Estruturas de Dados

Subrotinas e Parâmetros

De uma maneira simplificada, podemos dizer que uma subrotina é um trecho de programa que possui um nome. Para que as instruções de uma subrotina sejam executadas, basta invocar o seu nome, conforme as especificações da linguagem em que você estiver trabalhando. Todas as linguagens comerciais suportam a construção e a utilização de subrotinas e algumas, como a linguagem C, inclusive permitem que subrotinas sejam invocadas por meio de ponteiros.

Basicamente, são dois os casos em que recomenda-se a criação de subrotinas:

a. Repetição de código: trechos de programa que se repetem com alguma freqüência dentro da lógica. O código pode se repetir de forma exatamente igual em vários pontos ou pode possuir pequenas variações conforme o contexto em que é executado.

b. Identificação de hierarquias na lógica: colocar em subrotinas os trechos de programa que implementam tarefas secundárias ou mais específicas, de maneira a "enxugar" a lógica dos módulos principais dos detalhes de implementação de certas operações.

Através da utilização de subrotinas em nossos programas, obtemos uma série de benefícios, conforme pode ser observado na relação a seguir:

a. Reutilização de código: como os trechos de programa que se repetem com freqüência são colocados dentro de subrotinas que são escritas uma única vez, há um aumento de produtividade do programador, que não precisa escrever ou copiar inúmeras vezes a mesma seqüência de instruções.

b. Facilidade de manutenção do código: modificações feitas numa subrotina passam a ter efeito sobre todos os lugares onde são chamadas, desde que você compile e link-edite novamente os programas que a utilizam.

c. Padronização de programas: a centralização do código fonte de uma tarefa num único lugar garante ela funcione da mesma forma em todos os programas que a utilizam. Por exemplo, imagine a exibição de uma mensagem na tela: todos os programas que você escrever necessitarão implementar pelo menos uma vez esse tipo de tarefa. Caso você não a coloque numa subrotina para uso geral, muito provavelmente teremos pequenas diferenças entre as mensagens exibidas entre cada programa e, às vezes, até mesmo dentro de um mesmo programa. Esse tipo de inconsistência de interface é altamente indesejável e também impede que melhorias sejam implementadas com facilidade nos programas.

d. Aumento da clareza dos programas: ocorre devido à diminuição do tamanho das estruturas de controle; à utilização de rotinas com nomes mnemônicos e ao maior destaque das hierarquias entre as diversas partes do programa.

e. Diminuição do tamanho médio dos programas: proporcionada principalmente pela reutilização de código.

Podemos classificar as subrotinas em dois tipos principais: as subrotinas do tipo função e as subrotinas do tipo procedimento. Uma função executa seu processamento e ao final retorna um valor para o módulo que a chamou, como por exemplo uma função para calcular a área de um polígono ou determinar os dígitos verificadores de um CPF. O exemplo a seguir mostra uma função escrita na linguagem C para determinar o IMC (Índice de Massa Corporal) de um indivíduo.

```
float IMC(float Peso, float Altura)
{   return Peso / (Altura * Altura);
}
```

Observe que na sua especificação é indicado o tipo de dado que ela devolve que, em nosso exemplo é um número real de precisão simples (tipo `float`).

Uma subrotina do tipo procedimento, por sua vez, executa um processamento e não produz qualquer valor de retorno, como por exemplo uma rotina que faz a formatação da tela de um programa ou que emite um alarme sonoro. O exemplo a seguir mostra uma subrotina do tipo procedimento em C para converter o conteúdo de uma string para maiúsculas.

```
void Maiusculas(char *texto)
{   int cont;

    for (cont = 0; texto[cont] != '\0'; cont++)
        texto[cont] = toupper(texto[cont]);

    return;
}
```

Na linguagem C as subrotinas que não devolvem valor são indicadas como possuindo o tipo `void` e a instrução final `return` pode ser omitida.

Uma subrotina geralmente precisa ser informada, quando chamada, de alguns valores específicos que ela deve processar naquele momento. Fazemos isso por meio de parâmetros, e o módulo chamador é o responsável por informar à subrotina os parâmetros que esta deverá utilizar, como mostra o programa a seguir.

```
#include <stdio.h>

int main(void)
{   float vPeso, vAltura, ValorImc;

    printf("Informe o peso:\n");
    scanf("%f", &vPeso);
    printf("\nInforme a altura:\n");
    scanf("%f", &vAltura);

    ValorImc = IMC(vPeso, vAltura);

    if (ValorImc < 18.5)
        printf("\nMuito magro\n");
    else
        if (ValorImc >= 30.0)
            printf("\nMuito gordo\n");
        else
            printf("\nPeso razoável\n");

    return 0;
}
```

O programa do exemplo recebe dois valores numéricos, representando o peso e a altura de uma pessoa e os armazena nas variáveis locais `vPeso` e `vAltura`. Em seguida calcula, por meio da função `IMC` apresentada anteriormente, o valor do Índice de Massa Corporal correspondente e o armazena na variável `ValorImc`. Com base nesse dado emite então um display na tela indicando a situação do indivíduo. A rotina principal (`main`) ao chamar a função passa os valores que esta deve considerar em seu cálculo. Os conteúdos das variáveis `vPeso` e `vAltura` que são utilizados na invocação da função `IMC` são denominados de parâmetros reais daquela chamada, e as variáveis locais `Peso` e `Altura` definidas na rotina `IMC` são denominadas os parâmetros formais daquela chamada. É óbvio que os tipos de dados dos parâmetros formais e reais precisam ser compatíveis.

A comunicação entre as rotinas `main` e `IMC` ocorre então por meio da passagem de parâmetros da primeira para a segunda e pelo valor de retorno da segunda para a primeira. Em outras palavras: a rotina `main` "passa" os parâmetros para a rotina `IMC`. A rotina `IMC`, por sua vez, "devolve" um valor de retorno para a rotina `main`.

No exemplo dado, o conteúdo da variável `vPeso` da rotina principal é copiado para dentro da variável `Peso` da rotina `IMC`, e o conteúdo da variável `vAltura` da rotina principal é copiado para dentro da variável `Altura` da rotina `IMC`. Quando `IMC` começa seu processamento ela possui no interior de seus parâmetros formais os dados informados pela rotina chamadora. Dizemos que, nesse caso, os parâmetros foram **passados por valor**.

Existe uma outra maneira de se passar parâmetros, que é chamada de **passagem de parâmetros por referência**. Nesse caso o que é informado para a subrotina é o endereço de memória dos dados originais. O exemplo a seguir mostra o mesmo programa anterior utilizando a passagem de parâmetros por referência.

```
#include <stdio.h>

float IMC(float *Peso, float *Altura)
{   return *Peso / (*Altura * *Altura);
}

int main(void)
{   float vPeso, vAltura, ValorImc;

    printf("Informe o peso:\n");
    scanf("%f", &vPeso);
    printf("\nInforme a altura:\n");
    scanf("%f", &vAltura);

    ValorImc = IMC(&vPeso, &vAltura);

    if (ValorImc < 18.5)
        printf("\nMuito magro\n");
    else
        if (ValorImc >= 30.0)
            printf("\nMuito gordo\n");
        else
            printf("\nPeso razoável\n");

    return 0;
}
```

Agora a variável `Peso` da rotina `IMC` possui em seu interior as coordenadas de memória onde está a variável `vPeso` da rotina principal, e a variável `Altura` da rotina `IMC` possui em seu interior as coordenadas de memória onde está a variável `vAltura` da rotina principal.

Observe que tivemos mudanças na definição da rotina `IMC`, onde os parâmetros formais são agora ponteiros e a operação de cálculo necessita do operador '`*`' (que indica *'o conteúdo apontado por ...'*). Na rotina principal mudou apenas a forma de ativação da subrotina, pois ao explicitar os parâmetros precisamos agora do operador `&` (que indica *'o endereço de ...'*).

Na linguagem C os dados armazenados na forma de arrays (como as strings, por exemplo) são sempre passados por referência, como mostra o exemplo a seguir, que recebe uma palavra digitada pelo usuário e a exibe em maiúsculas na tela, fazendo a conversão por meio da subrotina `Maiusculas` apresentada anteriormente.

```
#include <stdio.h>

int main(void)
{   char palavra[20];

    printf("Informe a palavra:\n");
    scanf("%s", palavra);

    Maiusculas(palavra);

    printf("\nA palavra em maiúsculas eh %s\n", palavra);

    return 0;
}
```

Modularização de programas: coesão e acoplamento

O projeto de bons programas requer a preocupação tanto com a reutilização como com a posterior manutenção do código fonte. O código de um programa será considerado de difícil manutenção se não puder ser compreendido facilmente pelo programador ou se tiver sido escrito de maneira que as modificações sejam muito trabalhosas. A capacidade de reutilização do código, por sua vez, afeta tanto a produtividade do programador ao construir novos programas, por que permite aproveitar elementos já codificados e testados anteriormente, como na manutenção de programas já em funcionamento, pois sequências de instruções que se repetem explicitamente em diversos lugares do programa requerem que as modificações sejam feitas nos vários lugares em que elas aparecem, enquanto que se esse código repetitivo estiver definido dentro de uma subrotina, bastará modificar a subrotina e testar o programa para os diversos casos em que ela for chamada.

Utilizar subrotinas deve ser então uma prática comum para o programador, mas alguns critérios devem ser adotados para se avaliar a qualidade dos diversos módulos em que o programa for dividido. Os dois critérios de maior relevância são a coesão e o acoplamento verificado entre os módulos, que podem ser utilizados tanto na análise da qualidade de subrotinas comuns criadas pela programação estruturada como na análise da qualidade da estrutura das classes e métodos especificados na abordagem orientada a objetos.

O acoplamento indica o quanto dois módulos estão interligados entre si. Sempre haverá algum acoplamento entre as rotinas de um programa, afinal elas fazem parte de um sistema. A questão é decidir quanto de acoplamento devemos permitir, já que uma interligação muito forte entre dois módulos produz problemas na manutenção do programa, devido principalmente ao chamado ‘efeito de propagação’, em que uma modificação em algum aspecto de uma rotina leva a efeitos colaterais em outra, que por sua vez afetará uma terceira. Estamos interessados então em módulos com baixo acoplamento.

A coesão indica o quanto as diversas partes de um módulo estão interligadas, ou seja, em que grau o código de um módulo está concentrado em atingir seu objetivo. Rotinas coesas são aquelas em que todas as instruções contribuem especificamente para a realização do objetivo declarado da rotina. Rotinas pouco coesas são aquelas em que as diversas partes do código estão pouco integradas, possivelmente em uma rotina que tenta fazer várias coisas diferentes. O projetista de software deve buscar sempre módulos com coesão elevada, que possuem um objetivo claro e específico, onde todos os elementos contribuem diretamente para essa finalidade declarada.

O estudo de caso a seguir ilustra, por meio de um programa que, novamente, calcula o IMC de uma pessoa e exibe o diagnóstico na tela, os conceitos de coesão e acoplamento vistos aqui. A primeira versão do programa possui apenas a rotina principal e todo o processamento é realizado

em seu interior. É, então, uma rotina pouco coesa, pois trata da entrada e saída de dados, do cálculo do IMC e da determinação do diagnóstico correspondente. Verifique também que o valor numérico do IMC está sendo exibido com a formatação padrão do tipo `float` (`"%f"`, que exibe até 6 casas após a vírgula) e se desejarmos alterar essa formatação para duas casas após a vírgula apenas (com `"%.2f"`), teríamos que modificar 6 linhas do programa. Esse é um problema típico de módulos com modularidade deficiente.

```
#include <stdio.h>

int main(void)
{   float vPeso, vAltura, ValorIMC;

    printf("Informe o peso:\n");
    scanf("%f", &vPeso);
    printf("\nInforme a altura:\n");
    scanf("%f", &vAltura);

    ValorIMC = vPeso / (vAltura * vAltura);

    if (ValorIMC < 18.5)
        printf("\n\n\nValor do IMC: %f Diagnostico: %s\n",
                  ValorIMC, "DESNUTRIDO");
    else
        if (ValorIMC <= 25)
            printf("\n\n\nValor do IMC: %f Diagnostico: %s\n",
                    ValorIMC, "NORMAL");
        else
            if (ValorIMC <= 30)
                printf("\n\n\nValor do IMC: %f Diagnostico: %s\n",
                         ValorIMC, "SOBREPESO");
            else
                if (ValorIMC <= 40)
                    printf("\n\n\nValor do IMC: %f Diagnostico: %s\n",
                             ValorIMC, "OBESIDADE");
                else
                    if (ValorIMC <= 50)
                        printf("\n\n\nValor do IMC: %f Diagnostico: %s\n",
                                 ValorIMC, "OBESIDADE MORBIDA");
                    else
                        printf("\n\n\nValor do IMC: %f Diagnostico: %s\n",
                                ValorIMC, "SUPEROBESIDADE");
    return 0;
}
```

A modularização do programa levaria à construção das rotinas para calcular o valor do IMC e para determinar o diagnóstico correspondente. Uma possibilidade seria a apresentada a seguir.

```
#include <stdio.h>

float ValorIMC;

float IMC(float Peso, float Altura)
{
    return Peso / (Altura * Altura);
}

void DiagIMC()
{   if (ValorIMC < 18.5)
        printf("\n\n\nValor do IMC: %f Diagnostico: %s\n",
                  ValorIMC, "DESNUTRIDO");
    else
```

```
        if (ValorIMC <= 25)
            printf("\n\n\nValor do IMC: %f Diagnostico: %s\n",
                    ValorIMC, "NORMAL");
        else
            if (ValorIMC <= 30)
                printf("\n\n\nValor do IMC: %f Diagnostico: %s\n",
                         ValorIMC, "SOBREPESO");
            else
                if (ValorIMC <= 40)
                    printf("\n\n\nValor do IMC: %f Diagnostico: %s\n",
                             ValorIMC, "OBESIDADE");
                else
                    if (ValorIMC <= 50)
                        printf("\n\n\nValor do IMC: %f Diagnostico: %s\n",
                                 ValorIMC, "OBESIDADE MORBIDA");
                    else
                        printf("\n\n\nValor do IMC: %f Diagnostico: %s\n",
                                ValorIMC, "SUPEROBESIDADE");
}

int main(void)
{   float vPeso, vAltura;

    printf("Informe o peso:\n");
    scanf("%f", &vPeso);
    printf("\nInforme a altura:\n");
    scanf("%f", &vAltura);

    ValorIMC = IMC(vPeso, vAltura);

    DiagIMC();

    return 0;
}
```

Nesta versão, a rotina principal ficou mais ‘limpa’ dos detalhes do processamento, retendo apenas a estrutura geral do processamento. A função `IMC` está adequadamente projetada, pois executa uma tarefa bem definida e recebe os dados necessários por meio de parâmetros formais e devolve seu resultado por meio do valor de retorno explícito da função. A rotina `DiagIMC`, por outro lado, é pouco coesa, já que tanto determina o diagnóstico como também cuida de sua exibição na tela. Além disso, como agora utilizamos uma variável global para o valor numérico do IMC, há um acoplamento de conteúdo entre a rotina principal e `DiagIMC`, pois ambas utilizam explicitamente a mesma variável. Uma maneira de melhorar a qualidade da rotina `DiagIMC` poderia ser a apresentada a seguir.

```
#include <stdio.h>

float IMC(float Peso, float Altura)

{   return Peso / (Altura * Altura);

}


void DiagIMC(float VrIMC, char *diag)

{   if (VrIMC < 18.5)

        strcpy(diag, "SUBNUTRIDO");

    else

        if (VrIMC <= 25)

            strcpy(diag, "NORMAL");
```

```
        else
            if (VrIMC <= 30)
                strcpy(diag, "SOBREPESO");
            else
                if (VrIMC <= 40)
                    strcpy(diag, "OBESIDADE");
                else
                    if (VrIMC <= 50)
                        strcpy(diag, "OBESIDADE MÓRBIDA");
                    else
                        strcpy(diag, "SUPEROBESIDADE");
}

int main(void)
{   float vPeso, vAltura, ValorIMC;
    char diagnostico[20];

    printf("Informe o peso:\n");
    scanf("%f", &vPeso);
    printf("\nInforme a altura:\n");
    scanf("%f", &vAltura);

    ValorIMC = IMC(vPeso, vAltura);

    DiagIMC(ValorIMC, diagnostico);

    printf("\n\n\nValor do IMC: %f Diagnostico: %s\n",
                ValorIMC, diagnostico);

    return 0;
}
```

Nesta versão do programa não existem variáveis globais, e a rotina `DiagIMC` não faz a impressão dos valores na tela, apenas determina qual o diagnóstico e o armazena no segundo parâmetro, que foi recebido por referência. Se a linguagem C permitisse diretamente o retorno de strings por parte de subrotinas, teríamos a situação ideal em termos de acoplamento. Do ponto de vista da coesão o programa agora está correto.