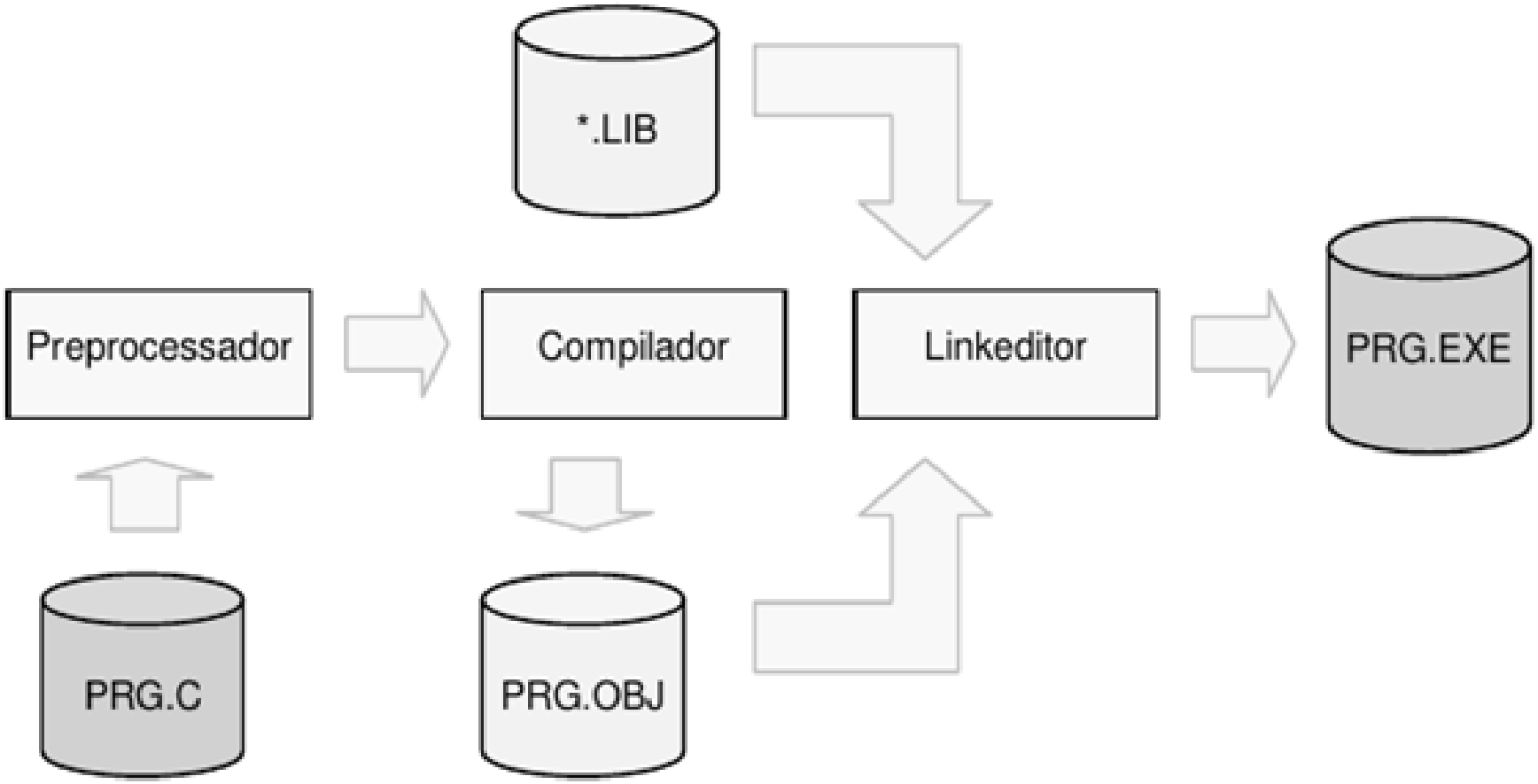
LINGUAGEM DE PROGRAMAÇÃO II

DIRETIVAS DE COMPILAÇÃO

Aula 4

Visão Geral


*.LIB
Preprocessador
Compilador
Linkeditor
PRG.EXE
PRG.C
PRG.OBJ

O Preprocessador

- O preprocessador realiza uma série de modificações no código-fonte antes da fase de análise, realizada pelo compilador.
- Essas modificações são especificadas através das diretivas de preprocessamento embutidas no código-fonte do programa.
- Existem diversas diretivas, sendo as principais:
  - #include e #define;

#define

- A diretiva #define serve para definir constantes simbólicas que aumentam a legibilidade do código-fonte.
- O texto associado ao identificador pode ser inclusive uma palavra reservada.
- Essa diretiva associa um identificador a um texto da seguinte maneira:
  - #define identificador texto

#define

- Exemplo:

```
#include <stdio.h>

#define diga  printf
#define oi    "\nOlá, tudo bem?"

main() {
   diga(oi);
}
```

#include

- Essa diretiva faz com que uma cópia de um arquivo de código (biblioteca ou código fonte), cujo nome é dado entre “<“ e “>”, seja incluído no código-fonte.

- A notação < .. > é preferencialmente utilizada para arquivos de inclusão padrão da linguagem C. Para incluir arquivos definidos pelo usuário, utilize aspas (″ .. ″)

FUNÇÕES

Aula 4

Funções

- Funções (também chamadas de **rotinas**, ou **sub-programas**) são a essência da programação estruturada.

- Funções são segmentos de programa que executam uma determinada tarefa específica.

- Já vimos o uso de funções providenciadas pelas bibliotecas-padrão do C (como o *printf*() e o *scanf()*), por exemplo.

Funções

- É possível escrever nossas próprias rotinas e, deste modo, segmentar um programa grande em vários programas menores.

- Esta segmentação é chamada de **modularização** e permite que cada segmento seja escrito, testado e revisado individualmente sem alterar o funcionamento do programa como um todo.

- Permite ainda que um programa seja escrito por vários programadores ao mesmo tempo, cada um escrevendo um segmento separado.

Funções

- A estrutura de uma função é muito semelhante a estrutura dos programas que escrevemos até agora. Ela é constituída por um **bloco de instruções** que definem os procedimentos efetuados pela função, um **nome** pelo qual a chamamos e uma **lista de argumentos** passados a função.

- De modo formal, a sintaxe de uma função é a seguinte:

```
tipo_de_retorno nome_da_função (tipo_1 arg_1, tipo_2 arg_2, ...)
{
  [bloco de instruções da função]
}
```

Funções

- A primeira linha da função contém a **declaração** da função. Na declaração de uma função se define o **nome** da função, seu **tipo de retorno** e a **lista de argumentos** que recebe. Em seguida, dentro de chaves {}, definimos o bloco de instruções da função.

- O **tipo de retorno** da função especifica qual o tipo de dado retornado pela função. Se a função não retorna nenhum valor, devemos definir o retorno como void, ou seja um retorno ausente.

- Vale notar que existe apenas **um** valor de retorno para funções em C. **Não** podemos fazer o retorno de **dois ou mais** valores!! Porém isto não é um limitação séria pois o uso de ponteiros contorna o problema, como veremos mais tarde.

Funções

- A **lista de argumentos** da função especifica quais são os valores que a função recebe. As variáveis da lista de argumentos são manipuladas normalmente no corpo da função.

- A chamada de uma função termina com a instrução ***return*** que transfere o controle para o programa chamador da função, aleém de retornar o resultado da função.

- Quando escrevemos a definição de uma função **antes** do programa principal, a sintaxe geral para isto é a seguinte:

```
tipo nomef(...){              // definição da função
  [corpo de função]
}
void main(){                  // programa principal
  ...
  var = nomef(...)            // chamada da função
  ...
}
```

Funções

- **Exemplo:**

```
float media2(float a, float b){
     float med;
     med = (a + b) / 2.0;
     return(med);
}

void main(){
     float num_1, num_2, med;
     puts(”Digite dois números:”);
     scanf(”%f %f”, &num_1, &num_2);
     med = media2(num_1, num_2);
     printf(”\nA media destes números e´ %f”, med);
}
```

Hierarquia de Funções

- É possível que um programa principal chame uma função que por sua vez chame outra função... e assim sucessivamente.

- Quando isto ocorre, devemos ter o cuidado de definir (ou incluir) as funções em ordem crescente de hierarquia, isto é, uma função **chamada** é escrita antes de uma função **chamadora**.

Escopo de Variáveis

- A regra de escopo define onde as variáveis e funções são reconhecidas.

- Em C, uma variável só pode ser **usada** após ser **declarada** e o local onde uma variável é declarada define seu **escopo** de validade. Uma variável pode ser **local**, **global** ou **formal** de acordo com o local de declaração.

Variáveis Locais

- São declaradas **dentro do bloco** de uma função.

- Uma variável local tem validade apenas dentro do bloco onde é declarada, isto significa que podem ser apenas acessadas e modificadas dentro de um bloco.

- O espaço de memória alocado para esta variável é **criado** quando a execução do bloco é iniciada e **destruído** quando encerrado, assim variáveis de mesmo nome mas declaradas em blocos distintos, são para todos os efeitos, variáveis distintas.

Variáveis Locais

- **Exemplo:**

```
float media2(float a, float b){
      float med;
      med = (a + b) / 2.0;
      return(med);
    }
```

- No exemplo:
  - a função ***media2*** recebe dois argumentos do tipo float: a e b.
  - A média desses valores é armazenada na variável interna ***med***.
  - A função retorna, para o programa que a chamou, o valor da variável med, usando a instrução ***return***.

Variáveis Locais

```
float media2(float a, float b){
       float med;
       med = (a + b) / 2.0;
       return(med);
     }
void main(){
       float num_1, num_2, med;
       puts(”Digite dois números:”);
       scanf(”%f %f”, &num_1, &num_2);
         med = media2(num_1, num_2);
       printf(”\nA media destes números e´ %f”, med);
}
```

- No exemplo, ***med*** é uma variável local definida pela função media(). Outra variável ***med*** é também definida pela função main(). Para todos os efeitos estas variáveis são distintas.

Variáveis Formais

- São variáveis locais declaradas na **lista de parâmetros** de uma função.
- Têm validade apenas dentro da função onde é declarada, porém servem de suporte para os valores passados pelas funções.
- As variáveis formais na **declaração** da função e na **chamada** da função podem ter nomes distintos. A única exigência é de que sejam do mesmo tipo.
- No exemplo anterior, a e b são parâmetros formais declarados na função media2(). Observe que a função é chamada com os valores de num_1 e num_2. Mesmo que os valores de a e b fossem alterados os valores de num_1 e num_2 não seriam alterados.

Variáveis Globais

- Uma variável é dita *global*, se for declarada **fora do bloco** de uma função.

- Uma variável global tem validade no escopo de todas as funções, isto é, pode ser acessada e modificada por qualquer função.

- O espaço de memória alocado para esta variável é **criado** no momento de sua declaração e **destruído** apenas quando o programa é encerrado.

Variáveis Globais

- **Exemplo:**

```
float a, b, med;
void media2(void){
     med = (a + b) / 2.0;
     }
void main(){
       puts(”Digite dois números:”);
       scanf(”%f %f”, &a, &b);
        media2();
       printf(”\nA media destes números e´ %f”, med);
}
```

- No exemplo acima, a, b, med são variáveis globais definidas fora dos blocos das funções media() e main(). Deste modo, as funções têm acesso às variáveis, que podem ser acessadas e modificadas por elas.

Tipos de Passagem de Parâmetros

- Por serem variáveis locais, os valores que uma função passa para outra não são alterados globalmente pela função chamada.

- Este tipo de passagem de argumentos é chamado de **passagem por valor** pois os valores das variáveis do programa chamador são copiados para as correspondentes variáveis da função chamada.

- Veremos mais adiante como alterar os valores das variáveis do programa chamador, quando estudarmos a **passagem de parâmetros por endereço**.

Recursividade

- Consiste no processo pelo qual uma função chama a si mesma, repetidamente, um número finito de vezes. Este recurso é muito útil em alguns tipos de algoritmos chamados de **algoritmos recursivos**.

- Vejamos um exemplo para esclarecermos o conceito: o cálculo do **fatorial** de um número.

- A definição de fatorial é, para um dado ***n***, inteiro positivo:

  $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$

  $0! = 1$

- Uma propriedade facilmente verificável dos fatoriais é que:

  $n! = n \cdot (n-1)!$

- Esta propriedade é chamada de propriedade recursiva: o fatorial de um numero pode ser calculado através do fatorial de seu antecessor.

Recursividade

- Ora, podemos utilizar esta propriedade para escrevermos uma rotina recursiva para o calculo de fatoriais. Para criarmos uma rotina recursiva, em C, basta criar uma chamada a própria função dentro dela mesma, como no exemplo a seguir:

```
long double fat(int n) {
   long double valor;
   if(n == 0){              // se fim da recursao
          valor = 1;     // calcula ultimo valor.
   }else{                    // senão
          valor = n * fat(n-1); // chama fat(n-1).
   };
   return(valor);           // retorna valor.
};
```

Recursividade

- Note que a linha grifada em azul representa o ponto onde a recursão termina. Esse resultado que indica o valor que encerra a recursão é OBRIGATÓRIO.

- Se ele não existir, a recursão será infinita!

```
long double fat(int n) {
   long double valor;
   if(n == 0){              // se fim da recursao
          valor = 1;     // calcula ultimo valor.
   }else{                    // senão
          valor = n * fat(n-1); // chama fat(n-1).
   };
   return(valor);           // retorna valor.
};
```

Exercícios

1) Escreva um programa que implemente uma calculadora com o seguinte menu de opções (use uma estrutura de repetição que só será encerrada quando o usuário escolher a opção FIM), onde TODAS as opções deverão ser implementadas usando uma FUNÇÃO RECURSIVA:
   1) Fatorial de N
   2) X^Y ($X^Y = X*X^{Y-1}$)
   3) X*Y (usando o método de somas sucessivas) (X*Y = X+(X*Y-1))
   4) X mod Y (X=X-Y enquanto X for maior do que Y)
   5) Sequência de N termos de Fibonacci ($N_i = N_{i-1}+N_{i-2}$)