos.h` nos arquivos-fontes onde `soma` é chamada, precisamos incluí-lo em `calculos.c` também, permitindo que o compilador verifique que o protótipo de `soma` em `calculos.h` corresponde à sua definição em `calculos.c`. É regra sempre incluir o arquivo-cabeçalho que declara uma função em um arquivo-fonte que contém a definição dessa função. Não fazê-lo pode ocasionar erros difíceis de encontrar. Se `calculos.c` contém outras funções, muitas delas devem ser declaradas no mesmo arquivo-cabeçalho onde foi declarada a função `soma`. Mesmo porque, as outras funções em `calculos.c` são de alguma forma relacionadas com `soma`. Ou seja, qualquer arquivo que contenha uma chamada à `soma` provavelmente necessita de alguma das outras funções em `calculos.c`. Funções cuja intenção seja usá-las apenas como suporte dentro de `calculos.c` não devem ser declaradas em um arquivo-cabeçalho.

Para ilustrar o uso de protótipos de funções em arquivos-cabeçalhos, vamos supor que queremos manter diversas definições de funções relacionadas a cálculos geométricos em um arquivo-fonte `geometricas.c`. As funções são as seguintes:

```
double perimetroQuadrado(double lado)
{
   return 4 * lado;
}

double perimetroTriangulo(double lado1, double lado2, double lado3)
{
   return lado1 + lado2 + lado3;
}

double perimetroCirculo(double raio)
{
   return 2 * PI * raio;
}

double areaQuadrado(double lado)
{
   return lado * lado;
}

double areaTriangulo(double base, double altura)
{
   return base * altura / 2;
}

double areaCirculo(double raio)
{
   return PI * raio * raio;
}

double volumeCubo(double lado)
{
   return lado * lado * lado;
}

double volumeTetraedro(double lado, double altura)
{
   return (double)1/3 * areaTriangulo(lado, lado * sqrt(altura) / 2) * altura;
}

double volumeEsfera(double raio)
{
   return (double)4/3 * PI * raio * raio;
}
```

Os protótipos dessas funções, além de uma definição de uma macro, serão mantidos em um arquivo-cabeçalho com nome `geometricas.h`:

```
double perimetroQuadrado(double lado);
double perimetroCirculo(double raio);
double perimetroTriangulo(double lado1, double lado2, double lado3);
double areaQuadrado(double lado);
double areaCirculo(double raio);
double areaTriangulo(double base, double altura);
double volumeCubo(double lado);
double volumeEsfera(double raio);
double volumeTetraedro(double lado, double altura);
```

Além desses protótipos, a macro `PI` também deve ser definida neste arquivo. Então, um arquivo-fonte `calc.c` que calcula medidas de figuras geométricas e que contém a função `main` pode ser construído. A figura 10.2 ilustra essa divisão.

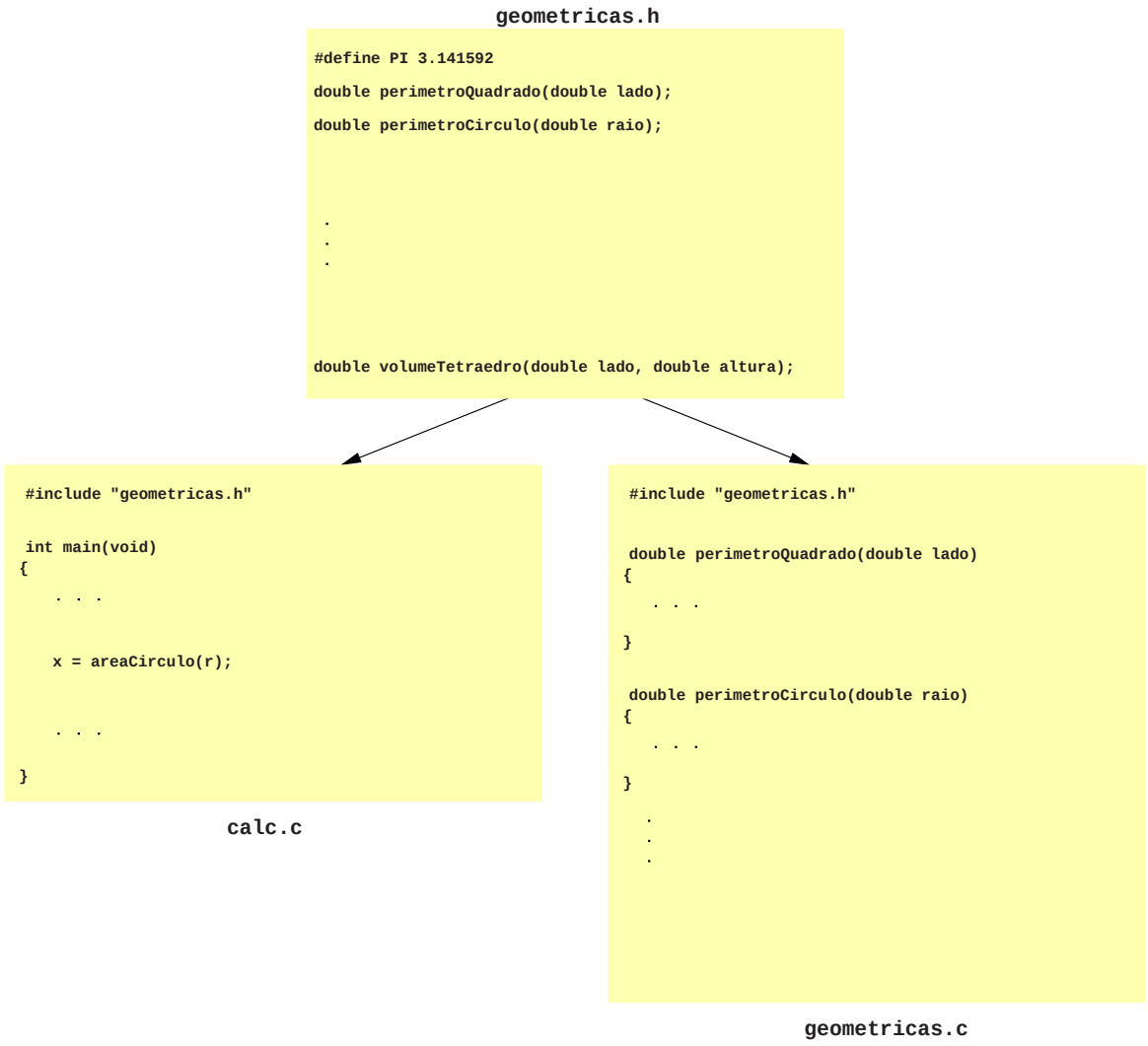

Figura 10.2: Relação entre protótipos, arquivo-fonte e arquivo-cabeçalho.

10.3 Divisão de programas em arquivos

Como vimos na seção anterior, podemos dividir nossos programas em diversos arquivos que possuem uma conexão lógica entre si. Usaremos agora o que conhecemos sobre arquivos-cabeçalhos e arquivos-fontes para descrever uma técnica simples de dividir um programa em arquivos. Consideramos aqui que o programa já foi projetado, isto é, já decidimos quais funções o programa necessita e como arranjá-las em grupos logicamente relacionados.

Cada conjunto de funções relacionadas é sempre colocado em um arquivo-fonte em separado. `geometricas.c` da seção anterior é um exemplo de arquivo-fonte desse tipo. Além disso, criamos um arquivo-cabeçalho com o mesmo nome do arquivo-fonte, mas com extensão `.h`. O arquivo-cabeçalho `geometricas.h` da seção anterior é um outro exemplo. Nesse arquivo-cabeçalho, colocamos os protótipos das funções incluídas no arquivo-fonte, lembrando que as funções que são projetadas somente para dar suporte às funções do arquivo-fonte não devem ter seus protótipos descritos no arquivo-cabeçalho. Então, devemos incluir o arquivo-cabeçalho em cada arquivo-fonte que necessite chamar uma função definida no arquivo-fonte. Além disso, incluímos o arquivo-cabeçalho no próprio arquivo-fonte para que o compilador possa verificar que os protótipos das funções no arquivo-cabeçalho são consistentes com as definições do arquivo-fonte. Mais uma vez, esse é o caso do exemplo da seção anterior, com arquivo-fonte `geometricas.c` e arquivo-cabeçalho `geometricas.h`. Veja novamente a figura 10.2.

A função principal `main` deve ser incluída em um arquivo-fonte que tem um nome representando o nome do programa. Por exemplo, se queremos que o programa seja conhecido como `calc` então a função `main` deve estar em um arquivo-fonte com nome `calc.c`. É possível que existam outras funções no mesmo arquivo-fonte onde `main` se encontra, funções essas que não são chamadas em outros arquivos-fontes do programa.

Para gerar um arquivo-executável de um programa dividido em múltiplos arquivos-fontes, os mesmos passos básicos que usamos para um programa em um único arquivo-fonte são necessários:

- **compilação:** cada arquivo-fonte do programa deve ser compilado separadamente; para cada arquivo-fonte, o compilador gera um arquivo contendo código objeto, que têm extensão `.o`;
- **ligação:** o ligador combina os arquivos-objetos criados na fase compilação, juntamente com código das funções da biblioteca, para produzir um arquivo-executável.

Muitos compiladores nos permitem construir um programa em um único passo. Com o compilador GCC, usamos o seguinte comando para construir o programa `calc` da seção anterior:

```
gcc -o calc calc.c geometricas.c
```

Os dois arquivos-fontes são primeiro compilados em arquivos-objetos. Esse arquivos-objetos são automaticamente passados para o ligador que os combina em um único arquivo. A opção `-o` especifica que queremos que nosso arquivo-executável tenha o nome `calc`.

Digitar os nomes de todos os arquivos-fontes na linha de comando de uma janela de um terminal logo torna-se uma tarefa tediosa. Além disso, podemos desperdiçar uma quantidade de tempo quando reconstruímos um programa se sempre recompilamos todos os arquivos-fontes, não apenas aqueles que são afetados pelas nossas modificações mais recentes.

Para facilitar a construção de grandes programas, o conceito de *makefiles* foi proposto nos primórdios da criação do sistema operacional UNIX. Um *makefile* é um arquivo que contém informação necessária para construir um programa. Um *makefile* não apenas lista os arquivos que fazem parte do programa, mas também descreve as dependências entre os arquivos. Por exemplo, da seção anterior, como `calc.c` inclui o arquivo `geometricas.h`, dizemos que `calc.c` 'depende' de `geometricas.h`, já que uma mudança em `geometricas.h` fará com que seja necessária a recompilação de `calc.c`.

Abaixo, listamos um *makefile* para o programa `calc`.

```
calc: calc.o geometricas.o
            gcc -o calc calc.o geometricas.o -lm
calc.o: calc.c geometricas.h
            gcc -c calc.c -lm
geometricas.o: geometricas.c geometricas.h
            gcc -c geometricas.c -lm
```

No arquivo acima, existem 3 grupos de linhas. Cada grupo é conhecido como um **regra**. A primeira linha em cada regra fornece um arquivo-**alvo**, seguido pelos arquivos dos quais ele depende. A segunda linha é um **comando** a ser executado se o alvo deve ser reconstruído devido a uma alteração em um de seus arquivos de dependência.

Na primeira regra, `calc` é o alvo:

```
calc: calc.o geometricas.o
            gcc -o calc calc.o geometricas.o -lm
```

A primeira linha dessa regra estabelece que `calc` depende dos arquivos `calc.o` e `geometricas.o`. Se qualquer um desses dois arquivos foi modificado desde da última construção do programa, então `calc` precisa ser reconstruído. O comando na próxima linha indica como a reconstrução deve ser feita, usando o GCC para ligar os dois arquivos-objetos.

Na segunda regra, `calc.o` é o alvo:

```
calc.o: calc.c geometricas.h
            gcc -c calc.c -lm
```

A primeira linha indica que `calc.o` necessita ser reconstruído se ocorrer uma alteração em `calc.c` ou `geometricas.h`. A próxima linha mostra como atualizar `calc.o` através da recompilação de `calc.c`. A opção `-c` informa o compilador para compilar `calc.c` em um arquivo-objeto, sem ligá-lo.

Tendo criado um *makefile* para um programa, podemos usar o utilitário **make** para construir ou reconstruir o programa. verificando a data e a hora associada com cada arquivo do programa, **make** determina quais arquivos estão desatualizados. Então, ele invoca os comandos necessários para reconstruir o programa.

Algumas dicas para criar *makefiles* seguem abaixo:

- cada comando em um *makefile* deve ser precedido por um caractere de tabulação horizontal **TAB**;
- um *makefile* é armazenado em um arquivo com nome **Makefile**; quando o utilitário **make** é usado, ele automaticamente verifica o conteúdo do diretório atual buscando por esse arquivo;
- use

  ```
  make alvo
  ```

  onde ***alvo*** é um dos alvos listados no *makefile*; se nenhum alvo é especificado, **make** construirá o alvo da primeira regra.

O utilitário **make** é complicado o suficiente para existirem dezenas de livros e manuais que nos ensinam a usá-lo. Com as informações desta aula, temos as informações básicas necessárias para usá-lo na construção de programas extensos divididos em diversos arquivos. Mais informações sobre o utilitário **make** devem ser buscadas no manual do GNU/Make.

Exercícios

10.1 (a) Escreva uma função com a seguinte interface:

```
void preenche_aleatorio(int n, int v[MAX])
```

que receba um número inteiro n, com $0 < 0 \leqslant 100$, e um vetor v de números inteiros e gere n números inteiros aleatórios armazenando-os em v. Use a função **rand** da biblioteca **stdlib**.

(b) Crie um arquivo-fonte com todas as funções de ordenação que vimos nas aulas 5, 6, 7 e 8. Crie também um arquivo-cabeçalho correspondente.

(c) Escreva um programa que receba um inteiro n, com $1 \leqslant n \leqslant 10000$, gere um seqüência de n números aleatórios e execute os métodos de ordenação que conhecemos sobre esse vetor, medindo seu tempo de execução. Use as funções **clock** e **difftime** da biblioteca **time**.

(d) Crie um *makefile* para compilar e ligar seu programa.