Programação Estruturada

Aula 19 - Escrevendo Programas Maiores

Yuri Malheiros (yuri@ci.ufpb.br)

Introdução

- Alguns programas cabem em apenas um arquivo, mas muitos não
- A maioria dos programas são escritos em múltiplos arquivos

Arquivos Fonte

- Os arquivos `.c` são os arquivos fonte
- Podemos ter vários no nosso programa
- Eles contêm principalmente definições de funções e variáveis
- Um dos arquivos fonte precisa ter uma função `main` que é o início do programa

Arquivos Fonte

- Por exemplo, suponha que queremos escrever um programa que calcula expressões matemáticas usando a Notação Polonesa Inversa
- Nela, os operandos aparecem antes do operador
    - Exemplo: `30 5 - 7 *`

Arquivos Fonte

- Nós precisamos das seguintes operações:
- Ler um "token"
- Se o token for um número, colocá-lo em uma pilha
- Se o token for um operador, retirar dois operandos, realizar o cálculo e colocar o resultado na pilha

Arquivos Fonte

- Nesse caso, poderíamos separar nosso programa em três arquivos:
- `token.c` - que tem as funções de leitura dos tokens
- `stack.c` - que tem a implementação da pilha
- `calc.c` - que tem a função `main`

Arquivos Fonte

- Dividir um programa em vários arquivos tem diversas vantagens:
- Agrupar funções e variáveis relacionados em um arquivo deixa mais clara a estrutura do programa
- Cada arquivo fonte pode ser compilado separadamente - isso pode fazer muito diferença em um programa muito grande
- Facilitar o reuso de código

Cabeçalhos

- Como uma função de um arquivo chama uma função definida em outro arquivo?
- Como uma função pode acessar uma variável declarada em outro arquivo?
- Como dois arquivos podem compartilhar uma definição de macro?

Cabeçalhos

- A diretiva `#include` possibilita o compartilhamento dessas informações
- O `#include` inclui o conteúdo de um arquivo em outro
- Se quisermos que vários arquivos compartilhem a mesma informação, então nós incluimos essa informação de um arquivo no outro
- Os arquivos incluídos são chamados de cabeçalho e tem a extensão `.h`

Cabeçalhos

- O `#include` tem duas formas:
- `#include <arquivo>` - usado para arquivos da biblioteca do C
- `#include "arquivo"` - usado para os outros arquivos de cabeçalho

Compartilhando definições de macros e tipos

- Vamos definir duas macros: `TRUE` e `FALSE`, e um tipo: `Bool`

```
#define TRUE 1
#define FALSE 0
typedef int Bool;
```

- Para que eles possam ser reusados, vamos colocá-los no arquivo `boolean.h`

Compartilhando definições de macros

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

int main(void) {

    if (TRUE) {
        printf("verdadeiro\n");
    }

    return 0;
}
```

Compartilhando definições de macros e tipos

- Colocar definições de macros e tipos em um cabeçalho possui vantagens:
  - Não precisamos copiar as definições em todos os arquivos fonte
  - Torna o programa mais fácil de modificar - para mudar uma definição, basta alterar o cabeçalho

Compartilhando protótipos de funções

- Como fazer para reusar uma função definida em outro arquivo?
- Para utilizar uma função, o programa precisa ter visto no mínimo o seu protótipo
- Podemos colocar o protótipo da função em um arquivo de cabeçalho e incluir ele nos arquivos fonte que queremos usar essa função

Compartilhando protótipos de funções

- E a definição da função? Ela fica em um arquivo fonte
- Se ela for definida num arquivo chamado `abc.c`, então o cabeçalho com o seu protótipo deve se chamar `abc.h`
- Também precisamos incluir `abc.h` em `abc.c` para que o compilador saiba que o cabeçalho casa com a implementação do arquivo fonte

Compartilhando protótipos de funções

- No exemplo da calculadora, nós teríamos um arquivo `stack.c` com as implementações das funções para tratar a pilha e `stack.h` sendo o cabeçalho com os protótipos das funções

Compartilhando protótipos de funções

- `stack.h`

```
void limpar(void);
int esta_vazia(void);
int esta_cheia(void);
void push(int i);
int pop(void);
```

Compartilhando protótipos de funções

- `calc.c`

```c
#include "stack.h"

int main(void) {
  limpar();
}
```

Compartilhando protótipos de funções

- `stack.c`

```c
#include "stack.h"

int conteudo[100];
int topo = 0;

void limpar(void) { ... }

int esta_vazia(void) { ... }

int esta_cheia(void) { ... }

void push(int i) { ... }

int pop(void) { ... }
```

Includes aninhados

- Um arquivo de cabeçalho pode conter diretivas `#include`
- Por exemplo, o arquivo `stack.h` tem os protótipos:
  - `int esta_vazia(void);`
  - `int esta_cheia(void);`
- Poderíamos usar uma definição de tipo como no `boolean.h` nesses protótipos:
  - `Bool esta_vazia(void);`
  - `Bool esta_cheia(void);`
- Para ter acesso ao tipo `Bool` temos que incluir `boolean.h` em `stack.h`

Protegendo um cabeçalho

- Se um código importar um cabeçalho mais de uma vez, isso pode causar problemas
- Por exemplo, se
  - arquivo1.h inclui arquivo3.h
  - arquivo2.h inclui arquivo3.h
  - prog.c inclui arquivo1.h e arquivo2.h
- arquivo3.h vai ser compilado duas vezes

Protegendo um cabeçalho

- Para nossa segurança, é uma boa ideia proteger todos os cabeçalhos para que eles não sejam incluídos mais de uma vez
- Para proteger um cabeçalho, temos:

```
#ifndef BOOLEAN_H
#define BOOLEAN_H

// código aqui

#endif
```

- Ao ser incluido pela primeira vez, `BOOLEAN_H` não existe, então as linhas entre `#ifndef` e `#endif` permanecem
- Ao ser incluído uma segunda vez, `BOOLEAN_H` vai existir e as linhas entre `#ifndef` e `#endif` são ignoradas

Dividindo um programa em arquivos

- Vamos separar funções relacionadas em um mesmo arquivo
- Cada conjunto de funções terá seu arquivo fonte `abc.c`
- Também precisamos criar um cabeçalho com o mesmo nome `abc.h`
- No `abc.h` colocaremos os protótipos das funções definidas em `abc.c`
- Funções que são usadas apenas em `abc.c` não precisam ser colocadas em `abc.h`
- Incluiremos `abc.h` em todos os arquivos que precisem das funções definidas em `abc.c`
- A função `main` ficará num arquivo com o nome do programa final

Exemplo

- Vamos implementar uma calculadora com a Notação Polonesa Inversa...

Compilando um programa com múltiplos arquivos

- A construção de um programa envolve duas etapas
  - Compilação: os arquivos fonte são compilados separadamente gerando um arquivo objeto
  - Linking: combina os arquivos objeto criados no passo anterior para produzir um executável
- `gcc calc.c stack.c token.c -o calc`

Makefile

- Colocando todos os arquivos fonte na linha de comando para compilação, nós recompilamos sempre todos os arquivos
- Podemos compilar apenas o que foi alterado
- Para gerenciar esse processo e facilitar a construção de programas grandes, usaremos o Makefile

Makefile

- O Makefile contém a lista dos arquivos do programa e especifica suas dependências

```
calc: calc.o stack.o token.o
        gcc -o calc calc.o stack.o token.o

calc.o: calc.c stack.h token.h
        gcc -c calc.c

stack.o: stack.c stack.h
        gcc -c stack.c

token.o: token.c token.h
        gcc -c token.c
```

Makefile

- No exemplo existem 4 grupos, cada um conhecido como uma regra

```
token.o: token.c token.h
        gcc -c token.c
```

- A primeira linha especifica o arquivo alvo e suas dependências
- Se uma das dependências for alterada, então o arquivo precisa ser reconstruído
- A segunda linha tem o comando que deve ser executado para reconstruir o arquivo alvo

Makefile

- O código do Makefile deve ser colocado em arquivo chamado `Makefile`
- Para executá-lo, usamos o comando `make alvo`, onde `alvo` é um dos alvos especificados no `Makefile`
- Podemos usar simplesmente `make`, nesse caso, o alvo será o da primeira regra