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
```

(Percebe 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 `*`: dadas as declarações acima, 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

Mais sobre a função `scanf`

Como já foi dito no segundo contacto com a função, a leitura de valores simples, introduzidos a partir do teclado, pode ser feita através da função `scanf`. Relembrando, esta função tem uma *interface*semelhante à da função *printf*:

`scanf(`*formato*`,` *referência*$_1$`,` *referência*$_2$`, ...,` *referência*$_n$`);`

com $n \geq 0$.

Nesta função, o *formato* indica o tipo de valores a ler (além das sequências `%d` para `int`, `%f` para `float` e `%lf` para `double`, podem ser usadas também as sequências `%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.

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*.

Valor devolvido

A função `scanf` devolve um valor inteiro (tipo `int`), que corresponde ao número de valores lidos.

Se a linha introduzida for `323 xpto 71.9909`, o código seguinte:

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

m = scanf("%d %s %lf", &n, s, &r);
printf("foram lidos %d valores\n", m);
```

produzirá a mensagem `foram lidos 3 valores`.

Se a função encontra o fim do *input* antes de ter conseguido ler algum valor, o valor devolvido é a constante `EOF`, definida no ficheiro `stdio.h`.

NOTA: quando um programa está a ser executado interactivamente, lendo de e escrevendo para a consola, o fim do *input* pode ser indicado com a combinação de teclas `C-d`.

Consulte a documentação da função (pode usar o comando `man scanf`) para obter mais informação sobre o seu funcionamento.

Estruturas (`struct`)

As *estruturas* do C são o equivalente das classes só com atributos no Java.

Enquanto que em Java se definiria 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. 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, no Java, 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	`Node node;`	`struct node *node;`
Criação	`node = new Node();`	`node = node_new();`

Construtor	`public Node()`
	`{`
	`  //`
	`inicializações...`
	`}`
		`struct node *node_new()`
		`{`
		`  struct node *node =`
		`malloc(sizeof(struct node));`
		`  // inicializações...`
		`  return node;`
		`}`

Acesso a um campo/atributo	`... node.element ...`	`... node->element ...`
Afectação de um campo/atributo	`node.element = value;` `node.next = null;`	`node->element = value;` `node->next = NULL;`
Afectação	`node = node.next;`	`node = node->next;`
Destruição	`node = null;`	`free(node);`

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 reservar o espaço de memória explicitamente 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.

Deve haver sempre uma função, como a função `node_new` acima, que cria, inicializa e devolve (uma referência para) uma estrutura criada dinamicamente, ie, que faz o equivalente ao `new` do Java.

Como a função `free` só liberta o espaço ocupado por uma estrutura, é também 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	`struct node one_node;`

Declaração com inicialização	`struct node one_node = { 0, NULL };`
Criação	*n/a (coincide com a declaração)*
Acesso a um campo	`... one_node.element ...`
Afectação de um campo	`one_node.element = value;` `one_node.next = NULL;`
Afectação	`one_node = another_node;`
Destruição	*n/a*

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.

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

```
one_node = another_node;
```

copia 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.

Pela mesma razão, deve, igualmente, ser evitado o uso de estruturas como argumentos de funções e de funções que devolvam estruturas. Efectuar a seguinte chamada de função, implica copiar todo o conteúdo da variável `one_node` para o argumento da função:

```
  ... f(one_node) ...
```

Se uma função opera sobre uma `struct`, o respectivo argumento deverá ser uma referência:

```
... f(struct node *node)
{
  ...
}

  ... f(&one_node) ...
```

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);
```

A função `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. A função `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`

O valor de `sizeof(`*tipo*`)` e de `sizeof(`*variável*`)` é, 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, ao 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`*.

O C permite escrever código como o seguinte:

```
typedef struct node *node;

...

{
  node * node = malloc(sizeof(node));
  ...
}
```

onde a variável `node` é declarada como sendo uma referência para um valor do tipo `node`, ou seja para uma `struct node`.

No entanto, na inicialização da variável `node`, o nome `node` já se refere à **variável**, e **não ao tipo** `node`. Assim, a função `malloc` vai reservar espaço de memória para uma referência, em vez de para uma `struct node`.