



Estruturas de Dados e Algoritmos II

[Página principal](#) ► [Licenciaturas](#) ► [2016/2017 - Semestre Par](#) ► [INF0869](#) ► [Práticas](#) ► [Notas de C](#)

Introdução ao C para programadores de Java

► [Índice](#)

Parte 1

O Java herdou do C uma grande parte da sua sintaxe, assim como tipos, instruções e operadores.

Sendo o Java uma linguagem de programação por objectos, os programas estruturam-se em classes, com os respectivos atributos e métodos. Por seu lado, os programas em C, uma linguagem imperativa, estruturam-se em funções. Se no Java todas as instruções do programa se encontram nos métodos, em C todas as instruções estão dentro das funções. Se retirarmos de um programa em Java o invólucro das definições das classes, obtemos um programa que parece um programa escrito em C.

O resto desta 1ª parte apresenta, primeiro, coisas que se mantêm e, depois, algumas das diferenças entre o Java e o C.

Coisas que se mantêm

Mantêm-se as estruturas de controlo: `if`, `while`, `for` (com um *caveat*), `do/while` e `switch`. Também se mantêm as instruções `return`, `break` e `continue` (as duas últimas só sem argumento).

Mantém-se a sintaxe da instrução de afectação.

Mantém-se a maioria dos operadores. No entanto, o C não tem o operador `+` para concatenar *strings*, nem o operador `>>>`.

Mantém-se a sintaxe das *strings* (delimitadas por aspas) e dos caracteres (delimitados por plicas e limitados a 8 bits). Mantém-se também a representação especial de alguns caracteres, como `\n` (fim de linha), `\t` (tab), `\\` (*backslash*), `"` (aspas) e `'` (plica), assim como a notação `\octal`, onde *octal* é um número em base 8, entre 0_8 e 377_8 .

Mantém-se a sintaxe dos comentários.

Mantém-se o uso do tipo `void` para as funções que não devolvem qualquer valor.

A sintaxe da definição de uma função em C é a sintaxe da definição de um método em Java, exceptuando a cláusula `throws`.

Tipos primitivos

Os tipos primitivos do C são os apresentados a seguir.

Valores inteiros

- `short` (inteiro de 16 bits)

- `int` (32 bits)
- `long` (32 ou 64 bits, dependendo da arquitectura da máquina)
- `long long` (64 bits)
- `char` (8 bits)

Os tamanhos indicados são os tamanhos comuns para os valores destes tipos.

Todos estes tipos correspondem a inteiros com sinal, mas para todos existe uma variante `unsigned` (sem sinal), que permite representar só números não negativos. Por exemplo, o tipo `unsigned int` é a variante sem sinal do tipo `int`.

Em C não existe o tipo `byte`, podendo o tipo `char` ser usado em vez dele.

Valores reais

Tipos `float` (precisão simples, com 32 bits) e `double` (precisão dupla, com 64 bits).

Valores lógicos

Tipo `bool`, com valores `true` e `false`. (Ver abaixo o que fazer para poder usar este tipo.)

Em C, qualquer valor pode ser testado. O teste falha (é `false`) se o valor for zero (ie, se todos os seus bits tiverem o valor 0), e sucede (é `true`) para qualquer valor diferente de zero (ie, se algum bit tiver o valor 1).

Funções

Declaração

A sintaxe das declarações de funções tem a forma seguinte, semelhante à das declarações dos métodos no Java:

```
tipo nome(argumentos...)
{
    instruções
}
```

Quando uma função não devolve nenhum valor (ie, trata-se de um procedimento), o seu tipo será `void`.

No exemplo seguinte, o tipo `void` no lugar da declaração dos argumentos significa que a função não tem qualquer argumento:

```
unsigned int sem_argumentos(void)
{
    return 123456789;
}
```

Uso de funções

Onde em Java pode aparecer a invocação de um método, nas formas `Classe.método(...)` ou `objecto.método(...)`, em C pode aparecer a invocação de uma função `função(...)`.

Função main

A função `main` de um programa em C desempenha o mesmo papel que o método `main` de um programa em Java.

Esta função devolve um valor de tipo `int` e uma instrução

```
return código;
```

tem, nesta função, o mesmo efeito que `System.exit(código);` em Java.

Um exemplo de função `main` é:

```
int main(void)
{
    return 0;
}
```

Nome dos ficheiros

Os ficheiros com código C devem ter extensão `.c`, por exemplo, `programa.c`.

(A extensão `.h` é usada para ficheiros que só contêm declarações, ie, onde se definem tipos e nomes (de funções ou de variáveis). O *h* vem de *header*.)

Inclusão de ficheiros

O compilador de C precisa de ter informação sobre as funções usadas no código *antes* de elas serem usadas. Para isso acontecer, a definição de uma função deve aparecer no ficheiro antes do seu uso.

Quando são usadas funções definidas externamente, a directiva `#include` do pré-processador de C (correspondente à directiva `import` do Java) permite importar as suas declarações para o ficheiro onde as funções são usadas. Estas declarações fazem parte do conteúdo dos ficheiros `.h`.

A directiva `#include` tem duas formas:

```
#include <ficheiro.h>
#include "ficheiro.h"
```

A primeira forma é usada quando se pretendem importar ficheiros pertencentes ao sistema ou ao compilador. A segunda utiliza-se para importar ficheiros do programador.

Pela razão apontada acima, todas as directivas `#include` devem ocorrer no início do ficheiro em que aparecem.

O tipo `bool` é uma adição ao C relativamente recente e, para o usar e ter acesso às constantes `true` e `false`, o programa deve incluir o ficheiro `stdbool.h` através da directiva:

```
#include <stdbool.h>
```

Escrita na consola

A função `printf` permite a um programa em C escrever mensagens na consola. O seu uso tem a forma:

```
printf(formato, expressão1, expressão2, ..., expressãon);
```

com $n \geq 0$.

O formato é uma *string* que controla como o valor de cada uma das expressões será escrito. O conteúdo do formato é escrito literalmente na consola, excepto quando aparece o carácter `%`. Quando este aparece, o que é escrito depende do que se lhe segue no formato. Algumas hipóteses são:

- `%` – é escrito o carácter `%`;
- `d` – é escrito o valor inteiro da próxima expressão a escrever;
- `c` – é escrito carácter correspondente ao valor da próxima expressão a escrever;

- s – é escrita a *string* correspondente à próxima expressão a escrever;
- f ou g – é escrito o valor real da próxima expressão a escrever.

(Há muitas outras possibilidades e variantes, descritas na documentação da função, acessível, em Linux, através do comando `man 3 printf`.)

Exemplos:

1. Se `idade` for uma variável inteira com valor 27 e `igual` for do tipo `char` e tiver o valor '=', a instrução:

```
printf("idade + 10 %c %d\n", igual, idade + 10);
```

escreverá na consola:

```
idade + 10 = 37
```

2. Se `taxa` for uma variável real com valor 0.75, a instrução:

```
printf("50%% de %f é %g\n", taxa, taxa / 2);
```

escreverá na consola:

```
50% de 0.750000 é 0.375
```

O formato deverá conter tantas ocorrências de % (seguido de qualquer coisa diferente de %) quantas as expressões que o seguem.

A presença do carácter '\n' no fim do formato faz terminar a linha que aparece na consola (ou seja, o que quer que seja escrito a seguir, sê-lo-á na linha seguinte da consola).

Para usar esta função (ou outras relacionadas com escrita/leitura), o programa deve incluir o ficheiro `stdio.h`.

Compilar código C

O `gcc` é um compilador de distribuição livre, e está habitualmente presente nos sistemas Linux (mas também está disponível para Windows).

O comando seguinte invoca o `gcc` para compilar o programa em C contido no ficheiro `programa.c` e diz-lhe para chamar `programa` ao executável criado:

```
gcc -Wall programa.c -o programa
```

O executável criado pode, depois, ser executado através do comando:

```
./programa
```

que indica que queremos executar o programa contido no ficheiro `programa` da directoria corrente (.).

(A opção `-Wall` indica ao compilador que deve assinalar todas as ocorrências, no código, de coisas que não constituem erros mas que podem não estar correctas.)

O for e o gcc

Algumas versões do `gcc` compilam código C de acordo com o *standard* C89, também conhecido como ANSI C, com algumas extensões. Este *standard* não contempla a possibilidade de declarar variáveis na zona de inicialização do ciclo `for`, que só apareceu no C99.

Se o `gcc` assinalar um erro numa instrução do tipo:

```
for (int i = 0; i < MAX; ++i) ...
```

isso significa que ele não está a compilar o código de acordo com o *standard* C99, e é necessário forçá-lo explicitamente a fazê-lo, incluindo no comando de compilação a opção `-std=c99` ou `-std=gnu99`, como no comando:

```
gcc -std=gnu99 -Wall programa.c -o programa
```

(Usando gnu99, em vez de c99, tem-se acesso a algumas funções que não fazem parte do C99 e que podem ser úteis, como a função `strdup`.)

Parte 2

Definição de constantes

Em C, podem-se definir constantes através de directivas do tipo

```
#define NOME EXPRESSÃO
```

A directiva `#define` diz ao pré-processador de C para substituir todas as ocorrências do *identificador* `NOME` no programa por `EXPRESSÃO`.

Dada a natureza da substituição, é necessário algum cuidado com o conteúdo da `EXPRESSÃO`. Se tivermos a constante `DOBR05` definida como

```
#define DOBR05 5 + 5
```

o pré-processador substituirá a instrução

```
vinte = 2 * DOBR05;
```

por

```
vinte = 2 * 5 + 5;
```

o que, provavelmente, não era o que se pretendia.

Para evitar situações destas, se `EXPRESSÃO` não for atómica (eg, uma constante, como 5, é uma expressão atómica) deverá estar entre parêntesis:

```
#define DOBR05 (5 + 5)
```

Assim, a instrução obtida após a substituição seria

```
vinte = 2 * (5 + 5);
```

Estas definições devem aparecer no início do ficheiro, a seguir aos `#include`. Convencionalmente, os nomes das constantes são escritos só com maiúsculas, para se distinguirem das variáveis.

Às constantes como `DOBR05` chama-se *constantes simbólicas*.

Tópicos relacionados: definição de macros em C.

BOAS PRÁTICAS

Evitar a ocorrência literal de constantes no código. O uso de constantes simbólicas não só contribui para a legibilidade do código como para facilitar o desenvolvimento e a manutenção dos programas. Preferir sempre algo como:

```
#define NLOCALIDADES 100000

unsigned short distancias[NLOCALIDADES];

...
for (int i = 0; i < NLOCALIDADES; ++i)
    ...
```

a:

```
unsigned short distancias[100000];

...
for (int i = 0; i < 100000; ++i)
    ...
```

Declarações de variáveis

A sintaxe das declarações de variáveis é como a do Java:

```
tipo nome, ...;
```

Dependendo de onde a declaração aparece, uma variável pode ser classificada como:

Global

Se a sua declaração ocorre *fora* de qualquer função. Estas variáveis são (potencialmente) acessíveis em todo o programa.

Global no ficheiro

Se a sua declaração ocorre *fora* de qualquer função e começa pela palavra reservada `static`. Estas variáveis só são acessíveis no ficheiro em que são declaradas.

Local

Se a sua declaração ocorre *dentro* de uma função. Em particular, a variável é local *ao bloco* em que é declarada. (Um *bloco* é uma sequência possivelmente vazia de declarações e de instruções delimitada por chavetas { e }.) Os argumentos de uma função são também locais à função.

Local persistente

Se a sua declaração ocorre *dentro* de uma função e começa pela palavra reservada `static`.

Variáveis locais persistentes

Uma variável local é persistente se na sua declaração for qualificada como `static`, eg,

```
{
    ...
    static tipo nome;
    ...
}
```

Estas variáveis mantêm o seu valor entre chamadas da função a que pertencem, ie, são partilhadas por todas as invocações da função. Por exemplo, se tiver o código seguinte

```
#define NELEMENTOS 10

float f(int k, float x)
{
    static float v[NELEMENTOS];

    if (x >= 0)
        v[k] = x;

    return v[k];
}
```

na sequência de invocações da função $f(4, -1)$, $f(4, 5.25)$ e $f(4, -1)$, os valores devolvidos serão 0.0, 5.25 e 5.25, por esta ordem (a explicação para o 0.0 é dada na secção seguinte).

Inicialização das variáveis

A inicialização de uma variável está associada à sua declaração. A inicialização pode ser implícita ou explícita.

A inicialização explícita tem a forma

```
tipo nome = expressão;
```

Neste caso, a variável *nome* é inicializada com o valor da expressão. Se a variável for global (ou local persistente), a expressão só pode envolver constantes.

Quando a declaração de uma variável global (ou local persistente) não a inicializa explicitamente, ela é implicitamente inicializada com o valor 0 (zero) correspondente ao seu tipo.

As variáveis locais não explicitamente inicializadas, não são inicializadas (ie, são inicializadas com o "lixo" que se encontra na zona memória que lhes corresponde).

A inicialização das variáveis locais explicitamente inicializadas é feita de cada vez que a função é chamada, mas a inicialização das variáveis locais persistentes só é feita *uma* vez durante a execução do programa.

Definição de constantes (v2)

Quasi-constantes simbólicas podem ser definidas, em C, recorrendo à sintaxe

```
const tipo NOME = expressão;
```

Neste caso, *NOME* é definida como uma *variável* cujo valor não pode ser alterado pelo programa. Sendo uma variável, *NOME* só pode ser utilizada durante a execução do programa, não podendo ocorrer em declarações de variáveis globais ou locais persistentes.

Tendo em conta as restrições apontadas, o exemplo anterior poderia ser reescrito como

```
const int DOBR05 = 5 + 5;
```

Arrays

Em Java, o uso típico de *arrays* é o seguinte:

```

{
    ...
    tipo[] nome;                // declaração
    ...
    nome = new tipo[NELEMENTOS]; // criação/inicialização
    ...
    ... nome[posição] ...      // acesso a uma posição
    ...
    objecto.método(..., nome, ...); // array como argumento
    ...
    ... nome.length ...          // nº de elementos do array
    ...
}

```

As principais diferenças do C consistem: na coincidência da declaração com a criação/inicialização; a localização dos parêntesis rectos na declaração; e a inexistência do atributo `length` (ou de qualquer outro, visto que não há a noção de `objecto`), não sendo, em geral, possível saber qual a dimensão de um *array*. Tal como em Java, a primeira posição de um array tem índice 0 (zero), e a última *número-de-elementos* - 1.

Uma declaração de um *array* em C tem a forma

```
tipo nome[NELEMENTOS];
```

e a sua inicialização segue as regras para a inicialização de variáveis já descritas. Se `nome` for uma variável global (ou local persistente), `NELEMENTOS` terá de ser uma expressão constante.

Um *array* pode ser inicializado explicitamente. A declaração

```
int a5[5] = { 10, 20, 30, 40, 50 };
```

inicializa `a5` com os valores 10, 20, ...

Quando a inicialização é explícita, o número de elementos do *array* pode ser omitido. A declaração seguinte é equivalente à anterior:

```
int a5[] = { 10, 20, 30, 40, 50 };
```

Se o número de valores iniciais for inferior ao número de elementos do *array*, os elementos restantes serão inicializados a 0. A declaração

```
int a5[5] = { 10, 20 };
```

inicializa `a5` com os valores 10, 20, 0, 0 e 0.

Arrays com mais de uma dimensão podem, igualmente, ser inicializados explicitamente:

```

int m3x3[3][3] = {
    { 1, 2, 3 },
    { 4 }
    { 7, 8, 9 }
};

```


Os 3 elementos da 2ª linha de `m3x3` serão inicializados com 4, 0 e 0, respectivamente.

Declaração de argumentos *arrays*

A declaração de um *array* como argumento de uma função pode apresentar várias formas:

tipo função(..., tipo' vector[NELEMENTOS], ...)

Dentro de função, *vector* é um *array* com NELEMENTOS de tipo *tipo*'.

tipo função(..., tipo' vector[], ...)

Dentro de função, *vector* é um *array* com um número indeterminado de elementos de tipo *tipo*' (neste caso, deverá ser usada uma convenção sobre o conteúdo do *vector* para detectar quando acaba).

tipo função(..., int n, tipo' vector[], ...)

Aqui, o argumento *n* é usado para indicar o número de elementos do *vector*.

tipo função(..., int n, tipo' vector[n], ...)

Situação semelhante à anterior, que associa explicitamente o argumento *n* ao número de elementos do *vector*.

Na lista de argumentos da função, *n* tem, obrigatoriamente, de aparecer antes de *vector*.

tipo função(..., tipo' matriz[LINHAS][COLUNAS], ...)

tipo função(..., tipo' matriz[][COLUNAS], ...)

tipo função(..., tipo' matriz[][5], ...)

tipo função(..., int n, int m, tipo' matriz[n][m], ...)

tipo função(..., int n, tipo' matriz[n][n], ...)

tipo função(..., int m, tipo' matriz[][m], ...)

O tratamento de *arrays* com mais de uma dimensão (nestes exemplos são duas) é idêntico ao dos vectores.

A única dimensão cujo número de elementos se pode omitir é a primeira. Não é possível ter `tipo' matriz[][]` ou `tipo' matriz[LINHAS][]`.

BOAS PRÁTICAS

Se o conteúdo do *array* argumento não usa alguma convenção para indicar o fim dos elementos, a dimensão do *array* deverá ser incluída na sua declaração, o que dá mais informação ao compilador:

```
double maximo(int n, double v[n])
{
    ...
}
```

Arrays de char e *strings*

Em C não há um tipo próprio para *strings*. Uma *string* corresponde a um *array* de caracteres que contém um carácter a indicar o fim da *string*.

As duas declarações seguintes são equivalentes:

```
char ola[] = "Bom dia!";
char ola[] = { 'B', 'o', 'm', ' ', 'd', 'i', 'a', '!', '\0' };
```

Quando se usa a notação "...", está implícita a presença do carácter `'\0'`, o *terminador*, no fim da *string*. O vector *ola* tem 9 elementos mas o comprimento da *string* "Bom dia!" é 8, ie, o terminador não é contado.

(O carácter '\0' é o carácter com código ASCII 0 (zero), que é diferente do carácter '0', correspondente ao algarismo 0 (e cujo código ASCII é 48). A notação *código*, onde *código* é um inteiro entre 0 e 255 escrito em base 8 (ou seja, entre 0₈ e 377₈), corresponde ao carácter com esse código ASCII. Por exemplo, '\60' é o carácter '0', ou seja '\60' == '0'.)

As funções que trabalham com *strings*, como a função `printf`, dependem da presença do terminador para determinarem onde elas acabam.

Tal como nos *arrays* de outros tipos de valores, as posições não inicializadas explicitamente são inicializadas com o valor 0 do tipo correspondente que, neste caso é '\0'. Na declaração

```
char ola[9] = "Bom";
```

as posições com índice de 4 a 8 são inicializadas com '\0' (assim como a com índice 3, mas esta é inicializada explicitamente).

Tópicos relacionados: funções que manipulam strings: `strlen`, `strcmp`, `strcpy`, etc.

Afectação de *arrays*

Em C, não é possível afectar *arrays*. As afectações do código seguinte são inválidas:

```
int a[5], b[5];
char s[10];

a = b;           // inválido
s = "string";    // inválido (só funciona na declaração)
```

Também não é possível uma função devolver um *array*.

Parte 3

Referências

A declaração

```
int n, *rn;
```

declara uma variável `n`, cujo tipo é `int` e cujos valores vão ser inteiros, e uma variável `rn`, cujo tipo é `int *` e cujos valores vão ser *referências* para inteiros.

Uma *referência* (ou *apontador*) indica uma zona da memória onde se pode encontrar um valor de um determinado tipo. Por exemplo, uma referência para `int` indica (é o *endereço* de) uma posição de memória onde está um inteiro.

O operador prefixo `&` (*endereço de*) permite obter o endereço da zona de memória reservada para uma variável, ou seja, uma referência para o seu conteúdo. O operador de *desreferenciação* `*` (também prefixo) permite aceder ao valor para que uma referência aponta. O fragmento de código

```

int n, *rn;

n = 5;           // atribui 5 a n
rn = &n;         // atribui o endereço de n a rn

printf("n = %d, *rn = %d\n", n, *rn);

*rn = *rn * 2;   // multiplica o valor apontado por rn por 2 e
                // guarda o resultado na mesma zona de memória

printf("n = %d, *rn = %d\n", n, *rn);

```

escreverá na consola

```

n = 5, *rn = 5
n = 10, *rn = 10

```

(Porque é 10 o valor de `n` na última linha?)

Referências e tipos

De acordo com a declaração

```
int i, *pi, **ppi;
```

a variável `i` tem tipo `int` (contém um valor inteiro), a variável `pi` tem tipo `int *` (contém uma referência para um valor inteiro), e a variável `ppi` tem tipo `int **` (contém uma referência para uma referência para um valor inteiro).

A declaração acima é equivalente às declarações

```

int i;
int *pi;
int **ppi;

```

Operadores & e *

A aplicação do operador `&` dá origem a um valor de um tipo com mais um nível de *indirecção*, ie, com mais um `*`: o tipo de `&i` é `int *`, o tipo de `&pi` é `int **`, e o tipo de `&ppi` é `int ***`.

Com cada aplicação do operador `*` obtém-se um valor de um tipo com menos um nível de indirecção, ie, com menos um `*`: o tipo de `*ppi` é `int *`, e o tipo de `**ppi` e de `*pi` é `int`.

A referência NULL

A referência NULL é a *referência vazia*, que não refere nada. Esta constante é definida nos ficheiros `stddef.h`, `stdio.h`, `stdlib.h` e `string.h`, entre outros. Para a poder utilizar, um destes ficheiros terá de ser incluído.

Esta referência é compatível com qualquer tipo de referência (tal como `null` em Java).

Referências genéricas

Referências com tipo `void *` são *referências genéricas* (ou *apontadores genéricos*) que podem ser convertidas para qualquer outro tipo de referência, assim como qualquer tipo de referência pode ser convertido para uma referência genérica. Essas conversões são, em geral, automáticas.

Referências para funções

A declaração

```
int (*f)(void *, void *);
```

declara `f` como uma referência para uma função com dois argumentos, ambos de tipo `void *`, que devolve um inteiro.

Referências para funções permitem usar funções como argumentos de outras funções e podem ser usadas para obter estruturas de dados genéricas.

Nota sobre *arrays* e referências

Tal como referido antes, em C, não é possível afectar *arrays*, sendo as afectações seguintes inválidas:

```
int a[5], b[5];
char s[10];

a = b;           // inválido
s = "string";    // inválido (só funciona na declaração)
```

No entanto, o "valor" de um *array* é o endereço da zona de memória que lhe está associada (e o tipo desse valor é o de uma referência para um elemento do *array*). Assim, o seguinte código já é válido:

```
int *a, b[5];
char *s;

a = b;           // a recebe o endereço de b
s = "string";    // s recebe o endereço de "string"
```

Quando uma variável tem um tipo referência, o C permite usá-la como se tratasse de um *array*. Na sequência do código anterior, seria, portanto, possível, escrever o código seguinte:

```
a[4] = 123;      // põe 123 na posição 4 de a (e de b)

for (int i = 0; s[i] != '\0'; ++i)
    printf("%c", s[i]); // escreve `string' na consola
printf("\n");
```

A função `scanf`

A leitura de valores simples, introduzidos a partir do teclado, pode ser feita através da função `scanf`. Esta função tem uma *interface* semelhante à da função `printf`.

```
scanf(formato, referencia1, referencia2, ..., referencian);
```

com $n > 0$.

Neste caso, o *formato* indica o tipo de valores a ler (eg, `%d` para inteiros, `%f` para floats, `%c` para caracteres, e `%s` para *strings* sem espaços). Para que `scanf` saiba onde guardar os valores lidos, os restantes argumentos são os endereços das zonas de memória onde esses valores serão guardados.

Por exemplo, se for executado o código:

```
int n;
double r;
char s[30];

scanf("%d %s %lf", &n, s, &r);
```

e for introduzida a linha

```
323 xpto 71.9909
```

a variável `n` ficará com o valor 323, a variável `s` com o valor "xpto", e a variável `r` com o valor 71,9909.

Repare que, como `s` é um *array* e o nome de um *array* representa a zona de memória que lhe está associada, não foi necessário usar o operador `&` para ler a *string*.

A documentação da função (obtida executando `man scanf`) explica o seu funcionamento, incluindo o uso de `%lf` acima e o valor que ela devolve.

Estruturas (struct)

As *estruturas* do C são o equivalente das classes só com atributos no Java. Por exemplo, se em Java se define a classe

```
class Node {
    int element;
    Node next;
}
```

em C usaríamos (note o ponto e vírgula final):

```
struct node {
    int element;
    struct node *next;
};
```

De acordo com esta definição, a estrutura `node` tem os *campos* `element` e `next`.

Dois pontos importantes:

1. O nome do tipo correspondente à estrutura assim definida é `struct node`, ao contrário do `Node` do Java, onde não aparece a palavra `class`.
2. No Java, o atributo `next` da classe `Node` *não* pode ser um objecto. Se fosse, ele conteria, por sua vez, um objecto no seu atributo `next`, que conteria um objecto, que conteria um objecto, que ...
A definição apresentada só é viável porque `Node.next` contém uma *referência* para um objecto, como acontece com todas as variáveis cujo tipo é uma classe (assim como com os *arrays*).

No C, o facto de uma variável ser uma referência é indicado explicitamente na sua declaração, através do símbolo `*`, como na declaração do campo `next`:

```
struct node *next;
```

Comparação entre o C e o Java

Operação	Java	C
Declaração	<code>Node node;</code>	<code>struct node *node;</code>
Criação	<code>node = new Node();</code>	<code>node = node_new();</code>
Construtor	<pre>public Node() { // inicializações... }</pre>	<pre>struct node *node_new() { struct node *node = malloc(sizeof(struct node)); // inicializações... return node; }</pre>
Acesso a um campo/atributo	<code>... node.element ...</code>	<code>... node->element ...</code>
Afectação de um campo/atributo	<code>node.element = value;</code> <code>node.next = null;</code>	<code>node->element = value;</code> <code>node->next = NULL;</code>
Afectação	<code>node = node.next;</code>	<code>node = node->next;</code>
Destruição	<code>node = null;</code>	<code>free(node);</code>

A principal diferença nos excertos de código acima é que enquanto que, em Java, se cria o objecto com `new Node()`, que reserva espaço para ele na memória, invoca o construtor e devolve a sua referência, em C é preciso explicitamente reservar o espaço de memória e proceder à inicialização dos campos. Outra diferença é o uso de `->` em vez de `.` no acesso aos campos de uma estrutura através da sua referência.

BOAS PRÁTICAS

Uma boa prática é ter uma função como a mostrada acima, que cria, inicializa e devolve (uma referência para) a estrutura, ie, que faz o equivalente ao `new` do Java.

Como a função `free` só liberta o espaço ocupado por uma estrutura, pode também ser boa ideia ter uma função que, além disso, liberte também o espaço ocupado por outros dados lá contidos. Esta função receberia a (referência para a) estrutura e teria uma assinatura do tipo:

```
void node_destroy(struct node *node);
```

No Java, a libertação do espaço ocupado por objectos não usados é feita automaticamente, através de um processo de *garbage collection*.

No C, é possível ter uma variável que é uma estrutura (note a ausência de `*` nas declarações):

Operação	C
Declaração	<code>struct node one_node;</code>
Declaração com inicialização	<code>struct node one_node = { 0, NULL };</code>
Criação	<i>n/a</i>
Acesso a um campo	<code>... one_node.element ...</code>

Afectação de um campo	<code>one_node.element = value;</code> <code>one_node.next = NULL;</code>
Afectação	<code>one_node = another_node;</code>
Destruição	<i>n/a</i>

Neste caso, já é usado o `.` para seleccionar o campo a que se quer aceder. Por outro lado, como a estrutura existe a partir do ponto em que é declarada, não há lugar a uma fase de criação explícita.

ATENÇÃO

Se `struct node` ocupa 1000 *bytes* e `one_node` e `another_node` são estruturas, a instrução

```
one_node = another_node;
```

copiar todos os 1000 *bytes* de `another_node` para a zona de memória reservada para `one_node`. Trata-se, portanto, de uma operação potencialmente cara e deve ser evitada.

Gestão dinâmica de memória

Para trabalhar com estruturas de dados dinâmicas, que possam crescer e diminuir durante a execução de um programa, é necessário poder reservar memória para instalar os novos elementos. Essa memória deve ser libertada quando deixar de ser precisa.

A função `malloc` é uma das funções que o C disponibiliza para reservar memória a pedido, memória essa que é libertada pela função `free`. Estas funções são declaradas no ficheiro `stdlib.h` com as seguintes assinaturas:

```
void *malloc(size_t size);
void free(void *ptr);
```

`malloc` reserva uma zona de memória com *size bytes* (`size_t` é um inteiro sem sinal) e devolve o seu endereço como um apontador genérico. `free` recebe a referência (genérica) de uma zona de memória reservada através de `malloc` e liberta-a, de modo a poder ser reutilizada por chamadas a `malloc` posteriores.

A instrução

```
node = malloc(sizeof(struct node));
```

reserva memória suficiente para guardar uma `struct node` e guarda o seu endereço na variável `node`. Para libertar esta memória, seria usada a instrução

```
free(node);
```

A função `malloc` *não* toca no conteúdo da zona de memória que reserva. Quaisquer inicializações que sejam necessárias devem ser efectuadas pelo programa, após a chamada a `malloc`.

Tópicos relacionados: as funções `calloc` e `realloc`.

O operador `sizeof`

`sizeof(tipo)` e `sizeof(variável)` calculam, respectivamente, o número de *bytes* ocupados por um elemento de tipo *tipo* e pela variável *variável*.

A expressão `sizeof(struct node)` indicará o tamanho ocupado por uma `struct node`, que não será inferior à soma dos tamanhos dos seus campos

```
sizeof(struct node) ≥ sizeof(int) + sizeof(struct node *),
```

ou seja, o espaço necessário para guardar um inteiro e uma referência para uma `struct node`.

(O espaço ocupado por uma estrutura pode ser superior à soma das dimensões dos seus campos por questões de alinhamento de valores em memória.)

A declaração typedef

Através da declaração `typedef` é possível definir novos tipos para uso no programa. A sua sintaxe é:

```
typedef tipo novo-tipo;
```

Por exemplo, tendo a definição de `struct node` acima, poder-se-ia definir um novo tipo `node_t` assim:

```
typedef struct node node_t;
```

Após esta definição, as declarações de variáveis que contivessem uma referência para uma `struct node` poderiam escrever-se:

```
node_t *node;
```

Também seria possível definir o tipo `Node` através da declaração

```
typedef struct node *Node;
```

que declara o nome `Node` como um sinónimo de `struct node *`, ie, como o *tipo das referências para struct node*.

Parte 4

Acesso a ficheiros

O ciclo de trabalho com um ficheiro inicia-se com a sua *abertura* (e *criação*, se ainda não existe), que é seguida por uma sequência de operações de *leitura* e/ou de *escrita* (intercaladas com pedidos de reposicionamento, quando aplicável), e termina com o *fecho* do ficheiro.

A cada ficheiro aberto está associada uma *cabeça* (virtual) de leitura/escrita (L/E), colocada em alguma posição do ficheiro enquanto ele está aberto. Na abertura de um ficheiro, a sua cabeça de L/E é colocada na posição 0, correspondente ao primeiro *byte* do ficheiro.

Cada operação de leitura (ou escrita) faz avançar a posição da cabeça de L/E tantas posições quantos os *bytes* lidos (ou escritos). Se as operações sobre um ficheiro consistirem numa sequência de leituras ou de escritas, temos um ficheiro com *acesso sequencial* (é o que acontece, em geral, com os ficheiros que só contêm texto, que são lidos ou escritos sequencialmente). É, no entanto, possível reposicionar a cabeça de L/E de um ficheiro e ter *acesso directo* a qualquer posição do ficheiro.

As funções seguintes, não sendo as únicas disponibilizadas pelo C, permitem executar as operações referidas acima.

int open(char *filename, int flags, mode_t mode)

A função `open` tenta abrir o ficheiro `filename` e devolve um inteiro não negativo, que será posteriormente usado no programa quando se quiser fazer alguma operação sobre este ficheiro. Este inteiro é normalmente designado como o *descriptor do ficheiro* (ou *file descriptor*). Se ocorrer algum erro durante a sua execução, a função devolve o valor -1.

O argumento `flags` é uma combinação de *flags* que condicionam a operação da função. Algumas *flags* são

O_RDONLY – a abertura é feita em modo de leitura
O_WRONLY – a abertura é feita em modo de escrita
O_RDWR – a abertura é feita em modo de leitura e escrita
O_CREAT – o ficheiro é criado se ainda não existe
O_TRUNC – se o ficheiro existe e é aberto O_WRONLY ou O_RDWR, o seu conteúdo é apagado

Uma das três primeiras *flags* tem de estar presente na chamada da função.

(A combinação de *flags* é feita através do operador `|`, que calcula o *OU-bit-a-bit* dos seus operandos. Por exemplo, o valor de $0101_2 \mid 0011_2$ é 0111_2 .)

Para a utilização de um ficheiro com uma estrutura de dados dinâmica, uma combinação útil de *flags* é `O_RDWR | O_CREAT`. Com este valor para as *flags*, a função abre o ficheiro para leitura e escrita, se ele já existe, criando-o antes se ele ainda não existir.

Só é necessário incluir o terceiro argumento se as *flags* contiverem `O_CREAT`. Se o ficheiro tiver de ser criado, `mode` indica as permissões de acesso que lhe ficarão associadas. Quando o ficheiro já existe, as suas permissões mantêm-se.

O valor de `mode` resulta, tal como o de `flags`, da combinação de um ou mais valores através do operador `|`. Para que o utilizador que executa o programa possa ler e escrever o ficheiro criado, `mode` poderá ser a combinação `S_IRUSR | S_IWUSR`.

Para usar esta função, deverá incluir os ficheiros `sys/types.h`, `sys/stat.h` e `fcntl.h`.

(Encontra muito mais informação sobre `open` executando o comando `man 2 open`.)

int close(int fd)

Fecha o ficheiro associado a `fd`, que deverá ser o valor devolvido por `open` quando o ficheiro foi aberto. Todos os ficheiros abertos deverão ser fechados quando deixar de ser necessário aceder-lhes.

Para usar esta função, deverá incluir o ficheiro `unistd.h`.

ssize_t read(int fd, void *address, size_t count)

Esta função tenta ler `count` *bytes* do ficheiro associado a `fd`. Os *bytes* efectivamente lidos serão colocados na zona de memória cujo endereço é indicado por `address`. Devolve o número de *bytes* transferidos ou -1, se ocorrer algum erro.

A função `read` devolve um inteiro não negativo diferente de `count` quando se tenta ler para além do fim do ficheiro. Se uma chamada a `open` que origina a criação do ficheiro for seguida de uma chamada a `read` (com `count` diferente de 0), esta última devolverá 0, porque o ficheiro está vazio. Esta sequência de operações pode ser usada para detectar que o ficheiro aberto é um ficheiro que não existia, como exemplifica o fragmento de código seguinte:

```

int fd;

fd = open(filename, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);

if (fd == -1)
{
    perror("open");

    return <VALOR-QUE-INDICA-ERRO>;
}

...

switch (read(fd, ...))
{
    case -1:                                // ocorreu um erro na leitura
        perror("read");

        ...                                // limpezas

        close(fd);

        return <VALOR-QUE-INDICA-ERRO>;
    case 0:                                // o ficheiro está vazio
        ...                                // inicializações

        /* fall through */
    default:                                // o ficheiro não está vazio
        return <VALOR>;
}

```

Para usar esta função, deverá incluir o ficheiro `unistd.h`.

(Os tipos `size_t` e `ssize_t` correspondem, respectivamente, a inteiros sem e com sinal.)

`ssize_t write(int fd, void *address, size_t count)`

Transfere *count bytes*, localizados na memória a partir do endereço *address*, para o ficheiro associado a *fd*, ie, escreve-os no ficheiro. Devolve o número de *bytes* escritos ou -1, se ocorrer algum erro.

Para usar esta função, deverá incluir o ficheiro `unistd.h`.

`off_t lseek(int fd, off_t offset, int whence)`

Cada operação de leitura (ou escrita) lê (ou escreve) *count bytes* a partir da posição corrente da cabeça, e fá-la avançar tantas posições quantos os *bytes* lidos (ou escritos). Uma operação de leitura (ou escrita) subsequente operará a partir da nova posição corrente.

A função `lseek` permite reposicionar a cabeça de L/E de um ficheiro e torna possível aceder directamente a qualquer *byte* do ficheiro.

Enquanto que o primeiro argumento da função é o já conhecido descritor do ficheiro, o terceiro argumento indica-lhe como deve interpretar o segundo:

Valor de whence	Interpretação de offset	Nova posição
SEEK_SET	posição absoluta a partir do início do ficheiro	offset
SEEK_CUR	distância à posição corrente (pode ser negativa)	posição corrente + offset
SEEK_END	distância ao fim do ficheiro (pode ser positiva)	tamanho do ficheiro + offset

O valor devolvido é a nova posição da cabeça de L/E, ou -1 se ocorrer algum erro (por exemplo, se a posição resultante for negativa).

A tabela seguinte apresenta as posições da cabeça de L/E resultantes de uma sequência de chamadas a `lseek` com os argumentos mostrados, para um ficheiro com 100 *bytes*. (Repare que, sendo 0 a posição do primeiro *byte* do ficheiro, a posição 100 é a do *byte* a seguir ao último *byte* do ficheiro.)

offset	whence	Nova posição
0	SEEK_SET	0
0	SEEK_CUR	0
10	SEEK_CUR	10
30	SEEK_CUR	40
-15	SEEK_CUR	25
0	SEEK_END	100 a)
-20	SEEK_END	80
20	SEEK_END	120 b)
50	SEEK_SET	50
-60	SEEK_CUR	50 c)

Notas:

- Quando `lseek` é chamada com offset 0 e whence igual a `SEEK_END`, a cabeça de L/E fica colocada imediatamente *a seguir* ao último *byte* do ficheiro.
- A cabeça de L/E fica colocada 20 *bytes* para lá do fim do ficheiro. Se for executada uma operação de escrita com a cabeça nesta posição, o sistema preenche as 20 posições intermédias (da 100 à 119) com o *byte* 0. (O que acontecerá se for executada uma operação de escrita quando a cabeça de L/E está na posição imediatamente a seguir ao último *byte* do ficheiro?)
- Esta chamada, que tenta colocar a cabeça de L/E *antes* da primeira posição do ficheiro, devolve -1 e não altera a posição corrente da cabeça.

Para usar esta função, deverá incluir os ficheiros `sys/types.h` e `unistd.h`.

void perror(char *prefix)

Todas as funções referidas acima sinalizam a ocorrência de um erro devolvendo -1, o que não permite distinguir entre os vários erros que podem ocorrer. Por exemplo, quando se tenta abrir um ficheiro para leitura e não se consegue, isso pode dever-se a não ter permissão para o ler (erro `EACCES`) ou a ele não existir (erro `ENOENT`). As funções com este tipo de comportamento guardam, normalmente, o código do erro que ocorreu numa variável global chamada `errno`, definida numa biblioteca do sistema. A função `perror` consulta esta variável e escreve na *consola de erro* (`stderr`) do programa a mensagem correspondente.

Se `unreadable` for um ficheiro que não podemos ler e se não existir um ficheiro chamado `nosuchfile`, o código seguinte

```
fd = open("unreadable", O_RDONLY);
if (fd == -1)
    perror("open: unreadable");

fd = open("nosuchfile", O_RDONLY);
if (fd == -1)
    perror("open: nosuchfile");
```

produzirá na consola

```
open: unreadable: Permission denied
open: nosuchfile: No such file or directory
```

ou o equivalente noutra língua.

O protótipo desta função encontra-se no ficheiro `stdio.h`.

Faça algumas experiências usando estas funções, criando e abrindo ficheiros, e lendo e escrevendo texto e números de e para ficheiros.

Funções ISO C

As funções para acesso a ficheiros e as constantes descritas acima são definidas no *standard* POSIX para (a biblioteca d) o C.

O *standard* ISO que define a linguagem C (eg, C89 ou C99) também inclui funções para o mesmo efeito, que são apresentadas nesta secção. (Uma implementação que respeite um *standard* ISO do C não tem de incluir as funções anteriores, enquanto que uma implementação POSIX do C terá de incluir aquelas e as do ISO C.) No Linux, as funções acima correspondem a chamadas ao sistema e são usadas pelas funções ISO C.

Tal como as funções acima, em caso de erro, estas funções guardam o código correspondente em `errno`.

Para usar estas funções, deverá ser incluído o ficheiro `stdio.h`.

FILE *fopen(char *path, char *mode)

A função `fopen` abre o ficheiro indicado por `path` em modo `mode`, uma *string* que pode começar por

r

A abertura é feita em modo de leitura. É um erro o ficheiro não existir.

w

A abertura é feita em modo de escrita. Se o ficheiro existir, o seu conteúdo é apagado, senão é criado um ficheiro com o nome dado.

a

A abertura é feita em modo de escrita, só permitindo acrescentar dados ao ficheiro. Se o ficheiro não existe, é criado.

Se `mode` for `"r+"`, `"w+"` ou `"a+"`, aplicam-se as observações anteriores, mas o ficheiro é aberto para leitura e para escrita.

(`mode` pode também conter o carácter `b`, que indica tratar-se de um ficheiro binário, ie, não de texto. Esta indicação não tem qualquer efeito nos sistemas POSIX, como o Linux.)

O valor devolvido deverá ser usado em todas as funções abaixo (corresponde ao argumento `stream`), quando se pretender realizar alguma operação sobre o ficheiro. Em caso de erro na abertura/criação, o valor devolvido será `NULL`.

int fclose(FILE *stream)

Fecha o ficheiro associado a `stream`, depois de ter garantido que todas as alterações ao ficheiro foram realmente feitas.

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)

Basicamente equivalente a

```
read(fd, ptr, size * nmemb).
```

size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *stream)

Basicamente equivalente a

```
write(fd, ptr, size * nmemb).
```

int fseek(FILE *stream, long offset, int whence)

Equivalente a

```
lseek(fd, offset, whence)
```

mas devolve 0 (sucesso) ou -1 (erro).

long ftell(FILE *stream)

Serve para obter a posição corrente da cabeça de L/E e é equivalente a

```
lseek(fd, 0, SEEK_CUR).
```

void rewind(FILE *stream)

Equivalente a

```
fseek(stream, 0, SEEK_SET).
```

int fflush(FILE *stream)

É usada para forçar as alterações feitas ao ficheiro a serem efectivamente escritas em memória secundária (disco).

int feof(FILE *stream)

Indica se foi atingido o fim do ficheiro.

int ferror(FILE *stream)

Indica se ocorreu um erro na utilização do ficheiro

Em C, existem três nomes pré-definidos com tipo `FILE *`:

stdin

A *entrada normal* de dados do programa, normalmente associada ao que é introduzido através do teclado.

stdout

A *saída normal* do programa, destinada ao *output* normal do programa, normalmente associada ao terminal onde o programa é executado.

stderr

A *saída de erro* do programa, destinada às mensagens de erro, normalmente associada ao terminal onde o programa é executado. (É usada, por exemplo, pela função `perror`.)

As funções `fread`, `fwrite` e `fseek` destinam-se, sobretudo, a ser usadas em ficheiro (binários) de acesso directo. O C possui, também, uma família de funções apropriadas para o uso em ficheiros de texto em que, tipicamente, o acesso é sequencial.

As primeiras são as generalizações das já conhecidas `printf` e `scanf`:

```
int fprintf(FILE *stream, char *format, ...)
int fscanf(FILE *stream, char *format, ...)
```

(De facto, `printf(...)` é o mesmo que `fprintf(stdout, ...)`, e `scanf(...)` é o mesmo que `fscanf(stdin, ...)`.)

Depois, há as funções `fgetc/getc` (leitura de um carácter), `getchar` (equivalente a `getc(stdin)`), `fgets` (leitura de uma linha), `fputc/putc` (escrita de um carácter), `putchar` (equivalente a `putc(c, stdout)`), `fputs`, (escrita de uma *string*) e `puts` (escrita de uma linha, equivalente a `fputs(string + "\n", stdout)`).

Parte 5

Os argumentos de main

O cabeçalho completo da função `main` é:

```
int main(int argc, char *argv[])
```

Os argumentos `argc` e `argv` desempenham o mesmo papel que o argumento `args` na declaração do método `main` em Java:

```
public static void main(String[] args)
```

Através de `argc` e de `argv` é possível aceder aos argumentos usados na invocação do programa.

- O valor de `argc` é o número de argumentos dados ao programa mais 1. Se o seu valor for 1, o programa foi chamado sem qualquer argumento.
- `argv` é um *array* de *strings* com `argc` elementos. Em `argv[0]` encontra-se o nome através do qual o programa foi invocado, e nas posições de 1 a `argc-1` encontram-se os argumentos do programa. Na posição `argc`, `argv` contém o valor `NULL`.

Se o programa for executado pelo comando

```
./o-meu-programa 1o-arg "o segundo" 3 olá mundo
```

`argc` terá o valor 6 e `argv` será

```
argv[] = {
    "./o-meu-programa",
    "1o-arg",
    "o segundo",
    "3",
    "olá",
    "mundo",
    NULL
}
```

O valor devolvido por main

O valor devolvido pela função `main` é um inteiro e é usado para comunicar, a quem executa o programa, como correu a sua execução. A convenção usada é:

- Para indicar que a execução decorreu sem problemas, a função `main` devolve o valor 0 (zero).
- Para indicar que ocorreu algum problema durante a execução do programa, a função `main` devolve um valor diferente de zero.

Experimente criar o programa `ok`, que devolve 0, o programa `not-ok`, que devolve 1, e executar os comandos:

```
if ./ok; then echo Ok; else echo 'Not ok'; fi
```

e

```
if ./not-ok; then echo Ok; else echo 'Not ok'; fi
```

A função `fgets`

O protótipo desta função, contido em `stdio.h`, é

```
char *fgets(char *s, int size, FILE *stream);
```

`fgets` lê uma linha de texto e coloca-a na zona de memória apontada por `s`, incluindo o carácter correspondente ao fim de linha `'\n'`, seguida do terminador `'\0'`. Se tudo isto ocupar mais do que `size bytes`, só lê `size-1 bytes` e coloca-os lá seguidos do terminador. (Uma chamada subsequente a esta ou a outra função de leitura, sobre o mesmo ficheiro, continuará a leitura a partir do ponto em que a anterior a tenha interrompido.)

O argumento `stream` indica o ficheiro de onde será feita a leitura e corresponderá a um valor devolvido, por exemplo, pela função `fopen`.

Se não se pretender ler de um ficheiro (para ler texto introduzido manualmente, por exemplo), o valor de `stream` deverá ser `stdin`, como no esquema de código seguinte:

```
#define MAXLINHA 4096

...

{
    char linha[MAXLINHA];

    ...

    ... fgets(linha, MAXLINHA, stdin) ...

    ...

}
```

`fgets` devolve `NULL` se o fim do ficheiro já foi atingido e não foi possível ler nada, e `s` (ie, o 1º argumento) se leu alguma coisa. Para indicar o fim dos dados quando se lê de `stdin`, prime-se `C-d`.

Tópicos relacionados: as funções (e macros) `fgetc`, `getc`, `getchar`, `fputs`, `fputc`, `putc`, `putchar` e `puts`.

A função getline

(Esta função não pertence ao standard ISO C, mas ao standard POSIX, e pode não ser suportada em todos os sistemas ou compiladores.)

O protótipo desta função, contido em `stdio.h`, é

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

A função `getline` faz o mesmo que `fgets`, com duas diferenças. A primeira, e maior, diferença é que pode aumentar o tamanho da zona de memória que lhe é passada, para garantir que há espaço para a linha lida. A segunda diferença reside no valor que é devolvido, que, neste caso, é o comprimento da linha lida, incluindo o fim de linha. (O valor devolvido será -1 se a função não conseguir ler uma linha.)

O primeiro argumento da função contém o *endereço* de uma variável que contém o *endereço* da zona de memória onde colocar o conteúdo da linha lida. O segundo argumento contém o *endereço* de uma variável que contém o tamanho dessa zona de memória. Esta zona de memória deverá ter sido obtida através da função `malloc`.

Se o valor de `*lineptr` for `NULL`, ie, se a chamada não indicar a zona de destino da linha lida, `getline` reserva uma zona de memória com o tamanho necessário (através de `malloc`), e coloca o seu endereço em `*lineptr` e o seu tamanho em `*n`.

Se o valor de `*lineptr` não for `NULL` e o tamanho indicado em `*n`, na altura da chamada, não for suficiente para albergar a linha, `getline` reserva uma nova zona de memória com a dimensão necessária (através da função `realloc`, que liberta também a zona de memória antiga), e coloca o novo endereço em `*lineptr` e a nova dimensão em `*n`.

Os valores `lineptr` e de `n` têm de ser endereços válidos, ie, diferentes de `NULL`.

O endereço contido em `*lineptr` depois de a função ter terminado deverá ser libertado pelo programa, quando essa memória deixar de ser necessária.

Última alteração: Segunda, 20 Março 2017, 11:34

NAVEGAÇÃO



Página principal

- Painel do utilizador

Páginas do site

Disciplina atual

INF0869

Participantes

Medalhas

Teóricas

Práticas

 **Notas de C**

 Exercícios

 Ficheiros

 Resoluções

 Sumários

Trabalho prático

Avaliação

As minhas disciplinas

ADMINISTRAÇÃO



Administração da disciplina

Nome de utilizador: SAMUEL MELO. (Sair)

INF0869

