Revisão

Prof. John Lenon C. Gardenghi

3.5 Funções

Um função é um segmento de código que recebe parâmetros de entrada e, com eles, gera alguma saída. Em linguagem C, declara-se funções da seguinte forma

```
tipo nome_funcao (tipo par1, tipo par2, ...) {
  /* comandos */
  return algo_do_tipo;
}
```

- `tipo nome_funcao(tipo par1, tipo par2, ...)` é chamado de **cabeçalho** da função e
- os comandos são chamados de **corpo** da função.

Exemplo 1. *Escreva uma função que receba dois números inteiros e retorne o maior entre eles.*

```
int max (int a, int b) {
  if (a > b) {
    return a;
  }
  else {
    return b;
  }
```

Importante:

- Uma função retorna apenas um elemento.
- Toda função deve retornar alguma coisa. Funções que não retornam alguma coisa são do tipo `void` e podem ter o comando `return`.

Exemplo 2. *Faça uma função que receba a idade de uma pessoa e imprima se ela já pode votar, se já pode dirigir, ou se não pode fazer nenhum dos dois.*

```
void verifica_idade (int idade) {
  if (idade >= 16) {
    printf ("Voce ja pode votar");
```

```
    if (idade >= 18) {
      printf (" e ja pode dirigir.\n");
    }
    else {
      printf (", mas nao pode dirigir.\n");
    }
  }
  else {
    printf ("Voce nao pode votar nem dirigir.\n");
  }
}
```

As funções são importantes, pois

- reduzem a *redundância* do código,
- favorecem a leitura do código por meio da *modularização*, é como dividir um texto em seções e subseções, e
- permitem a *abstração*, isto é, é possível criar caixas-pretas com funcionalidades que não precisamos entender, a priori, como são implementadas.

A ordem em que as funções são declaradas no código também é importante. Há duas opções:

1. As funções devem ser feitas **antes** de serem *chamadas*.
2. Os **protótipos** das funções devem ser declarados no início do código. (Mais recomendado!)

Protótipo da função é declarar apenas seu *cabeçalho* terminado em ponto-e-vírgula. No Exemplo 2 acima, declararíamos depois dos `#include`

```
void verifica_idade( int idade );
```

3.6 Estruturas heterogêneas

Estruturas heterogêneas são "pacotes" de variáveis, agrupadas em *registros*, que potencialmente possuem tipos diferentes. Cada variável é um *campo* do registro. Em linguagem C, esses registros são chamados `structs`.

Em C, declaramos uma `struct` da seguinte forma:

```
struct nome {
  int  campo1;
  char campo2;
  ...
};
```

Exemplo 3. *Faça um registro para representar um dia do ano.*

```
struct dma {
  int dia;
  int mes;
  int ano;
};
```

Deste modo, a `struct` pode ser usada como se fosse um tipo de variável. Para declará-la, basta

```
struct dma x;
```

Para fazer com que ela seja, de fato, um tipo, podemos usar a palavra-chave `typedef`.

```
typedef struct dma {
  int dia;
  int mes;
  int ano;
} data;
```

Deste modo, basta declarar

```
data x;
```

Exemplo 4. *Faça uma função que receba um registro de data e imprima a data.*

```
void imprime_data (data d) {
  printf ("%d/%d/%d.\n", d.dia, d.mes, d.ano);
}
```

3.7 Estruturas homogêneas

Estruturas homogêneas, ao contrário das heterogêneas, são *registros* que armazenam diversos elementos de **mesmo tipo**. Em linguagem C, temos os **vetores** e as **matrizes**.

Vetores são estruturas unidimensionais que são armazenadas *contiguamente* na memória. Declara-se como

```
tipo nome_vetor[tamanho_vetor];
```

Exemplo 5. *Faça uma função que retorne o máximo elemento de um vetor.*

```
int maximo (int v[], int n) {
  int maior = v[0];
  for (int i = 1; i < n; i++)
    if (v[i] > maior)
      maior = v[i];
  return maior;
}
```

3.8 Strings

Uma `string` é uma cadeia de caracteres. Em linguagem C, podemos definir um `string` como sendo um *vetor* de caracteres.

Declara-se

```
char str[tamanho+1];
```

Uma `string`, em C, é terminada com o caracter nulo, o famoso `\0`. Por isso, ao declarar uma `string`, é importante declararmos o vetor correspondente com o tamanho desejado da `string` mais um, para garantir espaço ao caracter nulo.

Cada caracter que pode ir numa `string` é um caracter constante na tabela ASCII[1].

Há algumas formas de ler e escrever `strings` na tela.

Escrita. Para escrever uma string na tela, podemos usar

```
printf ("%s\n", str);
```

Há ainda a função `fputs` (*file put string*) da biblioteca `stdio.h`. Essa função grava uma `string` num arquivo. Todavia, se o arquivo for `stdout` (*standard output* – saída padrão), a `string` é impressa na tela, que é a saída padrão.

```
fputs (str, stdout);
```

Leitura. Há várias formas de se ler `strings`, algumas mais automáticas, outras manuais. São elas:

1. A função `getchar` lê caracter por caracter. Costuma ser a forma mais segura de se ler `strings`, já que você pode ler até encontrar determinado caracter. Por exemplo, para ler linha por linha, poderíamos utilizar

   ```
   char c, str[50];
   int i = 0;
   getchar (c);
   while (c != '\n') {
     str[i] = c;
     i++;
     getchar (c);
   }
   ```

2. A função `scanf`. Usar apenas

   ```
   scanf ("%s", str);
   ```

   faz com que os caracteres sejam lidos até que se encontre um espaço, ou seja, lê-se uma palavra por vez. Para ler uma linha inteira, podemos usar uma *expressão regular* no especificador de formato do `scanf` da seguinte forma:

   ```
   scanf ("%[^\n]", str);
   ```

   A expressão `[\^n]` quer dizer "aceite qualquer caracter *exceto* (que é o `^`) o `\n`". Todavia, há de ser atenção com esse formato, pois podemos ter problemas quando estivermos interessados em ler uma sequência de `strings`. Por exemplo, implemente e rode o código a seguir.

[1] `https://www.ime.usp.br/~pf/algoritmos/apend/ascii.html`

```c
#include <stdio.h>
int main () {
  char str[100];
  for (int i = 0; i < 10; i++) {
    scanf ( "%[^\n]", str );
    printf ("String %d: %s\n", i+1, str );
  }
  return 0;
}
```

Você vai notar que a será impressa 10 vezes a primeira string que você digitar. E por que isso acontece? Isso acontece pois o comando

```c
scanf ( "%[^\n]", str );
```

lê a `string` até encontrar o `\n`, mas **não lê** o `\n`. Deste modo, na segunda iteração do laço, nada é lido, pois ainda há um `\n` na entrada padrão que não foi lido. Isso faz com que apenas a primeira `string` seja impressa até terminar o laço. Para resolver isso,

- O `\n` pode ser lido usando `getchar` depois do `scanf`. Assim:

```c
#include <stdio.h>
int main () {
  char str[100];
  for (int i = 0; i < 10; i++) {
    scanf ( "%[^\n]", str );
    getchar ();
    printf ("String %d: %s\n", i+1, str );
  }
  return 0;
}
```

- Você pode colocar um espaço antes do `%` dentro do `scanf`. Esse espaço extra fará com que o `scanf` consuma qualquer caracter do tipo espaço (espaço, tabulação e quebra de linha) antes de começar a ler a `string`. O código ficaria assim:

```c
#include <stdio.h>
int main () {
  char str[100];
  for (int i = 0; i < 10; i++) {
    scanf ( " %[^\n]", str );
    printf ("String %d: %s\n", i+1, str );
  }
  return 0;
}
```

3. A função fgets. Embora seja uma função para ler de um arquivo, se utilizarmos o arquivo `stdin`, será considerado o que é lido do teclado (*standard input*, entrada padrão).

```c
fgets (string, tamanho, stdin);
```

O problema do `fgets` é que ele lê um tamanho pré-determinado, inclusive as quebras de linhas que houver neste intervalo. Por isso, deve ser usado com cuidado.

4 Leiaute de um programa

Conteúdo extraído da página do Prof. Paulo Feofiloff[2]

Os dois elementos principais do leiaute são

1. a *indentação*, que é o recuo das linhas em relação à margem esquerda da página, e
2. os *espaços* entre as palavras (e demais símbolos) de uma linha.

O leiaute de um programa deve seguir os mesmos princípios que o leiaute de qualquer outro tipo de texto.

Para cuidar bem do leiaute em seu programa,

- use um espaço para separar cada palavra da palavra seguinte (os símbolos `=`, `<=`, `while`, `if`, `for` etc. contam como palavras);
- deixe um espaço depois, mas não antes, de cada sinal de pontuação;
- deixe um espaço depois, mas não antes, de parêntese e colchete direito;
- deixe um espaço antes, mas não depois, de parêntese e colchete esquerdo.

Por exemplo,

- jamais escreva `while(j < n)` no lugar de `while (j < n)`;
- jamais escreva `else{` no lugar de `else {`;
- não escreva `for (i=1;i<n;++i)` no lugar de `for (i = 1; i < n; ++i);` etc.

Há três exceções notórias às regras acima: escreva

- `x.prox` e não `x . prox`,
- `x[i]` e não `x [i]`,
- `x++` e não `x ++`.

Além disso, é usual não deixar espaços antes nem depois dos operadores de multiplicação e divisão. Assim, é usual escrever `x*y` e `x/y` em lugar de `x * y` e `x / y` respectivamente.

Sugiro deixar um espaço entre o nome de uma função e o parêntese esquerdo seguinte, ainda que isso contrarie a notação tradicional da matemática. Por exemplo,

```
funcao (arg1, arg2)
```

em vez de `funcao(arg1, arg2)`, pois o primeiro leiaute é mais legível que o segundo.

[2] `https://www.ime.usp.br/~pf/algoritmos/aulas/layout.html`