Laboratório de Algoritmos I
Funções e Pré-processador C

Função - Conceito

- Uma função é um conjunto de instruções desenhadas para cumprir uma tarefa particular e agrupadas numa entidade com um nome para referenciá-la.

- Por que usar funções?
  - aproveitamento de código já construído;
  - evitar repetição de código;
  - ajudar na legibilidade do programa;
  - ajudar na manutanção de código.

Funções pré-definidas

- Em nossos programas já utilizamos funções diversas vezes!
  - `printf()`: a função de impressão;
  - `scanf()`: a função de entrada de dados;
  - `strcmp()`, `strcpy()`, `strlen()` e `strcat()`: funções que trabalham sobre strings;
  - `rand()`: a função que gera números aleatórios.

- Além dessas funções, temos também a declaração `int main()` em todos os programas que escrevemos: o `main()` é uma função! Ele é, por padrão, a primeira função do programa a ser executada quando chamamos o executável do programa.

Funções – Formato Geral

- Para criar nossas próprias funções em C devemos usar o seguinte formato:

```
tipo_da_funcao NomeDaFuncao(Lista_de_Parametros) {
  // corpo da função
}
```

- A `Lista_de_Parametros` é opcional. Observe que o nosso `main()` segue esse formato:

```
int main(int argc, char *argv[]) {
  // corpo da função
}
```

Funções – Formato Geral

- A seguinte função, por exemplo:

```
float celsius(float fahr) {
    float c;
    c = (fahr - 32.0) * 5.0/9.0;
    return c;
}
```

- É uma função de nome celsius;
- É uma função do tipo float, ou seja, retorna um valor do tipo float;
- Possui somente um parâmetro, do tipo float e de nome fahr.

Funções - Tipo

- Uma função tem sempre um tipo. O tipo de uma função indica que tipo de valor ela vai retornar.

- A função `celsius()` mostrada anteriormente, por exemplo, é do tipo `float` e, portanto, deve retornar um valor do tipo `float`.

- A função `strlen()` é uma função do tipo `int`. Como podemos nos lembrar, ela retornava sempre um valor inteiro, o tamanho de uma string.

Funções - Tipo

- O valor retornado por uma função é indicado dentro da função pelo comando `return`.

- O valor retornado pelo comando return deve ser sempre do tipo da função declarada.

Funções - Tipo

- Uma função pode não ter nenhum valor de retorno!
  - nesse caso, ela não possui comandos return dentro de seu corpo.

- Uma função sem valor de retorno é chamada de **procedimento**.

- O tipo de um procedimento é sempre **void**.

```
void procedimento(int x, int y) {
     // corpo do procedimento
     // não há comando return
}
```

Funções - Chamada

- Chamar uma função é o meio pelo qual solicitamos ao programa para desviar o controle e passar a executar as instruções da função, e que ao término desta volte o controle para a posição seguinte à da chamada.

- Em nossos programas, quanto chamamaos `strlen()`, por exemplo, a execução do código sai do nosso `main()` e vai para o código da função `strlen()`. Ao terminar a execução do `strlen()`, a execução volta para o nosso `main()`.

- Para chamar uma função basta escrever seu nome, passando a lista de parâmetros entre parênteses.

Funções - Parâmetros

- Os parâmetros são dados que podem ser passados para a função. Dentro da função, os parâmetros são usados da mesma forma que usamos variáveis.

- Parâmetros podem ser de qualquer tipo:
- int func(int x) – parâmetro tipo int
- int func(float x) - parâmetro tipo float
- int func(char c) - parâmetro tipo char
- int func(char c[]) ou int func(char c[10]) - parâmetro tipo string

- Para passar matrizes como parâmetros:
- void func(int matriz[10][10]);
- void func(int matriz[][]) é **errado**!

Funções - Parâmetros

- Uma função pode ter um número qualquer de parâmetros.
- Na chamada da função, devemos passar um valor para cada parâmetro definido.
- Os parâmetros da função na sua declaração são chamados **parâmetros formais**. Na chamada da função os parâmetros são chamados **parâmetros reais**.
- Os parâmetros são passados para uma função de acordo com a sua posição. O primeiro parâmetro atual (da chamada) define o valor o primeiro parâmetro formal (na definição da função, e assim por diante.
- Os nomes dos parâmetros na chamada não tem relação com os nomes dos parâmetros na definição da função.

Funções - Protótipo

- Deve ser declarado antes da definição e antes das chamadas à função. Serve para informar ao compilador a existência da função, seu tipo e seus parâmetros. Exemplo:

```
float celsius(float);
```

- O protótipo não é obrigatório, desde que a função seja definida antes da função que a chama.

- Quando a função não tem parâmetros, seu protótipo deve incluir a palavra void no lugar dos parâmetros:

```
int funcaoteste(void);
```

Funções com mais de um comando return

- Uma função pode ter mais de um comando return. Quando o controle alcança um comando return, ele sai da função, retornando para o local de chamada o valor passado ao comando return.

Chamada a Função sendo usada como parâmetro para outra função

- Uma chamada a uma função pode ser usada como parâmetro para outra função, desde que o tipo da função (tipo do retorno) seja igual ao tipo do parâmetro. Exemplo:

```
printf("Celsius = %.2f\n", celsius(f));
```

Escopo de variáveis

- As variáveis em C podem ser declaradas, basicamente, de 3 maneiras diferentes:
  - dentro de uma função: variáveis locais;
  - fora de uma função: variáveis globais;
  - como parâmetro de uma função: parâmetros formais.

Escopo de variáveis

- **Variável local**: só pode ser utilizada pela função ou bloco que a declarou. Não é reconhecida por outras funções e só pode ser usada dentro do bloco de função onde está declarada. Uma variável local é criada quando a função começa a ser executada e destruída no final da execução dessa função.

```
void func() {
  int i;
  ...
}
int main() {
  int i;
  ...
}
```

- Nesse trecho, as duas variáveis i não possuem qualquer relação. São variáveis locais, cada uma em sua função.

Escopo de variáveis

- **Variável global**:existe durante a execução de todo o programa, e pode ser utilizada por qualquer função. Devem ser declaradas fora de qualquer função, inclusive do main(), e no início de um programa.

```
int x=5;
void func() {
 int c = x;
 ...
}
int main() {
 int i = x;
 ...
}
```

- Nesse trecho, a variável x é global. Ela pode ser acessada dentro de qualquer função.

Escopo de variáveis

- **Parâmetros formais**: são variáveis locais de uma função que são inicializadas no momento da chamada da função. Só existem dentro da função onde foram declarados. Podem ser utilizadas normalmente como uma variável local dentro do bloco de função onde estão.

```
void func(int x) {
  ...
}
```

- No trecho acima, x é um parâmetro formal. Ele será inicializado com um valor na chamada da função, e existe somente dentro dela

O Pré-Processador C

- É um programa que examina o programa fonte escrito em C e executa nele certas modificações antes da compilação, com base em instruções chamadas de diretivas.

- O pré-processamento faz parte do compilador, e é executado automaticamente antes da compilação

O Pré-Processador C

- As diretivas são geralmente usadas para tornar o programa mais claro e fácil de manter.

- Elas podem ser escritas em qualquer lugar do programa (geralmente no início) e se aplicam somente do ponto onde são escritas até o final do código.

A diretiva #include

- Causa a inclusão de outro arquivo em noso programa fonte. Na verdade, o compilador substitui a diretiva #include de noso programa pelo conteúdo do arquivo indicado, antes de compilar o programa.

- Quando usamos a diretiva #include com os sinais < e >, o arquivo é procurado somente na pasta include.

- Quando usamos aspas duplas, o arquivo é procurado primeiro na pasta atual e depois na pasta include.

A diretiva #define

- Na sua forma mais simples, é usada para definir constantes com nomes apropriados. Exemplo:

```
#define PI 3.14
```

- Onde PI é o **identificador**, e 3.14 é o **texto**.

A diretiva #define

- Por convenção, o identificador é sempre escrito em letras maiúsculas.

- Não há substituição dentro de cadeias de caracteres. Ou seja, se o código tiver, por exemplo, a palavra PIANO, ela continuará inalterada.

- Observe que não há ponto e vírgula após nenhuma diretiva do pré-processador.

Macros

- É quando utilizamos a diretiva #define com parâmetros. Exemplo:

```
#define PRN(n) printf("%.2lf\n", n)
```

- Obs.: nunca deve haver espaço em branco no identificador da macro. Exemplo:

`#define PRN (n)` é errado!

Macros

- Chamando a macro:

```
double n1 = 10;
PRN(n1);
```

Cuidados com Macros

- Considere a seguinte macro e sua chamada:

```
#define SOMA(x, y) x + y

z = 10 * SOMA(3, 4);
```

- Qual é o valor atribuído a z?

Cuidados com Macros

- Após a expansão da macro teremos:

```
z = 10 * 3 + 4;
```

- Como o operador * tem maior prioridade que o operador +, o resultado será 30 + 4 = 34.

Cuidados com Macros - solução

- Colocar o texto da macro entre parênteses:

```
#define SOMA(x,y) (x + y)

z = 10 * SOMA(3,4);
```

- Nesse caso, qual o valor atribuído a z?

Cuidados com Macros - solução

- Após a expansão da macro teremos:

```
z = 10 * (3 + 4);
```

- Neste caso a operação entre parênteses será realizada primeiro, então teremos z = 10 * (7) = 70.

Cuidados com Macros - solução

- Para evitar problemas deste tipo em qualquer situação, o ideal é sempre envolver cada parâmetro com parênteses. O nosso exemplo ficaria desta forma:

```
#define SOMA(x,y) ((x) + (y))
```

Usando macros em outras macros

- Exemplo:

```
#define AREACIRCULO(r)  (PI)*((r)*(r))
```

Funções versus Macros

- Macros são mais rápidas na execução e não é necessário definir os tipos dos parâmetros.
- No entanto, o código do programa compilado será maior, visto que o código da macro será duplicado a cada chamada.
- Além disso, definir macros complexas pode deixar o programa confuso.