Linguagens e Ambientes de Programação (2018/2019)

Teórica 13 (17/abr/2019)

Apontadores: Criação dinâmica de variáveis.
Compilação separada e modularidade.
Módulos abertos em C.
Módulos fechados em C.
Listas ligadas em C. Usando recursividade no processamento de estruturas com apontadores.

Apontadores: Criação dinâmica de variáveis

A linguagem C não dispõe dum gestor de memória automático. Criação e a eliminação de variáveis dinâmicas são responsabilidades do programador.

Usa-se a função de biblioteca `malloc` para criar e eliminar variáveis dinâmicas. A função `malloc` recebe o número de bytes do bloco de memória a reservar e retorna um apontador (de tipo `void *`) para o início desse bloco. Eis o cabeçalho desta função, tal como está definido no módulo `stdlib`:

```
void *malloc(size_t size);
```

Exemplo de utilização:

```
#include <stdlib.h> /* declara a função malloc, entre muitas outras coisas */

const int *a = malloc(10 * sizeof(int));   /* cria var. dinâmica; o respetivo apontador é guardado na constante a */
a[3] = 10;                                 /* altera uma célula do bloco de memória apontado (usando notação de vetor) */
```

A função `malloc` retorna a constante `NULL` no caso do gestor de memória não ter mais memória disponível.

Para libertar a memória ocupada por uma variável dinâmica que já não é mais necessária, usa-se a operação:

```
void free(void *ptr);
```

Interessa criar variáveis dinâmicas principalmente nas duas seguintes situações:

- Para criar um vetor, cujo tamanho só é conhecido depois do programa começar a correr.
- Para criar estruturas de dados dinâmicas, e.g. listas e árvores.

Mais abaixo, mostra-se como se manipulam com listas em C. Uma lista é uma estrutura de dados dinâmica que só pode ser manipulada através de apontadores. Para organizar bem o nosso exemplo, vamos apresentá-lo sob a forma de módulo. Mas primeiro temos de aprender a lidar módulos em C...

Compilação separada e modularidade

A linguagem C suporta modularidade e compilação separada. O controlo de visibilidade de nomes é feito ao nível do ficheiro.

Controlo de visibilidade

Por omissão, todas as variáveis globais e funções definidas num ficheiro são públicas, isto é podem ser acedidas a partir de outros ficheiros. Consegue-se impedir esse acesso precedendo a definição dessas entidades pela palavra **`static`**.

```
static Vetor privado;
static int f(int x) { return x + 1; }
```

Antes de se poder aceder a uma entidade definida noutro ficheiro é preciso declarar essa entidade para o compilador a ficar a conhecer. Isso faz-se usando a palavra reservada **`extern`**. No caso da declaração das funções o atributo **`extern`** pode ser omitido, pois o compilador vê que a declaração não tem corpo e assume que se trata duma entidade externa.

```
extern char key;
extern int f(int x);    /* no caso das funções, a palavra extern pode ser omitida */
```

Módulo

Jogando de forma disciplinada com os mecanismos de visibilidade atrás descritos, é possível implementar **módulos** em C.

Um módulo "fich" é tipicamente definido usando dois ficheiros:

- Um **ficheiro de interface** "fich.h";
- um **ficheiro de implementação** "fich.c".

No ficheiro de interface definem-se todas as entidades exportadas pelo módulo, o que pode incluir: funções, variáveis globais, tipos e constantes.

O cliente do módulo só precisa fazer

```
#include "fich.h"
```

para ter acesso à definição dos símbolos exportados.

Módulos abertos em C

Apresenta-se uma implementação de listas ligadas, feita num módulo aberto em C. Estas listas são homogéneas e decidimos que vão conter valores de tipo `double`.

Este módulo diz-se **aberto** porque a representação das listas é pública. Veremos na secção seguinte como ocultar essa representação.

Definimos um pequeno conjunto de operações, só para exemplificar.

Ficheiro LinkedList.h

A definição do símbolo `_LinkedList_`, protege o ficheiro de interface relativamente à possibilidade da inclusão múltipla do mesmo ficheiro.

Uma **lista ligada** consiste numa sequência de nós. Cada nó contém um valor dum dado tipo Data e um apontador indicando qual o nó seguinte. O valor NULL serve para indicar que não existe nó seguinte.

É interessante comparar a utilização de listas face à utilização de vetores, para guardar sequências de valores:

- A vantagem duma lista relativamente um vetor é o facto da sucessão de nós não ter de estar em posições contíguas de memória. Isso permite a inserção e remoção de valores em tempo constante (sem ser preciso estar a desviar parte do conteúdo para a frente).
- A desvantagem duma lista relativamente um vetor é não ser possível implementar acesso indexado em tempo constante.

No ficheiro ".h", as duas primeiras linhas servem para garantir que não haverá problema se o ficheiro for acidentalmente incluído duas ou mais vezes.

```
#ifndef _LinkedList_
#define _LinkedList_

#include <stdbool.h>

typedef double Data;
typedef struct Node {
    Data data;
    struct Node *next;
} Node, *List;  /* Uma lista e' um apontador para um no' */

typedef bool BoolFun(Data);

List listMakeRange(Data a, Data b);
int listLength(List l);
bool listGet(List l, int idx, Data *res);
List listPutAtHead(List l, Data val);
List listPutAtEnd(List l, Data val);
void listPrint(List l);
List listFilter(List l, BoolFun toKeep);
void listTest(void);
#endif
```

Ficheiro LinkedList.c

Repare que a função auxiliar `NewNode` é declarada como estática para ficar privada ao módulo. A palavra reservada `static`, quando aplicada a uma função ou a uma variável global, torna essas entidades privadas: portanto o **âmbito** da sua definição é o ficheiro onde essa definição ocorre.

Repare que todas as funções deste módulo estão programadas com base em raciocínios imperativos, sem usar recursividade. Essa é a forma normal de trabalhar em C.

Técnica do apontador para o último nó

A função `listMakeRange` ilustra uma técnica importante.

É sempre muito fácil fazer crescer as listas acrescentando nós novos à cabeça. Mas usando a técnica exemplificada, consegue-se criar uma nova lista acrescentando os novos nós no final. Repare que se usa de forma habilidosa um apontador auxiliar `last` que aponta sempre para o último nó da lista. Tome nota desta técnica, porque precisará de a usar muitas vezes.

Técnica do apontador atrasado

A função `listFilter` ilustra outra técnica importante.

De forma geral, as funções que apagam ou inserem nós precisam de alterar o campo next do nó anterior. Realmente, quando se descobre qual o ponto de inserção, o nó anterior já ficou para trás. Por isso, neste tipo de função, convém navegar na lista usando um apontador atrasado, que aponta para o nó que antecede aquele em que estamos interessados em cada momento.

Mas, levanta-se então o problema de como tratar o primeiro nó da lista, visto que ele não tem antecedente. A nossa solução foi inserir um nó auxiliar (com nome `dummy`) na primeira posição.

Recomendação geral para a escrita de funções sobre listas

As funções sobre listas costumam ser complicadas por envolverem vários detalhes de gestão dos apontadores.

Em primeiro lugar, é importante ir fazendo um desenho em papel para apoiar o raciocício.

Não interessa começar por escrever a inicialização! Interessa sim começar por escrever o ciclo, imaginando que vamos a meio dele, e numa primeira fase podem-se assumir algumas simplificações que facilitem a escrita duma primeira versão. Depois é preciso refinar o ciclo para tratar todas as situações possíveis. Finalmente, agora que já percebemos as necessidades do ciclo, podemos escrever a sua inicialização e finalização.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "LinkedList.h"

static List newNode(Data val, List next)
{
    List n = malloc(sizeof(Node));
    if( n == NULL )
        return NULL;
    n->data = val;
    n->next = next;
    return n;
}

List listMakeRange(Data a, Data b)
{  // TECNICA ESSENCIAL: Ir fazendo crescer a lista no ultimo no'.
    if( a > b )
        return NULL;
    double i;
    List l = newNode(a, NULL), last = l;
    for( i = a + 1 ; i <= b ; i++ )
        last = last->next = newNode(i, NULL);
    return l;
}

/* Outra maneira, mais palavrosa, de escrever a funcao anterior:

List listMakeRange(Data a, Data b)
{
    if( a > b )
        return NULL;
    double i;
    List l = newNode(a, NULL);
    List last = l;
    for( i = a + 1 ; i <= b ; i++ ) {
        List q = newNode(i, NULL);
        last->next = q;
        last = q;
    }
    return l;
}
*/

int listLength(List l) {
    int count;
    for( count = 0 ; l != NULL ; l = l->next, count++ );
    return count;
}

bool listGet(List l, int idx, Data *res)
{
```

```
    int i;
    for( i = 0 ; i < idx && l != NULL ; i++, l = l->next );
    if( l == NULL )
        return false;
    else {
        *res = l->data;
        return true;
    }
}

List listPutAtHead(List l, Data val)
{
    return newNode(val, l);
}

List listPutAtEnd(List l, Data val)
{
    if( l == NULL )
        return newNode(val, NULL);
    else {
        List p;
        for( p = l ; p->next != NULL ; p = p->next ); // Stop at the last node
        p->next = newNode(val, NULL);  // Assign to the next of the last node
        return l;
    }
}

List listFilter(List l, BoolFun toKeep)
{  // TECNICA ESSENCIAL: Adicionar um no' auxiliar inicial para permitir tratamento uniforme.
      // Tente fazer sem o no' suplementar e veja como fica muito mais complicado.
    Node dummy;
    dummy.next = l;
    l = &dummy;
    while( l->next != NULL )
        if( toKeep(l->next->data) )
            l = l->next;
        else {
            List del = l->next;
            l->next = l->next->next;
            free(del);
        }
    return dummy.next;
}

void listPrint(List l)
{
    for( ; l != NULL ; l = l->next )
        printf("%lf\n", l->data);
}

static bool isEven(Data data) {
    return (int)data % 2 == 0;
}

static bool isOdd(Data data) {
    return (int)data % 2 != 0;
}

void listTest(void) {
    List l = listMakeRange(1.1, 7.8);
    printf("----------\n");
    listPrint(l);
    printf("----------\n");
    l = listFilter(l, isEven);
    listPrint(l);
    printf("----------\n");
    l = listFilter(l, isOdd);
    listPrint(l);
    printf("----------\n");
}
```

Módulos fechados em C

Os módulos fechados em C baseiam-se na possibilidade de definir um tipo apontador para um tipo estrutura que ainda não foi definido, assim:

```
typedef struct Node *List;
```

Estão, onde é que finalmente se define o tipo `struct Node`? Somente no interior ficheiro LinkedList.c.

Ficheiro LinkedList.h

Para o módulo ficar opaco, altera-se apenas a definição do tipo List.

```
#ifndef _LinkedList_
#define _LinkedList_

#include <stdbool.h>

typedef double Data;
typedef struct Node *List; // so' mudou isto

typedef bool BoolFun(Data);

List listMakeRange(Data a, Data b);
int listLength(List l);
bool listGet(List l, int idx, Data *res);
List listPutAtHead(List l, Data val);
List listPutAtEnd(List l, Data val);
void listPrint(List l);
List listFilter(List l, BoolFun toKeep);
void listTest(void);
#endif
```

Ficheiro LinkedList.c

A definição do tipo passa para dentro do ficheiro ".c", mas sem a parte "*List", que já está presente no ficheiro ".h". O resto do ficheiro fica igual.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "LinkedList.h"

typedef struct Node {
    Data data;
    List next;
} Node;

...
```

Usando recursividade e método indutivo no processamento de estruturas com apontadores

Eis uma nova implementação do nosso módulo, agora usando recursividade e raciocínios indutivos, à maneira do OCaml.

A técnica indutiva tem a vantagem de dar origem a código mais legível, mas tem a desvantagem de ser pouco eficiente e fazer crescer muito a pilha de execução - listas longas causam sempre transbordo da pilha de execução. Note que, ao contrário do OCaml, a linguagem C não está otimizada para suportar recursividade de forma económica.

Portanto, a técnica recursiva é de evitar quando se processam listas.

Contudo, para processar árvores e outras estruturas não-lineares, já fará sentido usar recursividade em algumas das operações. Sem usar recursividade essas operações ficariam demasiado complicadas, além de que a complexidade espacial dos algoritmos recursivos sobre árvores é muitas vezes `log N` (pois depende da altura da árvore e não do seu tamanho), o que é bastante melhor do que a complexidade espacial linear típica das listas tratadas recursivamente.

Nos testes e exames serão pedidas explicitamente soluções imperativas, pelo que este estilo será considerado errado.

```
/* ESTE É UM EXEMPLO NEGATIVO. É má ideia programar listas
   ligadas em C desta forma, apesar de funcionar */

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "LinkedList.h"

static List newNode(Data val, List next)
{
    List n = malloc(sizeof(Node));
    if( n == NULL )
        return NULL;
    n->data = val;
    n->next = next;
    return n;
}

List listMakeRange(Data a, Data b)
```

```
{
    if( a > b )
        return NULL;
    else
        return newNode(a, listMakeRange(a+1, b));
}

int listLength(List l)
{
    if( l == NULL )
        return 0;
    else
        return 1 + listLength(l->next);
}

bool listGet(List l, int idx, Data *res)
{
    if( l == NULL )
        return false;
    else if( idx == 0 ) {
        *res = l->data;
        return true;
    }
    else
        return ListGet(l->next, idx-1, res);
}

List listPutAtHead(List l, Data val)
{
    return newNode(val, l);
}

List listPutAtEnd(List l, Data val)
{
    if( l == NULL )
        return newNode(val, NULL);
    else {
        l->next = listPutAtEnd(l->next, val);
        return l;
    }
}

void ListPrint(List l)
{
    if( l != NULL ) {
        printf("%lf\n", l->data);
        listPrint(l->next);
    }
}

List ListFilter(List l, BoolFun toKeep) {
    if( l == NULL )
        return NULL;
    else if( toKeep(l->data) ) {
        l->next = listFilter(l->next, toKeep);
        return l;
    }
    else
        return listFilter(l->next, toKeep);
}

void listTest(void) {
        // ...
}
```

Discussão da função `listFilter`

No módulo de listas ligadas, a função `listFilter` é a mais complicada. Resolvemos o problema usando um *apontador atrasado*. Eis a solução novamente:

```
List listFilter(List l, BoolFun toKeep)
{
    Node dummy;
    dummy.next = l;
    l = &dummy;
    while( l->next != NULL )
        if( toKeep(l->next->data) )
            l = l->next;
        else {
            List del = l->next;
            l->next = l->next->next;
            free(del);
        }
```

```
    return dummy.next;
}
```

Também se pode tentar resolver o problema sem inserir o nó auxiliar no início. Contudo, descobre-se que a solução fica bem mais complicada, especialmente porque surge o novo problema de descobrir qual vai ser a cabeça da lista resultado. Não gostamos desta solução, mas cá vai...

```
List listFilter2(List l, BoolFun toKeep)
{
    List res;
// Determina cabeça do resultado
    while( l != NULL && !toKeep(l->data) ) {
        List del = l;
        l = l->next;
        free(del);
    }
    res = l;
// Trata o resto da lista
    if( l != NULL && l->next != NULL ) {
                // este é o mesmo ciclo da primeira solução
        while( l->next != NULL )
            if( toKeep(l->next->data) )
                l = l->next;
            else {
                List del = l->next;
                l->next = l->next->next;
                free(del);
            }
    }
    return res;
}
```

Uma outra técnica possível, é usar um apontador diretamente para o campo next do nó que antecede aquele em que estamos interessados em cada momento. Note que se trata dum apontador para um apontador, pois o campo next contém um apontador.

Esta técnica tem a vantagem de não necessitar de nó artificial na primeira posição, visto que se pode iniciar o ciclo atribuindo ao apontador o endereço da variável que indica a cabeça da lista. Essa variável funciona como o campo next dum primeiro nó artificial imaginário.

Esta técnica tem a desvantagem de obrigar a trabalhar com apontadores para apontadores, o que complica os raciocínios.

```
List listFilter3(List l, BoolFun toKeep)
{
    List *p = &l;
    while( *p != NULL )
        if( toKeep((*p)->data) )
            p = &(*p)->next;
        else {
            List del = *p;
            *p = (*p)->next;
            free(del);
        }
    return l;
}
```

A conclusão é que a primeira solução, que usa um nó inicial auxiliar, parece ser a melhor.

#100