Programação Genérica em C

anr@di.uminho.pt

14.03.2012

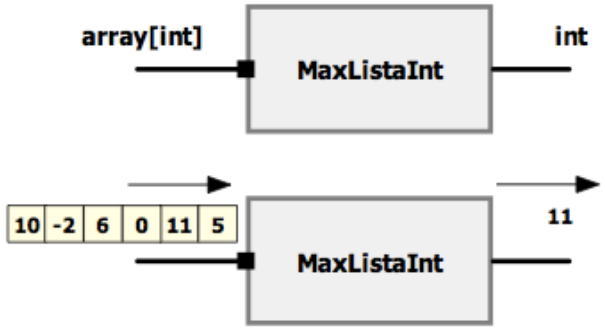
Objectivos

- compreender a necessidade de reutilizar definições mais complexos
  - não fazer repetidamente as mesmas soluções
- aumentar o grau de modularidade dos programas em C
- aproximar o modelo de programação do modelo da programação por objectos

Abstracção de controlo

- utilização de procedimentos e funções como mecanismos de incremento de reutilização
- não é necessário conhecer os detalhes do componente para que este seja utilizado
- procedimentos/funções são vistos como caixas negras (black boxes), cujo interior é desconhecido, mas cujas entradas e saídas são conhecidas

- por exemplo: ter uma função que dado um array de inteiros devolve o maior deles



- estes mecanismos suportam uma reutilização do tipo "copy&paste"
- a reutilização está muito dependente dos tipos de dados de entrada e saída

Módulos

- como forma de aumentar o grão da reutilização várias linguagens criaram a noção de **módulos**
- os módulos possuem declarações de dados e declarações de funções e procedimentos invocáveis do exterior
- possuem a (grande) vantagem de poderem ser compilados de forma autónoma
  - podem assim ser associados a diferentes programas

Tipos Abstractos de Dados

- os módulos para serem totalmente autónomos devem garantir que:
  - os procedimentos apenas acedem às variáveis locais ao módulo
  - não existem instruções de input/output no código dos procedimentos

Desenvolvimento em larga escala

- desta forma estamos a favorecer as metodologias de desenvolvimento para sistemas de larga escala
- Factores decisivos:
  - data hiding
  - implementation hiding
  - abstracção de dados
  - encapsulamento
  - independência contextual

Caso de Estudo

Lista Ligada

- Criação de um módulo de lista ligada que possa ser reutilizada com diferentes tipos de informação
- abstração de implementação da estrutura de dados (lista, array, etc.)
- abstracção dos dados e da sua alocação em memória

O que é particular em cada implementação?

- O tipo de dados de cada nodo?
  - não pode ser abstraído??
- a função de comparação dos dados
  - em implementações genéricas temos sempre de instanciar a forma como se comparam "coisas"

Como fazer?

- criar a implementação com base no tipo de dados (void *), isto é qualquer coisa
- como a implementação não sabe o que são dos dados que são passados, como é que aloca memória?
  - tem de ser fornecido o sizeof da estrutura de cada nodo
  - tem de ser passada a função de comparação

- 1ª abordagem

```
//--- ESTRUTURAS DE DADOS ----------------------------------------------------------

struct elemento {
    void *dados;
    struct elemento *proximo;
};

struct lista {
    size_t tamanho_dados;
    struct elemento *elementos;
};

//--- TYPEDEF ----------------------------------------------------------------------

typedef struct lista Lista;

//--- FUNCOES ----------------------------------------------------------------------

void inicia_sll(struct lista *,size_t);
int insere_cabeca_sll(struct lista *,void *);
int insere_ord_sll(struct lista *,void *,int (void *,void *));
int apaga_sll(struct lista *,void *,int (void *,void *));
void destroi_sll(struct lista *);
int procura_sll(struct lista ,void *,void *v,int (void *,void *));
void aplica_sll(struct lista ,void (void *));
void filtro_sll(struct lista *,void *,int (void *,void *));

//--- FIM --------------------------------------------------------------------------
```

- ainda se conhece a forma dos dados!

- 2ª abordagem

```
int init (int size, int (*compare)(void *, void *));
int insert (int handle, void * data);
int search (int handle, void *data);
int remove (int handle, void *data);
int clean (int handle);
```

- apenas se conhece a interface (API) e não se sabe mais nada da implementação

- o programa cliente apenas tem acesso a um handle que é o apontador para o início da lista

- o ficheiro .c começa com a definição da estrutura de dados

```
// Data type for each node
typedef struct {
    void *data;
    void *next;
} Node;

// Data Type for LL control
typedef struct {
    int sizeofData;
    int (*compare)(void *, void *);
    Node *root;
} LL_control;
```

- onde é que se implementa a função de comparação?
  - no programa cliente (programa teste)

```c
#include <stdio.h>
#include "LL_Generic.h"

typedef struct {
    int num;
    int idade;
    float taxa;
} My_Data;

int compara (void *a, void *b) {
    My_Data *aa, *bb;

    aa = (My_Data*) a;
    bb = (My_Data*) b;

    if (aa->num < bb->num) return -1;
    if (aa->num == bb->num) return 0;
    return 1;
}
```

- Vamos complicar um pouco mais...
- Este módulo pode gerir várias listas ligadas de "coisas" distintas
- Temos um único módulo/implementação para todas as listas de um programa

```
// maximum nbr of LL
#define NBR_LL 10

// data for controlling each LL
static LL_control control[NBR_LL];

// free pos in LL_control
static int next_LL=0;

// verify whether a handle is valid
static int valid_handle (int h){
    if (h>= next_LL) return 0;

    if (control[h].sizeofData==-1) return 0;

    return 1;
}
```

- init - cria uma lista

```
// returns -1 if fail.
// returns int (>=0) if success
// the returned value is the handle to identify the associated LL
int init (int size, int (*compare)(void *, void *)) {
    int handle, i;

    // discover free position on control
    for (i=0 ; i<next_LL ; i++) {
        if (control[i].sizeofData == -1) {
            handle = i;
            break;
        }
    }
    if (i==next_LL) {
        if (next_LL < NBR_LL) {
            handle = next_LL;
            next_LL++;
        }
        else
        {
            return -1;
        }
    }

    // init control
    control[handle].sizeofData = size;
    control[handle].compare = compare;
    control[handle].root = NULL;

    return handle;
}
```

- inserção ordenada

```
// returns 0 if failed, 1 otherwise
int insert (int handle, void * data) {
    void *data_ptr;
    Node *n, *act, *prev;

    // verify the handle is valid
    if (!valid_handle(handle)) return 0;

    // -



    // the new ...
    n = (Node*)malloc(sizeof(Node));
    if (!n) return 0;
    n->data = data_ptr;

    //search for insertion point
    act = control[handle].root ;
    prev = NULL;
    while ((act != NULL) && (control[handle].compare(n->data, act->data) > 0)) {
        prev = act;
        act = (Node *)act->next;
    }
    if (prev==NULL) {  // insertion on the 1st position
        control[handle].root = n;
    } else {
        prev->next = (Node *)n;
    }
    n->next = (Node *)act;

    return 1;
}
```

- procura

```
// searches for data whose key value is given on the 2nd parameter
// if found copies the associated data for the 2nd parameter
int search (int handle, void *data) {
    Node *act;
    int cmp;

    // verify the handle is valid
    if (!valid_handle(handle)) return 0;

    act = control[handle].root;

    while (act!=NULL) {
        cmp = control[handle].compare(data,act->data);
        if (cmp>0) {  // proceed on the list
            act = (Node *)act->next;
        } else if (cmp==0) {  // found!!!!
            memcpy (data, act->data, control[handle].sizeofData);
            return 1;
        } else {  // not found
            return 0;
        }
    }
    return 0;
}
```

- remoção

```
int remove (int handle, void *data) {
    Node *act, *prev;
    int cmp;

    // verify the handle is valid
    if (!valid_handle(handle)) return 0;

    prev = NULL;
    act = control[handle].root;

    while (act!=NULL) {
        cmp = control[handle].compare(data,act->data);
        if (cmp>0) {  // proceed on the list
            prev = act;
            act = (Node *)act->next;
        } else if (cmp==0) {  // found!!!!
            if (prev==NULL) {  // 1st element
                control[handle].root = (Node *) act->next;
            } else {
                prev->next = act->next;
            }
            free (act->data);
            free (act);
            return 1;
        } else {  // not found
            return 0;
        }
    }
    return 0;
}
```

- destruir (limpar) a lista

```
int clean (int handle) {
    Node *act, *rem;

    // verify the handle is valid
    if (!valid_handle(handle)) return 0;

    act = control[handle].root;
    while (act != NULL) {
        free (act->data);
        rem = act;
        act = (Node *)act->next;
        free (rem);
    }
    control[handle].root = NULL;
    control[handle].sizeofData = -1;
    return 1;
}
```

Como utilizar?

- inicialização da lista

```
LL = init (sizeof(My_Data), &compara);
if (LL==-1) {
    fprintf (stderr, "ERROR!\n");
    return;
}
```

- inserir um elemento

```
insert(LL, (void *)&d);
```

- remover um elemento

```
if (remove (LL, (void *)&d))
    printf ("Elemento %d removido!\n", d.num);
else
    printf ("Elemento %d NAO removido!\n", d.num);
```