Subprogramas

SCC 121 - Introdução à Programação

Introdução

- Para estruturação de programas, o conceito utilizado em C é denominado:

MODULARIZAÇÃO.

Introdução

- Modularizar um programa é **determinar** pequenos programas (subprogramas) que façam tarefas menores que irão auxiliar a execução de um programa, permitindo uma melhor **legibilidade** e **manutenibilidade** do programa.
- Existe uma forma básica para modularizar um programa em C:
  - Funções (*function*)

Introdução

- Uma função possui a seguinte forma:

```
tipo IDENTIFICADOR(lista_parâmetros)
{
   declarações de variáveis
   sequência de comandos
}
```

lista_parâmetros = tipo nome1, tipo nome2,...,tipo nomeN

- Uma função **sempre** retorna um tipo de dado específico (e.g. *int*, *float*, etc.).

Conceitos importantes

- Variáveis globais: variáveis declaradas no início de um programa, por exemplo,

```
 include <_____.h>
    int a, b;
main(){
     sequência_comandos;
 }
```

ou seja, são variáveis que podem ser manipuladas durante toda a execução do programa (e.g. a e b são variáveis globais).

Conceitos importantes

- Variáveis locais: variáveis declaradas no início de um subprograma, por exemplo,

```
include <_____.h>
   int a, b;

int funcao1()
{
   int c, d;
   sequência_comandos;
}
main(){
    sequência_comandos;
 }
```

as variáveis c e d são variáveis locais, isto é, são “visíveis” durante a execução de p1.

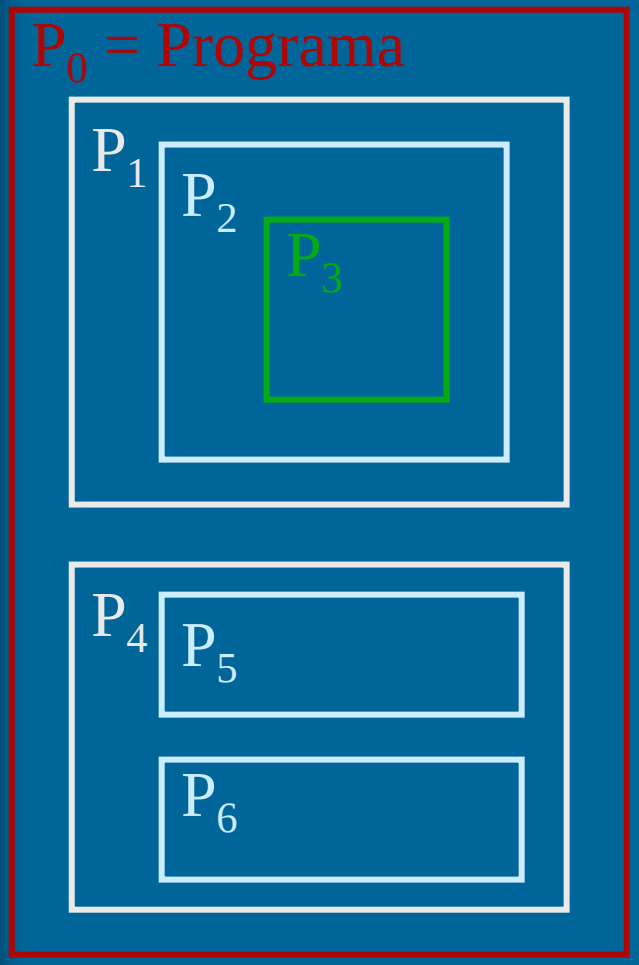
Conceitos importantes

- Uma seqüência de comandos contida em um programa é denominada como sendo um **bloco** de comandos.
- Quando subprogramas são declarados, blocos são associados a esses subprogramas, sendo assim, blocos contendo declarações locais a esses subprogramas.
- As declarações (e.g. variáveis) ocorridas em cada bloco de comandos podem ser denominadas, simplesmente, como **objetos**.

Conceitos importantes

- Os conceitos de variáveis globais e locais determinam um termo denominado **ESCOPO**.
- Variáveis que possuem escopo global são acessadas por todos os subprogramas que estão contidos em um programa.
- Variáveis que possuem escopo local são acessadas somente por subprogramas que estão contidos no subprograma.

Estrutura de Blocos



Nível		
	0	Programa
	1	P_1, P_4
	2	P_2, P_5, P_6
	3	P_3

Objetos definidos no bloco	São acessíveis nos blocos
P_0	$P_0, P_1, P_2, P_3, P_4, P_5, P_6$
P_1	P_1, P_2, P_3
P_2	P_2, P_3
P_3	P_3
P_4	P_4, P_5, P_6
P_5	P_5
P_6	P_6

Função

- Dado o programa abaixo, reescrevê-lo, usando uma função.

```
main()
{
   int i,n,fatnum;
   scanf("%d",n);
   fatnum = 1;
   for (i=1; i <= n; i++)
      fatnum = fatnum * i;
   printf("O fatorial de %d e %d\n" n, fatnum);

end.
```

Calculo do Fatorial atraves de função

```
#include <stdlib.h>

float fatorial(float *); // prototipo da funcao fatorial
main(){      // programa principal
 float  N;
 // Leitura dos dados
 printf("Entre com o valor de N:\n");
 scanf("%f", &N);
 while (N<0){
   printf(" Entre com um valor para N, nao negativo:\n");
   scanf("%f", &N);
}
// Impressão dos resultados
  printf("O valor do fatorial de: %f e igual a: %f", N, fatorial(N));
  getch();
}
```

```
// Definição da função que calcula o fatorial

float fatorial(float var){
  //  Calculo do fatorial
  float FAT, I;
  FAT =1;

  for (I=var;I>=1;I--){
      FAT = FAT * I;
      }
return(FAT);
}
```

Passagem de Parâmetros

- É a forma como é feita a **correspondência** entre **parâmetros** e **argumentos**.
- Existem duas formas básicas de passagem de parâmetros na linguagem C: **valor** ou **endereço**.
- A passagem de parâmetros por valor faz com que seja **criada** uma variável local ao subprograma e o valor do argumento é diretamente copiado nela. **Uma alteração no parâmetro não altera o argumento**.

Passagem de Parâmetros

- A passagem de parâmetros por endereço faz com que o subprograma trabalhe **diretamente** com a variável-argumento. **Uma alteração no parâmetro acarretará na modificação do argumento**.

Exemplo de passagem por endereço

```
#include <stdlib.h>
#include <conio.h>
#include <stdio.h>
float fatorial(float *); // prototipo da funcao fatorial
main(){                  // programa principal
 float  N;
 // Leitura dos dados
 printf("Entre com o valor de N:\n");
 scanf("%f", &N);
 while (N<0){
   printf(" Entre com um valor para N, nao negativo:\n");
   scanf("%f", &N);
}
// Impressão dos resultados
  printf("O valor do fatorial de: %f e igual a: %f", N, fatorial(&N));
  getch();
}
```

Passagem por endereço

```
float fatorial(float *var){
  //  Calculo do fatorial
  float FAT, I;
  FAT =1;

  for (I=*var;I>=1;I--){
      FAT = FAT * I;
      }
  *var+=50;
  printf("var=%f", *var);
  return(FAT);
}
```

O que aconteceu?

- Rodar o programa anterior.
- Verificar que nesse caso o valor de N (o no. lido) foi alterado também no programa principal

  Há casos, como este, que isto não é desejável.

Mas, existem casos em que desejamos que retorne mais um valor para o programa principal e dessa forma podemos usar os próprios argumentos da FUNÇÃO para fazer o retorno de mais valores;

Exemplo: retorno de mais de uma variável

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

float fatorial(float, float*); // prototipo da funcao fatorial

main(){
 float  N, N1;

 // Leitura dos dados
 printf("Entre com o valor de N:\n");
 scanf("%f", &N);
 while (N<0){
   printf(" Entre com um valor para N, nao negativo:\n");
   scanf("%f", &N);
}
// Impressão dos resultados
  printf("O valor do fatorial de: %f e igual a: %f\n e o valor de y e:%f", N, fatorial(N,&N1),N1);
  getch();
}
```

```
float fatorial(float var,float *y)// retorno: fatorial e seno
  //  Calculo do fatorial
  float FAT, I;
  FAT =1;
  printf("var=%f", var);
  for (I=var;I>=1;I--){
      FAT = FAT * I;
      }
  *y=sin(var);
  printf("valor do seno:%f\n", *y);
  return(FAT);
}
```

Subprogramas Recursivos

- O Escopo de um subprograma é delimitado desde de sua definição até o fim do bloco que está definido.
- Sendo assim, um subprograma pode ser chamado por um outro subprograma ou até mesmo por si próprio.
- Quando um subprograma contém uma chamada a si próprio, ele é dito um **subprograma recursivo**.

Subprogramas Recursivos

- Exemplos de definições recursivas:

(i) $fat(n) = n * fat(n-1)$, $n >= 1$

$fat(n) = 1$, $n = 0$

(ii) $fib(n) = fib(n-1)+fib(n-2)$, $n >= 2$

$fib(n) = 1$, $n = 1$

$fib(n) = 0$, $n = 0$

Subprogramas Recursivos

- Todas as definições recursivas têm em comum:
  - um índice para cada definição;
  - pelo menos, uma definição não-recursiva (condição de saída) que, ao ser alcançada garante a interrupção da recursão;
- Cada chamada faz a função ser executada novamente, com um argumento diferente para cada nova execução.

Subprogramas Recursivos

fib(n) = fib(n-1)+fib(n-2) , n >= 2
fib(n) = 1 , n = 1
fib(n) = 0 , n = 0

```
int fib(int n)
{
   if (n == 0)
      fib = 0;
   else if (n == 1) fib = 1;
        else fib = fib(n-1)+fib(n-2);
}
```

Subprogramas Recursivos

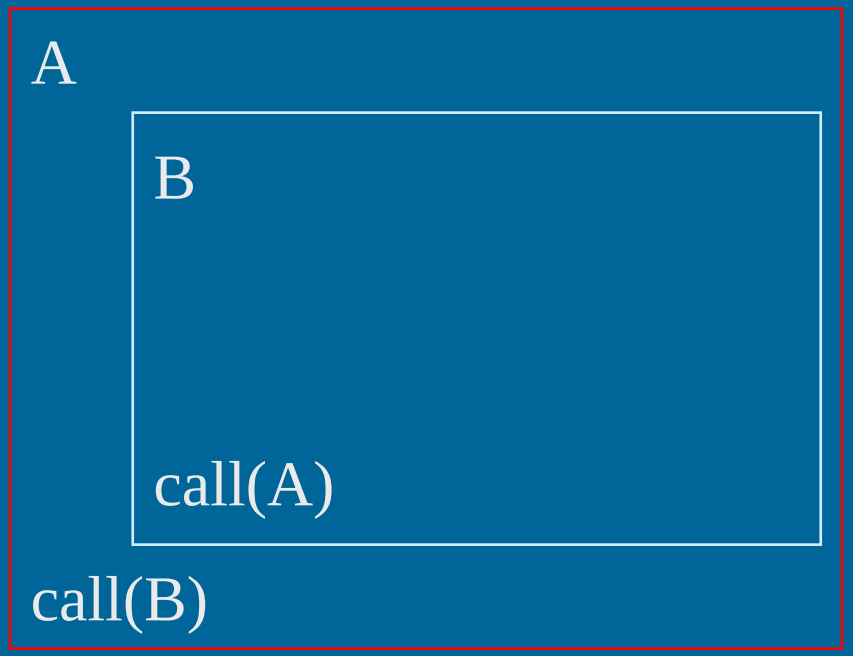
- Programas recursivos consomem **tempo de execução** e **espaço em memória** (pilha de ativação para cada procedimento recursivo) e **podem ser ineficientes** para alguns casos.
- Por exemplo, o cálculo do número de Fibonacci iterativo gasta menos recursos computacionais do que o cálculo recursivo.
- **Faça o programa fib(n) iterativo** para comprovar a eficiência perante o recursivo.

Recursão X Iteração

- Todo processo **recursivo** pode ser **transformado** em um processo **iterativo**, bastando **simular** a pilha de recursão, caso não se conheça outro tipo de definição não-recursiva.
- A versão **não-recursiva** é, em geral, mais **eficiente** do que a recursiva.
- A escolha por uma função recursiva é feita quando **tempo/espaço** não são problemáticos, ou se a versão recursiva for mais **simples**.

Recursão Mútua

- Um subprograma A contém uma chamada de um subprograma B, se B chama A. Isso é conhecido como **Recursividade Mútua**.



Aplicação de Recursão

- Como foi dito anteriormente, em alguns casos, a versão recursiva de um determinado problema é mais simples que a versão não-recursiva.
- Um exemplo disso, é o problema das Torres de Hanói, que consiste de 3 regras básicas:

  (i) somente 1 disco é movido por vez;

  (ii) nenhum disco pode ser colocado sobre um disco menor;

  (iii) qualquer disco pode ser movido de qualquer pino para qualquer outro desde que respeite a regra (ii)

Aplicação de Recursão (2 de 4)

- Para resolver o problema, pode-se utilizar 3 pinos (A,B,C). Sendo que, A é o pino origem, B é o pino destino e C é um pino auxiliar.
- Uma estratégia para resolver este problema é a seguinte:

  se n=1 **mova** de *origem* para *destino*

  senão

  (i) **mova** n-1 de *origem* para *auxiliar*, usando *destino* como *auxiliar*;

  (ii) **mova** disco de *origem* para *destino*;

  (iii) **mova** n-1 de *auxiliar* para *destino* usando *origem* como *auxiliar*

Aplicação de Recursão

```
void hanoi( int n, int origem, int destino,int aux)
{
   if (n == 1)
      printf(“Mova disco do pino %d para o pino %d\n”,origem,
   destino);
   else{
        hanoi(n-1, origem, aux, destino);
        printf(“Mova disco do pino %d para o pino %d\n“,origem,
   destino);
        hanoi(n-1, aux, destino, origem);
   }
}
```

Aplicação de Recursão

```
main(){
   hanoi(3, 'A', 'B', 'C');
   getch();
}
```

FIM