Programação Imperativa

Licenciatura em Ciências da Computação

Universidade do Minho

M. B. Barbosa AT di.uminho.pt

2013/2014

Apresentação e Introdução

Apresentação

Linguagens de Alto e Baixo Nível

Programação Imperativa

Linguagens de programação - Máquina

- Os computadores desempenham tarefas complexas.
- No entanto têm uma “mente” simples e dócil.
- Contrário ao possa parecer, não têm vontade própria.
- Necessitam de instruções exactas que possam ser executadas pelo “hardware”.
- Essas instruções têm de ser dadas numa linguagem muito limitada: a **linguagem da máquina**.
- As operações compreendidas pelo hardware são muito simples, por exemplo: “copia este número daqui para ali”, “multiplica estes dois números”, etc.
- Além disso, a representação natural dos programas, e dos dados por eles manipulados, é a codificação binária.

Linguagens de programação - Assembly

- Nos anos 40 e 50 os programas eram escritos directamente em linguagem máquina.
- Com sorte, os programas podiam ser escritos em **linguagem assembly**.
- O assembly é um conjunto de mnemónicas que evita codificar as instruções usando os seus códigos numéricos.
- A tradução para linguagem de máquina é feita por um **assembler**.
- Hoje o assembly é utilizado apenas para aquelas partes dos programas onde a velocidade de execução é crítica.

Linguagens de programação - Alto Nível

- Hoje, a maioria dos programas são 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 programas para linguagem máquina/assembly.
- O mesmo programa pode ser traduzido por diferentes compiladores para diversas máquinas (portabilidade).

Programação Imperativa

- A computação é descrita em termos de instruções que alteram um **estado** mantido pelo programa.
- Os programas são **sequências de comandos** que são executados pelo computador.
- Está mais próxima da forma como os computadores funcionam internamente:
  - o estado consiste nos dados armazenados (e.g. na memória).
  - o processador executa instruções simples sequencialmente.
- O conceito de função (procedimento ou subrotina) surge como forma de estruturar os programas.
- Por oposição, na programação declarativa (e.g. funcional) os programas descrevem apenas o resultado pretendido.

Programação Imperativa$_2$

Desenvolver um programa implica:

- Definir claramente o **problema** que se pretende resolver.
- Desenhar um **algoritmo** que resolva esse problema.
- **Implementar** esse algoritmo numa linguagem de programação.

Um algoritmo é um conjunto bem definito de instruções que indicam como resolver um determinado problema.

A sua implementação numa linguagem de alto nível implica:

- Traduzir o algoritmo para essa linguagem, armazenando o programa num conjunto de ficheiros.
- Compilar os ficheiros para produzir um novo conjunto de ficheiros contendo código objecto.
- Completar (ligar) esses ficheiros com bibliotecas de código pré-compilado.

Tour da Linguagem C

Origem e Evolução

Principas Características

Primeiro Exemplo

Processo de Compilação

Origens e Contexto

- Desenvolvida originalmente em 1972 por Dennis Ritchie da Bell Labs para implementar o Unix.
- Vocacionada para o desenvolvimento de software ao nível do sistema.
- Também muito utilizada para o desenvolvimento de aplicações.
- 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++ e o Java.

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: 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 tempo de aprendisagem é lento.
- 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: o exemplo "hello, world!"

```
#include <stdio.h>
int main(void)
{
    printf("hello, world!\n");
    return 0;
}
```

1. Edite um ficheiro de texto chamado `hello.c` com este conteúdo (e.g. utilizando `Emacs`, `Vim`, ou `TextMate`).
2. Compile o programa e execute-o num terminal:

```
> gcc hello.c -o hello
> ./hello
hello, world!
>
```

3. It Works!

Estrutura de um programa em C

- Os programas em C são construídos a partir de **funções**.
- As funções:
  - Recebem informação – **argumentos**.
  - Processam essa informação, possivelmente utilizando outras funções.
  - Retornam um resultado.
- A função `main` é o ponto de entrada no programa:
  - Recebe informação externa que o programa deve processar.
  - Controla a execução do programa, possivelmente chamando outras funções.

Código fonte vs Código executável

- O código C que escrevemos chama-se **código fonte**
- Existem dois tipos de ficheiros com código fonte:
  - Ficheiros "principais" (com extensão `.c`).
  - Ficheiros "cabeçalho" ou "header files" (com extensão `.h`).
- O compilador transforma código fonte em **código objecto** (ficheiros com extensão `.o`).
- O "linker" transforma o(s) ficheiro(s) com código objecto em ficheiros **executáveis** (sem qualquer extensão em particular).
- O programa `gcc` funciona como compilador e como "linker".
  - O comando "`gcc hello.c -o hello`" compila o ficheiro `hello.c` para `hello.o`.
  - Depois combina este ficheiro com um conjunto de bibliotecas pré-compiladas para gerar o ficheiro executável `hello`.
  - Finalmente, remove o ficheiro `hello.o`.

O essencial do C

- Funções
- Pré-Processador
- Variáveis e Constantes
- Expressões
- Comentários

Funções

A função é o elemento estrutural da linguagem C:

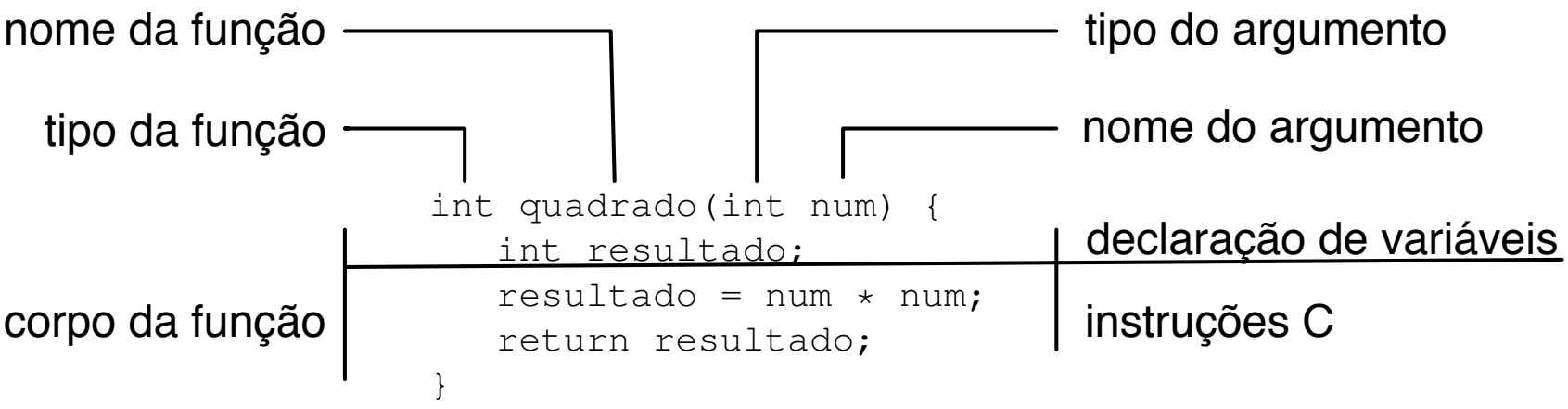
- É um conjunto de operações C.
- Desempenha uma operação mais complexa do que as operações base do C.
- Não deve ser tão complexa que seja difícil de compreender.

Os programas são construidos como *camadas* de funções:

- As funções de baixo nível desempenham tarefas simples.
- As funções de alto nível utilizam as anteriores.

Anatomia de uma função

Exemplo de uma função de baixo nível:



Chamar ou Invocar uma Função

Para chamar a função anterior:

```
/* noutra funcao ... */
int res;
int i = 10;
res = quadrado(10);
res = quadrado(5 + 5);
res = quadrado(i);
res = quadrado(i*5 + i/2);
```

A função recebe o parâmetro e retorna o resultado.

Nem todas as funções retornam valores

```
void escreve_num(int i)
{
   printf("num: %d\n", i);
}
```

- O tipo do resultado é `void`.
- O `return` é desnecessário, a não ser que seja colocado no meio da função.
- Para chamar esta função:

```
/* Noutra funcao ... */
int i = 10;
escreve_num(20);
escreve_num(i);
escreve_num(i*5 + i/2);
```

Nem todas as funções recebem argumentos

```
int cinco(void)
{
   return 5;
}
```

- Mais uma vez assinala-se com `void`.
- Para chamar a função:

```
int valor;
valor = cinco();
```

Funções importantes (1)

`int main(void)` – Todos os programas executáveis devem ter uma.

- Função especial que representa o início da execução.
- Para já, vamos ignorar os parâmetros da função.
- A função `exit(int)` permite terminar o programa.
- Um valor 0 significa terminação sem problemas.

```
#include <stdlib.h>

int main(void) {
   extern int quadrado(int);
   int solucao;

   solucao=quadrado(5);
   exit(0);
}
```

Funções importantes (2)

`int printf(const char *, ...)` – Gerar output.

- O primeiro parâmetro define um padrão a visualizar.
- Esse padrão tem "buracos" assinalados com `%`.
- Os buracos correspondem a valores a definir durante a execução do programa (`%d`, `%c`, `%f`, `%s`, ...).
- Os parâmetros seguintes fornecem os valores necessários para preencher esses buracos, e devem ser compatíveis.

```
int main(void) {
   extern int quadrado(int);
   int solucao;

   solucao=quadrado(5);
   printf("O quadrado de 5 e' %d.\n", solucao);
   return 0;
}
```

Funções importantes (3)

`int scanf(const char *, ...)` – Ler input.

- O primeiro parâmetro define um padrão a ler.
- Esse padrão tem "buracos" assinalados com `%`.
- Os buracos correspondem a valores a ler para variáveis (`%d`, `%c`, `%f`, `%s`, `...`).
- Os parâmetros seguintes fornecem as variáveis que serão preenchidos com os dados lidos.

```
int main(void) {
   int solucao;
   scanf("%d",&solucao);
   solucao=quadrado(5);
   printf("O quadrado de 5 e' %d.\n", solucao);
   return 0;
}
```

O Pré-Processador

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

- Para todas as directivas `#include`, inclui todo o código do ficheiro cabeçalho (*header file*) correspondente.
- 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.
- Para todas as directivas `#define`, substitui o valor do macro no em todas as ocorrências no código.
- Os macros permitem escrever código facilmente parametrizável.

Variáveis e Constantes

- **Constantes** Representam valores que nunca se alteram durante o programa.
- **Variáveis** Representam uma parte da memória do computador, cujo valor se vai alterando.

```
j = 15;
j = j + 4;
```

Os literais `4` e `15` representam valores constantes.

O nome `j` representa uma posição de memória, e.g. no endereco 4566:

- A primeira instrução preenche a posição de memória representada por `j`.
- A segunda instrução provoca três acções:
  1. ler o valor da posição de memória,
  2. somar-lhe 4,
  3. actualizar a posição de memória.
- O modificador `const` transforma uma variável num depósito de um valor constante.

Nomes

Em C é necessário nomear variáveis, constantes, funções, etc.

As regras de criação de nomes são as mesmas:

- Podem conter letras, números e o caracter _
- Têm de começar por uma letra ou caracter _

Os nomes são sensíveis a letras maiúsculas e minúsculas.

As palavras reservadas não podem ser usadas para nomes.

Devem ser evitados nomes utilizados em bibliotecas e em C++.

Não existe comprimento máximo para o nome (ANSI prevê pelo menos 31).

Boas práticas:

- Utilizar minúsculas para nomes de variáveis.
- Utilizar maiúsculas para macros.
- **Escolher nomes informativos**

Palavras reservadas

auto	enum	restrict	unsigned
break	extern	return	void
case	float	short	volatile
char	for	signed	while
const	goto	sizeof	continue
if	default	inline	struct
do	int	switch	double
long	typedef	else	register
union	...		

Expressões

Uma expressão é uma combinação de operadores, literais, e nomes.

Uma expressão representa a computação de um valor.

Exemplos: `5`, `5 * j + 6`, `f()`, `g()/5`.

Uma **atribuição** (=) altera o valor de uma posição de memória.

Essa posição será, muitas vezes, referida por uma variável.

O novo valor é fornecido por uma expressão.

Exemplo: `resultado = num * num;`

Comentários

```
/* Isto e um comentario. */
/*
 * Um comentario pode ocupar
 * diversas linhas
 */
// Isto nem sempre e um comentario!

/*
 * area: calcula a area de um circulo
 * argumentos: r: raio do circulo
 * retorna: a area calculada
 */
double area(double r) {
    double pi = 3.1415926;
    return (pi * r * r);
}
```

Comentários: Boas Práticas (1)

- Os comentários são vitais à boa programação.
- Servem para:
  - Explicar como as nossas funções devem ser utilizadas.
  - Explicar como as nossas funções funcionam.
  - Explicar o que não é evidente e não óbvio.
- Quem lê os comentários?
  - Qualquer pessoa que queira modificar o nosso código.
  - Nós próprios, algumas semanas, meses ou anos depois.

Comentários: Boas Práticas (2)

- Colocar comentários mesmo antes das funções:
  - Objectivo da função.
  - O que representam os argumentos.
  - O significado do resultado.
- Comentar tudo o que não é obvio.
- Presumir que outras pessoas vão ler o código.
- O estilo (ortografia e gramática) é importante!
- Comentários mal feitos $\Longrightarrow$ professor dá má nota.

Aula 4 – Tipos de Dados Simples

Tipos de Dados

Declaração de variáveis

Tipos Aritméticos

Arrays e Strings (Antevisão)

Tipos de Dados

A classificação da informação em **tipos** é central às linguagens modernas:

- Permite trabalhar sobre abstracções complexas.
- O programador pensa em termos de coisas familiares:
  - valores inteiros,
  - vírgula flutuante,
  - caracteres alfa-numéricos,
  - ...

  sobre as quais estão definidas operações naturais.
- O computador continua a lidar com objectos físicos.
- O compilador assegura que o código executado pela máquina implementa correctamente as abstracções entendidas pelo programador.

Sistema de Tipos do C

Tipos de Dados

- `void`
- Simples
  - Apontadores
  - Tipos numéricos (`short`,`long`, `signed`,`unsigned`)
    - Inteiros – `int`, `char`
    - Vírgula Flutuante – `float`, `double`
  - `enum`
- Agregados ou Compostos
  - Arrays
  - `struct`
  - `union`

Declaração de Variáveis e Funções

- Todos os dados manipulados em C têm de ter um tipo.
- Cada variável tem **apenas um tipo**.
- Cada variável tem de ser **declarada** antes de usada.

- Exemplos:

```
int i;    /* nome = i  tipo = int */
char c;   /* nome = c  tipo = char */
double d;
float some_float = 3.14;
```

- As declarações devem aparecer **antes** do código executável.
- As variáveis podem ser inicializadas na declaração (senão?).
- Os valores boleanos são representados como inteiros (Zero/Não Zero).

Variáveis locais e globais (1)

- A declaração de uma variável pode ser:
  - Local: dentro (e acessivel apenas) de uma função.
  - Global: fora (e sempre acessivel) de qualquer função.

```
int x;          /* Variavel global */
int y = 10;   /* Variavel global inicializada */
int foo(int z)
{
  int w;    /* variavel local */
  x = 42;   /* atribuicao a variavel global */
  w = 10;   /* atribuicao a variavel local */
  return (x + y + z + w);
}
```

Variáveis locais e globais (2)

- Em geral, **evitar variáveis globais**!
- Podem ser alteradas em qualquer parte do código.
- Tornam o *debugging* mais difícil.
- Nunca são necessárias, embora por vezes sejam convenientes.
- A excepção é a declaração de constantes globais.

Tipos nas Funções

- As funções também têm de ser declaradas.
- Essa declaração pode ser apenas uma identificação do tipo:

```
int func(float a, float b);
```

- Neste caso, a implementação aparecerá tipicamente mais adiante ou noutro ficheiro C.
- O tipo da função inclui o tipo de cada parâmetro e o do resultado.

```
int func(float a, float b) {
    int c;
    ...
    return c; /* O tipo de c tem de ser int */
}
```

- As variáveis locais são declaradas no início das funções.

Números inteiros: `int` e `char`

- O tipo `int` geralmente representa um número de 32 bits.
- Mas pode ser de 64 bits, dependendo da máquina.

- O modificador `long` permite obter inteiros “maiores”.
- Hoje em dia é geralmente igual a `int`.
- Modificador `short` geralmente fornece inteiros de 16 bits.

- O tipo `char` representa um inteiro de 8 bits.
- Todos estes tipos representam os inteiros com sinal.
- A representação é o complemento para dois.
- Para trabalhar sem sinal, utiliza-se `unsigned`.

Caracteres: `char`

- Na linguagem C, os caracteres são vistos como números.
- O valor numérico é o código ASCII: um valor com 8 bits.
- Uma notação especial permite abstrair este ponto:

```
char c,d,e;
c = 'A';
d = 65;
/* c e d tem mesmo valor */
c = c + 3;
d = 'D';
/* c e d tem mesmo valor */
scanf("%c",&e);
printf("O codigo do caracer e' %d\n",e);
```

- É possível representar caracteres especiais:

```
\n, \t, \\, \", \', \?, \132, \141, \0, \x33
```

Constantes inteiras

As constantes inteiras podem ser introduzidas em diversos formatos.

```
55          \* decimal *\
0x55        \* hexadecimal *\
013442      \* octal
```

A utilização de sufixos permite adicionar informação de tipo aos literais.

Formato	Tipos possíveis
Dec sem sufixo	`int`, `long int`, `unsigned long int`
Hex ou oct sem sufixo	`int`, `unsigned int`, `long int`, `unsigned long int`
Sufixo `U`	`unsigned int`, `unsigned long int`
Sufixo `L`	`long int`, `unsigned long int`

Vírgula Flutuante: `float` e `double`

Números não inteiros são representados de várias formas:

```
0.654    5.0    0.0000002    .6    1.
```

As variáveis que contêm estes valores podem ser de dois tipos:

- `float` Precisão capturável geralmente em 32 bits.
- `double` Precisão dupla (o dobro do `float`) geralmente em 64 bits.

A representação utilizada depende da máquina.

```
/* area: calcula a area de um circulo
 * argumentos: r: raio do circulo
 * retorna: a area calculada */
#define PI 3.1415926
double area(double r) {
    return (PI * r * r);
}
```

Gamas de Variação de Tipos Aritméticos

char	-128 a 127
short	-32,768 a 32,767
short int	-32,768 a 32,767
int	-2,147,483,648 a 2,147,483,647
long	-2,147,483,648 a 2,147,483,647
unsigned char	0 a 255
unsigned short	0 a 65,535
unsigned int	0 a 4,294,967,295
unsigned long	0 a 4,294,967,295
float (7 dig.)	$+/-3.4E10^{38}$ a $+/-3.4E10^{38}$
double (15 dig.)	$+/-1.7E10^{308}$ a $+/-1.7E10^{308}$

Conversões de tipos (1)

- Há vária formas de conversão entre diversos tipos numéricos.
- Explicitamente usando um *cast*:

```
int i = 10;
float f = (float) i;
double d = (double) i;
```

- Por vezes o compilador faz a conversão automaticamente
  - Em atribuições o valor toma o tipo do destino.
  - Em expressões, os parâmetros são tornados compatíveis.
  - Por vezes, os argumentos de funções são também convertidos.
- **As conversões implícitas são perigosas quando podem perder informação!**
  Isto só acontece nas atribuições.

Conversões de tipos (2)

- As conversões automáticas dentro de expressões ocorrem em duas situações:
  - Os inteiros mais pequenos que `int` são **sempre** convertidos para `int`.
  - Quando há expressões mistas, as conversões unificam os tipos dos parâmetros de cada operador.
- No segundo caso, as conversões seguem uma hierarquia de tipos:

  ```
  int < unsigned int < long int <
  < unsigned long int < float <
  < double < long double
  ```

- Quando o compilador encontra uma expressão como

  ```
  2.5 * 3
  ```

  O resultado será do tipo `float`, mais acima na hierarquia.

Tipos Enumerados

- Permitem definir variáveis que podem tomar um conjunto pequeno de valores constantes, com nomes legíveis.
- Permitem detectar erros na manipulação dos valores dessas variáveis.

```
enum { vermelho, verde, azul} cor;
enum { claro, escuro } tom;
cor = vermelho; /* OK */
cor = escuro; /* NAO OK */
```

- O compilador geralmente atribui um valor inteiro a cada nome.
- O primeiro nome toma o valor 0.
- Os seguintes tomam os inteiros sucessores.
- É possível fornecer esses valores explicitamente.

```
enum { vermelho=5 , verde=10 , azul=15} cor;
```

Arrays (Antevisão)

- Um *array* é uma colecção de variáveis do mesmo tipo.
- Essas variáveis estão em posições contíguas de memória.
- Este código:

```
int a[2]={0,0};
a[0] = 4;
a[1] = a[0]+2;
```

  É equivalente a:

```
int a0=0,a1=0;
a0 = 4;
a1 = a0+2;
```

- O que acontece se indexamos uma posição que não existe?

Strings (Antevisão)

- Um *string* é um *array* de caracteres.
- Por serem tão comuns, são suportados por uma sintaxe especial.
- Este código:

```
char a[4];
a[0] = 'a';
a[1] = 'b';
a[2] = 'c';
a[3] = '\0';
```

É equivalente a:

```
char a[4]="abc";
char a[4] = { 'a', 'b', 'c', '\0'};
```

- Os *strings* são sempre terminados pelo caracter `'\0'`.

Controlo da Execução

Condicionais

Ciclos

Condicionais (1)

- Um programa deve poder tomar decisões com base no seu estado actual:

```
int a = 10;
if (a == 20)
{
    printf("E' igual a 20\n");
}
else
{
   printf("Nao e' igual a 20\n");
}
```

- Os operadores relacionais retornam 1 ou 0.
- Na condição do `if` o "false" é 0.
- Tudo o resto é "true".
- Podemos fazer `if (j)` ou `if (f())` ...

Condicionais (2)

- A clausula `else` é opcional.
- Para decisões complexas, podemos encadear `else if`:

```
int a = 0;
if (a == 10) {
    printf("a igual a 10\n");
} else if (a < 10) {
    if (a > 5) {
       printf("a entre 5 e 10\n");
    }
} else {
    printf("a maior que 10\n");
}
```

- Um else corresponde **sempre** ao último `if`.
- A menos que se usem chavetas.

Conditionais: Erros Comuns

- Uma atribuição pode aparecer no lugar da condição!

```
int a = 10;
if (a = 20) /* sempre verdade! */
{
    printf("E' igual a 20\n");
}
```

  Solução: **nunca** usar atribuições no lugar de condições.

- Sem chavetas, apenas uma instrução é abrangida!

```
int a = 0, b;
if (a > 0) /* sem ; !!! */
    printf("Posso dividir: e' maior que 0\n");
    b = 10/a; /* BRONCA */
```

  Solução: usar **sempre** chavetas.

- Não há `;` entre o `if` e a instrução da condição verdadeira.

Condicionais em Forma Compacta

O C disponibiliza um operador que permite escrever condicionais em forma compacta.

O operador "`?  :`" recebe três argumentos semelhantes à construção `if`:

- Uma condição a avaliar.
- Um valor para o caso verdadeiro.
- Um valor para o caso falso.

```
int i = 10;
int j;
j = (i == 10) ? 20 : 5;
/* "(i == 10) ? 20 : 5" significa:
 * "If (i == 10) then 20 else 5." */
```

Construção `switch` (1)

- Lida com muitos casos de forma mais legível e eficiente:

```
switch (a) {
case '1':
  return 1;
case '2':
case '3':
  return 2;
default:
  return (-1);
}
```

- A expressão (`a`) tem de ser um tipo numérico inteiro.
- Os casos têm de corresponder a valores constantes.
- A execução salta para o primeiro caso com concordância.
- O `default` permite concordância com qualquer valor.

Construção `switch` (2)

- **Importante:** Não havendo saída explícita, a execução prossegue através dos casos subsequentes.
- A instrução `break` permite terminar a execução do `switch` e prosseguir na instrução que se lhe segue.

```
switch (a) {
case '1':
  printf("E um 1\n");
  break;
case '2':
  printf("E um 2\n");
  break;
default:
  printf("Nao sei o que e\n");
  break;
}
printf("Depois do break vem para aqui!\n");
```

Ciclos `while`

Repetir um conjunto de instruções, até que uma determinada condição se verifique

```
/* while (<teste nao zero>) {
       <executa corpo>
   } */
int a = 10;
while (a > 0)
{
    printf("a = %d\n", a);
    a--;
} /* execucao volta ao teste */
```

Podemos nunca executar o corpo do ciclo!

Podemos não saber à partida o número de iterações!

Ciclos `do ... while`

Repetir um conjunto de instruções, até que uma determinada condição se verifique

```
/* do {
       <executa corpo>
   } while (<teste nao zero>) */
int a = 10;
do
{
    printf("a = %d\n", a);
    a--; /* equivalente a a=a-1 */
} while (a > 0); /* execucao volta ao do */
```

Executamos sempre pelo menos uma vez o corpo do ciclo.

Exemplo (1)

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   int c;
   int num_espacos = 0;
   printf("Introduza uma frase: \n");
   c = getchar();
   while (c != '\n') {
      if (c == ' ') {
         num_espacos++;
      }
      c = getchar();
   }
   printf("Numero de espacos: %d\n",num_espacos);
   exit(0);
}
```

Exemplo (2)

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   int c;
   int num_espacos = 0;
   printf("Introduza uma frase: \n");
   do {
      c = getchar();
      if (c == ' ') {
         num_espacos++;
      }
   } while (c != '\n');
   printf("Numero de espacos: %d\n",num_espacos);
   exit(0);
}
```