MC-102 — Aula 13
Funções e Procedimentos II

Instituto de Computação – Unicamp

Segundo Semestre de 2009

Roteiro

Declarando funções depois do `main`

Até o momento, aprendemos que devemos declarar as funções antes do programa principal, mas o que ocorreria se declarassemos depois?

Veja os exemplos em `depois.c` e `depois2.c`. Para deixar mais aparente os problemas, compile com a opção `-Wall`.

Declarando uma função sem defini-la

- Para organizar melhor um programa ou para escrever um programa em vários arquivos podemos declarar uma função sem implementá-la ou defini-la (protótipo).
- Para declarar uma função sem a sua implementação. Substituímos as chaves e seu conteúdo por ponto-e-virgula.

```
tipo nome (tipo parâmetro1, tipo parâmetro2, ...,
           tipo parâmetroN);
```

- A declaração de uma função deve vir sempre antes do seu uso. A sua definição pode aparecer em qualquer lugar do programa.

Arquivos-Cabeçalhos ("header", .h)

Diretiva #include

- Inclui arquivos no ponto de chamada
- Texto do arquivo é "expandido" no ponto da chamada
- Sistema operacional é responsável por "encontrar o arquivo"

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

"Nossos" Arquivos-Cabeçalhos

- Arquivos com extensão ".h"
- Atenção: #include "arquivo.h"
- Mas o que são arquivos-cabeçalhos? arquivos que contém protótipos de funções
- Para que servem? para aproveitar uma função em vários programas futuros!

Exemplo:
ePar.c
soma2.c
multiplica2.c
funcoesCabecalho.h
progPrincipal.c

Usando "Nossos" Arquivos-Cabeçalhos

ePar.c
soma2.c
multiplica2.c
funcoesCabecalho.h
progPrincipal.c

```
gcc progPrincipal.c soma2.c multiplica2.c ePar.c -o saida
```

ou

```
gcc -c progPrincipal.c soma2.c multiplica2.c ePar.c
gcc progPrincipal.o soma2.o multiplica2.o ePar.o -o saida
```

Passagem de argumentos por valor

- Quando passamos argumentos a uma função, os valores fornecidos são copiados para os parâmetros formais da função. Este processo é idêntico a uma atribuição.
- Desta forma, alterações nos parâmetros dentro da função não alteram os valores que foram passados:

```
void nao_troca(int x, int y) {
  int aux;
  aux = x;
  x = y;
  y = aux;
}
```

Veja o exemplo em `nao_troca.c`.

Passagem de argumentos por referência

- Existe uma forma de alterarmos a variável passada como argumento, ao invés de usarmos apenas o seu valor.
- O artifício corresponde a passarmos como argumento o endereço da variável, e não o seu valor.
- Para indicarmos que será passado o endereço do argumento, usamos o mesmo tipo que usamos para declarar um variável que guarda um endereço:

```
tipo nome (tipo *parâmetro1, tipo *parâmetro2, ...,
           tipo *parâmetroN) {
      comandos;
}
```

Passagem de argumentos por referência

- Um endereço de variável passado com parâmetro não é muito útil. Para acessarmos o valor de uma variável apontada por um endereço, usamos o operador *:
- Ao precedermos uma variável que contém um endereço com este operador, obtemos o equivalente a variável armazenada no endereço em questão:

```
void troca(int *end_x, int *end_y) {
  int aux;
  aux = *end_x;
  *end_x = *end_y;
  *end_y = aux;
}
```

Veja o exemplo em `troca.c`.

Passagem de argumentos por referência

- Uma outra forma de conseguirmos alterar valores de variáveis externas a funções é usando variáveis globais.
- Nesta abordagem usamos variáveis globais no lugar de parâmetros e de valores de retorno.
- Porém, ao usar esta técnica estamos negando uma das principais vantagens de se usar funções, reaproveitamento de código.

Veja um exemplo em `vetor_global.c`.

Registros em funções

- Registros podem ser passados como parâmetros de uma função, como qualquer outro tipo.
- O registro deve ser declarado antes da função.
- O parâmetro formal recebe uma cópia do registro, da mesma forma que em uma atribuição envolvendo registros.
- Uma função pode retornar um registro, que é novamente copiado como resultado da expressão.

Veja o exemplo em `registro.c`.

Vetores em funções

- Ao contrário dos outros tipos e registros, vetores têm um comportamento diferente quando usados como parâmetros ou valores de retorno de funções.
- Por padrão, ao se indicar o tipo de um vetor, este sempre é interpretado pelo compilador como o endereço do primeiro elemento do vetor.
- Desta forma, sem precisarmos usar uma notação especial, os vetores são sempre passados por referência.
  Veja exemplos em `vetor_parametro.c` e `vetor_vs_variavel.c`.

Vetores em funções

- Devemos ficar atentos às implicações do fato dos vetores serem sempre passados por referência.
- Ao passar um vetor como parâmetro, se ele for alterado dentro da função, as alterações ocorrerão no próprio vetor e não em uma cópia.
- Ao retornar um vetor como valor de retorno, não é feita uma cópia deste vetor como no caso dos registros. Assim, o vetor "retornado" pode desaparecer se ele foi declarado no corpo da função.

Vetores multi-dimensionais e funções

- Ao passar um vetor como parâmetro não é necessário fornecer o seu tamanho na declaração da função. Porém, é importante lembrar que o vetor tem um tamanho que deve ser considerado, como no exemplo em `vetor_parametro.c`.
- Quando o vetor é multi-dimensional a possibilidade de não informar o tamanho na declaração se restringe a primeira dimensão apenas.

```
void mostra_matriz(int mat[][10], int n_linhas) {
  ...
}
```

Veja o exemplo em `matriz.c`.

Estruturas de dados complexas

- Observando as regras apresentadas até o momento, é possível combinar todas as redefinições de tipos e estruturas de dados estudadas até agora em funções.
- O exemplo do arquivo `vetor_registro.c` mostra como usar funções para simplificar a tarefa de iniciar e imprimir um vetor de registros.

Exercício
Achar o valor máximo em uma matriz (teste2)

Reestruture o programa do teste 2 utilizando funções de maneira que:

- As operações de leitura, escrita e verificação do maior valor da matriz fiquem em funções separadas.
- Lembre-se:
  - Assuma que o tamanho máximo da matriz é 100X100
  - O usuário deve informar quantas posições (linhas e colunas) deseja utilizar