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 DOBRO5 definida como

```
#define DOBRO5 5 + 5
```

o pré-processador substituirá a instrução

```
vinte = 2 * DOBRO5;
```

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 DOBRO5 (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 `DOBRO5` chama-se *constantes simbólicas*.

Tópicos relacionados: definição de macros em C.

Deve-se evitar a repetição de ocorrências literais 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 pelas 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.

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 persistentes) 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 da variáveis locais persistentes só é feita *uma* vez, no início da 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 DOBRO5 = 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.

Uma diferença importante entre o C e o Java é a localização dos *arrays* em memória. No C, a memória para o *array* seguinte

```
{
  int grande[MUITOS];
  ...
}
```

é reservada na pilha de execução do programa (o chamado *stack*). No Java, a memória para o *array* seguinte

```
{
  int grande[] = new int[MUITOS];
  ...
}
```

é reservada na zona de memória de gestão dinâmica, que é, em geral, maior.

Declaração de argumentos *arrays*

Arrays de `char` e *strings*

Afectação de *arrays*

Segundo contacto com a função `scanf`

A função `scanf` permite ler valores, como inteiros e reais, do *standard input* do programa (que está, geralmente, associado ao teclado). Esta função tem uma interface semelhante à da função `printf`:

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

com $n \geq 0$.
O *formato* tem uma sintaxe semelhante ao da função `printf` e indica o tipo de valores a ler. Para ler valores inteiros (`int`) usa-se a sequência `%d`, e para ler valores reais usa-se `%f` (`float`) ou `%lf` (`double`).

Se quisermos ler um valor para uma variável, `scanf` tem de conhecer a *localização em memória* da variável, para lá poder guardar o valor lido. Essa localização obtém-se aplicando o operador `&` à variável.

se pretendermos ler valores para guardar nas variáveis `n` (`int`) e `r` (`double`), por esta ordem, a chamada da função será:

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

Se for introduzida a linha

```
323 71.9909
```

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

Se o *`formato`* contiver algum carácter espaço, nesse ponto do processamento da linha introduzida, a função `scanf` saltará qualquer sequência de espaços, tabs (`\t`) e fins de linha (`\n`).

Os restantes caracteres que apareçam no *`formato`*, e que não façam parte de uma sequência começada por `%`, terão de aparecer explicitamente na linha introduzida.

Se a chamada da função for:

```
scanf("(%d)", &n);
```

o inteiro terá de aparecer entre parêntesis na linha introduzida (ie, `(1234)`).

A documentação da função (obtida executando `man scanf`) explica o seu funcionamento, incluindo a leitura de outros tipos de valores e o valor que a função devolve.