Tipos abstratos de dados e encapsulamento

Prof. Fernando Vanini

IC-UNICAMP

Klais Soluções Ltda

Um exemplo

```
#include <stdio.h>
...
int main(...){
   ...
   FILE* arq = fopen("saida.txt","w");
   ...
   fprintf(arq,...);
   ...
   flose(arq);
   ...
}
```

Um exemplo

- Nesse exemplo
  - a biblioteca stdio cria o tipo abstrato de dados FILE
  - o arquivo 'stdio.h' contém apenas as definições necessárias para que uma aplicação consiga usar variáveis do tipo FILE e as operações associadas (como fopen(), fclose(), fprintf(), etc.)
  - os detalhes de implementação do tipo FILE e operações associadas ficam escondidos dos programas de aplicação que o utilizam.

Os conceitos

Uma biblioteca como stdio se baseia em conceitos que

- norteiam a construção da maioria das bibliotecas padrão de C.
- são aplicáveis aos módulos de quaisquer aplicações.

Os conceitos

- abstração
- encapsulamento
- generalização

Abstração

- *abstração*: “eliminação do irrelevante e a amplificação do essencial” (Robert C. Martin).
- em programação, *abstração* é usada para separar os conceitos essenciais do problema a ser resolvido dos detalhes relacionados unicamente com programação.

Encapsulamento

- Consiste em manter dentro de cada módulo sistema os detalhes que só dizem respeito ao próprio.
- Cada módulo só 'exporta' as definições necessárias à utilização dos serviços que ele oferece.
- Em C, as definições exportadas por uma biblioteca, também chamada de 'interface', normalmente ficam num arquivo com extensão '.h' (de *header file*).

Generalização

- Consiste em projetar cada módulo ou biblioteca de forma que seja aplicável ao maior número de situações possível.

- Exemplo: no lab2 foi desenvolvida uma biblioteca para ordenação que pode ser utilizada praticamente por qualquer tipo de dado para o qual se consiga definir uma função de comparação.

Aplicação

- Os conceitos de abstração, encapsulamento e generalização são aplicáveis às estruturas de dados estudadas até aqui
  - listas
  - pilhas
  - filas
  - árvores binárias de busca
  - grafos
  - tabelas de hashing
  - etc.

Um exemplo

- Biblioteca para trabalhar com listas
- Operações:
  - criar uma nova lista
  - inserir e remover elementos (no início e fim de uma lista)
  - contar o número de elementos
  - acesso a um elemento pela posição
  - liberar uma lista

A interface

```
/****** lists.h *******/
typedef void* listptr;
listptr newList();
int frontInsert(listptr list, void * elementptr);
int rearInsert(listptr list, void * elementptr);
void * removefirst(listptr list);
void * removelast(listptr list);
void * getElement(listptr list, int index);
int removeElement(listptr list, int index);
int freeList(listptr list,void (*freeNode)(void*));
int size(listptr list);
```

As funções

`listptr newList()`

- **cria uma nova lista**

`int frontInsert(listptr list, void * elementptr)`

- **insere um elemento no início de uma lista.**

`int rearInsert(listptr list, void * elementptr)`

- **insere um elemento ao final de uma lista.**

`void * removefirst(listptr list);`

- **remove o primeiro elemento de uma lista.**

`void * removelast(listptr list);`

- **remove o último elemento de uma lista.**

As funções

`void * getelement(listptr list, int index)`

- **devolve o elemento da lista, na posição definida por index. Devolve NULL se o elemento não existir.**

`void * removeelement(listptr list, int index)`

- **exclui elemento da lista,na posição definida por index. Devolve NULL se o elemento não existir.**

`int freelist(listptr list,void (*freeNode)(void*));`

- **libera uma lista e o seu conteúdo. A função freenode() é usada para liberar o conteúdo de cada nó. Devolve o número de elementos liberados.**

`int size(listptr list);`

- **devolve o tamanho (número de elementos) de uma lista.**

A implementação

```
 /****** lists.c *******/
#include 'lists.h'
typedef struct node * nodeptr;
typedef struct node{
           void* info;
           nodeptr next,prev;
         } node;

typedef struct list{
           int count;
           nodeptr content;
         } list;

listptr newList(){return (listptr)malloc(sizeof(list)); }
```

A implementação

```
int frontInsert(listptr list, void * elementptr){...}

void * removefirst(listptr list){...}

void * removelast(listptr list) {...}

int size(listptr list){...}

int freelist(listptr list void (*freeNode)(void*))
{...}
void * getelement(listptr list, int index){ ... }

int removeelement(listptr list, int index) { ... }
```

Outro exemplo

- Biblioteca para construir dicionários (ou mapas), baseados, p. ex. em árvore binária de busca
- os dados armazenados (na forma [chave,valor]) são definidos pela aplicação.
- as funções de comparação e liberação de conteúdo de um nó são definidas na criação do dicionário.

As funções

- criação de um mapa
- inserção de um par (chave,valor)
- remoção de um par a partir da chave
- busca pelo valor associado a uma chave
- liberação de um mapa e pares associados

A interface

```
/***** maps.h *****/
typedef struct map * mapptr;
nodeptr newmap(void (*freekey)(void*),
               void (*freevalue)(void*),
               int (*cmp)(void*,void*));
int size(nodeptr map);
void * search(mapptr map, void* key);
int insert(mapptr map, void* key, void* value);
void * removekey(mapptr map, void* key);
int freemap(mapptr map);
```

A implementação

- Um mapa pode ser descrito por uma estrutura do seguinte tipo:

```
typedef struct map {
     void (*freekey)(void*),    // liberação de uma chave
     void (*freevalue)(void*), // liberação de um valor
     int (*cmp)(void*, void*); // comparação de duas chaves
     int size;                 // número de pares no mapa
     nodeptr root;             // raiz da árvore (*)
   } map;
```

A implementação

- Se o mapa for baseado numa árvore binária de busca, cada nó da árvore pode ser do seguinte tipo

```
typedef struct node *nodeptr;
typedef struct node{
        nodeptr right;   // sub-árvore esquerda
        nodeptr left;    // sub-árvore direita
        nodeptr father;  // nó pai
        void * keyptr;   // chave
        void * valueptr; // valor
      } node;
```

Interface x implementação

- Nos dois exemplos, a interface contém apenas as definições necessárias ao uso das estruturas implementadas.
- A interface esconde, propositadamente, os detalhes da implementação.
- Para uso das bibliotecas, um programador precisa conhecer apenas a interface. Nenhum conhecimento da implementação é necessário.

Estado e comportamento

A estrutura usada para descrever um mapa contém dois tipos de informação

- dados que definem o 'estado' da estrutura
- funções (apontadores para) que definem o comportamento da mesma

```
typedef struct map {
        void (*freekey)(void*),
        void (*freevalue)(void*),
         int (*cmp)(void*, void*);
         int size;
         nodeptr root;
       } map;
```

O encapsulamento de dados e comportamento numa única entidade constitui um dos pilares da programação orientada a objetos.

Outro exemplo

- Uma aplicação precisa trabalhar com progressões numéricas de diversos tipos
  - progressões aritméticas
  - progressões geométricas
  - progressões específicas
- Os diversos tipos de progressão serão manipulados através de operações como
  - acesso ao valor inicial e valor atual da progressão
  - acesso ao n-ésimo valor de uma progressão
  - cálculo do 'próximo valor' da progressão
  - imprimir os n primeiros termos de uma progressão
  - cálcular a soma dos n primeiros termos de uma progressão
  - ...

Um descritor para progressões

- A idéia de se acessar uma estrutura através de um descritor para a mesma, usada nos exemplos anteriores para listas, é aplicável às progressões:
  - um descritor para progressões teria apontadores para as funções responsáveis pelas operações associadas às progressões, conforme mostrado a seguir.

Um descritor para progressões

```
#define PROGRESSION progressionPtr

typedef struct progression* progressionPtr;
typedef struct progression{
          int  (*first)(PROGRESSION);
          int  (*next) (PROGRESSION);
          int  (*valueAt)(PROGRESSION,int);
          int  (*currentValue)(PROGRESSION);
          void (*print)(PROGRESSION,int);
          int  startValue;
          int  curValue;
          int  ratio;
        } progression;
```

Progressões aritméticas

- Na criação do descritor p/ uma progressão aritmética é necessário
  - definir as funções associadas às operações
  - definir os valores para o valor inicial e a razão da progressão.

Progressões aritméticas

```
PROGRESSION newArithProgression(int startValue,
                                int ratio)
{
    PROGRESSION prog =
            (PROGRESSION)malloc(sizeof(progression));
    prog->first = arithFirst;
    prog->next  = arithNext;
    prog->valueAt = arithValueAt;
    prog->currentValue = arithCurrentValue;
    prog->print = arithPrint;
    prog->startValue = startValue;
    prog->ratio = ratio;
    prog->curValue = startValue;
    prog->ratio = ratio;
    return prog;
}
```

Progressões aritméticas

- Acesso ao primeiro valor e ao próximo:

```
int arithFirst(PROGRESSION prog){
    return prog->startValue;
}

int arithNext(PROGRESSION prog){
    prog->curValue += prog->ratio;
    return prog->curValue;
}
```

Progressões aritméticas

- Acesso ao valor corrente e ao n-ésimo valor da progressão:

```
int arithCurrentValue(PROGRESSION prog){
    return prog->curValue;
}

int arithValueAt(PROGRESSION prog, int n){
    int i;
    prog->first(prog);
    for(i=0; i < n; i++) prog->next(prog);
    return prog->curValue;
}
```

Progressões aritméticas

- Impressão dos n primeiros valores

```
void arithPrint(PROGRESSION prog, int n){
    int i;
    prog->first(prog);
    for(i = 0; i < n; i++){
      printf("%d, ",prog->curValue);
      prog->next(prog);
    }
    printf("\n");
}
```

Progressões aritméticas

- Exemplo de criação e uso de uma progressão aritmética:

```
...
PROGRESSION prog1 = newArithProgression(1,2);
prog1->print(prog1,10);
...
```

Progressões geométricas

- O conjunto de operações para uma progressão geométrica é exatamente o mesmo que o conjunto para progressões aritméticas.
- As operações *first, valueAt, currentValue* **e** *print* para progressões geométricas podem ser as mesmas definidas para progressões aritméticas.
- A única operação que precisa ser redefinida é *next*.

Progressões Geométricas

A operação next:

```
int geomNext(PROGRESSION prog){
    prog->curValue *= prog->ratio;
    return prog->curValue;
}
```

A criação de uma progressão geométrica:

```
PROGRESSION newGeomProgression( int startValue,
                                int ratio)
{
    PROGRESSION prog =
          newArithProgression(startValue,ratio);
    prog->next  = geomNext;
    return prog;
}
```

Progressões Geométricas

- Utilização:

```
...
PROGRESSION prog = newArithProgression(1,2);
prog->print(prog,10);
...
PROGRESSION prog = newGeomProgression(1,2);
prog->print(prog,10);
...
```

Progressões

- Propriedades
  - ***herança***: progressões geométricas reusam (herdam) a maioria das definições feitas para as progressões aritméticas.
  - ***polimorfismo***: o uso de uma progressão (como p. ex. em 'prog->print(prog)') é independente de a progressão ser aritmética, geométrica ou qualquer outro tipo que herde as propriedades da progressão aritmética.
- *Encapsulamento*, *herança* e *polimorfismo* constituem as bases da programação orientada a objetos.