IMPLEMENTAÇÃO EM C DE ABSTRACÇÕES DE DADOS USANDO MÓDULOS DE DADOS

TÉCNICA DOS TIPOS INCOMPLETOS ou OPACOS

F. Mário Martins, MIEI, 2016-2017

Como vimos anteriormente, um **módulo de dados** é uma implementação encapsulada, protegida, segura e robusta de um *tipo abstracto de dados* (**TAD**), ou seja, um tipo de dados que pode ser representado de muitas formas mas que deve obedecer a um conjunto de propriedades de comportamento bem definidas. Procurando clarificar o que vamos apresentar em seguida, chamaremos ao tipo abstracto de dados, **TAD**, e à sua representação na linguagem (neste caso C) tipo concreto de dados, **TCD**.

Módulo = Abstracção de Dados
Módulo = Interface + Implementação de Estrutura de Dados

- API: Application Programmer´s Interface
Operações que são acessíveis do exterior, ou seja, são tornadas **PÚBLICAS**;
- ERROS: Apenas o código interior ao módulo pode provocar erros nos dados (Sherlock Holmes tem agora a vida muito facilitada);
- ABSTRACÇÃO: a utilização do módulo não obriga (antes pelo contrário) ter que saber qual a representação interna, mas apenas a API; Black-Box de software;
- REUTILIZAÇÃO: módulo é independente

Em C, a criação deste tipo de módulos de dados requer algum esforço de programação de modo a que propriedades como encapsulamento, robustez, segurança, etc., possam ser garantidas.

Em C, a API é definida no ficheiro .h e a implementação no ficheiro .c. Assim, no ficheiro .h deveremos definir o essencial do **TAD**, e no ficheiro .c a sua implementação concreta, ou seja, o correspondente **TCD**.

Vamos tomar como exemplo a implementação de uma Árvore Binária de Procura (ABP), e apresentar algumas técnicas fundamentais para podermos em C criar um correcto **módulo de dados**, reutilizável (ainda que não genérico) e com encapsulamento (implementação escondida e acesso apenas via API).

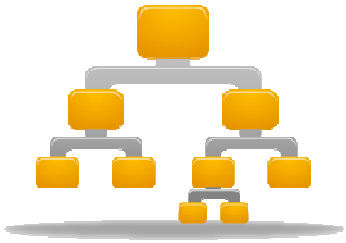
O nosso TAD é uma ABP. Uma ABP é uma estrutura não linear, **hierárquica**, formada por nós (ou **nodos**) dispostos segundo uma dada hierarquia e **ordem**.

Uma ABP é um conjunto finito de nodos que pode ser vazio ou pode ser particionado em três sub-conjuntos disjuntos: uma raiz e duas árvores binárias denominadas *sub-árvore da esquerda* e *sub-árvore da direita*.

Assim, cada nodo da árvore conterá a informação (também designada **chave ou valor**) a guardar na árvore, e as suas sub-árvores esquerda e direita.

As propriedades estruturais do tipo abstracto ABP são as seguintes:

- *A sub-árvore esquerda de um nodo contém apenas nodos cujos valores são inferiores ao valor do nodo;*
- *A sub-árvore direita de um nodo contém apenas nodos cujos valores são superiores ao valor do nodo;*
- *As sub-árvores direita e esquerda de um nodo são também ABP (recursividade);*
- *Não existem nodos duplicados (ou seja, valores duplicados).*



Será que já temos o nosso TAD ABP especificado? Não. Falta definir o *tipo dos valores dos nodos* e as *operações* que queremos definir sobre o TAD.

Vamos também dar um nome a este nosso TAD. Vamos chamar-lhe a partir de agora **TAD_ABP** (tipo abstracto de dados árvore binária de procura). Este é o tipo que os utilizadores irão usar através das operações que forem criadas na API do módulo que o vai implementar.

Vamos implementar em C este **TAD_ABP** usando um **TCD_ABP** (tipo concreto de dados árvore binária de procura).

Para este exemplo, vamos definir que as chaves ou valores são **strings**.

Quanto às operações, podemos desde já definir algumas operações comuns:

- Criar um novo TAD_ABP (ler criar uma nova árvore binária de procura);
- Inserir um novo valor no TAD_ABP;
- Verificar se existe um dado valor no TAD_ABP;
- Remover um dado valor do TAD_ABP;
- etc.

Especificado o TAD que pretendemos implementar, vamos passar de imediato à codificação em C tendo em atenção que necessitamos de criar os ficheiros .h e .c do módulo, a que chamaremos, por exemplo, **minhaABP.h** e **minhaABP.c**.

Uma questão muito importante antes mesmo de criarmos tais ficheiros, será pensarmos como vamos pretender que o nosso **TAD_ABP** seja usado, seja por nós mesmos ou por outros quaisquer clientes do módulo que vamos construir. Ou seja, **qual a sintaxe que vamos proporcionar na API do módulo para que a implementação da nossa abstracção seja usada.**

Antes mesmo de definirmos a própria API, vamos ver uma possibilidade e analisá-la.

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

.....
TAD_ABP abp1, abp2;
char* nome;
.................
/* Inicialização das ABP: de forma funcional */
abp1 = cria_ABP();
abp2 = cria_ABP();
.............
/* Operações diversas realizadas de forma funcional */
abp1 = insere_ABP(abp1, "Pedro Nuno");
abp2 = insere_ABP(abp2, "Rita Isabel");

.............
/* Delete */
delete_ABP(abp1);
delete_ABP(abp2);

......
```

A organização da informação pelos ficheiros C terá uma estrutura clara e uniforme para este tipo de preocupações de abstracção de dados e que é a seguinte:

Ficheiro `minhaABP.h`

- declaração do tipo de cada chave/valor de um nodo
- eventualmente declarar o tipo do nodo
- declaração abstracta da ABP, ou seja, do **TAD_ABP**

Ficheiro `minhaABP.c`

- #include **"minhaABP.h"**
- declaração do tipo do nodo (struct) e/ou do seu valor
- declaração do tipo concreto do TAD_ABP

Comecemos pelo ficheiro **`minhaABP.h`.**

A declaração do tipo chave/valor será feita escrevendo:

```
typedef char* ValorNodo;
```

A declaração abstracta da ABP esconderá dos utilizadores do módulo a implementação concreta pelo que não terão acesso à implementação.

Em C, é possível realizar a **declaração prévia** (no ficheiro **.h** portanto) de tipos que são apontadores para um tipo estruturado (baseado em **struct**). Tal é muito importante para a realização do encapsulamento de dados.

Assim, é possível ter no ficheiro **.h** uma declaração como:

```
typedef struct ABP_Nodo* TAD_ABP;
```

sem definir em tal ficheiro **.h** qual a estrutura de **ABP_Nodo**. Será apenas definida no respectivo ficheiro **.c**, daí o nome de **TIPO INCOMPLETO**.

O nosso tipo abstracto de dados, **TAD_ABP**, foi declarado no ficheiro .h como sendo um apontador para o seu TCD, um **ABP_Nodo**. **É esta a regra para se garantir em C "data hiding", ou seja, esconder dos utilizadores a implementação dos dados, garantindo assim que não lhes têm acesso directo.**

De facto, a estrutura **ABP_Nodo** não é declarada no ficheiro **.h**, pelo que assim se consegue esconder a verdadeira implementação de **ABP_Nodo**, que apenas será apresentada no ficheiro **.c**.

Assim, no ficheiro **minhaABP.h** teremos apenas, de momento, as declarações estruturais (de dados):

```
typedef char* ValorNodo;
typedef struct TCD_ABP* TAD_ABP;
```

Falta, naturalmente, declarar as funções que vamos tornar púbicas pela API.

Quanto ao ficheiro de implementação, **minhaABP.c** (que contém o TCD, ou seja, a verdadeira implementação do TAD), naturalmente que deverá realizar o **include** do ficheiro **minhaABP.h**, e, em seguida, concretizar o tipo **ABP_Nodo**.

Tal será feito definindo, por exemplo, os tipos:

```
typedef struct nodoABP {  // tipo do nodo da ABP
     ValorNodo valor;
     struct nodoABP* direito;
     struct nodoABP* esquerdo
} ABP_NODOT;

typdedef struct TCD_ABP {  // tipo concreto da ABP
     ABP_NODOT* raiz;
} TCD_ABP;
```

Finalmente, todas as funções a declarar no ficheiro .h devem possuir uma sintaxe que apenas refere o nome do TAD, neste caso, TAD_ABP, pois é o único tipo acessível do exterior (juntamente com o subtipo ValorNodo).

Teremos portanto, adicionalmente, em **minhaABP.h** as assinaturas (protótipos) das funções, tais como (seguindo um estilo funcional):

```
TAD_ABP cria_ABP();
TAD_ABP insere_ABP(TAD_ABP abp, ValorNodo valor);
TAD_ABP delete_ABP(TAD_ABP abp);
int existe_ABP(TAD_ABP abp, ValorNodo valor);
…
```

Finalmente, como nenhum utilizador sabe o que é um valor do tipo **ValorNodo**, pois também é relativamente opaco, necessitamos de uma função específica para criar um ValorNodo a partir de uma *string*. Terá a sintaxe:

```
ValorNodo cria_ValorNodo(char* string);
```

São exemplos comuns na literatura sobre tipos incompletos (opacos) em C declarações tais como:

```
typedef struct stack* Stack;
typedef struct contab* Contabilidade;
typedef struct myAbp* arvBinProcura;
```

Note-se finalmente que o posicionamento do * nestas declarações, sendo todas legais, correspondem apenas a uma preferência pessoal. Os puristas de C escreveriam certamente a última declaração apresentada como:

```
typedef struct myAbp *arvBinProcura;
```

Para o assunto em causa, tal é completamente irreleveante, tratando-se apenas de uma questão de estilo.