**MODULARIDADE EM PROGRAMAS C  
F. Mário Martins  
LABORATÓRIOS DE INFORMÁTICA III - MiEI**

**2013/2014 (rev. 2018/2019)**

# 1.- 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).

# 2.- 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, sob a forma da divisão do código em vários ficheiros fonte, torna-se quase fundamental para lidar com a complexidade do código e das arquitecturas dos projectos

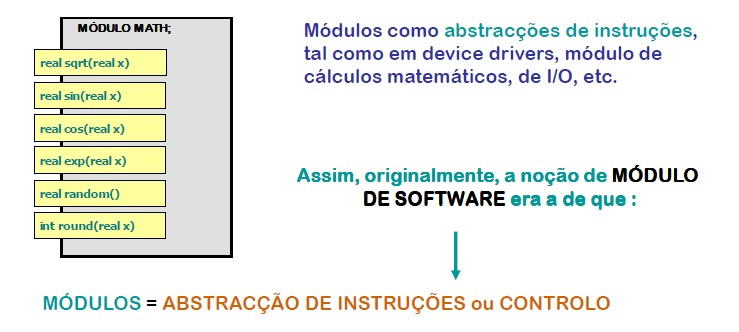
Vejamos algumas dessas razões:

* **Uma eventual mudança no código de um ficheiro implica apenas a recompilação desse ficheiro (cf. makefile);**
* **É mais fácil editar e navegar no código fonte em ficheiros mais pequenos e autónomos;**
* **Permite que num dado projecto pessoas diferentes implementem partes diferentes do mesmo, depois de definidas as APIs (ficheiros .h) de cada parte;**
* **É muito mais fácil reutilizar código codificado num ficheiro ou módulo autónomo, mas deve ser mesmo autónomo sem grandes dependências semânticas de outros;**
* **É muito mais fácil isolar erros de compilação ou de execução;**
* **Estaremos a aplicar em C o que é normal em C++, C# ou Java, ainda que sem termos hirarquias, herança e polimorfismo.**

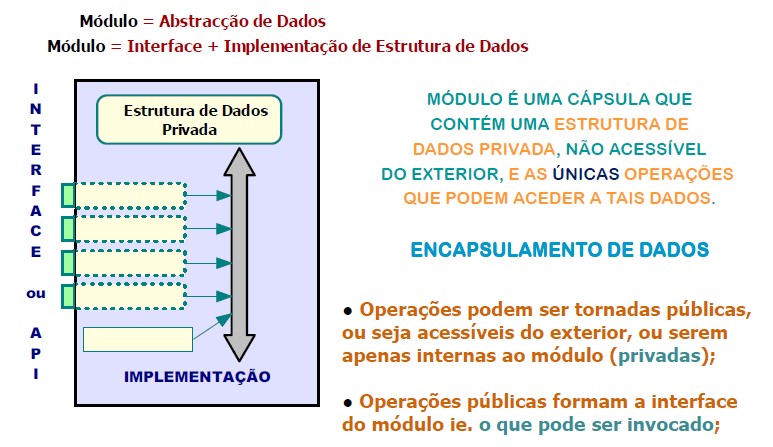
# 3.- Modularidade e Encapsulamento

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, math, string, etc., na linguagem C);**



* **De abstracção de dados, quando um módulo implementa uma estrutura de dados e todas as operações sobre a mesma (abstraindo de quem o usa os detalhes de implementação), sendo a estrutura de dados privada (não acessível directamente do exterior) e as operações tornadas públicas para acesso através de uma interface (API) declarada; Sendo a estrutura de dados privada, apenas as funções no interior do módulo podem aceder a tal estrutura, o que garante a sua protecção e, em caso de erro, determinar as instruções que o podem causar.**



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

Algumas linguagens de programação possuem construções que facilitam a criação de 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. Vamos fazê-o passo a passo.

# 4.- Modularidade básica usando múltiplos ficheiros.

## 4.1. 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 inicializar, inserir e remover dados da stack. Vamos dividi-lo em ficheiros e chamemos **módulo stack** ao código que implenta a stack de inteiros.

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, i.e. visível;

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.

## 4.2- A importância das *header files* (.h)

Uma *header file* (.h) é um ficheiro que deve conter apenas a informação que é necessária para que o código externo use as facilidades definidas no respectivo .c. Por isso, não deve declarar nem variáveis nem funções estáticas e não deve declarar tipos internos usados no .c, mas apenas 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 em particular os designados *protótipos das funções* (estrutura sintáctica) a exportar.

Uma *header file* (.h) deve garantir que um ficheiro que faça #include A.h possa fazer uso completo do que foi tornado público por A.c.

Também deve ser *auto-contida*, ou seja, garantir que A.c se conhece a si próprio. Para tal a primeira linha de A.c deverá ser #include "A.h".

Finalmente, o .h deve ser idempotente, ou seja, poder ser incluído mais do que uma vez sem erros de compilação (usando "ifndef .. #define .. #endif).

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)**, usando dois ficheiros stack.h e stack.c.

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*. Contém também quatro funções públicas 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. Teremos então:

**stack.h**

#ifndef stack\_h

#define stack\_h

**void initStack(void);**

**void pushOntoStack(int number);**

**int popFromStack(void);**

**int stackEmpty(void);**

#endif

## 4.3.- 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 reutilizáveis, fáceis de manter e de corrigir ou aumentar.

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 ou zero.

**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 básico da linguagem C para a implementação do mecanismo de protecção 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 stack.c 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/ficheiro tem variáveis globais não static então estas são também acessíveis de outro qualquer 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 módulo/ficheiro cuja variável é necessária a terceiros disponibilize funções para a consultar e modificar (cf. usuais get() e set() noutras linguagens).** **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;  
**}**

Note-se que se houver algum erro no uso da stack, apenas teremos que ver o que se passa em stack.c.

Temos assim um primeiro exemplo de modularidade em C pela lógica divisão do código por vários ficheiros. Mas estamos ainda um pouco longe do que pretendemos.

# 5.- Um módulo stack mais genérico

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

Porém, este programa poderá ainda ser muito 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 stacks, dado que, de facto**, o módulo não implementa um tipo de dados** apenas **cria uma estrutura de dados**, e uma estrutura de dados não é, por si só, um tipo.

De facto, em stack.c temos apenas um *array* e um *stackpointer*. São variáveis de implementação de uma stack mas não são um *tipo de dados*. **Um tipo de dados** (pré-definido ou definido pelo programador) deve permitir que várias variáveis sejam declaradas como sendo desse tipo, ou seja, permitir termos **várias instancias desse tipo**.

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** (tipos começam por regra por letra maiúscula) para que com ele se possam declarar várias variáveis que são stacks, ou, por exemplo, apontadores para stacks, cf. a declaração: **Stack\* stack1, stack2;**

O tipo de dados seria então definido usando **struct** e **typedef**,

**#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)++;

}

ou seja, recebe uma Stack\* s e insere um inteiro em s.

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** é exportado e tenta encapsular 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.

Vamos ver como ficaria agora o **main.c** e vamos testar o encapsulamento e a segurança da implementação.

**main.c (versão 2)**

#include <stdio.h>

#include <stdlib.h>

/\* Protótipos, etc. do módulo stack2 \*/

**#include "stack2.h"**

**int main(int argc, char\* argv[]) {**

const int VALOR\_SAIDA = -999;

int numero;

/\* declaração das duas stacks \*/

**Stack\* stk1; Stack\* stk2;**

/\* Inicialização das stacks \*/

**initStack(stk1); initStack(stk2);**

/\* Ciclo de leitura e push para **stk1** \*/

printf("Introduza um número: (%d = FIM):\n",

VALOR\_SAIDA);

/\* Leitura via teclado \*/

scanf(" %d", &numero);

while (numero != VALOR\_SAIDA){

**pushOntoStack(stk1, numero);**

printf("Introduza um número: (%d = FIM):\n",

VALOR\_SAIDA);

scanf(" %d", &numero);

}

/\* Ciclo de leitura e push para **stk2** \*/

printf("Introduza um número: (%d = FIM):\n",

VALOR\_SAIDA);

/\* Leitura via teclado \*/

scanf(" %d", &numero);

while (numero != VALOR\_SAIDA){

**pushOntoStack(stk2, numero);**

printf("Introduza um número: (%d = FIM):\n",

VALOR\_SAIDA);

scanf(" %d", &numero);

}

/\* Vamos escrever os números de stk1 pela ordem inversa da entrada \*/

if( **stackEmpty(stk1)** ) {

printf("A stack está vazia !.\n");

}

else {

printf("Números por ordem inversa:\n");

while ( ! **stackEmpty(stk1)** )

{

/\* Faz pop & escreve no ecrã \*/

**numero = popFromStack(stk1);**

printf("%d\n", numero);

}

}

/\* Vamos escrever os números de stk2 pela ordem inversa da entrada \*/

........................... (**igual com stk2)**

**/\*\*\* TESTE DE ROBUSTEZ E ENCAPSULAMENTO DOS DADOS \*\*\*/**

**stk1->stack[0] = 999;**

**stk2->stackPointer--;**

**printf("%d >> %d\n", stk1->stack[0],**

**stk2->stackPointer);**

return (EXIT\_SUCCESS);

**}**

**Números por ordem inversa:**

**33**

**22**

**-1**

**Números por ordem inversa:**

**2**

**1**

**999 >> -1**

O programa mostra que de facto definimos um tipo **Stack**, que podemos declarar inúmeras variáveis do tipo **Stack** ou **Stack\***, que conseguimos um desenho modular mas que, como as três linhas finais de teste o provam, temos acesso no **main** (ou noutro ficheiro qualquer) à representação interna do tipo **Stack** e acedê-la para a modificar. **A implementação é modular mas ainda não é protegida, ou seja, robusta.**

**E qual foi o erro que cometemos?**

Simples, deixamos que "todos" os que pretendem usar stack2.c saibam a sua representação/implementação interna, porque a exportámos no ficheiro stack2.h.

Se quisermos ver o mesmo tipo de erro, basta consultar a definição de base do tipo FILE de C (em /usr/include/stdio.h), que sendo supostamente protegido, não o é de facto e podemos escrever directamente nos campos da estrutura FILE e boicotar completamente as operações com ficheiros em C !

typedef struct {

int \_cnt;

unsigned char \*\_ptr;

unsigned char \*\_base;

int \_bufsiz;

short \_flag;

short \_file;

char \*\_\_newbase;

void \*\_lock;

unsigned char \*\_bufendp;

} **FILE**;

Precisamos de usar em C uma técnica mais segura para protecção das nossas implementações de MÓDULOS DE DADOS. **A técnica a empregar designa-se por TIPOS INCOMPLETOS** e é extremamente simples. Será apresentada em texto próprio.

# 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 ou assinaturas das funções, tipos, constantes e macros; Devem ser incluídos nos .c que necessitam de tais recursos;

⦁·Variáveis globais devem ser completamente evitadas;

⦁·Deve-se usar a declaração static para variáveis e funções que se pretende que sejam privadas num dado módulo; Deve-se declarar como extern variáveis que têm que ser usadas num ficheiro em que não foram declaradas, mas são de evitar via funções de consulta ou modificação próprias;

⦁·Incluir sempre o .h no respectivo .c de modo a que o compilador verifique a sintaxe das funções vs. os respectivos protótipos;

⦁·**Para implementar de forma robusta módulos de dados deve aplicar-se a técnica dos tipos incompletos ou opacos em C (ver em seguida).**

**F. Mário Martins**