Encapsulamento em Programas C

Laboratórios de Informática III
Guião #3-2

Departamento de Informática
Universidade do Minho

Outubro de 2023

Conteúdo

1 Introdução **2**

2 O que é e para que serve? **2**

3 Estratégias de encapsulamento em C **2**
3.1 Estruturas opacas 2
3.2 Getters e Setters 3
3.3 Partilha de informação 5

4 Exercícios **7**

1 Introdução

O presente guião introduz o conceito de encapsulamento, complementar ao conceito de modularidade para o desenvolvimento de código robusto e de fácil manutenção. Passa primeiro por explicar o que é o encapsulamento e quais são os principais motivos para a sua adoção, sendo de seguida apresentadas algumas estratégias de encapsulamento em C.

2 O que é e para que serve?

O conceito de encapsulamento refere-se a **restringir o acesso direto aos dados apenas ao módulo que os controla**. Desta forma, qualquer módulo externo que pretenda alterar os dados de outro não o poderá fazer diretamente, tendo de recorrer ao uso da interface previamente definida.

O encapsulamento de módulos oferece várias vantagens, sendo algumas apresentadas de seguida:

- **Flexibilidade** – Ao omitir a definição dos dados da interface, é possível que diferentes implementações do mesmo módulo usem diferentes representações internas, permitindo uma maior flexibilidade na construção de módulos;
- **Robustez** – Ao garantir que o acesso aos dados é apenas da responsabilidade do módulo que os controla, sabemos que qualquer erro introduzido nesse módulo será causado por ele mesmo. Isto não só reduz a probabilidade de erros, mas também torna mais fácil a sua identificação e correção;
- **Manutenção** – Podemos no futuro alterar a representação interna dos dados e o seu comportamento sem ter de alterar os módulos externos que os usam.

3 Estratégias de encapsulamento em C

3.1 Estruturas opacas

A primeira estratégia de encapsulamento passa por omitir a definição das estruturas das interfaces (ficheiros *header*), de modo a não poderem ser acedidas diretamente por módulos externos. Na interface ficam apenas as respetivas declarações. Isto significa que estas estruturas opacas – também denominadas por estruturas incompletas – continuam a poder ser usadas externamente, mas qualquer acesso terá que ser controlado pela *Application Programming Interface* (API) fornecida, i.e., no cabeçalho.

Considere o seguinte excerto da estrutura abstrata de dados *Stack*:

```c
// stack.h
typedef struct stackNode {
    void* data;
    struct stackNode* next;
} StackNode;

typedef struct stack {
    int size;
    StackNode* top;
} Stack;

Stack* stackCreate();
void stackPush(Stack* stack, void* data);
void* stackPop(Stack* stack);
```

```c
// stack.c
#include "stack.h"

StackNode* nodeCreate(void* data) {...}
Stack* stackCreate() {...}
void stackPush(Stack* stack, void* data) {...}
void* stackPop(Stack* stack) {...}
```

Neste momento, temos que ambas as estruturas usadas, `StackNode` e `Stack`, encontram-se definidas na API da *Stack*. Apesar da *Stack* fornecer as funções de criação e modificação, nada garante que um módulo externo não aceda diretamente aos dados guardados. Acessos indevidos podem então causar erros de difícil identificação no comportamento da estrutura. Por exemplo:

```c
Stack* s = ...;
s->top = NULL; // stack passa a ter o nó de topo inválido, impossibilitando o correto
↪  funcionamento da estrutura
```

A solução passa por definir as estruturas junto com a implementação, deixando na API apenas as declarações. Neste caso, definimos as estruturas `StackNode` e `Stack` no `stack.c`. Para que o tipo `Stack` continue a ser usado externamente, mantemos no `stack.h` o respetivo `typedef`:[1]

```c
// stack.h
typedef struct stack Stack;

Stack* stackCreate();
void stackPush(Stack* stack, void* data);
void* stackPop(Stack* stack);
```

```c
// stack.c
#include "stack.h"

typedef struct stackNode {
    void* data;
    struct stackNode* next;
} StackNode;

struct stack {
    int size;
    StackNode* top;
};

StackNode* nodeCreate(void* data) {...}
Stack* stackCreate() {...}
void stackPush(Stack* stack, void* data) {...}
void* stackPop(Stack* stack) {...}
```

Com isto, o exemplo acima onde alterávamos diretamente o campo `top` não compilará:

```c
Stack* s = ...;
s->top = NULL; // error: dereferencing pointer to incomplete type 'Stack' {aka 'struct stack'}
```

3.2 Getters e Setters

Ainda que estruturas opacas escondam a implementação interna dos dados, poderá ser útil fornecer aos utilizadores do módulo uma forma de aceder e modificar certas propriedades.

Pegando no exemplo da *Stack*, é por vezes útil saber qual é o seu tamanho. Contudo, visto que esta informação encontra-se agora encapsulada na implementação, já não é possível obter-lo a partir de `stack->size`. Podemos, no entanto, adicionar à API uma função para retornar o seu tamanho, que deverá ser definida por todas as implementações da *Stack*. Estas funções que retornam o valor de uma propriedade de uma estrutura são designadas por *getters*, e geralmente têm o prefixo *"get"* no seu nome. Voltando ao exemplo da *Stack*, um *getter* para o `size` pode ser definido da seguinte forma:

```c
// stack.h
typedef struct stack Stack;

Stack* stackCreate();
void stackPush(Stack* stack, void* data);
void* stackPop(Stack* stack);
int getSize(Stack* stack);
```

```c
// stack.c
#include "stack.h"

...

int getSize(Stack* stack) {
    return stack->size;
}
```

Para além da leitura, também é útil por vezes alterar o valor de uma determinada propriedade de um módulo. Para isso, definimos na API do módulo funções *setters* para expor essa funcionalidade. Por norma, este tipo de funções são

[1] Notar que o `StackNode`, não sendo usado externamente, não precisa de estar na interface do módulo.

designadas com o prefixo *"set"*. Por exemplo, consideremos um módulo `Product` que implementa a funcionalidade de um produto de uma loja de retalho. Caso pretendamos fornecer ao utilizador do módulo uma opção para alterar o preço do produto, podemos definir um *setter* sobre este:

```c
// product.h
typedef struct product Product;

Product* newProduct(int id, char* name,
↪  int price, int quantity);
int getPrice(Product*); // não precisamos
↪  de colocar o nome na declaração
void setPrice(Product*, int);
```

```c
// product.c
#include "product.h"

struct product {
    int id;
    char* name;
    int price; // cents
    int quantity;
};
...
void setPrice(Product* product, int newPrice) {
    product->price = newPrice;
}
```

Uma vantagem importante dos *getters/setters* é que permitem facilmente alterar o comportamento de uma propriedade caso seja necessário no futuro. Por exemplo, consideremos novamente o módulo `Product`. Neste momento, o *getter* para o preço irá simplesmente retornar-lo tal como está representado internamente:

```c
// product.h
...
int getPrice(Product*);
```

```c
// product.c
...
int getPrice(Product* product) {
    return product->price;
}
```

Consideremos agora que queremos adicionar descontos ao preço de cada produto. Com esta estratégia, podemos simplesmente adicionar um campo ao produto contendo o desconto e alterar o respetivo *getter*. Desta forma, todos os módulos que usem o módulo `Product` obterão o preço real sem precisarem de qualquer modificação:

```c
// product.h
...
int getPrice(Product*);
```

```c
// product.c
#include "product.h"
#include <math.h> // ceil

struct product {
    ...
    int discount; // percentage
};

int getPrice(Product* p) {
    return ceil(p->price * ((100 - p->discount)
↪  / 100.0));
}
```

Caso a estrutura estivesse exposta, teríamos de alterar o cálculo do preço em todos os módulos que fizessem uso deste, aumentando a complexidade e podendo levar a que módulos considerassem valores diferentes para o mesmo produto, em caso de esquecimento.

Em suma, *getters/setters* permitem:

- Que um módulo controle a forma como os seus dados são acedidos;
- Agregar a lógica de acesso/modificação num único sítio, permitindo que futuras alterações fiquem auto-contidas no módulo.

3.3 Partilha de informação

Mesmo com estruturas opacas, podemos ter funções que expõem para o exterior referências mutáveis para dados pertencentes ao módulo, levando à quebra do encapsulamento. Desta forma, a partilha de informação – seja a informação que sai ou entra num módulo – também tem de ser feita de forma cuidada.

Consideremos novamente o exemplo do módulo `Product`, nomeadamente o *getter* e o *setter* para o id e o nome:

```c
// product.h
...
int getId(Product*);
void setId(Product*, int);
char* getName(Product*);
void setName(Product*, char*);
```

```c
// product.c
#include "product.h"
...
int getId(Product* product) {
    return product->id;
}
void setId(Product* product, int newId) {
    product->id = newId;
}
char* getName(Product* product) {
    return product->name;
}
void setName (Product* product, char* name) {
    product->name = name;
}
```

Começando pela função `getId`, podemos ver que esta retorna diretamente o valor `int` armazenado. Como vimos no primeiro guião, o programa irá colocar uma cópia do valor na *stack* e retornará da função. Desta forma, quem executar esta função não será capaz de alterar diretamente o valor armazenado. Da mesma forma, a função `setId` receberá por argumento uma cópia do novo id, sendo assim possível simplesmente atualizar o campo com o novo valor recebido. Em suma, não existe quebra de encapsulamento no campo `id`.

Olhando agora para a função `getName`, verificamos que esta funciona de forma semelhante à `getId`. No entanto, **ao retornar o apontador** para `name`, apenas o endereço do nome será copiado para a *stack*. O conteúdo em si será o mesmo, sendo assim possível alterar diretamente o campo `name`, **quebrando o encapsulamento**:

```c
Product* product = ...;
char* name = getName(product)
name[0] = '\0'; // quebra de encapsulamento, pois conseguimos alterar o valor diretamente
```

Da mesma forma, **também existe quebra de encapsulamento na função `setName`**, pois de momento estamos apenas a atualizar o apontador para o endereço recebido, **havendo assim novamente partilha de referências**. Um possível erro comum acontece quando libertamos a memória associada e o `Product` passa a referenciar um nome inválido:

```c
char* name = nextToken(...);
setName(product, name);
free(name); // name no product passa a ter valor inválido
```

Uma possível solução para este problema passa por copiar o conteúdo dos apontadores recebidos no módulo ou retornados pelo módulo. Para o exemplo acima, teremos os seguintes *getter* e *setter* para o nome:

```c
// product.h
...
char* getName(Product*);
void setName(Product*, char*);
```

```c
// product.c
#include "product.h"
#include <string.h> // strdup
...
char* getName(Product* product) {
    return strdup(product->name);
}
void setName(Product* product, char* name) {
    free(product->name); // libertar antigo
    product->name = strdup(name);
}
```

Outra solução consiste em usar a *keyword* `const` junto com o tipo. Com isto, estamos a indicar ao utilizador que se trata de um valor que não deverá ser modificado. No caso de existir uma modificação, o compilador avisará do erro.[2]

Para o exemplo acima, podemos ter o *getter* do nome a retornar um `const char*`. Da mesma forma, podemos ter o *setter* a receber um `const char*`. Isto indica ao utilizador do *setter* que a sua *string* não será modificada pelo `Product`. É de notar que a cópia precisa na mesma de existir no *setter*:

```c
// product.h
...
const char* getName(Product*);
void setName(Product*, const char*);
```

```c
// product.c
#include "product.h"
#include <string.h> // strdup
...
const char* getName(Product* product) {
    return product->name;
}

void setName(Product* product, const char* name) {
    free(product->name);
    product->name = strdup(name);
}
```

Agora, se tentarmos alterar o campo `name` diretamente depois do `getName`, o compilador avisará do erro:

```c
Product* product = ...;
const char* name = getName(product)
name[0] = '\0'; // error: assignment of read-only location ‘*name’
```

Contudo, é de notar que esta restrição pode ser contornada com algumas técnicas em certos compiladores. É por isso importante garantir que estamos a usar o `const` de forma correta e adotamos as devidas *flags* de compilação. Por outro lado, o encapsulamento através de cópias garante uma maior segurança.

É de notar que o modificador const pode resultar em diferentes limitações de acesso dependendo do local onde é colocado, e.g., `const char *` vs `char * const`. É importante compreender estas diferentes variantes. Para informação mais detalhada sobre a aplicação de `const` clicar aqui.

[2] No caso desta estratégia ser adotada com o GCC, o programa deverá ser compilado com a *flag* `-Werror=discarded-qualifiers` para considerar o *warning* como um erro.

4 Exercícios

1. Considere o programa da `Deque` + *parsing* do guião 2. Reescreva os diferentes módulos do programa de forma a garantir o encapsulamento.

2. Considere o exercício do `Guião 3-1` referente a balcões que fornecem um conjunto de serviços. Esboce uma possível solução para esse sistema, seguindo a mesma divisão de módulos que considerou para a resolução desse guião, mas garantido agora o encapsulamento.