ANSI C para quem tem pressa

António Manuel de Sousa Barros (AMB)
amb@isep.ipp.pt

24 de Fevereiro de 2003

Conteúdo

Este pequeno apontamento serve para apresentar rapidamente as funcionalidades básicas da linguagem C. No entanto, muito fica por cobrir, devendo o interessado procurar outras fontes para aprender esta linguagem poderosa. O meu livro preferido é 'The C Programming Language' de Brian Kernighan e Dennis Ritchie (edição da Prentice Hall). O manual do UNIX (comando 'man') é um companheiro inseparável durante a escrita de um programa, pelo que é outra fonte que vivamente recomendo.

Capítulo 1

Noções básicas de sistemas informáticos.

Durante a segunda metade do século XX, o papel dos sistemas informáticos na sociedade teve um crescimento impressionante. Os computadores têm sido fundamentais no desenvolvimento científico e tecnológico (e.g. exploração espacial, projecto do Genoma Humano), mas tem também influenciado a sociedade em geral (e.g. telecomunicações, operações bancárias, Internet).

Um computador é uma máquina electrónica com memória, capaz de realizar vários cálculos aritméticos e lógicos por segundo. Porém, para cada aplicação que se pretenda que o computador realize, é necessário instruí-lo sobre as tarefas que deverá realizar com os dados. A esta tarefa chama-se **Programação**.

Um **programa** é uma sequência de instruções que o computador deve realizar de forma a processar os dados e obter os resultados correctamente.

1.1 A história da programação de computadores.

1.1.1 A linguagem máquina.

O primeiro computador electrónico foi o ENIAC. Construído nos EUA durante a II Guerra Mundial e concluído em 1946, tinha por finalidade calcular rapidamente as tabelas de tiro para peças de artilharia e bombardeamentos. Era uma máquina enorme construída por 18000 válvulas e 1500 relés, ocupando várias salas. A sua fiabilidade era reduzida, dado que as válvulas fundiam facilmente, devida à potência irradiada por estes componentes. Os programas eram inseridos através de um leitor de cartões perfurados (da IBM), sendo escritos directamente em código-máquina (utilizando unicamente os dois únicos símbolos binários **zero** e **um**). Adicionalmente, o ENIAC não era capaz de armazenar os programas em memória, pelo que era necessário configurar um conjunto de interruptores e ligações por cabos (trabalho realizado por seis técnicas) de acordo com o programa a ser executado. No entanto era capaz de calcular uma trajectória de 60 segundos em apenas 30 segundos, em oposição às 20 horas tomadas por um técnico-matemático com uma calculadora de secretária.

A linguagem máquina, sendo a única que os computadores entendem, oferece grandes dificuldades aos programadores:

- longo tempo e grandes custos de aprendizagem;

- máquinas diferentes entendem linguagens-máquina diferentes;
- muito tempo despendido para escrever um programa;
- depuramento e correcção de programas simplesmente infernal.

Desta forma, é fácil imaginar que a quantidade de programadores em todo o mundo era extremamente reduzida, o que implicava um reduzido número de aplicação de computadores.

1.1.2 A linguagem assembly.

Nos finais da década de 1950 e durante a década de 1960, ocorreu uma procura maciça de poder de cálculo devido aos seguinte factores:

- exploração espacial (fomentado pela Guerra Fria);
- desenvolvimento de armamento nuclear (também fomentado pela Guerra Fria);
- gestão das grandes corporações (IBM, General Motors, etc.).

A aplicação em máquinas de cálculo do recém inventado **transístor** permitiu a construção de computadores mais fiáveis (devido à muito menor irradiação de calor), compactas e **rápidas**.

Porém, o modelo de programação em linguagem-máquina, em que os progamas eram picotados em cartões perfurados não podia dar resposta às crescentes solicitações do mercado. Os engenheiros pensaram então numa linguagem de programação rudimentar, em que o nível de programação estaria muito próximo da linguagem-máquina, mas cuja escrita e leitura por parte de humanos fosse razoavelmente simples. Surgiu então a linguagem assembly. O programa seria então escrito pelos programadores em linguagem **assembly**, sendo por fim traduzido para linguagem máquina por um **assembler**, que recorreria a uma tabela de tradução.

1.1.3 As linguagens de alto nível.

O assembly permitiu aumentar o número de programadores e o desenvolvimento de aplicações para computadores. No entanto, não conseguia resolver satisfatoriamente todos os problemas de construção de aplicações:

- embora facilitado, o depuramento de programas ainda era complicado;
- era difícil "pegar" num programa escrito por outra pessoa;
- o programador tinha que ter um bom conhecimento da arquitectura do computador.

Em 1957 surgiu a primeira linguagem de alto-nível: o Fortran. A partir daí, surgiram outras linguagens de alto-nível (e.g. ALGOL, BASIC, COBOL, FORTRAN, C), que ofereciam aos programadores a possibilidade de escreverem programas em linguagens próximas do inglês. Desta forma:

- a aprendizagem de uma linguagem seria mais rápida;
- o depuramento de programas era muito mais fácil;

- o desenvolvimento de aplicações poderia ser feito em equipas;
- as aplicações poderiam ser mais complexas.

Dado que o nível de programação se situa muito acima da linguagem-máquina, a aplicação de simples tradutores como um assembler já não seria possível. Surgiram então os compiladores, aplicações capazes de procurar erros de sintaxe e concepção nos textos dos programas e transformar as instruções em linguagem quase-natural para linguagem máquina. Uma única instrução como

```
PRINT "Olá, tudo bem?"
```

em BASIC (ordenando o computador para imprimir uma frase no ecrã), seria traduzida em algumas dezenas de instruções em código máquina.

1.2 Os sistemas operativos.

Um computador é constituído por um conjunto de dispositivos electrónicos (placa gráfica, impressora, teclado, disco, etc.). De computador para computador, é normal encontrarmos placas gráficas diferentes, discos de diferentes capacidades ou fabricantes, pelo que seria muito difícil a um programador construir aplicações (por exemplo, um processador de texto) para uma máquina específica, e ter que rescrever o programa só para que ele funcionasse num outro computador com uma placa gráfica diferente. Surgiu então a necessidade de um sistema operativo, que oferecesse um serviço de acesso uniforme aos dispositivos físicos para as aplicações.

Com a existência de um sistema operativo, uma aplicação como um editor de texto só precisa de saber ordenar ao sistema operativo que quer que um dado documento seja impresso: caberá ao sistema operativo a função de enviar para a impressora todos os dados necessários para a impressão (ver figura 1.1).

Figura 1.1: Camadas Aplicação / Sistema Operativo / Hardware.

O sistema operativo gere todas as actividades de um computador:

- fornece o acesso aos dispositivos;
- realiza o escalonamento das aplicações;
- assegura o correcto funcionamento das aplicações.

O sistema operativo depende da arquitectura do computador, existindo para cada arquitectura pelo menos um sistema operativo. Na tabela 1.1 são apresentados alguns sistemas operativos para várias arquitecturas:

Tabela 1.1: Arquitecturas e Sistemas Operativos.

Arquitecturas	**Sistemas operativos**
Intel x86	Windows UNIX (Linux, Free BSD, Solaris, etc) BeOS
PowerPC (Mac)	MacOS ver 7, 8, 9 MacOS X (UNIX)
HP	HP-UX (UNIX)
SGI (Silicon Graphics)	IRIX (UNIX)
SUN	Solaris (UNIX)

1.3 Compiladores.

Tal como referido anteriormente, os compiladores transformam os textos dos programas (em linguagem de alto-nível) em código-máquina. Como o código-máquina é produzido para utilizar os recursos oferecidos pelo sistema operativo, facilmente se depreende que para cada sistema operativo deverá existir um compilador.

Assumindo que se escreveu um programa em linguagem C. Se esse programa for compilado para Windows, o código-máquina gerado não será inteligível para o sistema operativo Linux, nem para o MacOS. Para o programa ser executável em vários sistemas operativos, é pois então necessário compilá-lo para cada um dos sistemas. Num futuro capítulo, será explicado o processo de compilação de um programa.

Capítulo 2

Estrutura de um programa em C.

2.1 Directivas de pré-compilação.

Nesta primeira fase, é necessário indicar ao compilador quais as bibliotecas de funções que devem ser incluídas aquando da compilação, e definir os nomes e valores de constantes que serão utilizadas durante a escrita do programa. Um exemplo possível para este bloco seria o seguinte:

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

#define PI 3.14159265
```

As duas primeiras linhas indicam ao compilador que o programa utiliza funções que se encontram definidas nos ficheiros stdio.h (standard input/output) e math.h. Funções tipicamente utilizadas são:

- stdio.h - printf() e scanf() para saída e entrada de dados;
- math.h - sin(), sqrt() e outras operações matemáticas.

A terceira linha de código atribui ao nome PI o valor 3.14159265. Todas as referências PI ao longo do texto do programa serão substituídas por 3.14159265 antes da compilação.

2.2 Declaração de variáveis globais.

Uma variável que seja acessível pelo mesmo nome em qualquer função do programa é chamada global. A declaração de uma variável global é feita a seguir ao bloco de directivas de pré-compilação.

Uma variável é declarada indicando o tipo de valores que poderá conter seguido do identificador (nome) da variável. Atendendo ao seguinte exemplo de declaração de variáveis:

```
int contador = 0;

float cateto1, cateto2, hipotenusa;
```

A primeira linha declara a variável contador para conter números inteiros e inicializa-a com o valor 0.

Na segunda linha, são declaradas as variáveis cateto1, cateto2 e hipotenusa, para conter números com parte fraccionária.

2.3 As funções.

No último bloco escreve-se o código das funções (aquilo que realmente faz mexer o programa). As funções são declaradas através do tipo do valor retornado, o identificador da função e a lista dos argumentos que a função recebe.

No seguinte exemplo, teremos uma função que calcula o comprimento da hipotenusa de um triângulo rectângulo, recebendo como argumentos os comprimentos dos catetos:

```
float calculahipotenusa (float cat1, float cat2)
```

A função chamada calculahipotenusa() devolve um resultado do tipo float (à esquerda do nome da função), aceitando como argumentos dois valores do tipo float, que serão identificados dentro da função pelos nomes cat1 e cat2.

A sequência de instruções que a função deve realizar é encapsulada entre chavetas. Em C, as chavetas são utilizadas para agrupar sequências de instruções. Dentro do grupo de instruções, existe um primeiro bloco em que são declaradas as variáveis locais à função e, finalmente, a sequência de instruções. Continuando o exemplo, o código da função poderia ser o seguinte:

```
{
  float hip; /* Declaração da variável local hip */

  cat1 = cat1 * cat1; /* Eleva os catetos ao quadrado */

  cat2 = cat2 * cat2;

  hip = cat1 + cat2; /* Soma o quadrado dos catetos */

  hip = sqtr(hip); /* e determina a raiz quadrada */

  return(hip); /* Retorna o valor da variavel hip */
}
```

2.4 A função main().

O programa, depois de compilado, começa a executar as instruções contidas numa função especial: a função main(). Esta função deve conter o algoritmo principal, e chamar as funções necessárias à execução do algoritmo.

Tendo em conta os exemplos anteriores, poderíamos escrever a seguinte função main():

```
main()
{
  float c1, c2, hipotn; /* Declara as variáveis locais */

  /* Escreve no ecrã a frase entre aspas */
  printf("Cálculo de hipotenusa...\n");

  /* O utilizador introduz os comprimentos dos catetos */
  printf("Introduza o comprimento de um dos catetos: ");
  scanf("%f", &c1);
  printf("Introduza o comprimento do outro cateto: ");
  scanf("%f", &c2);

  /* A hipotenusa é calculada, chamando a */
  /* função calculahipotenusa() */
  hipotn = calculahipotenusa(c1, c2);

  printf("O comprimento da hipotenusa: %f\n", hipotn);
}
```

2.5 Exercícios.

1. Dados os valores de duas resistências, pretende-se obter os valores das resistências equivalentes quando associadas em série e em paralelo. Projecte o algoritmo de um programa capaz de realizar esta tarefa.

Capítulo 3

Estruturas de decisão e ciclos.

3.1 Estruturas de decisão.

As estruturas de decisão são úteis sempre que se tem que optar por um conjunto de operações, dependendo de uma condição. A execução do programa segue sempre "em frente" (não confundir com os ciclos).

3.1.1 if - ... - else - ...

Quantas vezes não somos confrontados com situações em que temos que tomar uma decisão em função de um conjunto de condições? Um caso que se aplica todos os dias (excepto quando estamos de cama, doentes) é quando temos que atravessar uma estrada. Mentalmente, utilizamos o seguinte algoritmo:

```
SE (não passam carros nos próximos instantes)
ENTÃO Atravesso a rua
SENÃO Sigo até aos semáforos.
```

A linguagem C, como foi criada por americanos, tem implementada esta estrutura de decisão, mas em inglês... Para facilitar a vida ao programador, o ENTÃO é omitido (quando se avalia algo, é de esperar que alguma acção possa ser despoletada). O SENÃO e acções subsequentes são opcionais (nem sempre temos que fazer qualquer coisa, caso o resultado da avaliação não nos interesse). O resultado é este:

```
if ( condição lógica )
{
  acções a tomar
}
else
{
  acções alternativas a tomar
}
```

No seguinte exemplo, caso o saldo de um cartão de telemóvel seja inferior ao valor de uma chamada, deverá ser emitida uma mensagem a informar que a chamada não se pode efectuar. Obviamente, se o saldo for maior, nada deverá ocorrer.

```
if (saldo < compra)
{
  printf("O saldo é insuficiente para realizar a compra.\n");
  printf("Por favor, efectue um carregamento do seu cartão.\n");
}
```

No seguinte caso, é avaliada a relação maior-igual-menor entre dois números:

```
if (a > b)
  printf("%d é maior que %d.\n", a, b);
else
  if (a < b)
    printf("%d é menor que %d.\n", a, b);
  else
    printf("São iguais.\n");
```

3.1.2 switch... case...

Quando o leque de acções a realizar depende do valor de uma variável, podemos utilizar a instrução switch(). Atendendo ao seguinte exemplo:

```
printf("Qual a operação que pretende efectuar? ");
operador = getch();
switch (operador) {
case '+':
  resultado = a + b;
  break;
case '-':
  resultado = a - b;
  break;
case '*':
  resultado = a * b;
  break;
case '/':
  resultado = a / b;
  break;
default:
  printf("Operação não permitida!\n");
  break;
}
```

Mediante o carácter teclado pelo utilizador, o programa irá realizar a operação pretendida. Para cada caso previsto, é indicada a sequência de instruções a realizar.

A palavra break termina a sequência de instruções para cada opção.

A opção default é realizada sempre que nenhuma das condições anteriores não tenha sido satisfeita.

3.2 Ciclos.

As ciclos decorrem da necessidade de repetir a mesma sequência de acções até que determinada condição seja alcançada.

3.2.1 Ciclos for().

Os ciclos for() são utilizados quando as operações de inicialização e incremento são simples, normalmente utilizando uma variável como contador do ciclo.

O método de declaração de um ciclo for() é o seguinte:

```
for(inicialização; condição; operação no fim de cada ciclo)
```

A inicialização é uma operação realizada antes de se iniciar a actividade do ciclo. O ciclo só é executado enquanto o valor da condição for 'verdade'. No final de cada ciclo, é executada a instrução que se encontra no terceiro argumento.

O seguinte exemplo:

```
soma = 0;
for(contador = 1; contador < 5; contador++)
{
  printf("Contador: %d\n", contador);
  soma += contador;
}
printf("Somatório: %d\n", soma);
```

produziria a saída:

```
Contador: 1
Contador: 2
Contador: 3
Contador: 4
Somatório: 10
```

3.2.2 Ciclos while().

Os ciclos while() são realizados enquanto uma determinada condição é válida. Se a condição à partida for inválida, o ciclo não chega sequer a ser executado (o número mínimo de execuções do ciclo é ZERO).

```
total = 0;

while(total < 50000)
{
  scanf("%d", &custo);
  total += custo;
}
```

O exemplo acima, permite introduzir custos enquanto o total de custos é menor do que 50000. Note que este exemplo seria mais difícil de implementar com um ciclo for(), já que a condição de incremento não é tão simples!

3.2.3 Ciclos do-...-while().

Os ciclos do ... while() asseguram que o conjunto de instruções dentro do ciclo é executado pelo menos UMA vez, isto porque a condição de execução do ciclo é avaliada apenas no fim.

```
do {
  printf("Insira um número entre 0 e 20: ");
  scanf("%d", &numero);
} while ( (numero < 0) && (numero > 20) );
```

3.3 Exercícios.

1. Dados dois valores, pretende-se saber se o primeiro é maior, igual ou menor do que o segundo. Construa uma hipótese possível de resolução deste problema.

2. Escreva um algoritmo que permita calcular o factorial de um número inteiro (n!). Tenha em atenção de que não existem factoriais de números negativos! Por definição, 0! é igual a 1 (um).

3. Num jogo, é introduzido um número inteiro que deverá ser adivinhado pelo jogador, através de palpites. Para cada palpite, deverá ser indicado ao jogador se o número a adivinhar é maior ou menor do que o palpite. Quando o jogador finalmente adivinha o valor, deverá ser indicado o número de palpites realizados.

Capítulo 4

Funções.

Uma função é um conjunto de instruções, capaz de realizar uma determinada tarefa.

A grande vantagem de escrever funções num programa em C, é podermos escrever uma única vez o conjunto de instruções (função), e chamá-la várias vezes durante a execução do programa.

As funções podem receber valores e realizar processamento sobre esses valores, retornando o resultado desse processamento.

4.1 As partes de uma função.

4.1.1 Cabeçalho ou Protótipo.

Uma função é declarada no seu cabeçalho. O cabeçalho de uma função contém os seguintes elementos:

- o tipo de valor que será retornado pela função (tipo do resultado);
- o nome da função;
- uma lista de parâmetros entre parêntesis, que são recebidos pela função.

4.1.2 Corpo de uma função.

Após o cabeçalho da função, deverá vir o corpo da função. O corpo de uma função é o conjunto de instruções que a função deverá realizar, delimitado por chavetas.

4.2 Funções sem parâmetros.

As funções sem parâmetros têm por função realizar uma tarefa, independentemente de quaisquer dados.

A seguinte função escreve um menu de selecção no ecrã. Imprimir estas quatro linhas de texto não requer a recepção de quaisquer valores.

```
void mensagem ()
{
  printf("\n\nEscolha a opção:\n");
```

```
  printf("1 - Inserir\n");
  printf("2 - Remover\n");
  printf("3 - Sair\n\n");
}
```

4.3 Funções com parâmetros.

Grande parte das funções processam dados que lhes são passados como parâmetros. Uma função que calcule o comprimento de uma hipotenusa, terá que necessariamente receber os valores dos comprimentos dos catetos.

Há duas formas de passar parâmetros:

- por valor;
- por referência.

4.3.1 Passagem de parâmetros por valor.

Neste caso, a função recebe cópias dos valores que foram fornecidos aquando da chamada da função. Quaisquer alterações dos valores das cópias não afectarão os valores originais. Suponha o seguinte exemplo:

```
#include <math.h>

(...)

float hipotenusa (float cateto1, float cateto2)
{
  float hipotenusa;
  cateto1 = cateto1 * cateto1;
  cateto2 = cateto2 * cateto2;
  hipotenusa = cateto1 + cateto2;
  hipotenusa = sqrt(hipotenusa);
  return (hipotenusa);
}

(...)

main ()
{
  float cat1 = 3.00;
  float cat2 = 4.00;
  float hip;

  hip = hipotenusa(cat1, cat2);
  (...)
}
```

Neste exemplo, as variaveis cateto1 e cateto2 da função hipotenusa() recebem cópias dos valores das variáveis cat1 e cat2 e da função main(). As operações realizadas nas

variaveis cateto1 e cateto2 (dentro da função hipotenusa) não afectam os valores das variáveis cat1 e cat2.

4.3.2 Passagem de parâmetros por referência.

Neste caso, a função recebe a localização em memória das variáveis na chamada. Conhecendo a localização das variáveis é possível altera os seus valores. Suponha o seguinte exemplo:

```
float troca (float *variavel1, float *variavel2)
{
  float auxiliar;

  auxiliar = *variavel1;
  *variavel1 = *variavel2;
  *variavel2 = auxiliar;
}

(...)

main ()
{
  float a = 3.00;
  float b = 4.00;

  troca(&a, &b);

  (...)
}
```

Repare que na chamada da função troca() na função main(), são passados os endereços das variáveis a e b, na forma &a e &b. No cabeçalho da função troca(), as variaveis variavel1 e variavel2 são apontadores (identificados pelo asterisco que os prefixa na declaração) para valores do tipo float. Um apontador é uma variável que guarda um endereço de memória. Nas linhas de instrução, o asterisco (*) indica que queremos aceder ao conteúdo do endereço apontado:

- variavel1 indica o endereço;
- variavel1 indica o conteúdo no endereço.

Desta forma, a função troca() realiza as seguintes acções:

1. a variável auxiliar recebe o conteúdo do endereço apontado pela variavel1;
2. o conteúdo do endereço apontado pela variavel2 é copiado para o conteúdo do endereço apontado pela variavel1;
3. o valor da variável auxiliar é copiado para o conteúdo do endereço apontado pela variavel2.

Capítulo 5

Vectores.

5.1 Operações com vectores.

Muitas vezes temos necessidade de operar com vários elementos que representam a mesma grandeza, e que se encontram relacionados: a posição de uma partícula ao longo do tempo, os valores de resistências eléctricas num circuito...

Tomemos por um circuito eléctrico que contém 10 resistências. É necessário escrever um programa que efectue uma série de cálculos utilizando os valores das resistências. Os valores das resistências deverão ser inseridos pelo utilizador.

Logo à partida, confrontamo-nos com a necessidade de declarar 10 variáveis, uma para cada resistência. A tarefa ainda se complica mais quando tivermos que repetir 10 vezes as instruções de inserção dos valores das resistências.

A utilização de vectores permite facilitar (e bastante) a escrita de programas que utilizem grandes quantidades de dados do mesmo tipo.

5.1.1 Declarar um vector.

A declaração de um vector é semelhante à de uma variável. No entanto, teremos que indicar (entre parêntesis rectos) a dimensão do vector. Entenda-se por dimensão do vector, o número de valores que o vector suporta.

```
main()
{
  float resistencias[10];

  (...)
}
```

No exemplo anterior, é definido um vector chamado resistencia, com capacidade para dez valores do tipo float.

5.1.2 Aceder aos elementos do vector.

A forma de aceder aos elementos de um vector é semelhante à forma de aceder ao conteúdo de uma variável. No entanto, será necessário indicar qual o índice do elemento ao qual se pretende aceder. Considere um vector como uma rua, e o índice como o número de uma casa nessa rua.

O primeiro elemento do vector encontra-se no índice zero. Desta forma, o último elemento encontra-se no índice dimensão-1, em que dimensão é dimensão do vector.

O seguinte exemplo demonstra alguns exemplos de utilização dos vectores.

```
main()
{
  float resist[10];

  /* Insere 6 valores de resistências para as */
  /* posições de 0 a 5. */
  for(i=0; i<6; i++)
  {
    printf("Insira o valor da resistência R%d: ", i);
    scanf("%f", &resist[i]);
  }

  resist[7] = (resist[0] + resist[1]) * 0.63;

  (...)

}
```

Capítulo 6

Apontadores.

6.1 Aceder aos endereços de memória onde se encontram as variáveis.

Em C, é possível determinar o endereço de memória em que uma variável armazena os seus valores, através dos apontadores. Consideremos, por exemplo, a seguinte definição de uma váriavel:

```
int contador = 4;
```

O identificador da variável - a palavra 'contador' - aponta implicitamente para uma posição em memória, onde se encontra armazenado o valor 4.

Endereço na memória	Conteúdo	Identificação da variável
0x1F22E	4	contador

Quando utilizamos o identificador da variável, acedemos ao seu conteúdo. Utilizando a seguinte expressão,

```
printf("O valor da variavel é:%d.\n", contador);
```

obteríamos o resultado

```
O valor da variavel é: 4.
```

Caso seja necessário aceder ao endereço de memória onde se guarda o valor da variável, utiliza-se o operador &:

```
printf("O endereço da variavel é: %p.\n", &contador);
```

que resultaria na seguinte saída:

```
O endereço da variavel é: 1F22E.
```

6.2 Variáveis apontadores.

Por vezes, é necessário manipular directamente os endereços de memória.Uma variável pode ser definida para conter endereços de memória da seguinte forma:

```
int *apontador;
```

O operador * indica que a variavel 'apontador' é um apontador para uma posição de memória onde se pretende armazenar - neste caso - um valor inteiro. Supondo o seguinte excerto de código-fonte:

```
int valor = 7;
int *apontador;

/* A variavel 'apontador' passa a apontar para
/* a posição de memória da variável 'valor' */
apontador = &valor;

printf("Endereço de 'valor': %p,\n", apontador);
printf("e actualmente contém o inteiro %d.\n", *apontador);
```

uma possível saída seria a seguinte:

```
Endereço de 'valor': 10FF2,
e actualmente contém o inteiro 7.
```

Note que a própria variável apontador encontra-se num dado endereço de memória. A tabela 6.1 resume os operadores de acesso a posições de memória:

Tabela 6.1: Significados na notação de apontadores.

Expressão	Significado
&apontador	Indica o endereço onde o conteúdo da variável apontador se encontra armazenado em memória.
apontador	Indica um determinado endereço em memória onde supostamente existe algum valor.
*apontador	Indica o valor existente na posição de memória apontada pela variavel apontador.

No seguinte caso, pretende-se que uma função altere / actualize os valores das variáveis que são passadas como argumentos. Suponha que num programa se pretende trocar os valores de duas variáveis, chamando a função troca.

```
troca(variavel1, variavel2);
```

Esta chamada da função forneceria apenas os valores das variáveis, sendo impossível à função actualizar os conteúdos das variaveis, dado que a função desconhece os endereços onde os valores se encontram. A função tem que conhecer os endereços onde as variáveis armazenam os seus valores. Isso é conseguido utilizando a seguinte chamada:

```
troca(&variavel1, &variavel2);
```

Desta forma, a função recebe não os valores das variáveis, mas sim os seus endereços. O código-fonte da função poderia ser o seguinte:

```
troca(int *v1, int *v2)
{
  int aux;
```

```
  /* aux passa a ter o valor apontado por v1 */
  aux = *v1;

  /* aux passa a ter o valor apontado por v1 */
  *v1 = *v2;

  /* aux passa a ter o valor apontado por v1 */
  *v2 = aux;
}
```

6.3 Exercícios.

1. Escreva um programa que defina uma variável para conter valores inteiros, e um apontador para inteiros. Ao apontador deverá ser atribuído o endereço da variável. Para a variável imprima as sua posição em memória e o seu valor. Para o apontador, imprima a sua posição em memória, o seu valor e o valor na posição de memória apontada.

2. Escreva uma função que retorne um apontador para a 1ª ocorrência de um carácter numa string ou o valor zero se o carácter não existir na string. A função deve ter como argumentos a string e o carácter a procurar.

Capítulo 7

Apontadores e memória dinâmica.

7.1 A necessidade de memória dinâmica.

Em várias aplicações, existe a necessidade de manipular estruturas de dados cuja dimensão não é inicialmente conhecida.

Suponha o caso em que se pretende conceber um programa que execute várias estatísticas sobre as idades dos alunos de uma turma. O número de alunos de uma turma é altamente variável, o que levanta dificuldades à utilização de vectores. Como se deveria dimensionar o vector?

- Uma dimensão razoável poderia ser insuficiente para certas turmas.
- Um vector muito grande estaria a desperdiçar memória. Este factor é crítico em aplicações que operam grandes quantidades de dados relativamente às capacidades da máquina.

A utilização dinâmica de memória permite adaptar as necessidades de memória ao problema concreto.

7.2 Manuseamento dinâmico de memória em C.

O C não oferece de raíz o suporte para operações de memória dinâmica, sendo necessário incluir a biblioteca standard do C: stdlib.h.

Nesta biblioteca, existem duas funções que oferecem as funcionalidades básicas para utilização dinâmica de memória: malloc() e free().

7.2.1 Reserva dinâmica de memória.

A função malloc() (memory allocation - trad. afectação de memória) reserva uma porção de memória, retornando um apontador genérico (tipo void *) para o inicio da porção reservada, ou o valor NULL no caso da reserva ser impossível. A sua utilização é representada no exemplo seguinte:

```
float *v;
int n;

printf("Quantos valores? ");
scanf("%d", n);

v = (float *) malloc(n * sizeof(float) );
```

Neste exemplo, é reservada uma porção de memória capaz de guardar n números reais (float), ficando o apontador v a apontar para o endereço inicial dessa porção de memória. O cast da função malloc() - (float *) - assegura que o apontador retornado é para o tipo especificado na declaração do apontador. Certos compiladores requerem obrigatoriamente o cast.

Conselho: não altere o valor do apontador que recebeu o retorno da função malloc(). Desta forma poderá sempre saber onde começa o bloco de memória dinâmica reservado. Utilize apontadores auxiliares para realizar operações (leitura, escrita) dentro do bloco de memória.

7.2.2 Libertar memória dinâmica.

Outra das vantagens da utilização dinâmica de memória é a possibilidade de libertar memória à medida que deixa de ser precisa. A memória é libertada utilizando a função free(). Supondo o exemplo da secção anterior, a porção de memória atribuída seria libertada da seguinte forma:

```
free(v);
```

7.2.3 Ajuste de memória dinâmica.

É possível alterar o tamanho do bloco de memória reservado, utilizando a função realloc(). Esta função salvaguarda os valores anteriormente em memória, até ao limite do novo tamanho (especialmente importante quando se reduz o tamanho do bloco de memória). O seguinte exemplo ilustra a forma de utilização desta função.

```
int *a;

a = (int *) malloc( 10 * sizeof(int) );

(...)

a = (int *) realloc( a, 23 * sizeof(int) );

(...)

free(a);
```

A chamada da função realloc() recebe como argumentos um apontador para o bloco de memória previamente reservado com uma função malloc() de forma a saber qual a porção de memória a ser redimensionada, e o novo tamanho absoluto para o bloco de memória.

7.3 Exercícios.

1. Leia uma sequência de 10 números do teclado usando apontadores em lugar de índices. Usando a mesma técnica (apontadores) determine o maior e o menor valor. Reserve memória dinâmica em vez de declarar o vector de uma forma estática.

2. Ler uma sequência de números do teclado (sequência terminada em zero).Escreva no ecrã os números que estão acima da média. Utilize um vector dinâmico para armazenar os números.

Capítulo 8

Estruturas de dados.

Uma estrutura é uma colecção de uma ou mais variáveis, possivelmente de diferentes tipos, agrupadas sob um único nome para facilitar o seu manuseamento. As estruturas permitem organizar dados complicados, porque permitem tratar um grupo de variáveis relacionadas como uma unidade, em vez de entidades separadas.

Como exemplos de estruturas teremos:

- um aluno é identificado pelo seu número e nome;
- um ponto é identificado pelas suas coordenadas cartesianas (x, y, z)...

Os componentes de uma estrutura (número e nome no caso do aluno) são chamados campos.

8.1 Declaração de uma estrutura em C.

A declaração de uma estrutura em C é iniciada pela palavra struct, seguida por uma lista de declarações entre chavetas. Uma possível declaração de uma estrutura para suportar informação relativa a um ponto seria:

```
struct ponto {
  float x;
  float y;
};
```

As estruturas declaradas podem ser utilizadas como tipos de campos de outras estruturas. Se pretendermos criar uma estrutura que defina um rectângulo a partir de dois vértices opostos, poderemos escrever:

```
struct rectangulo {
  struct ponto p1;
  struct ponto p2;
};
```

8.2 Variáveis do tipo estrutura.

8.2.1 Declaração de variáveis.

Após a definição da estrutura, esta pode ser utilizada para declarar váriaveis. A declaração das variáveis não é muito diferente do usual. Supondo que queremos declarar variáveis - r1 e r2 - para conter as posições de duas partículas, podemos escrever o seguinte:

```
struct ponto r1, r2 = {1.5, 3.6};
```

As variáveis são declaradas, sendo r2 inicializada com os valores:

- x = 1.5;
- y = 3.6;

8.2.2 Utilização das variáveis.

O campo de uma determinada estrutura pode ser acedido utilizando o operador '.' (ponto).

```
/* Atribui o valor 2.3 ao campo x da variável r1. */
r1.x = 2.3;

/* Atribui o valor guardado no campo y de r2 no campo y de r1. */
r1.y = r2.y;
```

Para além da atribuição individual a cada um dos campos, é ainda possível igualar dois registos rapidamente:

```
r1 = r2;
```

A expressão acima atribui a todos os campos de r1 os valores de todos os campos de r2, criando uma cópia efectiva de r2 em r1.

8.3 Vectores de estruturas.

O C permite a declaração de vectores de estruturas. A sua declaração e utilização é praticamente idêntica aos vectores de tipo "simples".

8.3.1 Declaração de vectores.

A declaração de um vector para 10 elementos da estrutura ponto definida acima, teria a seguinte forma:

```
struct ponto vp[10];
```

8.3.2 Operação de vectores.

Os campos dos diversos elementos de um vector são acedidos utilizando o operador '.' (ponto).

```
vp[0].x = 2.3;

vp[0].y = 5.2;

vp[1].x = vp[0].x * 4;

vp[2] = vp[0];
```

8.4 Apontadores para estruturas.

As estruturas, como agrupamento de variáveis que são, utilizam porções de memória que podem ser apontadas pelos apontadores. Vectores de estruturas em memória dinâmica podem igualmente ser criados, e operados com apontadores.

8.4.1 Apontadores - declaração e utilização.

Um apontador para uma estrutura é declarado da seguinte forma:

```
struct ponto *ap;
```

8.4.2 Atribuição dinâmica de memória para uma estrutura.

Utilizando o apontador definido acima, podemos reservar memória para 5 pontos escrevendo a seguinte instrução:

```
ap = (struct ponto *) malloc( 5 * sizeof(struct ponto) );
```

O apontador ap fica a apontar para o início de um bloco de memória suficiente para armazenar 5 elementos com a estrutura ponto.

A função realloc() funciona de forma semelhante. A função free() é trivial.

8.4.3 Operação dos conteúdos apontados.

Os conteúdos apontados são acedidos utilizando o operador '->' (hífen + maior que):

```
/* Atribui o valor 2.3 ao campo x no registo apontado por ap */
ap->x = 2.3;

/* Atribui o valor do campo y no registo apontado por ap *(
/* ao campo y do registo seguinte */
(ap+1)->y = ap->y;
```

Uma outra forma de aceder aos conteúdos apontados é da seguinte forma:

```
/* Equivalente a ap->x = 2.3; */
*ap.x = 2.3;

/* Equivalente a (ap+1)->y = 4.33; */
*(ap+1).y = 4.33;
```

8.5 Definir novos tipos.

Em C, é possível definir novos nomes de tipos, utilizando o typedef.

```
typedef struct ponto pontocart;

(...)

pontocart p1, p2;
```

Após a introdução typedef, o tipo struct ponto pode ser identificado simplesmente por pontocart.

8.6 Exercícios.

1. Um projéctil pode ser caracterizado pela sua posição e velocidade. Estas duas grandezas são vectoriais, sendo compostas pelas componentes x e y. A partir da sua posição inicial r0 = (r0x, r0y) e da sua velocidade inicial v0 = (v0x, v0y) é possível determinar a sua posição-velocidade para qualquer instante a partir das seguintes equações:

   - x = x0 + v0x * t;
   - y = y0 + v0y * t - 5 * t^2
   - vx = v0x;
   - vy = v0y - 10 * t;

   Pretende-se desenvolver um programa que a partir das condições iniciais de lançamento de um projéctil se determine a posição e velocidade para qualquer instante (o instante é inserido pelo utilizador). Na fase final de desenvolvimento, o programa deverá aceitar a velocidade de lançamento na forma de módulo e inclinação em relação à horizontal.

   Sugestão: crie uma estrutura para conter a informação de um vector (cartesiano). Crie depois uma outra estrutura onde os dados do projéctil (vectores posição e velocidade) recorram à anterior estrutura.

2. Declare uma estrutura capaz de armazenar o nome, o número, o ano de entrada, o curso e quantidade de cadeiras feitas. Defina um vector de 100 estruturas.

   (a) Escreva uma função para ler os valores para uma determinada estrutura do vector.

   (b) Para um determinado curso (a perguntar ao utilizador), o programa deve responder quantos alunos fizeram 10 ou menos cadeiras em mais de cinco anos.

   (c) Visualize no monitor todos os cursos inseridos sem repetições.

   (d) O programa deve dizer qual o curso com melhor aproveitamento (curso com maior média de cadeiras feitas por aluno).

Capítulo 9

Ficheiros.

Até ao momento, temos trabalhado apenas com memória. Infelizmente, sempre que um programa é terminado, os dados em memória são perdidos.

Uma forma de eliminar este inconveniente, consiste em guardar os dados em memória secundária (disco).

Os dados em disco são armazenados em ficheiros, unidades lógicas que agrupam dados relacionados entre si.

9.1 Ficheiros de texto.

Tal como o nome indica, os ficheiros de texto permitem armazenar informação na forma de texto.

9.1.1 Abrir um ficheiro de texto.

Antes de se realizar qualquer operação de leitura/escrita num ficheiro, é necessário abri-lo. Esta operação é realizada utilizando a função fopen().

```
FILE *fh; /* É declarado um apontador para ficheiro. */

(...)

/* É aberto o ficheiro agenda.txt para leitura. */
fh = fopen("agenda.txt", "r");
```

O primeiro parâmetro da função fopen() é uma string, contendo o nome de um ficheiro.

O segundo parâmetro da função fopen() é uma string, que especifica a forma como se pretende aceder ao ficheiro, conforme a tabela 9.1.

9.1.2 Escrever para um ficheiro de texto.

As funções de escrita para um ficheiro de texto são bastante semelhantes às funções de escrita para o monitor. As diferenças situam-se nos nomes das funções (prefixadas com a letra 'f') e a indicação do apontador para o ficheiro para o qual se pretende escrever.

Tabela 9.1: Modos de acesso a ficheiros

Modo de acesso	String	Comentário
Leitura	"r"	A leitura pode ser em qualquer ponto do ficheiro. A abertura do ficheiro falha se o ficheiro não existir.
Escrita	"w"	A escrita pode ser em qualquer ponto do ficheiro. Durante a abertura, se o ficheiro já existir, é apagado para criar um novo.
Escrita	"a"	A escrita apenas pode ser no final do ficheiro. Durante a abertura, caso o ficheiro não exista, é criado um novo.
Leitura / Escrita	"r+"	A leitura pode ser em qualquer ponto do ficheiro. A escrita pode ser em qualquer ponto do ficheiro.
Leitura / Escrita	"w+"	A leitura pode ser em qualquer ponto do ficheiro. A escrita pode ser em qualquer ponto do ficheiro.
Leitura / Escrita	"a+"	A leitura pode ser em qualquer ponto do ficheiro. A escrita apenas pode ser no final do ficheiro.

```
fputs("Esta frase vai aparecer no ficheiro!\n", fh);

numero = 2;
fprintf(fh, "Linha %d\n", numero);
```

9.1.3 Ler de um ficheiro de texto.

Tal como nas funções de escrita, as funções de leitura são semelhantes às funções de entrada (geralmente do teclado). O prefixo 'f' é característico nestas funções, sendo necessário identificar o ficheiro de onde se pretende ler, identificado pelo apontador de ficheiro.

```
char string[50];
int a, b;

(...)

/* Lê dois inteiro de uma linha de texto do ficheiro. */
fscanf(fh, "%d %d", &a, &b);

/* Lê para a variavel string 50 caracteres, no máximo. */
fgets(string, sizeof(string), fh);
```

9.1.4 Fechar um ficheiro de texto.

Após todas as operações de leitura / escrita terem sido realizadas, dever-se-á fechar o ficheiro, com a função fclose(). Esta operação assegura que todos os dados ficam correctamente armazenados no ficheiro.

```
fclose(fh);
```

9.2 Ficheiros binários.

Os ficheiros binários armazenam os dados tal e qual como eles se encontram na memória. Um int é representado num ficheiro binário tal como é guardado na memória (em formato máquina). Desta forma, espreitar para um ficheiro binário pode dar resultados ininteligíveis para nós, humanos.

9.2.1 Abrir um ficheiro de binário.

A abertura de um ficheiro binário é realizada recorrendo igualmente à função fopen(). No entanto, no modo de utilização do ficheiro deverá ser adicionada a letra 'b' para indicar que o ficheiro é binário (nem todos os compiladores necessitam da inclusão da letra 'b').

```
FILE *fh; /* É declarado um apontador para ficheiro. */

(...)

/* É aberto o ficheiro binário temperaturas.dat para leitura. */
fh = fopen("temperaturas.dat", "rb");
```

9.2.2 Escrever para um ficheiro binário.

As operações de escrita para um ficheiro de binário deverão ser realizadas com a função fwrite(). A função fwrite() toma como argumentos:

- o endereço do início do bloco de memória a copiar para o ficheiro;
- o tamanho de cada elemento (registo / ficha);
- o número de elementos (registos / fichas) a escrever;
- o apontador para o ficheiro para onde os dados deverão ser escritos.

```
typedef struct {
  char cidade[50];
  short temperatura;
} registo;

(...)

registo ficha;
FILE *fh;

fh = fopen("temperaturas.dat", "ab");

printf("Insira o nome da cidade: ");
gets(ficha.cidade);
printf("Insira a temperatura da cidade: ");
scanf("%hd", &ficha.temperatura);

fwrite(&ficha, sizeof(registo), 1, fh);
```

```
(...)
```

A linha de código que contêm a função fwrite() copia para o ficheiro a porção de memória identificada pela variável 'ficha'.

9.2.3 Ler de um ficheiro binário.

A leitura de um ficheiro binário é realizada utilizando a função fread(). A função fread() toma como argumentos:

- o endereço do início do bloco de memória para onde se deve copiar os conteúdos do ficheiro;
- o tamanho de cada elemento (registo / ficha);
- o número de elementos (registos / fichas) a ler;
- o apontador para o ficheiro de onde os dados deverão ser lidos.

```
typedef struct {
  char cidade[50];
  short temp;
} registo;

(...)

registo ficha;
FILE *fh;

fh = fopen("temperaturas.dat", "rb");
fread(&ficha, sizeof(registo), 1, fh);
printf("Cidade %s - Temp: %hd ºC.\n", ficha.cidade, ficha.temp);

(...)
```

A linha de código que contêm a função fread() copia do ficheiro para a variável 'ficha' um registo composto por um nome de cidade e respectiva temperatura.

9.2.4 Fechar um ficheiro binário.

Após todas as operações de leitura / escrita terem sido realizadas, dever-se-á fechar o ficheiro, com a função fclose(). Esta operação assegura que todos os dados ficam correctamente armazenados no ficheiro.

```
fclose(fh);
```

9.3 Orientação e navegação num ficheiro.

9.3.1 O ficheiro foi aberto correctamente?

A operação de abertura de um ficheiro pode falhar por várias razões, tais como:

- tentar abrir para leitura um ficheiro inexistente;
- criar um ficheiro num disco sem espaço disponível ou sem permissões de escrita (e.g. CD-ROM)...

Desta forma, convém certificar de que não houve quaisquer problemas antes de se proceder a qualquer leitura ou escrita no ficheiro. A função fopen() retorna um apontador nulo - NULL - quando falha. A seguinte rotina testa a abertura do ficheiro, e no caso de falha, aborta a execução do programa.

```
if ( fh = fopen("ficheiro", "w+") == NULL) {
  perror("fopen");
  exit(0);
}

(...)
```

9.3.2 Como é que eu sei que cheguei ao fim do ficheiro, quando estou a ler?

A função fread() retorna o número de elementos efectivamente lidos do ficheiro. Quando se chega ao fim do ficheiro, a função fread() não consegue ler mais elementos. A seguinte rotina detecta o fim do ficheiro.

```
if ( fh = fopen("ficheiro", "r") == NULL)
{
  printf("ERRO: Não foi possível abrir o ficheiro!\n");
  exit(0);
}

while( fread(&ficha, sizeof(registo), 1, fh) != 0 )
  printf("Cidade %s - Temp %hd ºC.\n", ficha.cidade, ficha.temp);
```

O ciclo while() é realizado enquanto o número de elementos lidos for diferente de zero.

9.3.3 Estive a consultar o ficheiro, mas agora queria realizar uma nova consulta... a partir do início do ficheiro!...

Existe um cursor que aponta para o local no ficheiro onde se deverá realizar a próxima escrita / leitura. Quando se abre o ficheiro nos modos "r" e ”w”, este cursor aponta para o início do ficheiro; quando o ficheiro é aberto em modo ”a”, o cursor é colocado no final do ficheiro.

Após uma operação de leitura ou escrita, este cursor avança. Quando se realizam leituras posteriores, são lidos os elementos seguintes. Esta funcionalidade facilita a leitura sequencial de vários elementos, mas não permite voltar atrás para ler elementos anteriores.

Existe a função rewind() que simplesmente coloca o cursor a apontar para o início do ficheiro. Aceita como único argumento o apontador para o ficheiro que se pretende "rebobinar".

```
rewind(fh);
```

Uma função mais versátil é a fseek() que é capaz de colocar o cursor em qualquer ponto do ficheiro. A função fseek() aceita como parâmetros:

- o apontador para o ficheiro;
- o "salto" (em número de bytes) a dar no ficheiro, podendo ser um número o positivo (salto para a frente); o negativo (salto para trás);
- a posição a partir da qual se pretende dar o "salto" o SEEK_SET se for a partir do início do ficheiro; o SEEK_CUR se for a partir da posição actual do cursor; o SEEK_END se for a partir do final do ficheiro.

```
/* Tem o mesmo efeito de rewind(fh); */
fseek(fh, 0, SEEK_SET);

(...)

/* Coloca o cursor no final do ficheiro. */
fseek(fh, 0, SEEK_END);
```

9.4 Exercícios.

1. Escreva um programa que realize as seguintes operações:
   - Deve escrever três frases (inseridas pelo utilizador) num ficheiro de texto.
   - Deverá ler as frases escritas no ficheiro referido na alínea anterior, e apresentá-las no ecrã.
2. Pretende-se armazenar as temperaturas de várias cidades de um determinado dia num ficheiro binário. O nome do ficheiro deverá seguir o seguinte formato DDMMAAAA.dat, em que:
   - DD é o dia;
   - MM é o mês;
   - AAAA é o ano. 2. Cada registo deverá conter o nome da cidade e a respectiva temperatura medida.

   (a) Escreva um programa que deverá guardar os registos das temperaturas relativas a uma data inserida pelo utilizador, num ficheiro binário.

   (b) Escreva um segundo programa que deverá apresentar as temperaturas medidas num determinado dia, e calcular a média das temperaturas.