MODULARIDADE EM PROGRAMAS C

F. Mário Martins

LABORATÓRIOS DE INFORMÁTICA III - LEI – 2015/2016

Introdução

A construção e o desenvolvimento de programas de dimensões já consideráveis, qualquer que seja a linguagem, envolvem a utilização de técnicas particulares que possam garantir que os projectos de software, apesar das suas dimensões, são controláveis e geríveis, quer no seu desenvolvimento quer no seu teste e manutenção.

Os conceitos de modularidade e encapsulamento são, em Engenharia de Software, cruciais para que o desenvolvimento de software se faça de forma controlada e reutilizável, e que o código gerado seja robusto, sendo os eventuais erros de detecção fácil e de fácil correcção.

Apresentam-se neste texto algumas das construções da linguagem C que devem ser usadas por forma a garantir que o desenvolvimento de aplicações de média e grande escala em C seja realizado à luz dos princípios da modularidade (divisão do código fonte em unidades separadas coerentes) e do encapsulamento (garantia de protecção e acessos controlados aos dados).

Porquê dividir o código fonte?

Quando se começa a programar em C é usual que os pequenos programas que desenvolvemos tenham todo o seu código fonte num único ficheiro, possivelmente grande. Edita-se o código, compila-se e executa-se e, assim sendo, para quê preocuparmo-nos em dividi-lo por diferentes ficheiros?

Esta é a mentalidade associada à designada *"small-scale programming"*. Em *"large-scale programming"* tudo é muito mais complexo e a modularidade torna-se quase fundamental para lidar com a complexidade do código e das arquitecturas dos projectos.

Vejamos algumas dessas razões:

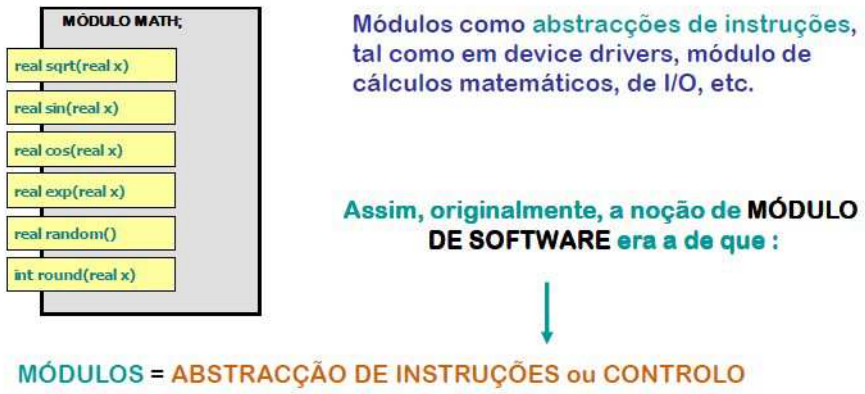
- **Uma mudança no código de um ficheiro implica apenas a recompilação desse ficheiro;**
- **É mais fácil editar e navegar no código fonte em ficheiros mais pequenos e autónomos;**

- **Permite que num dado projecto pessoas diferentes trabalhem em partes diferentes do mesmo;**
- **É muito mais fácil reutilizar código codificado num ficheiro ou módulo autónomo (mas deve ser autónomo, sem dependências de outros);**
- **É muito mais fácil isolar erros de compilação ou de execução;**
- **Estaremos a aplicar em C o que é normal em C++ or Java.**

Modularidade e Encapsulamento

A modularidade pode ser vista de duas formas distintas:

- **Funcional, procedimental ou de instruções, quando um módulo agrega (e de certa forma esconde a implementação) de um conjunto de funções que, em geral, possuem afinidades funcionais (cf. por exemplo um módulo matemático, um módulo de funções de I/O, etc.) e que é autónomo, podendo ser importado por um qualquer programa (exº stdio, strings, string, etc., na linguagem C);**



- **De abstracção de dados, quando um módulo implementa uma estrutura de dados e todas as operações sobre a mesma (escondendo as respectivas implementações), sendo a estrutura de dados privada (não acessível do exterior) e as operações tornadas públicas para acesso através de uma interface declarada; Sendo privada a estrutura de dados, apenas as funções no interior do módulo podem aceder a tal estrutura, o que garante a sua protecção e, em caso de erro, determinar a instrução que o pode causar.**

Todas as linguagens de programação permitem a criação de módulos de instruções (módulos de controlo) que são em geral os módulos que compõem as bibliotecas das linguagens, e que são módulos que em geral importamos (reutilizamos) nos nossos programas.

Algumas linguagens de programação possuem construções que facilitam a construção dos módulos mais modernos, vistos como abstracções de dados, cf. C++, C# e Java. Nestas linguagens, a concepção de programas consiste exactamente na definição destes módulos centrados nos dados.

Veremos aqui como poderemos criar módulos de software com estas propriedades usando as construções existentes em C.

Compilação de um programa C

Consideremos a título de exemplo a criação de um programa capaz de criar uma **stack de inteiros** e realizar as usuais operações de inserir e remover dados da stack.

Teremos certamente os ficheiros `main.c`, `stack.h` e `stack.c`, sendo:

- `main.c`: o programa principal que usa o módulo stack;
- `stack.c`: o código fonte do módulo stack;
- `stack.h`: a *header file* do módulo stack, que define o que é invocável do exterior;

Quando se faz o `make` do programa, cada ficheiro é pré-processado pelo pré-processador de C (`cpp`) que resolve todas as directivas `#include` e `#define`, produzindo novos ficheiros fonte contendo o código de todos os ficheiros incluídos e no qual todas as constantes e macros foram substituídas

pelo que representam. Em seguida, o(s) ficheiro(s) fonte `.c` são compilados e gerado o ficheiro objecto (`.o` ou `.obj`) final. Ficheiros `.h` não são compilados e apenas fornecem informação sintáctica ao compilador.

Finalmente, todos os ficheiros `.o` são ligados (*linked*), juntamente com mais algumas bibliotecas e código especial, criando um executável.

A importância das *header files* (.h)

Uma *header file* (`.h`) é um ficheiro que não deve conter qualquer código fonte C, e que deve ser usado exclusivamente para definir que funções, constantes ou outra informação que o respectivo ficheiro `.c` deseja exportar, ou seja, tornar acessível a outros módulos. Assim, em C, um ficheiro `.h` define a API do respectivo ficheiro `.c`, declarando os designados *protótipos das funções*, ou seja, a sua estrutura sintáctica.

No exemplo, pretendemos implementar um **módulo stack**, ou seja, **um módulo que implementa uma stack de inteiros e encapsula essa implementação interna e as funções para manipulação da stack (módulo de dados)**.

O código fonte escreve-se no ficheiro `stack.c`, e contém duas variáveis globais ao ficheiro, um array de inteiros para representar a stack e um inteiro como *stack pointer*, e quatro funções para manipulação da stack: *init*, *push*, *pop* e *empty*.

O ficheiro `stack.h` contém os protótipos das quatro funções e não faz qualquer referência às duas variáveis de `stack.c` dado que, naturalmente, pretendemos que as mesmas sejam **privadas** e, assim, **apenas acessíveis do exterior através das funções da sua API** (`stack.h`) , seja para consulta seja para modificação.

stack.h

```
#ifndef stack_h
#define stack_h

   void initStack(void);
   void pushOntoStack(int number);
   int popFromStack(void);
   int stackEmpty(void);

#endif
```

Manter a modularidade dos módulos

Ao escrevermos os nossos programas devemos pois ter a preocupação de dividir o código fonte em módulos. Módulos devem ser auto-suficientes, independentes de contextos e, assim, serem reutilizados, serem fáceis de manter e de corrigir.

Para que os módulos sejam verdadeiras cápsulas de software, robustas, seguras, utilizáveis apenas através das funções definidas nas suas APIs, as variáveis neles declaradas devem-no ser segundo regras específicas.

Relembremos os tipos de variáveis em C e seu alcance (ou *scope*).

Variáveis globais

Sempre que em C se declaram variáveis dentro de um bloco ou função elas são locais ao bloco ou função. Deixam de existir quando o bloco ou função terminam a sua execução.

Todas as variáveis declaradas fora destes contextos são *variáveis globais*, ou seja, são visíveis a partir de qualquer instrução ou função dentro do programa, podendo ser acedidas e alteradas.

Obviamente esta forma de programar é muito perigosa e a utilização de variáveis globais deve ser reduzida ao estritamente inevitável.

Variáveis `static`

A linguagem C fornece porém um excelente mecanismo para que se possa garantir que uma variável é privada (pelo menos num dado contexto). Se uma variável exterior a um bloco ou a qualquer função for declarada como `static`, ela é *global* dentro do ficheiro em que foi declarada mas torna-se *privada* para o exterior, ou seja, não acessível a qualquer instrução exterior a tal ficheiro. Tal acesso proibido é detectado em tempo de compilação.

As variáveis `static` são portanto um mecanismo fundamental para a implementação em C do mecanismo de encapsulamento de dados fundamental para a criação de módulos de dados com as propriedades anteriormente referidas.

As tais estruturas de dados privadas passam a ser em C estruturas de dados em que as variáveis são declaradas como `static`, tal como se fez com a estrutura de dados de implementação da stack em `stack.c`.

stack.c

```
#include <stdio.h>

#include "stack.h"

/* Inclui os seus próprios protótipos. Isto é uma boa prática.
   Sendo os ficheiros .c compilados em separado, quanto mais
   informação sintáctica tiver o compilador melhor. */

/* Variáveis que implementam a stack de inteiros.
   São globais no ficheiro mas privadas fora deste.
   stackPointer indica a próxima posição livre.
   Se stackPointer == 0 a stack está vazia.
*/

#define MAX_STACK_SIZE      500

static int stack[MAX_STACK_SIZE];
static int stackPointer = 0;

/*****************************************************/

void initStack(void)
{
  stackPointer = 0;
}

/*****************************************************/

void pushOntoStack(int number)
{
  stack[ stackPointer ] = number;
  stackPointer++;
}

/*****************************************************/

int popFromStack(void)
{
  stackPointer--;
  return stack[ stackPointer ];
}

/*****************************************************/

int stackEmpty(void)
{
  if (stackPointer > 0)
    return 0; /* false - stack não vazia */
  else
    return 1; /* true – stack vazia.*/
}

/*****************************************************/
```

É também possível ter funções `static`. Tal aplica-se a funções internas ou auxiliares às quais não pretendemos dar acesso noutros ficheiros. Os protótipos destas funções não devem aparecer no `.h` do módulo.

Declaração `extern`

Se um módulo tem variáveis globais não `static` então estas são acessíveis de outro módulo/ficheiro. Porém, para que este ficheiro externo que pretende usar a variável o possa fazer terá que declará-la como sendo uma variável que lhe é externa, usando a palavra reservada `extern`.

Porém, se num programa nosso tal situação se verificar, então será preferível que o ficheiro cuja variável é necessária a terceiros disponibilize funções para a consultar e modificar. É muito melhor prática que usar `extern`.

O programa `main.c`

`main.c`

```
#include <stdio.h>

/* Protótipos do módulo stack */
#include "stack.h"

/* MAIN:
* Realiza a leitura de números inteiros via teclado
* e usa uma stack para os escrever no ecrã por ordem
* inversa da leitura.
*/

int main(int argc, char* argv[])
{
  const int VALOR_SAIDA = -999;
  int numero;

  /* Inicialização da stack */
  initStack();
  /* Ciclo de leitura e push */
  printf("Introduza um número: (%d = FIM):\n", VALOR_SAIDA);
  /* Leitura via teclado */
  scanf(" %d", &numero);
  while (numero != VALOR_SAIDA)
  {
    pushOntoStack(numero);
    printf("Introduza um número: (%d = FIM):\n", VALOR_SAIDA);
    scanf(" %d", &numero);
  }
```

```
  /* Vamos escrever os números pela ordem inversa da entrada */
  if ( stackEmpty() )
  {
    printf("A stack está vazia !.\n");
  }
  else
  {
    printf("Números por ordem inversa:\n");
    while ( ! stackEmpty() )
    {
       /* Faz pop & escreve no ecrã */
       numero = popFromStack();
       printf("%d\n", numero);
    }
  }
  return 0;
}
```

Não sendo absolutamente necessário, é no entanto boa prática que o `main.c`, dado que necessita de usar as funções do **módulo stack**, faça `#include "stack.h"`. Com esta informação, porque em C cada `.c` é compilado separadamente, o compilador poderá fazer uma melhor verificação sintáctica de tais funções.

Um módulo stack mais genérico

O programa anterior, baseado no **módulo stack** desenvolvido, é uma melhoria indiscutível relativamente a um programa que tivesse todo o código fonte num só ficheiro.

Porém, mesmo este programa poderia ainda ser melhorado. Considere-se que no programa **main** tínhamos a necessidade de usar **duas stacks**. O módulo `stack.c` não é suficientemente genérico para nos permitir criar um qualquer número de instâncias de stack, dado que, de facto**, o módulo não implementa um tipo de dados**.

De facto, em `stack.c` temos apenas um *array* e um *stackPointer*. São variáveis mas não são um *tipo de dados*. **Um tipo de dados** permite que várias variáveis sejam declaradas como sendo desse tipo, ou seja, permite termos **várias instanciações**.

Como o poderíamos fazer sem grandes alterações no código que já temos?

Bom, de facto é muito simples. Necessitamos em primeiro lugar de criar um tipo de dados **`Stack`**, para que com ele se possam declarar várias variáveis que são stacks, cf**. `Stack* stack1, stack2;`**

O tipo de dados seria então definido como:

```
#define MAX_STACK_SIZE 500
typedef struct
{
  int stack[MAX_STACK_SIZE];
  int stackPointer;
}
Stack;
```

Agora, as funções terão que ser modificadas dado que em vez de trabalharem numa única stack fixa, devem agora ser generalizadas para trabalharem numa qualquer stack passada como parâmetro. Por exemplo, a função `pushOntoStack` passaria a ser:

```
void pushOntoStack(Stack* s, int num)
{
   s->stack[ s->stackPointer ] = num;
   (s->stackPointer)++;
}
```

O novo `stack2.h` passaria a ser:

stack2.h

```
#ifndef stack_h
#define stack_h
#define MAX_STACK_SIZE 500
typedef struct
{
  int stack[MAX_STACK_SIZE];
  int stackPointer;
}
Stack;

void initStack(Stack* s);
void pushOntoStack(Stack* s, int num);
int popFromStack(Stack* s);
int stackEmpty(Stack* s);

#endif
```

Assim, um tipo designado por **Stack** encapsula a real representação da stack, que continua a ser baseada num *array* e num inteiro. Porém, agora, no ficheiro `stack2.c` apenas implementaremos as funções.

stack2.c

```
#include <stdio.h>
#include "stack2.h"

/*******************************************************/
void initStack(Stack* s)
{
   s->stackPointer = 0;
}
/*******************************************************/
void pushOntoStack(Stack* s, int num)
{
  s->stack[ s->stackPointer ] = num;
  (s->stackPointer)++;
}
/*******************************************************/
int popFromStack(Stack* s)
{
  (s->stackPointer)--;
  return s->stack[ s->stackPointer ];
}
/*******************************************************/
int stackEmpty(Stack* s)
{
  if (s->stackPointer > 0)
    return 0; /* false – stack não vazia */
  else
    return 1; /* true – stack vazia */
}
/*******************************************************/
```

Está assim criado um módulo de dados que implementa o tipo de dados **`Stack`** que agora pode ser usado em qualquer contexto e nos permite criar um número arbitrário de instâncias.

Resumo

- O código dos programas deve ser dividido por unidades modulares razoavelmente pequenas e autónomas, devendo-se ter em especial atenção a criação de módulos que representam abstracções de dados;
- Os ficheiros `.h` (*header files*) não devem incluir instruções mas apenas protótipos, definições de tipos, de constantes e macros; Devem ser incluídos nos `.c` que usam tais funções;
- Variáveis globais devem ser completamente evitadas;
- Usar `static` para variáveis e funções que se pretende que sejam privadas num dado módulo;
- Incluir sempre o `.h` no respectivo `.c` de modo a que o compilador verifique a sintaxe das funções vs. os respectivos protótipos;

F. Mário Martins