Tipos abstratos de dados

Pilhas

Considere a implementação da pilha, onde o tipo do elemento é definido num arquivo de nome **`elemento.h`**.

```
#include "elemento.h"
#include <stdlib.h>
#define MAXN 1000

int topo;
static elemento * pilha;

/* inicia pilha */
void inicia_pilha () {
  topo = -1;
  pilha = (elemento *) malloc (MAXN*sizeof(elemento));
}

/* empilha novo elemento */
int empilha(elemento x) {
  if (topo == MAXN-1) return -1; /* não há mais espaço */
  topo++;
  pilha[topo] = x;
  return 0;
}

/* desempilha novo elemento */
int desempilha(elemento *x) {
  if (topo < 0) return -1; /* pilha vazia */
  *x = pilha[topo];
  topo--;
  return 0;
}
```

Para definir o tipo do elemento, vamos usar o `typedef` do `C`.

O `typedef` é apenas um modo de atribuirmos um sinônimo (em geral um nome mais simples) a um tipo (em geral mais complexo) na linguagem C. O formato geral desta declaração é:

```
typedef <tipo> <nome>;
```

Atribui-se o nome `<nome>` ao tipo `<tipo>`.

Alguns exemplos para o conteúdo do arquivo `elemento.h`.

1) `elemento.h` definindo o elemento como um `int`.

```
typedef int elemento;
```

2) `elemento.h` definindo o elemento como uma `struct`.

```
typedef struct {int a; float b;} elemento;
```

3) `elemento.h` definindo o elemento como uma `struct` um pouco mais complexa.

```
typedef struct {
  char nome[30];
  char endereco[50];
  struct datanasc {
     char dia;
     char mes;
     int ano;
  };
  double salario;
} elemento;
```

Com isso, podemos usar as mesmas funções para elementos de tipos diferentes.
Ou seja, as funções podem ser escritas sem que se saiba o conteúdo dos elementos que elas vão manipular. O tipo e conteúdo dos dados serão definidos pelo particular programa principal que usar tais funções.
Dizemos então que construímos um tipo abstrato de dados chamado **pilha** que pode ser manipulado através das operações:

```
inicia_pilha
empilha
desempilha
```

Pilhas – outro conjunto de operações

Vejamos agora outro conjunto de operações de pilha.
Nesta implementação vamos supor que a pilha nunca enche (é suficientemente grande) e que antes de desempilhar deve-se perguntar se a pilha está vazia.

```
#include "elemento.h"
#include <stdlib.h>
#define MAXN 1000

int topo;
```

```
static elemento * pilha;

/* inicia pilha */
void inicia_pilha () {
  topo = -1;
  pilha = (elemento *) malloc (MAXN*sizeof(elemento));
}

/* verifica se pilha esta vazia
   retorna 0 se pilha vazia e 1 senão */
int pilha_vazia() {
   return (topo >= 0);
}

/* empilha novo elemento
   sem verificar erro de pilha cheia */
void empilha(elemento x) {
  pilha[++topo] = x;
}

/* desempilha novo elemento */
elemento desempilha() {
  return pilha[topo--];
}
```

Filas

O mesmo ocorre para a implementação de filas.
Podemos usar as mesmas funções para manipular filas cujos elementos são de tipos diferentes.
Dizemos então que construímos um tipo abstrato de dados chamado **fila** que pode ser manipulado através das operações:

```
inicia_fila
insere_fila
remove_fila
```

Fica como exercício escrever as funções acima manipulando um tipo abastrato de dados.

Tipos abstratos de dados - vantagens

Assim como nos exemplo acima, é bom quando desenvolvemos um conjunto de funções que manipulam certo tipo de dados, fazer com que estas mesmas funções possam manipular outros tipos de dados. Ou seja, conseguir escrever um código que seja reutilizável em situações parecidas sem modificações ou com um mínimo possível de adaptações.

A criação de novos tipos de dados compostos por tipos elementares é uma ferramenta bastante útil. Permite programar funções que manipulam os elementos de uma forma mais

estruturada. Neste caso, não se trata de programar as funções independentemente do tipo e sim facilitar a programação criando novos tipos mais orientados ao objeto que as funções manipulam.

Veja abaixo exemplos de funções que manipulam números complexos, conjuntos e listas ligadas.

Outro exemplo – operações com números complexos

Um número complexo é um par `(a,b)` representando o número `a+bi`.
Neste caso, como um complexo é sempre uma dupla de reais, o tipo não é tão abstrato. A estrutura de dados pode ser definida como:

```
struct numero_complexo {
   double re;
   double im;
};

typedef struct numero_complexo Complexo;

Complexo inicia_complexo (double a, double b) {
    Complexo t;
    t.re = a; t.im = b;
    return t;
}

double parte_real(Complexo z) {
    return z.re;
}

double parte_im(Complexo z) {
    return z.im;
}

Complexo mult_complexo (Complexo u, Complexo v) {
    Complexo t;
    t.re = u.re*v.re - u.im*v.im;
    t.im = u.re*v.im + u.im*v.re;
    return t;
}

Complexo soma_complexo (Complexo u, Complexo v) {
    Complexo t;
    t.re = u.re+v.re;
    t.im = u.im+v.im;
    return t;
}
```

```
void imprime_complexo(char tit[], Complexo t) {
   printf("\n%s %5.1lf + %5.1lf i", tit, parte_real(t),
parte_im(t));
}
```

As funções acima nos permitem agora trabalhar com números complexos, de forma mais simples e independente dos tipos de dados usados para compor cada número complexo. Veja por exemplo o programa principal abaixo.

```
int main() {
     Complexo a,b,c,d;
     a = inicia_complexo(1.0, 2.0);
     b = inicia_complexo(3.0, 4.0);
     c = soma_complexo(a,b);
     d = mult_complexo(c,a);
     imprime_complexo("soma = ", soma_complexo(a,b));
     imprime_complexo("mult = ", mult_complexo(a,b));
}
```

Mesmo exemplo de números complexos – sem o uso de typedef

Observe também que o mesmo programa poderia ser feito sem o uso do `typedef`.

```
#include <stdio.h>

struct numero_complexo {
   double re;
   double im;
};

struct numero_complexo inicia_complexo (double a, double b) {
    struct numero_complexo t;
    t.re = a; t.im = b;
    return t;
}

double parte_real(struct numero_complexo z) {
    return z.re;
}

double parte_im(struct numero_complexo z) {
    return z.im;
}
```

```
struct numero_complexo mult_complexo (struct numero_complexo
u, struct numero_complexo v) {
    struct numero_complexo t;
    t.re = u.re*v.re - u.im*v.im;
    t.im = u.re*v.im + u.im*v.re;
    return t;
}

struct numero_complexo soma_complexo (struct numero_complexo
u, struct numero_complexo v) {
    struct numero_complexo t;
    t.re = u.re+v.re;
    t.im = u.im+v.im;
    return t;
}

void imprime_complexo(char tit[], struct numero_complexo t) {
   printf("\n%s %5.1lf + %5.1lf i", tit, parte_real(t),
parte_im(t));
}


int main() {
     struct numero_complexo a,b;
     a = inicia_complexo(1.0, 1.0);
     b = inicia_complexo(1.0, 1.0);
     imprime_complexo("soma = ", soma_complexo(a,b));
     imprime_complexo("mult = ", mult_complexo(a,b));
}
```

Mesmo exemplo de números complexos – agora com o uso de ponteiros

Outra forma de fazer a mesma coisa é usar o tipo **`Complexo`**, como um ponteiro para a estrutura:

```
struct numero_complexo {
   double re;
   double im;
};

typedef struct numero_complexo * Complexo;
```

`Complexo` agora é um ponteiro. Quando declaramos uma variável deste tipo, estamos declarando apenas o ponteiro. É necessário então alocar espaço em memória para este novo elemento criado. A função abaixo realiza exatamente esta função:

```
Complexo inicia_complexo (double a, double b) {
    Complexo t;
    /* como t é um ponteiro, não a estrutura, é necessário
       alocarmos espaço para t */
    t = malloc(sizeof(struct numero_complexo));
    t->re = a; t->im = b;
    return t;
}

double parte_real(Complexo z) {
    return z->re;
}

double parte_im(Complexo z) {
    return z->im;
}

Complexo mult_complexo (Complexo u, Complexo v) {
    Complexo t;
    t = inicia_complexo(0.0, 0.0);
    t->re = (u->re)*(v->re) - (u->im)*(v->im);
    t->im = (u->re)*(v->im) + (u->im)*(v->re);
    return t;
}

Complexo soma_complexo (Complexo u, Complexo v) {
    Complexo t;
    t = inicia_complexo(0.0, 0.0);
    t->re = (u->re)+(v->re);
    t->im = (u->im)+(v->im);
    return t;
}
```

Já que é necessário usar o `inicia_complexo` podemos escrever a soma como:

```
Complexo soma_complexo (Complexo u, Complexo v) {
    Complexo t;
    t = inicia_complexo((u->re)+(v->re),(u->im)+(v->im));
    return t;
}
```

Ou mesmo:

```
Complexo soma_complexo (Complexo u, Complexo v) {
    return inicia_complexo((u->re)+(v->re),(u->im)+(v->im));
}
```

```
void imprime_complexo(char tit[], Complexo t) {
   printf("\n%s %5.1lf + %5.1lf i", tit, parte_real(t),
parte_im(t));
}
```

A forma de usar é a mesma do exemplo anterior:

```
int main() {
     Complexo a,b;
     a = inicia_complexo(1.0, 2.0);
     b = inicia_complexo(3.0, 4.0);
     imprime_complexo("soma = ", soma_complexo(a,b));
     imprime_complexo("mult = ", mult_complexo(a,b));
}
```

Conjuntos

Vamos definir um tipo **`Conjunto`** e as algumas funções que realizam operações com conjuntos.
Um conjunto pode ser representado por um vetor de elementos do tipo **item** e a quantidade **`n`** de elementos.

```
#include <stdio.h>
#include <stdlib.h>
#include "item.h"
#define MAXN 100

struct str_conjunto {
   int n;
   item elementos[MAXN+1]; /* não vamos usar elementos[0] */
};

typedef struct str_conjunto * Conjunto;
```

Como **`Conjunto`** é um ponteiro, é necessário alocar espaço para o elemento quando o declaramos. A função **`vazio()`** abaixo, aloca o espaço e inicia o conjunto com zero elementos.

```
/* devolve um conjunto vazio */
Conjunto vazio() {
     Conjunto t = malloc(sizeof(struct str_conjunto));
     t->n = 0;
     return t;
}

/* devolve o numero de elementos de a */
int n_elem(Conjunto a) {
```

```
  return(a->n);
}

/* devolve o i-esimo elemento de a */
item elem(int i, Conjunto a) {
  return(a->elementos[i]);
}

/* devolve 1 se x pertence ao conjunto a e 0 senão */
int pertence(item x, Conjunto a) {
   int i,k;
   k=n_elem(a);
   for (i=1;i<=k;i++)
       if (x == elem(i,a)) return 1;
   return 0;
}

/* acrescenta x em a */
void insere_elem(item x, Conjunto a) {
  (a->n)++;
  a->elementos[a->n] = x;
}

/* devolve o conjunto união de a e b */
Conjunto uniao(Conjunto a, Conjunto b) {
   Conjunto c;
   item y;
   int i,k;
   c = vazio();
   k=n_elem(a);
   for (i=1;i<=k;i++) insere_elem(elem(i,a),c);
   k=n_elem(b);
   for (i=1;i<=k;i++) {
       y = elem(i,b);
       if(!pertence(y, a)) insere_elem(y,c);
   }
   return c;
}

/* devolve o conjunto intersecção de a e b */
Conjunto inter(Conjunto a, Conjunto b) {
   Conjunto c;
   item y;
   int i,k;
   k = n_elem(a);
   c = vazio();
   for (i = 1; i <= k; i++) {
```

```
        y = elem(i,a);
        if(pertence(y, b)) insere_elem(y,c);
    }
    return c;
}

/* Imprime o conjunto a.
   Esta função, ao contrário das outras é dependente do tipo
   dos elementos (int), pois é necessário dar o formato dos
   elementos no printf */
void imprime_conjunto(char nome[], Conjunto a) {
    int i,k;
    printf("\n%s = {", nome);
    k = n_elem(a);
    for (i = 1; i <= k; i++) printf("%5d", elem(i,a));
    printf("}");
}

/* exemplo de programa principal */
int main() {
    Conjunto c1,c2,d1,d2;

    c1=vazio();
    c2=vazio();
    d1=vazio();
    d2=vazio();

    insere_elem(0,c1);
    insere_elem(2,c1);
    insere_elem(1,c2);
    insere_elem(3,c2);
    insere_elem(0,d1);
    insere_elem(2,d1);
    insere_elem(0,d2);
    insere_elem(1,d2);

    imprime_conjunto("c1",c1);
    imprime_conjunto("c2",c2);
    imprime_conjunto("c1 U c2", uniao(c1,c2));
    imprime_conjunto("c1 ^ c2", inter(c1,c2));

    imprime_conjunto("d1",d1);
    imprime_conjunto("d2",d2);
    imprime_conjunto("d1 U d2", uniao(d1,d2));
    imprime_conjunto("d1 ^ d2", inter(d1,d2));
}
```

Lista Ligada

Cada elemento de uma lista possui um campo de informação que pode ser do tipo abstrato **item**.

```
typedef struct ElementoLL * link;
struct ElementoLL {
   item info;
   link prox;
};

/* função que cria um novo elemento com info=x,
   devolvendo ponteiro para o mesmo */
link CriaElementoLL (item x) {
  link t = (link) malloc (sizeof(struct ElementoLL));
  t->info = x;
  t->prox = NULL;
  return t;
}

/* Função que insere elemento no início da LL. O ponteiro
   para o início da lista é um parâmetro de saída */
void InsereElementoLL(link *p, item x) {
   link t = CriaElementoLL(x);
   t->prox = *p;
   *p = t; /* altera o ponteiro de início da LL */
}

/* Idem à função anterior, devolvendo o novo início da LL */
link InsereElementoLL(link p, item x) {
   link t = CriaElementoLL(x);
   t->prox = p;
   return t;
}

/* Função que procura o primeiro elemento com info=x.
   Devolve ponteiro para este elemento.
   Neste caso como há comparação, item tem que ser um
   tipo elementar (char, int, double, etc.) */
link ProcuraElementoLL(link p, item x) {
   link t = p;
   while (t != NULL) {
      if (t->info == x) return t;
      t = t->prox;
   }
   return NULL;
}
```

Exercícios

1) Usando a definição de conjunto do exemplo acima, faça as seguintes funções:

`int contido (Conjunto a, Conjunto b)` – que devolve `1` se `a` está contido em `b` e `0` caso contrário.

`int contem (Conjunto a, Conjunto b)` – que devolve `1` se `a` contem `b` e `0` caso contrário.

`Conjunto dif (Conjunto a, Conjunto b)` - devolve `a` diferença entre `a` e `b`.

2) Suponha agora que a definição de Conjunto não seja mais um ponteiro para uma estrutura e sim a própria estrutura:

```
typedef struct str_conjunto Conjunto;
```

Refaça as funções do exemplo acima e refaça também as funções do exercício 1.

3) Supondo agora que queremos conjuntos com inteiros somente. Vamos simplificar a definição. O elemento de índice zero contem o numero de elementos do conjunto. Assim a definição fica:

```
struct stconj {
   int elem[MAXN];
}

typedef  stconj * Conjunto;
```

Refaça as funções do exemplo acima e refaça também as funções do exercício 1.

4) A rigor nem é preciso ser ter a **`struct`**. Basta um ponteiro para um vetor que é alocado dinamicamente. Desta forma a definição ficaria:

```
typedef int * Conjunto;
```

A função **`Vazio()`** ficaria:

```
Conjunto Vazio () {
   Conjunto t = malloc(MAXN * sizeof(int));
   t[0] = 0; /* quantidade de elementos do conjunto */
   return t;
}
```

Refaça as funções do exemplo acima e refaça também as funções do exercício 1.

5) Um polinômio pode ser representado pela seguinte estrutura (supondo polinômios de no máximo grau MAX:

```
struct pol {
 int G; /* grau do polinômio */
 double Coef[MAX]; /* coeficientes – Coef[k] = coef. de x^k */
}

typedef struct pol * Polinomio;
```

Usando essa definição, faça as seguintes funções:

a) **`Polinomio somaPol (Polinomio A, Polinomio B)`** que devolve o polinômio soma de **`A`** e **`B`**.
b) **`Polinomio multPol (Polinomio A, Polinomio B)`** que devolve o polinômio produto de **`A`** e **`B`**.

6) Supondo agora a definição sem guardar o grau. Ou seja, todos os coeficientes devem ser considerados. Se o polinômio tem grau **`k`**, todos os coeficientes a partir do **`(k+1)`**ésimo são iguais a zero.

```
struct pol {
 double Coef[MAX]; /* coeficientes – Coef[k] = coef. de x^k */
}

typedef struct pol * Polinomio;
```

Refaça as funções solicitadas no exercício anterior.