MC102 - Algoritmos e programação de computadores

Aula 14: Funções e Procedimentos

Procedimentos

Procedimentos são estruturas que agrupam um conjunto de comandos, que são executados quando o procedimento é chamado.

```
scanf("%d", &x);
```

Funções

Funções são procedimentos que retornam um único valor ao final de sua execução.

```
x = sqrt(4);
```

```
if (scanf ("%d", &x) == EOF)
   printf("Fim de arquivo.\n");
```

Porque utilizar funções?

- Evitar que os blocos do programa fiquem grandes demais e, por conseqüência, mais difíceis de entender;
- Facilitar a leitura do programa-fonte;
- Separar o programa em partes(blocos) que possam ser logicamente compreendidos de forma isolada.

Porque utilizar funções?

- Permitir o reaproveitamento de código já construído (por você ou por outros programadores);
- Evitar que um trecho de código seja repetido várias vezes dentro de um mesmo programa;
- Permitir a alteração de um trecho de código de uma forma mais rápida. Com o uso de uma função é preciso alterar apenas dentro da função que se deseja;

Declarando uma função

Uma função possui o seguinte formato:

tipo `nome_da_funcao` (tipo $< parametro_1 >$,
tipo $< parametro_2 >$, ..., tipo $< parametro_n >$) {

```
    comandos;
    return( valor de retorno );
}
```

Declarando uma função

- Toda função deve ter um tipo. Esse tipo determina qual será o tipo de seu valor de retorno.
- Os parâmetros de uma função determinam qual será o seu comportamento (como se fosse uma função matemática, onde o parâmetro determina o valor da função).

Declarando uma função

- Uma função pode não ter parâmetros, basta não informá-los.
- A expressão contida dentro do comando `return` é chamado de valor de retorno, e corresponde a resposta de uma determinada função. Esse comando é sempre o último a ser executado por uma função, e nada após ele será executado.
- As funções devem ser declaradas fora do programa principal (`main()`).

Exemplo de função

A função abaixo soma dois valores, passados como parâmetro

```
int soma (int a, int b) {
  return (a + b);
}
```

Invocando uma função

Uma forma clássica de realizarmos a invocação (ou chamada) de uma função é atribuindo o seu valor a uma variável

```
variavel = funcao (parametros);
```

Na verdade, podemos invocar uma função em qualquer lugar onde faríamos a leitura de uma variável, mas nunca a escrita. Ex:

```
printf ("Soma de a e b: %d\n", soma(a, b));
```

Veja mais detalhes em `soma.c`

Invocando uma função

- Aqui, as variáveis passadas como parâmetros indicam quais são os valores com os quais a função irá trabalhar. Esses valores são copiados para os parâmetros da função, que pode manipulá-los.
- Os parâmetros passados pela função não necessariamente possuem os mesmos nomes que os parâmetros que a função espera.
- Esses parâmetros serão mantidos intocados durante a execução da função.

Veja mais detalhes em `parametros.c`

O tipo `void`

O tipo `void` é um tipo especial, utilizado principalmente em funções. Ele é um tipo que representa o "nada", ou seja, uma variável desse tipo armazena conteúdo indeterminado, e uma função desse tipo retorna um conteúdo indeterminado.

Procedimentos em C

Procedimentos em linguagem C nada mais são que funções do tipo `void`. Por exemplo, o procedimento abaixo imprime o número que for passado para ele como parâmetro.

```
void imprime (int numero) {
  printf ("Número %d\n", numero);
}
```

Podemos ignorar o valor de retorno de uma função e, para esta chamada, ela será equivalente a um procedimento.

Invocando um procedimento

Para invocarmos um procedimento, devemos utilizá-lo como utilizaríamos qualquer outro comando, ou seja:

```
procedimento (parametros);
```

Veja mais detalhes em `imprime.c`

A função `main`

- O chamado programa principal na verdade é uma função também, ele apenas é uma função especial, que é invocada automaticamente pelo programa quando esse inicia sua execução e possui um tipo fixo (`int`).
- Quando utilizado, o comando `return` informa ao sistema operacional se o programa funcionou corretamente ou não. O padrão é que um programa retorne zero caso tenha funcionado corretamente ou qualquer outro valor caso contrário.

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?

Mais detalhes em `depois.c` e `depois2.c`.

Compile com -Wall.

Declarando funções depois do `main`

- Embora aparentemente funcione, isso não pode ser feito na prática, pois alguns compiladores simplesmente não aceitam essa sintaxe, obrigando a declaração antes da função `main` (ou melhor, antes de qualquer outra função que utilize uma determinada função).
- A saída aqui é fazer uma declaração da função antes do main, mas colocar a sua implementação depois dele. Aguarde cenas do próximo slide...

Declarando uma função

Declarar uma função sem a sua implementação é muito semelhante a declará-la com a implementação. Substituímos as chaves e seu conteúdo por ponto-e-virgula, ou seja:

tipo `nome_da_funcao` (tipo $< parametro_1 >$,
tipo $< parametro_2 >$, . . . , tipo $< parametro_n >$) ;

No exemplo abaixo, somente declaramos a função imprime, sem implementá-la.

```
void imprime (int numero);
```

Declarando funções depois do `main`

- Com isso, separamos a declaração da implementação, permitindo que ela possa vir em qualquer lugar do código (antes ou depois de qualquer outra função).
- Além disso, um programa que declara todas as funções antes de usá-las tende a ser um programa mais claro, pois o programador já sabe qual o conjunto de funções que ele pode usar, sem se preocupar com a forma como elas foram implementadas (ou sequer como elas foram, caso o código esteja sendo desenvolvido por uma equipe).

Veja exemplo em `depois_corrigido.c`

Variáveis locais e variáveis globais

- Uma variável é chamada local se ela foi declarada dentro de uma função. Nesse caso, ela existe somente dentro daquela função e após o término da execução da mesma, a variável deixa de existir.
- Uma variável é chamada global se ela for declarada fora de qualquer função (ou seja, no mesmo lugar onde registros, tipos enumerados e procedimentos são declarados). Essa variável existe dentro de todas as funções e qualquer procedimento ou função pode alterá-la.

Variáveis globais

```
#include <stdio.h>
int variavel_global;
int main () {
  variavel_global = 0;
  printf ("%d", variavel_global);
}
```

Veja outro exemplo em `global2.c`

Escopo de variáveis

- Porém, agora podemos ter variáveis com o mesmo nome que, teoricamente, seriam válidas em um mesmo trecho de código. Imagine uma variável declarada globalmente para a qual exista uma outra variável, com o mesmo nome, declarada dentro de uma função. Qual das duas variáveis será lida ou escrita ?

Veja mais detalhes em `escopo.c`

Registros

- registros podem ser passados como parâmetro.
  - o registro deve ser declarado antes da função;
  - uma cópia do registro é feita antes da chamada.
- uma função pode retornar um registro.

Veja os exemplos `registro.c` e `vetor_registro.c`

Exercício

Estruturar o lab4 em termos de funções e incluir a opção "todas as verificações"

```
1 - Número primo
2 - Número par
3 - Quadrado perfeito
4 - Cubo perfeito
5 - Todas as verificações
6 - Sair
```

Exercício

Derivada de um polinômio

Reestruture o programa `deriv.c` utilizando funções de maneira que

- as operações de leitura, escrita e cálculo da derivada de um vetor fiquem em funções separadas.
- a escrita do polinômio cuide de detalhes como
  - não escrever coeficiente quando este for igual a 1.0
  - não escrever x^0
  - escrever apenas x ao invés de x^1