Estruturas de Dados

Engenharia Informática

1º Ano - 2º Semestre

Francisco Morgado
Escola Superior de Tecnologia e Gestão de Viseu

5. PROGRAMAÇÃO MODULAR

5.1 Conceito

5.2 Apontadores para funções

5.3 Apontadores para void

5.4 Módulos em C

5.1 Conceito

O paradigma da programação procedimental

Na implementação de aplicações informáticas de pequena e média complexidade utiliza-se o paradigma da ***programação procedimental*** que consiste em dividir um problema complexo num conjunto de problemas mais pequenos, de resolução mais simples que o problema inicial. Segue-se uma **abordagem descendente**, de **refinamento progressivo**.

O paradigma da programação procedimental pode enunciar-se do seguinte modo: "*decide os procedimentos que precisas e usa os melhores algoritmos possíveis*".

Este paradigma enfatiza as operações que é necessário efectuar sobre a estrutura de dados, modelando a solução do problema numa **perspectiva funcional** das operações, por vezes muito **dependente da estrutura de dados** escolhida.

Por outro lado, a estrutura de dados necessária para resolução de problemas complexos é por vezes difícil de gerir como um todo, necessitando de ser decomposta em diversas sub-estruturas especializadas.

O paradigma da programação modular

Progressivamente, a ênfase na implementação de software orientou-se para a organização de estruturas de dados cada vez mais complexas e especializadas, tendo sido proposta a metodologia de desenho estruturado que consistia na **decomposição funcional** de uma aplicação **num conjunto de módulos** bem estruturados, que cooperam para implementar a funcionalidade pretendida.

O **paradigma da programação modular** pode enunciar-se do seguinte modo: "*decide os módulos que precisas e decompõe o programa para que as estruturas de dados sejam encapsuladas nos módulos*" .

Nesta estratégia procede-se ao **encapsulamento da estrutura de dados** (*information hiding*): à abstracção dos algoritmos acrescenta-se a abstracção da estrutura de dados.

A **abstracção de dados** tem como objectivo **separar a definição** do módulo (a sua interface com o exterior) da sua **implementação**.

Ao possuir uma **estrutura de dados** interna (**independente do exterior**), o **módulo** é **mais versátil** e poderoso (mais usável) **que um subprograma**.

Potencialidades dos módulos

O módulo pode ser visto como um caixa preta que tem uma interface com o exterior, escondendo no entanto os pormenores, quer da estrutura de dados quer das operações que a manipulam.

- **Protecção da estrutura de dados** – inacessível a partir do exterior, a não ser pelas operações de manipulação disponibilizadas pelo módulo.
- **Virtualização das operações** disponibilizadas pelo módulo – podemos implementar vários algoritmos alternativos de processamento, através de implementações alternativas do módulo.

Os paradigmas de programação procedimental e modular COMPLEMENTAM-SE:

- **A modularidade decompõe a solução de um problema num conjunto de módulos**
- **A decomposição hierárquica analisa a funcionalidade de cada módulo com vista à sua implementação**

COESÃO E INTERLIGAÇÃO DOS MÓDULOS

A partição da solução de um problema em módulos deve assegurar a coesão de cada módulo e a sua interligação com os outros:

- **Um módulo deve ser completo e coerente com o propósito a que se destina - uma funcionalidade muito coesa;**
- **Um módulo deve ser o mais independente possível dos outros módulos, minimizando as dependências de informação entre módulos: a sua substituição não deve implicar alterações nos restantes módulos.**

5.2 Apontadores para funções

EXEMPLO 1

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

int quadrado (int x)
{
    return x * x;
}
int cubo (int x)
{
    return quadrado(x) * x;
}
int dobro (int x)
{
    return 2*x;
}
void escreve(int (*funcao)(int), int n)
{
    printf("\n<< %d >>\n", (*funcao)(n) );
}
```

```
void main()
{
   escreve(quadrado, 8);
   escreve(dobro   , 8);
   escreve(cubo    , 8);
   system("pause");
}
```

Caso se pretenda escrever o valor de funções reais com argumento inteiro, basta fazer:

```
void escreve(float (*funcao)(int), int n)
{
   printf("\n<< %.2f >>\n", (*funcao)(n) );
}
```

5.2 Apontadores para funções

EXEMPLO 2

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

int soma (int x, int y)
{
    return x+y;
}
void escreve2( int (*funcao)(int, int), int n, int m)
{
    printf("\n-> %d <-\n",  (*funcao)(n, m) );
}
```

```
void main()
{
    escreve2(soma, 3, 8);
    system("pause");
}
```

5.3 Apontadores para void

EXEMPLO 1

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

typedef struct
 {
   char nome[80];
   char bi[10];
 }Aluno, *ALUNO;
```

```
void escreve(void * pv)
{
   ALUNO a;
   a = (ALUNO) pv;
   puts(a->nome);
   puts(a->bi);
}
```

Apontador para qualquer tipo

```
void main()
{
  Aluno p;
  printf("\nQual o Nome? ");
  gets(p.nome);
  printf("\nQual o BI? ");
  gets(p.bi);

  escreve(&p);
  system ("pause");
}
```

EXEMPLO 2

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

typedef struct
 {
   char nome[80];
   char bi[10];
 }Aluno, *ALUNO;
```

```
void escreve(void * pv)
{
        ALUNO a;
        a = (ALUNO) pv;
        puts(a->nome);
        puts(a->bi);
}
```

Apontador para qualquer tipo

```
void main()
{
  ALUNO p =(ALUNO) malloc(sizeof(Aluno));
  printf("\nQual o Nome? ");
  gets(p->nome);
  printf("\nQual o BI? ");
  gets(p->bi);

  escreve(p);
  system ("pause");
}
```

5.4 Módulos em C

A linguagem C foi concebida tendo em mente facilitar a construção descentralizada de aplicações através da distribuição do código por diferentes ficheiros fonte. Por isso é necessário, mesmo em programas simples, recorrer à alusão a funções e definições feitas noutros ficheiros.

Para tornar mais rigorosa essa referência, foi criado o conceito de ficheiro de interface onde são colocadas todas as alusões a funções e definições, associadas a um certo tipo de funcionalidades.

Os ficheiros de interface distinguem-se dos ficheiros fonte propriamente ditos por terem a extensão **.h** em vez de **.c** (.**cpp** em C++)

Na linguagem C um módulo é composto por dois ficheiros:

- o ficheiro de **interface**, com a extensão **.h**
- o ficheiro de **implementação** com a extensão **.c** (.**cpp** em C++)

Para que uma aplicação utilize um módulo, o seu ficheiro de interface deve ser incluído nos ficheiros da aplicação, com a directiva *include*.

Em ambiente *Windows* (*Visual Studio*) os diferentes módulos são integrados no projecto.
Os ficheiros fonte têm a extensão .cpp e o ficheiro de interface a extensão .h

5.5 Aplicação - listas ligadas ordenadas

Ficheiros do projecto

- ***princ.cpp*** *-> ficheiro com código da parte principal do programa*
- ***modlista.cpp*** *-> módulo com funções para listas ligadas ordenadas genéricas*
- ***auxx.cpp*** *-> ficheiro com código das funções específicas para a lista pretendida*
- ***auxx.h*** *-> ficheiro com a definição da estrutura dos elementos da lista e com os protótipos das funções do aux.cpp*

```
#include <stdio.h>

typedef void *aptvoid;
typedef struct elemento
{
  char nome[80];
  char bi[12];
  int Numero;   //chave
  struct elemento *seg;
}*ELEMENTO;
```

Ficheiro ***auxx.h***

Contém a definição da estrutura dos elementos da lista e os protótipos das funções do auxx.cpp

```
void ler(aptvoid ele);
void ligar_xy(aptvoid x, aptvoid y);
void ligar_x_seg_y(aptvoid x, aptvoid y);
int compara(aptvoid x, aptvoid y);
ELEMENTO el_seg(aptvoid x);
int igual(aptvoid x, int ch);
void mostrar(aptvoid ele);
void inserir(aptvoid ele, aptvoid *cabeca);
aptvoid remover(int chave, aptvoid *cabeca);
```

Ficheiro *auxx.cpp*

Contém o código das funções específicas para a lista pretendida

```
#include "auxx.h"

void ler(aptvoid ele)
{
  ELEMENTO el=(ELEMENTO) ele;
  printf("\nQual o nome?");
  scanf("%s",el->nome);
  printf("\nQual o BI?");
  scanf("%s",el->bi);
  printf("\nQual o numero?");
  scanf("%d",&el->Numero);
}
```

```
void mostrar(aptvoid ele)
{
    int i=0;
    ELEMENTO el=(ELEMENTO) ele;
    while (el)
    {
     printf("\n nome:%s\t BI:%s",el->nome,el->bi);
     printf("\t\tNº de aluno: %d", el->Numero);
     el=el->seg;
    }
}
```

// continua

// continuação

Ficheiro ***auxx.cpp***

```
int compara(aptvoid x, aptvoid y)
{
    ELEMENTO a,b;
    a=(ELEMENTO)x;
    b=(ELEMENTO)y;
    return ((a->Numero) > (b->Numero));
}
```

```
int igual(aptvoid x, int ch)
{
    ELEMENTO a=(ELEMENTO)x;
     return (a->Numero == ch);
}
```

// continua

II continuação

```
void ligar_xy(aptvoid x, aptvoid y)
{
    ELEMENTO a=(ELEMENTO)x;
    ELEMENTO b=(ELEMENTO)y;
    a->seg=b;
}
```

Ficheiro ***auxx.cpp***

```
void ligar_x_seg_y(aptvoid x, aptvoid y)
{
    ELEMENTO a=(ELEMENTO)x;
    ELEMENTO b=(ELEMENTO)y;
    a->seg=b->seg;
}
```

```
ELEMENTO el_seg(aptvoid x)
{
    ELEMENTO a=
    (ELEMENTO)x;
    return a->seg;
}
```

Ficheiro **modlista.cpp**

Contém funções para listas ligadas ordenadas genéricas

```
void inserir(aptvoid ele, aptvoid *cabeca)
{   #include "auxx.h"
    int av=1; aptvoid ant,act;
    ligar_xy(ele, NULL);    //ele->seg=NULL;
    if (*cabeca == NULL)
  {
                ligar_xy(ele, (aptvoid)NULL);  //ele->seg=NULL;
                *cabeca=ele;
    }
    else
  {
                ant=act=*cabeca;
                while(av)
                if(act==NULL)  av=0;
                else
                        if (compara(act,ele))  // (act->Numero > ele->Numero)
                                av=0;
                        else {
                                ant=act;       act=el_seg(act); //act=act->seg;
                        }
                        if (act==*cabeca)
                                {
                                    ligar_xy(ele,*cabeca);  //ele->seg=*cabeca;
                                    *cabeca=ele;
                                }
                        else  {

                                        ligar_xy(ant,ele);      //ant->seg=ele;
                                ligar_xy(ele,act);      //ele->seg=act;
                        }
    }
}
```

II continuação

Ficheiro ***modLista.cpp***

```
aptvoid remover(int chave, aptvoid *cabeca)
{
    int av=1;
    aptvoid ret,ant,act;
    if (*cabeca==NULL)
                return NULL;
    ant=act=*cabeca;
    while(av)
                if(act==NULL)
                av=0;
                else if(igual(act,chave)) //(act->Numero == chave)
                                av=0;
                        else{
                                ant=act;
                                act=el_seg(act); //act=act->seg;
                            }
    if(act!=NULL)
   {
                ret=act;
                if(*cabeca==act)
                                *cabeca=el_seg(act);  //*cabeca=act->seg;
                else
                                ligar_x_seg_y(ant,act); //ant->seg=act->seg;
    }
    else
                ret=NULL;
    return ret;
}
```

Ficheiro ***princ.cpp***

Contém código da parte principal do programa

```
#include <stdio.h>

#include "auxx.h"

#include <windows.h>
```

```
char menu()
{
        char x;
        printf("\n Inserir Ficha ____(1)");
        printf("\n Retirar Ficha ____(2)");
        printf("\n Mostrar Fichas ___(3)");
        printf("\n Sair ___________(0)");
        do
        {
                printf("\nQual a opcao?");
                scanf(" %c",&x);
        }
        while (x<'0' || x>'3');
        return x;
}
```

// continua

II continuação

Ficheiro ***princ.cpp***

```
#define INSERIR ‘1’
#define REMOVER ‘2’
#define MOSTRAR ‘3’
#define SAIR ‘0’
```

```
void main()
{
char op; int nal;
ELEMENTO el;
aptvoid cabeca=NULL;
do {
 op=menu();
 switch (op)
 {
      case    INSERIR:el=(ELEMENTO)malloc(sizeof(elemento));
                  ler(el);
                  inserir( (aptvoid) el, &cabeca);
                break;
      case REMOVER:
                printf("\nNº aluno a remover? ");
                scanf("%d",&nal);
                el= (ELEMENTO) remover(nal,&cabeca);
                if (el)  {
                printf("\nRemoveu elemento de nome %s e BI %s",el->nome,el->bi);
                free(el);  }
                break;
        case MOSTRAR: mostrar(cabeca);  break;
        case SAIR:  exit(0);
   }
 }
 while (1);
 system("pause");
}
```

6. MACROS

6.1 Definição de Macros

6.2 Regras para colocação de parêntesis

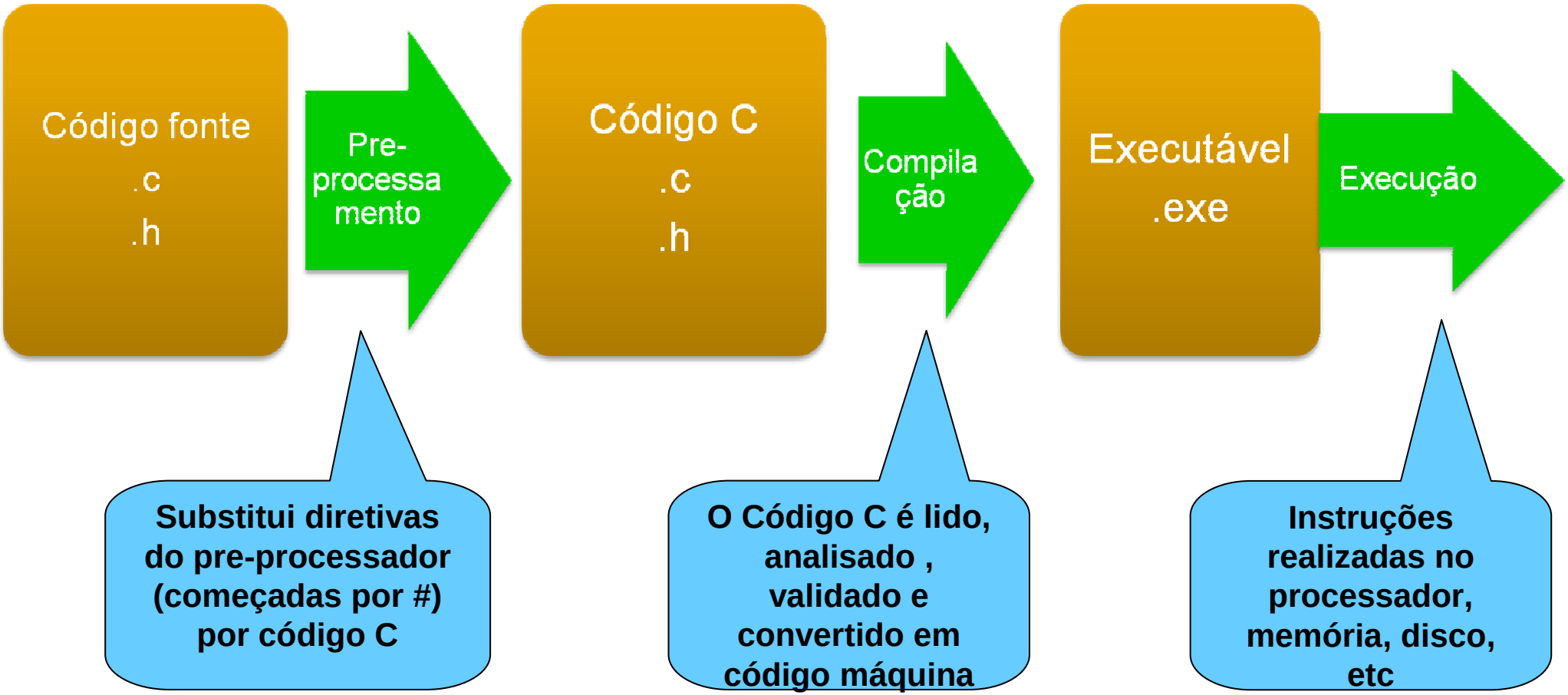
6.3 Operador de Continuidade

6.4 Operador #

6.5 Operador ##

6.6 Macros pré-definidas

Do código fonte à execução



6.1 Definição de Macros

As Macros são porções de código que são substituídas pelo pré-processador antes de o compilador passar pelo código.

Quando definimos **constantes simbólicas** estamos a usar Macros:

#define MAX 50

Este código não é C, mas uma ordem dada ao pré-processador para que, a partir desta linha, substitua todas as ocorrências de MAX pelo valor 50.

Mas o termo Macro não é normalmente associado à definição de constantes simbólicas. Refere-se, antes, a chamadas a "funções" que são substituídas por outro código pelo pré-processador.

EXEMPLO

Consideremos a Macro:

#define Mult(x,y) x*y

Na definição de uma Macro, a abertura do parêntesis dos parâmetros DEVE ficar imediatamente a seguir ao nome da macro.

Qual o resultado das seguintes secções de código?

a)

```
…
int x = 8, y = 3;
printf("%d * %d = %d", x, y, Mult(x,y) );
…
```

8 * 3 = 24

b)

```
…
int x = 1, y = 3;
printf("%d * %d = %d", x+2, y+1, Mult(x+2, y+1) );
…
```

3 * 4 = 8

Mult(x+2, y+1)
-> x+2*y+1

-> 1+2*3+1
-> 1+ 6 +1
-> 8

Substituindo a Macro anterior por:

#define Mult(x,y) (x)*(y)

tem-se

```
int x = 1, y = 3;
printf("%d * %d = %d", x+2, y+1, Mult(x+2, y+1) );
```

3 * 4 = 12

Mult(x+2, y+1)
-> (x+2)*(y+1)

-> (1+2)*(3+1)
-> 3 * 4
-> 12

Mas…

```
printf("%d", 1000/Mult(2+3, 7+3) );
```

2000

1000/Mult(2+3, 7+3)
-> 1000/(2+3)*(7+3)

-> 1000/(2+3)*(7+3)
-> 1000/5*10
-> 200*10
-> 2000

6.2 Regras para colocação de parêntesis

1. À volta de cada um dos parâmetros existentes na expansão da macro.
2. À volta de toda a macro

EXEMPLOS

EX1: Escrever a macro Max(a, b) que devolva o maior de dois valores dados

```
#define Max(a,b) ( (a)>(b) ? (a) : (b) )
```

EX2: Escrever a macro Abs(x) que devolva o valor absoluto de x

```
#define Abs(x) ( (x)>=0 ? (x) : -(x) )
```

6.3 Operador de Continuidade

- As macros DEVEM ser escritas numa única linha.
- Se houver necessidade de prolongar a macro por mais que uma linha, coloca-se, no fim de cada linha a continuar, o operador de continuidade das macros: \

EXEMPLO

```
#define Min(a,b) ( (a)<(b) ? \
 (a) : \
(b))
```

6.4 Operador # (quotation)

O operador #, usualmente chamado *quotation* fornece uma flexibilidade extra à escrita de macros.

Quando se encontra dentro de um dos parâmetros da macro, faz com que a expansão desse parâmetro seja realizada entre aspas, isto é convertida em *string*.

EXEMPLO

```
#define Escreve(x) printf(#x)
```

Ao ser invocado com o parâmetro

```
Escreve(\nManuel “” Maria)
```

a expansão gerada é

```
printf(“\nManuel \“\” Maria”)
```

6.5 Operador ## (concatenation)

O operador ##, usualmente chamado *concatenation* fornece outra flexibilidade extra à escrita de macros.

Concatena os dois argumentos (um antes do ## e outro depois) .

EXEMPLO

```
#define VarName(x) var ## x
```

Ao ser invocado com o parâmetro

VarName(1)

a expansão gerada é

var1

6.5 Macros pré-definidas

A norma ANSI define a obrigatoriedade de existir nos pré-processadores (compiladores) um conjunto de macros pré-definidas.

Estas macros têm dois *underscores* (__) à esquerda e outros dois à direita, para evitar que sejam acidentalmente redefinidas pelo programador.

As macros pré-definidas são:

__DATE__	***string*** **no formato "Mmmm dd yyyy"**
__TIME__	***string*** **no formato "hh:mm:ss"**
__LINE__	**valor numérico com o valor da linha actual**
__FILE__	***string*** **com o nome do ficheiro actual**