Programação Imperativa

Licenciatura em Ciências da Computação

Universidade do Minho

JBA AT di.uminho.pt

2015/2016

Apresentação e Introdução

Apresentação da UC

- Contacto: `jba@di.uminho.pt`
- Horário de Atendimento: 5ª, 14:00 – 16:00
- Avaliação
  - Teste prático eliminatório (16 Março)
  - Teste escrito (8 Junho)
  - Exame de recurso (29 Junho)
- Regras de funcionamento

Programa Resumido

- A linguagem *C*
- Arrays e apontadores
- Pesquisa e ordenação
- *Stacks* e *Queues*
- Memória dinâmica
- Listas e árvores binárias
- Manipulação de ficheiros

Recursos pedagógicos

- Referência bibliográfica

  
  Brian W. Kernighan & Dennis M. Ritchie
  The C Programming Language (2nd edition)
  Prentice-Hall, 1988.

- Material disponibilizado no *blackboard*:
  - Slides das teóricas
  - Colectânea de problemas
  - Fichas práticas
- Recursos *web*:
  - `http://codeboard.io` — IDE online (fins pedagógicos)
  - `http://c.learncodethehardway.org/book/` — livro *online*
  - `http://www.cprogramming.com` — variedade enorme de tutorials, faqs, code snippets, etc.
  - `http://www.learn-c.org` — curso online com exercícios práticos.
  - `http://www.loirak.com/prog/ctutor.php` — ...

Paradigma Imperativo

- Programas descrevem as *instruções* (comandos/acções) que deverão ser executadas pelo computador.
- Foca-se no **"como fazer"**, em oposição aos *paradigmas declarativos* (e.g. *paradigma funcional*) onde os programas descrevem apenas o resultado pretendido.
- Está mais próxima da forma como os computadores funcionam internamente (**linguagem máquina**):
  - o processador executa instruções simples sequencialmente;
  - a execução dessas instruções afecta o *estado* da computação (i.e. memória);
  - algumas instruções "interagem" com o ambiente (e.g. I/O).
- A natureza dos computadores modernos torna necessário que se utilizem **representações binárias** para os dados armazenados em memória.
- Exemplos de instruções: "ler informação contida numa dada posição da memória"; "adicionar dois números inteiros de 32bit"; etc.

Linguagens de programação - *Assembly*

- Nos anos 40 e 50 os programas eram escritos directamente em linguagem máquina.
- A **linguagem assembly** surge como conjunto de mnemónicas que facilita a memorização das instruções máquina.
- A tradução para linguagem de máquina é muito simples (o tradutor é designado por **assembler**).
- Hoje o assembly é utilizado apenas para aquelas partes dos programas onde a velocidade de execução é crítica, ou quando é necessária uma interacção grande com o *hardware* (e.g. *device drivers*).

```
        je      .L6
        negl    %edx
.L6:    movl    %edx, %eax
        addl    $4, %esp
        popl    %ebx
        popl    %ebp
        ret
```

Linguagens de programação - Alto Nível

- A maioria dos programas são hoje escritos em **linguagens de alto-nível**.
- O seu desenvolvimento teve início nos anos 50 e 60.
- Permitem aos utilizadores escrever programas numa linguagem mais adequada para seres humanos.
- Podem ser classificadas como estando algures entre o assembly e a linguagem natural (Português, Inglês, etc.).
- Têm grandes vantagens:
  - São fáceis de ler e (quando bem utilizadas) manter.
  - Permitem abstrair (alguns) detalhes da máquina.
- É necessário um compilador (ou um interpretador) para traduzir os programas para linguagem máquina/assembly.
- O mesmo programa pode ser traduzido por diferentes compiladores para diversas máquinas (portabilidade).

Resolução de Problemas Computacionais

Desenvolver um programa implica:

1. Definir claramente o **problema** que se pretende resolver
   - quais são os dados e como é que eles serão representados;
   - quais os resultados pretendidos.
2. Desenhar um **algoritmo** que descreva detalhadamente os passos necessários para produzir o efeito pretendido;
3. **Implementar** esse algoritmo numa linguagem de programação.

A sua implementação numa linguagem de alto nível implica:

- Exprimir o algoritmo usando as construções disponibilizada pela linguagem — **código fonte** (eventualmente recorrendo a bibliotecas...)
- Compilar os ficheiros para produzir o código de máquina respectivo (**código objecto**)
- Completar (ligar) esses ficheiros com as bibliotecas utilizadas (e outro código pré-compilado).

Linguagem C: Origens e Contexto

- Desenvolvida originalmente em 1972 por *Dennis Ritchie* da Bell Labs para implementar o Unix.
- Vocacionada originalmente para o desenvolvimento de software ao nível do sistema.
- É a primeira ou segunda linguagem de programação mais popular ao nível da oferta de emprego!
- Pode ser utilizada em quase todas as plataformas.
- Influenciou muitas outras linguagens como o C++, Java, C#, ...

História do C

- 1973: Uma das primeiras linguagem de alto nível utilizadas para escrever um sistema operativo.
- 1978: 1ª Edição do livro de Kernighan e Ritchie (K&R) "The C Programming Language":
  - Introduziu alguns melhoramentos na linguagem original.
  - Definiu uma biblioteca standard para I/O.
  - Funcionou durante muitos anos como standard de facto.
  - Incluiu o primeiro exemplo "Hello, World!". :-)
- 1988: 2a Edição do K&R actualizado para o ANSI C.
- 1990: Publicação do primeiro standard ANSI C.
  - Resposta ao aumento da popularidade.
  - Incorpora novas construções que foram surgindo ad-hoc.
  - Definição de bibliotecas standard.
  - Uniformização com C++.
- 1999; 2011: Actualização do standard ANSI-C.
  - Pequenos acertos e extensões.

Principais Características

Foi desenhada para

- Fornecer acesso directo à memória.
- Estar suficientemente próxima da linguagem máquina.
- Capturar todas as construções disponíveis a esse nível.
- Requerer uma compilação relativamente simples.
- Permitir substituir o assembly em muitas aplicações.

Mas, ainda assim

- Ser independente da máquina alvo.
- Remeter para bibliotecas as funcionalidades específicas de cada plataforma.
- Permitir modularidade, programação estruturada e reutilização de código.
- Ser uma linguagem pequena e fácil de utilizar.
- Fornecer algum apoio sobre tipos de dados.

Globalmente: **Trust The Programmer**.

Problemas do C

- Apesar de ser uma linguagem pequena, o seu domínio requer conhecimentos vastos (e.g. organização/gestão da memória; representação dos dados; etc.)
- Os programadores não são sempre de confiar, o que leva à ocorrência de problemas de fiabilidade e segurança.
- Demasiadas coisas estão ainda sujeitas à escolha do compilador, e.g. tamanho dos tipos numéricos, *endianness*, comportamentos indefinidos, etc.
- Apesar de ser utilizada para desenvolver a generalidade dos sistemas operativos (incluindo Windows) a forma de utilização pode variar significativamente.

Resumo: Prós e Contras do C

A linguagem C é muito utilizada porque permite

- Programação a um nível baixo de abstracção.
- Eficiência: tempo de execução e utilização de recursos.
- Portabilidade e popularidade (há quem discorde ...).

Coisas menos boas da linguagem C

- Não é uma linguagem *safe*.
- Programação a um nível baixo de abstracção.

Um clássico: `HelloWorld`

```c
/* ****************
     HelloWorld.c
   **************** */

#include <stdio.h>

int main () {
  printf("Hello, World!\n");
  return 0;
}
```

Comentários...

Directivas para o **Pré-processador**

Ponto de entrada no programa

Invocação de uma função da biblioteca

retorna código de erro (0 =sucesso)

O Pré-Processador

O pré-processador é um programa que transforma textualmente o código C antes de ser compilado:

- As linhas iniciadas por # são **directivas** do pré-processador
  - `include` — insere o ficheiro nomeado (se o nome for delimitado por ¨s procura-o na directoria corrente; se for delimitado por $<$ e $>$ procura nas directorias do sistema)
  - `define` — macros: palavras cujas ocorrências serão substituídas pelo pré-processador (e.g. `#define NMax 100`)
  - ...
- A directiva `#include` é normalmente utilizada para incluir **header-files** — ficheiros que contém declarações de variáveis, tipos, funções, etc. (e.g. das bibliotecas *standard*)
- As *header files* são críticas para uma boa organização do código porque permitem:
  - Separar as declarações das implementações.
  - Referir implementações contidas noutros ficheiros C.
  - Fazer manutenção ao código de forma modular.
  - Partilhar definições por vários ficheiros C.

O Processo de **Compilação**

- Existem dois tipos de ficheiros com **código fonte**:
  - Ficheiros "principais" (com extensão `.c` — que serão transformados pelo compilador).
  - Ficheiros "cabeçalho" ou "header files" (com extensão `.h` — a serem incluídos em programas).
- Ao executar o comando `gcc -o HelloWorld HelloWorld.c`, compilador irá transformar o código fonte:
  1. invoca pré-processador;
  2. **compila** código fonte produzindo código objecto (extensão `.o`)
  3. invoca o **linker** que transforma o(s) ficheiro(s) com código objecto em ficheiros **executáveis** (sem qualquer extensão em particular).

Um segundo exemplo

```
#include <stdio.h>

int quadrado (int a);

int main () {
  int x, y;
  x = 2;
  y = quadrado(2*x);
  printf("(2*%d)^2 = %d\n", x, y);
  return 0;
}

int quadrado (int a) {
  int x;
  x = a * a;
  return x;
}
```

Declaração de função (só informa o compilador do nome da função, tipo de argumentos e resultado)

Declaração de variáveis locais

Atribuições

Definição da função `quadrado` (inclui o corpo da função)

Definição da *Sintaxe*

- Os elementos constituintes de um programa C devem obedecer a uma *estrutura rígida* definida pela **sintaxe da linguagem**.
- Para apresentarmos as regras da sintaxe do C, vamos recorrer a gramáticas na *Backus-Naur Form (BNF)*:

$\langle$class$\rangle$	denota um **símbolo não-terminal** (representa uma classe de "frases" da linguagem);
$\mathbf{(}, \mathbf{)}, \mathbf{\{}, \ldots$	**símbolos terminais** serão representados a *negrito*. Representam texto que aparece literalmente nas respectivas frases da linguagem;
ϵ	denota a frase vazia;
$X \mid Y$	denota a alternativa de X ou Y;
X^+	denota a repetição uma ou mais vezes de X;
X^*	denota a repetição zero ou mais vezes de X (i.e. $X^* \equiv \epsilon \mid X^+$);
$[X]$	denota zero ou uma vez X (i.e. $[X] \equiv \epsilon \mid X$);

Exemplo de gramática

- Os *identificadores* em C obedecem à seguinte gramática:

$$
\begin{aligned}
\langle \text{ident} \rangle &\equiv \langle \text{startChar} \rangle \ \langle \text{restChar} \rangle^* \\
\langle \text{startChar} \rangle &\equiv \_ \mid \langle \text{alphaChar} \rangle \\
\langle \text{restChar} \rangle &\equiv \_ \mid \langle \text{alphaChar} \rangle \mid \langle \text{digitChar} \rangle \\
\langle \text{alphaChar} \rangle &\equiv \mathbf{A} \mid \mathbf{B} \mid \cdots \mid \mathbf{Z} \mid \mathbf{a} \mid \cdots \mid \mathbf{z} \\
\langle \text{digitChar} \rangle &\equiv \mathbf{0} \mid \mathbf{1} \mid \cdots \mid \mathbf{9}
\end{aligned}
$$

- Esta gramática diz-nos que um *identificador* em C deve começar por uma letra ou caracter *underscore* '_', seguido de zero ou mais letras, digitos ou *underscores*.
- Exemplos: `nome_aluno`, `nomeAluno`, `x239`, ...

Variáveis

- As **Variáveis** representam porções da memória apropriadas para armazenar valores de "um determinado tipo"
- A utilização de variáveis pressupõe a sua **declaração** prévia. Esta associa o nome da variável ao **tipo de dados** respectivo.
- Na perspectva do utilizador/programador, um tipo de dados determinado conjunto de valores (eventualmente, dispondo de suporte para determinadas operações sobre esses valores);
- Na perspectiva do compilador/linguagem, definem uma forma de codificação na memória dos valores pretendidos (e.g. o tipo `int` do C ocupa normalmente 32 bit em memória com a representação em complemento para 2, o que permite representar os números entre $-2,147,483,648$ e $2,147,483,647$).
- Alguns dos tipos básicos disponibilizados: `void`, `char`, `int`, `long`, `float`, `double`

Declaração de variáveis

- A sintaxe geral das declarações é:

$$[\langle \text{mods} \rangle] \ \langle \text{type} \rangle \ \langle \text{ident} \rangle \ [= \langle \text{expr} \rangle];$$

- Exemplos:

```
int x = 0;
unsigned char c;
static int z = 13;
const double PI = 3.1415;
```

- Modificadores:
  - `const`: declara variável como *read-only* (i.e. uma **constante**);
  - `signed/unsigned`: controla se representação inclui bit de sinal;
  - `auto/static`: controla visibilidade/tempo de vida da variável;
  - ...
- Se uma variável não for inicializada, o seu valor inicial é **indefinido**.

Atribuições

- As variáveis são identificadas por um **nome** que, quando inserido numa expressão, denota o **valor** armazenado na posição de memória que lhe está associada (e.g. `2*x`)
- As **atribuições** permitem actualizar o valor de uma variável. A sua sintaxe é

  $$\langle \text{lvalue} \rangle = \langle \text{expr} \rangle ;$$

  onde $\langle$lvalue$\rangle$ identifica a localização da memória afectada pela atribuição.
- O nome de uma variável, quando utilizado como $\langle$lvalue$\rangle$, representa a localização onde essa variável está armazenada.
- Exemplos de atribuições:

```
y = 2*x;
x = x+1;
```

- Não-exemplos de atribuições:

```
y+1 = 2*x;
```

= no C não é Igualdade! É Atribuição!

- Uma atribuição como `x=x+1` lembra-nos que em C não devemos ler o símbolo "=" como igualdade matemática, mas antes como "atribuição a uma variável".
- De facto, a leitura de "=" como igualdade matemática levar-nos-ia a uma contradição: *para qualquer valor de x, x nunca será igual a $x+1$.*
- Já lendo "=" como a atribuição do C, não existe qualquer contradição:

  1. vamos supor que a variável `x` contém o valor 3
  2. executar a atribuição `x=x+1` faz com que *a variável x tome o valor de $x+1$*
  3. Ora, como `x` contém o valor 3, então `x+1` é o valor 4;
  4. Logo, após a atribuição, o valor de `x` passará a ser 4.

- Note-se que o "efeito" da atribuição é uma alteração no estado do programa (memória) — onde antes estava armazenado um valor, passa a estar outro.

Variáveis: visibilidade e tempo de vida

Dois conceitos fundamentais ao lidar com variáveis em C são:

Visibilidade zona do código onde é possível referenciar a variável;

Tempo de vida período durante a execução do programa em que a variável tem existência em memória.

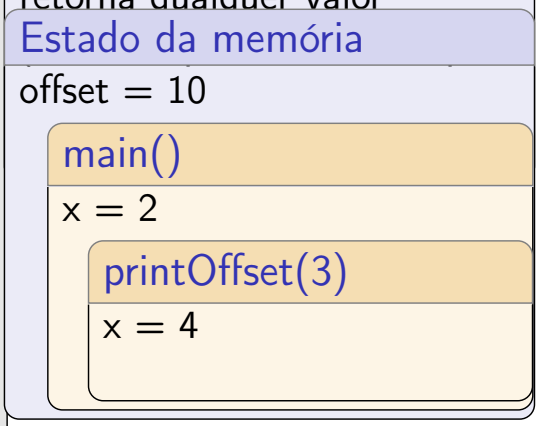
Duas classes de variáveis são possíveis em C:

- **variáveis globais**
  - são visíveis em todo o programa (podem no entanto ser "escondidas" por variáveis locais);
  - tempo de vida é todo o período de execução do programa.
- **variáveis locais**
  - visíveis apenas na função onde são declaradas;
  - tempo de vida é o período em que a função está activa.

Um erro grave (e muito comum): tentar aceder ao valor de uma variável quando o seu tempo de vida já expirou...

```c
#include <stdio.h>

int offset = 1;

void printOffset(int x) {
  x = offset + x;
  printf("%d\n", x);
}

void printOffset10(int x) {
  offset = 10;
  printOffset(x);
}

int main(void) {
  int x = 2;
  printOffset(x+1);
  printOffset10(x+2);
  printOffset(x+3);
  return 0;
}
```

Tipo `void` é usado para indicar que função não retorna qualquer valor

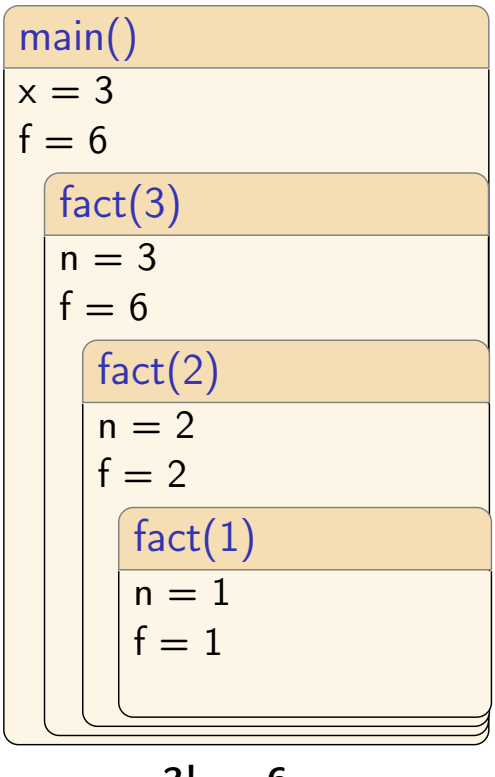


4
14
15

- Uma variável pode estar activa múltiplas vezes (em diferentes localizações na memória)
- Variáveis globais são "desaconselhadas":
  - Dificultam a compreensão do código (é preciso olhar para todo o programa para perceber como evolui o seu valor);
  - Comprometem a modularidade do código.
- Uma excepção comummente aceite é o caso das variáveis globais *read-only* (i.e. **constantes**).
- Uma *boa prática* em programação é manter o âmbito das variáveis tão restrito quanto possível

...mais um exemplo...

```
#include <stdio.h>

int fact(int n) {
 int f;
 if (n>1) {
  f = n * fact(n-1);
 } else {
  f = 1 ;
 }
 return f;
}

int main() {
 int x=3, f;
 f = fact(x);
 printf("%d! = %d\n", x, f);
 return 0;
}
```



3! = 6

Expressões

- Tal como a generalidade das linguagens, o C dispõe de um conjunto de operadores pré-definidos que permitem construir **expressões** a partir de expressões mais simples
- A cada expressão está associado um tipo de dados
- As **expressões** no C são formadas a partir de:

  Constantes (ou **literais**), como `123, 0xF3, -2, 1.32E2, 'A', ...`

  Variáveis o identificador de uma variável denota o valor dessa variável (do tipo declarado)

  Operações pré-definidas como as operações aritméticas, relacionais, lógicas, *bitwise*, *cast*s, de apontadores, de arrays, etc.

  funções C com tipo de retorno diferente de `void` (e.g. `2*quadrado(2)`)
- Como é habitual, existem regras de *precedência* e *associatividade* para desambiguar a construção de expressões (e.g. `3+2*2`, `2*3 > 5`)
- Em caso de necessidade (ou dúvida) devem-se usar parêntesis (e.g. `(3+2)*2`, `(2*3) > 5`).

Operadores aritméticos

- Estão definidos os operadores aritméticos habituais:

  unários: `-`

  binários: `+, -, *, /`
- com as regras de precedência usuais em matemática
- Estas operações estão disponíveis quer para inteiros quer para *float*s
- Para os inteiros, também está disponível a operação de "resto de divisão inteira" através do operador `%` (e.g. `5%2` avalia em `1`)

- Ainda para os inteiros, existem operadores que permitem operar ao nível da representação em bits (operadores *bitwise*).
- (obs: mais por curiosidade...) são eles: `<<, >>, ~, &, |, ^`

Conversão de tipos

- Nas operações binárias a selecção do tipo da operação é o máximo do tipo dos argumentos segundo a ordem:

  `int` < `unsigned int` < `long` < `unsigned long` < `float` < `double`.
- Diz-se por isso que existe uma **conversão de tipo implícita** do argumento com o tipo "menor".
- Exemplo:

  `5/2` é escolhido o tipo `int` (divisão inteira), pelo que o resultado é 2

  `5/2.0` é escolhido o tipo `float`, dando como resultado `2.5`.
- Outra situação onde pode ocorrer uma *conversão implícita* de tipos é nas atribuições: aí a conversão dá-se para o tipo da variável atribuída (mesmo que isso conduza a uma perca de informação)
- Pode-se no entanto explicitar a conversão com recurso a um ***cast***:

  (⟨type⟩**)** ⟨expr⟩
- O efeito de um *cast* é converter a expressão para o tipo "destino"
- Exemplo: `5/(float)2` já força a que o resultado da divisão seja um `float`.

Operador `sizeof`

- O operador `sizeof` permite determinar o espaço ocupado na memória pela representação de valores de um dado tipo.
- O seu argumento pode ser um tipo ou uma expressão

  sizeof **(**⟨type⟩**)** | ***sizeof*** **(**⟨expr⟩**)**
- O compilador sobstitui a expressão `sizeof(...)` pelo número de *byte*s ocupados.
- Exemplo (numa máquina concreta):

```
sizeof(char)   =  1
 sizeof(int)   =  4
sizeof(long)   =  8
sizeof(float)  =  4
sizeof(double) =  8
```

- Obs: quando se utiliza o `sizeof` com uma expressão, ela **não será avaliada** durante a execução do programa — o compilador só olha ao respectivo tipo.
- Exemplo: `sizeof(5/0.0)` retorna o mesmo que `sizeof(float)` (e não dá erro de divisão por zero).

Operadores relacionais e lógicos

- Em C, os valores boleanos são representados por inteiros.
  - 0 representa **Falso**
  - qualquer outro valor representa **Verdadeiro**
- Para comparar valores, dispomos dos operadores:

  `==, !=` teste de igual/diferente

  `>, >=, <, <=` testes de desigualdade
- Existem ainda as operações lógicas habituais

  `!` negação

  `&&, ||` conjunção e disjunção
- nas operações `&&, ||`, o argumento da direita só é calculado em caso de necessidade.

  Exemplo `(x!=0 && y/x==2)` nunca dará erro de divisão por zero!
- Outro operador útil é a **expressão condicional**.Tem a forma de uma operação ternária:

  $$\langle \text{expr} \rangle \mathbf{?} \langle \text{expr} \rangle \mathbf{:} \langle \text{expr} \rangle$$

  onde, dependendo de a primeira expressão ser verdadeira, avalia a segunda ou terceira expressões.
- Exemplo: `(x>=0 ? x : -x)` avalia no valor absoluto de `x`.

Expressões com Efeitos Laterais

- Já referimos que o resultado da avaliação de uma expressão é um valor de um dado tipo.
- Uma particularidade do C é que a avaliação de certas operações pode produzir "também" uma **alteração do estado do programa** (e.g. alterar o valor de uma variável).
- Dos exemplos mais úteis desse tipo são os **operadores de incremento/decremento**:

  `++x` (pré-incremento): como efeito lateral incrementa o valor armazenado na variável `x`, e como resultado retorna esse valor (já incrementado);

  `x++` (pós-incremento): como resultado retorna o valor de `x` (antes de ser alterado) e como efeito lateral incrementa o valor armazenado na variável `x`.

  De forma análoga, para os operadores de pré- e pós-decremento (`--x` e `x--`).

- Exemplo de utilização: vamos supor que se está num estado em que a variável inteira `x` tem armazenado o valor `5`. A avaliação das expressões seguintes produz como resultado/efeito:
  - `2 * ++x` retorna o valor 12 e `x` passa a valer 6.
  - `2 * x++` retorna o valor 10 e `x` passa a valer 6.

- A utilização de operadores de incremento/decremento inserida em expressões mais complexas é tipicamente apreciada por programadores experientes porque lhes permite ser muito concisos...
- ...mas conduz a código muito pouco legível, pelo que se recomenda que sejam sempre usados isoladamente.
- Exemplo de situação perigosa que resulta de não seguir a recomendação: qual é o resultado de avaliar `x=x++` ?

- Em rigor, a noção de *atribuição* que já foi apresentada é um exemplo de um efeito lateral na avaliação de expressões.

  > *(`x = e`) é uma expressão que como resultado retorna o valor da avaliação de `e`, e como efeito lateral atribuí à variável `x` esse valor.*

- Boa prática (ou recomendação veemente): nunca insira uma atribuição como sub-expressão de outra.
- Ainda relacionados com a atribuição, o C define operadores que combinam a atribuição com operações binárias (e.g. `+=`, `*=`, `...`)
- Estes operadores são simplesmente uma conveniência — e.g. `x += 3` é equivalente a `x = x+3`

Funções definidas pelo utilizador/bibliotecas

- Qualquer função definida em C pode ser usada na construção de expressões

  $$\langle \text{ident} \rangle \mathbf{(} \langle \text{expr} \rangle^{+} \mathbf{)}$$

  onde $\langle$ident$\rangle$ é o nome de uma função (previamente declarada/definida).
- O tipo da expressão resultante é o tipo de retorno da função.
- Os tipos das
- Outro exemplo de expressões que podem conter efeitos laterais é a invocação de funções definidas pelo programador... (e.g. `getchar()+1`)

Funções

- Funções são o principal elemento estruturante dos programas C
- Permitem decompor o problema computacional em sub-problemas de dimensão mais tratável.
- Formalmente, um programa em C consiste numa sequência de **declarações globais**:

  $$\begin{aligned} \langle \text{prog} \rangle &\equiv \langle \text{gdecl} \rangle^{+} \\ \langle \text{gdecl} \rangle &\equiv \langle \text{funDecl} \rangle \mid \langle \text{funDef} \rangle \mid \langle \text{varDecl} \rangle \mid \langle \text{typeDef} \rangle \end{aligned}$$

  Declaração de função informa o compilador da *assinatura* de uma função (i.e. nome e dos tipos dos argumentos/resultado);

  Definição de função , para além da assinatura inclui o **corpo da função** contendo o código C a executar pela função;

  Declaração de variável associa um identificador (nome de variável) a uma área de memória que alberga um valor de um dado tipo;

  Definição de tipo associa um identificador a um tipo de dados.
- Deve existir sempre, pelo menos, a função `main` (ponto de entrada na execução do programa).
  - assume normalmente o papel de "escalonador" da chamada a outras funções que realizam as tarefas pretendidas

Funções *puras* e *procedimentos*

- Já vimos que as funções C podem ser usadas em expressões
- São por isso o mecanismo indicado para codificar "funções matemáticas" como `sin, cos, fact, ...` (a esse respeito, de forma análoga ao que faríamos em, e.g. *Haskell*)
- Mas ATENÇÃO!!!
  - a avaliação de funções pode envolver *efeitos laterais* (e.g. alterar a memória ou realizar operações I/O)
  - ...esses efeitos irão ocorrer sempre que essa função for avaliada.
- Quando a avaliação de uma função em C não produz efeitos laterais, é normal chamar-se a essa função **pura**.
- Precisamente porque as funções em C são muitas vezes definidas "pelo efeito que produzem ao serem avaliadas" (e não pelo valor que calculam), existe o tipo `void` para assinalar que a função não retorna qualquer valor.
- Funções com tipo de retorno `void` são normalmente designadas por **procedimentos**.

Declaração de Funções

- A sintaxe para a declaração de funções é:

$$\begin{aligned} \langle \text{funDecl} \rangle &\equiv \langle \text{type} \rangle\ \langle \text{ident} \rangle \mathbf{(} \langle \text{paramList} \rangle \mathbf{);} \\ \langle \text{paramList} \rangle &\equiv \epsilon \mid \langle \text{type} \rangle\ \mathbf{,}\ \langle \text{paramList} \rangle \end{aligned}$$

  onde $\langle \text{type} \rangle$ é denota um tipo de dados do C
- Obs: na declaração podem-se incluir também os nomes dos parâmetros (útil para efeitos de documentação).
- Uma declaração de função deve ser utilizada sempre que a invocação de uma função antecede a sua definição;
- Em particular, as funções de bibliotecas nunca são definidas pelo que a sua utilização deve ser precedida das declarações respectivas;
- São normalmente inseridas nos programas por inclusão de *header-files* apropriadas.

Algumas funções úteis

- `putchar` — imprime em `stdout` um único caracter
- `printf` — imprime em `stdout` texto (*string*) formatado

```
int putchar(const int);
int printf(const char*, ...);
```

- No `printf`, o primeiro argumento (de tipo `char*`) é uma *String* que contém o *template* do texto a visualizar
  - pode conter "buracos" (assinalados com % seguidos de uma ou mais letras que detalha qual o tipo da informação a incluir, e como é que deve ser visualizada)
  - ver página do manual com descrição dos modificadores (executar: `man 3 printf`)
- Os argumentos seguintes vão conter as expressões que serão avaliadas nos valores que irão preencher esses "buracos" (devem por isso ser tantos como os "buracos")
- O `int` retornado é normalmente ignorado...

- `getchar` — lê do `stdin` um único caracter
- `scanf` — lê do `stdin` *input* formatado (*string*) formatado

```
int getchar(void);
int scanf(const char*, ...);
```

- Tal como no `printf`, no`scanf` o primeiro argumento é uma *String* que contém o *template* dos dados a ler
- Os argumentos seguintes contém **referências** para as variáveis a ler (nome da variável antecedido pelo caracter `&`).
- Exemplos de utilização:

```
int x;
scanf("%d", &x); // ler um inteiro

char c;
// vamos violar uma "boa prática" para ilustrar uma utilização comum
while ((c=getchar())!=EOF) //enquanto não acabar o ficheiro..
{ putchar(c); }           // imprime caracter lido
```

Definição de Funções

- A sintaxe para a definição de funções é:

$$\begin{aligned} \langle \text{funDef} \rangle &\equiv \langle \text{type} \rangle\ \langle \text{ident} \rangle \mathbf{(} \langle \text{paramList} \rangle \mathbf{)}\ \langle \text{instrBlock} \rangle \\ \langle \text{paramList} \rangle &\equiv \epsilon \mid \langle \text{type} \rangle\ \langle \text{ident} \rangle\ [\mathbf{,}\ \langle \text{paramList} \rangle] \\ \langle \text{instrBlock} \rangle &\equiv \mathbf{\{}\ \langle \text{varDecl} \rangle^*\ \langle \text{instr} \rangle^*\ \mathbf{\}} \end{aligned}$$

- Parâmetros das funções comportam-se como variáveis locais que são inicializadas com os valores passados como argumentos na invocação da função.
  - podem por isso ser alterados durante a execução (a menos que contenham o qualificador `const`)
- O **bloco de instruções** contém
  1. declaração das variáveis locais
  2. instruções a executar durante avaliação da função.
- Os nomes das variáveis locais de um bloco devem ser todos distintos entre si (obs: nem faria sentido se assim não fosse...)
- Mas podem-se "sobrepor" a outras variáveis visíveis nesse bloco
  - E.g. uma variável local com o mesmo nome de uma variável global — nesse caso, e enquanto a variável local estiver visível, perde-se a capacidade de aceder/alterar a variável global com o mesmo nome

Instruções básicas

$$\begin{aligned} \langle \text{instr} \rangle \equiv\ & \mathbf{;} \\ \mid\ & \langle \text{expr} \rangle \mathbf{;} \\ \mid\ & \langle \text{instrBlock} \rangle \mathbf{;} \\ \mid\ & \textbf{\textit{return}}\ \langle \text{expr} \rangle \mathbf{;} \\ \mid\ & \ldots \end{aligned}$$

- Note que todas as instruções em C terminam com **;** (ponto e vírgula).

  Instrução vazia — "não faz nada" (útil apenas em situações muito particulares)

  Expressão — qualquer expressão pode ser usada como uma instrução
  - o impacto dessa instrução é o **efeito lateral** da avaliação da expressão (e.g. atribuição, incremento, etc)
  - o valor final da expressão é ignorado

  bloco — impacto desta instrução consiste em avaliar todo o bloco (usado essencialmente para formar instruções compostas, como `if`s ou ciclos)

  return — finaliza avaliação da função

- A utilização mais comum para uma instrução constituída por uma expressão é a **atribuição** (e.g. `x=3;`)
- Boa prática: destacar as atribuições sempre como instruções isoladas
- Outros exemplos:
  - As instruções `x++;` e `++x;` são equivalentes em C
    - De facto, o efeito lateral produzido por cada uma das expressões utilizadas é o mesmo: incrementar a variável `x`
    - O facto de o "valor" das expressões ser distinto torna-se irrelevante porque, nas instruções, esse valor é descartado
    - Levando ao limite do "disparate", também podemos dizer que qualquer uma das instruções é equivalente a `5*x++;`
- A instrução `return` termina a avaliação da função corrente.
  - especifica o valor retornado pela função (se tipo de retorno diferente de `void`)
  - o `return` pode ser dispensado quando tipo de retorno é `void` (nesse caso, execução termina no final do bloco de instruções da função).

Instruções condicionais

$$\langle \text{instr} \rangle \equiv \quad \ldots \mid \textbf{\textit{if}}\ \textbf{(}\langle \text{expr} \rangle\textbf{)}\ \langle \text{instr} \rangle\ [\textbf{\textit{else}}\ \langle \text{instr} \rangle]\textbf{;} \mid \ldots$$

- A instrução `if` permite executar uma instrução apenas quando uma condição se verificar
- Pode opcionalmente conter a cláusula `else` que especifica uma instrução a ser executada quando a condição não se verifica (se não existir, não executa nada nesse caso)
- Normalmente, utilizam-se **blocos de instruções** para permitir executar múltiplas instruções condicionadas pela mesma condição
- ...e tem a vantagem de tornar o código mais claro (boa prática)

- Exemplos de utilização:

```
int x;

scanf("%d", &x);

if ((x%2)==1) {
  x--;
}
```

```
if (a == 10) {
  printf("igual a 10\n");
} else if (a < 10) {
  if (a > 5) {
    printf("entre 5 e 10\n");
  }
} else {
  printf("maior ou igual a 10\n");
}
```

- Erros comuns:

```
if (a>0);
  printf("Maior que zero\n");
```

```
if (a>0)
  printf("Maior que...");
  printf("...zero\n");
```

```
int a=0;

if (a=10) {
  printf("Igual a 10???\n");
}
```

- Boa prática: usar sempre blocos nos `ifs`

Selecção (`switch`)

- O C disponibiliza uma construção onde é possível discriminar diferentes valores de uma expressão inteira.

$$\begin{aligned} \langle \text{instr} \rangle &\equiv \ldots \mid \textbf{\textit{switch}}\ \textbf{(}\langle \text{expr} \rangle\textbf{)\{}\langle \text{switchEntry} \rangle^{+}\textbf{\}} \mid \ldots \\ \langle \text{switchEntry} \rangle &\equiv \langle \text{selCase} \rangle\textbf{:}\ \langle \text{instr} \rangle^{*} \\ \langle \text{selCase} \rangle &\equiv \textbf{\textit{case}}\ \langle \text{intLiteral} \rangle \mid \textbf{\textit{default}} \end{aligned}$$

```
switch (a) {
case '1':
  printf("um 1\n");
  break;
case '2':
case '3':
  printf("um 2 ou 3\n");
  break;
default:
  printf("Nao sei o que e\n");
  break;
}
printf("...continua o prog!\n");
```

- A instrução `break` permite terminar a execução do `switch` e prosseguir na instrução que se lhe segue.
- Não havendo saída explícita, a execução prossegue através dos casos subsequentes.
- O `default` permite concordância com qualquer valor.

Ciclos

- Para codificar acções **repetitivas** ou **iterativas** (i.e. percorrer uma gama de valores), o C possibilita a construção de **ciclos**.
- Trata-se de uma construção de natureza **imperativa** — de facto, envolve um "salto para trás" na execução do programa.
- Para maior flexibilidade, existem três construções diferentes para ciclos em C:

$$\begin{aligned}\langle \text{instr} \rangle \equiv\ & \ldots \\ \mid\ & \textbf{\textit{while}}\ \textbf{(}\langle \text{expr} \rangle\textbf{)}\ \langle \text{instr} \rangle \\ \mid\ & \textbf{\textit{do}}\ \langle \text{instr} \rangle\ \textbf{\textit{while}}\ \textbf{(}\langle \text{expr} \rangle\textbf{)} \\ \mid\ & \textbf{\textit{for}}\textbf{(}\langle \text{expr} \rangle\textbf{;}\langle \text{expr} \rangle\textbf{;}\langle \text{expr} \rangle\textbf{)}\ \langle \text{instr} \rangle \\ \mid\ & \ldots\end{aligned}$$

- A instução interior do ciclo é designada por **corpo do ciclo**
- Boa prática: tal como nos `ifs`, é conveniente que o corpo dos ciclos seja sempre um bloco de instruções (mesmo que contenha uma única instrução)

Ciclo `while`

- A versão mais simples é o ciclo `while`: "*executa o corpo do ciclo enquanto uma dada condição for verdadeira*"
- Mais em detalhe temos:
  1. é avaliada a condição
  2. se for falsa, sai do ciclo
  3. se for verdadeira: executa o **corpo** do ciclo e volta a (1)
- Note-se que pode acontecer de nunca executar o corpo do ciclo...
- ...ou nunca sair do ciclo.
- Exemplo:

```
char letra = 'A';
while (letra <= 'Z') {
  printf("O codigo de \%c e \%d\n", letra, letra);
  letra++;
}
```

- Imprime uma tabela com o código de todas as letras maiúsculas.
- Qual é o valor de `letra` quando o ciclo terminar?

Ciclo `for`

- É normal os ciclos conterem as seguintes componentes/fases:
  - **inicialização** onde se atribuem os valores iniciais para as variáveis que controlam o ciclo;
  - **condição** que regula se o corpo do ciclo deve ser executado;
  - **corpo do ciclo** (acção repetitiva realizada pelo ciclo)
  - **incremento** onde se actualizam as variáveis que controlam o ciclo.
- O ciclo `for` é uma sintaxe para ciclos onde estas componentes aparecem destacadas. Assim

  ***for*(init; cond; incr) body;**

  é equivalente a

  init; *while* (cond) {body; incr; }
- Exemplo:

```
char letra;
for (letra='A'; letra <= 'Z'; letra++) {
  printf("O codigo de \%c e \%d\n", letra, letra);
}
```

Ciclo `do_while`

- O ciclo `do_while` verifica a condição do ciclo apenas depois de executar o corpo do ciclo.
- Dessa forma o corpo do ciclo será executado sempre, pelo menos, uma vez.
- Tal como no caso do `for`, também podemos exprimir o funcionamento do ciclo `do_while` à custa de uma tradução para um ciclo `while` simples. Assim: Assim

  ***do* {body;} *while*(cond);**

  é equivalente a

  body; *while* (cond) {body;};
- Exemplo:

```
char letra = 'A';
do {
  printf("O codigo de \%c e \%d\n", letra, letra);
  letra++;
} while (letra <= 'Z');
```

`break` e `continue`

- Duas instruções que permitem um maior controlo sobre a execução dos ciclos:

  `break` sai imediatamente do ciclo

  `continue` sai da iteração corrente

- Quando utilizado com pouco critério, pode conduzir a código ilegível — utilizar com "muito" cuidado
- Um padrão que se encontra em certos programas avançados...

```
int x, sum=0;
while (1) {
  scanf("%d", &x);
  if (x==0) break;
  sum += x;
}
```

```
int x, i=0, sum=0;
while (i < 10) {
  scanf("%d", &x);
  if ((x%2)==0) continue;
  sum += x;
}
```

- Obs: simplesmente para efeitos de ilustração! Que não se entenda como uma recomendação...

Endereços de memória

- As variáveis de um programa C estão armazenadas em determinadas posição da memória
- ...mas o prograd or normalmalmente não necessita de lidar com isso explicitamente — essas variáveis são manipuladas através do seu nome
- Mas a linguagem C oferece também a capacidade de operar na memória praticamente ao nível do *hardware*, disponibilizando formas de ler/escrever o conteúdo de um qualquer **endereço de memória**
- Para tal, disponibiliza tipos de dados para manipular endereços de memória: os **apontadores** para um dado tipo

$$\langle \text{type} \rangle \quad \equiv \ldots \mid \langle \text{type} \rangle *$$

- E.g. um valor do tipo `int*` é um endereço (normalmente 64bit) para uma zona da memória onde deverá estar armazenado um valor to tipo `int`

Operações sobre apontadores

- O C lida com uma declaração de uma variável com tipo apontador exactamente como com os restantes tipos:
  - reserva uma zona da memória com o tamanho apropriado para armazenar um endereço de memória
  - o valor armazenado nessa zona da memória (o endereço) vai poder ser utilizado para consultar/alterar a informação contida nesse endereço
- Existem duas **operações** elementares sobre os apontadores:

$$\begin{aligned} \langle \text{expr} \rangle &\equiv \ldots \mid \&\langle \text{lvalue} \rangle \mid \ldots \\ \langle \text{lvalue} \rangle &\equiv \ldots \mid *\langle \text{expr} \rangle \mid \ldots \end{aligned}$$

`&` (**endereço de**): avalia no endereço de memória onde está armazenado o ⟨lvalue⟩ respecivo;

`*` (**conteúdo de**): avalia na zona de memória "apontado por" um dado apontador

Exemplo

```
#include <stdio.h>
int main () {
  int i, j, *a, *b;
  i=3; j=5;
  a = &i; b = &j;
  i++;
  j = i + *b;
  b = a;
  j = j + *b;
  printf ("%d\n", j);
  return 0;
}
```

main()

(@0xBFF15CA4) i = 4

(@0xBFF15CA8) j = 13

a = 0xBFF15CA4

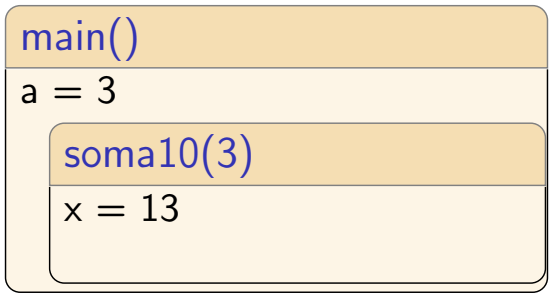
b = 0xBFF15CA4

13

Passagem de parâmetros **por valor**

- Já vimos que, numa função, os seus parâmetros comportam-se como variáveis locais inicializadas **com (uma cópia) do valor** passado como argumento (diz-se por isso que no C os parâmetros são passados **por valor**)
- Exemplo:

```
#include <stdio.h>
void soma10(int x) {
  x = x + 10;
}
int main() {
  int a=3;
  soma10(a);
  printf("a = %d\n", a);
  return 0;
}
```

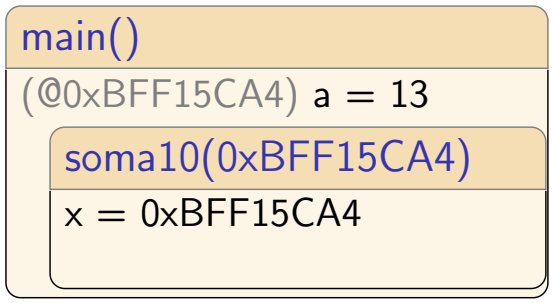


a = 3

Passagem de parâmetros **por referência**

- Por vezes, pretende-se que uma função possa efectivamente alterar o valor de uma variável que lhe seja passada como argumento (e.g. como no `scanf`)
- No C, isso é conseguido através da utilização de apontadores
  - em vez de receber um argumento de um dado tipo, a função recebe um apontador para esse tipo
  - em vez de manipular o valor do parâmetro, a função manipula o "conteúdo" apontado pelo apontador
- Voltando ao exemplo anterior:

```
#include <stdio.h>
void soma10(int *x) {
  *x = *x + 10;
}
int main() {
  int a=3;
  soma10(&a);
  printf("a = %d\n", a);
  return 0;
}
```



a = 13

Modificador const em apontadores

- No caso dos apontadores, o moficador `const` pode-se referir a dois aspectos distintos
  1. à variável "apontador" propriamente dito (i.e. o apontador, depois de inicializado, nunca mais pode ser alterado)
     - nesse caso, o modificador surge depois de `*`

```
int x=1, y=2;
int * const apt = &x;
apt = &y; //ERRO!!!
```

  2. à zona de memória referenciada pelo apontador (i.e. o apontador não poderá ser utilizado para alterar o seu conteúdo)
     - nesse caso, o modificador surge antes de `*`

```
int x=1, y=2;
int const *apt = &x;
apt = &y; //OK
*apt = 3; //ERRO!!!
```

- Numa declaração podem ser combinados ambos os aspectos (e.g. `int const * const apt;`)

Dificuldades na utilização de apontadores

- A utilização de apontadores é inerentemente mais complexa do que a utilização de variáveis de tipos básicos
  - acede-se/altera-se a informação de forma indirecta
  - é necessário ter pleno controlo sobre o endereço armazenado no apontador:
    - corresponde a uma zona de memória com um valor do tipo consistente com o tipo do apontador;
    - o acesso a essa zona de memória é legítimo (e.g. não corresponde a uma variável cujo tempo de vida expirou)
- Um erro comum é retornar um apontador para uma variável local

```
int* func(int x){
   int res;
   res = x+1;
   return &res; // PROBLEMA!!!
}
```

- Erro usual quando se acede a um apontador com um valor inapropriado: `segmentation fault`

Variáveis indexadas (Arrays)

- O C (como a generalidade das linguagens imperativas) permite operar com **variáveis indexadas** (ou *arrays*), i.e. variáveis que armazenam múltiplos valores de um dado tipo
- Os arrays são declarados acrescentando ao nome da variável o número de elementos pretendidos entre parêntesis rectos (e.g. `int a[10];`)
  - o compilador reserva uma zona de memória contígua para todos os elementos do array
  - na daclaração, é possível inicializar imediatamente o array com a seguinte sintaxe

```
int a[5] = {5,4,3,2,1};
```

- Para se aceder ao valor de um elemento de um array utiliza-se também o índice pretendido entre parêntesis rectos (e.g. `a[55]`)
  - os índices admissíveis para um array de `N` elementos são os números entre 0 e $(N-1)$
  - **ATENÇÃO:** o C não verifica os índices utilizados nos acessos são válidos!
- De resto, um elemento de um array com um dado tipo utiliza-se como uma variável desse tipo

Arrays vs. Apontadores

- O nome de um array é visto pelo C como um apontador para o seu primeiro elemento
- De facto, essa fusão de conceitos entre arrays e apontadores ainda vai mais longe:
  - podemos somar ou subtrair constantes inteiras a apontadores (aritmética de apontadores)

```
int a[5] = {5,4,3,2,1};
*a = *(a+2); // equivalente a: a[0] = a[2];
```

  - o efeito de somar uma constante a um apontador é o de avançar na memória o número correspondente de *slots* do tipo apontado
- Assim, em C, "`a[k]`" é simplesmente uma sintaxe alternativa para "`*(a+k)`"
- É evidente que só se deve utilizar aritmética de apontadores quando se tem garantia do que está armazenado nas posições contíguas ao apontador (que, na prática, só se verifica nos arrays!)

Arrays como argumentos de funções

- Outro ponto onde o C identifica arrays com apontadores é nos parâmetros das funções
  - um parâmetro de tipo array é sempre visto como um apontador para o tipo do array
  - Exemplo: as seguintes declarações de função são todas equivalentes

```
int sumArray(int a[100]);
int sumArray(int a[]);
int sumArray(int* a);
```

- Como consequência, em C todos **os arrays são sempre passados por referência**

```
/* ler o conteúdo de um array de N elementos */
void leArray(int a[], int N) {
  int i;
  for(i=0; i<N; i++) {
    printf("a[%d]=", i);
    scanf("%d", a+i);  //porque é que isto funciona???
  }
}
```

É possível uma função retornar um array?

- O C não permite definir uma função que retorne um array
- ...mas pode, naturalmente, retornar um apontador — afinal, não é isso a mesma coisa?
  - Cuidado! Devemos garantir que o apontador retornado é para uma zona de memória válida
- Erro típico (porquê??? correcção como exercício)

```
int* revArray(int a[], int N) {
  int i, new[N];
  for(i=0; i<N; i++) new[i] = a[N-1-i];
  return new; //PROBLEMA!!!
}
```

- Utilização legítima:

```
/* retorna array com maior soma */
int* maxSum(int a[], int b[], int N) {
  int i, suma=0, sumb=0;
  for (i=0; i<N; i++) {
    suma += a[i]; sumb += b[i];
  }
  return (suma >= sumb ? a : b);
}
```

```
#include <stdio.h>
#define N 10
int main() {
  int x[N], y[N], *a;
  leArray(x, N); leArray(y, N);
  a = maxSum(x, y, N);
  mostraArray(a, N);
}
```

Arrays de caracteres (*strings*)

- Uma **sequência de caracteres** (*string*) é representada em C como um array do tipo `char` (e.g. `char nome[50]`)
- Todas as *string*s em C terminam com um carácter especial: **\0**
  - por isso, ao declarar um array para armazenar uma *string*, devemos considerar o espaço ocupado pelo terminador
- Exemplos de declarações:

```
char texto[100]; // suporta strings de comp. máx. 99

const char nome[] = {'M', 'a', 'r', 'i', 'a', '\0'};

char str[11] = "Uma string"; // sintaxe alternativa de inic.
```

- Tipicamente, algoritmos sobre strings vão percorrer a string até encontrar o carácter \0

```
int strlen(char *s) {
  int count=0;
  for (; *s != '\0'; s++)
    count++;
  return count;
}
```

Arrays multidimensionais

- O C suporta ainda variáveis indexadas multidimensionais (e.g. matrizes)
- Na declaração, incluem-se as várias dimensões (cada uma entre parêntesis rectos)

```
int m1[2][4] = {{1,2,3,4}, {5,6,7,8}};
float cubo[4][4][4];
```

- Tal como no caso unidimensional, o compilador atribuí uma zona de memória contígua a todo o array
- Um array multidimensional acaba por ser uma abstracção do compilador para um array unidimensional (cujo tamanho é o produto das dimensões)
  - Um acesso `a[i][j]` (do tipo `int[M][N]` é equivalente a `*(((int*)a)+i*N+j)`
  - obs: mas, sem o *cast*, o compilador já entra em linha de conta com o número de colunas — fica simplesmente `*(*(a+i)+j))`
- O que explica porque é que, quando passados como parâmetros em funções, só a primeira dimensão pode ser omitida

```
void mulMat(float m1[][N2], float m2[N2][N3]) {... }
```