

Conceitos e Exercícios de Programação

Utilizando Linguagem C

Este texto destina-se a todos quanto se queiram iniciar na programação, desde as mais modestas às mais elevadas ambições. O método de aprendizagem consiste em explicar um conjunto de conceitos, suportados em exemplos simples, após o qual o leitor terá ao seu dispor um leque de exercícios desde simples a mais avançados, que lhe permitam interiorizar os conceitos com base na sua experiência de programação.

José Coelho
2010

PREFÁCIO

O presente texto destina-se a todos quantos queiram iniciar-se na programação. A linguagem de programação adoptada é a linguagem C, mas este texto não se centra na linguagem de programação mas sim nos conceitos, sendo a linguagem de programação apenas uma ferramenta para o ensino da programação.

Este texto tem origem num conjunto de explicações fornecidas no âmbito da Unidade Curricular (UC) de Programação da Universidade Aberta, de forma a responder às dificuldades sentidas pelos estudantes. As explicações foram organizadas em conceitos essenciais, que são introduzidos com uma necessidade que os conceitos introduzidos anteriormente não satisfazem, e constituídos por uma explicação do conceito, seguida de um ou mais exemplos. Os exemplos são tão simples quanto possível, de forma a ser clara a aplicação do conceito. A compreensão completa dos exemplos passa por analisar a execução passo a passo de forma a remover qualquer dúvida que o leitor tenha sobre a forma como é executado o exemplo que exemplifica o conceito em causa, e desta forma facilmente interiorizar os conceitos essenciais. Cada conceito termina com um conjunto de erros comuns mais directamente ligados ao conceito, e respectivos exemplos, para que ao serem compreendidos não sejam cometidos. No anexo está a lista completa de erros comuns.

A estes conceitos juntou-se o pacote de actividades formativas a realizar na UC, aqui chamados de exercícios, de nível de dificuldade desde o mais simples (exercícios verdes), ao mais desafiante (exercícios vermelhos), passando pela dificuldade média (exercícios azuis). Foi tido o maior cuidado na escolha dos exercícios, de forma a pedir apenas exercícios não abstractos, que possam ser reutilizados, e inspirados em matérias mais avançadas a abordar em outras UCs, para assim tirar o maior partido possível do trabalho do leitor. Aos enunciados dos exercícios foi dada a maior atenção, tendo sido já utilizados e revistos com o decorrer da UC, para que o leitor não sinta que a principal dificuldade é compreender o enunciado do exercício mas sim na sua realização. No anexo encontram-se mais dicas sobre os exercícios, uma forma de verificar se o exercício está correcto sem ver uma resolução, e as resoluções dos exercícios para comparação, após resolvê-los.

Os conceitos foram organizados primeiramente em páginas isoladas, e posteriormente ordenadas por ordem de importância, o mais importante primeiro. Os conceitos passaram a capítulos neste texto, que estão agrupados em três partes, a que correspondem os módulos na UC, e em que são pontos apropriados para uma paragem mais longa no estudo, ou para a realização de actividades de avaliação. Os exercícios estão associados às partes, e não aos capítulos.

A metodologia de estudo proposta assenta no princípio que programar aprende-se programando, pelo que não se apresentam capítulos com muitos exercícios resolvidos, mas apenas os conceitos essenciais para que o leitor possa resolver por si os exercícios e adquirir a experiência de programação que é apenas sua e intransmissível. Se o leitor tiver aspirações modestas e disponha de pouco tempo, deve no final de cada parte fazer apenas os exercícios verdes. Caso não consiga realizar algum exercício, deve reler os capítulos anteriores. Aconselha-se a planear um estudo regular, cada sessão de estudo com 1 a 2 horas, e uma periodicidade entre 1 dia a 1 semana. Não deve ler mais que um capítulo na mesma sessão de estudo, e deve fazer os exercícios em sessões distintas da leitura dos capítulos. Caso o leitor tenha aspirações mais elevadas e disponha de mais tempo, deve fazer tudo o que atrás se aconselhou, mais a realização dos exercícios azuis e vermelhos.

Se o leitor tiver oportunidade de estudar em grupo, aconselha-se a que aplique em estudo isolado a metodologia atrás referida, e partilhe ideias e exercícios após os resolver, comentando as opções

tomadas. O estudo deve estar sincronizado para poder haver alguma vantagem na troca de ideias após a leitura de cada capítulo e análise dos exercícios realizados. Não deve no entanto cair no erro de aguardar pela resolução dos exercícios, ou pelas explicações do colega sobre uma parte do texto. Ninguém pode aprender por outra pessoa. Deve tirar partido de haver outras pessoas a estudar o mesmo assunto, e não prescindir de aprender.

A escolha da linguagem de programação C deve-se a essencialmente dois motivos. Primeiro trata-se de uma linguagem estruturada imperativa, que se considera ideal para aprender a programar. Numa linguagem orientada por objectos tem que se de utilizar muitas caixas pretas, e dificilmente se consegue explicar alguns conceitos que apenas fazem sentido em programas de maior dimensão, algo que não deve ser prioridade de quem se inicia na programação. As linguagens funcionais e declarativas, atendendo a que pertencem a paradigmas computacionais alternativos, embora possíveis na iniciação à programação, não se consideram apropriadas uma vez que os ganhos seriam muito limitados devido às suas fracas utilizações reais. Segundo, a linguagem C é uma linguagem de alto nível, bastante antiga, amplamente divulgada e em utilização, implementada e disponível em praticamente todas as plataformas. Com a sua simplicidade permite compiladores que tirem o melhor partido da máquina, resultando normalmente em binários mais eficientes que em outras linguagens. Todos os conceitos aqui introduzidos serão úteis não só em outras linguagens estruturadas, como na programação orientada por objectos.

ÍNDICE

PARTE I – VARIÁVEIS E ESTRUTURAS DE CONTROLO	6
1. Primeiro Programa.....	7
2. Variáveis.....	9
3. Condicionais	16
4. Ciclos.....	23
1) Exercícios.....	29
PARTE II – FUNÇÕES, VECTORES E RECURSÃO	39
5. Funções	40
6. Mais Ciclos e Condicionais.....	49
7. Vectores	57
8. Procedimentos	69
9. Recursão.....	78
2) Exercícios.....	83
PARTE III – MEMÓRIA, ESTRUTURAS E FICHEIROS	99
10. Memória	100
11. Estruturas.....	113
12. Ficheiros.....	130
13. Truques	143
3) Exercícios.....	151
PARTE IV – ANEXOS	160
14. Compilador e Editor de C	161
15. Não consigo perceber o resultado do programa	164
16. Funções standard mais utilizadas	169
17. Erros Comuns	170

18.	Exercícios: Dicas, Respostas e Resoluções.....	176
------------	---	------------

PARTE I – VARIÁVEIS E ESTRUTURAS DE CONTROLO

Na Parte I são introduzidos os primeiros conceitos de programação, existentes em todas as linguagens de programação modernas. A leitura atenta e pausada é essencial para a compreensão destes conceitos que embora aparentemente simples, a sua não compreensão tem consequências em todo o resto do texto, mas que poderá detectar de imediato no final da Parte I ao não conseguir resolver alguns dos exercícios propostos. Após a realização desta parte, fica com as bases para programar em qualquer linguagem de programação simples, para implementação de pequenas funcionalidades, como Javascript para adicionar dinamismo a uma página Web, VBA para programar macros no Microsoft Excel, ou Linguagem R para realizar cálculos estatísticos.

1. PRIMEIRO PROGRAMA

Um **programa** é uma sequência de instruções, dirigidas a um computador, para que este execute uma determinada função pretendida.

A actividade de **programar** consiste na elaboração dessa sequência de instruções numa determinada linguagem de programação, que satisfaz o que é pretendido.

A **linguagem de programação** tem que seguir uma sintaxe rígida, de forma a poder ser convertida em instruções que o computador possa compreender, mas por outro lado deve ser facilmente escrita e lida por uma pessoa.

Há diferentes linguagens de programação, utilizaremos a **linguagem C**, no entanto não será a sintaxe da linguagem o mais relevante a aprender, mas sim competências que são independentes da linguagem.

A linguagem C é uma **linguagem imperativa**, cada instrução é uma ordem para o computador. Existem outros paradigmas de programação, mas fora do âmbito deste texto.

Manda a tradição que o primeiro programa a desenvolver numa linguagem, seja o **olá mundo**. Não a vamos quebrar, este é o programa mais curto possível, apenas escreve um texto e termina.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Ola Mundo!");
6 }
```

Programa 1-1 Olá Mundo

O programa acima está escrito em linguagem C, e pode ser executado pelo computador. No entanto o computador não entende directamente a linguagem C, apenas ficheiros executáveis (no Windows com extensão `exe`). No entanto, através de um programa chamado de **Compilador**, podemos converter o código C acima num executável. Esse executável contém já instruções em código máquina, cujo computador pode executar.

Vamos ver em que consiste o código acima:

- Existe **numeração das linhas e cores**, que são apenas para nossa conveniência. A numeração das linhas não faz parte do ficheiro C, e as cores são colocadas pelo editor. Um programa em C é um ficheiro de texto plano, com a extensão `.c` e pode ser editado num editor de texto plano como o Notepad, sem qualquer cor ou numeração das linhas.
- **Linha 1:** começa por um `#` (cardinal), pelo que esta linha é uma instrução não para o computador mas sim para o compilador. Essas instruções chamam-se de comandos de **pré-processamento**, dado que são executadas antes do processamento.
- O comando **`#include`** instrui o compilador a incluir o ficheiro indicado de seguida. Esse ficheiro é uma biblioteca standard do C (existe em todos os compiladores de C), `stdio.h`, e consiste na declaração de funções de entrada e saída. Estas funções podem ser utilizadas de seguida no programa, sendo utilizada neste caso a função `printf`.
- **Linha 2:** esta linha não tem nada. Podem existir quantas linhas em branco o programador quiser, o compilador não ficará atrapalhado, nem fará nada com elas, mas tal pode ser útil

para espaçar os comandos e tornar mais legível o programa. O mesmo é válido para os espaços ou tabs numa linha de código, como por exemplo os espaços da linha 5 que antecedem o `printf`.

- **Linha 3:** começa a implementação da função `main`. O programa começa sempre na função `main`. Para já, vamos deixar por explicar o `int`, e os parênteses vazios. Realçar que as instruções que dizem respeito à função `main`, e portanto ao que o programa tem de fazer, estão limitadas por duas chavetas a abrir na linha 4 e a fechar na linha 6. Todas as instruções entre estas chavetas, pertencem à função `main`.
- **Linha 5:** nesta linha está a única instrução. Existe uma chamada à função `printf`, e sabemos que é uma função porque após o seu nome tem uns parênteses a abrir. Dentro dos parênteses colocam-se os argumentos a passar à função. Neste caso colocamos o texto que queremos imprimir: "Ola mundo!". O texto coloca-se entre aspas, para não se confundir com instruções.

Se não o fez já, crie e execute o programa acima. Veja as instruções para instalar um compilador no anexo "Compilador e Editor de C".

É tudo. Acabou de dar o primeiro passo no mundo da programação, editou e compilou o primeiro programa. Houve conceitos referidos aqui que não compreende agora completamente: biblioteca; função; argumentos. Para já não lhes dê muito importância.

2. VARIÁVEIS

Uma **variável** num programa é uma entidade que tem um valor a cada instante, podendo esse valor ao longo do programa ser utilizado e/ou alterado.

É nas variáveis que se **guarda a informação** necessária para realizar a função pretendida pelo programa. Podem existir quantas variáveis forem necessárias, mas cada uma **utiliza memória** do computador.

O programa seguinte tem por objectivo trocar o valor de duas variáveis:

```
1 #include <stdio.h>
2
3 int main()
4 {
5     /* declaração de três variáveis inteiras */
6     int x=3;
7     int y=7;
8     int aux;
9
10    /* trocar o valor de x com y */
11    aux=x;
12    x=y;
13    y=aux;
14
15    /* mostrar os valores em x e em y */
16    printf("x: %d, y: %d",x,y);
17 }
```

Programa 2-1 Troca o valor de duas variáveis

Comentários:

- Este programa tem uma estrutura inicial idêntica ao Programa 1-1, nomeadamente nas linhas 1 a 4, e linha 17.
- A linha 5, 10 e 15 têm **comentários** ao código. Os comentários são todo o texto entre /* e */, sendo ignorados pelo compilador. O objectivo é que seja possível ao programador colocar texto explicativo de partes mais complexas do programa, não para que o compilador o melhor entenda, mas para que um outro programador que leia o código, ou o próprio programador mais tarde, compreenda mais facilmente o código. Os comentários ficam a cor verde, dado que o editor identifica que o conteúdo dessas linhas será ignorado pelo compilador¹.
- Nas linhas 6 a 8, estão **declaradas as variáveis** x, y e aux. As variáveis x e y são não só criadas, como também inicializadas. Ao ficarem com um valor inicial, podem desde já ser utilizadas. Na linguagem C, todas as variáveis utilizadas têm de ser declaradas no início da função².
- Nas linhas 11 a 13, estão comandos que são **atribuições**. Numa atribuição, coloca-se uma **variável do lado esquerdo** da igualdade, e uma **expressão do lado direito**. Neste caso, todas as atribuições têm expressões com uma só variável.

¹ A linguagem C++ tem os comentários até ao final da linha, após duas barras: "//". A maior parte dos compiladores de C também permite este tipo de comentários.

² A linguagem C++ bem como a generalidade dos compiladores de C permitem a declaração fora do início das funções, mas pelos motivos referidos no erro "Declaração de variáveis fora do início das funções", do anexo "Erros Comuns" esta restrição será mantida.

- Na linha 16, é mostrado o valor das variáveis. Notar que a função `printf` tem 3 argumentos separados por vírgulas. O primeiro é **uma string**, em que **começa com aspas e acaba nas aspas seguintes**, o segundo uma expressão com a variável `x`, seguido de uma expressão com a variável `y`. Na string existem dois `%d`. Em cada `%d` o `printf` em vez de mostrar no ecrã `%d` coloca o valor da variável seguinte na sua lista de argumentos. No primeiro `%d` vai colocar `x`, e no segundo `%d` vai colocar o `y`.
- Notar que cada instrução acaba com um ponto e vírgula, sendo esta uma característica do C.

Execução do programa:

```
C:\>troca
x: 7, y: 3
```

Os valores estão correctamente trocados, mas vamos ver com mais detalhe qual o valor das variáveis a cada momento, na **execução passo-a-passo**, à direita.

Passo	Linha	Instrução	Resultado		
1	6	<code>int x=3;</code>	x=3		
2	7	<code>int y=7;</code>	x=3	y=7	
3	8	<code>int aux;</code>	x=3	y=7	aux=?
4	11	<code>aux=x;</code>	x=3	y=7	aux=3
5	12	<code>x=y;</code>	x=7	y=7	aux=3
6	13	<code>y=aux;</code>	x=7	y=3	aux=3
7	16	<code>printf(...);</code>	x: 7, y: 3	x=7	y=3

A execução passo-a-passo tem em cada linha da tabela um passo. Em cada passo é indicada a linha de código correspondente, e a respectiva instrução por vezes encurtada como no passo 7. A coluna resultado mostra o resultado da instrução, excepto se tratar de uma declaração, em que uma ou mais variáveis ocupam a coluna seguinte disponível, ou uma atribuição, em que a respectiva variável fica com o valor atribuído. Para poder-se escrever a linha de um determinado passo, é suficiente a informação da linha do passo anterior.

A pergunta "Qual o valor de `x`?" não tem muito sentido, a não ser que se identifique o instante. No passo 3 ou no final do programa? Uma variável tem um valor até que lhe seja atribuído outro.

Vamos agora resolver o mesmo problema, troca dos valores de duas variáveis, mas sem utilizar uma variável auxiliar:

```
1 #include <stdio.h>
2
3 int main()
4 {
5     /* declaração de duas variáveis inteiras */
6     int x=3, y=7;
7
8     /* trocar o valor de x com y */
9     x=x+y;
10    y=x-y;
11    x=x-y;
12
13    /* mostrar os valores em x e em y */
14    printf("x: %d, y: %d",x,y);
15 }
```

Programa 2-2 Troca o valor de duas variáveis sem variável auxiliar

A diferença para o exemplo anterior é que não se declarou uma variável `aux`, e declarou-se ambas as variáveis `x` e `y` num só comando. As atribuições utilizam agora expressões com duas variáveis, incluindo a própria variável que está a ser atribuída.

Passo	Linha	Instrução	Resultado		
1	6	<code>int x=3, y=7;</code>	x=3	y=7	
2	9	<code>x=x+y;</code>	x=3+7=10	y=7	
3	10	<code>y=x-y;</code>	x=10	y=10-7=3	
4	11	<code>x=x-y;</code>	x=10-3=7	y=3	
5	14	<code>printf(...);</code>	x: 7, y: 3	x=7	y=3

Embora este programa pareça confuso, o certo é que tem a mesma funcionalidade que o anterior como se comprova de seguida, vendo a execução passo-a-passo. No passo 2, o valor de `x` utilizado na expressão é o que existia antes do passo 2, o valor 3, passando a ser o valor 10 após a execução do passo 2. Não há qualquer problema em colocar a variável `x` de um lado e do outro de uma atribuição. A variável `x` à esquerda, significa que ficará com o resultado da expressão, e a variável `x` à direita, significa que se deve utilizar o valor da variável `x` antes desta instrução.

Os tipos de dados utilizados podem não ser apenas variáveis inteiras. Os **tipos elementares** da linguagem C são os seguintes:

- **char** - um carácter, ou um inteiro muito pequeno (guarda apenas 256 valores distintos)
- **short** - inteiro pequeno (pouco utilizado)
- **int** - tipo standard para inteiros
- **long** - inteiro longo
- **float** - número real (pode ter casas decimais)
- **double** - número real com precisão dupla (igual ao `float` mas com mais precisão)

Pode-se nos tipos inteiros (`char` a `long`), anteceder com as palavras `unsigned` ou `signed`, para indicar que o número inteiro tem ou não sinal. Se omitir, é considerado que têm sinal. Pode-se também considerar o tipo **long long** para um inteiro muito longo³, ou o **long double**, mas esses tipos nem sempre estão implementados.

Quais os valores máximos/mínimos que podemos atribuir às variáveis de cada tipo? Isso depende do espaço reservado para cada tipo de variável, que por sua vez depende tanto do compilador, como do computador em que o programa for compilado e executado. Para saber quanto espaço é necessário para guardar cada variável, pode-se utilizar o operador **sizeof**. O resultado de `sizeof(int)` retorna o número de bytes que uma variável do tipo inteiro ocupa na memória.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("sizeof(int): %d",sizeof(int));
6 }
```

Programa 2-3 Imprime o número de bytes que um inteiro ocupa

Execução do programa:

```
C:\>sizeofint
sizeof(int): 4
```

Execute este programa para ver quantos bytes ocupa um inteiro no seu compilador e computador. Para que os resultados sejam idênticos na resolução dos exercícios, é importante que todos utilizem inteiros do mesmo tamanho. Se o resultado não for no seu caso 4, experimente o `sizeof(short)` ou `sizeof(long)`, e **utilize o tipo que tiver tamanho 4** para todas as variáveis inteiras nos exercícios das actividades formativas.

As variáveis do tipo `char` e `float/double`, não utilizam a mesma **string de formatação** no `printf` da que os inteiros. Para os inteiros, utiliza-se o `%d` na string do primeiro argumento, mas esse

³ O tipo `long long` não faz parte da norma inicial da linguagem C, actualmente a generalidade dos compiladores implementa-o com um inteiro de 64 bits. A string de formatação varia: `%I64d`; `%lld`. Para uma solução mais portátil, ver `inttypes.h` e utilizar `PRId64`.

código significa que se pretende colocar lá um valor inteiro. Se pretendermos colocar um carácter ou um valor real, tem que se utilizar o **%c** (carácter) ou **%f** (float), ou ainda **%g** (para mostrar o real na notação científica). Note-se no entanto que o carácter tanto pode ser considerado como um carácter ou como com um inteiro pequeno.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char c='a';
6     float x=12354234.2346;
7     double y=12354234.2346;
8
9     /* mostrar valores */
10    printf("c: %c (%d), x: %f, y: %g", c, c, x, y);
11 }
```

Programa 2-4 Imprime diferentes tipos de variáveis

Comentários:

- O carácter **c** é atribuído com a letra "a". Os **caracteres são delimitados entre plicas**, em vez de aspas, para não se confundirem com outros identificadores.
- O **printf** tem ordem para:
 - No **%c** ir buscar a primeira variável e colocar um carácter (a variável **c**)
 - No **%d** ir buscar a segunda variável (**c** novamente), e colocar o seu valor numérico
 - No **%f** ir buscar a terceira variável (**x**), e coloca o número real com parte decimal
 - No **%g** ir buscar a quarta variável (**y**), e coloca o número real em notação científica

Execução do programa:

```
C:\>charfloat
c: a (97), x: 12354234.000000, y: 1.23542e+007
```

Note que o valor numérico da letra 'a' é o 97. Existe uma correspondência entre as letras e os números pequenos, mas esta questão será tratada mais à frente.

O valor de **x** não é o que está no programa. Isto acontece porque o tipo **float** tem uma precisão não muito grande (ocupa **4 bytes**). O tipo **double** tem uma precisão maior, mas em qualquer caso não se pode contar com a **precisão infinita** do valor lá colocado. Uma variável real fica sempre com o valor mais próximo que a representação interna do número permite, e não com o valor exacto.

A função **scanf** é a função inversa ao **printf**, e serve para **introduzir valores em variáveis**, **valores** introduzidos pelo utilizador, em vez de os mostrar como no **printf**.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char c;
6     int x;
7     double d;
8
9     printf("Introduza um caracter: ");
10    scanf("%c", &c);
11    printf("Introduza um inteiro: ");
12    scanf("%d", &x);
13    printf("Introduza um real: ");
14    scanf("%lf", &d);
15 }
```

```
16 printf("valores introduzidos: %c %d %f",c,x,d);
17 }
```

Programa 2-5 Pede valores de variáveis de diferentes tipos que depois imprime

No código acima pode-se ver parecenças do `scanf` com o `printf`, mas há um **&** antes de cada variável. Porquê? Esta questão apenas pode ser respondida completamente na Parte III, por agora explica-se apenas que se destina a poder **fazer uma atribuição** e não uma leitura às variáveis. Notar que foi utilizado para ler o valor para `double`, `%lf`, e não `%f`, dado que o real é um `double`, e não um `float`. Pela mesma razão que deve utilizar inteiros de 4 bytes, deve utilizar **para os números reais o tipo `double`**.

Execução do programa:

```
C:\>scanf
Introduza um caracter: c
Introduza um inteiro: 123
Introduza um real: 142323.2435
Valores introduzidos: c 123 142323.243500
```

Na **execução do programa** o texto introduzido pelo utilizador vem a **bold** e *itálico*. Como o programa mostra o conteúdo das variáveis que leu na última linha, está verificado que as variáveis foram lidas correctamente.

Consegue agora criar variáveis dos tipos básicos na linguagem C, ler e mostrar os valores das variáveis, efectuar atribuições e se necessário fazer uma execução detalhada e ver o valor das variáveis em cada instante.

No anexo está listado um conjunto de **erros comuns** para o ajudar a melhorar a qualidade do seu código. No entanto, muitos dos erros só os compreenderá após o estudo da matéria correspondente, e outros apenas quando lhe for apontado no seu código uma ocorrência do erro. No final de cada capítulo são introduzidos os erros comuns, que estão mais directamente relacionados com o capítulo. Neste capítulo temos quatro erros:

- **Declarações ou atribuições a variáveis nunca utilizadas**
 - **Forma:** Por vezes declaram-se variáveis, ou fazem-se atribuições a variáveis, um pouco para ver se o programa passa a funcionar. No entanto, após muito corte e costura, por vezes ficam atribuições a variáveis que na verdade nunca são utilizadas, embora tal não afecte o bom funcionamento do programa.
 - **Problema:** Se há uma variável declarada que não é utilizada, essa variável pode ser removida e o código fica igual. Se há uma atribuição a uma variável que depois não é utilizada em nenhuma expressão, então a atribuição é desnecessária. Ter variáveis, expressões, atribuições a mais, é uma situação indesejável, dado que compromete a leitura do código que realmente funciona.
 - **Resolução:** Verificar se há variáveis declaradas que não são utilizadas, e apagá-las. Para cada atribuição, seguir o fluxo do código até uma possível utilização. Se não existir, apagar a atribuição, reiniciando o processo de verificação. No final o código real poderá ser muito mais reduzido, do que o gerado em situações de stress em modo de tentativa/erro.
- **Uma atribuição, uma leitura**

- **Forma:** Uma variável é atribuída com um valor, para depois utilizá-lo na instrução seguinte, não sendo mais necessário o valor da variável. Por exemplo, para reunir os valores necessários à chamada de uma função.
- **Problema:** Esta situação pode indicar a existência de uma variável desnecessária, e quantas mais variáveis desnecessárias o código tiver, mais complexo fica, para não falar que ocupam memória desnecessariamente. É apenas justificável no caso de a expressão ser muito complexa, de forma a clarificar o código.
- **Resolução:** Ao fazer a atribuição do valor à variável, utiliza uma expressão. Essa variável é depois utilizada de seguida também uma só vez, pelo que mais vale colocar a própria expressão no local onde a variável é utilizada.
- **Variáveis desnecessárias**
 - **Forma:** Ao criar uma variável por cada necessidade de registo de valores, pode acontecer haver variáveis sempre com o mesmo valor, mas tal não impede que o programa funcione correctamente.
 - **Problema:** Ter muitas variáveis com a mesma função tem o mesmo problema que “uma atribuição, uma leitura”. Fica o código mais complexo e difícil de ler sem qualquer ganho. Pode acontecer que este problema se reflita não apenas por ter duas variáveis sempre com os mesmos valores, mas terem valores com uma relação simples (o simétrico, por exemplo).
 - **Resolução:** Deve identificar as variáveis com atribuições e utilizações perto uma da outra, eventualmente os nomes, e verificar se pode a cada momento utilizar o valor de uma variável em vez da outra. Por vezes não é simples a identificação/remoção destas situações quando há muitas variáveis. É preferível aplicar primeiramente a remoção de declarações/atribuições desnecessárias, e aplicar a resolução do erro “uma atribuição, uma leitura”, antes deste erro.
- **Comentários explicam a linguagem C**
 - **Forma:** De forma a explicar tudo o que se passa no programa, e não ser penalizado, por vezes desce-se tão baixo que se explica inclusive linguagem C.
 - **Problema:** Esta situação é um claro excesso de comentários, que prejudica a legibilidade. Quem lê código C sabe escrever código C, pelo que qualquer comentário do tipo “atribuir o valor X à variável Y”, é desrespeitoso para quem lê, fazendo apenas perder tempo. Ao comentar instruções da linguagem e deixar de fora comentários mais relevantes, pode indicar que o código foi reformulado por quem na verdade não sabe o que este realmente faz.
 - **Resolução:** Escreva como se estivesse a escrever a si próprio, supondo que não se lembrava de nada daqui a uns 6 meses.

No Programa 2-4 podemos ver uma ocorrência do erro “uma atribuição, uma leitura”, dado que todas as variáveis são atribuídas na declaração e utilizadas uma só vez. Neste caso pretende-se mostrar a string de formatação para os diversos tipos, mas poder-se-ia igualmente ter colocado os próprios valores no `printf`. O mesmo programa tem ainda uma ocorrência do erro “comentários explicam a linguagem C”, e este erro será cometido ao longo dos programas apresentados neste texto, dado que o código se destina a quem está a aprender, mas num programa C não faz sentido colocar um comentário “mostrar valores” antes do `printf`.

Não há ocorrências dos outros dois erros nos programas deste capítulo, e a sua exemplificação seria forçada. A melhor fonte de exemplos tem de ser o seu código, após resolver os exercícios, procure identificar ocorrências desses erros.

O conceito de variável, aparentemente simples, não é no entanto tão facilmente interiorizado como possa pensar. Saberá no primeiro exercício que seja necessária uma variável auxiliar não declarada no enunciado, se consegue ou não identificar a variável e seu modo de utilização, ou se por outro lado considera o enunciado complexo e confuso.

Se for o segundo caso, antes de passar para o exercício seguinte, é imperativo que olhe para a solução que acabará por obter, com a sua máxima atenção. A experiência levará a que identifique imediatamente as variáveis que necessita para resolver cada problema, sendo esta uma das principais características de um programador experiente.

3. CONDICIONAIS

As instruções dadas nos programas até aqui são sequenciais, isto é, existindo 4 instruções estas vão sendo executadas sem sequência, numa determinada ordem fixa. No entanto, o que fazer se pretendemos ter um programa que tenha mais que uma **execução alternativa**? Por exemplo, estamos interessados em determinar se um determinado número inteiro introduzido pelo utilizador, é par ou ímpar. Como há duas possibilidades para apresentar ao utilizador, tem de existir duas instruções alternativas, uma para o caso do número ser par, e outra para o caso de ser ímpar. Para além disso, temos de saber determinar se o número é par ou ímpar.

A linguagem C permite que um determinado código seja executado dependente do valor de uma **expressão lógica**. Se a expressão for verdadeira, executa uma instrução, caso contrário executa a outra instrução.

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int numero;
6     printf("Indique um numero: ");
7     scanf("%d",&numero);
8     if(numero%2==0)
9         printf("par");
10    else
11        printf("impar");
12 }
```

Programa 3-1 Determina se um número é par ou ímpar

Comentários:

- Linha 8: a instrução **if** tem entre parênteses uma expressão lógica. Essa expressão utiliza o operador lógico **==** (não confundir com a atribuição), e do lado esquerdo utiliza o operador **%** (resto da divisão). Se o resto da divisão do número por 2 for zero, então a expressão lógica é verdadeira e executa a instrução da linha 9
- Linha 10: existe uma instrução **else**. Essa instrução está associada ao **if**, mas o **if** pode ser utilizado sem o **else**. Se existir indica uma instrução a executar no caso da expressão lógica ser falsa

Execução do programa:

```

C:\>paridade
Indique um numero: 234
par
C:\>paridade
Indique um numero: 23
impar
```

O resultado foi o pretendido, vejamos agora a execução passo-a-passo. Nesta execução o número introduzido foi o "234", levando a que o condicional fosse verdadeiro e que no passo 5 a linha 9 fosse executada.

Passo	Linha	Instrução	Resultado	
1	5	int numero;		numero=?
2	6	printf(...);	Indique um numero:	numero=?
3	7	scanf(...,&numero);	234	numero=234
4	8	if(numero%2==0)	0==0 Verdade	numero=234
5	9	printf("par");	par	numero=234

Na segunda execução, o valor introduzido foi 23, levando a que o condicional fosse falso e que a linha 11 fosse executada no passo 5, em vez da linha 9.

Passo	Linha	Instrução	Resultado	
1	5	int numero;		numero=?
2	6	printf(...);	Indique um numero:	numero=?
3	7	scanf(...,&numero);	23	numero=23
4	8	if(numero%2==0)	1==0 Falso	numero=23
5	11	printf("impar");	impar	numero=23

Podem ser utilizados os operadores lógicos normais em expressões lógicas, e os parênteses curvos que forem necessários:

- A e B são expressões lógicas:
 - A || B » A ou B
 - A && B » A e B
 - ! A » não A
- A e B expressões numéricas:
 - A == B » A igual a B
 - A != B » A diferente de B
 - A > B » A maior que B
 - A >= B » A maior ou igual a B
 - A < B » A menor que B
 - A <= B » A menor ou igual a B
- Nas expressões numéricas:
 - A + B » A mais B
 - A - B » A menos B
 - A * B » A vezes B
 - A / B » A a dividir por B
 - A % B » resto da divisão de A por B

Vamos aplicar estes operadores num problema mais complexo. Pretende-se saber se um ano é ou não bissexto. Um ano é bissexto se for múltiplo de 4, mas de 100 em 100 anos esta regra não é válida, excepto de 400 em 400 anos. Temos um problema que também requer apenas duas execuções alternativas, mas com uma expressão lógica mais complexa. O programa vai ter a mesma estrutura que o anterior, mas tem que se construir a expressão lógica. Vejamos:

- Múltiplo de 4:
 - ano % 4 == 0 » resto da divisão com 4 é nula, portanto o ano é múltiplo de 4
- Excepto de 100 em 100 anos:
 - ano % 4 == 0 && ano % 100 != 0 » retorna verdadeiro para múltiplos de 4, mas se for múltiplo de 100 já não
- Excepto de 400 em 400 anos:
 - (ano % 4 == 0 && ano % 100 != 0) || ano % 400 == 0 » como a excepção é positiva, coloca-se uma disjunção e não uma conjunção como no caso anterior

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int ano;
6     printf("Indique ano: ");
7     scanf("%d", &ano);
8
9     /* teste de ano bissexto */
10    if(ano%4==0 && ano%100!=0 || ano%400==0)
11        printf("Bissexto");
12    else
13        printf("Normal");
14 }

```

Programa 3-2 Determina se um ano é normal ou bissexto

Notar que não foram colocados parênteses para a conjunção. Não é necessário dado que tem sempre prioridade relativamente à disjunção.

Execução do programa:

```

C:\>bissexto
Indique ano: 2345
Normal
C:\>bissexto
Indique ano: 2344
Bissexto

```

Na execução passo-a-passo pode-se ver que na expressão lógica do passo 4, o segundo argumento do operador && (AND) não foi avaliado porque o primeiro argumento já era falso.

Passo	Linha	Instrução	Resultado	
1	5	int ano;		ano=?
2	6	printf(...);	Indique ano:	ano=?
3	7	scanf(...,&ano);		2345 ano=2345
4	8	if(ano%4==0 && ...)	(1==0 && (?) 345==0) Falso	ano=2345
5	13	printf("Normal");	Normal	ano=2345

Desta vez o segundo argumento do operador || (OR) não foi avaliado porque o primeiro argumento já era verdadeiro.

Passo	Linha	Instrução	Resultado	
1	5	int ano;		ano=?
2	6	printf(...);	Indique ano:	ano=?
3	7	scanf(...,&ano);		2344 ano=2344
4	8	if(ano%4==0 && ...)	(0==0 && 44!=100 (?)) Verdade	ano=2344
5	11	printf("Bissexto");	Bissexto	ano=2344

Pode acontecer que seja necessário mais que uma instrução dentro do if, ou que sejam necessárias várias instruções if encadeadas. É o caso do programa seguinte, em que está implementado um algoritmo para determinar o número de dias do mês, dado o número do ano e mês.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int ano, mes, dias;
6      printf("Indique ano: ");
7      scanf("%d", &ano);
8      printf("Indique mes: ");
9      scanf("%d", &mes);
10
11     if(mes==2)
12     {
13         /* teste de ano bissexto */
14         if(ano%400==0 || ano%4==0 && ano%100!=0)
15             printf("29");
16         else
17             printf("28");
18     } else if(mes==1 || mes==3 || mes==5 || mes==7 ||
19             mes==8 || mes==10 || mes==12)
20     {
21         printf("31");
22     } else
23     {
24         printf("30");
25     }
26 }

```

Programa 3-3 Determina o número de dias de um mês/ano

Comentários:

- O primeiro `if` testa se o mês é 2 (Fevereiro). Neste caso tem que se testar se o ano é ou não bissexto de forma a retornar 29 ou 28. Utilizamos o código do exercício anterior, mas teve de ser colocado dentro do `if`. Para não haver confusões com o `else` a que pertence este `if`, colocou-se o `if` num bloco (dentro de chavetas).
- Um **bloco de código** pode ser colocado em qualquer lado em que se coloca um só comando, como é o caso do `if`. Assim é possível executar quantas instruções se quiser utilizando o mesmo condicional.
- Após se saber que o mês não é 2, tem que se testar se é um mês de 31 dias, ou se é um mês de 30 dias. Tem de portanto existir outro `if` logo a seguir ao `else`. Embora seja um `if` distinto, está em sequência do primeiro `if` e na verdade permite que o programa siga um de três caminhos possíveis (Fevereiro / mês 31 / mês 30), e não apenas 2 como aconteceria se utilizar apenas um só `if`.
- A expressão lógica para determinar que o mês tem 31 dias é longa, pelo que muda de linha. Não há problema com isso, mas neste caso foi uma má opção, podendo-se ter testado se o mês tinha 30 dias, que são menos meses (4, 6, 9 e 11).

Execução do programa:

```

C:\>diasdomes
Indique ano: 2344
Indique mes: 2
29
C:\>diasdomes
Indique ano: 2342
Indique mes: 4
30

```

Esta execução tem dois condicionais encadeados, o passo 7

Passo	Linha	Instrução	Resultado			
1	5	int ano, mes, dias;		ano=?	mes=?	dias=?
2	6	printf(...);	Indique ano:	ano=?	mes=?	dias=?
3	7	scanf(...,&ano);	2344	ano=2344	mes=?	dias=?
4	8	printf(...);	Indique mes:	ano=2344	mes=?	dias=?
5	9	scanf(...,&mes);	2	ano=2344	mes=2	dias=?
6	11	if(mes==2)	(2==2) Verdade	ano=2344	mes=2	dias=?
7	14	if(ano%400==0 ...)	(344==0 0==0 && 44!=0) Verdade	ano=2344	mes=2	dias=?
8	15	printf("29");	29	ano=2344	mes=2	dias=?

está um segundo condicional que apenas é executado porque no passo 6 o primeiro condicional retornou verdadeiro. Notar que há uma variável `dias` que afinal não é necessária. Neste caso nem sequer é atribuída, pelo que pode ser apagada do código.

Na segunda execução o resultado no passo 6 é falso, pelo que a execução segue no passo 7 para um outro condicional sobre o respectivo `else`, não confundindo o `else` do condicional interior, dado que está dentro de um bloco de código.

Passo	Linha	Instrução	Resultado			
1	5	<code>int ano, mes, dias;</code>		ano=?	mes=?	dias=?
2	6	<code>printf(...);</code>	Indique ano:	ano=?	mes=?	dias=?
3	7	<code>scanf(...,&ano);</code>		2342	ano=2342	mes=? dias=?
4	8	<code>printf(...);</code>	Indique mes:	ano=2342	mes=?	dias=?
5	9	<code>scanf(...,&mes);</code>		4	ano=2342	mes=4 dias=?
6	11	<code>if(mes==2)</code>	(4==2) Falso	ano=2344	mes=4	dias=?
7	18	<code>if(mes==1 mes==3 ...)</code>	Falso	ano=2344	mes=4	dias=?
8	24	<code>printf("30");</code>		30	ano=2344	mes=4 dias=?

Não se pode deixar de referir que as **expressões lógicas em C** são valores inteiros, ou se quiser, um inteiro é o tipo de dados utilizado nas expressões lógicas, em que o valor zero significa o falso, e o valor não zero significa o verdadeiro. Com este conhecimento pode-se em vez do teste utilizado no primeiro exemplo:

```

8   if(numero%2==0)
9       printf("par");
10  else
11      printf("impar");

```

Fazer simplesmente:

```

8   if(numero%2)
9       printf("impar");
10  else
11      printf("par");

```

No entanto não se aconselha a fazer utilização abusiva deste conhecimento, dado que torna mais pesada a leitura do código. Cada expressão lógica deve ter pelo menos um operador lógico, de forma a facilitar a leitura do código e identificar a expressão como uma expressão lógica, e não confundir com uma expressão numérica, embora em C qualquer expressão numérica é uma expressão lógica.

Os erros comuns mais directamente associados a este capítulo são:

- **Indentação⁴ variável**

- **Forma:** Como a indentação é ao gosto do programador, cada qual coloca a indentação como lhe dá mais jeito ao escrever.
- **Problema:** A leitura fica prejudicada se a indentação não for constante. Por um lado não é fácil identificar se há ou não algum erro do programador em abrir/fechar chavetas, dado que a indentação pode ser sua opção e não esquecimento de abrir/fechar chavetas. Por outro lado, não é possível a visualização rápida das instruções num determinado nível, dado que o nível é variável, sendo necessário para compreender algo, de analisar o código por completo. Com o código indentado, pode-se ver o correcto funcionamento de um ciclo externo, por exemplo, sem ligar ao código interno do ciclo, e assim sucessivamente.

⁴ A indentação de uma instrução consiste em anteceder-lhe de um número de espaços constante (são utilizados 4 espaços neste texto), por cada bloco em que a instrução está inserida

- **Resolução:** Indente o código com 2 a 8 espaços, mas com o mesmo valor ao longo de todo o código. Se utilizar tabs, deve indicar o número de espaços equivalente, no cabeçalho do código fonte, caso contrário considera-se que o código está mal indentado. Se pretender subir a indentação com a abertura de uma chaveta, pode fazê-lo mas tem de utilizar sempre chavetas, caso contrário ficam instruções que estão no mesmo nível mas no código ficam e indentações distintas. Qualquer que seja as opções, não pode existir instruções no mesmo alinhamento que pertençam a níveis distintos (ou vice-versa). A posição das chavetas é indiferente, mas tem que se apresentar um estilo coerente ao longo do código.
- **Duas instruções na mesma linha**
 - **Forma:** Quando as instruções são pequenas e estão relacionadas, é mais simples colocá-las todas na mesma linha. Como é uma questão de estilo, não afecta o funcionamento.
 - **Problema:** Se há instruções relacionadas, estas devem ser colocadas na mesma função, se tal se justificar. Colocar duas ou mais instruções na mesma linha vai prejudicar a legibilidade e não só. O compilador quando encontra algum problema refere a linha do código, e existindo mais que uma instrução nessa linha, a informação é menos precisa. Deixa de ser possível também seguir um nível de indentação para passar por todas as instruções nesse nível, tem que se fazer uma leitura atenta por todo o código do bloco
 - **Resolução:** Em todas as linhas com mais que uma instrução, colocar uma instrução por linha
- **Linhas de código nunca executadas**
 - **Forma:** Por vezes em situações de stress tentam-se várias alternativas, e fica algum código que na verdade nunca tem hipótese de ser executado, mas que também não atrapalha.
 - **Problema:** Se há código a mais, este deve ser removido sobre pena de perda de legibilidade e manutenção. Esta situação é ampliada quando há código de diversas proveniências, em que ao integrá-lo ninguém se atreve a reeditar código que funciona, para seleccionar a parte do código que deve ser integrado.
 - **Resolução:** Identificar as zonas de código que de certeza que não são executadas de modo algum, e apagá-las. Tanto pode ser feito por inspecção ao código como através de testes, submetendo o código a diversos casos de teste, desde que previamente se tenha colocado informação de debug em cada bloco de código. Os blocos de código em que não tiver sido produzida nenhuma informação de debug, são provavelmente os blocos de código que nunca são executados.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int ano, mes, dias;
6     printf("Indique ano: "); scanf("%d", &ano);
7     printf("Indique mes: "); scanf("%d", &mes);
8
9     if(mes==2)
10    if(ano%400==0 || ano%4==0 && ano%100!=0)
11        printf("29");
12    else
13        printf("28");
14    else if(mes==1 || mes==3 || mes==5 || mes==7 ||
15            mes==8 || mes==10 || mes==12)
16        printf("31");
17    else if(mes!=2)
18        printf("30");
19    else
20        printf("erro");
21 }
```

O código acima é uma réplica do Programa 3-3 com a mesma funcionalidade, no entanto contendo os 3 erros indicados. A indentação na linha 5 é a 4 espaços, mas na linha 10 quando o condicional está debaixo do condicional na linha 9, não sobe a indentação. O leitor nessa situação não sabe se o que aconteceu foi um esquecimento da instrução para o primeiro condicional, ou se o que é pretendido é realmente um condicional sobre o outro. Os outros dois erros, o `printf` e o `scanf` estão na mesma linha, e a linha 20 nunca tem hipótese de ser executada, já que o condicional da linha 17 é sempre verdadeiro, uma vez que é complementar do condicional no mesmo nível na linha 9.

Os condicionais é um conceito relativamente simples de interiorizar, muito provavelmente não terá dificuldade em identificar a necessidade de execuções alternativas. No entanto, na escrita do código é muito frequente cometer erros nas expressões lógicas, e ao ler pensa que a expressão tem um efeito quando na realidade tem outro. A utilização de expressões simples, e bons nomes nas variáveis, é o aconselhado para que o código tenha uma boa leitura, e consequentemente tenha menos erros.

4. CICLOS

Um programa serve para executar um conjunto de instruções. No entanto, com o que aprendemos até agora, cada linha de código é executada quanto muito uma vez. Isto limita muito a utilidade de programas.

Consideremos que se pretende calcular a soma dos primeiros 4 quadrados, não conhecendo nenhuma expressão matemática para obter o valor directamente. O programa seguinte resolveria o problema:

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int soma=0;
6     soma=soma+1*1;
7     soma=soma+2*2;
8     soma=soma+3*3;
9     soma=soma+4*4;
10    printf("Soma dos primeiros 4 quadrados: %d", soma);
11 }
```

Programa 4-1 Soma dos primeiros 4 quadrados naturais

Execução do programa:

```
C:\>somaquadrados
Soma dos primeiros 4 quadrados: 30
```

Execução passo-a-passo não apresenta novidades.

Passo	Linha	Instrução	Resultado	
1	5	int soma=0;		soma=0
2	6	soma=soma+1*1;		soma=0+1*1=1
3	7	soma=soma+2*2;		soma=1+2*2=5
4	8	soma=soma+3*3;		soma=5+3*3=14
5	9	soma=soma+4*4;		soma=14+4*4=30
6	10	printf(...);	Soma dos primeiros 4 quadrados: 30	soma=30

Problemas com este programa:

- Se pretender obter a soma dos primeiros 9 quadrados, temos de refazer o programa. É sempre mais interessante ter um programa que pode ser utilizado numa maior gama de situações, que um programa que serve apenas para uma situação específica.
- O código repetido pode até ser facilmente copiável, mas se for necessário alterar algum pormenor, essa alteração tem de ser reproduzida em todas as cópias realizadas.
- A variável soma é repetida antes e depois da atribuição. A repetição da mesma entidade ou bloco de código "pesa" não só ao editar e corrigir algum ponto, como também na leitura.
- O tamanho do programa cresce se pretender um cálculo com um valor maior.

Vale a pena ter um programa assim? É claro que sim, fazer as contas à mão é que não. No entanto se não existisse alternativa, um computador teria uma utilidade pouco mais do que se obtém com uma calculadora. A solução passa por utilizar um ciclo, em que no C o mais simples é o ciclo **while**. É muito parecido com o condicional **if**, mas ao contrário do **if** o código é repetido enquanto a expressão lógica for verdadeira.

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int soma=0, i=1;
6     while(i<=4)
7     {
8         soma+=i*i;
9         i++;
10    }
11    printf("Soma dos primeiros 4 quadrados: %d", soma);
12 }

```

Programa 4-2 Soma dos primeiros 4 quadrados naturais utilizando um ciclo

Comentários:

- O ciclo while avalia a expressão lógica, $i \leq 4$, e enquanto esta for verdadeira soma o quadrado de i e incrementa o valor de i .
- Utilizou-se o operador $+=$, que em vez de efectuar uma atribuição soma à variável à esquerda o valor da expressão à direita. Existem também os operadores equivalentes $-$, $*=$, $/=$, $\%=$.
- Para incrementar o valor de i , existe ainda um operador especial $++$, já que se pretende somar apenas o valor 1. Existe também o operador inverso $--$ que subtrai uma unidade.
- Neste programa, se for necessário troca-se facilmente a constante 4 por uma variável que é introduzida pelo utilizador, ficando o programa mais genérico. O número de instruções não depende agora do argumento pretendido.

A execução passo-a-passo é um pouco mais longa. É importante que veja com atenção esta execução passo-a-passo, para compreender o funcionamento do ciclo. Repare que **cada linha no ciclo foi executada várias vezes**, apenas os valores que as variáveis tinham quando essas linhas foram executadas é que eram diferentes.

Passo	Linha	Instrução	Resultado		
1	5	int soma=0, i=1;		soma=0	i=1
2	6	while(i<=4)	1<=4 Verdade	soma=0	i=1
3	8	soma+=i*i;		soma+=1*1=1	i=1
4	9	i++;		soma=1	i=2
5	6	while(i<=4)	2<=4 Verdade	soma=1	i=2
6	8	soma+=i*i;		soma+=2*2=5	i=2
7	9	i++;		soma=5	i=3
8	6	while(i<=4)	3<=4 Verdade	soma=5	i=3
9	8	soma+=i*i;		soma+=3*3=14	i=3
10	9	i++;		soma=14	i=4
11	6	while(i<=4)	4<=4 Verdade	soma=14	i=4
12	8	soma+=i*i;		soma+=4*4=30	i=4
13	9	i++;		soma=30	i=5
14	6	while(i<=4)	5<=4 Falso	soma=30	i=5
15	11	printf(...);	Soma dos primeiros 4 quadrados: 30	soma=30	i=5

Como é que o ciclo foi construído? Tínhamos um conjunto de operações seguidas para realizar, e criou-se uma variável auxiliar para contar o número de vezes que o ciclo foi executado, e também para controlar a paragem do ciclo. Pretendíamos 4 passos, colocou-se a condição de paragem no 4 e incrementou-se em cada ciclo o valor da variável.

Este tipo de variável é praticamente forçosa em todos os ciclos, tendo merecido a distinção de se chamar uma **variável iteradora**. É habitual utilizar-se os nomes **i** e **j** para variáveis iteradoras. Vamos agora apresentar um outro exemplo.

Pretende-se saber o número associado aos caracteres de 'a' a 'z'.

```

1 #include <stdio.h>
2
3 int main()
4 {
5     char c='a';
6     while(c<='z')
7     {
8         printf("\nLetra %c = %d",c,c);
9         c++;
10    }
11 }

```

Programa 4-3 Imprime o código numérico correspondente a cada letra

Comentários:

- Neste caso temos a variável iteradora como um carácter (tipo char). Como o tipo char é também um inteiro pequeno, não há problema em utilizar desigualdades com caracteres, incluindo `c<='z'`. As letras têm o código numérico por ordem.
- No `printf`, após as aspas temos `\n`. A `\` nas strings significa que se segue um carácter especial, neste a mudança de linha. O `\n` é traduzido no código final para um valor que é interpretado no sistema operativo como uma instrução para reposicionar o cursor na linha seguinte. Desta forma as letras são colocadas em linhas distintas. O `printf` utiliza o que já vimos para a letra 'a', colocando a variável char no `printf` tanto em formato de carácter como em formato numérico.

Execução do programa:

C:\>*caracteres*

```

Letra a = 97
Letra b = 98
Letra c = 99
Letra d = 100
Letra e = 101
Letra f = 102
Letra g = 103
Letra h = 104
...
Letra w = 119
Letra x = 120
Letra y = 121
Letra z = 122

```

Apresenta-se a execução passo-a-passo com os primeiros 7 passos. Estes números correspondem ao código associado a cada letra, que irá ser explicado quando se falar de vectores e strings.

Vamos agora dar um exemplo mais complexo. Pretende-se saber para um determinado número inteiro K, quantos pares de números inteiros (A,B) existem que verifiquem as seguintes condições: $A+B \leq K$, $A*B \leq K$, $A \geq 1$ e $B \geq 1$.

Passo	Linha	Instrução	Resultado	
1	5	char c='a';		c='a' (97)
2	6	while(c<='z')	'a'<='z' Verdade	c='a' (97)
3	8	printf(...);	Letra a = 97	c='a' (97)
4	9	c++;		c='b' (98)
5	6	while(c<='z')	'b'<='z' Verdade	c='b' (98)
6	8	printf(...);	Letra b = 98	c='b' (98)
7	9	c++;		c='c' (99)
...				

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int A,B,K,contagem;
6     printf("Indique K: ");
7     scanf("%d",&K);
8     contagem=0;
9     A=1;
10    while(A<=K)
11    {
12        B=1;
13        while(B<=K)
14        {
15            if(A+B<=K && A*B<=K)
16                contagem++;
17            B++;
18        }
19        A++;
20    }
21    printf("Total: %d",contagem);
22 }

```

Programa 4-4 Cálculo do número de pares de inteiros que respeitam uma determinada condição

Comentários:

- O programa necessita iterar em A e em B. Como A e B têm de ser maior ou igual a 1, e nenhum pode ser maior que K, cada ciclo tem de ser feito entre 1 e K.
- A expressão lógica é colocada conforme a definição. A expressão lógica do condicional poderia ser utilizada directamente no segundo ciclo. No entanto nesse caso seria conveniente um bom comentário a explicar a mudança, dado que embora mais eficiente o código fica mais difícil de compreender.
- Com um ciclo dentro do outro, as variáveis A e B a começarem em 1 e a acabarem em K, todos os pares (A,B) foram verificados. Repare que a inicialização de B=1; está dentro do ciclo da variável A. Cada variável (A e B) são inicializadas antes do ciclo, utilizadas no condicional do ciclo, e actualizadas (incrementadas) no final do seu ciclo.

Execução do programa:

```

C:\>somamu1
Indique K: 10
Total: 25

```

Vamos ver a execução passo-a-passo para K=3.

Este é um exemplo já bastante longo, mas é importante que o veja com algum cuidado, dado que é o primeiro **ciclo dentro de outro ciclo**. Escolha um passo aleatoriamente, e verifique se consegue construir o passo seguinte. Se compreender esta execução passo-a-passo, muito provavelmente não terá problemas em construir ciclos.

Os ciclos são fáceis de identificar, tal

Passo	Linha	Instrução	Resultado	A=?	B=?	K=?	contagem=?
1	5	int A,B,K,contagem;		A=?	B=?	K=?	contagem=?
2	6	printf(...);	Indique K:	A=?	B=?	K=?	contagem=?
3	7	scanf(...,&K);		3	A=?	B=?	K=3 contagem=?
4	8	contagem=0;		A=?	B=?	K=3	contagem=0
5	9	A=1;		A=1	B=?	K=3	contagem=0
6	10	while(A<=K)	1<=3 Verdade	A=1	B=?	K=3	contagem=0
7	12	B=1;		A=1	B=1	K=3	contagem=0
8	13	while(B<=K)	1<=3 Verdade	A=1	B=1	K=3	contagem=0
9	15	if(A+B<=K && A*B<=K)	1+1<=3 && 1*1<=3 Verdade	A=1	B=1	K=3	contagem=0
10	16	contagem++;		A=1	B=1	K=3	contagem=1
11	17	B++;		A=1	B=2	K=3	contagem=1
12	13	while(B<=K)	2<=3 Verdade	A=1	B=2	K=3	contagem=1
13	15	if(A+B<=K && A*B<=K)	1+2<=3 && 1*2<=3 Verdade	A=1	B=2	K=3	contagem=1
14	16	contagem++;		A=1	B=2	K=3	contagem=2
15	17	B++;		A=1	B=3	K=3	contagem=2
16	13	while(B<=K)	3<=3 Verdade	A=1	B=3	K=3	contagem=2
17	15	if(A+B<=K && A*B<=K)	1+3<=3 && (ignorado) Falso	A=1	B=3	K=3	contagem=2
18	17	B++;		A=1	B=4	K=3	contagem=2
19	13	while(B<=K)	4<=3 Falso	A=1	B=4	K=3	contagem=2
20	19	A++;		A=2	B=4	K=3	contagem=2

como os condicionais, mas quando é necessário um ciclo dentro do outro, ou mesmo ainda um terceiro ciclo, a compreensão do código baixa dado que é necessário em cada ciclo considerar os ciclos sobre o qual está a correr. Cada ciclo tem uma expressão lógica, que por si só pode ser complexa tal como nos condicionais, mas como tem uma variável iteradora para fazer variar a expressão lógica, leva a que deva ser dada a máxima atenção nos ciclos.

Erros comuns mais directamente associados a este capítulo:

- **Utilização do “goto”**

- **Forma:** Numa dada instrução, sabe-se a condição pretendida para voltar para cima ou saltar para baixo. É suficiente colocar um label no local

desejado, e colocar o goto dentro do condicional. Este tipo de instrução pode ser motivado pelo uso de fluxogramas

- **Problema:** Perde-se nada mais nada menos que a estrutura das instruções. A utilização ou não desta instrução, é que define se a linguagem é estruturada, ou não estruturada (por exemplo o Assembly). Se de uma linha de código se poder saltar para qualquer outra linha de código, ao analisar/escrever cada linha de código, tem que se considerar não apenas as linhas que a antecedem dentro do bloco actual, como todas as linhas de código, dado que de qualquer parte do programa pode haver um salto para esse local. Para compreender 1 linha de código, é necessário considerar todas as instruções no programa. A complexidade do programa nesta situação cresce de forma quadrática com o número de linhas.

- **Resolução:** Utilizar as estruturas de ciclos disponíveis, bem como condicionais, e funções⁵. Se o salto é para trás dentro da mesma função é certamente um ciclo o que é pretendido, mas se é para a frente, provavelmente é um condicional. Se é um salto para uma zona muito distante, então é provavelmente uma função que falta. No caso de utilizar fluxogramas, deve considerar primeiro utilizar o seu tempo de arranque vendo execuções passo-a-passo de forma a compreender o que é realmente um programa, sem segredos, e resolver exercícios

- **Instruções parecidas seguidas**

- **Forma:** Quando é necessário uma instrução que é parecida com a anterior, basta seleccioná-la e fazer uso de uma das principais vantagens dos documentos digitais: copy/paste

Passo	Linha	Instrução	Resultado				
21	10	while(A<=K)	2<=3 Verdade	A=2	B=4	K=3	contagem=2
22	12	B=1;		A=2	B=1	K=3	contagem=2
23	13	while(B<=K)	1<=3 Verdade	A=2	B=1	K=3	contagem=2
24	15	if(A+B<=K && A*B<=K)	2+1<=3 && 2*1<=3 Verdade	A=2	B=1	K=3	contagem=2
25	16	contagem++;		A=2	B=1	K=3	contagem=3
26	17	B++;		A=2	B=2	K=3	contagem=3
27	13	while(B<=K)	2<=3 Verdade	A=2	B=2	K=3	contagem=3
28	15	if(A+B<=K && A*B<=K)	2+2<=3 && (Ignorado) Falso	A=2	B=2	K=3	contagem=3
29	17	B++;		A=2	B=3	K=3	contagem=3
30	13	while(B<=K)	3<=3 Verdade	A=2	B=3	K=3	contagem=3
31	15	if(A+B<=K && A*B<=K)	2+3<=3 && (Ignorado) Falso	A=2	B=3	K=3	contagem=3
32	17	B++;		A=2	B=4	K=3	contagem=3
33	13	while(B<=K)	4<=3 Falso	A=2	B=4	K=3	contagem=3
34	19	A++;		A=3	B=4	K=3	contagem=3
35	10	while(A<=K)	3<=3 Verdade	A=3	B=4	K=3	contagem=3
36	12	B=1;		A=3	B=1	K=3	contagem=3
37	13	while(B<=K)	1<=3 Verdade	A=3	B=1	K=3	contagem=3
38	15	if(A+B<=K && A*B<=K)	3+1<=3 && (Ignorado) Falso	A=3	B=1	K=3	contagem=3
39	17	B++;		A=3	B=2	K=3	contagem=3
40	13	while(B<=K)	2<=3 Verdade	A=3	B=2	K=3	contagem=3

Passo	Linha	Instrução	Resultado				
41	15	if(A+B<=K && A*B<=K)	3+2<=3 && (Ignorado) Falso	A=3	B=2	K=3	contagem=3
42	17	B++;		A=3	B=3	K=3	contagem=3
43	13	while(B<=K)	3<=3 Verdade	A=3	B=3	K=3	contagem=3
44	15	if(A+B<=K && A*B<=K)	3+3<=3 && (Ignorado) Falso	A=3	B=3	K=3	contagem=3
45	17	B++;		A=3	B=4	K=3	contagem=3
46	13	while(B<=K)	4<=3 Falso	A=3	B=4	K=3	contagem=3
47	19	A++;		A=4	B=4	K=3	contagem=3
48	10	while(A<=K)	4<=3 Falso	A=4	B=4	K=3	contagem=3
49	21	printf(..., contagem);	Total: 3	A=4	B=4	K=3	contagem=3

⁵ O conceito de função é introduzido no capítulo seguinte

- **Problema:** O código ficará muito pesado de ler, podendo também prejudicar a escrita. É sinal que está a falhar um ciclo, em que as pequenas mudanças entre instruções, em vez de serem editadas e alteradas, essa alteração é colocada dependente da variável iteradora, fazendo numa só instrução a chamada a todas as instruções parecidas, debaixo de um ciclo. O código não só fica mais simples de ler, como se for necessária outra instrução basta alterar a expressão lógica de paragem do ciclo, em vez de um copy/paste e edição das alterações
- **Resolução:** Se encontra situações destas no seu código, estude alternativas para utilizar ciclos

O segundo erro foi cometido no Programa 4-1, de forma a motivar os ciclos. O primeiro erro está cometido no programa seguinte, que será também o único programa neste texto a utilizar a instrução goto.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int soma=0, i=1;
6 loop:
7     soma+=i*i;
8     i++;
9     if(i<=4)
10         goto loop;
11     printf("Soma dos primeiros 4 quadrados: %d", soma);
12 }
```

Tem que existir um “label” na linha 6 de forma a utilizá-lo na instrução goto da linha 10. Os “labels” são identificadores a terminarem com dois pontos. Código deste tipo nem sequer pode ter uma indentação com algum significado, já que as instruções nas linhas 7 e 8 não podem estar indentadas, no entanto estão sobre um ciclo feito com if/goto.

É imperativo que realize agora actividades formativas para consolidar os conceitos introduzidos. Nesta altura está em condições de fazer todos os exercícios, no entanto irá sentir dificuldades nos azuis e vermelhos, o que é normal, dado que fazem uso mais abusivamente dos conceitos introduzidos, que necessitam tempo de maturação e consolidação.

1) Exercícios

Notas gerais a ter em atenção em todos os exercícios:

- Escreva sem acentos;
- Utilize para as variáveis inteiras, o tipo `int`, confirmando que tem 4 bytes no exercício `olamundosizeof.c`, caso contrário utilize o tipo equivalente;
- Utilize para as variáveis reais a precisão dupla, o tipo `double`;
- Durante o desenvolvimento do programa, imprima resultados parciais, de forma a garantir o que o programa faz, eventualmente comentando esse código na versão final;
- O separador decimal da linguagem C é o ponto final, pelo que será utilizado nos resultados dos exercícios (12.435 é correcto, 12,435 é incorrecto), mas na introdução dos dados e impressão dos resultados, poderá variar conforme as configurações no computador;
- Se não consegue resolver um exercício, pense numa variável auxiliar que lhe dê jeito;
- Existem dicas adicionais sobre alguns exercícios no anexo Exercícios: Dicas, Respostas e Resoluções, bem como forma de validar a resposta de um exercício sem ver uma resolução, e resoluções dos exercícios para comparar após resolvê-los.

olamundosizeof.c



Faça um programa que coloque **Olá Mundo!** em bom português (com acentos), e que indique o tamanho em bytes (operador `sizeof`) dos seguintes tipos de dados: `char`; `short`; `int`; `long`; `long long`; `float`; `double`; `long double`.

Notas:

- Se os caracteres não lhe aparecerem bem na linha de comando, antes de executar o programa execute o comando `C:\>chcp 1252`

Execução de exemplo:

```
C:\>chcp 1252
Active code page: 1252
C:\>olamundosizeof

Olá Mundo!
sizeof(char): xx
sizeof(short): xx
sizeof(int): xx
sizeof(long): xx
sizeof(long long): xx
sizeof(float): xx
sizeof(double): xx
sizeof(long double): xx
```

Pergunta: qual a soma dos tamanhos dos tipos pedidos compilado no tcc?

soma.c

Somar os primeiros N números inteiros, sendo N definido pelo utilizador:

$$\sum_{i=1}^N i$$

Notas:

- Escreva sem acentos neste e nos restantes exercícios, para que o código funcione sempre correctamente mesmo sem mudar o código de página para 1252.
- Durante o desenvolvimento do programa, imprima resultados parciais, de forma a garantir o que o programa faz, eventualmente comentando esse código na versão final, dado que esse texto poderá fazer sentido apenas para o programador e não para o utilizador.

Execução de exemplo:

```
C:\>soma
Calculo da soma dos primeiros N numeros.
Indique N: 10
```

```
adicionar 1, parcial 1
adicionar 2, parcial 3
adicionar 3, parcial 6
adicionar 4, parcial 10
adicionar 5, parcial 15
adicionar 6, parcial 21
adicionar 7, parcial 28
adicionar 8, parcial 36
adicionar 9, parcial 45
adicionar 10, parcial 55
Total: 55
```

Pergunta: qual a soma dos primeiros 21090 números inteiros?

hms.c

Faça um programa que leia as horas, minutos e segundos, e calcule o número de segundos que passaram desde o início do dia.

Notas:


- Não faça verificação da validade dos parâmetros de entrada

Execução de exemplo:

```
C:\>hms
Calculo do numero de segundos desde o inicio do dia.
Hora: 2
Minuto: 15
Segundos: 30
Numero de segundos desde o inicio do dia: 8130
```

Pergunta: qual a resposta no caso de se colocar a 25ª hora, o minuto 100 e o segundo 200?

produto.c

 Multiplicar os primeiros N números inteiros positivos (factorial de N), sendo N definido pelo utilizador:


$$N! = \prod_{i=1}^N i$$

Execução de exemplo:

```
C:\>produto
Calculo do produto dos primeiros N numeros.
Indique N: 5
Factorial(1)=1
Factorial(2)=2
Factorial(3)=6
Factorial(4)=24
Factorial(5)=120
Resultado: 120
```

Pergunta: qual é o factorial de 12?

arranjos.c

 Calculo dos arranjos de N, R a R: multiplicar os números de N-R+1 até N:

$$A(N, R) = \frac{N!}{(N-R)!} = \prod_{i=N-R+1}^N i$$

Notas:

- Atenção que R tem de ser menor que N
- Os arranjos de 3 elementos {A, B, C}, 2 a 2, são os seguintes 6: (A,B); (A,C); (B,A); (B,C); (C,A); (C,B).

Execução de exemplo:

```
C:\>arranjos
Calculo dos arranjos de N, R a R:
Indique N: 5
Indique R: 3
i=3; arranjos=3
i=4; arranjos=12
i=5; arranjos=60
Resultado: 60
```

Pergunta: qual é o resultado retornado dos arranjos de 20, 8 a 8?

somadigitos.c

Calcule a soma dos quadrados dos dígitos de um número introduzido pelo utilizador.

Notas:

- Pode obter o valor do dígito mais baixo, calculando o resto da divisão por 10.
- Mostre o resultado parcial, neste e nos restantes exercícios.

Execução de exemplo:

```
C:\>somadigitos
Calculo da soma do quadrado dos digitos de um numero:
Numero: 1234
  n=1234; soma=16
  n=123; soma=25
  n=12; soma=29
  n=1; soma=30
Resultado: 30
```

Pergunta: se o utilizador introduzir o número 856734789, qual é o resultado do programa?

fibonacci.c

Calcular o valor da função fibonacci⁶, para um dado argumento N. Para N=1 ou 2, deve retornar N, caso contrário retorna a soma dos dois valores anteriores:

$$F(n) = \begin{cases} n & , n \leq 2 \\ F(n-1) + F(n-2) & , n > 2 \end{cases}$$

Notas:

- Utilize duas variáveis auxiliares.

Execução de exemplo:

```
C:\>fibonacci
Calculo do valor da funcao Fibonacci:
Indique N: 6
  Fib(3)=3
  Fib(4)=5
  Fib(5)=8
  Fib(6)=13
Resultado: 13
```

Pergunta: qual o valor do programa para N=40?

⁶ definição da função Fibonacci não padrão

combinacoes.c

Calcule as combinações de N, R a R. A fórmula é idêntica à dos arranjos, multiplicar de N-R+1 até N, mas tem de se dividir pelo factorial de R:

$$C(N, R) = \frac{N!}{(N-R)!R!} = \prod_{i=N-R+1}^N i / \prod_{i=1}^R i$$

Notas:

- Se efectuar as multiplicações e só dividir no final, rapidamente ultrapassa o limite do inteiro. Se multiplicar e depois dividir em cada passo, consegue obter valores correctos para uma maior gama de números.
- As combinações de 3 elementos {A, B, C}, 2 a 2, são as seguintes 3: {A,B}; {A,C}; {B,C}. Notar que relativamente aos arranjos, neste caso a ordem não interessa.

Execução de exemplo:

```
C:\>combinacoes
Cálculo das combinacoes de N, R a R:
Indique N: 5
Indique R: 3
1*3=3/1=3
3*4=12/2=6
6*5=30/3=10
Resultado: 10
```

Pergunta: qual o número de combinações de 26, 13 a 13?

euler.c

Calcular o número de Euler e , através da utilização da série de Taylor para e^x quando $x = 1$ (soma do inverso dos factoriais):

$$e = \sum_{n=0}^K \frac{1}{n!}$$

Notas:

- Considere o factorial de zero como sendo 1
- Utilize a precisão dupla para valores reais, neste e em outros exercícios
- Na função `printf` pode imprimir um número real em notação científica, e especificando a precisão a 16 dígitos utilizando a string de formatação: `%.16g`

Execução de exemplo:

```
C:\>euler
0: 1
1: 2
...
19: 2.7183
20: 2.7183
Resultado: 2.71828182845xxxx
```

Pergunta: qual é o valor com precisão 16, em notação científica, da execução da fórmula acima com K=20?

trococ.c

Faça um programa que receba um montante em euros (com cêntimos), e que determina o menor número de moedas de cada tipo necessário para perfazer esse montante. Pode utilizar moedas de euros de todos os valores disponíveis (2€, 1€, ...).

Notas:

- Deve efectuar os arredondamentos para a unidade mais próxima, no caso de o utilizador introduzir um valor com precisão abaixo do cêntimo.

Execução de exemplo:

```
C:\>trococ
Introduza um montante em euros, podendo ter centimos: 1.79
1 euro: 1
50 centimos: 1
20 centimos: 1
5 centimos: 1
2 centimos: 2
```

Pergunta: para devolver 19.99 euros, quantas moedas são retornadas?

primo.c

Faça um programa que verifica se um determinado número N é um número primo. Um número é primo se é divisível apenas por ele próprio e pela unidade. Se não for primo deve identificar o menor número pelo qual é divisível.

Notas:

- É suficiente testar até à raiz quadrada de N (pode utilizar a função `sqrt` da biblioteca `math.h`)
- Em vez de calcular a raiz quadrada de N, pode calcular o quadrado do divisor.

Execução de exemplo:

```
C:\>primo
Funcao que verifica se um numero N e' primo:
Indique N: 99
2
Numero divisível por 3
```

```
C:\>primo
Funcao que verifica se um numero N e' primo:
Indique N: 97
2 3 4 5 6 7 8 9
Numero primo!
```

Pergunta: Some os números retornados dos seguintes números (no caso dos números primos não some nada): 241134319; 241234319; 13212311.

triplasoma.c

Dado um inteiro positivo N, escrever todas as decomposições distintas possíveis como soma de três inteiros positivos (considerar iguais as triplas com os mesmos valores mas por outra ordem). Calcular também o número de somas distintas.

Notas:

- Assumir que os números mais altos aparecem sempre primeiro.

Execução de exemplo:

```
C:\>triplasoma
Escreva um numero para decompor em somas de tres parcelas.
Numero: 7
5+1+1
4+2+1
3+3+1
3+2+2
Numero de somas: 4
```

Pergunta: quantas somas existem para N=1000?

pi.c

Calcular o valor de π com base na fórmula de Ramanujan:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)! \times (1103 + 26390k)}{(k!)^4 \times 396^{4k}}$$

Notas:

- Utilize precisão dupla
- Pode utilizar a função `sqrt` para calcular a raiz quadrada, da biblioteca `math.h`


Execução de exemplo:

```
C:\>pi
```

```
valor de PI (x iteracoes): 3.14159265xxxxxxxx
```

Pergunta: qual o valor de π com precisão 17 em notação científica, para K=2?

formularesolvente.c

 Faça um programa que peça os coeficientes de um polinómio do segundo grau, e retorna as raízes reais, caso existam. Adicionalmente o programa deve retornar todos os conjuntos de coeficientes inteiros, que têm apenas raízes inteiras reais. Os coeficientes estão entre -K e K não tendo nenhum coeficiente nulo (K é introduzido pelo utilizador e é um inteiro pequeno). Relembra-se a fórmula resolvente:

$$ax^2 + bx + c = 0 \Leftrightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Notas:

- Para calcular a raiz quadrada, utilize a função `sqrt`, disponível na biblioteca `math.h`
- Faça três ciclos um dentro do outro, um ciclo por cada um dos coeficientes, e varie a variável iteradora entre -K e K.

Execução de exemplo:

```
C:\>formularesolvente
```

```
Equacao do segundo grau a*x^2+b*x+c=0.
```

```
Indique a b c: 2 4 2
```

```
Delta: 0.000000
```

```
A equacao tem uma raiz unica, x=-1.000000
```

```
Calculo de coeficientes entre -K e K inteiros nao nulos, com raízes inteiras.
```

```
Introduza K:2
```

```
Coeficientes de -2 a 2 inteiros nao nulos, com raízes inteiras:
```

```
[-1 -2 -1] [-1 -1 2] [-1 1 2] [-1 2 -1] [1 -2 1] [1 -1 -2] [1 1 -2] [1 2 1]
```

```
Total: 8
```

Pergunta: quantas equações do segundo grau existem, com coeficientes entre -10 e 10 inteiros não nulos, e com raízes inteiras?

PARTE II – FUNÇÕES, VECTORES E RECURSÃO

Apenas com a Parte I é possível fazer programas, mas não há nenhuma ferramenta de controlo de complexidade. O código à medida que cresce requer cada vez mais atenção para o escrever e manter. Na Parte II vamos dar uma ferramenta da programação que permite controlar a complexidade de um programa: a abstracção funcional. O bom uso desta ferramenta leva a que a nossa atenção apenas tenha de estar a cada momento centrada num reduzido conjunto de linhas de código, e mesmo assim garante que toda a funcionalidade do programa é satisfeita. Desta forma é possível escrever programas de qualquer dimensão. Serão apresentados também os vectores, de forma a poder lidar com grande volume de variáveis, bem como alternativas mais elegantes para ciclos e condicionais. Após a realização da Parte II, ficará com bases sólidas para escrever programas de qualquer dimensão, sem grandes limitações.

5. FUNÇÕES

O termo "**função**" em português é muito lato, aplicável a diversas entidades na linguagem C sejam instruções sejam variáveis. Tudo tem uma função, caso contrário não estaria na linguagem. As atribuições têm a função de atribuir o valor de uma expressão a uma variável, os condicionais têm a função de executar uma instrução mediante o resultado de uma expressão lógica, os ciclos têm a função de executar um conjunto de instruções enquanto uma expressão lógica se mantiver verdadeira, e as variáveis têm a função de manter um valor de forma a este ser utilizado em expressões, ou ser trocado por outro através de uma atribuição.

Na linguagem C o termo função tem no entanto um sentido muito concreto, e deve o seu nome precisamente devido à importância que tem.

Uma **função é um bloco de código que pode ser chamado de qualquer parte do programa**, quantas vezes se quiser. Por chamar, significa que se tem uma instrução a ordenar o computador a executar o bloco de código correspondente à função identificada, e só depois continuar com a execução das restantes linhas de código.

Vamos clarificar este ponto com o seguinte exemplo, ainda sem utilizar funções. Pretende-se um programa que oferece um menu ao utilizador de 3 opções, e uma opção de saída do programa.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int opcao;
6     /* mostrar as opções do menu */
7     printf("\nMenu:\n1 - opcao A\n2 - opcao B\n3 - opcao C\n0 - sair");
8     printf("\nOpcao: ");
9     scanf("%d",&opcao);
10
11     while(opcao>0)
12     {
13         /* ver qual é a opção */
14         if(opcao==1)
15             printf("Opcao escolhida A");
16         else if(opcao==2)
17             printf("Opcao escolhida B");
18         else if(opcao==3)
19             printf("Opcao escolhida C");
20         else
21             printf("Opcao invalida");
22
23         /* mostrar as opções do menu */
24         printf("\nMenu:\n1 - opcao A\n2 - opcao B\n3 - opcao C\n0 - sair");
25         printf("\nOpcao: ");
26         scanf("%d",&opcao);
27     }
28     printf("Fim do programa.");
29 }
```

Programa 5-1 Escolha de opções num menu

Este programa utiliza apenas conceitos já introduzidos. Nada de novo.

Execução do programa:

C:\>*menu*

Menu:

1 - opcao A

2 - opcao B

3 - opcao C

0 - sair

Opcao: **2**

Opcao escolhida B

Menu:

1 - opcao A

2 - opcao B

3 - opcao C

0 - sair

Opcao: **5**

Opcao invalida

Menu:

1 - opcao A

2 - opcao B

3 - opcao C

0 - sair

Opcao: **0**

Fim do programa.

Passo	Linha	Instrução	Resultado	
1	5	int opcao;		opcao=?
2	7	printf(...);	Menu:	opcao=?
3	8	printf(...);	Opcao:	opcao=?
4	9	scanf(...,&opcao);		2 opcao=2
5	11	while(opcao>0)	2>0 Verdade	opcao=2
6	14	if(opcao==1)	2==1 Falso	opcao=2
7	16	if(opcao==2)	2==2 Verdade	opcao=2
8	17	printf(...);	Opcao escolhida B	opcao=2
9	24	printf(...);	Menu:	opcao=2
10	25	printf(...);	Opcao:	opcao=2
11	26	scanf(...,&opcao);		5 opcao=5
12	11	while(opcao>0)	5>0 Verdade	opcao=5
13	14	if(opcao==1)	5==1 Falso	opcao=5
14	16	if(opcao==2)	5==2 Falso	opcao=5
15	18	if(opcao==3)	5==3 Falso	opcao=5
16	21	printf(...);	Opcao invalida	opcao=5
17	24	printf(...);	Menu:	opcao=5
18	25	printf(...);	Opcao:	opcao=5
19	26	scanf(...,&opcao);		0 opcao=0
20	11	while(opcao>0)	0>0 Falso	opcao=0
21	28	printf(...);	Fim do programa.	opcao=0

O programa funcionou de acordo com o esperado. No entanto o código correspondente ao menu está repetido tanto no início do programa, como no ciclo. Isto porque para entrar no ciclo é necessário que o menu já tenha sido mostrado, e o utilizador já tenha introduzido uma opção.

O código repetido não teve grande trabalho a ser escrito, foi apenas uma operação de **Copy/Paste**. Se fosse necessário em mais partes do programa, fazia-se mais cópias desse código. Esta operação tão vulgar em aplicações informáticas, é no entanto proibida em programação. Porquê?

- Para fazer uma alteração numa parte do código duplicada, tem que se conhecer as partes para a qual o código foi copiado para reproduzir a alteração nessas partes;
- Se o código duplicado sofrer pequenas alterações não aplicáveis ao código original, torna-se complicado manter ambos os códigos, dado que algumas das futuras correcções não serão aplicáveis a ambos os códigos;
- O custo de leitura poderá ser até maior que o custo de escrita, o que dificulta a manutenção do código;
- Um bloco de código pode funcionar bem numa zona, mas após ser copiado para outra zona, devido às instruções precedentes ou sucessoras, pode não ter o funcionamento esperado.

Qual a solução? Fazer uma função, ou seja, um bloco de código com as instruções correspondente ao menu, e chamar essa função nos dois locais onde o menu é necessário.

```
1 #include <stdio.h>
2
3 int Menu()
4 {
5     int opcao;
6     /* mostrar as opções do menu */
7     printf("\nMenu:\n1 - opcao A\n2 - opcao B\n3 - opcao C\n0 - sair");
8     printf("\nOpcao: ");
9     scanf("%d",&opcao);
10    /* retornar a opção seleccionada */
11    return opcao;
12 }
13
14 int main()
15 {
16     int opcao;
17
18     opcao=Menu();
19
20     while(opcao>0)
21     {
22         /* ver qual é a opção */
23         if(opcao==1)
24             printf("Opcao escolhida A");
25         else if(opcao==2)
26             printf("Opcao escolhida B");
27         else if(opcao==3)
28             printf("Opcao escolhida C");
29         else
30             printf("Opcao invalida");
31
32         opcao=Menu();
33     }
34     printf("Fim do programa.");
35 }
```

Programa 5-2 Escolha de opções num menu, utilizando uma função

Dentro da função `main`, chama-se a função `Menu()`, e atribui-se o valor retornado à variável `opcao` que já era utilizada para esta função. Este procedimento é repetido nos dois locais onde é necessário o menu. Os parêntesis curvos após `Menu`, significam que o identificador é uma função.

O código correspondente à função está definido em cima, e é idêntico ao formato da função `main`, mas com o nome `Menu`. Este nome foi dado de forma a ficar o mais perto possível relativamente à funcionalidade que implementa. Dentro da função `Menu` não existe a variável `opcao`, que está declarada na função `main`, pelo que é necessário declarar também aqui a variável `opcao` para a poder utilizar. São variáveis distintas, embora tenham o mesmo nome, uma está na função `main` e outra na função `Menu`. A função `Menu` na última instrução “`return opcao;`” retorna o valor da variável `opcao`, sendo esse valor utilizado no local onde a função foi chamada.

Na execução passo-a-passo, a chamada à função `Menu` leva a uma criação de uma variável, que é o nome da função chamada, sendo as variáveis criadas dentro da função à direita, sendo destruídas quando a função retorna. Verifique que no passo 6 a variável `opcao` da função `Menu` ficou com o valor atribuído, mas apenas no passo 8 é que este valor passou para a variável `opcao` da função `main`.

A execução de ambos os programas é igual, mas os problemas apontados já não estão presentes no segundo programa:

- Para fazer uma alteração, basta editar o código da função e a alteração fica válida para todos os locais em que a função é utilizada;
- Se for necessário utilizar a função Menu com ligeiras diferenças, pode-se utilizar argumentos e mantém-se o código num só local.
- O custo de leitura é mais baixo, já que não há linhas repetidas;
- O código da função Menu funciona independentemente do local onde é chamado, já que apenas depende das instruções que estão na função. Se fosse copiado, necessitava de estar declarada uma variável `opcao` do tipo `int`, que não estivesse a ser utilizada para outros fins.

A principal vantagem das funções não é no entanto evitar os problemas acima descritos inerentes ao Copy/Paste, mas sim possibilitar a **abstracção funcional**. Não há nenhuma outra ferramenta na informática, mais poderosa que a abstracção funcional.

O termo "**abstracção**" em português significa que nos podemos abstrair de parte do problema. Um problema dividido fica mais simples, num caso extremo fica-se com problemas muito pequenos que são facilmente implementados. É precisamente esse caso extremo que se pretende, uma vez que não só a capacidade do programador é limitada, como quanto mais simples um programa estiver escrito, mais facilmente é lido e menos erros terá.

Na programação aplica-se a abstracção às funções, começando pela função `main`, que deve implementar o problema completo. Esta pode ir sendo desagregada em funções mais simples, até se obter dimensões de funções razoáveis, facilmente implementáveis e de leitura simples.

Ao implementar uma função só interessa saber **o que a função tem de fazer**. O resto do código não tem qualquer relevância, podemos abstrair-nos dele, nem sequer nos interessa saber onde a função será chamada.

Ao utilizar uma função só interessa saber **o que a função faz**. Como a função está implementada não tem qualquer relevância, podemos abstrair-nos disso, nem sequer interessa saber se a função utiliza condicionais ou ciclos, se tem muitas ou poucas linhas de código.

Desta forma, mesmo um **programador muito limitado** poderá implementar um **problema muito complexo**, aparentemente fora do alcance de alguns. Na verdade, por vezes acontece que um **programador sem "restrições"**, ao fazer as opções na escolha das funções a utilizar adaptadas para a sua maior capacidade de programação, arrisca-se a ficar com funções com grande

Passo	Linha	Instrução	Resultado			
1	16	<code>int opcao;</code>		<code>opcao=?</code>		
2	18	<code>opcao=Menu();</code>		<code>opcao=?</code>	Menu	
3	5	<code>int opcao;</code>		<code>opcao=?</code>	Menu	<code>opcao=?</code>
4	7	<code>printf(...);</code>	Menu:	<code>opcao=?</code>	Menu	<code>opcao=?</code>
5	8	<code>printf(...);</code>	Opcao:	<code>opcao=?</code>	Menu	<code>opcao=?</code>
6	9	<code>scanf(...,&opcao);</code>		2 <code>opcao=?</code>	Menu	<code>opcao=2</code>
7	11	<code>return opcao;</code>		<code>opcao=?</code>	Menu=2	<code>opcao=2</code>
8	18	<code>opcao=Menu();</code>		<code>opcao=2</code>		
9	20	<code>while(opcao>0)</code>	2>0 Verdade	<code>opcao=2</code>		
10	23	<code>if(opcao==1)</code>	2==1 Falso	<code>opcao=2</code>		
11	25	<code>if(opcao==2)</code>	2==2 Verdade	<code>opcao=2</code>		
12	26	<code>printf(...);</code>	Opcao escolhida 2	<code>opcao=2</code>		
13	32	<code>opcao=Menu();</code>		<code>opcao=2</code>	Menu	
14	5	<code>int opcao;</code>		<code>opcao=2</code>	Menu	<code>opcao=?</code>
15	7	<code>printf(...);</code>	Menu:	<code>opcao=2</code>	Menu	<code>opcao=?</code>
16	8	<code>printf(...);</code>	Opcao:	<code>opcao=2</code>	Menu	<code>opcao=?</code>
17	9	<code>scanf(...,&opcao);</code>		5 <code>opcao=2</code>	Menu	<code>opcao=5</code>
18	11	<code>return opcao;</code>		<code>opcao=2</code>	Menu=5	<code>opcao=5</code>
19	32	<code>opcao=Menu();</code>		<code>opcao=5</code>		
20	20	<code>while(opcao>0)</code>	5>0 Verdade	<code>opcao=5</code>		
21	23	<code>if(opcao==1)</code>	5==1 Falso	<code>opcao=5</code>		
22	25	<code>if(opcao==2)</code>	5==2 Falso	<code>opcao=5</code>		
23	27	<code>if(opcao==3)</code>	5==3 Falso	<code>opcao=5</code>		
24	30	<code>printf(...);</code>	Opcao invalida	<code>opcao=5</code>		
25	32	<code>opcao=Menu();</code>		<code>opcao=5</code>	Menu	
26	5	<code>int opcao;</code>		<code>opcao=5</code>	Menu	<code>opcao=?</code>
27	7	<code>printf(...);</code>	Menu:	<code>opcao=5</code>	Menu	<code>opcao=?</code>
28	8	<code>printf(...);</code>	Opcao:	<code>opcao=5</code>	Menu	<code>opcao=?</code>
29	9	<code>scanf(...,&opcao);</code>		0 <code>opcao=5</code>	Menu	<code>opcao=0</code>
30	11	<code>return opcao;</code>		<code>opcao=5</code>	Menu=0	<code>opcao=0</code>
31	32	<code>opcao=Menu();</code>		<code>opcao=0</code>		
32	20	<code>while(opcao>0)</code>	0>0 Falso	<code>opcao=0</code>		
33	34	<code>printf(...);</code>	Fim do programa.	<code>opcao=0</code>		

número de variáveis, condicionais e ciclos, levando muito mais tempo a implementar e testar e ficando o código de menor qualidade que o programador mais limitado. Este, ao não conseguir implementar funções acima da sua capacidade, procura sempre dividir o problema até que este tenha uma dimensão apropriada, pelo que encontrará sempre uma tarefa facilitada.

Vamos voltar ao programa que calcula se um ano é ou não bissexto, mas utilizando funções (ver Programa 3-2).

```

1 #include <stdio.h>
2
3 int Bissexto(int ano)
4 {
5     return ano%400==0 || ano%4==0 && ano%100!=0;
6 }
7
8 int main()
9 {
10     int ano;
11     printf("Indique ano: ");
12     scanf("%d", &ano);
13
14     /* teste de ano bissexto */
15     if(Bissexto(ano))
16         printf("Bissexto");
17     else
18         printf("Normal");
19 }

```

Programa 5-3 Determina se um ano é normal ou bissexto utilizando funções

Este programa manteve o funcionamento igual ao original, mas foi feita uma função **Bissexto** para obter o resultado da expressão lógica que indica se um ano é ou não bissexto. Esta função recebe no entanto um argumento, a variável ano. A função poderia ela própria pedir o valor do ano, e nesse caso não necessitava do argumento, mas assim não poderia ser utilizada numa parte do código onde não tivesse interesse pedir o ano. Com o argumento o código funciona de acordo com o valor recebido, podendo ser utilizado em maior número de situações.

Os argumentos de uma função são colocados entre os parênteses, tanto na declaração como na utilização. Se houver mais que um argumento, estes têm de ser separados por vírgulas, tal como já temos vindo a fazer no `printf` e no `scanf`. Na função **Bissexto** tem uma variável local ano, uma vez que os argumentos de uma função são também variáveis locais à função.

Passo	Linha	Instrução	Resultado			
1	10	int ano;		ano=?		
2	11	printf(...);	Indique ano:	ano=?		
3	12	scanf(...,&ano);		2345 ano=2345		
4	15	if(Bissexto(ano))		ano=2345 Bissexto		
5	3	int ano=2345;		ano=2345 Bissexto	ano=2345	
6	5	return ano%400==0 ...;	345==0 1==0 && (?) Falso	ano=2345 Bissexto=Falso	ano=2345	
7	15	if(Bissexto(ano))	Falso	ano=2345		
8	18	printf(...);	Normal	ano=2345		

Na execução passo-a-passo pode-se ver no passo 5 a declaração e atribuição do parâmetro da função. Os parâmetros das funções têm de ser declaradas como qualquer outra variável local.

Passo	Linha	Instrução	Resultado			
1	10	int ano;		ano=?		
2	11	printf(...);	Indique ano:	ano=?		
3	12	scanf(...,&ano);		2344 ano=2344		
4	15	if(Bissexto(ano))		ano=2344 Bissexto		
5	3	int ano=2344;		ano=2344 Bissexto	ano=2344	
6	5	return ano%400==0 ...;	344==0 0==0 && 44!=0 Verdade	ano=2344 Bissexto=Verdade	ano=2344	
7	15	if(Bissexto(ano))	Verdade	ano=2344		
8	16	printf(...);	Bissexto	ano=2344		

Neste caso pode-se considerar que não houve grande vantagem em fazer a função `Bissexto`, no entanto, atendendo a que há outro projecto dos dias do mês que necessita desta função, essa vantagem realmente existe, dado que desta forma é possível passar a função de um projecto para o outro sem qualquer alteração da função, nem ser necessário rever a implementação do código da função.

```

1  #include <stdio.h>
2
3  int Bissexto(int ano)
4  {
5      return ano%400==0 || ano%4==0 && ano%100!=0;
6  }
7
8  int DiasDoMes(int mes, int ano)
9  {
10     if(mes==2)
11     {
12         /* teste de ano bissexto */
13         if(Bissexto(ano))
14             return 29;
15         else
16             return 28;
17     } else if(mes==1 || mes==3 || mes==5 || mes==7 ||
18             mes==8 || mes==10 || mes==12)
19     {
20         return 31;
21     } else
22     {
23         return 30;
24     }
25 }
26
27 int main()
28 {
29     int ano, mes, dias;
30     printf("Indique ano: ");
31     scanf("%d", &ano);
32     printf("Indique mes: ");
33     scanf("%d", &mes);
34
35     printf("%d", DiasDoMes(ano, mes));
36 }

```

Programa 5-4 Determina o número de dias de um mês/ano utilizando funções

Este programa tem a mesma funcionalidade que o Programa 3-3.

Execuções passo-a-passo:

Passo	Linha	Instrução	Resultado							
1	29	int ano, mes;		ano=?	mes=?					
2	30	printf(...);	Indique ano:	ano=?	mes=?					
3	31	scanf(...,&ano);	2344	ano=2344	mes=?					
4	32	printf(...);	Indique mes:	ano=2344	mes=?					
5	33	scanf(...,&ano);	2	ano=2344	mes=2					
6	35	printf(...,DiasDoMes(mes,ano));		ano=2344	mes=2	DiasDoMes				
7	8	int mes=2, int ano=2344		ano=2344	mes=2	DiasDoMes	ano=2344	mes=2		
8	10	if(mes==2)	2==2 Verdade	ano=2344	mes=2	DiasDoMes	ano=2344	mes=2		
9	13	if(Bissexto(ano))		ano=2344	mes=2	DiasDoMes	ano=2344	mes=2	Bissexto	
10	3	int ano=2344;		ano=2344	mes=2	DiasDoMes	ano=2344	mes=2	Bissexto	ano=2344
11	5	return ano%400==0 ...;	344==0 0==0 && 44!=0 Verdade	ano=2344	mes=2	DiasDoMes	ano=2344	mes=2	Bissexto=Verdade	ano=2344
12	13	if(Bissexto(ano))	Verdade	ano=2344	mes=2	DiasDoMes	ano=2344	mes=2		
13	14	return 29;		ano=2344	mes=2	DiasDoMes=29	ano=2344	mes=2		
14	35	printf(...,29);	29	ano=2344	mes=2					

Passo	Linha	Instrução	Resultado					
1	29	int ano, mes;		ano=?	mes=?			
2	30	printf(...);	Indique ano:	ano=?	mes=?			
3	31	scanf(...,&ano);	2342	ano=2342	mes=?			
4	32	printf(...);	Indique mes:	ano=2342	mes=?			
5	33	scanf(...,&ano);	4	ano=2342	mes=4			
6	35	printf(...,DiasDoMes(mes,ano));		ano=2342	mes=4	DiasDoMes		
7	8	int mes=4, int ano=2342		ano=2342	mes=4	DiasDoMes	ano=2342	mes=4
8	10	if(mes==2)	4==2 Falso	ano=2342	mes=4	DiasDoMes	ano=2342	mes=4
9	17	if(mes==1 mes==3 ...)	4==1 4==3 ... Falso	ano=2342	mes=4	DiasDoMes	ano=2342	mes=4
10	23	return 30;		ano=2342	mes=4	DiasDoMes=30	ano=2342	mes=4
11	35	printf(...,30);	30	ano=2342	mes=4			

Comecemos pela função `main`. Esta função ficou muito curta quando comparado com o código original, dado que o cálculo do dia do mês é feito numa função que retorna o número de dias. A função `main` apenas tem de ler os argumentos e chamar a função respectiva. Para chamar a função `DiasDoMes` não interessa sequer saber que esta utiliza a função `Bissexto`.

A função `DiasDoMes` tem os testes que estavam na função `main` do código original, mas em vez do condicional mais complexo do ano bissexto, tem uma chamada à função `Bissexto`. Esta função tem dois parâmetros, ao contrário da função `Bissexto`. Desta forma move-se complexidade do problema para outra função, tornado a leitura mais simples de cada parte do código.

A função `Bissexto` foi copiada do outro programa, portanto reutilizada. Quanto mais genérica for a função maior o seu potencial para ser útil tanto em várias zonas do código, como em outros programas. Ao copiar código de um programa para outro pode-se utilizar o código copiado com alguma garantia, dado que já foi testado e utilizado em outro programa, e é provável que contenha poucos erros.

Erros comuns mais directamente associados a este capítulo:

- **Funções com parâmetros no nome**

- **Forma:** Para distinguir uma constante importante na função, basta colocar o valor do parâmetro no nome da função, por exemplo `funcao12`, para a função que retorne o resto da divisão por 12. Desta forma evita-se utilizar um argumento
- **Problema:** Se distingue uma constante importante no nome da função, então deve colocar a constante como argumento da função, ficando a função a funcionar para qualquer constante. Caso não o faça, corre o risco de ao lado de `funcao12`, ser necessário a `funcao4`, `funcao10`, etc., ficando com instruções parecidas. Não poupa sequer na escrita dado que coloca no nome da função a constante que utiliza na função
- **Resolução:** Se tem casos destes no seu código, deve fazer uma troca simples: substituir todas as constantes dependentes de 12 pelo argumento, ou uma expressão dependente do argumento.

- **Instruções parecidas não seguidas**

- **Forma:** Quando é necessário uma ou mais instruções que são parecidas com outras que já escritas noutra parte do código, basta seleccionar e fazer uso de uma das principais vantagens dos documentos digitais: `copy/paste`
- **Problema:** O `copy/paste` vai dificultar a leitura, e também a escrita tem de ser com muito cuidado, para poder editar as diferenças. Se esta situação ocorrer, é uma

indicação clara que é necessária uma função com as instruções que são parecidas e devem ser reutilizadas. Nos argumentos da função deve ir informação suficiente para que a função possa implementar as diferenças

- **Resolução:** Se aconteceu no seu código, deve estudar quais as diferentes alternativas para utilização de funções que tem, de forma a garantir que não lhe escapam hipóteses antes de optar
- **Funções específicas**
 - **Forma:** Para uma situação é necessária uma função concreta, pelo que é essa função que é definida, utilizando as constantes necessárias dentro dessa função
 - **Problema:** Se as funções são muito específicas, não podem ser reutilizadas, tanto no mesmo programa, como em outros programas. Quanto mais genérica é a função, maior é o seu grau de reutilização, sendo mais simples de manter e detectar algum problema
 - **Resolução:** Reveja todas as constantes que tem numa função específica, e passe as constantes que sejam passíveis de serem mudadas para argumentos da função, de forma a aumentar as possibilidades da função ser útil sem alterações em outras situações
- **Declarações de variáveis fora do início das funções**
 - **Forma:** As variáveis devem ser declaradas o mais perto possível do local onde são utilizadas, e como alguns compiladores de C permitem a declaração fora do início das funções, há que aproveitar
 - **Problema:** É verdadeira a frase, mas não para quem se inicie na programação, nem para a linguagem C. A generalidade dos compiladores de C permite declaração das variáveis em qualquer parte do código, mas nem todos, pelo que perde compatibilidade. Mais relevante é a má influência que tal situação terá, facilitando a existência de funções grandes. Com as variáveis declaradas em cada bloco, não irá “sofrer” os efeitos de funções grandes tão intensamente, o que o pode levar a não sentir a necessidade de criar funções
 - **Resolução:** Todas as declarações fora do início das funções passam para o início das funções. Reveja os nomes das variáveis se necessário

O Programa 5-1 tem um exemplo de um erro de instruções parecidas não seguidas, de forma a motivar a utilização de funções. Um exemplo de funções com parâmetros no nome é no Programa 5-4 existir uma função `DiasDoMes2010`, que recebe um mês e retorna o número de dias desse mês em 2010. Esta versão é também um exemplo de uma função específica, dado que contém uma constante que poderia estar num argumento, mas a situação da constante ir para o nome da função é de maior gravidade. A existência de uma constante interna a uma função, justifica-se se o seu valor raramente fizer sentido ser distinto do utilizado na implementação.

Estes exemplos são muito curtos, mas num conceito tão importante quanto este, tem que se dar tempo para que entre devagar e seja compreendido parte da sua potencialidade. Com a experiência ficará claro quais as funções que deve criar, de forma a obter código de qualidade, modular, reutilizável, e de simples leitura. Por agora, duas características de uma boa função:

- **Uma função, uma operação** (não juntar alhos com bugalhos);
- Se não tem uma **funcionalidade ou nome claro**, então não é uma função.

Reveja código que tenha feito no módulo anterior, e faça uma versão modular de cada um, utilizando funções.

Da boa divisão do problema em funções resultará o menor trabalho necessário para implementá-lo, enquanto uma má divisão resulta não a inviabilização da implementação, mas sim um aumento de trabalho necessário. Quanto mais trabalho, provavelmente menor qualidade terá o código.

6. MAIS CICLOS E CONDICIONAIS

Este capítulo começa em "mais", porque na verdade não introduz novos conceitos, mas sim apresenta alternativas que a linguagem C oferece para aplicar esses conceitos: ciclos e condicionais.

A primeira situação refere-se à constatação que a maior parte dos ciclos tem uma variável iteradora, que controla o teste de paragem do ciclo. Essa variável iteradora tem de ser inicializada antes do ciclo, e incrementada no final do ciclo, de forma a ficar com o valor válido quando a expressão lógica é avaliada. Exemplo disso é o primeiro programa no capítulo Ciclos.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int soma=0, i=1;
6     while(i<=4)
7     {
8         soma+=i*i;
9         i++;
10    }
11    printf("soma dos primeiros 4 quadrados: %d", soma);
12 }
```

Programa 6-1 Réplica do Programa 4-2

Note-se na **atribuição da variável i ao valor 1** antes do ciclo, e da **operação de incremento** no final. No entanto a operação que interessa fazer é a operação de `soma+=i*i`; . Esta situação prejudica a legibilidade do código porque há duas instruções de controlo do ciclo, uma antes do ciclo e outra no ciclo, que nada têm a ver com o que se pretende fazer, apenas estão lá para controlar a paragem do ciclo, perturbando a leitura.

A linguagem C permite efectuar ciclos iterativos juntando a inicialização e actualização de variáveis, junto com o teste de paragem, podendo ficar tudo numa só linha, de forma a deixar as restantes linhas para instruções não relacionadas com o controlo do ciclo.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int soma=0, i;
6
7     for(i=1; i<=4; i++)
8         soma+=i*i;
9
10    printf("soma dos primeiros 4 quadrados: %d", soma);
11 }
```

Programa 6-2 Soma dos primeiros 4 quadrados naturais com o ciclo "for"

O código acima tem a mesma funcionalidade que o original, mas todas as instruções de controlo do ciclo na linha **for**. O ciclo **for**, ao contrário do ciclo **while**, aceita três parâmetros entre parênteses, mas separados por ponto e vírgulas. Na primeira zona está a inicialização, na zona central está a expressão lógica tal como no **while**, e na última zona está a actualização da variável iteradora.

Pode-se ver que na execução passo-a-passo, o ciclo **for** fica sempre em dois passos. Na primeira passagem, tem que se executar a inicialização, e num segundo passo mantendo na mesma linha, tem que se avaliar a expressão lógica. Nas restantes passagens, tem que se executar a actualização, e de

seguida a avaliação. Para não haver confusão entre a zona do ciclo for que está a ser executada num dado passo, as zonas não executadas são substituídas por um cardinal (#).

Têm que existir sempre três parâmetros no ciclo for, mesmo que um campo seja vazio, tem que se colocar o ponto e vírgula. Se houver lugar a mais que uma instrução de inicialização, estas devem estar separadas por uma vírgula, acontecendo o mesmo para a instrução de actualização. Por exemplo, pode-se querer inicializar o valor da variável soma no ciclo.

Passo	Linha	Instrução	Resultado		
1	5	int soma=0, i;		soma=0	i=?
2	7	for(i=1; #; #)		soma=0	i=1
3	7	for(#; i<=4; #)	1<=4 Verdade	soma=0	i=1
4	8	soma+=i*i;		soma+=1*1=1	i=1
5	7	for(#; #; i++)		soma=1	i=2
6	7	for(#; i<=4; #)	2<=4 Verdade	soma=1	i=2
7	8	soma+=i*i;		soma+=2*2=5	i=2
8	7	for(#; #; i++)		soma=5	i=3
9	7	for(#; i<=4; #)	3<=4 Verdade	soma=5	i=3
10	8	soma+=i*i;		soma+=3*3=14	i=3
11	7	for(#; #; i++)		soma=14	i=4
12	7	for(#; i<=4; #)	4<=4 Verdade	soma=14	i=4
13	8	soma+=i*i;		soma+=4*4=30	i=4
14	7	for(#; #; i++)		soma=30	i=5
15	7	for(#; i<=4; #)	5<=4 Falso	soma=30	i=5
16	10	printf(...);	Soma dos primeiros 4 quadrados: 30	soma=30	i=5

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int soma, i;
6
7     for(i=1, soma=0; i<=4; i++)
8         soma+=i*i;
9
10    printf("Soma dos primeiros 4 quadrados: %d", soma);
11 }
```

Programa 6-3 Utilização alternativa do ciclo “for” na soma dos 4 quadrados naturais

Este programa é idêntico ao anterior, mas não se aconselha efectuar inicializações nem actualizações no ciclo for, que não digam respeito à variável iteradora e ao controle do ciclo. Caso contrário o código torna-se de mais leitura pesada. Em vez de se esperar instruções menores de controlo de ciclo, dentro do ciclo for passa a poder estar apenas instruções com significado mais relevante para o algoritmo a implementar.

Quanto mais ciclos, maior são as vantagens do ciclo for relativamente ao ciclo while, como se pode ver na versão mais simples do Programa 4-4.

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int A,B,K,contagem;
6     printf("Indique K: ");
7     scanf("%d",&K);
8     contagem=0;
9     for(A=1; A<=K; A++)
10         for(B=1; B<=K; B++)
11             if(A+B<=K && A*B<=K)
12                 contagem++;
13     printf("Total: %d",contagem);
14 }
```

Programa 6-4 Versão com ciclos “for” do Programa 4-4

Nesta versão do código observa-se uma clara diminuição do número de instruções e aumento de legibilidade, dado que tudo o que trata de cada ciclo está numa só linha, e não em diversas linhas e alternado linhas de controlo de um ciclo com as do outro.

Mostra-se a execução passo-a-passo dos primeiros 20 passos. Esta execução, quando comparada com a versão utilizando `while`, pode-se verificar que não há diferença excepto na troca do `for` pelo `while`.

Passo	Linha	Instrução	Resultado	A=?	B=?	K=?	contagem=?
1	5	int A,B,K,contagem;		A=?	B=?	K=?	contagem=?
2	6	printf(...);	Indique K:	A=?	B=?	K=?	contagem=?
3	7	scanf(...,&K);		A=?	B=?	K=3	contagem=?
4	8	contagem=0;		A=?	B=?	K=3	contagem=0
5	9	for(A=1; #; #)		A=1	B=?	K=3	contagem=0
6	9	for(#; A<=K; #)	1<=3 Verdade	A=1	B=?	K=3	contagem=0
7	10	for(B=1; #; #)		A=1	B=1	K=3	contagem=0
8	10	for(#; B<=K; #)	1<=3 Verdade	A=1	B=1	K=3	contagem=0
9	11	if(A+B<=K && A*B<=K)	1+1<=3 && 1*1<=3 Verdade	A=1	B=1	K=3	contagem=0
10	12	contagem++;		A=1	B=1	K=3	contagem=1
11	10	for(#; #; B++)		A=1	B=2	K=3	contagem=1
12	10	for(#; B<=K; #)	2<=3 Verdade	A=1	B=2	K=3	contagem=1
13	11	if(A+B<=K && A*B<=K)	1+2<=3 && 1*2<=3 Verdade	A=1	B=2	K=3	contagem=1
14	12	contagem++;		A=1	B=2	K=3	contagem=2
15	10	for(#; #; B++)		A=1	B=3	K=3	contagem=2
16	10	for(#; B<=K; #)	3<=3 Verdade	A=1	B=3	K=3	contagem=2
17	11	if(A+B<=K && A*B<=K)	1+3<=3 && (ignorado) Falso	A=1	B=3	K=3	contagem=2
18	10	for(#; #; B++)		A=1	B=4	K=3	contagem=2
19	10	for(#; B<=K; #)	4<=3 Falso	A=1	B=4	K=3	contagem=2
20	9	for(#; #; A++)		A=2	B=4	K=3	contagem=2

Para que não exista qualquer dúvida, o ciclo `for` genérico:

```
for(«inicialização»; «expressão lógica»; «actualização»)
    «instrução»;
```

É equivalente ao ciclo `while` genérico:

```
«inicialização»;
while(«expressão lógica»)
{
    «instrução»;
    «actualização»;
}
```

Se a expressão lógica for falsa logo no primeiro ciclo, as instruções do ciclo nunca são executadas, quer se utilize o ciclo `for` ou o ciclo `while`. Pode-se sempre alterar a expressão lógica para que seja verdadeira no primeiro ciclo, mas complicando a expressão, pelo que a linguagem C tem uma outra forma de executar ciclos, o **do-while**:

```
do
{
    «instrução»;
} while(«expressão lógica»);
```

Exemplifica-se o ciclo `do-while` com o exercício do menu em Funções, que pode ser reescrito da seguinte forma:

```

1  #include <stdio.h>
2
3  int Menu()
4  {
5      int opcao;
6      /* mostrar as opções do menu */
7      printf("\nMenu:\n1 - opcao A\n2 - opcao B\n3 - opcao C\n0 - sair");
8      printf("\nopcao: ");
9      scanf("%d",&opcao);
10     /* retornar a opção seleccionada */
11     return opcao;
12 }
13
14 int main()
15 {
16     int opcao;
17     do
18     {
19         opcao=Menu();
20         /* ver qual é a opção */
21         if(opcao==1)
22             printf("Opcao escolhida A");
23         else if(opcao==2)
24             printf("Opcao escolhida B");
25         else if(opcao==3)
26             printf("Opcao escolhida C");
27         else if(opcao!=0)
28             printf("Opcao invalida");
29     } while(opcao>0);
30     printf("Fim do programa.");
31 }

```

Programa 6-5 Versão com ciclos “do-while” do Programa 5-2

Mostra-se a execução passo-a-passo dos primeiros 20 passos. Este programa tem o mesmo funcionamento que o original, mas com a utilização do ciclo do-while é possível chamar apenas uma só vez a função Menu. Com esta solução pode-se ponderar a necessidade de uma função para o menu, uma vez que a função é chamada apenas uma só vez, e tem uma dimensão reduzida.

Passo	Linha	Instrução	Resultado			
1	16	int opcao;		opcao=?		
2	19	opcao=Menu();		opcao=?	Menu	
3	5	int opcao;		opcao=?	Menu	opcao=?
4	7	printf(...);	Menu:	opcao=?	Menu	opcao=?
5	8	printf(...);	Opcao:	opcao=?	Menu	opcao=?
6	9	scanf(...,&opcao);		2 opcao=?	Menu	opcao=2
7	11	return opcao;		opcao=?	Menu=2	opcao=2
8	19	opcao=Menu();		opcao=2		
9	21	if(opcao==1)	2==1 Falso	opcao=2		
10	23	if(opcao==2)	2==2 Verdade	opcao=2		
11	24	printf(...);	Opcao escolhida B	opcao=2		
12	29	while(opcao>0)	2>0 Verdade	opcao=2		
13	19	opcao=Menu();		opcao=2	Menu	
14	5	int opcao;		opcao=2	Menu	opcao=?
15	7	printf(...);	Menu:	opcao=2	Menu	opcao=?
16	8	printf(...);	Opcao:	opcao=2	Menu	opcao=?
17	9	scanf(...,&opcao);		5 opcao=2	Menu	opcao=5
18	11	return opcao;		opcao=2	Menu=5	opcao=5
19	19	opcao=Menu();		opcao=5		
20	21	if(opcao==1)	5==1 Falso	opcao=5		

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int opcao;
6      do
7      {
8          /* mostrar as opções do menu */
9          printf("\nMenu:\n1 - opcao A\n2 - opcao B\n3 - opcao C\n0 - sair");
10         printf("\nopcao: ");
11         scanf("%d",&opcao);
12         /* ver qual é a opção */
13         if(opcao==1)
14             printf("opcao escolhida A");
15         else if(opcao==2)
16             printf("opcao escolhida B");
17         else if(opcao==3)
18             printf("opcao escolhida C");
19         else if(opcao!=0)
20             printf("opcao invalida");
21     } while(opcao>0);
22     printf("Fim do programa.");
23 }

```

Programa 6-6 Versão removendo a função Menu do Programa 6-5

Os ciclos `while` e `do-while` devem ser utilizados nos ciclos menos ortodoxos quando não há uma variável iteradora clara, ou a expressão lógica é muito complexa, de forma a não ter numa só linha do ciclo `for` muito longa. Nos ciclos normais, a opção pelo ciclo `for` resulta quase sempre em ganho de legibilidade.

Os condicionais têm também uma alternativa à cadeia `if-else` utilizada no programa acima. Em situações em que as expressões lógicas na cadeia `if-else` são comparações com valores concretos, como é o caso do exemplo do menu, pode-se utilizar o **switch**, que recebe uma variável e lista-se todos os casos.

Passo	Linha	Instrução	Resultado	
1	5	int opcao;		opcao=?
2	9	printf(...);	Menu:	opcao=?
3	10	printf(...);	Opcao:	opcao=?
4	11	scanf(...,&opcao);		2 opcao=2
5	13	if(opcao==1)	2==1 Falso	opcao=2
6	15	if(opcao==2)	2==2 Verdade	opcao=2
7	16	printf(...);	Opcao escolhida B	opcao=2
8	21	while(opcao>0)	2>0 Verdade	opcao=2
9	9	printf(...);	Menu:	opcao=2
10	10	printf(...);	Opcao:	opcao=2
11	11	scanf(...,&opcao);		5 opcao=5
12	13	if(opcao==1)	5==1 Falso	opcao=5
13	15	if(opcao==2)	5==2 Falso	opcao=5
14	17	if(opcao==3)	5==3 Falso	opcao=5
15	19	if(opcao!=0)	5!=0 Verdade	opcao=5
16	20	printf(...);	Opcao invalida	opcao=5
17	21	while(opcao>0)	5>0 Verdade	opcao=5
18	9	printf(...);	Menu:	opcao=5
19	10	printf(...);	Opcao:	opcao=5
20	11	scanf(...,&opcao);		0 opcao=0
21	13	if(opcao==1)	0==1 Falso	opcao=0
22	15	if(opcao==2)	0==2 Falso	opcao=0
23	17	if(opcao==3)	0==3 Falso	opcao=0
24	19	if(opcao!=0)	0!=0 Falso	opcao=0
25	21	while(opcao>0)	0>0 Falso	opcao=0
26	22	printf(...);	Fim do programa.	opcao=0

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int opcao;
6      do
7      {
8          /* mostrar as opções do menu */
9          printf("\nMenu:\n1 - opcao A\n2 - opcao B\n3 - opcao C\n0 - sair");
10         printf("\nOpcao: ");
11         scanf("%d",&opcao);
12         /* ver qual é a opção */
13         switch(opcao)
14         {
15             case 1:
16                 printf("Opcao escolhida A");
17                 break;
18             case 2:
19                 printf("Opcao escolhida B");
20                 break;
21             case 3:
22                 printf("Opcao escolhida C");
23                 break;
24             case 0:
25                 break;
26             default:
27                 printf("Opcao invalida");
28         }
29     } while(opcao>0);
30     printf("Fim do programa.");
31 }

```

Programa 6-7 Versão com “switch” do Programa 6-6

O switch tem de ter o valor de uma **expressão numérica como base**, e não uma expressão lógica, sendo colocada a execução na linha de código que tiver o caso que ocorreu (case «valor»:). Se o valor não existir em nenhum dos casos listados, a execução passa para a linha onde estiver "default:" se existir. Com estas regras, poderia haver problema em utilizar o switch não fosse a existência dos comandos break; que saem nesse instante do comando switch. Se não fosse assim, no caso da opção 1 ser escolhida, o programa passaria para a linha do caso 1, e de seguida seguiria normalmente para o caso 2, 3 e a opção inválida, só porque estão nas linhas de código após a opção 1. Colocando um break; após o fim do código que trata cada opção, resolve-se o problema.

Pode-se ver na execução passo-a-passo que nos passos 5, 13 e 20, o switch passou imediatamente para a linha que trata o caso pretendido, em vez de executar vários condicionais.

Passo	Linha	Instrução	Resultado	
1	5	int opcao;		opcao=?
2	9	printf(...);	Menu:	opcao=?
3	10	printf(...);	Opcao:	opcao=?
4	11	scanf(...,&opcao);		2 opcao=2
5	13	switch(opcao)	case 2	opcao=2
6	18	case 2:		
7	19	printf(...);	Opcao escolhida B	opcao=2
8	20	break;		
9	29	while(opcao>0)	2>0 Verdade	opcao=2
10	9	printf(...);	Menu:	opcao=2
11	10	printf(...);	Opcao:	opcao=2
12	11	scanf(...,&opcao);		5 opcao=5
13	13	switch(opcao)	default	opcao=5
14	26	default:		opcao=5
15	27	printf(...);	Opcao invalida	opcao=5
16	29	while(opcao>0)	5>0 Verdade	opcao=5
17	9	printf(...);	Menu:	opcao=5
18	10	printf(...);	Opcao:	opcao=5
19	11	scanf(...,&opcao);		0 opcao=0
20	13	switch(opcao)	case 0	opcao=0
21	24	case 0:		opcao=0
22	25	break;		opcao=0
23	29	while(opcao>0)	0>0 Falso	opcao=0
24	30	printf(...);	Fim do programa.	opcao=0

Num outro exemplo podemos ver uma implementação alternativa da função DiasDoMes, extraída do Programa 5-4:

```
1 int DiasDoMes(int mes, int ano)
2 {
3     switch(mes)
4     {
5         case 2:
6             /* teste de ano bissexto */
7             if(Bissexto(ano))
8                 return 29;
9             else
10                return 28;
11            /* meses de dias 31 */
12        case 1:
13        case 3:
14        case 5:
15        case 7:
16        case 8:
17        case 10:
18        case 12:
19            return 31;
20        default:
21            return 30;
22    }
23 }
```

Programa 6-8 Função DiasDoMes com “switch” do Programa 5-4

Neste caso pretende-se identificar todos os casos de meses de 31, pelo que basta colocar os diferentes casos em cada linha, e retornar o valor correcto apenas após a linha que identifica o último mês de 31 dias. Como aqui utilizam-se comandos `return`, que saem imediatamente da função, não há necessidade de utilizar a instrução `break`; no `switch`.

O comando **`break`**; pode ser utilizado também em ciclos, para sair do ciclo actual. Existe ainda o comando **`continue`**; utilizável dentro de ciclos, que permite saltar para a próxima iteração do ciclo. Estes comandos não se devem no entanto utilizar com frequência, dado que dessa forma as expressões lógicas, inicializações e actualizações associados ao ciclo, não controlam completamente o ciclo, sendo auxiliadas por eventuais condicionais dentro do ciclo, sobre a qual os comandos `break`; e `continue`; estão. Sendo de evitar a utilização destes comandos, não será dado nenhum exemplo em que tais comandos sejam úteis.

Erros comuns mais directamente associados a este capítulo:

- **Ciclos contendo apenas uma variável lógica no teste de paragem**
 - **Forma:** Esta estratégia consiste em utilizar nos ciclos (normalmente `while`), uma variável no teste de paragem, atribuindo o valor verdadeiro e dentro do ciclo, quando for necessário sair dele, fazer uma atribuição à variável a falso
 - **Problema:** O real teste de paragem fica diluído nas instruções que alteram esta variável, bem como condicionais que utilizem o `break`, e ter algo em vez de num local em vários, torna sempre mais complicada a leitura e manutenção de código. Esta situação ocorre porque falhou na identificação do real teste de paragem. É pior se necessitar de acompanhar esta variável com uma outra variável que altera de valor em cada ciclo (uma variável iteradora)
 - **Resolução:** Se optar com frequência para esta situação, deve rever o seu código procurando alternativas

Um exemplo simples deste erro é a seguinte versão do Programa 4-2:

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int soma=0, i=1, continua=1;
6     while(continua)
7     {
8         soma+=i*i;
9         i++;
10        if(i>4)
11            continua=0;
12    }
13    printf("Soma dos primeiros 4 quadrados: %d", soma);
14 }
```

A variável lógica `continua` pode ser actualizada em qualquer parte do ciclo, sendo a verdadeira condição de paragem distribuída pelos condicionais associados à sua alteração. Este exemplo é tão pequeno que não dá para dividir a condição de paragem, apenas deslocá-la do ciclo `while` para o final do ciclo, mas mesmo assim pode-se verificar a degradação da clareza do código. Uma outra versão a evitar teria sido em vez da variável `continua` utilizar-se a constante 1 de forma a obter-se um ciclo infinito, utilizando-se a instrução `break`; em vez de `continua=0`; para sair do ciclo.

Todos os ciclos têm a mesma complexidade, pode-se trocar um ciclo por outro, no entanto com o ciclo `for` o código fica normalmente melhor arrumado. A utilização do `switch` normalmente não melhora a leitura do código, mas é sempre uma opção em relação ao `if`. Agora que conhece todas as alternativas, faça uma revisão aos exercícios que fez e refaça os ciclos e condicionais de forma a melhorar a simplicidade e leitura do seu código.

7. VECTORES

Nos programas tratados até este momento, foram utilizadas variáveis de diferentes tipos, mas nunca se abordou **conjuntos de variáveis**.

Suponhamos que estamos interessados em registrar um valor por cada dia da semana. A natureza dos valores não é relevante, tanto pode ser o número de emails enviados, como o número de cafés tomados, o que interessa é que se pretende guardar um valor por cada dia da semana, e no final obter a soma e a média dos valores introduzidos.

Com o que sabemos, o seguinte programa resolveria o problema:

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int segunda, terca, quarta, quinta, sexta, sabado, domingo;
6     int total;
7
8     /* introdução de valores */
9     printf("Segunda: ");
10    scanf("%d",&segunda);
11    printf("Terca: ");
12    scanf("%d",&terca);
13    printf("Quarta: ");
14    scanf("%d",&quarta);
15    printf("Quinta: ");
16    scanf("%d",&quinta);
17    printf("Sexta: ");
18    scanf("%d",&sexta);
19    printf("Sabado: ");
20    scanf("%d",&sabado);
21    printf("Domingo: ");
22    scanf("%d",&domingo);
23
24    /* calculos */
25    total=segunda+terca+quarta+quinta+sexta+sabado+domingo;
26    printf("Soma: %d\n",total);
27    printf("Media: %f\n",total/7.0);
28 }

```

Programa 7-1 Calculo do total e média de um indicador por cada dia da semana

Execução do programa:

```

C:\>semana
Segunda: 2
Terca: 3
Quarta: 1
Quinta: 2
Sexta: 3
Sabado: 4
Domingo: 1
Soma: 16
Media:
2.285714

```

Se necessitamos de um valor por cada dia da semana, temos de criar uma variável por cada dia da semana. Neste caso a

Passo	Linha	Instrução	Resultado	segunda=?	terca=?	quarta=?	quinta=?	sexta=?	sabado=?	domingo=?	
1	5	int segunda, ...;		segunda=?	terca=?	quarta=?	quinta=?	sexta=?	sabado=?	domingo=?	
2	6	int total;		segunda=?	terca=?	quarta=?	quinta=?	sexta=?	sabado=?	domingo=?	total=?
3	9	printf(...);		segunda=?	terca=?	quarta=?	quinta=?	sexta=?	sabado=?	domingo=?	total=?
4	10	scanf(...,&segunda);	2	segunda=2	terca=?	quarta=?	quinta=?	sexta=?	sabado=?	domingo=?	total=?
5	11	printf(...);		segunda=2	terca=?	quarta=?	quinta=?	sexta=?	sabado=?	domingo=?	total=?
6	12	scanf(...,&terca);	3	segunda=2	terca=3	quarta=?	quinta=?	sexta=?	sabado=?	domingo=?	total=?
7	13	printf(...);		segunda=2	terca=3	quarta=?	quinta=?	sexta=?	sabado=?	domingo=?	total=?
8	14	scanf(...,&quarta);	1	segunda=2	terca=3	quarta=1	quinta=?	sexta=?	sabado=?	domingo=?	total=?
9	15	printf(...);		segunda=2	terca=3	quarta=1	quinta=?	sexta=?	sabado=?	domingo=?	total=?
10	16	scanf(...,&quinta);	2	segunda=2	terca=3	quarta=1	quinta=2	sexta=?	sabado=?	domingo=?	total=?
11	17	printf(...);		segunda=2	terca=3	quarta=1	quinta=2	sexta=?	sabado=?	domingo=?	total=?
12	18	scanf(...,&sexta);	3	segunda=2	terca=3	quarta=1	quinta=2	sexta=3	sabado=?	domingo=?	total=?
13	19	printf(...);		segunda=2	terca=3	quarta=1	quinta=2	sexta=3	sabado=?	domingo=?	total=?
14	20	scanf(...,&sabado);	4	segunda=2	terca=3	quarta=1	quinta=2	sexta=3	sabado=4	domingo=?	total=?
15	21	printf(...);		segunda=2	terca=3	quarta=1	quinta=2	sexta=3	sabado=4	domingo=?	total=?
16	22	scanf(...,&domingo);	1	segunda=2	terca=3	quarta=1	quinta=2	sexta=3	sabado=4	domingo=1	total=?
17	25	total=segunda+...;		segunda=2	terca=3	quarta=1	quinta=2	sexta=3	sabado=4	domingo=1	total=16
18	26	printf(...);	Soma: 16	segunda=2	terca=3	quarta=1	quinta=2	sexta=3	sabado=4	domingo=1	total=16
19	27	printf(...);	Media: 2.285714	segunda=2	terca=3	quarta=1	quinta=2	sexta=3	sabado=4	domingo=1	total=16

operação pedida é muito simples, a variável `total` pode ir sendo actualizada à medida que se introduz os valores, podendo-se evitar ter criado uma variável por cada dia. Mas é apenas um exemplo que pretende ser o mais simples possível. Se quiser calcular a variância, tem-se de fazer outra passagem pelos valores da semana após obter a média, sendo nesse caso essencial que os valores fiquem em memória.

Como temos variáveis distintas, não podemos fazer um ciclo para fazer a operação de soma, por exemplo. Evidentemente que esta opção não seria prática se em vez de termos 7 variáveis, fossem necessárias 31 para cada dia do mês, ou 365 para cada dia do ano.

A solução é poder criar com um só nome, um conjunto de variáveis, quantas forem necessárias, e ter uma forma de acesso através de um **índice**.

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int diasDaSemana[7];
6     int total=0,i;
7
8     /* introdução de valores */
9     printf("segunda: ");
10    scanf("%d",&diasDaSemana[0]);
11    printf("Terça: ");
12    scanf("%d",&diasDaSemana[1]);
13    printf("Quarta: ");
14    scanf("%d",&diasDaSemana[2]);
15    printf("Quinta: ");
16    scanf("%d",&diasDaSemana[3]);
17    printf("Sexta: ");
18    scanf("%d",&diasDaSemana[4]);
19    printf("Sabado: ");
20    scanf("%d",&diasDaSemana[5]);
21    printf("Domingo: ");
22    scanf("%d",&diasDaSemana[6]);
23
24    /* calculos */
25    for(i=0;i<7;i++)
26        total+=diasDaSemana[i];
27    printf("Soma: %d\n",total);
28    printf("Media: %f\n",total/7.0);
29 }
```

Programa 7-2 Calculo do total e média, versão com vectores

Note-se na declaração da variável na linha 5. Além do nome, tem entre parêntesis rectos, o número 7. Significa isto que a declaração não se refere a uma só variável inteira, mas sim a um **conjunto de 7 variáveis**, um **vector com 7 elementos**. Em vez de `int` poderia estar qualquer outro tipo, não há diferença nenhuma, podem-se declarar vectores de qualquer tipo.

Passo	Linha	Instrução	Resultado			
1	5	int diasDaSemana[7];		diasDaSemana={?,?,?,?,?,?,?}		
2	6	int total=0,i;		diasDaSemana={?,?,?,?,?,?,?}	total=0	i=?
3	9	printf(...);		diasDaSemana={?,?,?,?,?,?,?}	total=0	i=?
4	10	scanf(...,&diasDaSemana[0]);	2	diasDaSemana={2,?,?,?,?,?,?}	total=0	i=?
5	11	printf(...);		diasDaSemana={2,?,?,?,?,?,?}	total=0	i=?
6	12	scanf(...,&diasDaSemana[1]);	3	diasDaSemana={2,3,?,?,?,?,?}	total=0	i=?
7	13	printf(...);		diasDaSemana={2,3,?,?,?,?,?}	total=0	i=?
8	14	scanf(...,&diasDaSemana[2]);	1	diasDaSemana={2,3,1,?,?,?,?}	total=0	i=?
9	15	printf(...);		diasDaSemana={2,3,1,?,?,?,?}	total=0	i=?
10	16	scanf(...,&diasDaSemana[3]);	2	diasDaSemana={2,3,1,2,?,?,?}	total=0	i=?
11	17	printf(...);		diasDaSemana={2,3,1,2,?,?,?}	total=0	i=?
12	18	scanf(...,&diasDaSemana[4]);	3	diasDaSemana={2,3,1,2,3,?,?}	total=0	i=?
13	19	printf(...);		diasDaSemana={2,3,1,2,3,?,?}	total=0	i=?
14	20	scanf(...,&diasDaSemana[5]);	4	diasDaSemana={2,3,1,2,3,4,?}	total=0	i=?
15	21	printf(...);		diasDaSemana={2,3,1,2,3,4,?}	total=0	i=?
16	22	scanf(...,&diasDaSemana[6]);	1	diasDaSemana={2,3,1,2,3,4,1}	total=0	i=?
17	25	for(i=0; i<7; i++)		diasDaSemana={2,3,1,2,3,4,1}	total=0	i=0
18	25	for(i=0; i<7; i++)	0<7 Verdade	diasDaSemana={2,3,1,2,3,4,1}	total=0	i=0
19	26	total+=diasDaSemana[i];		diasDaSemana={2,3,1,2,3,4,1}	total+=2=2	i=0
20	25	for(i=0; i<7; i++)		diasDaSemana={2,3,1,2,3,4,1}	total=2	i=1

Como aceder a cada uma das 7 variáveis? Simplesmente utilizando também entre parêntesis rectos, o índice da variável, de 0 a 6. É isso que é feito na introdução dos valores, e que permite na linha 25 e 26 construir um ciclo que processa todos os dias da semana.

Se em vez de 7 fossem 31, não havia problema, bastaria alterar a constante 7 para 31. Não há problema excepto na introdução de dados, que agora teria de ser copiada e estendida para introduzir 31 valores em vez de 7. Será possível colocar também esta zona de baixo de um ciclo? Sim, desde que se tenha as strings também debaixo de um vector. Mas afinal qual é o tipo associado à string?

Uma string é um conjunto de caracteres. Acabámos de fazer

um conjunto de 7 valores inteiros, com um vector, e sabemos como criar variáveis que sejam caracteres. O que obtemos se declararmos simplesmente **um vector de caracteres?**

```
char str[10];
```

A resposta é simples, uma string. A declaração da variável `str` acima como sendo um vector de 10 caracteres, é uma string. Na primeira posição `str[0]` está o primeiro carácter da string, na segunda posição `str[1]` está o segundo carácter, e assim sucessivamente.

No entanto há aqui um problema. Se der a variável `str` a uma função, tem de passar também o tamanho da string, caso contrário se estivesse a processar a string, haveria o risco de aceder fora dela, em `str[10]`, `str[11]`, etc.

Na declaração dos vectores na linguagem C, o espaço em memória para guardar as 10 variáveis pedidas é reservado, e apenas isso, não é guardado nenhum registo do tamanho do vector. Nas instruções de acesso aos elementos, se existir um acesso a elementos fora da dimensão alocada vai-se estar a aceder a espaço de memória que poderá estar a ser utilizado para outras variáveis, provavelmente as variáveis que estão declaradas a seguir, e que podem até ser de outro tipo. É da responsabilidade do programador garantir que isto não aconteça.

Para as strings, de forma a evitar andar a guardar o número de caracteres, e também por outros motivos, convencionou-se que o último elemento da string é o carácter 0. Esse carácter não faz parte da string, apenas indica que a string acabou. Portanto na declaração é conveniente alocar espaço suficiente para guardar todos os caracteres da string, mais um que é o carácter terminador.

Passo	Linha	Instrução	Resultado			
21	25	for(;; i<7; #)	1<7 Verdade	diasDaSemana={2,3,1,2,3,4,1}	total=2	i=1
22	26	total+=diasDaSemana[i];		diasDaSemana={2,3,1,2,3,4,1}	total+=3=5	i=1
23	25	for(;; #; i++)		diasDaSemana={2,3,1,2,3,4,1}	total=5	i=2
24	25	for(;; i<7; #)	2<7 Verdade	diasDaSemana={2,3,1,2,3,4,1}	total=5	i=2
25	26	total+=diasDaSemana[i];		diasDaSemana={2,3,1,2,3,4,1}	total+=1=6	i=2
26	25	for(;; #; i++)		diasDaSemana={2,3,1,2,3,4,1}	total=6	i=3
27	25	for(;; i<7; #)	3<7 Verdade	diasDaSemana={2,3,1,2,3,4,1}	total=6	i=3
28	26	total+=diasDaSemana[i];		diasDaSemana={2,3,1,2,3,4,1}	total+=2=8	i=3
29	25	for(;; #; i++)		diasDaSemana={2,3,1,2,3,4,1}	total=8	i=4
30	25	for(;; i<7; #)	4<7 Verdade	diasDaSemana={2,3,1,2,3,4,1}	total=8	i=4
31	26	total+=diasDaSemana[i];		diasDaSemana={2,3,1,2,3,4,1}	total+=3=11	i=4
32	25	for(;; #; i++)		diasDaSemana={2,3,1,2,3,4,1}	total=11	i=5
33	25	for(;; i<7; #)	5<7 Verdade	diasDaSemana={2,3,1,2,3,4,1}	total=11	i=5
34	26	total+=diasDaSemana[i];		diasDaSemana={2,3,1,2,3,4,1}	total+=4=15	i=5
35	25	for(;; #; i++)		diasDaSemana={2,3,1,2,3,4,1}	total=15	i=6
36	25	for(;; i<7; #)	6<7 Verdade	diasDaSemana={2,3,1,2,3,4,1}	total=15	i=6
37	26	total+=diasDaSemana[i];		diasDaSemana={2,3,1,2,3,4,1}	total+=1=16	i=6
38	25	for(;; #; i++)		diasDaSemana={2,3,1,2,3,4,1}	total=16	i=7
39	25	for(;; i<7; #)	7<7 Falso	diasDaSemana={2,3,1,2,3,4,1}	total=16	i=7
40	27	printf(...);	Soma: 16	diasDaSemana={2,3,1,2,3,4,1}	total=16	i=7
41	28	printf(...);	Media: 2.285714	diasDaSemana={2,3,1,2,3,4,1}	total=16	i=7

```

1 #include <stdio.h>
2
3 int main()
4 {
5     char str[10];
6     int i;
7
8     printf("Introduza uma string: ");
9     gets(str);
10
11     for(i=0;str[i]!=0;i++)
12         printf("\nCaracter %d: %c",i,str[i]);
13 }

```

Programa 7-3 Confirmação que uma string é um vector de caracteres

Execução do programa:

```

C:\>string
Introduza uma string: asdf

Caracter 0: a
Caracter 1: s
Caracter 2: d
Caracter 3: f

```

Pode-se verificar que condição para continuar no ciclo é que carácter actual não seja igual a zero. Veja este exemplo com atenção para compreender como a string introduzida na verdade é realmente apenas um vector do tipo char, com a particularidade de acabar em 0.

Sem esta particularidade seria necessário ter outra variável com a dimensão da string introduzida. Repare que o tamanho da string na declaração é uma constante do programa, enquanto o tamanho da string introduzida foi apenas de 4 caracteres.

Passo	Linha	Instrução	Resultado		
1	5	char str[10];		str={?,...}	
2	6	int i;		str={?,...}	i=?
3	8	printf(...);	Introduza uma string:	str={?,...}	i=?
4	9	gets(str);	asdf	str="asdf"	i=?
5	11	for(i=0; #; #)		str="asdf"	i=0
6	11	for(;; str[i]!=0; #)	'a'!=0 Verdade	str="asdf"	i=0
7	12	printf(...);	Caracter 0: a	str="asdf"	i=0
8	11	for(;; #; i++)		str="asdf"	i=1
9	11	for(;; str[i]!=0; #)	's'!=0 Verdade	str="asdf"	i=1
10	12	printf(...);	Caracter 1: s	str="asdf"	i=1
11	11	for(;; #; i++)		str="asdf"	i=2
12	11	for(;; str[i]!=0; #)	'd'!=0 Verdade	str="asdf"	i=2
13	12	printf(...);	Caracter 2: d	str="asdf"	i=2
14	11	for(;; #; i++)		str="asdf"	i=3
15	11	for(;; str[i]!=0; #)	'f'!=0 Verdade	str="asdf"	i=3
16	12	printf(...);	Caracter 3: f	str="asdf"	i=3
17	11	for(;; str[i]!=0; #)	0!=0 Falso	str="asdf"	i=4

No exemplo acima a variável `str`, que tem não um mas 10 caracteres, foi utilizada sem parênteses rectos na linha 9. Na verdade, enquanto a variável `str[2]` é do tipo char, a variável `str` é do tipo **char[]**, ou **char***, e representa um vector (a dimensão não é relevante). O que a função **gets** recebe é portanto o vector e não um só carácter. Atendendo a que este tipo não é mais que o endereço de memória onde os caracteres estão alojados, é também chamado de apontador. São os apontadores e não cada elemento do vector que é passado para as funções, quando se utiliza vectores. Vamos fazer uma função **strlen** que retorna o tamanho da string, que faz parte da biblioteca `string.h`, mas vamos implementá-la para ilustrar a passagem de vectores para dentro das funções.

```

1 #include <stdio.h>
2
3 int strlen(char str[])
4 {
5     int i;
6     for(i=0;str[i]!=0;i++);
7     return i;
8 }
9
10 int main()
11 {
12     char str[10];
13     printf("Introduza uma string: ");
14     gets(str);
15     printf("A string '%s', tem %d caracteres.",str,strlen(str));
16 }

```

Programa 7-4 Calcular o tamanho de uma string

Execução do programa:

```

C:\>strlen
Introduza uma string: sdf sd
A string 'sdf sd', tem 6 caracteres.

```

A passagem da string na linha 15, foi igual à passagem da string na linha 14, e na declaração da string na função `strlen` na linha 3, utilizou-se o tipo `char[]` (em alternativa pode-se utilizar o tipo `char*`). Na linha 3 não se indicou a dimensão do vector, dado que essa não é conhecida, apenas que é um vector de caracteres. Para a função é passado apenas o endereço do vector, mas com isso a função consegue aceder a todos os elementos do vector, como se pode ver na linha 6.

Na execução passo-a-passo está apenas a chamada à função `strlen`. O argumento passado é apenas a referência ao vector, e não o conteúdo do vector, neste caso 7 caracteres. Os valores acedidos por `str[i]` são os caracteres que estão declarados na função `main`, e não uma cópia.

Falta agora responder a uma questão. Como inicializar os valores num vector? Para as variáveis basta colocar uma atribuição na declaração, mas para os vectores tem-se de colocar uma atribuição para vários valores. É isso mesmo que é feito, como podemos ver neste exemplo da função dos `DiasDoMes`.

Passo	Linha	Instrução	Resultado		
1	3	char str[]="sdf sd"		str="sdf sd"	
2	5	int i;		str="sdf sd"	i=?
3	6	for(i=0; #; #)		str="sdf sd"	i=0
4	6	for(;; str[i]!=0; #)	's'!=0 Verdade	str="sdf sd"	i=0
5	6	for(;; #; i++)		str="sdf sd"	i=1
6	6	for(;; str[i]!=0; #)	'd'!=0 Verdade	str="sdf sd"	i=1
7	6	for(;; #; i++)		str="sdf sd"	i=2
8	6	for(;; str[i]!=0; #)	'f'!=0 Verdade	str="sdf sd"	i=2
9	6	for(;; #; i++)		str="sdf sd"	i=3
10	6	for(;; str[i]!=0; #)	'!'!=0 Verdade	str="sdf sd"	i=3
11	6	for(;; #; i++)		str="sdf sd"	i=4
12	6	for(;; str[i]!=0; #)	's'!=0 Verdade	str="sdf sd"	i=4
13	6	for(;; #; i++)		str="sdf sd"	i=5
14	6	for(;; str[i]!=0; #)	'd'!=0 Verdade	str="sdf sd"	i=5
15	6	for(;; #; i++)		str="sdf sd"	i=6
16	6	for(;; str[i]!=0; #)	0!=0 Falso	str="sdf sd"	i=6
17	7	return i;		6 str="sdf sd"	i=6

```

int DiasDoMes(int mes, int ano)
{
    int diasDoMes[] =
    {
        31,28,31,30,31,30, /* Janeiro a Junho */
        31,31,30,31,30,31 /* Julho a Dezembro */
    };

    if(mes==2 && Bissexto(ano))
        return 29;

    return diasDoMes[mes-1];
}

```

Programa 7-5 Função DiasDoMes com um vector inicializado na declaração

Nesta implementação do DiasDoMes, o número de dias que cada mês tem é colocado na **inicialização do vector**. Neste caso até a dimensão do vector é deixada em aberto, mas poderia estar lá o 12. Como não está no vector fica com o tamanho da lista indicada na atribuição. Ao colocar os valores em memória, deixam de ser necessários os condicionais utilizados anteriormente, e é suficiente retornar o valor com o índice correspondente. Notar que tem que se subtrair por 1, uma vez que o primeiro mês é o 1, enquanto o primeiro índice do vector é o 0.

Agora como é que são **inicializadas as strings**? Será que as strings no printf são diferentes das que falamos aqui, já que são delimitadas por aspas? Vamos ver as respostas no resultado do seguinte programa:

```

1 #include <stdio.h>
2
3 int main()
4 {
5     char str1[]="ABC";
6     char str2[]={ 'A', 'B', 'C', '\0' };
7     char str3[]={65,66,67,0};
8
9     printf("str1: %s\nstr2: %s\nstr3: %s",str1,str2,str3);
10 }

```

Programa 7-6 Exemplificação das diferentes maneiras de se inicializar strings

Execução do programa:

```

C:\>string2
str1: ABC
str2: ABC
str3: ABC

```

As três strings são declaradas e inicializadas de maneiras distintas, no entanto ficam exactamente com o mesmo valor, como se pode confirmar na execução do programa. A string `str1` utiliza o formato que temos vindo a utilizar no `printf`, ou seja, delimitar as strings com aspas. A string entre aspas é convertida para algo idêntico à versão 2, ou seja, uma lista como no exemplo anterior, mas em vez de números tem letras, à qual se junta um último carácter com o valor zero. A versão 3, segue o exemplo anterior, considerando os caracteres como números, fazendo uso do conhecimento do número do carácter A é o 65, e como as letras são números, os três formatos são exactamente iguais. A string tem 3 caracteres, e ocupa 4 bytes em qualquer dos casos.

Estamos agora em condições de resolver o problema inicial de forma mais adequada, ou seja, colocar em ciclo a entrada de dados. Para tal basta colocar num vector as strings a utilizar no `printf`.


```

1 #include <stdio.h>
2
3 int main()
4 {
5     int diasDaSemana[7];
6     int total=0,i;
7     char str[10];
8     char prompt[7][] = {
9         "Segunda: ",
10        "Terca: ",
11        "Quarta: ",
12        "Quinta: ",
13        "Sexta: ",
14        "Sabado: ",
15        "Domingo: "
16    };
17
18    /* introdução de valores */
19    for(i=0;i<7;i++)
20    {
21        printf(prompt[i]);
22        scanf("%d",&diasDaSemana[i]);
23    }
24
25    /* calculos */
26    for(i=0;i<7;i++)
27        total+=diasDaSemana[i];
28    printf("Soma: %d\n",total);
29    printf("Media: %f\n",total/7.0);
30 }

```

Programa 7-7 Calculo do total e média, sem código repetido, utilizando vectores

Ao colocar a informação variável na inicialização da variável prompt, foi possível colocar o código genérico junto, permitindo assim executar ciclos onde estava código duplicado com ligeiras diferenças. Assim será mais fácil alterar o código para funcionar de forma idêntica mas com outros dados, uma vez que se separou a parte genérica da parte específica.

Execução passo-a-passo dos primeiros e últimos passos.

A variável prompt é na verdade um conjunto de conjuntos de caracteres, o que é o

Passo	Linha	Instrução	Resultado			
1	5	int diasDaSemana[7]	diasDaSemana={?,?,?,?,?,?,?}			
2	6	int total=0, i;	diasDaSemana={?,?,?,?,?,?,?}	total=0	i=?	
3	8	char prompt[7][]={...};	diasDaSemana={?,?,?,?,?,?,?}	total=0	i=?	prompt={"Segunda:", "Terca:", ...}
4	19	for(i=0; #; #)	diasDaSemana={?,?,?,?,?,?,?}	total=0	i=0	prompt={"Segunda:", "Terca:", ...}
5	19	for(#; i<7; #)	diasDaSemana={?,?,?,?,?,?,?}	total=0	i=0	prompt={"Segunda:", "Terca:", ...}
6	21	printf(prompt[i]);	Segunda:	total=0	i=0	prompt={"Segunda:", "Terca:", ...}
7	22	scanf(...,&diasDaSemana[i]);	2	total=0	i=0	prompt={"Segunda:", "Terca:", ...}
8	19	for(#; #; ++)	diasDaSemana={2,?,?,?,?,?,?}	total=0	i=1	prompt={"Segunda:", "Terca:", ...}
9	19	for(#; i<7; #)	diasDaSemana={2,?,?,?,?,?,?}	total=0	i=1	prompt={"Segunda:", "Terca:", ...}
10	21	printf(prompt[i]);	Terca:	total=0	i=1	prompt={"Segunda:", "Terca:", ...}
11	22	scanf(...,&diasDaSemana[i]);	3	total=0	i=1	prompt={"Segunda:", "Terca:", ...}
12	19	for(#; #; ++)	diasDaSemana={2,3,?,?,?,?,?}	total=0	i=2	prompt={"Segunda:", "Terca:", ...}
13	19	for(#; i<7; #)	diasDaSemana={2,3,?,?,?,?,?}	total=0	i=2	prompt={"Segunda:", "Terca:", ...}
14	21	printf(prompt[i]);	Quarta	total=0	i=2	prompt={"Segunda:", "Terca:", ...}
15	22	scanf(...,&diasDaSemana[i]);	1	total=0	i=2	prompt={"Segunda:", "Terca:", ...}
16	19	for(#; #; ++)	diasDaSemana={2,3,1,?,?,?,?}	total=0	i=3	prompt={"Segunda:", "Terca:", ...}
17	19	for(#; i<7; #)	diasDaSemana={2,3,1,?,?,?,?}	total=0	i=3	prompt={"Segunda:", "Terca:", ...}
18	21	printf(prompt[i]);	Quinta	total=0	i=3	prompt={"Segunda:", "Terca:", ...}
19	22	scanf(...,&diasDaSemana[i]);	2	total=0	i=3	prompt={"Segunda:", "Terca:", ...}
20	19	for(#; #; ++)	diasDaSemana={2,3,1,2,?,?,?}	total=0	i=4	prompt={"Segunda:", "Terca:", ...}
Passo	Linha	Instrução	Resultado			
41	26	for(#; i<7; #)	2<7 Verdade	diasDaSemana={2,3,1,2,3,4,1}	total=5	i=2 prompt={"Segunda:", "Terca:", ...}
42	27	total+=diasDaSemana[i];		diasDaSemana={2,3,1,2,3,4,1}	total+=1=6	i=2 prompt={"Segunda:", "Terca:", ...}
43	26	for(#; #; ++)		diasDaSemana={2,3,1,2,3,4,1}	total=6	i=3 prompt={"Segunda:", "Terca:", ...}
44	26	for(#; i<7; #)	3<7 Verdade	diasDaSemana={2,3,1,2,3,4,1}	total=6	i=3 prompt={"Segunda:", "Terca:", ...}
45	27	total+=diasDaSemana[i];		diasDaSemana={2,3,1,2,3,4,1}	total+=2=8	i=3 prompt={"Segunda:", "Terca:", ...}
46	26	for(#; #; ++)		diasDaSemana={2,3,1,2,3,4,1}	total=8	i=4 prompt={"Segunda:", "Terca:", ...}
47	26	for(#; i<7; #)	4<7 Verdade	diasDaSemana={2,3,1,2,3,4,1}	total=8	i=4 prompt={"Segunda:", "Terca:", ...}
48	27	total+=diasDaSemana[i];		diasDaSemana={2,3,1,2,3,4,1}	total+=3=11	i=4 prompt={"Segunda:", "Terca:", ...}
49	26	for(#; #; ++)		diasDaSemana={2,3,1,2,3,4,1}	total=11	i=5 prompt={"Segunda:", "Terca:", ...}
50	26	for(#; i<7; #)	5<7 Verdade	diasDaSemana={2,3,1,2,3,4,1}	total=11	i=5 prompt={"Segunda:", "Terca:", ...}
51	27	total+=diasDaSemana[i];		diasDaSemana={2,3,1,2,3,4,1}	total+=4=15	i=5 prompt={"Segunda:", "Terca:", ...}
52	26	for(#; #; ++)		diasDaSemana={2,3,1,2,3,4,1}	total=15	i=6 prompt={"Segunda:", "Terca:", ...}
53	26	for(#; i<7; #)	6<7 Verdade	diasDaSemana={2,3,1,2,3,4,1}	total=15	i=6 prompt={"Segunda:", "Terca:", ...}
54	27	total+=diasDaSemana[i];		diasDaSemana={2,3,1,2,3,4,1}	total+=1=16	i=6 prompt={"Segunda:", "Terca:", ...}
55	26	for(#; #; ++)		diasDaSemana={2,3,1,2,3,4,1}	total=16	i=7 prompt={"Segunda:", "Terca:", ...}
56	26	for(#; i<7; #)	7<7 Falso	diasDaSemana={2,3,1,2,3,4,1}	total=16	i=7 prompt={"Segunda:", "Terca:", ...}
57	28	printf(...);	Soma: 16	diasDaSemana={2,3,1,2,3,4,1}	total=16	i=7 prompt={"Segunda:", "Terca:", ...}
58	29	printf(...);	Media: 2.285714	diasDaSemana={2,3,1,2,3,4,1}	total=16	i=7 prompt={"Segunda:", "Terca:", ...}

mesmo que dizer que é um **conjunto de strings**. Um exemplo de uma variável muito parecida com a prompt é o argumento **argv** que a função **main** pode ter, e pretende precisamente ter também um conjunto de strings que foram colocadas como argumentos do programa.

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     printf("Argumentos: %d\nArgumento 0: %s",argc,argv[0]);
6 }
```

Programa 7-8 Leitura de argumentos passados ao programa

Execução do programa:

```
C:\>arg sdfsd
Argumentos: 2
Argumento 0: arg
```

O programa imprime o primeiro argumento, que é o nome do próprio programa. A declaração de **argv** não tem dois pares de parênteses rectos vazios, dado que apenas pode haver um, que é o último, embora alguns compiladores o permitam mais que um, como é o caso do TCC. A alternativa aos parênteses rectos vazios é o asterisco antes. Outra declaração válida seria **char **argv**.

Nos restantes tipos não há problema em utilizar **vectores de vectores**, portanto **matrizes**, ou vectores de vectores de vectores. É normal referir-se a estes tipos como sendo **vectores multi-dimensionais**. Na linguagem C não há limite de dimensões pré-definido, mas não se aconselha a utilizar mais que 3 dimensões num vector. Vamos dar um exemplo da soma de duas matrizes.

```
1 #include <stdio.h>
2 #define SIZE 10
3
4 int main()
5 {
6     int matriz1[SIZE][SIZE];
7     int matriz2[SIZE][SIZE];
8     int matrizSoma[SIZE][SIZE];
9     int i,j;
10
11     /* inicializar matrizes */
12     for(i=0;i<SIZE;i++)
13         for(j=0;j<SIZE;j++)
14         {
15             matriz1[i][j]=i+j;
16             matriz2[i][j]=i*j;
17         }
18
19     /* somar matrizes */
20     for(i=0;i<SIZE;i++)
21         for(j=0;j<SIZE;j++)
22             matrizSoma[i][j]=matriz1[i][j]+matriz2[i][j];
23
24     /* mostrar resultado */
25     for(i=0;i<SIZE;i++)
26     {
27         /* mudar de linha */
28         printf("\n");
29         for(j=0;j<SIZE;j++)
30             printf("%3d ",matrizSoma[i][j]);
31     }
32 }
```

Programa 7-9 Soma de duas matrizes

Execução do programa:

C:\>*matriz*

```

0   1   2   3   4   5   6   7   8   9
1   3   5   7   9  11  13  15  17  19
2   5   8  11  14  17  20  23  26  29
3   7  11  15  19  23  27  31  35  39
4   9  14  19  24  29  34  39  44  49
5  11  17  23  29  35  41  47  53  59
6  13  20  27  34  41  48  55  62  69
7  15  23  31  39  47  55  63  71  79
8  17  26  35  44  53  62  71  80  89
9  19  29  39  49  59  69  79  89  99

```

O acesso a um valor da *matriz* (vector de vectores) é idêntico ao dos vectores, mas **em vez de um índice tem que se fornecer dois**, um para indicar o vector, e outro para indicar o elemento dentro desse vector.

No printf foi utilizado **%3d** e não **%d**, uma vez que se pretende que os números da matriz estejam alinhados, sendo importante que todos os números ocupem o mesmo espaço. Para indicar o número de caracteres a utilizar num número inteiro, pode-se colocar esse valor entre a % e o **d**.

Este exemplo começa com uma instrução de pré-processador ainda não conhecida, o **#define**. Estas instruções não são para o computador executar, mas sim para o compilador na altura em que está a processar o código do programa. A instrução é uma **macro** que indica que deve ser substituído o texto "SIZE" pelo valor "10" em todo o código onde o texto "SIZE" ocorra. Desta forma é possível alterar essa constante num só local, e essas alterações reflectirem-se em todo o código. É o que se vai fazer para poder apresentar uma execução passo-a-passo para este programa, caso contrário torna-se demasiado longo, em que foi alterado para **#define SIZE 2**.

Passo	Linha	Instrução	Resultado						
1	6	int matriz1[2][2];		matriz1=([?,?],[?,?])					
2	7	int matriz2[2][2];		matriz1=([?,?],[?,?])	matriz2=([?,?],[?,?])				
3	8	int matrizSoma[2][2];		matriz1=([?,?],[?,?])	matriz2=([?,?],[?,?])	matrizSoma=([?,?],[?,?])			
4	9	int i,j;		matriz1=([?,?],[?,?])	matriz2=([?,?],[?,?])	matrizSoma=([?,?],[?,?])	i=?	j=?	
5	12	for(i=0; #; #)		matriz1=([?,?],[?,?])	matriz2=([?,?],[?,?])	matrizSoma=([?,?],[?,?])	i=0	j=?	
6	12	for(;; i<2; #)	0<2 Verdade	matriz1=([?,?],[?,?])	matriz2=([?,?],[?,?])	matrizSoma=([?,?],[?,?])	i=0	j=?	
7	13	for(j=0; #; #)		matriz1=([?,?],[?,?])	matriz2=([?,?],[?,?])	matrizSoma=([?,?],[?,?])	i=0	j=0	
8	13	for(;; j<2; #)	0<2 Verdade	matriz1=([?,?],[?,?])	matriz2=([?,?],[?,?])	matrizSoma=([?,?],[?,?])	i=0	j=0	
9	15	matriz1[i][j]=i+j;		matriz1=([0,?],[?,?])	matriz2=([?,?],[?,?])	matrizSoma=([?,?],[?,?])	i=0	j=0	
10	16	matriz2[i][j]=i*j;		matriz1=([0,?],[?,?])	matriz2=([0,?],[?,?])	matrizSoma=([?,?],[?,?])	i=0	j=0	
11	13	for(;; #; j++)		matriz1=([0,?],[?,?])	matriz2=([0,?],[?,?])	matrizSoma=([?,?],[?,?])	i=0	j=1	
12	13	for(;; j<2; #)	1<2 Verdade	matriz1=([0,?],[?,?])	matriz2=([0,?],[?,?])	matrizSoma=([?,?],[?,?])	i=0	j=1	
13	15	matriz1[i][j]=i+j;		matriz1=([0,1],[?,?])	matriz2=([0,?],[?,?])	matrizSoma=([?,?],[?,?])	i=0	j=1	
14	16	matriz2[i][j]=i*j;		matriz1=([0,1],[?,?])	matriz2=([0,0],[?,?])	matrizSoma=([?,?],[?,?])	i=0	j=1	
15	13	for(;; #; j++)		matriz1=([0,1],[?,?])	matriz2=([0,0],[?,?])	matrizSoma=([?,?],[?,?])	i=0	j=2	
16	13	for(;; j<2; #)	2<2 Falso	matriz1=([0,1],[?,?])	matriz2=([0,0],[?,?])	matrizSoma=([?,?],[?,?])	i=1	j=2	
17	12	for(;; #; i++)		matriz1=([0,1],[?,?])	matriz2=([0,0],[?,?])	matrizSoma=([?,?],[?,?])	i=1	j=2	
18	12	for(;; i<2; #)	1<2 Verdade	matriz1=([0,1],[?,?])	matriz2=([0,0],[?,?])	matrizSoma=([?,?],[?,?])	i=1	j=2	
Passo	Linha	Instrução	Resultado						
21	15	matriz1[i][j]=i+j;		matriz1=([0,1],[1,?])	matriz2=([0,0],[?,?])	matrizSoma=([?,?],[?,?])	i=1	j=0	
22	16	matriz2[i][j]=i*j;		matriz1=([0,1],[1,?])	matriz2=([0,0],[0,?])	matrizSoma=([?,?],[?,?])	i=1	j=0	
23	13	for(;; #; j++)		matriz1=([0,1],[1,?])	matriz2=([0,0],[0,?])	matrizSoma=([?,?],[?,?])	i=1	j=1	
24	13	for(;; j<2; #)	1<2 Verdade	matriz1=([0,1],[1,?])	matriz2=([0,0],[0,?])	matrizSoma=([?,?],[?,?])	i=1	j=1	
25	15	matriz1[i][j]=i+j;		matriz1=([0,1],[1,2])	matriz2=([0,0],[0,?])	matrizSoma=([?,?],[?,?])	i=1	j=1	
26	16	matriz2[i][j]=i*j;		matriz1=([0,1],[1,2])	matriz2=([0,0],[0,1])	matrizSoma=([?,?],[?,?])	i=1	j=1	
27	13	for(;; #; j++)		matriz1=([0,1],[1,2])	matriz2=([0,0],[0,1])	matrizSoma=([?,?],[?,?])	i=1	j=2	
28	13	for(;; j<2; #)	2<2 Falso	matriz1=([0,1],[1,2])	matriz2=([0,0],[0,1])	matrizSoma=([?,?],[?,?])	i=1	j=2	
29	12	for(;; #; i++)		matriz1=([0,1],[1,2])	matriz2=([0,0],[0,1])	matrizSoma=([?,?],[?,?])	i=2	j=2	
30	12	for(;; i<2; #)	2<2 Falso	matriz1=([0,1],[1,2])	matriz2=([0,0],[0,1])	matrizSoma=([?,?],[?,?])	i=2	j=2	
31	20	for(i=0; #; #)		matriz1=([0,1],[1,2])	matriz2=([0,0],[0,1])	matrizSoma=([?,?],[?,?])	i=0	j=2	
32	20	for(;; i<2; #)	0<2 Verdade	matriz1=([0,1],[1,2])	matriz2=([0,0],[0,1])	matrizSoma=([?,?],[?,?])	i=0	j=2	
33	21	for(j=0; #; #)		matriz1=([0,1],[1,2])	matriz2=([0,0],[0,1])	matrizSoma=([?,?],[?,?])	i=0	j=0	
34	21	for(;; j<2; #)	0<2 Verdade	matriz1=([0,1],[1,2])	matriz2=([0,0],[0,1])	matrizSoma=([?,?],[?,?])	i=0	j=0	
35	15	matrizSoma[i][j]=matriz1[i][j]+matriz2[i][j];		matriz1=([0,1],[1,2])	matriz2=([0,0],[0,1])	matrizSoma=([0,?],[?,?])	i=0	j=0	
36	21	for(;; #; j++)		matriz1=([0,1],[1,2])	matriz2=([0,0],[0,1])	matrizSoma=([0,?],[?,?])	i=0	j=1	
37	21	for(;; j<2; #)	1<2 Verdade	matriz1=([0,1],[1,2])	matriz2=([0,0],[0,1])	matrizSoma=([0,?],[?,?])	i=0	j=1	
38	15	matrizSoma[i][j]=matriz1[i][j]+matriz2[i][j];		matriz1=([0,1],[1,2])	matriz2=([0,0],[0,1])	matrizSoma=([0,1],[?,?])	i=0	j=1	
39	21	for(;; #; j++)		matriz1=([0,1],[1,2])	matriz2=([0,0],[0,1])	matrizSoma=([0,1],[?,?])	i=0	j=2	
40	21	for(;; j<2; #)	2<2 Falso	matriz1=([0,1],[1,2])	matriz2=([0,0],[0,1])	matrizSoma=([0,1],[?,?])	i=0	j=2	

O valor máximo das dimensões dos vectores, strings, matrizes, é normal definir-se desta forma, através de macros. Isto porque não podem ser utilizadas variáveis nas dimensões dos vectores, até porque as declarações têm de estar no início das funções, e nessa altura pode não estar ainda calculada a dimensão que é necessária. Tem que se utilizar um valor maior que o necessário, e se houver problemas, altera-se a macro e fica todo o código actualizado. No módulo 3 vamos saber como alocar memória indicando a dimensão necessária em tempo de execução.

Passo	Linha	Instrução	Resultado					
41	20	for(#; #; i++)		matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[?,?]]	i=1	j=2
42	20	for(#; i<2; #)	1<2 Verdade	matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[?,?]]	i=1	j=2
43	21	for(j=0; #; #)		matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[?,?]]	i=1	j=0
44	21	for(#; j<2; #)	0<2 Verdade	matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[?,?]]	i=1	j=0
45	15	matrizSoma[i][j]=matriz1[i][j]+matriz2[i][j];		matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,?]]	i=1	j=0
46	21	for(#; #; j++)		matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,?]]	i=1	j=1
47	21	for(#; j<2; #)	1<2 Verdade	matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,?]]	i=1	j=1
48	15	matrizSoma[i][j]=matriz1[i][j]+matriz2[i][j];		matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=1	j=1
49	21	for(#; #; i++)		matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=1	j=2
50	21	for(#; j<2; #)	2<2 Falso	matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=1	j=2
51	20	for(#; #; i++)		matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=1	j=2
52	20	for(#; i<2; #)	2<2 Falso	matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=2	j=2
53	25	for(i=0; #; #)		matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=0	j=2
54	25	for(#; i<2; #)	0<2 Verdade	matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=0	j=2
55	28	printf("\n");	\n	matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=0	j=2
56	29	for(j=0; #; #)		matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=0	j=0
Passo	Linha	Instrução	Resultado					
61	30	printf(...);	1	matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=0	j=1
62	29	for(#; #; j++)		matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=0	j=2
63	29	for(#; j<2; #)	2<2 Falso	matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=0	j=2
64	25	for(#; #; i++)		matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=1	j=2
65	25	for(#; i<2; #)	1<2 Verdade	matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=1	j=2
66	28	printf("\n");	\n	matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=1	j=2
67	29	for(j=0; #; #)		matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=1	j=0
68	29	for(#; j<2; #)	0<2 Verdade	matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=1	j=0
69	30	printf(...);	1	matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=1	j=0
70	29	for(#; #; j++)		matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=1	j=1
71	29	for(#; j<2; #)	1<2 Verdade	matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=1	j=1
72	30	printf(...);	3	matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=1	j=1
73	29	for(#; #; i++)		matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=1	j=2
74	29	for(#; j<2; #)	2<2 Falso	matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=1	j=2
75	25	for(#; #; i++)		matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=2	j=2
76	25	for(#; i<2; #)	2<2 Falso	matriz1=[[0,1],[1,2]]	matriz2=[[0,0],[0,1]]	matrizSoma=[[0,1],[1,3]]	i=2	j=2

Erros comuns mais directamente associados a este capítulo:

- **Variáveis com nomes quase iguais**

- **Forma:** Algumas variáveis necessárias têm na verdade o mesmo nome, pelo que se utiliza os nomes nomeA, nomeB, nomeC, nomeD, ou nome1, nome2, nome3, nome4
- **Problema:** Esta situação é uma indicação clara que necessita de um vector nome[4]. Se não utilizar o vector não é possível depois fazer alguns ciclos a iterar pelas variáveis, e irão aparecer forçosamente instruções idênticas repetidas por cada variável
- **Resolução:** Deve nesta situação procurar alternativas de fazer código em que esta situação lhe tenha ocorrido. Não utilize código de outra pessoa, apenas o seu código com este tipo de erro lhe poderá ser útil

- **Má utilização do switch ou cadeia if-else**

- **Forma:** De forma a executar instruções dependente do valor de um índice, utilizar um switch colocando as diversas instruções para cada caso do índice
- **Problema:** Muitas vezes ao utilizar no switch um índice (ou cadeia if-else), as instruções em cada caso ficam muito parecidas, ficando com código longo e de custo de manutenção elevado. Se essa situação ocorrer deve rever o seu código, dado que está provavelmente a falhar uma alternativa mais potente, que é a utilização de vectores
- **Resolução:** Criar um vector com as constantes necessárias para chamar numa só instrução todas as outras dependentes do índice

- **Macros em todas as constantes**

- **Forma:** Tudo o que é constante tem direito a uma macro, de forma a poder ser facilmente alterada
- **Problema:** A utilização de macros tem um custo, que incorre nas linhas de código onde é utilizada: não é conhecido o valor real da macro. Se esta for utilizada em situações em que o fluxo de informação seja claro, como um valor inteiro limite sendo os ciclos desde uma constante até esse limite, não há qualquer problema. Se no entanto for utilizada em locais dependentes de outras constantes, sejam em macros sejam no código, ou que seja importante saber o valor da macro para saber qual a sequência de instruções, ou num caso extremo, o valor da macro é necessário para saber se uma instrução é válida, evidentemente que o valor da macro em si não pode ser alterado sem que o código seja revisto. Perde portanto o sentido, uma vez que prejudica a leitura do código e não facilita qualquer posterior alteração. As macros devem ser independentes entre si e de outras constantes
- **Resolução:** Remova do código as macros que são dependentes entre si, ou que dificilmente podem mudar sem implicar outras alterações no código

O exemplo do Programa 7-1 para motivar a utilização de vectores é um exemplo da ocorrência do primeiro erro, variáveis com nomes quase iguais, neste caso relacionados. Por vezes a evidência é superior, sendo utilizadas várias variáveis do mesmo tipo e com o mesmo prefixo. As versões da função `DiasDoMes` anteriores à apresentada no Programa 7-5 foram utilizadas para introduzir diversos conceitos, mas são na verdade ocorrências do erro da má utilização do `switch` ou cadeia `if-else`. Têm no entanto a atenuante de resultar em apenas 3 situações, e não um valor distinto por cada dia do mês. Sobre o erro das macros, vamos dar uma versão do Programa 7-9 com um número exagerado de erros deste tipo para melhor ilustrar o custo da utilização das macros.

```

1 #include <stdio.h>
2 #define SIZE 10
3 #define INICIO 0
4 #define NOVALINHA "\n"
5 #define IMPRIMEVALOR "%3d "
6 #define OPERACAO1 i+j;
7 #define OPERACAO2 i*j;
8 #define OPERACAO3 matriz1[i][j]+matriz2[i][j];
9 #define ACESSO(a,b) matrizSoma[a][b]
10 #define ITERAR_i for(i=INICIO;i<SIZE;i++)
11 #define ITERAR_j for(j=INICIO;j<SIZE;j++)
12 #define ITERAR_ij ITERAR_i ITERAR_j
13
14 int main()
15 {
16     int matriz1[SIZE][SIZE];
17     int matriz2[SIZE][SIZE];
18     int matrizSoma[SIZE][SIZE];
19     int i,j;
20
21     /* inicializar matrizes */
22     ITERAR_ij
23     {
24         matriz1[i][j]=OPERACAO1
25         matriz2[i][j]=OPERACAO2
26     }
27
28     /* somar matrizes */
29     ITERAR_ij
30         matrizSoma[i][j]=OPERACAO3
31
32     /* mostrar resultado */
33     ITERAR_i

```

```
34     {  
35         /* mudar de linha */  
36         printf(NOVALINHA);  
37         ITERAR_j  
38         printf(IMPRIMEVALOR, ACESSO(i,j));  
39     }  
40 }
```

As macros estão definidas no início do programa, e por ordem de gravidade de erro. A primeira, `SIZE 10`, é aconselhada, resulta em apenas vantagens dado que a constante ao ser alterada não influencia em nada o bom funcionamento do programa. A segunda macro, `INICIO 0`, cai no erro dado que não pode e não faz sentido ter um valor distinto do zero, o primeiro índice de um vector é o zero, faz parte da linguagem de programação. Utilizar esta macro apenas dá flexibilidade para cometer um erro, e ao ler o código não se percebe no local que o processamento é desde o início da matriz.

As duas macros `NOVALINHA` e `IMPRIMEVALOR`, colocando todas as strings do programa em macros, são até um procedimento normal para programas com diversas línguas, em que as strings têm de ser traduzidas e colocadas num local à parte do código, não utilizando obrigatoriamente macros. No entanto as strings colocadas em macros não têm texto para traduzir, e a sua alteração pode levar a que o ciclo final não funcione, e quem lê os `printfs` no código, não sabe o que estes fazem até conhecer o valor das macros.

As macros `OPERACAO1/2/3` são já uma agressão ao leitor, dado que acabam por ser uma espécie de funções sem abstracção, que utilizam as variáveis do local onde são chamadas. A macro `ACESSO(a,b)`, com argumentos, é uma forma curta de aceder a uma variável que já está declarada, e cujo acesso é tão simples quanto possível, pelo que se pode considerar uma reformatação da linguagem de programação. Deve-se utilizar o facto de o leitor conhecer a sintaxe da linguagem C, e não reformatar a linguagem para aceder neste caso a um elemento de matriz. Soma de forma alternativa. Finalmente, acaba-se com o pior que se pode fazer utilizando macros, as macros `ITERAR`, que encapsulam os próprios ciclos. Para este código ser lido ter-se-ia de andar com os olhos para cima e para baixo para desvendar o que faz. Uma situação extrema pode-se com as macros tornar uma outra linguagem de programação válida, tendo o leitor que ao ler as macros aprender a sintaxe dessa linguagem de forma a poder ler o programa.

Está introduzido o principal conceito em estruturas de dados, os vectores. Com base neste conceito será possível abordar uma maior variedade de problemas. Tal como o conceito de variável, a determinação dos vectores necessários para resolver um problema, aparentemente simples, é das partes mais complexas e importantes a decidir antes de começar a escrever um programa. Irá aperceber-se desse problema quando for necessário utilizar vectores auxiliares, que não sejam dados de entrada e/ou saída do problema em questão.

8. PROCEDIMENTOS

Como vimos no capítulo de Funções, uma função permite definir uma abstracção no cálculo de um valor. Quando se implementa a função, apenas nos preocupamos como com os argumentos recebidos se calcula o valor pretendido, abstraindo das possíveis utilizações da função. Quando se utiliza a função, apenas nos preocupamos com a passagem dos valores correctos à função, de forma a obter o valor pretendido calculado, abstraindo da forma como esta está implementada.

O conceito de "**Procedimento**" é idêntico ao da função, mas na verdade não calcula um valor, mas sim **executa um determinado procedimento**, não retornando nenhum valor. Esta pequena diferença relativamente ao conceito função, não conduz a grande distinção da forma como são implementados procedimentos ou funções, não merecendo na linguagem C uma distinção, muito embora exista em outras linguagens. A distinção é feita aqui a nível de conceito.

Um **procedimento**, **faz algo**, logo o seu nome tem de estar **associado a um verbo**. Uma **função calcula algo**, logo o seu nome tem de estar **associado à grandeza** calculada.

Um **procedimento** é chamado como uma **instrução isolada**, uma **função** é chamada **numa expressão**, nem que a expressão seja apenas a função, para que o valor retornado seja utilizado ou numa atribuição a uma variável, ou num argumento de uma função/procedimento.

Vamos ver um exemplo. Precisamos de fazer uma função para implementar o primeiro exemplo dado no capítulo Variáveis, a troca do valor de duas variáveis. Esta função não deve retornar nada, deve apenas trocar o valor das variáveis. Qual deverá ser o seu valor de retorno? Qual deverá ser o seu nome? A resposta a ambas as perguntas é simples: não tem valor de retorno; o nome deve conter o verbo da acção efectuada, ou seja "Trocar" ou se preferir "Troca". Trata-se de um procedimento.

```
1 #include <stdio.h>
2
3 int Troca(int x, int y)
4 {
5     int aux=x;
6     x=y;
7     y=aux;
8 }
9
10 int main()
11 {
12     int x=123, y=321;
13     Troca(x,y);
14     printf("x: %d y: %d",x,y);
15 }
```

Programa 8-1 Tentativa de implementação de uma função para trocar o valor de duas variáveis

Este programa tem uma função Troca, que é um procedimento, pelo que se chamou numa instrução isolada, o seu valor de retorno seria desperdiçado caso existisse. Em todo o caso não retorna nada, apenas tem o código de troca de duas variáveis, o x e o y.

Execução do programa:

```
C:\>troca
x: 123 y: 321
```

A execução do programa não correu como o previsto, a variável *x* continuou com 123, e a variável *y* com 321. Porquê? Apenas se trocou os valores das variáveis *x* e *y* da função *Troca*, mas os valores que estamos interessados é trocar os valores das variáveis *x* e *y* da função *main*.

Pode-se da função *main* chamar *Troca(2*x,10*y)*; e nessa altura o que é que trocava de valor? A função *Troca* não conhece nem tem de conhecer as expressões utilizadas para a chamar, só sabe que os valores passados estão nas variáveis locais *x* e *y*. Apenas retornará um valor, e esse valor será a sua contribuição para o programa.

Passo	Linha	Instrução	Resultado						
1	12	int x=123, y=321;		x=123	y=321				
2	13	Troca(x,y)		x=123	y=321	Troca			
3	3	int x=123, y=321;		x=123	y=321	Troca	x=123	y=321	
4	5	int aux=x;		x=123	y=321	Troca	x=123	y=321	aux=123
5	6	x=y;		x=123	y=321	Troca	x=321	y=321	aux=123
6	7	y=aux;		x=123	y=321	Troca	x=321	y=123	aux=123
7	14	printf(...,x,y);	x: 123 y: 321	x=123	y=321				

Temos de primeiro ter uma forma de dizer que a função *Troca* não retorna um valor do tipo inteiro, nem qualquer outro. Se omitir o tipo de retorno a linguagem C assume que é inteiro, pelo que é necessário utilizar um outro tipo disponível, o tipo **void**. Este tipo não é tipo de variável nenhum, ideal para os procedimentos em que não se pretende retornar nada.

Em segundo lugar temos de dar permissão à função *Troca* para alterar os valores das variáveis *x* e *y* da função *main*. Não é no entanto necessário um comando extra da linguagem C, na verdade isso já foi feito no capítulo Vectores, no exemplo `int strlen(char str[])`, em que passamos uma string para um argumento. Foi passado na verdade o endereço do vector, e a função conseguiu aceder aos valores de todos os elementos do vector. Neste caso não temos vector, mas como um vector é um conjunto de elementos, um elemento também é um vector.

```

1 #include <stdio.h>
2
3 void Troca(int x[], int y[])
4 {
5     int aux=x[0];
6     x[0]=y[0];
7     y[0]=aux;
8 }
9
10 int main()
11 {
12     int x[1]={123}, y[1]={321};
13     Troca(x,y);
14     printf("x: %d y: %d",x[0],y[0]);
15 }
```

Programa 8-2 Função Troca que realmente troca o valor de duas variáveis

Execução do programa:

```

C:\>troca2
x: 321 y: 123
```

O programa funcionou como esperado. No entanto teve de haver algum cuidado ao declarar as variáveis como vectores de uma unidade na função *main*, para ter acesso ao

Passo	Linha	Instrução	Resultado						
1	12	int x[1]={123}, y[1]={321};		x={123}	y={321}				
2	13	Troca		x={123}	y={321}	Troca			
3	3	int x[]=x, y[]=y;		x={123}	y={321}	Troca	x=x	y=y	
4	5	int aux=x[0];		x={123}	y={321}	Troca	x=x	y=y	aux=123
5	6	x[0]=y[0];		x={321}	y={321}	Troca	x=x	y=y	aux=123
6	7	y[0]=aux;		x={321}	y={123}	Troca	x=x	y=y	aux=123
7	14	printf(...);	x: 321 y: 123	x={321}	y={123}				

endereço das variáveis *x* e *y*. Esta situação resolve-se com o **operador &** da linguagem C, que **aplicado a uma variável retorna o seu endereço**, que é precisamente o que necessitamos para não ter de declarar um vector de uma variável.

```
10 int main()
11 {
12     int x=123, y=321;
13     Troca(&x,&y);
14     printf("x: %d y: %d",x,y);
15 }
```

Falta ainda na função *Troca*, uma forma mais simples de aceder ao primeiro elemento, até porque não há mais elementos, apenas é pretendido o acesso ao valor do primeiro elemento. A linguagem C tem também o **operador ***, que **aplicado a um endereço, retorna o valor nesse endereço**:

```
3 void Troca(int x[], int y[])
4 {
5     int aux = *x;
6     *x = *y;
7     *y = aux;
8 }
```

Os argumentos também sugerem que se trata de um vector *x* e *y*, enquanto se pretende apenas o endereço de duas variáveis, ou se quiser um vector unitário. Já vimos que o `int x[]` é equivalente a `int *x`, e é precisamente esta uma das situações mais aconselhável para a segunda versão, de forma a dar indicações ao leitor que se pretende apenas aceder ao valor no endereço e não a um conjunto de valores a começar nesse endereço. No entanto não é demais realçar que para a linguagem C esta diferença não tem significado. Vejamos então a versão final.

```
1 #include <stdio.h>
2
3 void Troca(int *x, int *y)
4 {
5     int aux = *x;
6     *x = *y;
7     *y = aux;
8 }
9
10 int main()
11 {
12     int x=123, y=321;
13     Troca(&x,&y);
14     printf("x: %d y: %d",x,y);
15 }
```

Programa 8-3 Versão final da função de troca de duas variáveis

A este tipo de passagem de parâmetros é designada de **passagem por referência**, uma vez que se envia a referência da variável, e não a própria variável, para que o conteúdo da variável possa ser alterado.

Passo	Linha	Instrução	Resultado						
1	12	int x=123, y=321;		x=123	y=321				
2	13	Troca(&x,&y);		x=123	y=321	Troca			
3	3	int *x=&x, *y=&y;		x=123	y=321	Troca	x=&x	y=&y	
4	5	int aux=*x;		x=123	y=321	Troca	x=&x	y=&y	aux=123
5	6	*x=*y;		x=321	y=321	Troca	x=&x	y=&y	aux=123
6	7	*y=aux;		x=321	y=123	Troca	x=&x	y=&y	aux=123
7	14	printf(...);	x: 321 y: 123	x=321	y=123				

Todos os procedimentos devem ter algum parâmetro passado por variáveis por referência, caso contrário não há qualquer efeito da sua chamada. Há no entanto duas situações: a sua acção incide sobre recursos externos, como imprimir uma mensagem no ecrã; altera valores de variáveis globais.

A linguagem C permite que existam declarações não no início de funções, mas sim fora delas. As variáveis declaradas fora das funções são **variáveis globais**, criadas no início do programa, e **acessíveis a todas as funções**. Vejamos outra versão da troca de duas variáveis.

```

1 #include <stdio.h>
2
3 int x=123, y=321;
4
5 void Troca()
6 {
7     int aux=x;
8     x=y;
9     y=aux;
10 }
11
12 int main()
13 {
14     Troca();
15     printf("x: %d y: %d",x,y);
16 }

```

Programa 8-4 Exemplo de utilização de variáveis globais

Neste caso o procedimento Troca nem sequer necessita de argumentos, dado que vai trocar o valor de x e y, e são exactamente essas variáveis que são impressas no printf do main.

O primeiro passo não é na primeira linha da função main, mas sim na declaração das variáveis globais. Estas variáveis estão fora da função main, e acessíveis a todas as funções. O passo 2 chama a função main. Nas execuções passo-a-passo sem variáveis globais, assume-se que a função main acabou de ser chamada antes do primeiro passo.

Passo	Linha	Instrução	Resultado					
1	3	int x=123, y=321;		x=123	y=321			
2	12	main();		x=123	y=321	main		
3	14	Troca();		x=123	y=321	main	Troca	
4	7	int aux=x;		x=123	y=321	main	Troca	aux=123
5	5	x=y;		x=321	y=321	main	Troca	aux=123
6	6	y=aux		x=321	y=123	main	Troca	aux=123
7	15	printf(...);	x: 321 y: 123	x=321	y=123	main		

Pode-se pensar que assim é melhor, dado que nem sequer é preciso passar argumentos. Nada mais errado. Nesta versão do programa, a função Troca ao perder os argumentos, perdeu generalidade, apenas pode trocar o valor de x e y, quando anteriormente poderia trocar o valor de quaisquer duas variáveis inteiras. Por outro lado, como as variáveis x e y estão disponíveis em todo o programa, tanto a utilização como a atribuição dos valores destas variáveis têm de considerar todas as restantes linhas de código. Pode uma zona do código estar a fazer uma utilização a estas variáveis, e outra zona de código uma utilização diferente. Não é portanto possível que ao implementar uma função que utilize estas variáveis, abstrair-se do resto do programa, a vantagem da abstracção funcional. Nos outros exemplos, cada uma das variáveis declaradas apenas eram utilizadas/alteradas com instruções na função onde foram declaradas, não sendo necessário considerar o resto do código para utilizar/alterar cada variável.

As **variáveis globais são desaconselhadas** dado que quebram a abstracção funcional, mas por vezes são necessárias por motivos práticos. Existe no entanto uma forma para em certas situações utilizar-se uma "falsa" variável global, que na verdade é uma variável global mas apenas acessível a uma só função, não quebrando desta forma a abstracção funcional. Trata-se de **variáveis estáticas (static)**, que podem ser declaradas numa função, mas mantêm o valor entre chamadas, que é o mesmo que uma variável global.


```
1 #include <stdio.h>
2
3 int Troca(int *x, int *y)
4 {
5     static int trocas=0;
6     int aux = *x;
7     *x = *y;
8     *y = aux;
9     return ++trocas;
10 }
11
12 int main()
13 {
14     int x=123, y=321;
15     printf("Troca %d: x=%d, y=%d\n", Troca(&x,&y),x,y);
16     printf("Troca %d: x=%d, y=%d\n", Troca(&x,&y),x,y);
17     printf("Troca %d: x=%d, y=%d\n", Troca(&x,&y),x,y);
18 }
```

Programa 8-5 Exemplo de utilização de variáveis estáticas (“static”)

Execução do programa:

```
C:\>troca
Troca 1: x=321, y=123
Troca 2: x=123, y=321
Troca 3: x=321, y=123
```

Neste programa troca-se o valor das variáveis `x` e `y` várias vezes. A função `Troca` retorna agora o número de trocas que já efectuou. Tem uma variável declarada com `static`, o que significa que é uma variável estática, mantendo o valor entre chamadas. Pode assim ser utilizada para contador, sendo o valor dessa variável o valor retornado. A função `printf` chama a função `Troca` de forma a imprimir esse valor, e depois utiliza o valor de `x` e `y`, entretanto já alterados pela função `Troca`. Ao executar 3 vezes, podemos ver que o contador está a funcionar e que os valores das variáveis vão trocando.

A execução passo-a-passo trata as variáveis estáticas da mesma forma que as variáveis globais, muito embora estejam acessíveis apenas para as funções onde estão declaradas. A linha 5 nunca é executada quando é chamada a função Troca, mas sim no início da execução.

A utilização de variáveis estáticas não deve estar ligada com a funcionalidade pretendida. Contadores do número de chamadas a determinada função é um bom exemplo, mas se estiver a tentar utilizá-la como uma funcionalidade pretendida, desaconselha-se.

Passo	Linha	Instrução	Resultado									
1	5	static int trocas=0;		(Troca) trocas=0								
2	12	int main();		(Troca) trocas=0	main							
3	14	int x=123, y=321;		(Troca) trocas=0	main	x=123	y=321					
4	15	printf(...);		(Troca) trocas=0	main	x=123	y=321	Troca				
5	3	int *x=&x, *y=&y;		(Troca) trocas=0	main	x=123	y=321	Troca	x=&x	y=&y		
6	6	int aux=x;		(Troca) trocas=0	main	x=123	y=321	Troca	x=&x	y=&y	aux=123	
7	7	*x=*y;		(Troca) trocas=0	main	x=321	y=321	Troca	x=&x	y=&y	aux=123	
8	8	*y=aux;		(Troca) trocas=0	main	x=321	y=123	Troca	x=&x	y=&y	aux=123	
9	9	return ++trocas;		(Troca) trocas=1	main	x=321	y=123	Troca	x=&x	y=&y	aux=123	
10	15	printf(...);	Troca 1: x:=321, y= 123	(Troca) trocas=1	main	x=321	y=123					
11	16	printf(...);		(Troca) trocas=1	main	x=321	y=123	Troca				
12	3	int *x=&x, *y=&y;		(Troca) trocas=1	main	x=321	y=123	Troca	x=&x	y=&y		
13	6	int aux=x;		(Troca) trocas=1	main	x=321	y=123	Troca	x=&x	y=&y	aux=321	
14	7	*x=*y;		(Troca) trocas=1	main	x=123	y=123	Troca	x=&x	y=&y	aux=321	
15	8	*y=aux;		(Troca) trocas=1	main	x=123	y=321	Troca	x=&x	y=&y	aux=321	
16	9	return ++trocas;		(Troca) trocas=2	main	x=123	y=321	Troca	x=&x	y=&y	aux=321	
17	16	printf(...);	Troca 2: x:=123, y= 321	(Troca) trocas=2	main	x=123	y=321					
18	17	printf(...);		(Troca) trocas=2	main	x=123	y=321	Troca				
19	3	int *x=&x, *y=&y;		(Troca) trocas=2	main	x=123	y=321	Troca	x=&x	y=&y		
20	6	int aux=x;		(Troca) trocas=2	main	x=123	y=321	Troca	x=&x	y=&y	aux=123	
21	7	*x=*y;		(Troca) trocas=2	main	x=321	y=321	Troca	x=&x	y=&y	aux=123	
22	8	*y=aux;		(Troca) trocas=2	main	x=321	y=123	Troca	x=&x	y=&y	aux=123	
23	9	return ++trocas;		(Troca) trocas=3	main	x=321	y=123	Troca	x=&x	y=&y	aux=123	
24	17	printf(...);	Troca 3: x:=321, y= 123	(Troca) trocas=3	main	x=321	y=123					

Não se pode acabar esta secção sem referir que o que se acabou de fazer com a passagem de argumentos por referência foi introduzir **apontadores** na linguagem C. As declarações na função Troca com argumentos, podem estar na declaração das variáveis locais de qualquer função. Podemos ter apontadores para qualquer variável, e fazer operações sobre apontadores, como incrementar, passando o apontador para o endereço da variável seguinte, ou somar, saltando-se tantos endereços como o valor somado.

Vamos exemplificar implementando uma função que recebe duas strings, e retorna um apontador para a primeira ocorrência de uma string na outra (**strstr**). Esta função faz parte da biblioteca **string.h**, e pode ser utilizada tal como a **strlen** já referida, às quais se juntam também as funções **strcpy** e **strcat** de forma a se obter o conjunto de funções essenciais sobre strings. A primeira, **strcpy**, copia o conteúdo de uma string para a outra, e a segunda, **strcat**, adiciona o conteúdo de uma string na outra. Todas as funções da **string.h** assumem que foi declarado uma string com espaço suficiente para as operações que executam, e que as strings terminam com zero.

```
1 #include <stdio.h>
2
3 char *strstr(char *find, char *str)
4 {
5     int i;
6     /* percorrer toda a string str */
7     while(*str!=0)
8     {
9         /* tentar fazer match a começar em str */
10        for(i=0; str[i]!=0 && find[i]!=0 && find[i]==str[i]; i++);
11        /* se chegou ao fim da string de procura, então há match */
12        if(find[i]==0)
13            /* retornar o início do match */
14            return str;
15        /* incrementar o início da string */
16        str++;
17    }
18
19    return NULL;
20 }
21
22 int main()
23 {
24     char str[]="percorrer toda a string str";
25     char find[10];
26     char *resultado;
27
28     printf("Introduza a string de procura: ");
29     gets(find);
30     resultado=strstr(find,str);
31     printf("Resultado: %s\n",resultado);
32 }
```

Programa 8-6 Procura a primeira ocorrência de uma string em outra (strstr)

Repare-se que se utiliza o facto de **str** ser um apontador para o início da string, para no ciclo principal servir desta própria variável como variável iteradora. É procurado no segundo ciclo um match entre **find** e os primeiros caracteres de **str**. Se tal não for conseguido, incrementa-se o início da string (variável **str**) e repete-se o ciclo.

Na função **main** guarda-se o resultado da chamada à função numa variável declarada como apontador **resultado**. Poder-se-ia ter chamado a função **strstr** logo na função **printf**, mas pretendeu-se ilustrar que se pode guardar um endereço sem problema, neste caso com a primeira ocorrência de **find** em **str**.

Execução do programa:

```
C:\>strstr
Introduza a string de procura: str
Resultado: string str
```

```
C:\>strstr
Introduza a string de procura: abc
Resultado: (null)
```

Como se pode ver na primeira execução do programa, o resultado foi a impressão da primeira ocorrência até ao final da string, uma vez que uma string apenas acaba com o carácter zero. O segundo caso está relacionado com o valor retornado da função, em caso de não encontrar nada. Retorna **NULL**, que é o endereço vazio. Quando não se pretende atribuir nenhum valor a um apontador, deve-se utilizar esta constante, que pertence à linguagem C.

O programa tentou imprimir uma string a apontar para NULL na segunda execução, o que não deve ser feito, atendendo a que NULL é o endereço vazio e portanto não tem caracteres. Por motivos práticos, a string "(null)" aparecerá como resultado desta operação na maior parte dos compiladores actuais. Igualmente válido embora não pertencente à linguagem C, será considerar o valor de NULL como sendo zero, podendo-se fazer uso deste facto em expressões lógicas omitindo operadores lógicos e considerando NULL como falso e todos os restantes endereços como verdadeiro, mas desaconselha-se esse tipo de utilização pelos motivos já explicados no capítulo Condicionais.

A execução passo-a-passo é bastante extensa, está aqui uma versão cortada, mas ainda assim pode-se verificar o funcionamento de um programa com apontadores. Nestes programas move-se o

Passo	Linha	Instrução	Resultado						
1	24	char str[]="percorrer...";		str="percorrer..."					
2	25	char find[10];		str="percorrer..."	find=""				
3	26	char *resultado;		str="percorrer..."	find=""	resultado?			
4	28	printf(...);	Introduza a string...	str="percorrer..."	find=""	resultado?			
5	29	gets(find);	str	str="percorrer..."	find="str"	resultado?			
6	30	resultado=strstr(find,str);		str="percorrer..."	find="str"	resultado?	strstr		
7	3	char *find=find, char *str=str;		str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str
8	5	int i;		str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str i=?
9	7	while(*str!=0)	'p'!=0 Verdade	str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str i=?
10	10	for(i=0; #; #)		str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str i=0
11	10	for(;; str[i] != 0 && ...; #)	'p'!=0 && 's'!=0 && 's'=='p' Falso	str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str i=0
12	12	if(find[i]==0)	's'==0 Falso	str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str i=0
13	16	str++;		str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str+1 i=0
14	7	while(*str!=0)	'e'!=0 Verdade	str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str+1 i=0
15	10	for(i=0; #; #)		str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str+1 i=0
16	10	for(;; str[i] != 0 && ...; #)	'e'!=0 && 's'!=0 && 's'=='e' Falso	str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str+1 i=0
17	12	if(find[i]==0)	's'==0 Falso	str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str+1 i=0
18	16	str++;		str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str+2 i=0
...									
Passo	Linha	Instrução	Resultado						
...									
1001	7	while(*str!=0)	's'!=0 Verdade	str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str+17 i=0
1002	10	for(i=0; #; #)		str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str+17 i=0
1003	10	for(;; str[i] != 0 && ...; #)	's'!=0 && 's'!=0 && 's'=='s' Verdade	str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str+17 i=0
1004	10	for(;; i++)		str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str+17 i=1
1005	10	for(;; str[i] != 0 && ...; #)	't'!=0 && 't'!=0 && 't'=='t' Verdade	str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str+17 i=1
1006	10	for(;; i++)		str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str+17 i=2
1007	10	for(;; str[i] != 0 && ...; #)	'r'!=0 && 'r'!=0 && 'r'=='r' Verdade	str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str+17 i=2
1008	10	for(;; i++)		str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str+17 i=3
1009	10	for(;; str[i] != 0 && ...; #)	't'!=0 && '0'!=0 && (?) Falso	str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str+17 i=3
1010	12	if(find[i]==0)	0==0 Verdadeiro	str="percorrer..."	find="str"	resultado?	strstr	find=find	str=str+17 i=3
1011	14	return str;		str="percorrer..."	find="str"	resultado?	strstr+str+17	find=find	str=str+17 i=3
1012	30	resultado=str+17;		str="percorrer..."	find="str"	resultado=str+17			
1013	31	printf(...);	Resultado: string str	str="percorrer..."	find="str"	resultado=str+17			

endereço da variável em análise, em vez de utilizar um índice para aceder a determinada variável num vector. Esta técnica acaba por ser mais eficiente, uma vez que o acesso a uma posição arbitrária de um vector é mais lento que o acesso ao valor de uma variável através do endereço, mas pode resultar também em código mais confuso.

Erros comuns mais directamente associados a este capítulo:

- **Variáveis globais**

- **Forma:** Em vez de utilizar alguns dos argumentos nas funções, declara-se a variável globalmente em vez de na função main, de forma a não só ser acessível a todas as funções, como não ser necessário passá-la como argumentos nas funções
- **Problema:** Ao fazer isto, não só mascara as más opções na divisão do código por funções, dado que não controla os argumentos da função, como mais importante, perde principal vantagem das funções: quando implementa/revê a função, apenas tem de considerar a função e pode abstrair-se do resto do código. Como há variáveis globais, ao implementar a função tem que se considerar todos os restantes locais que essas variáveis são utilizadas (leitura / atribuição), para que o código na função seja compatível. Por outro lado, ao não se aperceber quando cria a função do número de argumentos que recebe e número de variáveis que altera, pode estar a criar funções que reduzem muito pouco a complexidade, sendo a sua utilidade diminuta
- **Resolução:** Utilize o seu código com variáveis globais para fazer uma revisão geral. Deve remover todas as variáveis globais e colocá-las na função main, deixando apenas as constantes globais que façam sentido estar definidas globalmente, e macros. Depois deve ir refazendo os argumentos das funções, e eventualmente rever as funções que necessita de forma a ter funções com poucos argumentos e com uma funcionalidade clara e simples

- **Nomes sem significado**

- **Forma:** Atendendo a que um bom nome gasta tempo e é irrelevante para o bom funcionamento do código, qualquer nome serve
- **Problema:** O código deve ser lido não só pelo compilador como por outras pessoas, pelo que a má utilização de nomes revela por um lado a má organização mental de quem o escreve, e por outro um desrespeito por quem lê o código. A troca de todos identificadores por nomes abstractos e sem sentido, pode ser uma forma para tornar o código ilegível, no caso de se pretender protegê-lo. Pode também ser uma situação confundida com uma situação de fraude, resultante de uma tentativa de um estudante alterar os identificadores de um código obtido ilicitamente
- **Resolução:** Verificar todos os identificadores de uma só letra, ou muito poucas letras, bem como nomes abstractos, funções sem uma grandeza associada, e procedimentos sem verbo. Se não consegue encontrar nomes claros para um identificador, seja uma variável, uma função, o nome de uma estrutura, etc., pense numa palavra em que explique para que a variável serve, o que a função/procedimento faz, o que contém a estrutura, etc. Se não consegue explicar isso, então o identificador não existe

- **Funções muito grandes**

- **Forma:** Tudo o que é necessário fazer, vai sendo acrescentado na função, e esta fica com cada vez uma maior dimensão. Esta situação ocorre devido ao problema ser muito complexo

- **Problema:** Ao ter uma função grande, vai ter problemas de complexidade. Não só as declarações de variáveis da função ficam longe do código onde são utilizadas, como não consegue visualizar a função de uma só vez, sendo muito complicado verificar o código. Esta é uma situação clara de má utilização da principal ferramenta da programação, que lhe permite controlar a complexidade do código: abstracção funcional
- **Resolução:** Deve dividir a função em partes que façam sentido, de forma a ter funções pequenas e com poucos argumentos. Dividir a função às fatias é má ideia, dado que os argumentos necessários para cada fatia podem ser muitos. Se houver várias fases, sim, colocar cada fase numa função, caso contrário deve verificar se os ciclos interiores podem construir funções que não só utilizem poucos argumentos como também tenham possibilidade de ser utilizadas em outras situações. Se a função reunir estas condições, terá certamente um nome claro
- **Funções com muitos argumentos**
 - **Forma:** Após criar a função, ao adicionar como argumentos as variáveis que esta utiliza (leitura / atribuição), este valor revela-se elevado, mas é o que é necessário
 - **Problema:** Se o número de argumentos é muito elevado, a abstracção que a função representa é fraca, uma vez que para utilizar a função, é necessário ter muita informação. Para além disso, se a função for muito curta, pode dar mais trabalho a arranjar os parâmetros certos para chamar a função, que colocar o seu código, sendo assim a função mais prejudicial que útil. No caso dos diversos argumentos estarem relacionados, significa que não foram agregados numa estrutura
 - **Resolução:** Deve procurar alternativas na criação de funções. Dividir uma função existente a meio, provoca situações destas. Para desagregar uma função, deve procurar o código repetido, se não existir, o código no interior dos ciclos é tipicamente um bom candidato a função, de forma a manter em cada função um ou dois ciclos. No caso de as variáveis estarem relacionadas, criar uma estrutura para as agregar

O Programa 8-4 é um exemplo de uma ocorrência do erro das variáveis globais. Um exemplo de uma ocorrência de nomes sem significado seria se em vez de utilizar `Troca` para o nome da função que troca o valor de duas variáveis, utilizar o nome `Processa`. Esse nome nada diz sobre o que o procedimento faz, pelo que o leitor fica sem saber se quem o escreveu sabe o que fez, uma vez que não foi capaz de dar um nome condicente com a sua função. Exemplos dos outros dois erros, não se encontram neste texto, dado que são utilizados sempre exemplos o mais curto possível para introduzir cada conceito. O leitor provavelmente encontrará ocorrências desses erros no seu próprio código, ao realizar os exercícios azuis e vermelhos.

Foram introduzidos alguns conceitos nesta secção, não só **procedimentos**, como também **passagem de valores por referência**, variáveis **globais** e **estáticas**, e **apontadores**. Todos estes conceitos não têm uma grande novidade, são construídos com base nos conceitos já dados anteriormente, nomeadamente em funções e vectores, mas são essenciais a uma boa estruturação do código. Cuidado especial deve ser dado no caso de se sentir tentado a recorrer com frequência a variáveis globais, sendo esse um dos sintomas de que não está a aproveitar as totais potencialidades da abstracção funcional. Num exercício em que tal lhe aconteça, analise com redobrada atenção resoluções alternativas.

9. RECURSÃO

A "Recursão" não é mais que a **aplicação da abstracção funcional, na própria função a implementar**. A abstracção funcional permite implementar uma função com base em apenas os seus argumentos, e ignorar tudo o resto. Ao mesmo tempo podemos utilizar funções sem querer saber como estão implementadas. É possível utilizar a função que se está a implementar?

Certamente que não se pode utilizar a função que se está a implementar para implementá-la, excepto se utilizar argumentos distintos. Vamos supor que se tem casos particulares muito simples, e para esses sabemos o resultado sem problema. No caso da função factorial, esses casos são quando o número é 0 ou 1, o valor do factorial é 1. Pode-se fazer a função para esses casos, e no caso genérico, utilizar o facto de $N! = (N-1)! \times N$. Assim podemos utilizar a função `Factorial` para implementar a função `Factorial`, mas utilizando argumentos mais baixos, e no limite irá ser utilizado os casos particulares simples. Vamos implementar no exercício seguinte esta função tanto no modo recursivo como no modo iterativo.

```
1 #include <stdio.h>
2
3 int FactorialR(int n)
4 {
5     if(n<2)
6         return 1;
7     return FactorialR(n-1)*n;
8 }
9
10 int FactorialI(int n)
11 {
12     int i, resultado=1;
13     for(i=2; i<=n; i++)
14         resultado*=i;
15     return resultado;
16 }
17
18 int main()
19 {
20     int n=5;
21     printf("Factorial de %d: %d = %d", n, FactorialR(n), FactorialI(n));
22 }
```

Programa 9-1 Factorial, versão iterativa e recursiva

Execução do programa:

```
C:\>factorial
Factorial de 5: 120 = 120
```

Repare-se na simplicidade da função `FactorialR`, tratamento dos casos simples, e tratamento do caso genérico recorrendo à mesma função mas com um argumento mais baixo, que acabará por tender para o caso simples. Esta função nem requer a criação de variáveis locais, apenas utiliza o argumento.

Ao contrário, a função `FactorialI` implementa o factorial todo, e retorna o valor. No entanto foi necessário um ciclo, e criar uma variável iteradora para iterar de 2 a N, bem como criar uma variável para guardar o resultado. Ambas as funções retornam o mesmo resultado, como se pode verificar na execução do programa, mas a versão recursiva é consideravelmente mais simples.

Para compreender bem a recursão, analise com cuidado a execução passo-a-passo acima. As diversas chamadas a `FactorialR` não se confundem em momento algum. Cada linha de código diz respeito sempre à última função que foi chamada, e assim permanecerá até que a função retorne, passando a execução para a linha de onde a função foi chamada, ou para a linha seguinte. Ao contrário da versão recursiva, o `FactorialI` não se chama a si próprio, mas utiliza duas variáveis `i` e `resultado` para calcular o valor pretendido.

Passo	Linha	Instrução	Resultado						
1	20	int n=3;		n=3					
2	21	printf(...,FactorialR,FactorialI);		n=3	FactorialR				
3	3	int n=3;		n=3	FactorialR	n=3			
4	5	if(n<2)	3<2 Falso	n=3	FactorialR	n=3			
5	7	return FactorialR(n-1)*n;		n=3	FactorialR	n=3	FactorialR		
6	3	int n=2;		n=3	FactorialR	n=3	FactorialR	n=2	
7	5	if(n<2)	2<2 Falso	n=3	FactorialR	n=3	FactorialR	n=2	
8	7	return FactorialR(n-1)*n;		n=3	FactorialR	n=3	FactorialR	n=2	FactorialR
9	3	int n=1;		n=3	FactorialR	n=3	FactorialR	n=2	FactorialR
10	5	if(n<2)	1<2 Verdade	n=3	FactorialR	n=3	FactorialR	n=2	FactorialR
11	6	return 1;		n=3	FactorialR	n=3	FactorialR	n=2	FactorialR=1
12	7	return 1*n;		n=3	FactorialR	n=3	FactorialR=2	n=2	
13	7	return 2*n;		n=3	FactorialR=6	n=3			
14	21	printf(...,6,FactorialI);		n=3	FactorialI				
15	10	int n=3;		n=3	FactorialI	n=3			
16	12	int i, resultado=1;		n=3	FactorialI	n=3	i=?	resultado=1	
17	13	for(i=2; #; #)		n=3	FactorialI	n=3	i=2	resultado=1	
18	13	for(#; i<=n; #)	2<=3 Verdade	n=3	FactorialI	n=3	i=2	resultado=1	
19	14	resultado*=i;		n=3	FactorialI	n=3	i=2	resultado*=2=2	
20	13	for(#; #; i++)		n=3	FactorialI	n=3	i=3	resultado=2	
21	13	for(#; i<=n; #)	3<=3 Verdade	n=3	FactorialI	n=3	i=3	resultado=2	
22	14	resultado*=i;		n=3	FactorialI	n=3	i=3	resultado*=3=6	
23	13	for(#; #; i++)		n=3	FactorialI	n=3	i=4	resultado=6	
24	13	for(#; i<=n; #)	4<=3 Falso	n=3	FactorialI	n=3	i=4	resultado=6	
25	15	return resultado;		n=3	FactorialI=6	n=3	i=4	resultado=6	
26	21	printf(...,6,6);	Factorial de 3: 6=6	n=3					

A recursão pode ser utilizada para substituir ciclos, tal como é exemplificado no factorial.

Um exemplo clássico quando se introduz a recursão é o problema das torres de Hanoi.

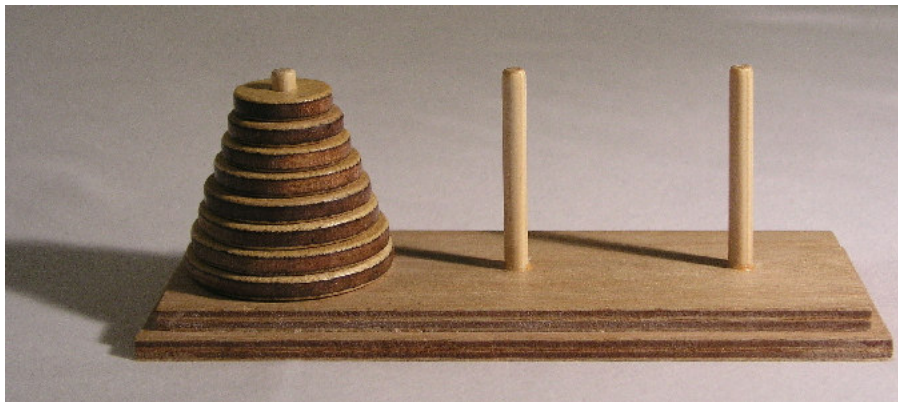


Figura 9-1 Problema das Torres de Hanoi, em madeira. Fonte: wikipédia

Tem-se 3 zonas onde se podem colocar discos numa pilha: esquerda, meio, direita. Os discos têm dimensões distintas e estão inicialmente na pilha da esquerda ordenados por dimensão, o disco maior em baixo, o mais pequeno em cima. Pode-se movimentar um disco de cada vez, do topo da pilha, para uma outra pilha que esteja vazia ou tenha um disco maior, mas nunca para cima de um disco menor. Pretende-se obter os movimentos necessários para passar todos os discos da pilha da esquerda para a pilha da direita.

Este problema aparentemente complexo, pode ser resolvido facilmente pela via recursiva, tratando primeiro os casos especiais. Os casos especiais são aqui o movimento de 1 só disco, já que isso é

simples, basta mover o disco da origem para o destino. Agora qual é o caso genérico? Será possível mover N discos da pilha A para a pilha B, sabendo como mover N-1 discos de uma pilha para outra? A resposta é sim, dado que o problema é mover o disco maior. Se mover os discos acima, portanto os N-1 discos, de A para uma pilha intermédia C, depois estamos em condições de mover 1 disco de A para B e de seguida mover os N-1 discos que estavam em C para B.

```

1 #include <stdio.h>
2
3 void Hanoi(int de, int para, int medio, int discos)
4 {
5     static char pilhas[3][] = { "esquerda", "meio", "direita" };
6     if(discos==1)
7         printf("\nMover do(a) %s para %s", pilhas[de], pilhas[para]);
8     else if(discos>1)
9     {
10         Hanoi(de,medio,para,discos-1);
11         Hanoi(de,para,medio,1);
12         Hanoi(medio,para,de,discos-1);
13     }
14 }
15
16 int main()
17 {
18     int discos;
19     printf("Torres de Hanoi. Quantos discos quer movimentar? ");
20     scanf("%d",&discos);
21     Hanoi(0,2,1,discos);
22 }

```

Programa 9-2 Torres de Hanoi

Repare na simplicidade da expressão genérica. Mesmo o movimento de um só disco pode ser reutilizado o código já feito na condição acima, chamando a função recursivamente. Como todas as chamadas à função têm o argumento disco com um valor inferior ao recebido, este tipo de utilização é lícito, uma vez que se está a diminuir a complexidade do problema até ao ponto ser um caso especial, o movimento de um só disco.

Execução do programa:

```

C:\>hanoi
Torres de Hanoi. Quantos discos quer movimentar? 2

Mover disco do(a) esquerda para meio
Mover disco do(a) esquerda para direita
Mover disco do(a) meio para direita
C:\>hanoi
Torres de Hanoi. Quantos discos quer movimentar? 4

Mover disco do(a) esquerda para meio
Mover disco do(a) esquerda para direita
Mover disco do(a) meio para direita
Mover disco do(a) esquerda para meio
Mover disco do(a) direita para esquerda
Mover disco do(a) direita para meio
Mover disco do(a) esquerda para meio
Mover disco do(a) esquerda para direita
Mover disco do(a) meio para direita
Mover disco do(a) meio para esquerda
Mover disco do(a) direita para esquerda
Mover disco do(a) meio para direita
Mover disco do(a) esquerda para meio
Mover disco do(a) esquerda para direita
Mover disco do(a) meio para direita

```


Erros comuns mais directamente associados a este capítulo:

- **Má utilização da recursão**

- **Forma:** Uma função tem um conjunto de instruções que têm de ser executadas várias vezes. Uma forma de fazer isto é no final da função colocar um teste a controlar o número de execuções, e chamar a função recursivamente
- **Problema:** Se a chamada recursiva está no final, e o código tem de passar sempre por esse condicional, então pode remover a chamada substituindo por um ciclo. Ao chamar a função recursivamente, as variáveis locais já criadas não vão mais ser necessárias, mas por cada passo do ciclo ficará um conjunto de variáveis locais à espera de serem destruídas. Se o ciclo for muito grande, o stack irá certamente rebentar
- **Resolução:** Substituir a chamada recursiva por um ciclo e respectivo teste de paragem. Caso a chamada recursiva não seja no final da função, ou exista mais que uma chamada recursiva, então sim, é preferível utilizar-se recursão

A função `FactorialR` do Programa 9-1 é um exemplo deste tipo de erro.

Nos exercícios da Parte II encontrará alguns que apenas se podem resolver de forma relativamente simples, através da recursão. Tenha em mente que não deve utilizar recursão para o que não é útil. Por exemplo, o caso do `Factorial`, embora mais simples, o ganho de simplicidade do código não é relevante comparativamente à versão iterativa. Já nas torres de Hanoi o ganho é muito relevante. A função `Fibonacci` é um exemplo aparentemente em que se deve aplicar a recursão, mas resulta em perda de eficiência.

A recursão é uma ferramenta muito poderosa que tem de ser utilizada com muito cuidado. Tanto pode reduzir um problema muito complexo num problema trivial, como pode criar problemas de performance e mesmo de memória.

2) Exercícios

argumentos.c



Faça um programa que indique os argumentos recebidos na linha de comando.

Notas:

- Utilize os parâmetros da função `main`, `argc` e `argv`

Execução de exemplo:

```
C:\>argumentos abcd 1234  
Argumento 0: argumentos  
Argumento 1: abcd  
Argumento 2: 1234
```

Pergunta: assumindo que o nome do programa é `argumentos`, qual deve ser a linha de comando para ter dois argumentos, o primeiro **ABC_123** e o segundo **123_ABC**?

inverte.c



Faça um programa que peça ao utilizador para inserir uma string, e utiliza uma função que recebe a string, e inverte a ordem dos caracteres da string.

Notas:


- Utilize a função `gets(str)`, para ler uma string, e `strlen(str)` para obter o número de caracteres na string, pertencente à biblioteca `string.h`.
- Considere que as strings inseridas pelo utilizador não ultrapassam os 255 caracteres.
- Imprima os resultados intermédios.

Execução de exemplo:

```
C:\>inverte  
Inversao de um texto.  
Texto: inverte  
0: enverti  
1: etverni  
2: etrevni  
Texto invertido: etrevni
```

Pergunta: Aplique ao programa a seguinte string, e coloque na resposta o texto invertido:
lcdsfkgjzoxilsd sdsdfd sdhjkafads dfwerwqad dsfs dfsdfsdkj

rand.c

 Faça um programa que utiliza uma função para gerar números pseudo-aleatórios, de acordo com a seguinte fórmula:

$$x(n+1) = \text{mod}(a \times x(n) + b, m)$$

Em que a , b e m são constantes grandes e primos. Os números pseudo-aleatórios são sempre inicializados por um primeiro valor, a semente (seed), que será o valor de $x(0)$ na expressão acima. Utilize o valor retornado por `time(NULL)`; (número de segundos no instante em que a função é chamada), de forma a inicializar a semente com o tempo, tendo assim resultados distintos em cada execução do programa. O programa deve gerar 10 números pseudo-aleatórios, entre 0 e um valor máximo, introduzido pelo utilizador.

Notas:


- `mod` é o resto da divisão, que em C é o operador `%`
- Deve incluir a biblioteca `time.h` para poder utilizar a função `time`.
- Se tem um número pseudo-aleatório grande, e pretende um número de 0 a N-1, pode utilizar o resto da divisão desse número por N para obter o valor pretendido.

Execução de exemplo:

```
C:\>rand
Gerador de numeros aleatorios inteiros.
Valor maximo: 99
seed=1282148053: 33 52 2 2 26 44 66 29 68 24
```

Pergunta: se a , b e m forem respectivamente 231533, 82571 e 428573, e com `seed=1` temporariamente, quais os três primeiros números gerados para valores entre 0 e 9? Escreva os números separados por espaços.

find.c

 Faça um programa que utiliza uma função que recebe um vector de inteiros, o seu tamanho, e um elemento, e retorna a primeira posição em que o elemento é encontrado no vector, ou -1 se o elemento não estiver no vector. O programa deve procurar o número 2.

Notas:

- Utilize a função standard `rand()` para gerar valores aleatórios (e não a que fez no exercício anterior), e inicialize a semente chamando a função `srand(1)`, (ambas as funções pertencem à biblioteca `stdlib.h`) antes de inicializar o vector com elementos aleatórios de 0 a 999. Utilize um vector com 1000 elementos.
- Mostre os 10 primeiros elementos do vector

Execução de exemplo:

```
C:\>find
Vector: 41 467 334 500 169 724 478 358 962 464
Posicao de 2: xxx
```

Pergunta: Qual é a primeira posição em que aparece o elemento 2?

maximo.c

Faça um programa que utiliza uma função para retornar o valor máximo num vector de reais (float), inicializado com valores gerados pela distribuição exponencial negativa:

$$-\log(\text{Uniforme}(0,1))$$

Notas:

- A função `log` está disponível na biblioteca `math.h`
- A função "Uniforme" devolve valores reais entre 0 e 1, mas não existe nas bibliotecas. Gere valores reais entre 0 e 1 (nunca gerando o zero), gerando um valor entre 1 e 10000 com a função `rand` (inicialize com `srand(1)`), após o qual efectue a divisão real por 10000, para assim ficar com valores desde 0,0001 até 1,0000.

Execução de exemplo:

```
C:\>maximo
Vector: 5.47 0.17 0.46 0.43 0.09 0.56 1.91 0.07 0.36 0.81
0: 5.47
54: 5.57
268: 6.12
915: 7.xx
Valor maximo: 7.xxxxxx
```

Pergunta: Gerando um vector com 1000 elementos, utilize a função desenvolvida para calcular o maior valor no vector. Indique na resposta esse valor com precisão de 6 dígitos.

mdc.c

Faça um programa que utiliza uma função recursiva para calcular o máximo divisor comum, utilizando o seguinte algoritmo recursivo: $\text{mdc}(x,y)=x$ se $y=0$, caso contrário $\text{mdc}(x,y)=\text{mdc}(y,\text{mod}(x,y))$, em que $\text{mod}(x,y)$ é o resto da divisão de x por y

Notas:

- O resto da divisão em C é o operador `%`

Execução de exemplo:

```
C:\>mdc
```

Calculo do maximo divisor comum entre dois numeros.


Indique x e y: **48 120**

MDC: 24

A soma dos MDC dos pares de numeros de 1 a 100: **xxxxx**

Pergunta: percorra todos pares de números distintos entre 1 e 100, não percorrendo o mesmo par duas vezes, e some os máximos divisores comuns. Indique o resultado na resposta.

sort.c

 Faça um programa que utiliza uma função para ordenar um vector de inteiros, com o seguinte algoritmo: percorre todos os pares sequencialmente, e se estiverem pela ordem inversa, troca os elementos.

Notas:

- Percorrer todos os pares apenas uma vez independente da ordem, não é necessário analisar o par 10, 35, e o par 35, 10.

Exemplo passo-a-passo:

Iteração	v[0]	v[1]	v[2]	v[3]
0	478	358	962	464
1	358	478	962	464
2	358	478	962	464
3	358	478	962	464
4	358	478	962	464
5	358	464	962	478
6	358	464	478	962

Na tabela acima, cada linha é uma iteração, sendo o estado do vector as 4 últimas colunas dessa linha. A bold estão os dois elementos em comparação no vector. Se estiverem pela ordem inversa, trocam de lugar. Após a última iteração, em que são comparados os elementos das posições 2 e 3, o vector fica ordenado.

Execução de exemplo:

```
C:\>sort
```

Vector antes de ordenado: 41 467 334 500 169 724 478 358 962 464

Vector depois de ordenado: 1 2 3 3 3 6 7 7 8 8

Quartil 25: **xxx**

Quartil 50: **xxx**

Quartil 75: **xxx**

Pergunta: utilizando a mesma inicialização de um vector de 1000 inteiros do exercício find.c, ordene o vector e indique o quartil de 25, 50 e 75% (valores nas posições 250, 500 e 750). Indique os três valores separados por um espaço.

removedups.c

Faça um programa que utilize uma função que remove os elementos duplicados num vector de inteiros ordenados. Deve retornar o número de elementos finais no vector.

Notas:

- Para remover um elemento no vector, todos os elementos seguintes têm de ser copiados para as novas posições.

Execução de exemplo:

C:\>*removedups*

Vector inicial: 41 467 334 500 169 724 478 358 962 464
 Elementos que permanecem no vector: **xxx**
 Vector final: 1 2 3 6 7 8 9 10 11 15

Pergunta: utilizando a mesma inicialização de um vector de 1000 inteiros do exercício find.c, após ordená-lo com código do exercício sort.c, execute esta função e indique na resposta o número de elementos que permanecem no vector.

somanumeros.c

Faça um programa que peça ao utilizador para introduzir uma string com números reais separados por espaços, e retorne a sua soma. Utilize precisão dupla e apresente o resultado em notação científica com precisão 15.

Notas:

- Pode utilizar a função strtok da biblioteca string.h, que tem a seguinte utilização:

```
char *pt=(char*)strtok(«string», " ");
while(pt!=NULL) {
    «processar pt»
    pt=(char*)strtok(NULL, " ");
}
```


- Pode utilizar a função atof que recebe uma string e retorna o seu valor numérico. Deve incluir a biblioteca stdlib.h, para utilizar esta função.

Execução de exemplo:

C:\>somanumeros
 Introduza um conjunto de numeros reais separados por espacos.
1.2 3.4 5.6 7.8
 1.20 (1.2)
 4.60 (3.4)
 10.20 (5.6)
 18.00 (7.8)
 A soma dos numeros e' 18.

Pergunta: Qual é o resultado para a seguinte string: "1 1.12320023400234044 1e+5 1e-5 -4"?

troc2.c

 Faça um programa com a mesma funcionalidade que o exercício 1) trocos.c, agora utilizando uma função que dado um montante, o devolve em moedas, mas em vez de utilizar código repetido para cada moeda, utilize um ciclo em que a informação de cada moeda está num vector previamente inicializado. Calcule também o total de moedas necessário, para os montantes 0.01; 0.03; 0.07; 0.15; ... $i=i*2+0.01$; ...; 20.

Notas:


- Os vectores previamente inicializados, podem conter o valor das moedas e os textos a apresentar.
- Chame uma função trocos, para saber quantas moedas são necessárias para cada um dos montantes, e faça um ciclo a começar em 0.01, e com a fórmula acima descrita

Execução de exemplo:

```
C:\> troc2
Introduza um montante em euros, podendo ter centimos: 3.49
2 euros: 1
1 euro: 1
20 centimos: 2
5 centimos: 1
2 centimos: 2
Total moedas: xx
```

Pergunta: Qual o total de moedas necessário para os montantes acima descritos?

jogodados.c

 Faça um programa que implemente um jogo de dados em que um jogador lança sucessivamente um dado, até optar por parar ficando com o número de pontos somados. Se saírem dois dados iguais consecutivos, o jogador fica com os pontos que tem, mas negativos. Faça um programa para simular o jogo, mediante o número de lançamentos do jogador.

Notas:

- Para responder à pergunta em baixo, execute N jogos para cada número de lançamentos, com N suficientemente grande de forma a obter um valor médio relativamente estável.
- Para obter execuções distintas, inicialize a semente com base no tempo, chamando no início do programa uma só vez: `srand(time(NULL));`

Exemplo de jogos:

```
Lançamentos: 4
jogo1: 3 1 6 5 » +15
jogo2: 3 4 4 » -11
...
```


Execução de exemplo:

```
C:\>jogodados
Jogo do lançamento de dados.
Indique numero de lançamentos: 4
3 4 3 1 Pontos: 11
```

```
valores medios dos pontos:
Com 1 lançamento(s): xxx
Com 2 lançamento(s): xxx
Com 3 lançamento(s): xxx
Com 4 lançamento(s): xxx
Com 5 lançamento(s): xxx
Com 6 lançamento(s): xxx
Com 7 lançamento(s): xxx
Com 8 lançamento(s): xxx
Com 9 lançamento(s): -5.06
```

Pergunta: Indique por ordem e separado por um espaço, os números de lançamentos cujo valor esperado seja superior a 3.

baralhar.c

Faça um programa que utiliza uma função que recebe um vector e a sua dimensão, e baralhe os elementos nele contidos. Atenção que deve realizar o número mínimo de trocas, e a probabilidade de cada elemento estar em qualquer posição deve ser igual.

Notas:

- Começando pelo primeiro elemento, tem de haver um valor aleatório para decidir qual será o primeiro elemento. O segundo valor aleatório decidirá qual será o segundo elemento (não removendo o primeiro elemento que já foi escolhido aleatoriamente), e assim sucessivamente para todos os restantes elementos.

Exemplificação:

Na tabela abaixo, em cada linha está uma iteração, bem como o valor das diversas posições do vector, e o resultado de um valor aleatório gerado em cada iteração para decidir qual o elemento nessa iteração. O valor aleatório pertence a um intervalo cada vez mais pequeno, até que o último valor é um valor aleatório 0 ou 1, para decidir a ordem das duas últimas posições no vector.

Iteração	Valor Aleatório	0	1	2	3
0		48	22	80	25
1	rand()%4 = 2	80	22	48	25
2	rand()%3 = 0	80	22	48	25
3	rand()%2 = 1	80	22	25	48

Execução de exemplo:

```
C:\>baralhar
Vector identidade: 0 1 2 3 4 5 6 7 8 9
Vector baralhado: 41 486 348 581 249 804 550 568 186 689
Na posicao 250, 500, 750: xxx xxx xxx
```

Pergunta: crie um vector de 1000 elementos com valores de 0 a 999 seguidos, e chame a função para os baralhar (chame `srand(1)`). Indique na resposta os elementos nas posições 250, 500 e 750, separados por espaços.

mergesort.c



Faça um programa que utiliza uma função que ordena os elementos de um vector da seguinte forma: divide o vector em duas metades, ordena cada metade separadamente, e depois junta ambas as metades ordenadas.

Notas:

- Ao juntar duas partes ordenadas, basta ver apenas os elementos no topo de cada metade, já que ambas as metades estão ordenadas.
- Como há duas operações separadas, ordenar parte do vector e juntar duas partes de um vector, é mais fácil criar uma função para cada uma destas operações.
- Para a pergunta em baixo, ao declarar um vector de 1.000.000 de elementos, utilize a keyword `static`, ou uma variável global, de forma a evitar que o vector fique no stack⁷ uma vez que é muito grande.


Execução de exemplo:

```
C:\>mergesort
Ordenado de 0 a 124999
Ordenado de 125000 a 249999
Ordenado de 0 a 249999
Ordenado de 250000 a 374999
Ordenado de 375000 a 499999
Ordenado de 250000 a 499999
Ordenado de 0 a 499999
Ordenado de 500000 a 624999
Ordenado de 625000 a 749999
Ordenado de 500000 a 749999
Ordenado de 750000 a 874999
Ordenado de 875000 a 999999
Ordenado de 750000 a 999999
Ordenado de 500000 a 999999
Ordenado de 0 a 999999
Quantis: xxx xxx xxx
```

Pergunta: gerando agora um vector aleatório com 1.000.000 de elementos, com valores de 0 a 999 (utilize `rand`, e chame `srand(1)`), utilize a função de ordenação desenvolvida para ordenar o vector e retorne os quantis de 25, 50 e 75% (valores nas posições 250000, 500000, e 750000), separados por espaços.

⁷ O stack será dado no capítulo Memória

binarysearch.c

 Faça um programa que utiliza uma função que recebe um vector ordenado, tamanho e um elemento, e retorna a posição em que o elemento se encontra (ou -1 se não estiver no vector), efectuando uma procura binária: primeiro verifica a posição do meio, se for igual retorna, caso contrário selecciona apenas o segmento em que poderá estar o elemento (à esquerda ou direita do valor testado), e volta a testar a posição do meio desse segmento. O processo continua recursivamente até que o segmento apenas tenha um elemento.

Notas:


- Inclua o código do exercício mergesort.c de modo a obter o vector de elementos ordenados, e coloque uma mensagem no início da função recursiva.

Execução de exemplo:

```
C:\>binarysearch
Procurar 250 entre 0 e 999999
Procurar 250 entre 0 e 499998
...
Procurar 500 entre 0 e 999999
Procurar 500 entre 500000 e 999999
...
Procurar 750 entre 0 e 999999
Procurar 750 entre 500000 e 999999
...
Posicao de 250, 500 e 750: 251952 503905 755858
```

Pergunta: utilizando o mesmo vector ordenado do exercício anterior, procure as posições dos valores 250, 500 e 750. Indique o número de chamadas recursivas para cada uma das procuras, separadas por espaços.

numeracaoromana.c

 Faça um programa que utiliza duas funções, uma para converter um número em numeração romana, e outro para a operação inversa. Lembra-se as correspondências entre letras e números: M=1000; D=500; C=100; L=50; X=10; V=5; I=1. No sistema de numeração romano as letras devem situar-se da ordem de maior valor para a de menor valor. Não se deve escrever mais de três I, X ou C em qualquer número. Se estas letras se situam à esquerda de um V, L ou D, subtrai-se o seu valor ao das respectivas letras. Exemplo: IX, XC ou XL correspondem a 9, 90 e 40 respectivamente.

Notas:

- Idealize o algoritmo você mesmo. Caso não vislumbre nenhuma solução, comece por considerar o problema até ao número 10.

Execução de exemplo:

```

C:\>numeracaoromana
Conversao de numeracao arabe para romana e vice-versa.
Numero arabe: 1234
234 M
134 MC
34 MCC
24 MCCX
14 MCCXX
4 MCCXXX
0 MCCXXXIV
Resultado: MCCXXXIV
Numero romano: MCCXXXIV
1000 CCXXXIV
1100 CXXXIV
1200 XXXIV
1210 XXIV
1220 XIV
1230 IV
1234
Resultado: 1234
Caracteres dos numeros de 1 a 999, em numeracao romana: xxxx

```

Pergunta: calcule o número de caracteres romanos necessários para os números de 1 a 999 (somar o número de letras desses números).

pokerdados.c

Faça um programa para o jogo do Poker de Dados. Joga-se com 5 dados, em que um jogador lança os dados (cada dado tem um valor de 1 a 6), podendo relançar qualquer número de dados uma segunda vez. Ganha o jogador que ficar com o melhor lançamento. Pretende-se que simule o lançamento de um jogador neste jogo, podendo o jogador escolher os dados a relançar, e diga qual o resultado. Os resultados são os seguintes: poker real = 5 iguais; poker quadruplo = 4 iguais; fullen = trio + par; sequência = 1 a 5 ou 2 a 6; trio = 3 iguais; duplo par = par + par; par = 2 iguais; nada = não se verifica nenhum dos casos anteriores.

Notas:

- Faça uma função que para um conjunto de 5 dados, retorne o resultado do jogo.

Execução de exemplo:

```

C:\>pokerdados
Valores dos dados lancados: 6 1 2 1 1
Frequencia dos valores: 3 1 0 0 0 1 (trio)
Retencao (1-reter, 0-descartar): 01011
Valores dos dados finais: 4 1 2 1 1
Frequencia dos valores: 3 1 0 1 0 0 (trio)
nada: xx
par: xxx
duplo par: xxx

```

Pergunta: para responder a esta pergunta chame `srand(1)`, e faça 1000 lançamentos de dados, e indique quantas vezes saiu: nada; par; duplo par. Coloque na resposta estes três valores separados por espaços.

strreplace.c

■ Faça um programa que utiliza uma função que substitui numa string, todas as ocorrências de uma string A por uma string B, assumindo que há espaço devidamente alocado (não há realocação de strings).

Notas:

- Se a string B conter A, a string em A em B não deve ser substituída recursivamente.

Execução de exemplo:

```
C:\>strreplace
Substituir num texto, as ocorrencias de uma string A por uma string B.
Texto: texto a transformar troco o "o" por "os"
String A: o
String B: os
  textos a transformar troco o "o" por "os"
  textos a transformar trocos o "o" por "os"
  textos a transformar trocos os "o" por "os"
Resultado (3 trocas): textos a transformar trocos os "o" por "os"
```

Nas strings introduzidas "o" e "os" adicionou-se um espaço.

Pergunta: Coloque na resposta o resultado da substituição de "xju" por "uxjuc" em "sdxjukflj sxjuxjudfgklj sdfli sdfkxjuj erlijs sdfikxjxjuulsj xju xdvflkj wsefrlij wefsxjuld dfg".

ndamas.c

■ Faça um programa que peça um número N, e liste todas as possíveis posições para colocar N damas, num tabuleiro de Xadrez de lado N, sem que nenhuma dama ataque outra (mesma diagonal, linha ou coluna). Faça um programa recursivo. Mostre apenas a primeira posição, e conte quantas posições existem.

Notas:

- Considere um valor máximo de 16 damas

Execução de exemplo:

```
C:\>ndamas
Conta quantas posicoes existem, num tabuleiro de NxN, para colocacao de
N damas sem que estas se ataquem mutuamente.
Indique N (maximo 16): 5
```

```
+-----+
| #   .   .   .   . |
| .   .   #   .   . |
| .   .   .   .   # |
| .   #   .   .   . |
| .   .   .   #   . |
+-----+
```

Total: 10

Pergunta: quantas posições existem para 13 damas?

doisdados.c



Escreva um programa que lance dois dados um número elevado de vezes, e registe a frequência absoluta da soma dos dados. Faça também uma função para apresentar um gráfico de barras com o histograma, que escale os valores a apresentar de forma a utilizar 16 linhas, utilizando uma coluna por cada valor da série, neste caso, um resultado de lançamento de dados, sendo o valor o número de lançamentos.

Notas:

- Faça uma função separada e a mais genérica possível para representar o gráfico de barras
- Como tem de imprimir o gráfico na horizontal, tem de imprimir o gráfico de barras para um vector bidimensional antes de o imprimir

Execução de exemplo:

```
C:\>doisdados
Simulacao de 10000 lancamentos de dois dados:
1677|      #  xxxx
1565|      #  xxxx
1453|     ##  xxxx
1342|     ##  xxxx
1230|    ###xxxx
1118|    ###xxxx
1006|    ####xxxx
 894|    ####xxxx
 783|    ####xxxx
 671|    #####xxxx
 559|    #####xxxx
 447|    #####xxxx
 335|    #####xxxx
 224|    #####xxxx
 112|    #####xxxx
  0|    #####
    +-----+
      23456789DOD

valores: 250 546 780 1117 1468 1677 1330 xxxx xxx xxx xxx
Numero de caracteres # presentes no histograma: xx
```

Pergunta: faça o histograma utilizando 10.000 lançamentos, e chamando srand(1). Indique o número de cardinais no histograma.

calendario.c



Faça um programa que imprima um calendário mensal, para um ano e mês introduzidos pelo utilizador, com o dia da semana nas colunas, e atendendo às seguintes regras e feriados nas notas.

Notas:

- Determinação de ano bissexto: de 4 em 4 anos, é um ano bissexto; de 100 em 100 anos não é ano bissexto; de 400 em 400 anos é ano bissexto; prevalecem as últimas regras sobre as primeiras.
 - Exemplo: O ano 2000 é um ano bissexto, sendo 1 de Janeiro a um sábado
- Dias dos meses: 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
- Páscoa⁸:
 - $a = \text{ano} \bmod 19$
 - $b = \text{ano} \setminus 100$ (divisão inteira)
 - $c = \text{ano} \bmod 100$
 - $d = b \setminus 4$
 - $e = b \bmod 4$
 - $f = (b + 8) \setminus 25$
 - $g = (b - f + 1) \setminus 3$
 - $h = (19 * a + b - d - g + 15) \bmod 30$
 - $i = c \setminus 4$
 - $k = c \bmod 4$
 - $l = (32 + 2 * e + 2 * i - h - k) \bmod 7$
 - $M = (a + 11 * h + 22 * l) \setminus 451$
 - $Mês = (h + l - 7 * M + 114) \setminus 31$
 - $\text{Dia} = ((h + l - 7 * M + 114) \bmod 31) + 1$
- 2ª-feira de Carnaval: Páscoa - 48 dias
- 3ª-feira de Carnaval: Páscoa - 47 dias
- 6ª-feira Santa: Páscoa - 2 dias
- Corpo de Deus: Páscoa + 60 dias
- Feriados fixos: ano novo 1/1; dia da liberdade 25/4; dia do trabalhador 1/5; dia de Portugal 10/6; implantação da República 5/10; restauração da independência 1/12; Natal 25/12.
- Faça várias pequenas funções, uma por cada funcionalidade que necessite.

Execução de exemplo:


```
C:\>calendario
Calendario mensal, com indicacao de feriados.
Indique ano mes:2010 10
```

```
Outubro de 2010:
#####
# D S T Q Q S S #
#####
#           1 2 #
# 3 4 F 6 7 8 9 #
#10 11 12 13 14 15 16 #
#17 18 19 20 21 22 23 #
#24 25 26 27 28 29 30 #
#31           #
#####
Resposta: xxxxx
```

Pergunta: some para o mês de Fevereiro, desde 2000 a 2099, o número de feriados, o número de dias do mês, e o dia da semana para o dia 1 (Domingo vale 0, Sábado vale 6). Coloque o resultado na resposta.

⁸ Fonte: http://pt.wikipedia.org/wiki/C%C3%A1culo_da_P%C3%A1scoa

editdistance.c

 Faça um programa que utiliza uma função que recebe duas strings e retorna a distância de edição (mostrando informação passo a passo), de acordo com o seguinte algoritmo⁹:

```

m[i,j] = distance(s1[1..i], s2[1..j])
m[0,0] = 0
m[i,0] = i, i=1..|s1|
m[0,j] = j, j=1..|s2|
for i=1..|s1|
  for j=1..|s2|
    if s1[i]=s2[j] then
      m[i,j] = m[i-1,j-1]
    else
      m[i,j] = min(
        m[i-1,j-1] + 1, /* substituição */
        m[i-1,j] + 1,   /* remoção      */
        m[i,j-1] + 1 ) /* inserção     */

```

Notas:

- Considere strings até um tamanho máximo, 256

Execução de exemplo:

```

C:\>editdistance
Calculo da distancia de edicao entre duas strings.
Indique s1: Portugal
Indique s2: Brasil
Matriz de distancias:
B r a s i l
0 1 2 3 4 5 6
P 1 1 2 3 4 5 6
o 2 2 2 3 4 5 6
r 3 3 2 3 4 5 6
t 4 4 3 3 4 5 6
u 5 5 4 4 4 5 6
g 6 6 5 5 5 5 6
a 7 7 6 5 6 6 6
l 8 8 7 6 6 7 6
Operacoes:
Portugal (l) Portugal
Portugal [-a] Portugl
Portugl [g/i] Portuyl
Portuyl [u/s] Portsil
Portsil [t/a] Porasil
Porasil (r) Porasil
Porasil [-o] Prasil
Prasil [P/B] Brasil
Apagar: 2, Inserir: 0, Substituir: 4
Resultado: 6

```

Pergunta: Calcule as três distâncias de edição entre: "Sport Lisboa e Benfica", "Sporting Clube de Portugal", e "Futebol Clube do Porto". Indique na resposta para cada uma das distâncias, separado por espaços, o número de substituições.

⁹ Fonte: http://en.wikipedia.org/wiki/Levenshtein_distance

PARTE III – MEMÓRIA, ESTRUTURAS E FICHEIROS

Na Parte II pouco foi dito sobre a gestão da memória, as variáveis relacionadas entre si eram declaradas em separado, nem tão pouco se falou em ficheiros. Na Parte III vamos finalmente abordar estas questões, reservando ainda um capítulo para matéria menos relevante mas característica da linguagem C, a que se chamou de “Truques”. Mais relevante e complementar à abstracção funcional é a **abstracção de dados**, que é introduzida nesta parte, sendo essencial utilizar nas grandes aplicações. A abstracção de dados é a base da programação por objectos. No final desta parte estará munido de todo o conhecimento que necessita de saber para construir aplicações em programação estruturada.

10. MEMÓRIA

Até este momento o conhecimento sobre a memória num programa resume-se no seguinte: sempre que é necessário, declaram-se variáveis no início das funções, que na forma de vectores podem ser arbitrariamente grandes.

Nada mais haveria a dizer, não fossem os seguintes problemas:

- Todo o espaço necessário a cada momento tem de ser conhecido antes de compilar o programa (em tempo de compilação / compile time);
- Não pode uma função criar memória para ser utilizada na função que a chamou, toda a memória que uma função pode aceder, tem de ser declarada no início da função.

O primeiro problema é deveras desagradável quando não se sabe à partida o tamanho de um vector. Para o resolver, até aqui declarava-se uma macro com um tamanho por excesso, a utilizar na declaração do vector de forma a facilitar a sua alteração se necessário. No caso de o programa necessitar um valor maior enquanto está a correr (em tempo de corrida / run time), tem-se de informar o utilizador que o programa não tinha sido compilado a contar com memória suficiente para satisfazer o seu pedido.

O valor por excesso pode levar a que exista um desperdício de memória, inviabilizando várias aplicações a correr em simultâneo apenas porque têm uma reserva de memória feita em tempo de compilação, e não quando for conhecido o valor necessário em tempo de corrida.

Vamos ver uma versão generalizada do exemplo das semanas no capítulo Vectores, em que se pede para introduzir valores de dimensão variável, obtendo-se no final a média e o desvio padrão.

```
1 #include <stdio.h>
2 #include <math.h>
3
4 #define MAXVECTOR 10
5
6 int main()
7 {
8     float valor[MAXVECTOR];
9     float media=0,variancia=0;
10    int i,n;
11
12    printf("Quantos valores tem o vector: ");
13    scanf("%d",&n);
14    if(n<=0 || n>MAXVECTOR)
15    {
16        printf("Programa compilado para permitir no maximo %d elementos.\n",
17              MAXVECTOR);
18        return;
19    }
20
21    /* introdução de valores */
22    for(i=0;i<n;i++)
23    {
24        printf("Valor %d: ",i);
25        scanf("%f",&valor[i]);
26    }
27
28    /* calculos */
29    for(i=0;i<n;i++)
30        media+=valor[i];
31    media/=n;
```

```

32     for(i=0;i<n;i++)
33         variancia+=(media-valor[i])*(media-valor[i]);
34     variancia/=n;
35
36     printf("Media: %g\n",media);
37     printf("Desvio padrao: %g\n",sqrt(variancia));
38 }

```

Programa 10-1 Generalização do Programa 7-2, calculando a média e o desvio padrão

Execução de exemplo:

```

C:\>media
Quantos valores tem o vector: 3
Valor 0: 123
Valor 1: 12
Valor 2: 10
Media: 48.3333
Desvio padrao: 52.8036
C:\>media
Quantos valores tem o vector: 11
Programa compilado para permitir no maximo 10 elementos.

```

Na primeira execução, precisávamos 3 valores, mas foi utilizado um vector de 10, e na segunda execução, eram precisos 11 valores, mas o programa não estava compilado para poder retornar a média e o desvio padrão de um conjunto de 11 valores. Em todo o caso é preferível assim a fazer um acesso fora do espaço alocado.

O segundo problema é de maior gravidade, uma vez que quanto muito uma função pode eventualmente utilizar memória já alocada nas funções que já a chamaram, acedendo à memória de outras funções por referência, através de apontadores, mas nunca pode ela própria criar memória para que as restantes funções a utilizem, inviabilizando abstracções de funções que necessitem criar memória.

Por exemplo, poderíamos estar interessados em fazer isto:

```

/* retorna um vector de inteiros com dimensão n, com valores aleatórios */
? VectorAleatorio(int n);

```

Em vez disso tem que se criar o vector com a dimensão n, e passar como argumento junto com o valor n.

STACK

Para resolvermos os dois problemas enunciados, é necessário saber como é que um programa realmente cria o espaço necessário para as suas variáveis. Para tal vamos olhar de novo para o primeiro exemplo de recursão, não por causa do código, mas pela execução passo-a-passo.

Em cada um dos 26 passos, o programa necessita de ter o código na memória, e tem a linha em que está actualmente, o que permite obter a instrução actual. Há também um espaço para "Resultado", após o qual há um número variável de colunas necessárias para registar o valor das variáveis. Ora, ao espaço necessário para escrever uma linha nesta execução passo-a-passo, corresponde também espaço em memória necessário pelo computador. Há os registos internos fixos, um deles tem realmente a instrução a executar e outros podem ser utilizados para valores temporários, como os colocados na coluna "Resultado", sendo estes registos dependentes da arquitectura do computador.

Passo	Linha	Instrução	Resultado							
1	20	int n=3;		n=3						
2	21	printf(...,FactorialR,FactorialI);		n=3	FactorialR					
3	3	int n=3;		n=3	FactorialR	n=3				
4	5	if(n<2)	3<2 Falso	n=3	FactorialR	n=3				
5	7	return FactorialR(n-1)*n;		n=3	FactorialR	n=3	FactorialR			
6	3	int n=2;		n=3	FactorialR	n=3	FactorialR	n=2		
7	5	if(n<2)	2<2 Falso	n=3	FactorialR	n=3	FactorialR	n=2		
8	7	return FactorialR(n-1)*n;		n=3	FactorialR	n=3	FactorialR	n=2	FactorialR	
9	3	int n=1;		n=3	FactorialR	n=3	FactorialR	n=2	FactorialR	n=1
10	5	if(n<2)	1<2 Verdade	n=3	FactorialR	n=3	FactorialR	n=2	FactorialR	n=1
11	6	return 1;		n=3	FactorialR	n=3	FactorialR	n=2	FactorialR=1	n=1
12	7	return 1*n;		n=3	FactorialR	n=3	FactorialR=2	n=2		
13	7	return 2*n;		n=3	FactorialR=6	n=3				
14	21	printf(...,6,FactorialI);		n=3	FactorialI					
15	10	int n=3;		n=3	FactorialI	n=3				
16	12	int i, resultado=1;		n=3	FactorialI	n=3	i=?	resultado=1		
17	13	for(i=2; #; #)		n=3	FactorialI	n=3	i=2	resultado=1		
18	13	for(;; i<=n; #)	2<=3 Verdade	n=3	FactorialI	n=3	i=2	resultado=1		
19	14	resultado*=i;		n=3	FactorialI	n=3	i=2	resultado*=2=2		
20	13	for(;; #; i++)		n=3	FactorialI	n=3	i=3	resultado=2		
21	13	for(;; i<=n; #)	3<=3 Verdade	n=3	FactorialI	n=3	i=3	resultado=2		
22	14	resultado*=i;		n=3	FactorialI	n=3	i=3	resultado*=3=6		
23	13	for(;; #; i++)		n=3	FactorialI	n=3	i=4	resultado=6		
24	13	for(;; i<=n; #)	4<=3 Falso	n=3	FactorialI	n=3	i=4	resultado=6		
25	15	return resultado;		n=3	FactorialI=6	n=3	i=4	resultado=6		
26	21	printf(...,6,6);	Factorial de 3: 6=6	n=3						

Algo que é comum a todos os programas a correr em computadores actuais, é o espaço necessário de memória, mais vasto que registos, e de dimensão variável, para poder guardar o valor das variáveis declaradas e as funções que são chamadas: **stack** (pilha). Este espaço de memória tem o seu nome dado que as variáveis são colocadas em posições de memória consecutivas pela ordem que são declaradas, tal como na execução passo-a-passo se coloca uma nova variável na próxima coluna livre, mas por outro lado, quando a função retorna, o espaço para as variáveis declaradas nessa função é libertado, ficando disponível para outras funções. Na execução acima, pode-se ver que a função **FactorialI**, utilizou as mesmas colunas que eram utilizadas pela função **FactorialR**, que tinha corrido anteriormente, mas como já tinha terminado não necessitava mais de memória.

Há problemas de memória relacionados apenas com o **stack** que convém realçar. Note-se que o próprio **stack** é um bloco de memória, e como tal foi alocado. Se o programa abusar do **stack**, pode acontecer que não exista espaço no **stack** para alocar mais variáveis, tal como a folha de papel na execução passo-a-passo pode atingir a borda. Isto pode acontecer se alocar variáveis muito grandes, ou se a recursão for exagerada, provocando muita alocação de memória.

Vamos continuar no exemplo do **factorial**, para ter um problema de falta de espaço no **stack**.

```

1  #include <stdio.h>
2
3  int FactorialR(int n)
4  {
5      if(n<2)
6          return 1;
7      return FactorialR(n-1)*n;
8  }
9
10 int FactorialI(int n)
11 {
12     int i, resultado=1;
13     for(i=2; i<=n; i++)
14         resultado*=i;
15     return resultado;
16 }
17
18 int main()
19 {
20     int n=1000;
21     printf("Factorial de %d: %d = %d", n, FactorialR(n), FactorialI(n));
22 }

```

Programa 10-2 Réplica do Programa 9-1

Execução de exemplo para n=1000:

```

C:\>factorial
Factorial de 1000: 0 = 0

```

O valor zero é normal, já que o valor do factorial largamente ultrapassa a dimensão representável por um número de 4 bytes, e assim que o resultado acumulado seja nulo, será sempre nulo até ao fim do produto. O importante aqui foi verificar que ambas as formas de cálculo do factorial correram bem e retornaram correctamente.

Execução de exemplo para n=1000000:

```

C:\>factorial

```

O programa abortou. A versão recursiva estourou o `stack`, se utilizar apenas a versão iterativa o resultado é fornecido. Repare-se que a versão recursiva para n=1000000 significa 1000000 de chamadas, tendo que ser reservado para cada uma o apontador de onde a função foi chamada, e o espaço para guardar o parâmetro n. A versão iterativa é chamada uma só vez, e aloca apenas espaço para as suas variáveis locais uma só vez.

O `stack` não é o local apropriado para guardar muita memória. Do exemplo, conclui-se que não se deve contar com um número infindável de chamadas recursivas de funções, porque em cada chamada tem que se contar com o espaço para declarar as variáveis locais da função, e o `stack` tem limites.

HEAP

Para resolver os problemas enunciados, temos de ter uma forma de alocar memória, e uma vez alocada, quando já não for precisa tem que se poder libertar, em vez de se declarar variáveis no `stack` sendo estas libertas automaticamente quando a função retorna. Existem precisamente duas funções para este efeito, uma para alocar memória **malloc** e outra para libertar memória alocada pelo `malloc`: **free**¹⁰.

```
void *malloc(int);
void free(void*);
```

`malloc` retorna um apontador para um tipo `void`, apontando para a memória alocada.

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <stdlib.h>
4
5 void Media(int n)
6 {
7     float *valor;
8     float media=0, variancia=0;
9     int i;
10    valor=(float*)malloc(sizeof(float)*n);
11    if(valor!=NULL)
12    {
13        /* introdução de valores */
14        for(i=0; i<n; i++)
15        {
16            printf("valor %d: ", i);
17            scanf("%f", &valor[i]);
18        }
19
20        /* calculos */
21        for(i=0; i<n; i++)
22            media+=valor[i];
23        media/=n;
24        for(i=0; i<n; i++)
25            variancia+=(media-valor[i])*(media-valor[i]);
26        variancia/=n;
27        printf("Media: %g\n", media);
28        printf("Desvio padrao: %g\n", sqrt(variancia));
29        free(valor);
30    } else
31        printf("Memoria insuficiente para alocar %d elementos.\n",
32              n);
33 }
34
35 int main()
36 {
37     int n;
38
39     printf("Quantos valores tem o vector: ");
40     scanf("%d", &n);
41
42     Media(n);
43 }
```

Programa 10-3 Calculo da média e desvio padrão, alocando apenas a memória necessária

¹⁰ As funções `malloc` e `free` pretendem à biblioteca `stdlib.h`

Execução de exemplo:

```
C:\>media2
Quantos valores tem o vector: 3
Valor 0: 123
Valor 1: 12
Valor 2: 10
Media: 48.3333
Desvio padrao: 52.8036
C:\>media2
Quantos valores tem o vector: 11
Valor 0: 11
Valor 1: 2
Valor 2: 2
Valor 3: 3
Valor 4: 4
Valor 5: 5
Valor 6: 6
Valor 7: 7
Valor 8: 23
Valor 9: 1
Valor 10: 3
Media: 6.09091
Desvio padrao: 5.99173
C:\>media2
Quantos valores tem o vector: 1000000000
Memoria insuficiente para alocar 1000000000 elementos.
```

O espaço alocado foi o necessário para efectuar as operações pretendidas.

Ao pedir um valor muito elevado para o vector, na última execução, a função `malloc` não consegue reservar memória, e retorna `NULL`, dando erro, mas não dependente dos valores utilizados na compilação, mas sim dos recursos disponíveis na máquina quando o programa correu. O heap tem limites, se não consegue alocar memória a função `malloc` retorna `NULL`.

Na linha 10, o valor retornado pelo `malloc` é do tipo `void*`, mas na verdade é pretendido um tipo `float*`. É lícito efectuar-se a atribuição, sendo convertido o tipo `void*` para `float*`, mas a instrução tem um `cast` para `(float*)` para que o compilador force a conversão na expressão mesmo antes da atribuição, e por outro lado ao ler-se o código não se tenha dúvidas, que a conversão é pretendida e não se trata de um erro de atribuição entre variáveis de tipos distintos. O `cast` já poderia ter sido referido anteriormente, dado que se pode utilizar em qualquer parte de uma expressão para a converter para o tipo pretendido, no entanto a sua importância é maior no contexto dos apontadores. Por exemplo para fazer a divisão real entre duas variáveis inteiras `x` e `y`, pode-se utilizar a seguinte expressão: `(float)x/y`. Para fazer a divisão inteira e converter o resultado para `float`, será: `(float)(x/y)`.

Vamos agora fazer um programa que vai lendo strings, e as junta numa só string:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define MAXSTR 255
6
7 char *Concatenar(char *str, char *str2)
8 {
9     char *pt;
10    /* duplicar str2 se str é nulo */
```

```

11     if(str==NULL)
12     {
13         pt=(void *)malloc(strlen(str2)+1);
14         if(pt!=NULL)
15             strcpy(pt,str2);
16     } else {
17         pt=(void *)malloc(strlen(str)+strlen(str2)+1);
18         if(pt!=NULL)
19         {
20             strcpy(pt,str);
21             strcat(pt,str2);
22             free(str);
23         }
24     }
25     return pt;
26 }
27
28 int main()
29 {
30     char *texto=NULL, str[MAXSTR];
31     do {
32         gets(str);
33         texto=Concatenar(texto,str);
34     } while(strlen(str)>0);
35     printf("%s",texto);
36     free(texto);
37 }

```

Programa 10-4 Concatena as strings introduzidas, utilizando apenas a memória necessária

Execução de exemplo:

```

C:\>strings
primeira linha
segunda linha
última linha
primeira linhasegunda linhaultima linha

```

A vantagem neste exemplo, é permitir ir alocando memória conforme as necessidades. Se o utilizador escrever pouco texto, o programa necessita de pouca memória, requisitando mais memória à medida que o utilizador escreve mais texto.

Repare-se que a função Concatenar, não só liberta o espaço que já não é necessário, como aloca o novo espaço, e retorna o apontador para que na função main o apontador se mantenha actualizado.

Pode-se agora fazer abstracções funcionais, que aloquem memória, como por exemplo a criação de um vector de inteiros aleatórios:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int *VectorAleatorio(int n, int base)
5  {
6      int *vector,i;
7      vector=(int*)malloc(sizeof(int)*n);
8      if(vector!=NULL)
9      {
10         for(i=0;i<n;i++)
11             vector[i]=rand()%base;
12     }
13     return vector;
14 }
15
16 void MostraVector(int *vector, int n)
17 {
18     int i;
19     for(i=0;i<n;i++)

```

```

20         printf("%d ",vector[i]);
21     }
22
23     int main()
24     {
25         int *vector,n,base;
26         srand(1);
27
28         printf("Dimensao: ");
29         scanf("%d",&n);
30         printf("Valor maximo: ");
31         scanf("%d",&base);
32
33         vector=VectorAleatorio(n,base);
34         if(vector!=NULL)
35         {
36             MostraVector(vector,n);
37             free(vector);
38         }
39     }

```

Programa 10-5 Geração de um vector aleatório

Execução de exemplo:

```

C:\>vectoraleatorio
Dimensao: 10
Valor maximo: 10
1 7 4 0 9 4 8 8 2 4

```

O vector alocado na função `VectorAleatorio` foi passado para a função `MostraVector`, sem problema. Pode-se na função `main` abstrair dos detalhes não só da inicialização do vector, como também da sua criação.

Toda esta flexibilidade não vem sem um preço. Temos agora uma nova dimensão nos erros possíveis com memória, ao ponto de outras linguagens de programação tomarem o seu espaço devido a terem formas automáticas de evitar parte destes problemas.

Vamos considerar os seguintes erros:

- Ler uma posição de memória não alocada;
- Alterar uma posição de memória não alocada (e possivelmente em uso);
- Não libertar memória após alocar;
- Libertar memória de um endereço inválido:
 - Endereço nulo;
 - Endereço não alocado com o `malloc` (pode ser do `stack`, ou conter lixo);
 - Endereço já libertado anteriormente.

O primeiro erro pode acontecer da seguinte maneira: pode-se por algum motivo não alocar memória (por falta de memória ou por não chamar a função `malloc`), e mesmo assim ler-se o conteúdo:

```

27     ...
28     int main()
29     {
30         char *texto=NULL, str[MAXSTR];
31         do {
32             gets(str);
33             texto=Concatenar(texto,str);
34         } while(strlen(str)>0);
35         free(texto);

```

```

36     texto=NULL;
37     printf("%s",texto);
38 }

```

Execução de Exemplo:

```

C:\>strings2
primeira linha
segunda linha
terceira linha
(null)

```

Repare-se que neste caso, a string foi libertada e atribuído o valor NULL, e depois foi tentado aceder à string na posição de memória não alocada. O texto impresso é "(null)". De onde vem este texto? Como já foi referido no capítulo Procedimentos, são estes realmente os caracteres que estão nos primeiros bytes de memória, dado que este erro é comum e desta forma o programador sabe que está a imprimir um apontador para NULL.

Outra forma deste erro ocorrer é pedir-se uma posição de memória acima do vector que foi alocado. Vamos supor que no exemplo dos vectores aleatórios, a função `MostraVector` tinha um número desfasado relativamente ao índice correcto.

```

15 ...
16 void MostraVector(int *vector, int n)
17 {
18     int i;
19     for(i=1;i<=n;i++)
20         printf("%d ",vector[i]);
21 }
22 ...

```

Execução de exemplo:

```

C:\>vectoraleatorio2
Dimensao: 10
Valor maximo: 10
7 4 0 9 4 8 8 2 4 393364

```

O último valor não foi gerado, foi lido de uma posição de memória não alocada, e estava o valor 393364 como poderia estar outro qualquer, depende da última utilização desse endereço de memória.

Este tipo de erro normalmente não leva ao crash da aplicação, excepto se o valor lido incorrecto for o de um apontador, dado que nesse caso pode-se de seguida ler ou escrever outros valores a partir desse apontador que está incorrecto.

O segundo erro tem normalmente efeitos mais devastadores que o primeiro, dado que pode ter consequências num local completamente distinto do local onde está o erro. O endereço pode ser no espaço em que está código, o que tem consequências imprevisíveis. Vamos supor que ao alocar espaço para as strings, não se contava com o carácter terminador.

```

6 ...
7 char *Concatenar(char *str, char *str2)
8 {
9     char *pt;
10    /* duplicar str2 se str é nulo */
11    if(str==NULL)
12    {
13        pt=(void *)malloc(strlen(str2));
14        if(pt!=NULL)
15            strcpy(pt,str2);
16    } else {
17        pt=(void *)malloc(strlen(str)+strlen(str2));
18        if(pt!=NULL)
19        {
20            strcpy(pt,str);
21            strcat(pt,str2);
22            free(str);
23        }
24    }
25    return pt;
26 }
27 ...

```

Execução de exemplo:

```

C:\>strings3
primeira linha
segunda linha
terceira linha
primeira linhasegunda linhaterceira linha

```

Por acaso não houve problema, porque o espaço de memória utilizado indevidamente não foi utilizado para mais nada. Vamos agravar a situação, supondo que se escrevia para a posição de memória NULL, o que pode acontecer se não se verificar se a operação de alocação de memória foi bem sucedida.

```

6 ...
7 char *Concatenar(char *str, char *str2)
8 {
9     char *pt;
10    /* duplicar str2 se str é nulo */
11    if(str==NULL)
12    {
13        pt=(void *)malloc(strlen(str2)+1);
14        if(pt!=NULL)
15            strcpy(pt,str2);
16    } else {
17        pt=(void *)malloc(strlen(str)+strlen(str2)+1);
18        if(pt!=NULL)
19        {
20            pt=NULL;
21            strcpy(pt,str);
22            strcat(pt,str2);
23            free(str);
24        }
25    }
26    return pt;
27 }
28 ...

```

Execução de exemplo:

```
C:\>strings4
primeira linha
segunda linha
"An unhandle win32 exception occurred in strings4.exe [1488]"
```

A mensagem de erro varia conforme o sistema operativo, e a sua forma de lidar com estas situações. O programa no entanto é que deveria ter o cuidado de evitar que esta situação ocorra.

O terceiro erro ocorre quando se cria memória em diversos locais, mas para cada bloco não é garantido que seja chamada a função de libertação free. O apontador para o bloco pode até ficar com outro bloco, ficando blocos de memória perdidos.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define MAXSTR 255
6
7 char *Concatenar(char *str, char *str2)
8 {
9     char *pt;
10    /* duplicar str2 se str é nulo */
11    if(str==NULL)
12    {
13        pt=(void *)malloc(strlen(str2)+1);
14        if(pt!=NULL)
15            strcpy(pt, str2);
16    } else {
17        pt=(void *)malloc(strlen(str)+strlen(str2)+1);
18        if(pt!=NULL)
19        {
20            strcpy(pt, str);
21            strcat(pt, str2);
22        }
23    }
24    return pt;
25 }
26
27 int main()
28 {
29     char *texto=NULL, str[MAXSTR];
30     do {
31         gets(str);
32         texto=Concatenar(texto, str);
33     } while(strlen(str)>0);
34     printf("%s", texto);
35 }
```

Execução de exemplo:

```
C:\>strings5
primeira linha
segunda linha
terceira linha
primeira linhasegunda linhaterceira linha
```

Nenhuma das alocações é libertada. Ficam blocos de memória alocados perdidos até que a aplicação acabe. Não há erro da aplicação. Se existirem muitos blocos de memória não libertos, a memória acabará por se esgotar, ou pelo menos, a aplicação utiliza mais memória que necessita, e que poderia ser utilizada em outras aplicações. Nas aplicações que correm durante muito tempo, por exemplo em servidores, não pode haver nenhum bloco alocado não liberto, porque com o tempo de corrida acabará sempre por esgotar a memória disponível. Para que este problema não ocorra, tem

que se ter em mente quando se aloca memória, o local onde esse bloco será liberto, nem que seja no final do programa.

O quarto erro é o que tem efeitos normalmente de crash da aplicação, mas o efeito varia conforme o ambiente em que corre. Em todo o caso deve-se evitar que ocorram.

Libertação de o apontador para NULL:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     free(NULL);
7     printf("Libertacao ok");
8 }
```

Libertação de um apontador para um vector no stack:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int vector[10];
7     free(vector);
8     printf("Libertacao ok");
9 }
```

Libertação dupla do mesmo bloco de memória:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int *vector;
7     vector=(int*)malloc(sizeof(int)*10);
8     free(vector);
9     free(vector);
10    printf("Libertacao ok");
11 }
```

A última forma de erro é normalmente a mais comum. Com medo de que um bloco não seja liberto, por vezes há mais que um local onde o mesmo bloco é liberto. Uma boa prática é após libertar um bloco de memória, atribuir o apontador para NULL ou outro valor. Como o anterior bloco de memória já não é válido, e poderá ser utilizado em outra função, não faz sentido manter um apontador a referenciá-lo.

A linguagem C não faz testes de acessos fora dos vectores ou a posições não alocadas. Dada a sua característica de estar perto da máquina, não seria lícito retirar tempo de processamento numa operação tão comum como o acesso a uma posição de memória, pelo que é o programador que deve garantir que o seu programa não acede fora das zonas alocadas. Os sistemas operativos modernos fazem no entanto essas verificações ao nível do programa, de forma a garantir que um programa não acede a espaço de memória que não lhe pertence, o que a acontecer poderia representar um risco de segurança.

Erros comuns mais directamente associados a este capítulo:

- **Atribuições entre variáveis de tipos distintos**
 - **Forma:** Principalmente quando se utiliza apontadores, por vezes é necessário fazer atribuições entre apontadores de tipos distintos, ou entre apontadores e inteiros. Isso não é problema porque um apontador é um número de 32 bits.
 - **Problema:** Se atribui um apontador de um tipo a outro, este vai aceder ao endereço de memória apontado como se fosse o segundo tipo, quando este foi alocado e tem dados como se fosse o primeiro tipo. Pode ter consequências imprevisíveis, uma vez que os dados ficam com o acesso incoerente. Em caso algum deve assumir o tipo do apontador, uma vez que tal pode variar conforme o compilador/computador em que o código for utilizado
 - **Resolução:** Se for necessária atribuição entre apontadores de tipos distintos, ao alocar memória por exemplo, então deve utilizar no código um *cast* para o tipo de destino, de forma a clarificar que não se trata de um erro. Fora a situação de alocação de memória, não precisa de qualquer outra atribuição entre tipos distintos. Em situações menos ortodoxas, pode utilizar o tipo *union*, para fundir dados de mais de um tipo
- **Não libertar memória após alocar**
 - **Forma:** Atendendo a que não há erros por não libertar memória, e sendo a aplicação de curta duração, justifica-se a não libertação da memória, já que quando a aplicação terminar tudo é libertado.
 - **Problema:** A aplicação tem os endereços dos blocos de memória que alocou, já não precisa deles, pode libertá-los. Se não o fizer, não há garantia da altura em que esses blocos serão libertos, nem se o custo computacional de os libertar é o mesmo. O argumento da pequena aplicação não deve ser justificação para qualquer erro, o código que funciona em pequenas aplicações poderá ser reutilizado em outras aplicações, e se for bem feito, não necessitará de alterações
 - **Resolução:** Tenha sempre atenção ao alocar memória do local exacto onde esse bloco será certamente libertado
- **Alocar memória e não testar se a operação foi bem sucedida**
 - **Forma:** A alocação de memória é uma operação comum, e se o programa não utiliza grandes quantidades de memória, é sempre bem sucedida, pelo que não vale a pena efectuar a verificação já que complica desnecessariamente o código.
 - **Problema:** Mesmo que a aplicação utilize pouca memória, a alocação de memória pode falhar se o sistema operativo tiver a memória esgotada por outras aplicações, ou por outras instâncias da mesma aplicação. A não verificação pode provocar acesso a zonas de memória proibidas com resultados imprevisíveis
 - **Resolução:** Em todas as alocações de memória, verifique se estas foram bem sucedidas, e no caso de falharem, certifique-se que o programa tem um procedimento válido

11. ESTRUTURAS

Cada variável é identificada por um nome. Quantas forem necessárias, quantas são criadas. No entanto, por vezes há variáveis que estão relacionadas entre si.

No exemplo do número de dias do mês, o ano e mês estão relacionadas uma vez que fazem parte de uma data. No entanto a linguagem C não tem o tipo data, daí ter-se de utilizar o código utilizado no capítulo Funções:

```
1 #include <stdio.h>
2
3 int Bissexto(int ano)
4 {
5     return ano%400==0 || ano%4==0 && ano%100!=0;
6 }
7
8 int DiasDoMes(int mes, int ano)
9 {
10     if(mes==2)
11     {
12         /* teste de ano bissexto */
13         if(Bissexto(ano))
14             return 29;
15         else
16             return 28;
17     } else if(mes==1 || mes==3 || mes==5 || mes==7 ||
18             mes==8 || mes==10 || mes==12)
19     {
20         return 31;
21     } else
22     {
23         return 30;
24     }
25 }
26
27 int main()
28 {
29     int ano, mes, dias;
30     printf("Indique ano: ");
31     scanf("%d", &ano);
32     printf("Indique mes: ");
33     scanf("%d", &mes);
34
35     printf("%d", DiasDoMes(ano,mes));
36 }
```

Programa 11-1 Réplica do Programa 5-4

Se em vez de uma data, que tem três valores numéricos, pretendermos guardar por exemplo informação sobre uma pessoa, com o nome, sexo, data de nascimento, morada, email e telefone, já seriam muitas variáveis relacionadas. Se pretender que o programa registe várias pessoas, então tem-se de ter vários vectores em paralelo para guardar esta informação. Evidentemente que esta solução torna-se facilmente impraticável, tanto para passar a informação relativa a uma pessoa para uma função, como para guardar vários registos de uma pessoa.

As estruturas pretendem precisamente resolver este problema, permitindo agrupar variáveis relacionadas entre si.

```
struct
{
    int ano, mes, dia;
} data;
```

A variável `data` é declarada não como sendo o tipo `int`, mas como sendo do tipo:

```
struct { int ano, mes, dia; }
```

O acesso à variável deve ser feito a cada campo da variável, através do operador `"."`:

```
data.ano=2010;
data.mes=12;
data.dia=1;
```

Existe não apenas uma variável `data`, mas sim três variáveis: `data.ano`, `data.mes`, `data.dia`¹¹. No entanto pode-se referenciar a variável `data`, e copiar por exemplo o conteúdo das três variáveis, para outra variável do mesmo tipo:

```
/* copia todos os campos da variável "data" para os campos
da variável "data2" */
data2=data;
```

A variável `data2` teria de ser declarada da mesma forma, tendo que se repetir os campos. Para evitar esta tarefa, e eventuais erros nos nomes dos campos, pode-se dar nome à estrutura e assim declara-se apenas uma só vez a estrutura, fora de funções, e esta passa a estar disponível para declarações de variáveis desse tipo:

```
struct SData
{
    int ano, mes, dia;
};
struct SData data, data2;
```

Podemos agora reescrever o exemplo do número de dias do mês:

```
1 #include <stdio.h>
2
3 struct SData
4 {
5     int ano, mes, dia;
6 };
7
8 int Bissexto(int ano)
9 {
10     return ano%400==0 || ano%4==0 && ano%100!=0;
11 }
12
13 int DiasDoMes(struct SData data)
14 {
15     if(data.mes==2)
16     {
17         /* teste de ano bissexto */
18         if(Bissexto(data.ano))
19             return 29;
```

¹¹ Inicialização da estrutura na declaração da variável: `struct {int ano,mes,dia;} data={2010,12,1};` colocam-se os valores de cada campo da estrutura pela mesma ordem com que estão declarados.

```

20         else
21             return 28;
22     } else if(data.mes==1 || data.mes==3 || data.mes==5 || data.mes==7 ||
23         data.mes==8 || data.mes==10 || data.mes==12)
24     {
25         return 31;
26     } else
27     {
28         return 30;
29     }
30 }
31
32 int main()
33 {
34     struct SData data;
35     int dias;
36     printf("Indique ano: ");
37     scanf("%d", &data.ano);
38     printf("Indique mes: ");
39     scanf("%d", &data.mes);
40
41     printf("%d", DiasDoMes(data));
42 }

```

Programa 11-2 Calculo dos dias de um mês/ano, utilizando estruturas

Uma estrutura pode estar dentro de outra. Vejamos o registo de uma pessoa, que utiliza a data de nascimento:

```

1  #include <stdio.h>
2
3  #define MAXSTR 255
4
5  struct SData
6  {
7      int ano, mes, dia;
8  };
9
10 struct SPessoa
11 {
12     char *nome;
13     char sexo;
14     struct SData nascimento;
15     char *morada;
16     char *email;
17     long telefone;
18 };
19
20 struct SPessoa LerPessoa()
21 {
22     struct SPessoa pessoa;
23     char str[MAXSTR];
24     printf("Nome: ");
25     gets(str);
26     pessoa.nome=(char*)malloc(strlen(str)+1);
27     strcpy(pessoa.nome, str);
28     printf("Sexo: ");
29     scanf("%c", &pessoa.sexo);
30     printf("Data de nascimento (dd-mm-aaaa): ");
31     scanf("%d-%d-%d",
32         &pessoa.nascimento.dia,
33         &pessoa.nascimento.mes,
34         &pessoa.nascimento.ano);
35     printf("Morada: ");
36     gets(str);
37     gets(str);
38     pessoa.morada=(char*)malloc(strlen(str)+1);
39     strcpy(pessoa.morada, str);
40     printf("Email: ");
41     gets(str);
42     pessoa.email=(char*)malloc(strlen(str)+1);

```

```

43     strcpy(pessoa.email, str);
44     printf("Telefone: ");
45     scanf("%ld", &pessoa.telefone);
46     return pessoa;
47 }
48
49 void MostraPessoa(struct SPessoa pessoa)
50 {
51     printf("\nNome: %s\nSexo: %c\nData de Nascimento: %d-%d-%d\n",
52           pessoa.nome, pessoa.sexo,
53           pessoa.nascimento.dia,
54           pessoa.nascimento.mes,
55           pessoa.nascimento.ano);
56     printf("Morada: %s\nEmail: %s\nTelefone: %d\n",
57           pessoa.morada,
58           pessoa.email,
59           pessoa.telefone);
60 }
61
62 void Libertar(struct SPessoa pessoa)
63 {
64     free(pessoa.nome);
65     free(pessoa.morada);
66     free(pessoa.email);
67 }
68
69 int main()
70 {
71     struct SPessoa pessoa;
72     pessoa=LerPessoa();
73     MostraPessoa(pessoa);
74     Libertar(pessoa);
75 }

```

Programa 11-3 Registo de uma estrutura com dados de diferentes tipos de pessoas

Execução de exemplo:

```

C:\>pessoa
Nome: Joaquim Lima
Sexo: M
Data de nascimento (dd-mm-aaaa): 1-1-1980
Morada: Avenida da Liberdade n.1
Email: joaquim@gmail.com
Telefone: 123456789
Nome: Joaquim Lima
Sexo: M
Data de Nascimento: 1-1-1980
Morada: Avenida da Liberdade n.1
Email: joaquim@gmail.com
Telefone: 123456789

```

Não é necessário passar as estruturas por referência nos argumentos das funções, e pode-se retornar uma estrutura sem problema, como se fosse do tipo inteiro. Todas as variáveis da estrutura são copiadas, tanto na passagem de parâmetros como ao retornar da função. Como há apenas uma variável pessoa, para poupar tempo de cópia é conveniente passar a estrutura por referência, de forma a ser copiado apenas um apontador e não todos os campos da estrutura.

Para aceder a um campo de uma estrutura, utiliza-se o **operador** `"."`. Se em vez da estrutura utilizarmos um apontador, tem que se aceder a um campo primeiro com o **operador** `*` para identificar o conteúdo, e depois o ponto: **(*pessoa).nome**. Este tipo de anotação pode baixar a legibilidade do código, no caso de haver muitas estruturas passadas por referência, como é esperado. Para evitar essa situação, a linguagem C tem um **operador** `->`, equivalente: `pessoa->nome`

Pode-se estar interessado em declarar uma variável de um tipo "inteiro", e não estar interessado sequer em saber o tipo concreto do inteiro (long, short, int), e querer poder decidir mais tarde, mas que tal não force a edição de todas as declarações de variáveis desse tipo.

Embora as estruturas permitam criar facilmente variáveis de tipos complexos, o termo `struct` tem de ser utilizado sempre que se declara uma variável complexa, e seria interessante em vez de ter `struct SData variavel`; se tivesse `TData variavel`;

A utilização de macros poderia ser uma opção para resolver estes dois problemas embora de risco muito elevado, dado que podem ser substituídos textos em locais impróprios. Os tipos das variáveis têm tendência para serem curtos, dado que são normalmente conceitos simples e claros, pelo que o risco de substituições indevidas aumenta.

O **typedef** serve precisamente para resolver este problema, sem o risco de trocas acidentais que a utilização de macros teria.

```
/* a partir deste momento, existe o tipo inteiro, que é um int */  
typedef int inteiro;
```

Notar que se existir a necessidade de em outro compilador/computador o inteiro ser do tipo long, ou short, basta editar esta linha e todo o código fica actualizado.

```
/* TData passa a ser um tipo struct SData */  
typedef struct SData TData;
```

O efeito é o pretendido. No entanto o identificador `SData` foi criado apenas para ser utilizado no `typedef`. Tal não é necessário, pode-se simplesmente utilizar o `typedef` logo na declaração da estrutura:

```
typedef struct  
{  
    int ano, mes, dia;  
} TData;
```

Vejamos o exemplo modificado com passagem de estruturas por referência, operador `->`, e `typedef`:

```
1 #include <stdio.h>  
2  
3 #define MAXSTR 255  
4  
5 typedef struct  
6 {  
7     int ano, mes, dia;  
8 } TData;  
9  
10 typedef struct  
11 {  
12     char *nome;  
13     char sexo;  
14     TData nascimento;  
15     char *morada;  
16     char *email;  
17     long telefone;  
18 } TPessoa;  
19  
20 void LerPessoa(TPessoa *pessoa)  
21 {  
22     char str[MAXSTR];
```

```

23     printf("Nome: ");
24     gets(str);
25     pessoa->nome=(char*)malloc(strlen(str)+1);
26     strcpy(pessoa->nome,str);
27     printf("Sexo: ");
28     scanf("%c",&pessoa->sexo);
29     printf("Data de nascimento (dd-mm-aaaa): ");
30     scanf("%d-%d-%d",
31           &pessoa->nascimento.dia,
32           &pessoa->nascimento.mes,
33           &pessoa->nascimento.ano);
34     printf("Morada: ");
35     gets(str);
36     gets(str);
37     pessoa->morada=(char*)malloc(strlen(str)+1);
38     strcpy(pessoa->morada,str);
39     printf("Email: ");
40     gets(str);
41     pessoa->email=(char*)malloc(strlen(str)+1);
42     strcpy(pessoa->email,str);
43     printf("Telefone: ");
44     scanf("%ld",&pessoa->telefone);
45 }
46
47 void MostraPessoa(TPessoa *pessoa)
48 {
49     printf("\nNome: %s\nSexo: %c\nData de Nascimento: %d-%d-%d\n",
50           pessoa->nome, pessoa->sexo,
51           pessoa->nascimento.dia,
52           pessoa->nascimento.mes,
53           pessoa->nascimento.ano);
54     printf("Morada: %s\nEmail: %s\nTelefone: %d\n",
55           pessoa->morada,
56           pessoa->email,
57           pessoa->telefone);
58 }
59
60 void Libertar(TPessoa *pessoa)
61 {
62     free(pessoa->nome);
63     free(pessoa->morada);
64     free(pessoa->email);
65 }
66
67 int main()
68 {
69     TPessoa pessoa;
70     LerPessoa(&pessoa);
71     MostraPessoa(&pessoa);
72     Libertar(&pessoa);
73 }

```

Programa 11-4 Versão do registo de pessoas, mais legível e eficiente

Esta versão é mais legível, e também mais eficiente.

O tipo **TPessoa** é útil apenas para a aplicação em desenvolvimento, e a existir em outras aplicações será certamente diferente. Pelo contrário, o tipo **TData** pode ser reutilizado em várias aplicações. Vamos introduzir um tipo com um potencial de reutilização superior à data: **vector**. Um vector normalmente está associado à sua dimensão, a qual tem que ser transportada num parâmetro à parte para dentro das funções, devendo ser utilizado esse valor para percorrer todo o vector, ou testar se o índice pedido está alocado ou não. Temos portanto um potencial para definir um tipo com uma alta taxa de reutilização.

```

typedef struct
{
    int n;

```

```
    int *valor;
} TVectorInt;
```

Vamos refazer o exemplo do vector aleatório com base nesta estrutura:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct
5  {
6      int n;
7      int *valor;
8  } TVectorInt;
9
10 TVectorInt vectorAleatorio(int n, int base)
11 {
12     TVectorInt vector;
13     int i;
14     vector.valor=(void*)malloc(sizeof(int)*n);
15     if(vector.valor!=NULL)
16     {
17         vector.n=n;
18         for(i=0;i<n;i++)
19             vector.valor[i]=rand()%base;
20     } else
21         vector.n=0;
22     return vector;
23 }
24
25 void MostraVector(TVectorInt vector)
26 {
27     int i;
28     for(i=0;i<vector.n;i++)
29         printf("%d ",vector.valor[i]);
30 }
31
32 int main()
33 {
34     TVectorInt vector;
35     int n,base;
36     srand(1);
37
38     printf("Dimensao: ");
39     scanf("%d",&n);
40     printf("Valor maximo: ");
41     scanf("%d",&base);
42
43     vector=vectorAleatorio(n,base);
44     if(vector.valor!=NULL)
45     {
46         MostraVector(vector);
47         free(vector.valor);
48     }
49 }
```

Programa 11-5 Estrutura de um vector com os valores e a sua dimensão

Este exemplo foi melhorado no sentido em que se poupou a passagem para `MostraVector` de um segundo argumento relacionado com o vector, a sua dimensão, que assim está mesmo associado ao vector. Note-se que neste caso a estrutura é passada por valor, uma vez que tem apenas dois campos, nunca sendo o vector em si copiado.

Infelizmente as tarefas específicas desta estrutura continuam espalhadas pelo código:

- Alocação do vector
- Acesso a uma posição do vector
- Libertação do vector

As estruturas ao permitirem o encapsulamento de dados são a base para a **abstracção de dados**, também chamado de **tipos abstractos de dados**. Embora este conceito tenha ficado obsoleto com o conceito de classe da programação orientada por objectos, é também a sua base, pelo que a boa utilização de tipos abstractos de dados, será o suporte para a compreensão de classes. Na linguagem C esta é a única ferramenta de controle de complexidade dos dados.

Tal como a abstracção funcional permite que se utilize uma função sem saber como está implementada, e ao implementar a função não necessitamos de saber como será utilizada, permitindo desta forma controlar a complexidade de um programa, e implementar um programa de dimensão arbitrária, os tipos abstractos de dados permitem que se utilize dados sem saber como estão realmente declarados, e que ao declarar um tipo abstracto de dados, não se esteja a preocupar como o tipo será utilizado.

Um tipo abstracto de dados consiste em definir não só uma estrutura, como também um conjunto de funções para acesso à estrutura, não devendo a estrutura ser acedida por qualquer outra função directamente. Quando for necessário alterar a estrutura, o resto do programa ficará a funcionar, não sendo necessário revê-lo, e o mesmo se passa com o resto do programa, pode utilizar os métodos de acesso sem se preocupar como estão implementados.

```
TVectorInt VICriar(int);
int VITamanho(TVectorInt);
int VIV valorO(TVectorInt, int);
void VIV valorI(TVectorInt, int, int);
void VILibertar(TVectorInt*);
```

Estas funções são as necessárias para conter a complexidade de dados associada ao TVectorInt no código apresentado. Desde que se verifique a condição de que nenhuma outra função acede directamente à estrutura de dados, e que estas funções de acesso não têm nada específico da aplicação, será sempre um bom tipo abstracto de dados. Apenas desta forma este tipo abstracto de dados poderá ser reutilizado em outras aplicações, e fornecer realmente uma abstracção de dados.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* Tipo de dados abstracto: TVectorInt */
5 typedef struct
6 {
7     int n;
8     int *valor;
9 } TVectorInt;
10
11 TVectorInt VICriar(int n)
12 {
13     TVectorInt vector;
14     vector.valor=(void*)malloc(sizeof(int)*n);
15     if(vector.valor!=NULL)
16         vector.n=n;
17     else
18         vector.n=0;
19     return vector;
20 }
21
22 int VITamanho(TVectorInt vector)
23 {
24     return vector.n;
25 }
26
27 void VIV valorI(TVectorInt vector, int i, int valor)
28 {
29     if(i>=0 && i<vector.n)
```



```

30     vector.valor[i]=valor;
31 }
32
33 int VIValorO(TVectorInt vector, int i)
34 {
35     if(i>=0 && i<vector.n)
36         return vector.valor[i];
37     return 0;
38 }
39
40 void VILibertar(TVectorInt *vector)
41 {
42     if(vector->valor!=NULL)
43     {
44         free(vector->valor);
45         vector->valor=NULL;
46         vector->n=0;
47     }
48 }
49
50 /* Programa */
51
52 TVectorInt VectorAleatorio(int n, int base)
53 {
54     TVectorInt vector;
55     int i;
56
57     vector=VICriar(n);
58
59     for(i=0;i<VITamanho(vector);i++)
60         VIValorI(vector,i,rand()%base);
61
62     return vector;
63 }
64
65 void MostraVector(TVectorInt vector)
66 {
67     int i;
68     for(i=0;i<VITamanho(vector);i++)
69         printf("%d ",VIValorO(vector,i));
70 }
71
72 int main()
73 {
74     TVectorInt vector;
75     int n,base;
76     srand(1);
77
78     printf("Dimensao: ");
79     scanf("%d",&n);
80     printf("Valor maximo: ");
81     scanf("%d",&base);
82
83     vector=VectorAleatorio(n,base);
84     MostraVector(vector);
85     VILibertar(&vector);
86 }

```

Programa 11-6 Tipo abstracto de dados TVectorInt

Na implementação das funções `VIValorI` e `VIValorO` de acesso ao vector foi colocado um teste de validade, podendo assim num só local controlar se esse teste deve ou não existir. Esta opção não seria possível sem um tipo abstracto de dados. Na função `VILibertar` foram tomadas medidas para que o vector não seja libertado se não estiver alocado, ou se já foi libertado, sendo a única função a passar a estrutura por referência, uma vez que altera o valor dos seus campos. O programa em si, ao ficar liberto das especificidades do tipo de dados abstracto, é também simplificado.

Este tipo pode no entanto ser melhorado, dado que para criar o vector tem de se saber o seu tamanho, sendo os pedidos de acesso fora do espaço inicialmente criado recusados. O procedimento poderia ser distinto, por exemplo, não ser necessário definir o tamanho do vector inicialmente, e o vector ser realocado sempre que necessário. Adicionava-se alguma complexidade ao tipo abstracto de dados, mas o programa ficaria liberto da necessidade de especificar a dimensão do vector.

Foi adicionado ao programa código para medir o tempo utilizado na criação do vector, de forma a podermos medir a sua performance.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 /* Tipo de dados abstracto: TVectorInt */
6 typedef struct
7 {
8     int n;
9     int *valor;
10 } TVectorInt;
11
12 TVectorInt VICriar()
13 {
14     TVectorInt vector;
15     vector.valor=NULL;
16     vector.n=0;
17     return vector;
18 }
19
20 int VITamanho(TVectorInt *vector)
21 {
22     return vector->n;
23 }
24
25 void VIInternoRealocar(TVectorInt *vector, int i)
26 {
27     int k, *vectorAntigo;
28     vectorAntigo=vector->valor;
29     vector->valor=(int*)malloc(sizeof(int)*(i+1));
30     if(vector->valor!=NULL)
31     {
32         for(k=0;k<vector->n;k++)
33             vector->valor[k]=vectorAntigo[k];
34         vector->n=i+1;
35     } else
36         vector->n=0;
37     if(vectorAntigo!=NULL)
38         free(vectorAntigo);
39 }
40
41 void VIVvalorI(TVectorInt *vector, int i, int valor)
42 {
43     /* acesso fora dos parâmetros, realocar */
44     if(i>=vector->n)
45         VIInternoRealocar(vector,i);
46
47     if(i>=0 && i<vector->n)
48         vector->valor[i]=valor;
49 }
50
51 int VIVvalorO(TVectorInt *vector, int i)
52 {
53     /* não colocar a realocação aqui, já que esta operação é de
54        leitura e não de escrita */
55     if(i>=0 && i<vector->n)
56         return vector->valor[i];
57     return 0;
```

```

58 }
59
60
61 void VILibertar(TVectorInt *vector)
62 {
63     if(vector->valor!=NULL)
64     {
65         free(vector->valor);
66         vector->valor=NULL;
67         vector->n=0;
68     }
69 }
70
71 /* Programa */
72
73 TVectorInt VectorAleatorio(int n, int base)
74 {
75     TVectorInt vector;
76     int i;
77     vector=VICriar();
78     for(i=0;i<n;i++)
79         VIValorI(&vector,i,rand()%base);
80     return vector;
81 }
82
83 void MostraVector(TVectorInt *vector)
84 {
85     int i;
86     for(i=0;i<VITamanho(vector);i++)
87         printf("%d ",VIValorO(vector,i));
88 }
89
90 int main()
91 {
92     TVectorInt vector;
93     int n,base;
94     clock_t instante;
95     srand(1);
96
97     printf("Dimensao: ");
98     scanf("%d",&n);
99     printf("Valor maximo: ");
100    scanf("%d",&base);
101
102    instante=clock();
103    vector=VectorAleatorio(n,base);
104    if(n<100)
105        MostraVector(&vector);
106    VILibertar(&vector);
107    printf("Tempo (s): %.3f", (float)(clock()-instante)/CLOCKS_PER_SEC);
108 }

```

Programa 11-7 Versão de TVectorInt com apontadores, e controle do tempo

Todos os métodos do tipo abstracto de dados foram passados para referência, dado que é necessário realocação em `VIValorI`, e para não estar metade dos métodos com a estrutura por valor e outra metade por referência, colocou-se tudo em referência.

Feito o tipo robusto, agora vamos optimizá-lo. Atendendo a que o tempo de realocação força a que todos os elementos sejam copiados para o novo vector, para adicionar elementos de 1 a 1, para um vector de tamanho N , o número de cópias a realizar é da ordem de N^2 . O primeiro elemento é copiado N vezes, o segundo elemento é copiado $N-1$ vezes, etc.

Vamos executar o código com valores elevados, de forma a ter uma ideia do tempo utilizado pela via experimental.

Execução de exemplo:

```
C:\>vectoraleatorio5
Dimensao: 1000
Valor maximo: 100
Tempo (s): 0.016
C:\>vectoraleatorio5
Dimensao: 2000
Valor maximo: 100
Tempo (s): 0.032
C:\>vectoraleatorio5
Dimensao: 10000
Valor maximo: 100
Tempo (s): 0.390
C:\>vectoraleatorio5
Dimensao: 20000
Valor maximo: 100
Tempo (s): 1.343
C:\>vectoraleatorio5
Dimensao: 100000
Valor maximo: 100
Tempo (s): 31.375
```

É clara a drástica mudança ao atingir valores mais elevados. Se com valores até 10000 os tempos são desprezáveis e aparentemente com crescimento linear, os valores acima de 20000 são bastante significativos e crescimento quadrático.

Uma forma de melhorar a performance, uma vez que se tivéssemos alocado logo o vector com a dimensão final, o número de cópias era apenas N, é sempre que o vector não chega, alocar logo o dobro do espaço necessário. Assim, o número de realocações baixa consideravelmente, sendo num vector de N elementos, cada elemento quanto muito é atribuído uma vez e copiado outra vez, ou seja, da ordem de N operações. Para fazer isto, é necessário ter a dimensão alocada do vector, e a dimensão lógica do vector. Como utilizamos tipos abstractos de dados, podemos fazer as alterações necessárias para que o programa fique com esta optimização, alterando apenas funções do tipo abstracto de dados.

Sendo este o primeiro teste da abstracção de dados, é de total importância que não seja necessário alterar o programa, caso contrário confirmaríamos que a abstracção de dados não estava correctamente definida.

```
5 /* Tipo de dados abstracto: TVectorInt */
6 typedef struct
7 {
8     int n,alocado;
9     int *valor;
10 } TVectorInt;
11
12 TVectorInt VICriar()
13 {
14     TVectorInt vector;
15     vector.valor=NULL;
16     vector.n=0;
17     vector.alocado=0;
18     return vector;
19 }
20
21 int VITamanho(TVectorInt *vector)
```

```

22 {
23     return vector->n;
24 }
25
26 void VIInternoRealocar(TVectorInt *vector, int i)
27 {
28     int k, *vectorAntigo;
29     vectorAntigo=vector->valor;
30     /* alocar o dobro do necessário */
31     vector->valor=(int*)malloc(sizeof(int)*(i+1)*2);
32     if(vector->valor!=NULL)
33     {
34         for(k=0;k<vector->n;k++)
35             vector->valor[k]=vectorAntigo[k];
36         vector->alocado=(i+1)*2;
37     } else
38         vector->alocado=0;
39     if(vectorAntigo!=NULL)
40         free(vectorAntigo);
41 }
42
43 void VIValorI(TVectorInt *vector, int i, int valor)
44 {
45     /* acesso fora dos parâmetros, realocar */
46     if(i>=vector->alocado)
47         VIInternoRealocar(vector,i);
48
49     if(i>=0 && i<vector->alocado)
50     {
51         vector->valor[i]=valor;
52         /* atualizar a dimensão lógica do vector */
53         if(i>=vector->n)
54             vector->n=i+1;
55     }
56 }
57
58 int VIValorO(TVectorInt *vector, int i)
59 {
60     /* não colocar a realocação aqui, já que esta operação é de
61        leitura e não de escrita */
62     if(i>=0 && i<vector->n)
63         return vector->valor[i];
64     return 0;
65 }
66
67
68 void VILibertar(TVectorInt *vector)
69 {
70     if(vector->valor!=NULL)
71     {
72         free(vector->valor);
73         vector->valor=NULL;
74         vector->alocado=0;
75         vector->n=0;
76     }
77 }

```

Programa 11-8 Ajustes feitos no tipo abstracto de dados TVectorInt, para melhoria da eficiência

Foi adicionado um campo, e feitos os ajustes necessários, apenas no tipo abstracto de dados. Vamos agora confirmar pela via experimental a vantagem desta mudança.

Execução de exemplo:

```
C:\>vectoraleatorio6  
Dimensao: 1000  
Valor maximo: 100  
Tempo (s): 0.000  
C:\>vectoraleatorio6  
Dimensao: 10000  
Valor maximo: 100  
Tempo (s): 0.016  
C:\>vectoraleatorio6  
Dimensao: 100000  
Valor maximo: 100  
Tempo (s): 0.015  
C:\>vectoraleatorio6  
Dimensao: 1000000  
Valor maximo: 100  
Tempo (s): 0.078
```

Nem o valor mais alto do teste anterior, 100000, tem um tempo que se considere digno de alguma precisão. Foi necessário aumentar o valor para 1000000 para ter quase uma décima de segundo. Face a estes resultados, até a utilização da anterior implementação até 10000 valores deixa de ser razoável, dado que a alternativa é não gastar praticamente tempo nenhum. Esta solução tem no entanto o problema de desperdiçar até 50% do espaço alocado, o que pode não ser razoável em algumas aplicações.

Está introduzida a vantagem dos tipos abstractos de dados. Não seria correcto no entanto terminar esta secção, tendo já sido dada a alocação de memória, sem a introdução da estrutura de dados mais famosa que utiliza estruturas: **listas**.

As listas pretendem guardar uma sequência de elementos, tal como os vectores. No entanto, ao contrário destes, alocam um bloco de memória por cada elemento, não havendo necessidade de realocações, e por outro lado utilizam apenas pequenos blocos de memória, facilitando assim a tarefa do gestor de memória. Cada elemento tem de ter no entanto um apontador para o próximo elemento, pelo que esta estrutura tem de utilizar o tipo struct:

```
typedef struct SListaInt  
{  
    int valor;  
    struct SListaInt *seguinte;  
} TListaInt;
```

Repare-se que a definição da estrutura é recursiva. Não é portanto possível deixar de fazer um paralelo entre listas (recursivo) e vectores (iterativo), da mesma forma que existe funções recursivas (chamam-se a si próprias) e iterativas (processam os valores iterativamente).

Para identificar uma lista, basta ter um apontador para o primeiro elemento, sendo normalizado não só que a lista vazia é o apontador para NULL, como também o elemento seguinte ao último elemento é o valor NULL, ou seja, há uma lista vazia após o último elemento.

Como utilizamos o tipo abstracto de dados, mesmo com esta mudança radical na estrutura definida, podemos fazer a mudança envolvendo apenas o tipo abstracto de dados. Mas como as listas têm um acesso natural distinto dos vectores, vamos definir um tipo de dados abstracto mais adequado para a utilização de listas.

```

TListaInt* LIAdicionar(TListaInt *lista, int valor);
TListaInt *LILibertar(TListaInt *lista)

```

O acesso é feito nos próprios membros, também no programa, dado que o código no programa é simplificado graças a esta estrutura como se verá.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  /* Tipo de dados abstracto: TListaInt */
6  typedef struct SListaInt
7  {
8      int valor;
9      struct SListaInt *seguinte;
10 } TListaInt;
11
12 TListaInt* LIAdicionar(TListaInt *lista, int valor)
13 {
14     TListaInt *novo;
15     novo=(TListaInt*)malloc(sizeof(TListaInt));
16     if(novo!=NULL)
17     {
18         novo->valor=valor;
19         novo->seguinte=lista;
20         return novo;
21     }
22     return lista;
23 }
24
25 void LILibertar(TListaInt *lista)
26 {
27     TListaInt *aux;
28     while(lista!=NULL)
29     {
30         aux=lista->seguinte;
31         free(lista);
32         lista=aux;
33     }
34 }
35
36
37 /* Programa */
38
39 TListaInt *ListaAleatoria(int n, int base)
40 {
41     TListaInt *lista=NULL; /* nova lista */
42     int i;
43
44     for(i=0;i<n;i++)
45         lista=LIAdicionar(lista,rand()%base);
46
47     return lista;
48 }
49
50 void MostraLista(TListaInt *lista)
51 {
52     while(lista!=NULL)
53     {
54         printf("%d ", lista->valor);
55         lista=lista->seguinte;
56     }
57 }
58
59 int main()
60 {
61     TListaInt *lista;
62     int n,base;
63     clock_t instante;
64     srand(1);

```

```

65
66     printf("Dimensao: ");
67     scanf("%d",&n);
68     printf("Valor maximo: ");
69     scanf("%d",&base);
70
71     instante=clock();
72     lista=ListaAleatoria(n,base);
73     if(n<100)
74         MostraLista(lista);
75     LILibertar(lista);
76     printf("Tempo (s): %.3f", (float)(clock()-instante)/CLOCKS_PER_SEC);
77 }

```

Programa 11-9 Tipo abstracto de dados TListaInt

Repare-se na simplicidade do ciclo while, em `MostraLista`. Apenas são passados apontadores, e no método `LIAdicionar`, como a lista é alterada, tem de existir também uma atribuição para o valor retornado, que representa o início da lista, como se pode ver em `ListaAleatoria`. Alternativamente poder-se-ia ter enviado o endereço da lista, mas este formato considera-se mais simples e revelador que o topo da lista vai alterando.

Execução de exemplo:

```

C:\>listaaleatoria
Dimensao: 10
Valor maximo: 10
4 2 8 8 4 9 0 4 7 1 Tempo (s): 0.000
C:\>listaaleatoria
Dimensao: 1000
Valor maximo: 100
Tempo (s): 0.000
C:\>listaaleatoria
Dimensao: 10000
Valor maximo: 100
Tempo (s): 0.000
C:\>listaaleatoria
Dimensao: 100000
Valor maximo: 100
Tempo (s): 0.094
C:\>listaaleatoria
Dimensao: 1000000
Valor maximo: 100
Tempo (s): 0.578

```

Os valores em termos de tempos são ligeiramente superiores à melhor versão dos vectores, uma vez que o número de alocações de memória é superior, mas o número de operações é idêntico e não desperdiça memória excepto o espaço de um apontador por cada elemento. As listas constituem assim uma alternativa válida aos vectores, resultando em código mais simples em grande parte dos casos.

O preço a pagar pela utilização das listas, é no método de acesso aleatório: `LIVvalor(lista,i)`. Para implementar um método deste tipo, tem que se percorrer a lista desde o início até obter o valor na posição `i`, enquanto com vectores o acesso é imediato.

A **estrutura de dados** de uma aplicação é o conjunto de estruturas definidas que permitem carregar os dados da aplicação em memória, e suportam as operações necessárias. As opções tomadas na definição da estrutura de dados são as mais determinantes para que a aplicação seja facilmente implementada e mantida. No entanto, questões relacionadas com a estrutura de dados não são aprofundadas neste texto, dado que estas questões são mais relevantes apenas para

aplicações de média e grande dimensão. Aconselha-se apenas que antes de pensar no algoritmo que lhe resolve o problema, pensar nos dados que tem, e como os coloca em memória, ou seja, definir a estrutura de dados, e apenas depois nos algoritmos que necessita de implementar que utilizam a estrutura de dados para produzir o resultado pretendido. É normal que se sinta tentado a fazer o inverso, dado que os problemas tratados não têm tratado com estruturas de dados complexas. Para definir a estrutura de dados apenas necessita de conhecer o problema, para implementar um algoritmo, necessita de ter também a estrutura de dados definida.

12. FICHEIROS

Um programa pode fazer processamentos fantásticos, mas no final termina e tudo o que aconteceu, desaparece. Uma das grandes vantagens do computador, é a possibilidade guardar ficheiros com informação, ao longo do tempo, caso contrário ficaria idêntico a uma máquina de calcular. Os ficheiros servem para guardar a informação de forma mais permanente, de modo a poder ser utilizada mais tarde, pela mesma aplicação ou outra.

No entanto, ao executar um programa, todos os dados têm de ser introduzidos pelo teclado, e todos os dados processados são reproduzidos no ecrã. Seria interessante ter uma forma de indicar um ficheiro de origem dos dados, e eventualmente um ficheiro de saída dos dados.

É precisamente esse efeito que têm os **operadores de linha de comando** **< e >**:

```
C:\> prog < in.txt
```

O ficheiro `in.txt` será colocado na entrada de dados de `prog`.

Vejamos o exemplo do número de dias do mês já conhecido.

Execução de exemplo:

```
C:\>diasdomes  
Indique ano: 2010  
Indique mes: 2  
28
```

No ficheiro `in.txt` coloca-se o texto introduzido:

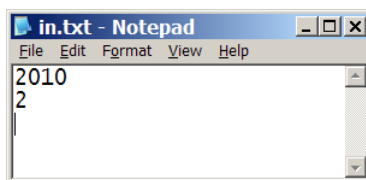


Figura 12-1 Conteúdo do ficheiro `in.txt`

Execução de exemplo:

```
C:\>diasdomes < in.txt  
Indique ano: Indique mes: 28
```

Repare-se que o programa não ficou à espera que se introduza o ano e o mês, estes valores foram lidos do ficheiro `in.txt`, sendo o resultado final 28. Para o processo inverso, redireccionar o texto de saída para um ficheiro, utiliza-se o **operador** **>**

```
C:\> prog > out.txt
```

Para o ficheiro `out.txt` será enviado a saída de dados do programa.

Execução de exemplo:

```
C:\>diasdomes > out.txt
2010
2
```

A aplicação já não mostra o texto "Indique ano:" nem "Indique mes:" dado que todo o texto que deveria de ir para o ecrã está a ser colocado no ficheiro out.txt. No final o ficheiro out.txt fica deste modo:

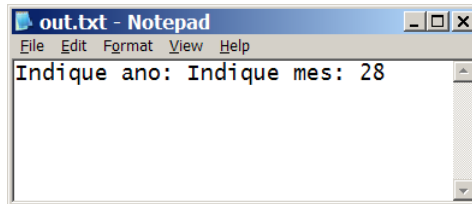


Figura 12-2 Conteúdo do ficheiro out.txt

Parte dos dados que são para especificar o que o utilizador deve introduzir, não fazem sentido neste tipo de utilizações, em que o utilizador sabe bem o que quer, ao ponto de colocar os dados de entrada num ficheiro, portanto não está dependente do texto que a aplicação lhe mostra para colocar os dados correctamente. Vamos mostrar um exemplo mais real, alterando o exemplo da geração dos números aleatórios para não mostrar texto de identificação da informação esperada ao utilizador, e mostrar todos os valores gerados independente do número de valores.

```
100 ...
101 int main()
102 {
103     TVectorInt vector;
104     int n,base;
105     srand(1);
106
107     scanf("%d",&n);
108     scanf("%d",&base);
109
110     vector=VectorAleatorio(n,base);
111     MostraVector(&vector);
112     VILibertar(&vector);
113 }
```

Execução de exemplo:

```
C:\>vectoraleatorio7
10
10
1 7 4 0 9 4 8 8 2 4
C:\>vectoraleatorio7 > out.txt
1000
100
```

Os 1000 valores aleatórios foram colocados no ficheiro out.txt. O ficheiro ficou com o seguinte conteúdo:

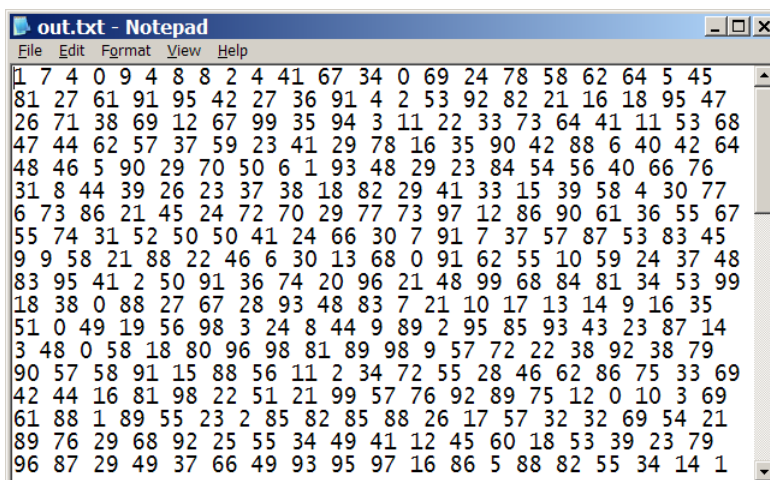


Figura 12-3 Novo conteúdo do ficheiro out.txt

Por vezes pretende-se juntar informação no ficheiro, e não criar um ficheiro novo. Para tal pode-se utilizar o **operador de linha de comando >>**, que em vez de apagar o ficheiro antes de colocar a informação do programa, adiciona a informação ao final do ficheiro:

Execução de exemplo:

```
C:\>vectoraleatorio7 >> out.txt
10
10
```

Pode-se confirmar que os últimos valores no ficheiro out.txt são menores que 10:

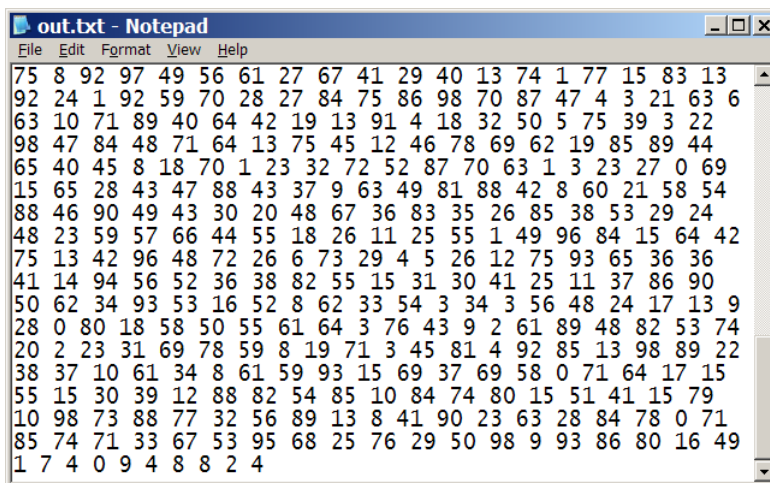


Figura 12-4 Ficheiro out.txt com algum texto adicionado pela última operação

Sobre redireccionamento dos dados de entrada e saída para ficheiros é tudo. Pode no entanto ser interessante enviar os dados de um programa, não para um ficheiro mas sim para outro programa. Desta forma tem-se dados a passar directamente de um programa para outro. Isso é possível através do **operador de linha de comando |**:

```
C:\> prog1 | prog2
```

Os dados de saída de `prog1`, são os dados de entrada de `prog2`. Vamos a um exemplo concreto. Suponhamos que pretendemos gerar valores aleatórios para testar o programa `diasdomes`, e pretendemos utilizar o programa de geração de vectores aleatórios. O seguinte comando satisfaz o pretendido. A entrada de dados inicial está em `in.txt`, e a saída de dados final vai para `out.txt`:

```
C:\>vectoraleatorio7 < in.txt / diasdomes > out.txt
```

O conteúdo dos ficheiros é o seguinte:

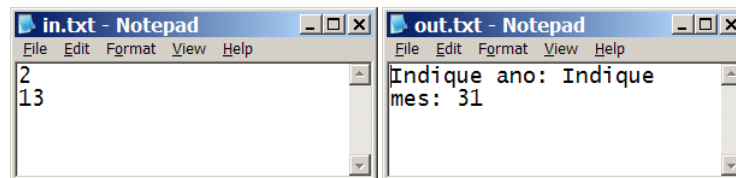


Figura 12-5 Conteúdo actual dos ficheiros `in.txt` e `out.txt`

Na entrada de dados foram pedidos dois valores aleatórios até 13, uma vez que o mês não pode ser superior a 12. Poderia ter sido gerado um valor inválido para o mês (o valor 0), mas tal não aconteceu, os valores gerados foram "2 7". Os valores gerados foram passados para o programa `diasdomes` que finalmente colocou o resultado no ficheiro `out.txt`.

Pode-se encadear quantos programas forem necessários, podendo inclusive estar desenvolvidos em linguagens de programação distintas. Neste caso `vectoraleatorio7` está a ser utilizado como um gerador de casos de teste para `diasdomes`. Para colocar `diasdomes` sobre muitas situações válidas, o ideal será utilizar um programa específico para gerar valores aleatórios na gama de valores que `diasdomes` suporta. Assim podem ser efectuados muitos testes, que de outra forma, testados manualmente, não só levaria muito tempo como provavelmente não seriam detectados tantos problemas. Pode-se fazer também um programa para verificar o correcto output de `diasdomes`, e dar um alerta no caso de detectar algum problema.

O redireccionamento da entrada e saída de dados, não resolve no entanto o problema de forma completa. É sem dúvida uma ótima ferramenta, mas apenas se pode ter um ficheiro de entrada e outro de saída. Pode haver necessidade de se querer utilizar o teclado, e mesmo assim obter o conteúdo de um ficheiro, ou utilizar o ecrã, e mesmo assim gravar num ficheiro, ou ainda abrir dois ou mais ficheiros.

O ideal seria poder abrir os ficheiros que fossem necessários, e aceder a eles como se estivesse a ler do teclado e a imprimir para o ecrã. É precisamente para isso que servem as funções de ficheiros, em modo de texto: **`fopen/fprintf/fscanf/fgets/feof/fclose`**

LER:

```

/*abertura em modo texto, para leitura, retorna NULL se não abrir */
FILE *f=fopen(ficheiro,"rt");
/* idêntico a scanf, mas antecedendo um ficheiro aberto para leitura */
fscanf(f,...);
/* idêntico a gets, mas com indicação do tamanho máximo da string12 */
fgets(str,MAXSTR,f);
/* testa se foi atingido o final do ficheiro f */
feof(f);
/* fecha o ficheiro após já não ser necessário */
fclose(f);

```

A função `fopen` retorna um apontador para o tipo `FILE`, com a referência a utilizar em todas as funções sobre o ficheiro, sendo fechado por `fclose`. Pode-se fazer um paralelo entre `fopen/fclose` e o `malloc/free`, enquanto estes últimos alocam/libertam blocos de memória, os primeiros abrem/fecham ficheiros.

Programa dos dias do mês reescrito para ler informação do ficheiro `in.txt`:

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int ano, mes, dias;
6     FILE *f;
7     f=fopen("in.txt","rt");
8     if(f==NULL)
9         return;
10    fscanf(f,"%d", &ano);
11    fscanf(f,"%d", &mes);
12    fclose(f);
13
14    if(mes==2)
15    {
16        /* teste de ano bissexto */
17        if(ano%400==0 || ano%4==0 && ano%100!=0)
18            printf("29");
19        else
20            printf("28");
21    } else if(mes==1 || mes==3 || mes==5 || mes==7 ||
22            mes==8 || mes==10 || mes==12)
23    {
24        printf("31");
25    } else
26    {
27        printf("30");
28    }
29 }

```

Programa 12-1 Número de dias dado mês/ano, lendo dados a partir do ficheiro “in.txt”

Execução de exemplo:

```

C:\> diasdomes2
28

```

¹² A função `fgets` retrona a string lida, a não ser que tenha existido um erro, e nesse caso retorna `NULL`

GRAVAR:

```
/*abertura em modo texto, para leitura, retorna NULL se não abrir*/
FILE *f=fopen(ficheiro,"wt");
/* idêntico a scanf, mas antecedendo um ficheiro aberto para leitura */
fprintf(f,...);
fclose(f); /* fecha o ficheiro após já não ser necessário*/
```

Pode-se gravar dados adicionando-os ao final do ficheiro, neste caso o modo de leitura seria "at".

Programa dos dias do mês reescrito para ler de in.txt e gravar para out.txt:

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int ano, mes, dias;
6     FILE *f,*f2;
7     f=fopen("in.txt","rt");
8     f2=fopen("out.txt","wt");
9     if(f==NULL || f2==NULL)
10        return;
11    fscanf(f,"%d", &ano);
12    fscanf(f,"%d", &mes);
13    fclose(f);
14
15    if(mes==2)
16    {
17        /* teste de ano bissexto */
18        if(ano%400==0 || ano%4==0 && ano%100!=0)
19            fprintf(f2,"29");
20        else
21            fprintf(f2,"28");
22    } else if(mes==1 || mes==3 || mes==5 || mes==7 ||
23             mes==8 || mes==10 || mes==12)
24    {
25        fprintf(f2,"31");
26    } else
27    {
28        fprintf(f2,"30");
29    }
30    fclose(f2);
31 }
```

Programa 12-2 Número de dias do mês dado mês/ano lendo dados de “in.txt” e escrevendo para “out.txt”

Execução de exemplo:

C:\>*díasdomes3*

Ficheiro out.txt:

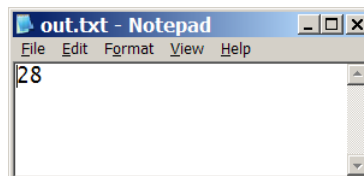


Figura 12-6 Conteúdo do ficheiro “out.txt” após execução do Programa 12-2

Tudo o resto é igual à utilização normal do printf/scanf/gets, tão igual que na verdade é mesmo igual. Existe um ficheiro que está sempre aberto para a leitura: **stdin**. Existe outro ficheiro que está

sempre aberto para escrita: **stdout**. Estes ficheiros não devem ser atribuídos a nada, nem fechados com **fclose**.

```
printf(...); é igual a fprintf(stdout,...);
scanf(...); é igual a fscanf(stdin,...);
gets(...); é igual a fgets(...,MAXSTR,stdin);
```

Entre o **gets** e **fgets** há mais um argumento, para proteger a dimensão do vector alocada.

Estaria tudo dito relativamente aos ficheiros, não fosse agora haver outro tipo de utilização: gravar informação para depois ser lida e processada automaticamente. Nesta situação, não faz sentido os textos de identificação do campo a introduzir, como passa a ser necessário gravar e ler o ficheiro num formato específico, para que cada dado seja gravado/lido correctamente.

Vamos reutilizar o exemplo da ficha de uma pessoa para gravar numa lista todas as pessoas introduzidas, de forma a ficarem disponíveis na vez seguinte que se abrir a aplicação. O formato é simples: uma linha por cada campo pedido, da mesma forma que são introduzidos os dados.

```
1 #include <stdio.h>
2
3 #define MAXSTR 255
4
5 /* Tipo de dados abstracto: TListaPessoa */
6
7 typedef struct
8 {
9     int ano, mes, dia;
10 } TData;
11
12 typedef struct SPessoa
13 {
14     char *nome;
15     char sexo;
16     TData nascimento;
17     char *morada;
18     char *email;
19     long telefone;
20     struct SPessoa *seguinte;
21 } TListaPessoa;
22
23 TListaPessoa* LAdicionar(TListaPessoa *lista, TListaPessoa pessoa)
24 {
25     TListaPessoa *novo;
26     novo=(TListaPessoa*)malloc(sizeof(TListaPessoa));
27     if(novo!=NULL)
28     {
29         (*novo)=pessoa;
30         novo->seguinte=lista;
31         return novo;
32     }
33     return lista;
34 }
35
36 void LPLibertar(TListaPessoa *lista)
37 {
38     TListaPessoa *aux;
39     while(lista!=NULL)
40     {
41         aux=lista->seguinte;
42         free(lista->nome);
43         free(lista->morada);
44         free(lista->email);
45         free(lista);
46         lista=aux;
47     }
```



```

48 }
49
50 /* Programa */
51
52 /* colocar fout=NULL para não imprimir texto dos campos a ler */
53 void LerPessoa(TListaPessoa *pessoa, FILE *fin, FILE *fout)
54 {
55     char str[MAXSTR];
56     if(fout!=NULL)
57         fprintf(fout, "Nome: ");
58     fgets(str, MAXSTR, fin);
59     pessoa->nome=(char*)malloc(strlen(str)+1);
60     strcpy(pessoa->nome, str);
61     if(fout!=NULL)
62         fprintf(fout, "Sexo: ");
63     fscanf(fin, "%c", &pessoa->sexo);
64     if(fout!=NULL)
65         fprintf(fout, "Data de nascimento (dd-mm-aaaa): ");
66     fscanf(fin, "%d-%d-%d",
67           &pessoa->nascimento.dia,
68           &pessoa->nascimento.mes,
69           &pessoa->nascimento.ano);
70     if(fout!=NULL)
71         fprintf(fout, "Morada: ");
72     fgets(str, MAXSTR, fin);
73     fgets(str, MAXSTR, fin);
74     pessoa->morada=(char*)malloc(strlen(str)+1);
75     strcpy(pessoa->morada, str);
76     if(fout!=NULL)
77         fprintf(fout, "Email: ");
78     fgets(str, MAXSTR, fin);
79     pessoa->email=(char*)malloc(strlen(str)+1);
80     strcpy(pessoa->email, str);
81     if(fout!=NULL)
82         fprintf(fout, "Telefone: ");
83     fscanf(fin, "%ld", &pessoa->telefone);
84 }
85
86 /* grava para fout, mas apenas mostra texto com o nome dos
87    campos se prompt=1 */
88 void MostraPessoa(TListaPessoa *pessoa, FILE *fout, int prompt)
89 {
90     if(prompt==0)
91     {
92         fprintf(fout, "%s%c\n%d-%d-%d\n%s%s%d",
93               pessoa->nome,
94               pessoa->sexo,
95               pessoa->nascimento.dia,
96               pessoa->nascimento.mes,
97               pessoa->nascimento.ano,
98               pessoa->morada,
99               pessoa->email,
100              pessoa->telefone);
101     } else {
102         fprintf(fout, "\nNome: %s\nSexo: %c\nData de Nascimento: %d-%d-%d\n",
103               pessoa->nome,
104               pessoa->sexo,
105               pessoa->nascimento.dia,
106               pessoa->nascimento.mes,
107               pessoa->nascimento.ano);
108         fprintf(fout, "Morada: %s\nEmail: %s\nTelefone: %d\n",
109               pessoa->morada,
110               pessoa->email,
111               pessoa->telefone);
112     }
113 }
114
115 int main()
116 {
117     FILE *f;
118     TListaPessoa *lista=NULL, *i, pessoa;

```

```

119  /* ler as pessoas todas do ficheiro pessoas.txt */
120  f=fopen("pessoas.txt","rt");
121  if(f!=NULL)
122  {
123      while(!feof(f))
124      {
125          /* lê uma pessoa do ficheiro */
126          LerPessoa(& Pessoa, f, NULL);
127          lista=LPAñadir(lista,Pessoa);
128      }
129      fclose(f);
130  }
131  /* mostrar as pessoas lidas do ficheiro */
132  for(i=lista; i!=NULL; i=i->seguinte)
133      MostraPessoa(i,stdout,1);
134
135  /* lê uma pessoa do teclado */
136  LerPessoa(& Pessoa, stdin, stdout);
137  lista=LPAñadir(lista,Pessoa);
138
139  /* grava tudo para o ficheiro */
140  f=fopen("pessoas.txt","wt");
141  if(f!=NULL)
142  {
143      for(i=lista; i!=NULL; i=i->seguinte)
144          MostraPessoa(i,f,0);
145      fclose(f);
146  }
147  LPLibertar(lista);
148 }

```

Programa 12-3 Ficha de pessoa, com gravação e leitura do ficheiro

Realçar na reutilização dos métodos para ler uma ficha e gravar, aos quais apenas se permitiu fazer essa operação através de um ficheiro. Na função main é que se chama esses métodos, quer com o stdin/stdout, como com um ficheiro aberto para escrita e leitura. No modo de gravação para ficheiro, é muito importante que os campos sejam gravados exactamente da mesma forma que são lidos, caso contrário os diferentes valores podem não ir para os locais correctos.

Execução de exemplo:

```

C:\> pessoa3
Nome: Heraclito de Efeso
Sexo: M
Data de nascimento (dd-mm-aaaa): 1-1--540
Morada: Efeso
Email: heraclito@filosofia.com
Telefone: 111111111
C:\> pessoa3
Nome: Heraclito de Efeso
Sexo: M
Data de Nascimento: 1-1--540
Morada: Efeso
Email: heraclito@filosofia.com
Telefone: 111111111
Nome: Afonso Henriques
Sexo: M
Data de nascimento (dd-mm-aaaa): 25-7-1109
Morada: Guimaraes
Email: afonso@conquistador.com
Telefone: 111111111
C:\> pessoa3
Nome: Heraclito de Efeso
Sexo: M
Data de Nascimento: 1-1--540
Morada: Efeso

```

Email: heraclito@filosofia.com
 Telefone: 111111111
 Nome: Afonso Henriques
 Sexo: M
 Data de Nascimento: 25-7-1109
 Morada: Guimaraes
 Email: afonso@conquistador.com
 Telefone: 111111111
 Nome: **Ludwig van Beethoven**
 Sexo: **M**
 Data de nascimento (dd-mm-aaaa): **16-12-1770**
 Morada: **Viena**
 Email: **ludwig@musica.com**
 Telefone: **111111111**

No final o ficheiro de texto pessoas.txt ficou com o seguinte conteúdo:

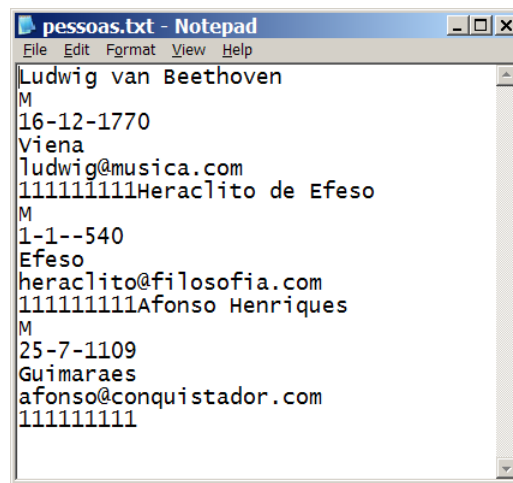


Figura 12-7 Ficheiro “pessoas.txt” com o conteúdo gravado/lido pelo Programa 12-3

Repare-se que o nome de uma pessoa está colado ao último campo da pessoa anterior. Isto ocorreu devido à formatação dos dados de entrada/saída. O ficheiro pode ser editado, reflectindo-se essas alterações no programa, mas se o formato não for o correcto, os dados deixam de se ler correctamente.

Os ficheiros gravados por estarem em **modo de texto**¹³ podem ser editados por outra aplicação, e alterados facilmente. Utilizam no entanto muito espaço, enquanto para guardar um número inteiro são necessários 4 bytes em memória, para o escrever em modo de texto sem contar com os caracteres que antecedem o número, podem ser necessários até 9 bytes.

O modo de texto assume que apenas caracteres imprimíveis são utilizados, e aplica algumas operações (no Windows substitui um \n por um \r\n), existindo o conceito de linha de texto do ficheiro. O **modo binário**¹⁴ é para programas ou ficheiros em outro formato não organizado por linhas e texto, em que a existência de um carácter com o código de \n é pura coincidência, já que não há linhas.

¹³ Os modos de texto mais utilizados: “rt” – leitura; “wt” – escrita num ficheiro novo; “at” – escrita no final do ficheiro

¹⁴ Os modos binários mais utilizados: “rb” – leitura; “wb” – escrita num ficheiro novo; “ab” – escrita no final do ficheiro; “r+b” – leitura e escrita em ficheiro existente; “w+b” – leitura e escrita num ficheiro novo

Como não existe linhas, o ficheiro é gravado/lido com blocos de memória, sendo para isso que as funções **fread/fwrite**¹⁵ existem (ler/gravar um bloco de memória). As funções **fseek/ftell** permitem recolocar a posição do ficheiro e ler a posição actual no ficheiro, respectivamente. Com a utilização destas funções, em vez do **fprintf/fscanf/fgets**, pode-se facilmente gravar e ler dados em modo binário.

Para exemplificar, vamos acrescentar ao tipo abstracto de dados vector, a possibilidade de se gravar/ler para um ficheiro em modo binário. No exemplo do ficheiro aleatório, assim que o utilizador coloque um pedido, verifica se o ficheiro existe e nesse caso lê os valores do ficheiro, caso contrário gera os valores e grava-os no ficheiro, para que da próxima vez não seja necessário gerar os valores.

```

80 ...
81 void VILer(TVectorInt *vector, FILE *f)
82 {
83     int n;
84     vector->valor=NULL;
85     vector->alocado=0;
86     vector->n=0;
87     /* ler a dimensão do vector */
88     fread(&n,sizeof(int),1,f);
89     /* realocar para a dimensão exacta */
90     VIInternoRealocar(vector,n/2);
91     /* ler o vector inteiro */
92     if(vector->valor!=NULL)
93     {
94         fread(vector->valor,sizeof(int),n,f);
95         vector->n=n;
96     }
97 }
98
99 void VIGravar(TVectorInt *vector, FILE *f)
100 {
101     fwrite(&(vector->n),sizeof(int),1,f);
102     fwrite(vector->valor,sizeof(int),vector->n,f);
103 }
104
105 /* Programa */
106 ...
107 int main()
108 {
109     TVectorInt vector;
110     FILE *f;
111     char str[MAXSTR];
112     int n,base;
113     srand(1);
114
115     scanf("%d",&n);
116     scanf("%d",&base);
117
118     /* verificar se existe um ficheiro com os dados gravados */
119     sprintf(str,"n%db%d.dat",n,base);
120     f=fopen(str,"rb");
121     if(f!=NULL)
122     {
123         VILer(&vector,f);
124         fclose(f);
125     } else {
126         vector=VectorAleatorio(n,base);
127         f=fopen(str,"wb");
128         if(f!=NULL)

```

¹⁵ Utilização: **fread/fwrite(tipo*, sizeof(tipo), número de blocos, f)**; ambas as funções recebem um apontador a memória a ler/gravar no ficheiro, bem como o tamanho de cada bloco e número de blocos.

```

149     {
150         VIGravar(&vector,f);
151         fclose(f);
152     }
153 }
154 MostraVector(&vector);
155 VILibertar(&vector);
156 }

```

Programa 12-4 Actualização do tipo abstracto TVectorInt para ler/gravar para ficheiro, em modo binário

Execução de exemplo:

```

C:\>vectoraleatorio8
10
10
1 7 4 0 9 4 8 8 2 4
C:\>vectoraleatorio8
10
100
41 67 34 0 69 24 78 58 62 64
C:\>vectoraleatorio8
10
100
41 67 34 0 69 24 78 58 62 64

```

A última execução foi lida do ficheiro, e não gerada. Neste caso a operação de gerar um valor aleatório tem um custo computacional muito baixo, não vale a pena gravar para ficheiro. Se fosse uma operação mais demorada, o programa tiraria vantagem de ter já no passado feito essa computação, e agora seria respondido ao pedido de forma imediata. Se por algum motivo os ficheiros fossem apagados, o programa voltaria a fazer os cálculos. Este é um tipo de estratégia utilizado inclusive em algoritmos e com memória em vez de ficheiros, guardando resultado de alguns cálculos pesados em memória, para no caso de serem necessários no futuro poupar-se o tempo de os recalcular.

Notar que as funções VILer e VIGravar são muito curtas, uma vez que as funções fread e fwrite estão adaptadas à leitura de vectores de estruturas, sendo necessário apenas alocar a memória necessária, e chamar as funções. Os dois ficheiros que foram criados automaticamente na execução de exemplo, n10b10.dat e n10b100.dat ocupam 44 bytes cada, que é precisamente o espaço necessário para guardar em memória os 10 valores inteiros de 4 bytes, e um outro com o tamanho do vector.

Se a informação for lida sequencialmente, as funções fseek¹⁶ e ftell¹⁷ não são necessárias, caso contrário pode-se utilizar estas funções para reposicionar o ponto de leitura ou escrita do ficheiro. Este tipo de utilização foi no entanto mais relevante no passado, em que havia pouca memória e os ficheiros eram utilizados como se fossem blocos de memória, ficando o ficheiro sempre aberto, e o algoritmo em vez de aceder a um valor em memória teria de aceder a um registo no ficheiro. É um processo muito mais lento que o acesso à memória, mas necessário em tempos em que a memória principal era escassa, e actualmente pode ainda ser utilizado em casos muito particulares.

¹⁶ Utilização para posicionamento no ficheiro: fseek(f, posicao, SEEK_SET); outras constantes podem ser utilizadas, SEEK_CUR para a posição ser relativa à posição corrente, e SEEK_END para a posição ser relativa ao fim do ficheiro

¹⁷ Leitura da posição actual no ficheiro: posicao=ftell(f);

No modo texto pode-se editar com editores externos, no modo binário tal é mais difícil, mas ocupa menos espaço e é mais rápido de ler. Estas duas possibilidades são a base para a tomada de decisão, que tem vindo cada vez mais a recair para o modo de texto em XML, formato este que não deixa de ser texto, mas segue um conjunto de regras mínimas da organização da informação em tags. A razão desta opção é simples: os dados muitas vezes duram mais que as aplicações. Não deve ser optado pelo modo binário como forma de segurança, dado que facilmente se faz um editor que leia o dados, apenas encriptando é possível adicionar segurança.

Erros comuns mais directamente associados a este capítulo:

- **Abrir um ficheiro e não chegar a fechar**
 - **Forma:** Por vezes abre-se o ficheiro, mas a operação de fechar não causa erros e é desnecessária dado que o ficheiro será fechado quando o programa acabar
 - **Problema:** O número de ficheiros abertos é um recurso limitado pelo sistema operativo. Se um ficheiro não é mais preciso, deve ser imediatamente fechado de forma a libertar recursos. Se esta prática não for feita, pode acontecer que pedidos de abertura de ficheiros por esta ou outra aplicação sejam recusados
 - **Resolução:** Verificar o último local onde o ficheiro é necessário, e chamar a correspondente função `fclose` de forma a fechar o ficheiro
- **Abrir um ficheiro e não testar se foi bem aberto**
 - **Forma:** Quando se sabe que o ficheiro existe, não é necessário testar se a operação de abertura do ficheiro foi bem sucedida, é perda de tempo
 - **Problema:** Pode acontecer mesmo quando há a certeza que o ficheiro existe, que a operação de abertura de ficheiro seja mal sucedida, no caso de estarem demasiados ficheiros abertos, ou por questões de permissões, ou ainda por estarem duas aplicações a tentar abrir o ficheiro em modo de escrita. As operações seguintes, que utilizam o apontador retornado para o ficheiro, ficam com um procedimento incerto
 - **Resolução:** Testar sempre se o ficheiro foi bem aberto, e apenas utilizar o apontador para o ficheiro no caso de este ser diferente de `NULL`

Exemplo destes dois erros pode ser a simplificação do Programa 12-1 apagando as linhas 8 e 9, bem como a linha 12.

13. TRUQUES

Vamos acabar, não com o mais importante, mas sim com o que há de menos importante na linguagem C, que são os operadores binários (que manipulam bits), bem perto do assembly. Falaremos também do operador `?` e da estrutura `union`.

Antes dos operadores binários, introduz-se o **operador `?`**, que permite fazer um condicional dentro de uma expressão:

`(cond?exp1:exp2)`

Este operador é útil em situações que tanto `cond`, como `exp1` e `exp2` são pequenas, e levaria à criação de um condicional `if` quando uma pequena expressão servia. O exemplo clássico de boa utilização é a seguinte:

```
x=(a>b?a:b); /* em x fica o máximo de a e b */

if(a>b)
    x=a;
else
    x=b;
```

A versão com o `if` é claramente mais complexa. O operador `?` pode no entanto ser mal utilizado se `cond`, `exp1` ou `exp2` forem muito longos, ficando o código de difícil leitura. O exemplo do cálculo dos dias do mês pode ser reduzido ao cálculo de uma só expressão:

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int ano, mes, dias;
6     printf("Indique ano: ");
7     scanf("%d", &ano);
8     printf("Indique mes: ");
9     scanf("%d", &mes);
10
11     dias=
12         (mes==2 ?
13             (ano%400==0 || ano%4==0 && ano%100!=0 ? 29 : 28) :
14             (mes==1 || mes==3 || mes==5 || mes==7 ||
15             mes==8 || mes==10 || mes==12 ? 31 : 30));
16
17     printf("%d",dias);
18 }
```

Programa 13-1 Versão condensada do número de dias de um dado mês/ano

A funcionalidade deste programa é igual aos restantes. No entanto as expressões lógicas utilizadas são muito grandes, pelo que esta versão perde legibilidade. Mesmo na expressão lógica, este operador pode ser utilizado. Ainda no exemplo dos dias do mês, pode-se substituir as diversas igualdades para teste de mês de 31 dias por um operador `?`

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int ano, mes, dias;
6      printf("Indique ano: ");
7      scanf("%d", &ano);
8      printf("Indique mes: ");
9      scanf("%d", &mes);
10
11     dias=
12         (mes==2 ?
13             (ano%400==0 || ano%4==0 && ano%100!=0 ? 29 : 28) :
14             ((mes%2==1?mes<=7:mes>=8) ? 31 : 30));
15
16     printf("%d", dias);
17 }

```

Programa 13-2 Versão ainda mais condensada do número de dias de um dado mês/ano

Neste caso tem-se compactação da expressão, mas a legibilidade é muito reduzida, e a utilizar tem-se de acompanhar o código de um bom comentário explicativo.

Este operador apenas está nesta secção, dado que quem está a aprender é conveniente não ver este tipo de controlo de fluxo, porque na verdade o controle de fluxo é dentro de uma expressão, que para o alto nível da linguagem C, representa uma só instrução. Não há controlo de fluxo sem várias instruções. Na verdade não há contradição, dado que a implementação de uma expressão de C em assembly resulta em múltiplas instruções.

Uma particularidade da linguagem C, utilizado por vezes em conjunção com o operador `?`, é a possibilidade de **utilizar atribuições como expressões**. Isto é, como uma atribuição retorna o valor atribuído, pode ser utilizado dentro de uma expressão. A situação mais natural para fazer uso deste conhecimento é a atribuição do mesmo valor a várias variáveis, por exemplo:

```
x=y=z=0;
```

Neste exemplo a expressão resultante de `z=0` foi utilizada para a atribuição a `y`, que por sua vez foi utilizada como expressão para a atribuição a `x`. No entanto pode ser utilizado em situações mais imaginativas, sem benefício para a legibilidade do código:

```
a=(b>(c=0)?b=-b:++c);
```

A expressão anterior é abstracta, dado que para má utilização do conhecimento não existem bons exemplos. A única variável que tem de ter um valor antes da instrução acima é a variável `b`. Por exemplo, se esta for 10, os valores das variáveis ficam `a=-10`; `b=-10`; `c=0`; . Caso seja -10, os valores ficam `a=1`; `b=-10`; `c=1`; . A sequência de operações é a seguinte:

1. Avaliado "`b>(c=0)`"
 - a. Atribuído "`c=0`"
 - b. Retornado "`c`"
2. Se verdadeiro, avaliado "`b=-b`"
 - a. Atribuído "`b=-b`"
 - b. Retornado "`b`"
3. Se falso, avaliado "`++c`"
 - a. Incrementado "`c`"
 - b. Retornado "`c`"
4. Atribuído a "`a`" o valor retornado em 2 ou 3

A mais-valia dos **operadores binários** (que manipulam os bits) é disponibilizar em linguagem de alto nível, operações de custo muito baixo em assembly e que existem em todos os computadores modernos, que podem ser úteis, para além dos habituais truques envolvendo operações com potências de 2, para a implementação eficiente de algumas operações normalmente envolvendo tipos de dados abstractos. Todas as operações envolvendo tipos de dados, são tipicamente muito utilizadas, pelo que devem ser optimizadas o melhor possível.

O **operador >>** (e correspondente >>=, bem como as versões inversas << e <<=) são operadores de deslocação de bits dentro da variável. Os bits encostados perdem-se.

Deslocação:

```
a = a >> b; ( a>>=b; )
a = a << b; ( a<<=b; )
```

Tal como num número em base decimal, esta operação significa deslocar os dígitos de um número para a esquerda ou direita. Na base decimal, o resultado é multiplicar ou dividir por 10, por cada deslocação. Em base binária, o resultado é multiplicar ou dividir por 2. Como esta operação é mais rápida que qualquer operação aritmética, é comum multiplicar/dividir por potências de 2 utilizando deslocamento de bits.

```
1 #include <stdio.h>
2 #include <time.h>
3
4 #define CICLOS 100000000
5
6 int main()
7 {
8     int valor=12345,i;
9     clock_t instante;
10
11     instante=clock();
12     for(i=0;i<CICLOS;i++);
13     printf("\nTempo de %d ciclos vazios: %.3f",
14           CICLOS, (float)(clock()-instante)/CLOCKS_PER_SEC);
15
16     instante=clock();
17     for(i=0;i<CICLOS;i++)
18     {
19         valor*=2;
20         valor/=2;
21     }
22     printf("\nTempo de %d multiplicacoes e divisoes por 2: %.3f",
23           CICLOS, (float)(clock()-instante)/CLOCKS_PER_SEC);
24     instante=clock();
25     for(i=0;i<CICLOS;i++)
26     {
27         valor<<=1;
28         valor>>=1;
29     }
30     printf("\nTempo de %d operacoes de deslocao: %.3f",
31           CICLOS, (float)(clock()-instante)/CLOCKS_PER_SEC);
32 }
```

Programa 13-3 Instrução de shift vs multiplicação/divisão por 2

Execução de exemplo:

```
C:\>deslocacao
Tempo de 100000000 ciclos vazios: 0.454
Tempo de 100000000 multiplicacoes e divisoes por 2: 2.093
Tempo de 100000000 operacoes de deslocao: 0.688
```

Como se pode observar, a vantagem é tentadora. O ciclo vazio é feito inicialmente, uma vez que estas operações têm um custo muito baixo, e o custo de operação do ciclo é relevante. Pode-se estimar o tempo gasto realmente nas operações como sendo 0,234 e 1,639 segundos, pelo que a operação de deslocação de um bit é cerca de 7 vezes mais rápida que a operação de multiplicação/divisão por 2.

As **operações binárias lógicas** implementam cada função lógica mas bit a bit. Estas operações têm tal como as operações de deslocamento, um custo muito baixo de execução.

Operações lógicas binárias:

```
a=a|b; (a|=b;) » or
a=a&b; (a&=b;) » and
a=a^b; (a^=b;) » xor
~a » complemento (negação binária)
```

Um número em base decimal, o dígito menos significativo é também o resto da divisão do número por 10, e da mesma forma, os dígitos menos significativos em binário, são o resto da divisão por a correspondente potência de 2. Com os operadores binários, o resto da divisão por 2 obtém-se simplesmente: **a&1**. Notar que apenas o dígito mais baixo poderá ficar não nulo. O resto da divisão com 4 seria: **a&3**. Esta operação é naturalmente mais rápida que a operação normal **a%2**, e **a%4**.

Vamos mostrar agora os **bits num número real**, de forma a exemplificar alguma manipulação de bits. Vamos ler um real para um inteiro, e mostrar no ecrã todos os seus bits.

```
1 #include <stdio.h>
2 #include <time.h>
3
4 int main()
5 {
6     unsigned int valor, bitAlto;
7     int i;
8
9     printf("valor real:");
10    scanf("%g",&valor);
11
12    bitAlto=1<<31;
13
14    for(i=0;i<32;i++)
15    {
16        printf("%c", (valor&bitAlto?'1':'0'));
17        valor<<=1;
18    }
19 }
```

Programa 13-4 Representação binária de um número real

Execução de exemplo:

```
C:\>bits
valor real:1
00111111100000000000000000000000
C:\>bits
valor real:-1
10111111100000000000000000000000
C:\>bits
valor real:2
01000000000000000000000000000000
C:\>bits
valor real:1.5
00111111110000000000000000000000
```

```

C:\>bits
Valor real:0.1
00111101110011001100110011001101
C:>bits
Valor real:0.125
00111110000000000000000000000000
C:\>bits
Valor real:1.5
00111111110000000000000000000000

```

No exemplo colocou-se numa variável o bit mais alto, de forma a detectar o seu valor, e foi-se fazendo operações de deslocação até estarem todos os bits processados. Pode-se assim ver como é composto o número do tipo real, com o primeiro bit sendo o **signal**, seguido de 8 a comporem o **expoente**, e os restantes a **mantissa**, seguindo a fórmula: $(-1)^{SINAL} \times 1, MANTISSA_2 \times 2^{EXPOENTE-255}$. No entanto o tipo real tem mais particularidades, nomeadamente situações especiais de infinito, e número não válido, pelo que não se aconselha fazer a manipulação binária de reais, este exemplo apenas pretende ilustrar optimizações possíveis.

Tivemos que fazer um artifício para ver o real como inteiro, foi indicar ao scanf um apontador para inteiro sem sinal, quando o scanf esperava um real. A linguagem C tem uma estrutura **union** que permite uma mesma variável comportar-se como sendo de diferentes tipos, evitando assim o artifício:

```

1 #include <stdio.h>
2 #include <time.h>
3
4 int main()
5 {
6     unsigned int bitAlto;
7     int i;
8     union {
9         unsigned int inteiro;
10        float real;
11    } valor;
12
13    printf("valor real:");
14    scanf("%g",&valor.real);
15
16    bitAlto=1<<31;
17
18    for(i=0;i<32;i++)
19    {
20        printf("%c",(valor.inteiro&bitAlto?'1':'0'));
21        valor.inteiro<<=1;
22    }
23
24 }

```

Programa 13-5 Versão do Programa 13-4 utilizando a estrutura “union”

A estrutura union é idêntica a struct, mas a sua utilidade é muito inferior. Apenas em raras excepções poderá esta estrutura ser útil.

O conhecimento da representação binária dos números reais, poderia ser útil para pequenas operações, como saber se o número é negativo bastará comparar o bit mais alto, ou para multiplicar/dividir por 2, bastará somar/subtrair a zona do expoente, o que corresponde a fazer uma soma/subtracção inteira por o número 1 devidamente deslocado. Vamos medir o ganho potencial desta última operação:

```

1 #include <stdio.h>
2 #include <time.h>
3
4 #define CICLOS 100000000
5
6 int main()
7 {
8     union {
9         unsigned int inteiro;
10        float real;
11    } valor;
12    int i, incremento;
13    clock_t instante;
14
15    valor.real=1.0;
16    incremento=1<<23;
17
18    printf("\nReal: %f", valor.real);
19    valor.inteiro+=incremento;
20    printf("\nReal *2: %f", valor.real);
21    valor.inteiro-=incremento;
22    printf("\nReal /2: %f", valor.real);
23
24    instante=clock();
25    for(i=0; i<CICLOS; i++);
26    printf("\nTempo de %d ciclos vazios: %.3f",
27        CICLOS, (float)(clock()-instante)/CLOCKS_PER_SEC);
28
29    instante=clock();
30    for(i=0; i<CICLOS; i++)
31    {
32        valor.real*=2;
33        valor.real/=2;
34    }
35    printf("\nTempo de %d operacoes *=2 e /=2 float: %.3f",
36        CICLOS, (float)(clock()-instante)/CLOCKS_PER_SEC);
37    instante=clock();
38    for(i=0; i<CICLOS; i++)
39    {
40        valor.inteiro+=incremento;
41        valor.inteiro-=incremento;
42    }
43    printf("\nTempo de %d operacoes += e -= int: %.3f",
44        CICLOS, (float)(clock()-instante)/CLOCKS_PER_SEC);
45 }

```

Programa 13-6 Utilização do formato do número real para aumentar a eficiência de algumas operações

Execução de exemplo:

```

C:\>mult
Real: 1.000000
Real *2: 2.000000
Real /2: 1.000000
Tempo de 100000000 ciclos vazios: 0.453
Tempo de 100000000 operacoes *=2 e /=2 float: 1.266
Tempo de 100000000 operacoes += e -= int: 0.718

```

Há aqui um ganho potencial de 3 vezes os produtos efectuados na versão inteira.

Este tipo de utilização poderá no entanto dar mais trabalho que recompensa, na medida que as condições optimizáveis que fogem ao procedimento normal não são muito frequentes, e podem ser executadas um número diminuto de número de vezes no algoritmo não compensando nem o trabalho nem a perda de legibilidade que essa utilização implica.

Uma boa utilização dos operadores binários é para colocar numa só variável inteira, várias variáveis booleanas. Exemplifica-se com características de produtos:

```
1 #include <stdio.h>
2
3 #define NACIONAL 1
4 #define BARATO 2
5 #define QUALIDADE 4
6 #define LEVE 8
7
8 void MostrarProduto(int produto)
9 {
10     if(produto&NACIONAL)
11         printf("Nacional; ");
12     if(produto&BARATO)
13         printf("Barato; ");
14     if(produto&QUALIDADE)
15         printf("Qualidade; ");
16     if(produto&LEVE)
17         printf("Leve; ");
18 }
19
20 main()
21 {
22     int produto[4],i;
23
24     /* 0: produto barato e de qualidade */
25     produto[0]=BARATO|QUALIDADE;
26     /* 1: produto nacional e leve */
27     produto[1]=NACIONAL|LEVE;
28     /* 2: igual ao produto 0, mas não barato */
29     produto[2]=produto[0]&(~BARATO);
30     /* 3: todas as qualidades do 1 e 2,
31        excepto o ser nacional que é inverso */
32     produto[3]=(produto[1]|produto[2])^NACIONAL;
33
34     /* mostrar as características dos produtos */
35     for(i=0;i<4;i++)
36     {
37         printf("\nProduto %d: ");
38         MostrarProduto(produto[i]);
39     }
40 }
```

Programa 13-7 Exemplo de utilização de várias variáveis booleanas numa só variável inteira

Execução de exemplo:

C:\> **flags**

Produto 0: Barato; Qualidade;
Produto 1: Nacional; Leve;
Produto 2: Qualidade;
Produto 3: Qualidade; Leve;

No exemplo colocou-se macros para cada variável booleana, e em cada uma está uma potência de 2 distinta. Ao atribuir um valor basta fazer uma disjunção com a macro, como é feito no exemplo nos produtos 0 e 1. Para remover um valor, basta fazer uma conjunção com o complementar da variável, como é feito no produto 2. No produto 3 troca-se um valor utilizando o XOR binário. Na função `MostrarProduto` pode-se ver como se lê facilmente os valores, fazendo a conjunção com a macro respectiva.


Erros comuns mais directamente associados a este capítulo:

- **Comentários inexistentes em partes complexas**
 - **Forma:** O código deve ser lido por quem perceba não só da linguagem de programação C, como de algoritmos, e que seja suficientemente inteligente para o compreender
 - **Problema:** Quem lê código C deve naturalmente possuir todas essas características, mas tal não é motivo para não explicar as partes complexas. Os comentários podem afectar a legibilidade do código, e quem está a ler código em que tem dificuldade em compreender, não irá apreciar a forma rebuscada e hábil de quem o escreveu, mas sim a simplicidade e clareza com que o fez, ou o explica. Se não conseguir compreender algo, será certamente por culpa de quem escreveu que não sabia o que fazia e lá acabou por obter uma solução funcional em modo de tentativa e erro, de baixa qualidade
 - **Resolução:** Nas zonas menos claras, comentar explicando todos os truques / habilidades aplicadas. Se ficar algum por explicar, será confundido certamente com código que nem quem o fez sabe realmente porque funciona, e mais vale refazer novamente se houver algum problema

Exemplo de ocorrências deste erro encontrará certamente ao rever o seu código antigo. É um bom exercício rever o seu próprio código, para poder com alguma distância da altura em que o escreveu, reflectir sobre o que necessita realmente de ter em comentário no código para o ajudar a compreendê-lo quando já não se lembrar dele.

3) Exercícios

adiciona.c

 Faça uma estrutura para uma lista de inteiros de ligação simples (apontador para o elemento seguinte da lista). Implemente as funções para adicionar um elemento à lista, remover o elemento no topo da lista, e para calcular o número de elementos na lista.

Notas:


- Não se esqueça de remover todos os elementos na lista, antes de sair do programa. Tudo o que alocar deve libertar.

Execução de exemplo:

```
C:\>adiciona
Numero de elementos: 1000
Elementos: xxx 116 80 686 693 309 598 650 629 676
```

Pergunta: adicione 1000 números aleatórios entre 0 e 999 a uma lista, e calcule o número de elementos na lista, bem como o valor do primeiro elemento na lista. Indique na resposta o primeiro elemento na lista.

more.c

 Faça um programa que receba um ficheiro de texto em argumento, e o mostre no ecrã, utilizando os primeiros 4 caracteres para numerar a linha. Se uma linha tiver mais de 75 caracteres, a mesma é mostrada em várias linhas, mas o número de linha apresentado é sempre a do ficheiro. Considere que o comprimento máximo de uma linha é de 1024 caracteres.

Notas:

- Abra um ficheiro em modo de texto, e não se preocupe agora com a alocação de memória
- Verifique se o ficheiro foi bem aberto, e se foi dado um argumento com o nome do ficheiro

Execução de exemplo:

```
C:\>more more.c
1:#include <stdio.h>
2:
3:#define MAXSTR 1024
4:
5:int main(int argc, char **argv)
6:{
7:    FILE *f;
8:    char str[MAXSTR], *pt, c;
9:    int count = 0;
10:
```

```
11:    /* verificar se foi dado um argumento */
12:    if(argc <= 1)
13:    {
14:        printf("Utilização: more <ficheiro>\n");
15:        return;
16:    }
...
```

Pergunta: copie o texto desde "more.c" até às notas acabando em "nome do ficheiro", para um ficheiro de texto (abra no Notepad), e grave. Assegure-se que o ficheiro de texto no Notepad tem apenas 5 linhas (não utilizar o Word wrap), cada linha não começa com espaços (se tiver apague-os), não existindo *bullets*. Abra de seguida o ficheiro com a aplicação *more*, e indique na resposta separado por espaços: número de linhas (do ficheiro); número de linhas (do ficheiro) cortadas; palavras cortadas na mudança de linha por ordem (e separadas por um espaço, não incluindo vírgulas ou pontos se existirem).

insere.c



Sobre o código desenvolvido no exercício *adiciona.c*, faça uma função para inserir um elemento numa lista de inteiros, após um elemento menor e antes de um elemento maior ou igual (insere de forma ordenada). Faça ainda uma outra função para apagar os elementos de uma lista de inteiros, que sejam iguais a um determinado valor.

Notas:


- Pode chamar as operações atómicas desenvolvidas de adição, remoção e número de elementos, para implementar a inserção ordenada e remoção dos elementos com um determinado valor

Execução de exemplo:

```
C:\>insere
Elementos apos inserir 1000 elementos: 1 2 3 3 3 6 7 7 8 8
Numero de elementos na lista apos apagar: xxx
Elementos apos apagar: 1 3 3 3 7 7 9 11 11 15
```

Pergunta: insira ordenadamente na lista 1000 elementos aleatórios de valores entre 0 e 999, chamando *srand(1)*, e apague todos os números na lista que sejam pares. Indique como resposta o número de elementos da lista.

insertsort.c

 Utilizando as funções desenvolvidas dos exercícios anteriores, nomeadamente a inserção de um elemento por ordem, faça uma função para ordenar uma lista através do algoritmo InsertSort: inserir cada elemento ordenadamente numa lista nova.

Notas:


- Convém ter em atenção para não deixar blocos alocados não libertados. Veja com atenção se todos os blocos alocados no seu código são libertados no final.

Execução de exemplo:

```
C:\>insertsort  
Lista nao ordenada: 249 116 80 686 693 309 598 650 629 676  
Lista: 249  
Lista: 116 249  
Lista: 80 116 249  
Lista: 80 116 249 686  
Lista: 80 116 249 686 693  
Lista: 80 116 249 309 686 693  
Lista: 80 116 249 309 598 686 693  
Lista: 80 116 249 309 598 650 686 693  
Lista: 80 116 249 309 598 629 650 686 693  
Lista: 80 116 249 309 598 629 650 676 686 693  
Lista ordenada: x x x x x x 7 7 8 8
```

Pergunta: construa uma lista com elementos aleatórios entre 0 e 999 (chame `srand(1)`), e ordene no final a lista. Indique na resposta os três primeiros elementos da lista, separados por espaços.

enesimo.c

 Sobre o código desenvolvido no exercício `insertsort.c`, faça uma função para retornar o enésimo elemento numa lista de inteiros.

Notas:


- Ao reutilizar todo o exercício `insertsort.c`, apenas tem de implementar a nova função.

Execução de exemplo:

```
C:\>enesimo  
elementos 250, 500 e 750: xxx xxx xxx
```

Pergunta: utilize o vector do exercício `insertsort.c`, e indique como resposta os elementos nas posições 250, 500 e 750 separados por espaços.

palavras.c

 Faça um programa que recebe o nome de um ficheiro, e retorna o número de palavras distintas do ficheiro, que tenham apenas letras separadas por espaços, tabs e fins-de-linha.

Notas:

- Pode utilizar a função `strtok`


Execução de exemplo:

```
C:\>palavras palavras.c
[1] #define MAXSTR 1024
[10] /* função que dada uma string retorna o número de palavras */
[1] int Palavras(char *str)
[1] char strBackup[MAXSTR];
...

```

Pergunta: faça um ficheiro com o enunciado (desde "palavras.c:" até "fins-de-linha."), e indique quantas palavras são retornadas.

cifracesar.c

 Faça um programa que recebe um ficheiro de texto, e uma chave K, e mostra o ficheiro codificado com a cifra de Júlio César. Esta cifra substitui cada carácter por K caracteres à frente. Aplique a cifra apenas às letras e dígitos, dando a volta tanto nas letras como nos dígitos, isto é, 0 com um K de 11, fica 1 (10 dígitos mais 1). As letras maiúsculas e minúsculas não se misturam, dão a volta separadamente.

Notas:

- Identifique a fórmula que permite obter o dígito certo, dando a volta.

Execução de exemplo:

```
C:\>cifracesar cifracesar.c 13
#vapyhqr <fgqvb.u>
#vapyhqr <pglcr.u>
#vapyhqr <fgevat.u>


#qrsvar ZNKFGE 588

ibvq PvsenQrPrfne(pune *fge, vag punir)
{
    vag v, punirq;
...

```

Pergunta: coloque num ficheiro o texto "lroajlnbja 32101" codificada pela cifra de César com o código "21090". Indique na resposta o seu conteúdo original (utilizar o valor simétrico para decodificar).

more.c

 Faça agora o exercício more, mas para modo binário. Coloque 16 caracteres por linha, na zona da direita coloque os caracteres imprimíveis, na zona da esquerda coloque o número do byte do primeiro carácter na linha, e na zona central 16*3=48 caracteres com o valor hexadecimal de cada carácter.

Notas:

- isprint é uma função da biblioteca ctype.h, que pode ser utilizada para saber se um carácter é imprimível
- Para imprimir a versão hexadecimal de um número, a string de formatação do printf é %x

Execução de exemplo:

C:\>more *more.c*


```

0|23 69 6e 63 6c 75 64 65 20 3c 73 74 64 69 6f 2e |#include <stdio.
16|68 3e 0d 0a 23 69 6e 63 6c 75 64 65 20 3c 63 74 |h> #include <ct
32|79 70 65 2e 68 3e 0d 0a 0d 0a 23 64 65 66 69 6e |ype.h> #defin
48|65 20 4d 41 58 53 54 52 20 31 30 32 34 0d 0a 0d |e MAXSTR 1024
64|0a 69 6e 74 20 6d 61 69 6e 28 69 6e 74 20 61 72 | int main(int ar
80|67 63 2c 20 63 68 61 72 20 2a 2a 61 72 67 76 29 |gc, char **argv)
96|0d 0a 7b 0d 0a 20 20 20 20 46 49 4c 45 20 2a 66 |{ FILE *f
112|3b 0d 0a 20 20 20 20 63 68 61 72 20 2a 70 74 3b |; char *pt;
128|0d 0a 20 20 20 20 69 6e 74 20 69 2c 20 6a 2c 20 | int i, j,
...

```

Pergunta: faça um ficheiro com o enunciado do problema (desde "more.c" até "carácter."), e veja o ficheiro com o utilitário desenvolvido (certifique-se que o ficheiro tem 2 linhas, e cada linha não começa com espaços). Indique na resposta, separado por um espaço, a posição no ficheiro das palavras: "more.c"; "Coloque"; "16*3=48"

basededados.c

 Faça um programa que guarde num ficheiro os registos de contactos, com nome, email, telefone, cidade e descrição. O programa permite manter os registos entre execuções, utilizando um ficheiro binário, e permite listar registos, adicionar/ver/editar qualquer registo.

Notas:

- Assuma que as strings têm um valor máximo de 255 caracteres

Execução de exemplo:

```

C:\>basededados

#####
# Menu: #
#####
# 1 | LISTAR registos #
# 2 | ADICIONAR registo #
# 3 | VER registo #
# 4 | EDITAR registo #
#####
Opção: 2
Nome: José coelho
Telefone: 123 456 789
Cidade: Lisboa
Descrição: ...

#####
# Menu: #
#####
# 1 | LISTAR registos #
# 2 | ADICIONAR registo #
# 3 | VER registo #
# 4 | EDITAR registo #
#####
Opção: 1

#####
# Lista:
#####
# 0 | José coelho
#####

#####
# Menu: #
#####
# 1 | LISTAR registos #
# 2 | ADICIONAR registo #
# 3 | VER registo #
# 4 | EDITAR registo #
#####
Opção: 4
ID: 0
Nome: José Coelho
Telefone: 123 456 789
Cidade: Lisboa
Descrição: ...

#####
# Menu: #
#####
# 1 | LISTAR registos #
# 2 | ADICIONAR registo #
# 3 | VER registo #
# 4 | EDITAR registo #
#####
Opção: 1

#####
# Lista:
#####
# 0 | José Coelho
#####

#####
# Menu: #
#####
# 1 | LISTAR registos #
# 2 | ADICIONAR registo #
# 3 | VER registo #
# 4 | EDITAR registo #
#####
Opção: 2
C:\>basededados

#####
# Menu: #
#####
# 1 | LISTAR registos #
# 2 | ADICIONAR registo #
# 3 | VER registo #
# 4 | EDITAR registo #
#####
Opção: 2
Nome: Afonso Henriques
Telefone: 123 456 789
Cidade: Guimarães
Descrição: ...

#####
# Menu: #
#####
# 1 | LISTAR registos #
# 2 | ADICIONAR registo #
# 3 | VER registo #
# 4 | EDITAR registo #
#####
Opção: 3
ID: 0

#####
# Nome | José Coelho
# Telefone | 123 456 789
# Cidade | Lisboa
# Descrição | ...
#####

#####
# Menu: #
#####
# 1 | LISTAR registos #
# 2 | ADICIONAR registo #
# 3 | VER registo #
# 4 | EDITAR registo #
#####
Opção:

C:\>

```

Pergunta: qual foi o modo de abertura do ficheiro da base de dados, que utilizou para gravar um registo após ser editado?

mergesort.c



Faça uma função para ordenar uma lista de inteiros, com o algoritmo do MergeSort: ordenar cada metade da lista (não obrigatoriamente a primeira metade, apenas metade dos elementos), recursivamente, e no final juntar ambas as metades ordenadas, numa só lista ordenada.

Notas:

- Basta reutilizar as funções de adição e remoção de elementos numa lista, dos exercícios anteriores sobre listas de inteiros.
- Atenção ao enunciado, esta função MergeSort em listas é mais simples que a correspondente versão em vectores

Execução de exemplo:

```
C:\>mergesort
Elemento na posição 1000: x
Elemento na posição 10000: xx
```

Pergunta: Adicione 100.000 de elementos, gerados aleatoriamente e com valores entre 0 e 999 (chame `srand(1);`), e ordene através desta função. Indique como resposta o elemento na posição 1000, seguido do elemento na posição 10.000.

sort.c

Adapte a lista de inteiros para uma lista de strings, e carregue um ficheiro de texto que recebe como argumento do programa. Deve carregar cada linha do ficheiro em cada elemento da lista de strings. Adapte agora o MergeSort de listas de inteiros para listas de strings, e imprima o ficheiro por ordem alfabética das linhas.

Notas:

- Utilize `strcmp` que compara duas strings, devolvendo 0 se as strings são iguais, e valor positivo se uma é maior que a outra na ordem alfabética, e negativo no caso contrário.

Execução de exemplo:

```
C:\>sort sort.c
lista = Adiciona(lista, lista1->str);
lista = Adiciona(lista, lista2->str);
lista = Adiciona(lista, str);
lista = Remove(lista);
lista1 = Remove(lista1);
lista2 = Adiciona(lista2, lista->str);
lista2 = Remove(lista2);
/* alocar espaço para a string, e copiar a string */
...
```

Pergunta: execute a aplicação dando como entrada o ficheiro de texto que tem mais à mão: código fonte C. Qual o primeiro carácter na última linha de praticamente todos os ficheiros com código fonte C?

find.c

Faça um programa que recebe um ficheiro e uma string de procura, com wilcards (* significa zero ou mais caracteres, e ? significa um carácter qualquer), e retorne todas as linhas de match positivas. Considere que um match ocorre apenas dentro de uma só linha.

Notas:

- Faça a função `Find`, que recebe uma string, e um texto de procura, e retorna a primeira ocorrência desse texto na string. A função deve aceitar wilcards e poder ser chamada recursivamente. Se não conseguir os wilcards, faça primeiro sem wilcards, e depois adicione o ?, e finalmente o *
- Pode utilizar como ponto de partida o exercício 3) `more.c`

Execução de exemplo:

```

C:\>find find.c if(*==*)
16:         if(find[ifind] == 0)
19:         if(find[ifind] == '?')
24:         else if(find[ifind] == '*')
33:         else if(find[ifind] == str[istr])
64:     if(f == NULL)
86:         if(pt == str)

```

Pergunta: copie a resolução proposta do exercício 3) `more.c` (primeiro exercício) para um ficheiro de texto, e execute as seguintes procuras sobre esse ficheiro por ordem: "!" ; "<*>" ; "(*,*,*)" ; "[?]" . Indique na resposta os números das linhas retornadas, por ordem e separados por espaços.

histograma.c

Faça um programa que recebe um ficheiro, e faça um histograma dos caracteres alfanuméricos no ficheiro. As maiúsculas e minúsculas devem ser considerados caracteres distintos.

Notas:

- Pode reutilizar a função desenvolvida no exercício 2) `doisdados.c`

Execução de exemplo:


```

C:\>histograma histograma.c
130|                                     #
121|                                     #
113|                                     #
104|                                     #          #
 95|                                     #          #
 87|                                     #          #
 78|                                     #          #
 69|                                     #          #
 61|                                     #          #
 52|                                     #          #
 43|                                     #          #
 35|                                     #          #
 26|                                     #          #
 17| #                                     #          #
  9| ###  #          #          ##          #          #
  0| #####
    +-+-----+-----+-----+-----+-----+-----+-----+-----+
    0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
Numero de cardinais no histograma: 164

```

Pergunta: aproveitando o ficheiro 3) `more.c`, utilizado no exercício anterior, faça o seu histograma e indique na resposta: a frequência absoluta mais alta; letra (ou número) com a frequência mais alta; três letras minúsculas com frequência nulas (ordem alfabética). Coloque os três valores separados por espaços.

indicador.c

 Faça um programa que leia um ficheiro de código em linguagem C, e processe todos os blocos globais, sendo um bloco global todos os pares de chavetas que não estão dentro de nenhum outro par de chavetas. O programa deve retornar o seguinte indicador de complexidade, bem como os seus parciais:

$$I = \sum_{i=1...N} (Conds_i + 2Ciclos_i + 1)^2 + Instruções_i$$

Notas:

- $Conds_i$ é o número de condicionais no bloco global i (if/case)
- $Ciclos_i$ é o número de ciclos no bloco global i (for/while)
- $Instruções_i$ é o número de instruções no bloco global i (contabilizar os ponto e vírgula).
- O programa deve indicar os parciais por cada bloco global, indicando o número de condicionais, ciclos, comandos e o parcial até ao momento após o processamento desse bloco.
- Tenha em atenção aos comentários, strings e caracteres que não são necessários para o cálculo deste indicador

Execução de exemplo:

C:\>*indicador indicador.c*

conds	ciclos	comandos	parcial
0	0	2	3
1	0	6	13
1	0	5	22
0	1	4	35
0	3	5	89
4	3	17	227
2	2	10	286
4	6	24	599
3	3	18	717

Indicador: 717

Pergunta: aproveitando o ficheiro 3) *more.c*, utilizado nos exercícios anteriores, indique na resolução todos os parciais (4 números por cada bloco), separados por espaços.

PARTE IV – ANEXOS

14. COMPILADOR E EDITOR DE C

Neste anexo estão as instruções para a instalação do compilador e editor aconselhado. Pode naturalmente utilizar qualquer outro. No entanto, é de evitar a utilização de um ambiente de desenvolvimento com o compilador, editor e *debugger* integrado, como o Visual Studio, Dev-C++, ou o Eclipse. Terá oportunidade de o fazer quando tiver experiência de programação, actualmente um ambiente de desenvolvimento pode ser um factor de distracção, e em nada contribui para a aquisição das competências: capacidade de implementação de pequenos programas. Aconselha-se um pequeno compilador de linha de comando, e opcionalmente um editor com algumas facilidades de edição e visualização de código C.

TCC: TINY C COMPILER

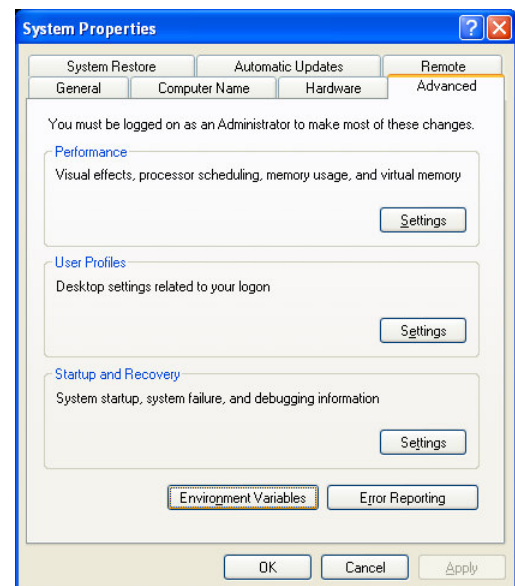
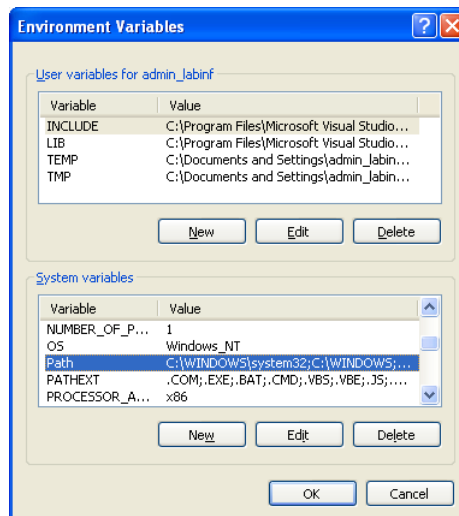
URL: <http://bellard.org/tcc/>

Instalação:

1. Descarregar (Windows binary distribution) do site;
2. Descompactar os ficheiros para "C:\Program Files";
3. Colocar no path a directoria "C:\Program Files\tcc";
4. Escrever e executar o programa "Ola Mundo!".

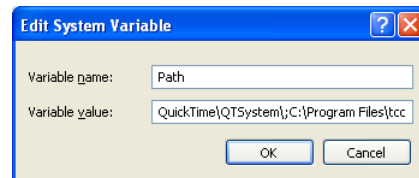
Colocar no path a directoria "C:\Program Files\tcc":

- Control Panel » System » Advanced » Environment Variables:
- Na dialog das variáveis de ambiente, seleccione a variável Path (cuidado para não apagar o seu conteúdo):



- Ao editar, deve adicionar no final a directoria "C:\Program Files\tcc". Atenção que as directorias têm de estar separadas por um ponto e vírgula.

Exemplo de utilização:



Abrir uma linha de comandos:

Iniciar » Todos os Programas » Acessórios » Linha de Comandos

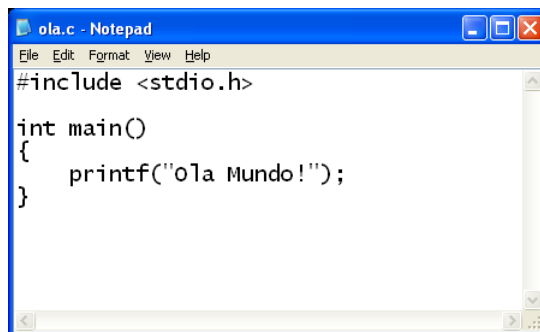
Na linha de comandos vá para a directoria onde pretende ter os exercícios. Para mudar de directoria utilize o comando `cd` (change directory), seguido da directoria (neste exemplo, para não indicar qualquer directoria, utiliza-se a directoria raiz, mas deve criar uma directoria para os exercícios):

```
C:\...\>cd \ ENTER
```

Editar e criar um ficheiro de texto com o Notepad:

```
C:\>notepad ola.c ENTER
```

Ao abrir o Notepad, como o ficheiro "ola.c" não existe, há uma mensagem a dizer que este não existe e se o quer criar. Responda que sim.



Compilar o ficheiro criado:

```
C:\>tcc ola.c ENTER
```

Erros possíveis:

- 'tcc' is not recognized as an internal or external command, operable program or batch file.
 - provavelmente a variável path não tem a directoria do tcc correctamente configurada. Para confirmar isso, execute o comando "C:\>path" e verifique se a directoria do TCC está no path.
- tcc: file 'ola.c' not found
 - provavelmente não editou e gravou um ficheiro ola.c com o Notepad, na mesma directoria onde está. Não se esqueça de copiar exactamente o que está no exemplo, e ao sair do Notepad gravar
- ola.c:1: ';' expected
 - provavelmente o ficheiro gravado não está igual ao exemplo. Copie exactamente o texto no exemplo letra por letra e grave ao sair do Notepad.

Se não houve erros, não é devolvido texto nenhum e aparece novamente a prompt "C:\>". Nesse caso executar o programa criado:

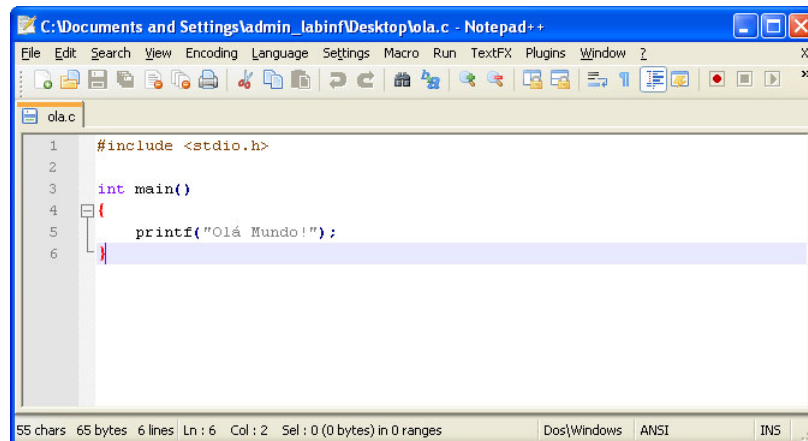
```
C:\>ola ENTER
Ola Mundo!
C:\>
```

Nota: é criado um executável com o mesmo nome que o ficheiro fonte, mas com extensão `exe`. Nesta UC crie programas utilizando apenas um só ficheiro em linguagem C, neste caso o ficheiro `ola.c`.

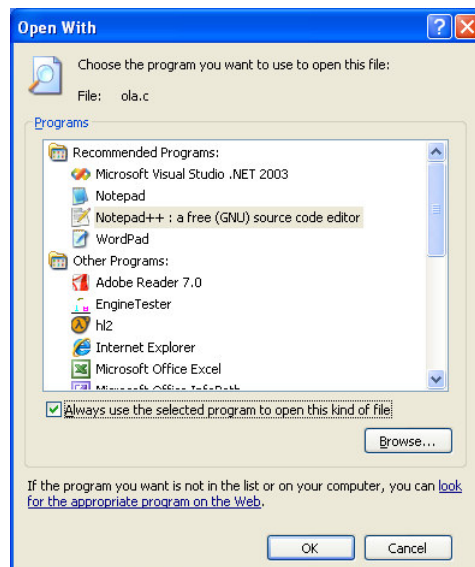
NOTEPAD++

URL: <http://notepad-plus-plus.org/>

Este editor utiliza cores conforme a sintaxe do C (suporta também outras linguagens), de forma a evitar alguns erros sintáticos que porventura se podem cometer por engano no Notepad, e que seriam detectados apenas quando se fosse compilar.



Associe o Notepad++ aos ficheiros com extensão C, de forma a poder abrir o ficheiro no Notepad++ escrevendo apenas o nome do ficheiro na linha de comando, ou clicando no ficheiro no Windows. Clique com o botão direito no ficheiro, e abra seleccionando outra aplicação (selecione o Notepad++, se não estiver listado, utilize o botão "Browse"). Não se esqueça de seleccionar a caixa de selecção em baixo.



Para abrir o ficheiro ola.c com o Notepad++, basta agora escrever o nome do ficheiro:

C:\>ola.c ENTER

15. NÃO CONSIGO PERCEBER O RESULTADO DO PROGRAMA

Ainda não fiz os **exercícios verdes**, e não consigo perceber o que está a acontecer no programa, e porque não tem o resultado que espero:

1. Faça a **execução passo-a-passo** do seu programa. Não há nada como o papel e lápis para compreender um conceito. Irá ver que o programa não faz o que pretende, e assim reflectir sobre a mudança que necessita efectuar no código para ter o comportamento esperado.
2. Não há segundo ponto, se não conseguir fazer a execução passo-a-passo, reveja os capítulos, ou peça ajuda a terceiros.

Já fiz **exercícios azuis**, e tenho um programa que tem um comportamento e/ou resultado incompreensível, aparenta mesmo ter vontade própria:

1. Utilizar os **parâmetros mais simples** possível, se funcionar complicar ligeiramente os parâmetros até obter uma situação de erro;
2. **Imprimir no ecrã os valores** atribuídos a variáveis, na zona suspeita;
3. **Comentar código que é insuspeito**, e verificar se o erro se mantém, continuando até obter o programa mínimo que dá o erro;
4. Tentar **formalizar a questão** para que uma pessoa externa ao projecto possa analisar o problema.

Ao utilizar parâmetros mais simples e verificar que funciona bem, pode identificar a zona do código onde não há problema, ficando apenas parte do código suspeito. Ao imprimir os valores das variáveis, pode detectar o local e a altura em que os valores deixam de estar correctos. Se o programa funcionar comentando o código insuspeito, pode-se confirmar que o problema reside na zona suspeita. Finalmente, ao tentar formalizar a questão, removendo portanto tudo o que é específico do problema e colocando o código no abstracto de forma a se colocar a questão a outra pessoa, pode-se identificar o problema mesmo antes de consultar a pessoa externa ao projecto.

Pode acontecer que seja mais simples a estratégia inversa no **ponto 3**, ou seja, comentar o **código suspeito**. Se não for simples manter o programa a funcionar comentando o código insuspeito, pode ser mais fácil manter o programa a funcionar comentando o código suspeito, e verificar se o problema se mantém. Se desaparecer, pode-se reduzir a zona do código suspeito a comentar, de forma a localizar o problema.

Situação de desespero: estou às voltas com isto, é mesmo incompreensível!

Não há nada incompreensível, o computador não tem vontade própria, nem o problema que está a lidar é complicado demais. Apenas devido ao cansaço não consegue ver pormenores que são simples mas lhe passam ao lado e lhe permitiriam descobrir a causa do problema. O melhor é parar e continuar numa outra altura, após uma noite de descanso. O mais provável será que ao retornar ao problema já tenha encontrado resposta para o problema sem saber como.

EXEMPLO DE CÓDIGO INCOMPREENSÍVEL

Este exemplo foi reportado por um estudante¹⁸, sendo a parte incompreensível o facto de existirem valores finais distintos, com e sem o `printf` na linha 60. Tal não deveria acontecer porque o `printf` não altera valores de variáveis.

Ao código apresentado foi cortado o bloco inicial do `main` que não está relacionada com o problema, e comentários:

```
1 #include <stdio.h>
2 int mdc(int x, int y)
3 {
4     int i, aux;
5     int n = 0;
6     int resto;
7     int mdcl = 1;
8     int vl[1000];
9     resto = y % x;
10    if(resto == 0)
11    {
12        mdcl = x;
13    }
14    else if(resto > 0)
15    {
16        x--;
17        for(i = 1; i < x; i++)
18        {
19            resto = y % i;
20            if(resto == 0)
21            {
22                vl[n] = i;
23                mdcl = i;
24                n++;
25            }
26        }
27        n--;
28        x++;
29        aux = 0;
30        for(i = 1; i <= n; i++)
31        {
32            resto = x % vl[i];
33
34            if(resto == 0)
35            {
36                mdcl = vl[i];
37                aux++;
38            }
39        }
40    }
41    if(aux == 0)
42        mdcl = 1;
43    return mdcl;
44 }
45
46
47
48 void main()
49 {
50     int x, y, i, j;
51     int mdcr = 0;
52     int mdcl = 0;
53     int soma = 0;
54
55     for(i = 1; i <= 100; i++)
56     {
```

¹⁸ Espaço central, UC Programação, 2010/2011

```

57     for(j = i + 1; j <= 100; j++)
58     {
59         soma += mdc(i, j);
60         /* printf(" (%d,%d) mdc =%d\n", i, j, mdc(i, j)); */
61     }
62     printf("A soma intermedia: %d\n", soma);
63 }
64
65 printf("A soma dos MDC dos pares de numeros de 1 a 100: %d\n", soma);
66 }

```

Programa 15-1 Código incompreensível

Execução de exemplo:

```

C:\>exemplo1
A soma intermedia: 99
A soma intermedia: 197
A soma intermedia: 294
A soma intermedia: 414
A soma intermedia: 509
A soma intermedia: 667
...
A soma intermedia: 10147
A soma intermedia: 10148
A soma intermedia: 10148
A soma dos MDC dos pares de numeros de 1 a 100: 10148

```

Este valor está incorrecto, mas descomentando o `printf`, o valor fica correcto. Sabendo que este tipo de funcionamento é originado por acesso a valores de memória fora dos locais declarados, pode-se analisar com atenção o código, de forma a verificar todos os locais em que tal ocorre:

- Utilização de uma variável não inicializada:
 - Todas as variáveis declaradas não inicializadas, têm um comando de atribuição antes da sua utilização
 - Os elementos do vector, antes de serem utilizados são sempre inicializados
- Escrita de uma variável fora da declaração:
 - O único código suspeito é o vector **v1**, que poderia inadvertidamente aceder fora do espaço declarado, nomeadamente para valores negativos: após uma inspecção atenta, verifica-se que o vector é inicializado até **n**, e o ciclo seguinte apenas utiliza valores até **n**, nunca podendo ser negativos

Se quiser pare de ler aqui e tente encontrar o problema com o código acima.

Falhando a inspecção ao código, parte-se para a sua alteração, nomeadamente colocando parâmetros mais pequenos, cortando código irrelevante, e colocando `printfs` com valores das variáveis, para se poder reduzir o código suspeito. É o que é feito no código seguinte, em que o valor 100 nos ciclos foi reduzido para 6, o `printf` com argumentos, foram retirados os argumentos, e adicionados dois `printfs` na função `mdc`, de forma a saber em que valores da função os valores retornados são distintos:

```

1 #include <stdio.h>
2 int mdc(int x, int y)
3 {
4     int i, aux;
5     int n = 0;
6     int resto;
7     int mdc1 = 1;
8     int v1[1000];
9     printf(" (%d,%d)", x, y);
10    resto = y % x;
11    if(resto == 0)

```

```

12     {
13         mdc1 = x;
14     }
15     else if(resto > 0)
16     {
17         x--;
18         for(i = 1; i < x; i++)
19         {
20             resto = y % i;
21             if(resto == 0)
22             {
23                 v1[n] = i;
24                 mdc1 = i;
25                 n++;
26             }
27         }
28         n--;
29         x++;
30         aux = 0;
31         for(i = 1; i <= n; i++)
32         {
33             resto = x % v1[i];
34
35             if(resto == 0)
36             {
37                 mdc1 = v1[i];
38                 aux++;
39             }
40         }
41     }
42     if(aux == 0)
43         mdc1 = 1;
44     printf("%d", mdc1);
45     return mdc1;
46 }
47
48 void main()
49 {
50     int x, y, i, j;
51     int mdcr = 0;
52     int mdc1 = 0;
53     int soma = 0;
54
55     for(i = 1; i < 6; i++)
56     {
57         for(j = i + 1; j <= 6; j++)
58         {
59             printf(" ");
60             soma += mdc(i, j);
61         }
62         printf("A soma intermedia: %d\n", soma);
63     }
64
65     printf("A soma dos MDC dos pares de numeros de 1 a 100: %d\n", soma);
66 }

```

Programa 15-2 Programa incompreensível, preparado para debug

Execução de exemplo com printf na linha 59:

```

C:\>exemplo2
(1,2)=1 (1,3)=1 (1,4)=1 (1,5)=1 (1,6)=1A soma intermedia: 5
(2,3)=1 (2,4)=2 (2,5)=1 (2,6)=2A soma intermedia: 11
(3,4)=1 (3,5)=1 (3,6)=3A soma intermedia: 16
(4,5)=1 (4,6)=2A soma intermedia: 19
(5,6)=1A soma intermedia: 20
A soma dos MDC dos pares de numeros de 1 a 100: 20

```

Execução de exemplo sem printf na linha 59:

```
C:\>exemplo2
(1,2)=1 (1,3)=1 (1,4)=1 (1,5)=1 (1,6)=1A soma intermedia: 5
(2,3)=1 (2,4)=1 (2,5)=1 (2,6)=1A soma intermedia: 9
(3,4)=1 (3,5)=1 (3,6)=1A soma intermedia: 12
(4,5)=1 (4,6)=2A soma intermedia: 15
(5,6)=1A soma intermedia: 16
A soma dos MDC dos pares de numeros de 1 a 100: 16
```

Pode-se agora ver o que não era possível com 100 valores, despistando também a situação da função `mdc` ser chamada duas vezes no ciclo, a segunda no `printf`. As diferenças estão nos pares (2,4), (3,6), (2,6), que são 1 em vez de serem 2, 3 e 2.

Esta informação limita grandemente o código suspeito, sendo agora a inspecção ao código centrada na execução da função `mdc` com esses valores. Torna-se claro qual é o problema: quando o resto da divisão é nulo, a variável `aux` não está inicializada, sendo no entanto utilizada. Se o seu valor desconhecido for nulo, o resultado da função é incorrecto. A inicialização da variável `aux` para um valor não nulo, é suficiente para resolver este problema.

16. FUNÇÕES STANDARD MAIS UTILIZADAS

Exemplos de chamadas:

- `printf("texto %d %g %s %c", varInt, varDouble, varStr, varChar);`
Imprime no ecrã uma string formatada, em que é substituído o %d pela variável inteira seguinte na lista, o %g pela variável real na lista, o %s pela variável string na lista, o %c pela variável carácter na lista.
- `scanf("%d", &varInt); gets(str);`
`scanf` é a função inversa do `printf`, lê um inteiro e coloca o seu resultado em `varInt`, cujo endereço é fornecido. A função `gets` lê uma string para `str`.

Protótipos:

- `int atoi(char *str); float atof(char *str);`
Converte uma string num número inteiro/real respectivamente
- `int strlen(char *str);`
Retorna o número de caracteres da string `str`
- `strcpy(char *dest, char *str); [strcat]`
Copia `str` para `dest`, ou junta `str` no final de `dest`, respectivamente
- `char *strstr(char *str, char *find);`
`char *strchr(char *str, char find);`
Retorna a primeira ocorrência de `find` em `str`, ou NULL se não existe. Na versão `strchr` `find` é um carácter.
- `char *strtok(char *string, char *sep);`
`char *strtok(NULL, char *sep);`
Retorna um apontador para uma token, delimitada por `sep`. A segunda chamada retorna a token seguinte, na mesma string, podendo-se continuar a chamar a função até que retorne NULL, o que significa que a string inicial não tem mais tokens para serem processadas.
- `sprintf(char *str, ...); sscanf(char *str,...);`
Estas funções têm o mesmo funcionamento de `printf/scanf`, mas os dados são colocados (ou lidos) em `str`.
- `int strcmp(char *str1, char *str2);`
Retorna 0 se `str1` é igual a `str2`, retornando um valor negativo/positivo se uma string é maior/menor que a outra
- `int isalpha(int c); [isdigit, isalnum, islower, isupper, isprint]`
Retorna true se `c` é uma letra / dígito numérico / letra ou dígito / minúscula / maiúscula / imprimível.
- `void *malloc(size_t); free(void *pt);`
`malloc` retorna um apontador para um bloco de memória de determinada dimensão, ou NULL se não há memória suficiente, e a função `free` liberta o espaço de memória apontado por `pt` e alocado por `malloc`
- `FILE *fopen(char *fich, char *mode); fclose(FILE *f);`
`fopen` abre o ficheiro com nome `fich`, no modo `mode` ("rt" - leitura em modo texto, "wt" - escrita em modo texto), e `fclose` fecha um ficheiro aberto por `fopen`
- `fprintf(f,...); fscanf(f,...); fgets(char *str, int maxstr, FILE *f);`
idênticos ao `printf/scanf` mas direccionados para o ficheiro, e `fgets` é uma versão do `gets` mas com limite máximo da string indicado em `maxstr`.
- `int feof(FILE *f);`
`feof` retorna true se o ficheiro `f` está no fim, e false c.c.
- `fseek(f, posicao, SEEK_SET); fwrite/fread(registo, sizeof(estrutura), 1, f);`
funções de leitura binária (abrir em modo "rb" e "wb"). `fseek` posiciona o ficheiro numa dada posição, `fwrite/fread` escrevem/lêem um bloco do tipo `estrutura` para o endereço de memória `registo`.
- `int rand(); srand(int seed);`
`rand` retorna um número pseudo aleatório e `srand` inicializar a sequência pseudo aleatória
- `time_t time(NULL); clock_t clock();`
`time` retorna um número segundos que passaram desde uma determinada data, e `clock` o número de instantes (há CLOCKS_PER_SEC instantes por segundo)
- `double sin(double x); [cos, log, log10, sqrt]`
`double pow(double x, double y);`
Funções matemáticas mais usuais, com argumentos e valores retornados a `double`

17. ERROS COMUNS

Neste anexo compilam-se os erros mais comuns encontrados em actividades de avaliação, de forma a auxiliar a própria pessoa possa rever o seu código, bem como o avaliador a corrigir trabalhos, que assim é suficiente referir que para o bloco de código X aplica-se o erro Y, não sendo necessário reproduzir toda a argumentação em volta do erro. Cada erro é identificado com um **nome**, a **forma** em que ocorre do ponto de vista de quem comete o erro, o **problema** que o erro implica, e a **resolução** a adoptar por quem tenha o código com o erro. No problema é indicada a gravidade do erro, um de três níveis: baixa, moderada, elevada. Este indicador de gravidade foi feito tendo em vista a descontar, num critério em que estes erros sejam contabilizados, 33% pela existência de um erro de gravidade elevada, 25% pela existência de um erro de gravidade moderada, e 10% pela existência de um erro de gravidade baixa. Apenas os 3 erros mais graves devem ser contabilizados, para que quem tenha cometido apenas erros de gravidade baixa, sofra um desconto máximo de 30%.

Erro	Forma	Problema	Resolução
Funções com parâmetros no nome	Para distinguir uma constante importante na função, basta colocar o valor do parâmetro no nome da função, por exemplo funcao12, para a função que retorne o resto da divisão por 12. Desta forma evita-se utilizar um argumento.	Se distingue uma constante importante no nome da função, então deve colocar a constante como argumento da função, ficando a função a funcionar para qualquer constante. Caso não o faça, corre o risco de ao lado de funcao12, ser necessário a funcao4, funcao10, etc., ficando com instruções parecidas. Não poupa sequer na escrita dado que coloca no nome da função a constante que utiliza na função. Gravidade: elevada	Se tem casos destes no seu código, deve fazer uma troca simples: substituir todas as constantes dependentes de 12 pelo argumento, ou uma expressão dependente do argumento.
Variáveis Globais	Em vez de utilizar alguns dos argumentos nas funções, declara-se a variável globalmente em vez de na função main, de forma a não só ser acessível a todas as funções, como não ser necessário passá-la como argumentos nas funções	Ao fazer isto, não só mascara as más opções na divisão do código por funções, dado que não controla os argumentos da função, como mais importante, perde principal vantagem das funções: quando implementa/revê a função, apenas tem de considerar a função e pode abstrair-se do resto do código. Como há variáveis globais, ao implementar a função tem que se considerar todos os restantes locais que essas variáveis são utilizadas (leitura / atribuição), para que o código na função seja compatível. Por outro lado, ao não se aperceber quando cria a função do número de argumentos que recebe e número de variáveis que altera, pode estar a criar funções que reduzem muito pouco a complexidade, sendo a sua utilidade diminuta. Gravidade: elevada	Utilize o seu código com variáveis globais para fazer uma revisão geral. Deve remover todas as variáveis globais e colocá-las na função main, deixando apenas as constantes globais que façam sentido estar definidas globalmente, e macros. Depois deve ir refazendo os argumentos das funções, e eventualmente rever as funções que necessita de forma a ter funções com poucos argumentos e com uma funcionalidade clara e simples.
Variáveis com nomes quase iguais	Algumas variáveis necessárias têm na verdade o mesmo nome, pelo que se utiliza os nomes nomeA, nomeB, nomeC, nomeD, ou nome1, nome2, nome3, nome4	Esta situação é uma indicação clara que necessita de um vector nome[4]. Se não utilizar o vector não é possível depois fazer alguns ciclos a iterar pelas variáveis, e irão aparecer forçosamente instruções idênticas repetidas por cada variável. Gravidade: elevada	Deve nesta situação procurar alternativas de fazer código em que esta situação lhe tenha ocorrido. Não utilize código de outra pessoa, apenas o seu código com este tipo de erro lhe poderá ser útil.
Nomes sem significado	Atendendo a que um bom nome gasta tempo e é irrelevante para o bom funcionamento do código, qualquer nome serve.	O código deve ser lido não só pelo compilador como por outras pessoas, pelo que a má utilização de nomes revela por um lado a má organização mental de quem o escreve, e por outro um desrespeito por quem lê o código. A troca de todos identificadores por nomes abstractos e sem sentido, pode ser uma forma para tornar o código ilegível, no caso de se pretender protegê-lo. Pode também ser uma situação confundida com uma situação de fraude, resultante de uma tentativa de um estudante alterar os identificadores de um código obtido ilicitamente. Gravidade: elevada	Verificar todos os identificadores de uma só letra, ou muito poucas letras, bem como nomes abstractos, funções sem uma grandeza associada, e procedimentos sem verbo. Se não consegue encontrar nomes claros para um identificador, seja uma variável, uma função, o nome de uma estrutura, etc., pense numa palavra em que explique para que a variável serve, o que a função/procedimento faz, o que contém a estrutura, etc. Se não consegue explicar isso, então o identificador não existe.

Erro	Forma	Problema	Resolução
Utilização do goto	Numa dada instrução, sabe-se a condição pretendida para voltar para cima ou saltar para baixo. É suficiente colocar um <i>label</i> no local desejado, e colocar o <i>goto</i> dentro do condicional. Este tipo de instrução pode ser motivado pelo uso de fluxogramas.	Perde-se nada mais nada menos que a estrutura das instruções. A utilização ou não desta instrução, é que define se a linguagem é estruturada, ou não estruturada (por exemplo o Assembly). Se de uma linha de código se poder saltar para qualquer outra linha de código, ao analisar/escrever cada linha de código, tem que se considerar não apenas as linhas que a antecedem dentro do bloco actual, como todas as linhas de código, dado que de qualquer parte do programa pode haver um salto para esse local. Esta situação é ainda mais grave que a utilização de variáveis globais, dado que para compreender 1 linha de código, não é apenas os dados que necessitam de ser considerados, como também todas as instruções no programa. A complexidade do programa nesta situação cresce de forma quadrática com o número de linhas. Mesmo sem goto, no caso de variáveis globais, a complexidade do programa crescerá de forma linear, enquanto sem variáveis globais a complexidade é igual à maior função, pelo que cresce menos que o número de linhas de código. Gravidade: elevada	Utilizar as estruturas de ciclos disponíveis, bem como condicionais, e funções. Se o salto é para trás dentro da mesma função é certamente um ciclo o que é pretendido, mas se é para a frente, provavelmente é um condicional. Se é um salto para uma zona muito distante, então é provavelmente uma função que falta. No caso de utilizar fluxogramas, deve considerar primeiro utilizar o seu tempo de arranque vendo execuções passo-a-passo de forma a compreender o que é realmente um programa, sem segredos, e resolver exercícios.
Funções muito grandes	Tudo o que é necessário fazer, vai sendo acrescentado na função, e esta fica com cada vez uma maior dimensão. Esta situação ocorre devido ao problema ser muito complexo.	Ao ter uma função grande, vai ter problemas de complexidade. Não só as declarações de variáveis da função ficam longe do código onde são utilizadas, como não consegue visualizar a função de uma só vez, sendo muito complicado verificar o código. Esta é uma situação clara de má utilização da principal ferramenta da programação, que lhe permite controlar a complexidade do código: abstracção funcional. Gravidade: moderada	Deve dividir a função em partes que façam sentido, de forma a ter funções pequenas e com poucos argumentos. Dividir a função às fatias é má ideia, dado que os argumentos necessários para cada fatia podem ser muitos. Se houver várias fases, sim, colocar cada fase numa função, caso contrário deve verificar se os ciclos interiores podem construir funções que não só utilizem poucos argumentos como também tenham possibilidade de ser utilizadas em outras situações. Se a função reunir estas condições, terá certamente um nome claro.
Funções com muitos argumentos	Após criar a função, ao adicionar como argumentos as variáveis que esta utiliza (leitura / atribuição), este valor revela-se elevado, mas é o que é necessário.	Se o número de argumentos é muito elevado, a abstracção que a função representa é fraca, uma vez que para utilizar a função, é necessário ter muita informação. Para além disso, se a função for muito curta, pode dar mais trabalho a arranjar os parâmetros certos para chamar a função, que colocar o seu código, sendo assim a função mais prejudicial que útil. No caso dos diversos argumentos estarem relacionados, significa que não foram agregados numa estrutura. Gravidade: moderada	Deve procurar alternativas na criação de funções. Dividir uma função existente a meio, provoca situações destas. Para desagregar uma função, deve procurar o código repetido, se não existir, o código no interior dos ciclos é tipicamente um bom candidato a função, de forma a manter em cada função um ou dois ciclos. No caso de as variáveis estarem relacionadas, criar uma estrutura para as agregar.
Indentação variável	Como a indentação é ao gosto do programador, cada qual coloca a indentação como lhe dá mais jeito ao escrever.	A leitura fica prejudicada se a indentação não for constante. Por um lado não é fácil identificar se há ou não algum erro do programador em abrir/fechar chavetas, dado que a indentação pode ser sua opção e não esquecimento de abrir/fechar chavetas. Por outro lado, não é possível a visualização rápida das instruções num determinado nível, dado que o nível é variável, sendo necessário para compreender algo de o código analisá-lo por completo. Com o código indentado, pode-se ver o correcto funcionamento de um ciclo externo, por exemplo, sem ligar ao código interno do ciclo, e assim sucessivamente. Gravidade: moderada	Indentar o código com 2 a 8 espaços, mas com o mesmo valor ao longo de todo o código. Se utilizar tabs, deve indicar o número de espaços equivalente, no cabeçalho do código fonte, caso contrário considera-se que o código está mal indentado. Se pretender subir a indentação com a abertura de uma chaveta, pode fazê-lo mas tem de utilizar sempre chavetas, caso contrário ficam instruções que estão no mesmo nível mas no código ficam e indentações distintas. Qualquer que seja as opções, não pode existir instruções no mesmo alinhamento que pertençam a níveis distintos (ou vice-versa). A posição das chavetas é indiferente, mas tem que se apresentar um estilo coerente ao longo do código.

Erro	Forma	Problema	Resolução
Instruções parecidas seguidas	Quando é necessário uma instrução que é parecida com a anterior, basta seleccioná-la e fazer uso de uma das principais vantagens dos documentos digitais: copy/paste.	O código ficará muito pesado de ler, podendo também prejudicar a escrita. É sinal que está a falhar um ciclo, em que as pequenas mudanças entre instruções, em vez de serem editadas e alteradas, essa alteração é colocada dependente da variável iteradora, fazendo numa só instrução a chamada a todas as instruções parecidas, debaixo de um ciclo. O código não só fica mais simples de ler, como se for necessária outra instrução basta alterar a expressão lógica de paragem do ciclo, em vez de um copy/paste e edição das alterações. Gravidade: moderada	Se encontra situações destas no seu código, estude alternativas para utilizar ciclos.
Instruções parecidas não seguidas	Quando é necessário uma ou mais instruções que são parecidas com outras que já escritas noutra parte do código, basta seleccionar e fazer uso de uma das principais vantagens dos documentos digitais: copy/paste.	O copy/paste vai dificultar a leitura, e também a escrita tem de ser com muito cuidado, para poder editar as diferenças. Se esta situação ocorrer, é uma indicação clara que é necessária uma função com as instruções que são parecidas e devem ser reutilizadas. Nos argumentos da função deve ir informação suficiente para que a função possa implementar as diferenças. Gravidade: moderada	Se aconteceu no seu código, deve estudar quais as diferentes alternativas para utilização de funções que tem, de forma a garantir que não lhe escapam hipótese antes de optar.
Utilização de “system”	Para implementar algumas funções, que existem no sistema operativo, pode-se utilizar a função <code>system</code> , pelo que há que aproveitar.	Quer seja a mudar o código de página, quer seja a limpar o texto da consola, ou a copiar um ficheiro de um lado para o outro, qualquer comando que escreva, é dependente do sistema operativo. Perde portabilidade e o seu programa em vez de fazer uma função concreta, não mais é que um script que chama outros programas, e para isso todos os sistemas operativos têm uma linguagem de scripting mais adequada. Se quiser utilizar código de outros programas, deve integrá-los. Gravidade: moderada	Remova todas as chamadas a “system”, e faça o que é pretendido fazer no programa com o seu próprio código.
Má utilização do switch ou cadeia if-else	De forma a executar instruções dependente do valor de um índice, utilizar um switch colocando as diversas instruções para cada caso do índice.	Muitas vezes ao utilizar no <code>switch</code> um índice (ou cadeia <code>if-else</code>), as instruções em cada caso ficam muito parecidas, ficando com código longo e de custo de manutenção elevado. Se essa situação ocorrer deve rever o seu código, dado que está provavelmente a falhar uma alternativa mais potente, que é a utilização de vectores. Gravidade: moderada	Criar um vector com as constantes necessárias para chamar numa só instrução todas as outras dependentes do índice.
Comentários explicam a linguagem C	De forma a explicar tudo o que se passa no programa, e não ser penalizado, por vezes desce-se tão baixo que se explica inclusive linguagem C.	Esta situação é um claro excesso de comentários, que prejudica a legibilidade. Quem lê código C sabe escrever código C, pelo que qualquer comentário do tipo “atribuir o valor X à variável Y”, é desrespeitoso para quem lê, fazendo apenas perder tempo. Ao comentar instruções da linguagem e deixar de fora comentários mais relevantes, pode indicar que o código foi reformulado por quem na verdade não sabe o que este realmente faz. Gravidade: moderada	Escreva como se estivesse a escrever a si próprio, supondo que não se lembrava de nada daqui a uns 6 meses.
Má utilização da Recursão	Uma função tem um conjunto de instruções que têm de ser executadas várias vezes. Uma forma de fazer isto é no final da função colocar um teste a controlar o número de execuções, e chamar a função recursivamente.	Se a chamada recursiva está no final, e o código tem de passar sempre por esse condicional, então pode remover a chamada substituindo por um ciclo. Ao chamar a função recursivamente, as variáveis locais já criadas não vão mais ser necessárias, mas por cada passo do ciclo ficará um conjunto de variáveis locais à espera de serem destruídas. Se o ciclo for muito grande, o stack irá certamente rebentar. Gravidade: moderada	Substituir a chamada recursiva por um ciclo e respectivo teste de paragem. Caso a chamada recursiva não seja no final da função, ou exista mais que uma chamada recursiva, então sim, é preferível utilizar-se recursão.

Erro	Forma	Problema	Resolução
Atribuições entre variáveis de tipos distintos	Principalmente quando se utiliza apontadores, por vezes é necessário fazer atribuições entre apontadores de tipos distintos, ou entre apontadores e inteiros. Isso não é problema porque um apontador é um número de 32 bits.	Se atribui um apontador de um tipo a outro, este vai aceder ao endereço de memória apontado como se fosse o segundo tipo, quando este foi alocado e tem dados como se fosse o primeiro tipo. Pode ter consequências imprevisíveis, uma vez que os dados ficam com o acesso incoerente. Em caso algum deve assumir o tipo do apontador, uma vez que tal pode variar conforme o compilador/computador em que o código for utilizado. Gravidade: moderada	Se for necessária atribuição entre apontadores de tipos distintos, ao alocar memória por exemplo, então deve utilizar no código um cast para o tipo de destino, de forma a clarificar que não se trata de um erro. Fora a situação de alocação de memória, não precisa de qualquer outra atribuição entre tipos distintos. Em situações menos ortodoxas, pode utilizar o tipo union, para fundir dados de mais de um tipo.
Declarações ou atribuições a variáveis nunca utilizadas	Por vezes declaram-se variáveis, ou fazem-se atribuições a variáveis, um pouco para ver se o programa passa a funcionar. No entanto, após muito corte e costura, por vezes ficam atribuições a variáveis que na verdade nunca são utilizadas, embora tal não afecte o bom funcionamento do programa.	Se há uma variável declarada que não é utilizada, essa variável pode ser removida e o código fica igual. Se há uma atribuição a uma variável que depois não é utilizada em nenhuma expressão, então a atribuição é desnecessária. Ter variáveis, expressões, atribuições a mais, é uma situação indesejável, dado que compromete a leitura do código que realmente funciona. Gravidade: moderada	Verificar se há variáveis declaradas que não são utilizadas, e apagá-las. Para cada atribuição, seguir o fluxo do código até uma possível utilização. Se não existir, apagar a atribuição, reiniciando o processo de verificação. No final o código real poderá ser muito mais reduzido, do que o gerado em situações de stress em modo de tentativa/erro.
Linhas de código nunca executadas	Por vezes em situações de stress tentam-se várias alternativas, e fica algum código que na verdade nunca tem hipótese de ser executado, mas que também não atrapalha.	Se há código a mais, este deve ser removido sobre pena de perda de legibilidade e manutenção. Esta situação é ampliada quando há código de diversas proveniências, em que ao integrá-lo ninguém se atreve a reeditar código que funciona, para seleccionar a parte do código que deve ser integrado. Gravidade: moderada	Identificar as zonas de código que de certeza que não são executadas de modo algum, e apagá-las. Tanto pode ser feito por inspecção ao código como através de testes, submetendo o código a diversos casos de teste, desde que previamente se tenha colocado informação de debug em cada bloco de código. Os blocos de código em que não tiver sido produzida nenhuma informação de debug, são provavelmente os blocos de código que nunca são executados.
Alocar memória e não testar se a operação foi bem sucedida	A alocação de memória é uma operação comum, e se o programa não utiliza grandes quantidades de memória, é sempre bem sucedida, pelo que não vale a pena efectuar a verificação já que complica desnecessariamente o código.	Mesmo que a aplicação utilize pouca memória, a alocação de memória pode falhar se o sistema operativo tiver a memória esgotada por outras aplicações, ou por outras instâncias da mesma aplicação. A não verificação pode provocar acesso a zonas de memória proibidas com resultados imprevisíveis. Gravidade: moderada	Em todas as alocações de memória, verifique se estas foram bem sucedidas, e no caso de falharem, certifique-se que o programa tem um procedimento válido.
Funções específicas	Para uma situação é necessária uma função concreta, pelo que é essa função que é definida, utilizando as constantes necessárias dentro dessa função.	Se as funções são muito específicas, não podem ser reutilizadas, tanto no mesmo programa, como em outros programas. Quanto mais genérica é a função, maior é o seu grau de reutilização, sendo mais simples de manter e detectar algum problema. Gravidade: baixa	Reveja todas as constantes que tem numa função específica, e passe as constantes que sejam passíveis de serem mudadas para argumentos da função, de forma a aumentar as possibilidades da função ser útil sem alterações em outras situações.
Variáveis desnecessárias	Ao criar uma variável por cada necessidade de registo de valores, pode acontecer haver variáveis sempre com o mesmo valor, mas tal não impede que o programa funcione correctamente.	Ter muitas variáveis com a mesma função tem o mesmo problema que “uma atribuição, uma leitura”. Fica o código mais complexo e difícil de ler sem qualquer ganho. Pode acontecer que este problema se reflecta não apenas por ter duas variáveis sempre com os mesmos valores, mas terem valores com uma relação simples (o simétrico, por exemplo). Gravidade: baixa	Deve identificar as variáveis com atribuições e utilizações perto uma da outra, eventualmente os nomes, e verificar se pode a cada momento utilizar o valor de uma variável em vez da outra. Por vezes não é simples a identificação/remoção destas situações quando há muitas variáveis. É preferível aplicar primeiramente a remoção de declarações/atribuições desnecessárias, e aplicar a resolução do erro “uma atribuição, uma leitura”, antes deste erro.

Erro	Forma	Problema	Resolução
Declaração de variáveis fora do início de funções	As variáveis devem ser declaradas o mais perto possível do local onde são utilizadas, e como alguns compiladores de C permitem a declaração fora do início das funções, há que aproveitar.	É verdadeira a frase, mas não para quem se inicie na programação, nem para a linguagem C. A generalidade dos compiladores de C permite declaração das variáveis em qualquer parte do código, mas nem todos, pelo que perde compatibilidade. Mais relevante é a má influência que tal situação terá, facilitando a existência de funções grandes. Com as variáveis declaradas em cada bloco, não irá “sofrer” os efeitos de funções grandes tão intensamente, o que o pode levar a não sentir a necessidade de criar funções. Gravidade: baixa	Todas as declarações fora do início das funções passam para o início das funções. Reveja os nomes das variáveis se necessário.
Duas instruções na mesma linha	Quando as instruções são pequenas e estão relacionadas, é mais simples colocá-las todas na mesma linha. Como é uma questão de estilo, não afecta o funcionamento.	Se há instruções relacionadas, estas devem ser colocadas na mesma função, se tal se justificar. Colocar duas ou mais instruções na mesma linha vai prejudicar a legibilidade e não só. O compilador quando encontra algum problema refere a linha do código, e existindo mais que uma instrução nessa linha, a informação é menos precisa. Deixa de ser possível também seguir um nível de indentação para passar por todas as instruções nesse nível, tem que se fazer uma leitura atenta por todo o código do bloco. Gravidade: baixa	Em todas as linhas com mais que uma instrução, colocar uma instrução por linha.
Ciclos contendo apenas uma variável lógica no teste de paragem	Esta estratégia consiste em utilizar nos ciclos (normalmente while), uma variável no teste de paragem, atribuindo o valor verdadeiro e dentro do ciclo, quando for necessário sair dele, fazer uma atribuição à variável a falso.	O real teste de paragem fica diluído nas instruções que alteram esta variável, bem como condicionais que utilizem o break, e ter algo em vez de num local em vários, torna sempre mais complicada a leitura e manutenção de código. Esta situação ocorre porque falhou na identificação do real teste de paragem. É pior se necessitar de acompanhar esta variável com uma outra variável que altera de valor em cada ciclo (uma variável iteradora). Gravidade: baixa	Se optar com frequência para esta situação, deve rever o seu código procurando alternativas.
Nomes não normalizados	Os nomes para as variáveis, funções, estruturas não afectam o bom funcionamento do código, pelo que não vale a pena perder muito tempo. Ora ficam de uma maneira, ora ficam de outra.	A leitura é severamente penalizada se não existir uma norma clara seguida para os identificadores no código, na separação de palavras e utilização de abreviaturas. Um identificador com um mau nome pode ser trocado o seu significado sem que o leitor se aperceba, a não ser que tenha muita atenção, resultando em horas perdidas à procura de um bug que facilmente encontraria com nomes normalizados. As abreviaturas a utilizar, devem incidir apenas para relacionar funções, por exemplo, e nunca os nomes devem ser apenas abreviaturas ou abstractos. Gravidade: baixa	Refazer os identificadores para utilizar uma só norma de separação de nomes/abreviaturas. Se houver código de diversas proveniências, aproveitar para rever e reformular de forma a permitir a sua manutenção.
Macros em todas as constantes	Tudo o que é constante tem direito a uma macro, de forma a poder ser facilmente alterada	A utilização de macros tem um custo, que incorre nas linhas de código onde é utilizada: não é conhecido o valor real da macro. Se esta for utilizada em situações em que o fluxo de informação seja claro, como um valor inteiro limite sendo os ciclos desde uma constante até esse limite, não há qualquer problema. Se no entanto for utilizada em locais dependentes de outras constantes, sejam em macros sejam no código, ou que seja importante saber o valor da macro para saber qual a sequência de instruções, ou num caso extremo, o valor da macro é necessário para saber se uma instrução é válida, evidentemente que o valor da macro em si não pode ser alterado sem que o código seja revisto. Perde portanto o sentido, uma vez que prejudica a leitura do código e não facilita qualquer posterior alteração. As macros devem ser independentes entre si e de outras constantes. Gravidade: baixa	Remova do código as macros que são dependentes entre si, ou que dificilmente podem mudar sem implicar outras alterações no código.

Erro	Forma	Problema	Resolução
Nomes muito longos	Para explicar bem o identificador, este tem que ter três e quatro palavras, ou mesmo mais.	Com os identificadores muito longos, o código torna-se menos legível, e provavelmente haverá mais instruções a ocuparem mais de uma linha de código. Gravidade: baixa	Deve procurar o essencial da variável / função / estrutura, de forma a dar o nome sobre o essencial, e comentar se necessário na declaração as particularidades que não são cobertas pelo nome do identificador. Utilize no máximo 3 palavras, separadas por maiúsculas ou por “_”.
Comentários inexistentes em partes complexas	O código deve ser lido por quem perceba não só da linguagem de programação C, como de algoritmos, e que seja suficientemente inteligente para o compreender.	Quem lê código C deve naturalmente possuir todas essas características, mas tal não é motivo para não explicar as partes complexas. Os comentários podem afectar a legibilidade do código, e quem está a ler código em que tem dificuldade em compreender, não irá apreciar a forma rebuscada e hábil de quem o escreveu, mas sim a simplicidade e clareza com que o fez, ou o explica. Se não conseguir compreender algo, será certamente por culpa de quem escreveu que não sabia o que fazia e lá acabou por obter uma solução funcional em modo de tentativa e erro, de baixa qualidade. Gravidade: baixa	Nas zonas menos claras, comentar explicando todos os truques / habilidades aplicadas. Se ficar algum por explicar, será confundido certamente com código que nem quem o fez sabe realmente porque funciona, e mais vale refazer novamente se houver algum problema.
Uma atribuição, uma leitura	Uma variável é atribuída com um valor, para depois utilizá-lo na instrução seguinte, não sendo mais necessário o valor da variável. Por exemplo, para reunir os valores necessários à chamada de uma função.	Esta situação pode indicar a existência de uma variável desnecessária, e quantas mais variáveis desnecessárias o código tiver, mais complexo fica, para não falar que ocupam memória desnecessariamente. É apenas justificável no caso de a expressão ser muito complexa, de forma a clarificar o código. Gravidade: baixa	Ao fazer a atribuição do valor à variável, utiliza uma expressão. Essa variável é depois utilizada de seguida também uma só vez, pelo que mais vale colocar a própria expressão no local onde a variável é utilizada.
Não libertar memória após alocar	Atendendo a que não há erros por não libertar memória, e sendo a aplicação de curta duração, justifica-se a não libertação da memória, já que quando a aplicação terminar tudo é libertado.	A aplicação tem os endereços dos blocos de memória que alocou, já não precisa deles, pode libertá-los. Se não o fizer, não há garantia da altura em que esses blocos serão libertos, nem se o custo computacional de os libertar é o mesmo. O argumento da pequena aplicação não deve ser justificação para qualquer erro, o código que funciona em pequenas aplicações poderá ser reutilizado em outras aplicações, e se for bem feito, não necessitará de alterações. Gravidade: baixa	Tenha sempre atenção ao alocar memória do local exacto onde esse bloco será certamente libertado.
Abrir um ficheiro e não chegar a fechar	Por vezes abre-se o ficheiro, mas a operação de fechar não causa erros e é desnecessária dado que o ficheiro será fechado quando o programa acabar.	O número de ficheiros abertos é um recurso limitado pelo sistema operativo. Se um ficheiro não é mais preciso, deve ser imediatamente fechado de forma a libertar recursos. Se esta prática não for feita, pode acontecer que pedidos de abertura de ficheiros por esta ou outra aplicação sejam recusados. Gravidade: baixa	Verificar o último local onde o ficheiro é necessário, e chamar a correspondente função <code>fclose</code> de forma a fechar o ficheiro.
Abrir um ficheiro e não testar se foi bem aberto	Quando se sabe que o ficheiro existe, não é necessário testar se a operação de abertura do ficheiro foi bem sucedida, é perda de tempo.	Pode acontecer mesmo quando há a certeza que o ficheiro existe, que a operação de abertura de ficheiro seja mal sucedida, no caso de estarem demasiados ficheiros abertos, ou por questões de permissões, ou ainda por estarem duas aplicações a tentar abrir o ficheiro em modo de escrita. As operações seguintes, que utilizam o apontador retornado para o ficheiro, ficam com um procedimento incerto. Gravidade: baixa	Testar sempre se o ficheiro foi bem aberto, e apenas utilizar o apontador para o ficheiro no caso de este ser diferente de NULL.

18. EXERCÍCIOS: DICAS, RESPOSTAS E RESOLUÇÕES

Neste anexo são dadas dicas aos exercícios, forma para se validar a resolução, e resoluções.

DICAS EXTRA

Exercício / problema	Explicação / sugestão / dica
1) soma.c não compreendo a fórmula	Na fórmula $\sum_{i=1}^N i$ o símbolo \sum significa que se deve somar a expressão seguinte, tendo em conta que existe uma variável que vai variar entre dois valores (valor indicado em baixo, até ao indicado em cima). A variável utilizada neste caso é a variável i , desde o valor 1 até ao valor N . Se $N=4$, a fórmula acima é equivalente a $\sum_{i=1}^4 i = 1 + 2 + 3 + 4$
1) produto.c não compreendo a fórmula	Na fórmula $N! = \prod_{i=1}^N i$ o símbolo \prod é o equivalente ao \sum mas para produtos. Neste caso, se $N=4$ a expressão acima fica $4! = \prod_{i=1}^4 i = 1 \times 2 \times 3 \times 4$
1) arranjos.c não compreendo a fórmula	A fórmula $A(N, R) = \frac{N!}{(N-R)!} = \prod_{i=N-R+1}^N i$ utiliza apenas elementos já introduzidos nos exercícios anteriores. Neste caso, para $N=4$ e $R=2$ a expressão fica: $A(4, 2) = \frac{4!}{(4-2)!} = \frac{4!}{2!} = \prod_{i=4-2+1}^4 i = \prod_{i=3}^4 i = 3 \times 4$
1) arranjos.c dá-me 5079110400, que é o valor correcto mas não está nas respostas possíveis	O valor correcto de combinações 20, 8 a 8 é realmente esse. No entanto essa não é a resposta correcta ao exercício, que deve ser resolvido com inteiros de 4 bytes. Troque todas as variáveis inteiras para inteiros de 4 bytes e execute novamente o programa.
1) arranjos.c Dá-me valores estranhos, calculo $N!$ e $(N-R)!$, e depois divido um valor pelo o outro	Calculando ambos os factoriais está a fazer não só mais contas, como também ultrapassa o limite dos inteiros mais cedo, atendendo que o factorial cresce muito rapidamente. Se tiver um valor de R pequeno, pode conseguir calcular arranjos mesmo com um N grande, em que não consegue obter o factorial de N , mas consegue obter arranjos de N, R a R . Utilizar um tipo mais potente, resolve o problema do limite para valores pequenos, mas teria o problema à mesma para muitos valores que poderiam ser calculados pela segunda fórmula. Atenção que todos os tipos inteiros devem ter 4 bytes, atendendo a que os exercícios estão equilibrados para esse tipo de dados.

<p>1) fibonacci.c não compreendo a fórmula</p>	$F(n) = \begin{cases} n & , n \leq 2 \\ F(n-1) + F(n-2) & , n > 2 \end{cases}$ <p>Esta fórmula tem uma definição recursiva. Para o valor 1 e 2, retorna 1 e 2 respectivamente (primeiro condicional). Para 3 ou superior, retorna a soma dos dois valores imediatamente anteriores.</p> <p>Exemplos:</p> <ul style="list-style-type: none"> • $F(3) = F(2) + F(1) = 2 + 1 = 3$ • $F(4) = F(3) + F(2) = 3 + 2 = 5$ • $F(5) = F(4) + F(3) = 5 + 3 = 8$ • $F(6) = F(5) + F(4) = 8 + 5 = 13$
<p>1) combinacoes.c não compreendo a fórmula</p>	$C(N, R) = \frac{N!}{(N-R)!R!} = \prod_{i=N-R+1}^N i / \prod_{i=1}^R i$ <p>Veja primeiramente as explicações para 1) arranjos.c. Para N=5, R=3, a expressão acima fica:</p> $\frac{3 \times 4 \times 5}{1 \times 2 \times 3}$ <p>Veja neste pequeno exemplo, como fica a nota sobre a multiplicação e divisão em simultâneo a cada passo.</p>
<p>1) combinacoes.c obtenho o valor incorrecto</p>	<p>Verifique se não estará a utilizar a fórmula incorrecta, por exemplo:</p> $\prod_{i=R}^N i / \prod_{i=1}^R i$
<p>1) euler.c não compreendo a fórmula</p>	<p>É pretendido o cálculo do valor da constante de euler através da fórmula:</p> $e = \sum_{n=0}^K \frac{1}{n!}$ <p>Por exemplo, se K=9, a expressão pretendida é:</p> $\frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \dots + \frac{1}{9!}$ <p>Utilizando apenas as 4 operações básicas fica:</p> $\frac{1}{1} + \frac{1}{1} + \frac{1}{1 \times 2} + \dots + \frac{1}{1 \times 2 \times 3 \times \dots \times 9}$
<p>1) trocos.c não percebo a pergunta</p>	<p>Tem de contar o número de moedas retornadas. Se fosse pedido o montante do exemplo, isto é, 1.79€, o valor a colocar na resposta seria 6 (1 moeda de 1€, 50, 20 e 5 cêntimos e 2 moedas de 2 cêntimos).</p>

<p>1) trocos.c com o montante 0.02 não dá o valor correcto</p>	<p>Pode estar a utilizar em números reais igualdades. Não deve fazê-lo, dado que a representação dos reais tem precisão limitada. Veja-se o programa apresentado no fórum¹⁹:</p> <pre> 1 #include <stdio.h> 2 int main() 3 { 4 float montante; 5 printf("Indique um montante igual a 0.02: "); 6 scanf("%f", & montante); 7 if(montante == 0.02) 8 printf("montante = 0.02\n"); 9 else 10 printf("montante diferente de 0.02\n"); 11 printf("montante-0.02= %.16g\n", montante-0.02); 12 } </pre> <p>Execução do programa:</p> <pre> C:\>erro Indique um montante igual a 0.02: 0.02 montante diferente de 0.02 montante-0.02= -4.470348362317633e-010 </pre> <p>O teste <code>montante == 0.02</code> não deve ser feito, quanto muito deve-se efectuar duas desigualdades:</p> <pre>montante >= 0.01999999 && montante <= 0.02000001</pre> <p>No caso do exercício dos trocos, para ter em atenção o arredondamento, tem de se adicionar 0.5 valores da unidade que se pretende arredondar. Assim, se pretende cêntimos, antes de se efectuar os testes de desigualdades, deve-se somar 0.005 (uma vez que a unidade é o 0.01). Em alternativa pode-se converter o valor real para inteiros, portanto cêntimos, mas na conversão tem que se ter em atenção com o arredondamento, caso contrário o problema mantém-se, há o risco de 0.02×100 passar a 1 em inteiro.</p>
<p>1) primo.c não percebo a pergunta</p>	<p>Deve somar apenas os números divisores mínimos, não somando nada para o caso dos números primos. Não deve somar todos os números testados. No caso do exemplo do enunciado, a resposta seria 3, dado que do número 99 o divisor mínimo é o número 3, e o número 97 é primo. Atenção que um dos números na pergunta, apenas é divisível por um número superior a 100.</p>
<p>1) pi.c obtenho valores muito diferentes de π</p>	<p>Atenção que o factorial de 0 é 1. Note ainda que a fórmula calcula $1/\pi$</p>
<p>1) pi.c obtenho valores muito perto de π</p>	<p>Utilize o tipo <code>double</code> e a string de formatação <code>%.17g</code> para obter o valor com precisão 17 em notação científica.</p>
<p>2) rand.c por vezes tenho valores negativos</p>	<p>Para geração de números aleatórios devem-se utilizar inteiros positivos:</p> <pre>unsigned int v;</pre>

¹⁹ Espaço central, UC Programação, 2010/2011

<p>2) rand.c não percebo o resultado 33</p>	<p>Primeira parte da fórmula: $231533 \times 1282148053 + 82571 = 296859585237820$ No entanto o número tem de caber num inteiro de 4 bytes, que tem um total de hipóteses de: $2^{32} = 4294967296$ Portanto, o valor que fica guardado no inteiro de 4 bytes será: $296859585237820 \% 4294967296 = 35672892$ Falta agora efectuar a segunda parte da fórmula, o resto da divisão com m: $35672892 \% 428573 = 101333$ Finalmente, como o utilizador quer números entre 0 e 99, tem de efectuar o resto da divisão por 100: $101333 \% 100 = 33$ O valor de seed ficará com 101333, já que esta última operação do resto da divisão por 100 foi o utilizador da função que fez, mas a implementação da função rand apenas se preocupa com o resto da divisão por m, e retorna 101333.</p>
<p>2) find.c não obtenho os mesmos valores</p>	<p>Verifique se a função rand gera os mesmos valores aleatórios:</p> <pre> 1 #include <stdio.h> 2 3 int main() 4 { 5 int i; 6 srand(1); 7 for(i=0; i<10; i++) 8 printf("%d ", rand()); 9 } </pre> <p>Execução do programa: C:\> testerand 41 18467 6334 26500 19169 15724 11478 29358 26962 24464</p> <p>Se estes valores não coincidirem, então utilize a seguinte função para gerar valores aleatórios:</p> <pre> 1 unsigned int randaux() 2 { 3 static long seed=1; 4 return(((seed = seed * 214013L + 2531011L) >> 16) & 0x7fff); 5 } </pre>
<p>2) find.c é pedido a posição do primeiro 2, é o valor absoluto ou o dígito 2?</p>	<p>Se os números a saírem fossem estes: 13 15 678 32 45 26 2 Dever-se-ia retornar 6, e não a posição do 2 no número 32.</p>
<p>2) maximo.c no primeiro valor do exemplo dá-me 5.50 e não 5.47</p>	<p>Atenção à nota sobre a geração dos números, não deve gerar valores entre 0 e 9999, porque isso poderia levar a gerar o zero, e não lhe interessa calcular o logaritmo de zero. O primeiro valor gerado da função rand() é 41, para colocá-lo entre 0,0001 e 1,0000, temos de somar 1 (porque a função rand() pode devolver 0), ficando 42, dividir por 10000, ficando 0,0042, e depois chamar a função log (ln na calculadora). Ora $\ln(0.0042) = -5,47267\dots$, enquanto que $\ln(0.0041) = -5,4967$.</p>
<p>2) maximo.c obtenho os valores do exemplo correctos, mas a resposta dá-me incorrecta</p>	<p>Não se está a esquecer de fazer o resto da divisão por 10000? Está a gerar um vector com 1000 elementos (e não 10.000 elementos)? Pode ainda estar a fazer $\text{rand()} \% 9999 + 1$ em vez de $\text{rand()} \% 10000 + 1$. Se fizer o resto da divisão por 9999 não obtém os elementos distintos necessários, já que são precisos 10000 valores distintos (início em 0 e fim em 9999).</p>

2) mdc.c que pares utilizar, por exemplo, os pares (1,1), e (2,2) devem ser utilizados?	Não devem ser utilizados todos os "pares de números" distintos, nem os pares de números iguais: (1,1) e (2,2), etc. Relativamente ao números distintos significa que não devem ser utilizados os pares (1,2) e (2,1), apenas um. Conjunto de pares = {(1,2);(1,3);(1,4)...(1,100); (2,3);(2,4);(2,5)...(2,100); ... (97,98);(97,99);(97,100); (98,99);(98,100); (99,100)}
2) sort.c qual a relação entre ambos os vectores mostrados?	Os elementos são exactamente os mesmos, mas no <i>output</i> são mostrados apenas os 10 primeiros, embora o vector tenha 1000 elementos. O segundo vector está ordenado, daí mostrar elementos diferentes, que são os 10 mais baixos.
2) trocos2.c qual o significado do 20? O último montante ou o número de montantes?	É o montante maior possível, a iteração deve seguir a expressão indicada, até ao valor máximo de 20 euros. Não significa que o último elemento será exactamente 20, significa que o último elemento não será superior a 20.
2) jogodados.c obtenho os mesmos valores entre chamadas, com precisão a duas casas decimais	<code>srand(time(NULL))</code> ; deve ser chamado uma só vez, no início do programa. O gerador de números aleatórios não é tão aleatório assim, e normalmente existe correlação dos primeiros números gerados. Por outro lado a função <code>time</code> retorna o número de segundos passados desde uma determinada data, e entre chamadas pode retornar o mesmo valor, levando a que sejam gerados os mesmos números. Com a função <code>clock</code> seria melhor mas é a mesma coisa porque 1/1000 segundos para um computador é ainda assim tempo suficiente para muitas instruções.
2) jogodados.c não percebo a pergunta	Pretende-se o número de lançamentos, e não os seus valores, (ex: "6 7 8" e não "3.08 3.55 4.21").
2) strreplace.c a string pode aumentar de tamanho?	Sim, se a string a substituir for maior.
3) adiciona.c o 116 é o segundo elemento da lista?	O 116 é o segundo valor na lista, mas o penúltimo a ser adicionado.

VERIFICAÇÃO DAS RESPOSTAS

Para validar uma resposta de um exercício, verifique se a mesma está na seguinte lista:

'#'	'26 30 1 13 30 34 47 19 21'	'71 r bjk'
'%'	'26 30 1 13 30 34 48 19 20'	'714140004'
'&'	'26 30 1 14 30 35 48 19 22'	'714443004'
'('	'26 30 2 12 30 35 46 19 20'	'714443104'
')'	'26 30 2 13 30 35 49 19 21'	'72 r bjk'
'/'	'26 30 2 15 31 36 50 19 23'	'725'
'@'	'267'	'73712'
'['	'271'	'74 r bj'
']'	'28'	'74993'
'{'	'286'	'75 s bk'
'}'	'3 2 3 1'	'7500'
'0 160 241'	'3 5 4'	'77 r bk'
'0 160 41'	'3.14159265358979'	'784143104'
'0 2 3 0'	'3.1415926535897931'	'8 8'
'0 60 241'	'3.1415926535897962'	'8 9 8'
'1 1 3 6'	'3.15926535897'	'8273'
'1 160 41'	'3.15926535897931'	'83311'
'1 2 3'	'3000'	'83322'
'1 2 3 4'	'32'	'83333'
'1 3 4 6'	'3278'	'84'
'1 4 3 6'	'339'	'896'
'1 60 241'	'34'	'9'
'10400600'	'37'	'9 1 9'
'113670040'	'390'	'9 8 9'
'12'	'393'	'9 98'
'12444'	'4 4 3'	'91'
'126'	'4 6 3'	'924'
'128 554 902'	'40'	'94'
'13015'	'410001344'	'942'
'14361'	'420'	'95134'
'15 2 15 405 5 2 38 531'	'423500044'	'96200'
'15 2 15 415 5 2 18 533'	'423501344'	'98023'
'15 2 16 435 5 2 20 536'	'43'	'998.12321023'
'15 2 17 425 5 2 19 532'	'479001600'	'9998.12321234'
'15 2 51 405 5 2 28 530'	'488224456'	'99998.1132102111'
'15 2 51 405 5 2 8 537'	'5 2 18 118'	'99998.123210234'
'15231'	'5 3 linhas que alocação com'	'aaaa abc'
'155'	'5 3 linhas que com'	'abcd'
'16'	'5 3 no de nome'	'argumentos ABC 123 123 ABC'
'16442663'	'5 3 no linha comprimento de nome'	'argumentos ABC_123 123_ABC'
'165580141'	'5 4 linhas alocação com'	'argumentos ABC123 123ABC'
'169 464 242'	'5 4 linhas que alocação'	'argumentos ABC-123 123-ABC'
'17034'	'5 5 ecrã linha maximo de ficheiro'	'bbb abcd'
'172'	'5 7 4'	'ccc abc'
'176324603'	'50'	'cesar 21090 cifra'
'178'	'503 702 194'	'cifra 012210 cesar'
'18'	'507'	'cifracesar 21090'
'18 89'	'519 464 48'	'cifra-cesar21090'
'184'	'55'	'dcba'
'19 18'	'59'	'jkdsfdfsfd sfsd daqwrewfd
'19 62 48'	'59 462 248'	'sdafakjhds dfdsds
'2.718281828459'	'5995'	'dslxozjgkfsdcl'
'2.718281828459046'	'6000'	'prog 1 2 3'
'2.71828182845955'	'603255503'	'prog 123'
'2.8281828459'	'612'	'prog a b c'
'2.8281828459046'	'621'	'prog abc'
'20'	'6222446'	'r+b'
'222404595'	'63'	'r+t'
'225561900'	'68 s bk'	'r+w'
'23 25 12 25 29 40 17 19'	'69 62 48'	'r+wb'
'240 404 731'	'695'	'r+wt'
'240 503 731'	'698'	'rb'
'240 734 243'	'7 78'	'rstux'
'248 496 745'	'7 8 3'	'rt'
'249'	'7 8 9'	'rtxuv'
'249 404 902'	'7.82101'	'rw+b'
'249 503 734'	'7.82405'	'rw+t'
'249 503 734'	'7.82444'	'rwb'
'249 649 574'	'70'	'rwt'
'249 649 745'	'70 s bj'	'sduxjuckflj suxjucuxjucdfgklj
'249 764 731'	'70 s bjk'	'sdfli sdlfkuxjucj erlijs sdf
'25'	'700 404 243'	'ikxjuxjuculsj uxjuc xdvflkj
'25 29 13 29 34 47 18 21'	'702 764 419'	'wsefrlij wfsuxjucld dfg'
'25003'		'sfgha'
'258 496 574'		'ssdfd'
'258 649 740'		

RESOLUÇÕES DOS EXERCÍCIOS

Exercícios resolvidos, para comparação após resolver. Se não conseguir mesmo resolver um exercício com as dicas disponíveis, peça a alguém que o ajude, e que em último caso veja o código por si para lhe dar uma dica extra, mas não veja um exercício sem o resolver. O objectivo dos exercícios é colocá-lo perante a realização de actividades reais, para os quais o texto e exemplos fornecidos nos capítulos o ajudam a enfrentar. No entanto, apenas a realização de exercícios lhe poderá dar a experiência de programação que é apenas sua e intransmissível. Se não consegue resolver um exercício, deve ver essa situação como uma oportunidade de evolução e procurar eliminar a raiz do problema, e não desperdiçá-la vendo a resolução do exercício.

```

1 /* olamundosizeof.c */
2 int main()
3 {
4     /* texto com acentos: */
5     printf("\nOlá Mundo!");
6
7     /* tamanho em bytes dos diversos tipos */
8     printf("\nsizeof(char): %d", sizeof(char));
9     printf("\nsizeof(short): %d", sizeof(short));
10    printf("\nsizeof(int): %d", sizeof(int));
11    printf("\nsizeof(long): %d", sizeof(long));
12    printf("\nsizeof(long long): %d", sizeof(long long));
13    printf("\nsizeof(float): %d", sizeof(float));
14    printf("\nsizeof(double): %d", sizeof(double));
15    printf("\nsizeof(long double): %d", sizeof(long double));
16 }

```

```

1 /* soma.c */
2 int main()
3 {
4     int n, i, soma;
5
6     printf("Calculo da soma dos primeiros N numeros.\nIndique N:");
7     /* ler um número inteiro */
8     scanf("%d", &n);
9     /* na variável soma, será acumulado o resultado */
10    soma = 0;
11    /* a variável i vai iterar de 1 a N */
12    i = 1;
13    while(i <= n)
14    {
15        /* a variável soma vai acumulando o resultado de 1 até i */
16        soma = soma + i;
17        /* mostrar o resultado parcial */
18        printf("\n adicionar %d, parcial %d", i, soma);
19        /* incrementar a variável i */
20        i = i + 1;
21    }
22    /* mostrar resultado final */
23    printf("\nTotal: %d\n", soma);
24 }

```

```

1 /* hms.c */
2 int main()
3 {
4     int horas, minutos, segundos;
5     printf("Calculo do numero de segundos desde o inicio do dia.\nHora: ");
6     scanf("%d", &horas);
7     printf("Minuto: ");
8     scanf("%d", &minutos);
9     printf("segundos: ");
10    scanf("%d", &segundos);
11
12    printf("Numero de segundos desde o inicio do dia: %d",
13          segundos + 60 *(minutos + 60 *horas));
14 }

```

```

1 /* produto.c */
2 int main()
3 {
4     int n, i, produto;
5
6     printf("Calculo do produto dos primeiros N numeros.\nIndique N:");
7     scanf("%d", &n);
8
9     produto = 1;
10    i = 1;
11    while(i <= n)
12    {
13        /* utilização do operador *= em vez da atribuição e multiplicação */
14        produto *= i;
15        /* resultado parcial */
16        printf(" Factorial(%d)=%d\n", i, produto);
17        /* utilização do operador de incremento ++, em vez da atribuição

```

```

18     e soma com a unidade. */
19     i++;
20 }
21
22 printf("Resultado: %d", produto);
23 }

```

```

1  /* arranjos.c */
2  int main()
3  {
4      int i, n, r, arranjos;
5      printf("Calculo dos arranjos de N, R a R:\nIndique N:");
6      scanf("%d", &n);
7      printf("Indique R:");
8      scanf("%d", &r);
9
10     /* verificação da validade dos argumentos introduzidos */
11     if(n < r || r < 1)
12         printf("Erro: N tem de ser maior que R e este maior que 0.\n");
13     else
14     {
15         /* inicialização da variável iteradora é a expressão
16         indicada no enunciado */
17         i = n - r + 1;
18         arranjos = 1;
19         while(i <= n)
20         {
21             arranjos *= i;
22             /* parcial */
23             printf(" i=%d; arranjos=%d\n", i, arranjos);
24             i++;
25         }
26         printf("Resultado: %d\n", arranjos);
27     }
28 }

```

```

1  /* somadigitos.c */
2  int main()
3  {
4      /* variável soma declarada e inicializada no mesmo comando */
5      int n, soma = 0;
6      printf("Calculo da soma do quadrado dos digitos de um numero:\nNumero: ");
7      scanf("%d", &n);
8
9      while(n != 0)
10     {
11         /* extrair o dígito mais baixo com o resto da divisão
12         por 10 e somá-lo o seu quadrado */
13         soma += (n % 10) * (n % 10);
14         /* mostrar parcial */
15         printf(" n=%d; soma=%d\n", n, soma);
16
17         /* divisão inteira por 10, de forma a que o segundo
18         dígito mais baixo, passe a ser o mais baixo */
19         n /= 10;
20     }
21
22     printf("Resultado: %d\n", soma);
23 }

```

```

1  /* fibonacci.c */
2  int main()
3  {
4      /* é necessário duas variáveis auxiliares para guardar os
5      dois últimos valores */
6      int i, n, fib1, fib2, resultado;
7
8      printf("Calculo do valor da funcao Fibonacci:\nIndique N:");
9      scanf("%d", &n);
10
11     if(n < 1)
12         printf("Erro: n tem de ser maior ou igual a 1.\n");
13     else
14     {
15         /* caso n<=2, o resultado é n */
16         resultado = n;
17         if(n > 2)
18         {
19             /* os dois primeiros valores de fib é 1 e 2 */
20             fib1 = 1;
21             fib2 = 2;
22             /* começar no número 3, que é o primeiro cujo
23             resultado é a soma dos dois anteriores */
24             i = 3;
25             while(i <= n)
26             {
27                 /* após calcular o fib de i, actualizar as variáveis
28                 auxiliares com os dois valores anteriores, para o
29                 passo seguinte, o fib de i+1 */
30                 resultado = fib1 + fib2;
31                 fib1 = fib2;
32                 fib2 = resultado;
33                 /* mostrar valor actual */
34                 printf(" Fib(%d)=%d\n", i, resultado);
35                 i++;

```

```

36     }
37     }
38     printf("Resultado: %d\n", resultado);
39 }
40 }

1 /* combinacoes.c */
2 int main()
3 {
4     int i, n, r, combinacoes;
5     printf("Calculo das combinacoes de N, R a R:\nIndique N:");
6     scanf("%d", &n);
7     printf("Indique R:");
8     scanf("%d", &r);
9
10    if(n < r || r < 1)
11        printf("Erro: N tem de ser maior que R e este maior que 0.\n");
12    else
13    {
14        /* não efectuar as multiplicações e depois as divisões,
15         c.c. rapidamente se atinge o limite do inteiro, tem de
16         se multiplicar e dividir ao mesmo tempo */
17        i = 1;
18        combinacoes = 1;
19        while(i <= r)
20        {
21            /* mostrar valor actual */
22            printf(" %d", combinacoes);
23            /* a iteração é em R, o numerador tem de ser modificado
24             para ir de N-R+1 (quando i=1) até N (quando i=R) */
25            combinacoes *= (n - r + i);
26            /* mostrar valor após produto */
27            printf("%d=%d", n - r + i, combinacoes);
28            combinacoes /= i;
29            /* mostrar valor após divisão */
30            printf("/%d=%d\n", i, combinacoes);
31            i++;
32            /* Nota: é possível a divisão inteira, dado que quando i=2 já
33             dois números foram multiplicados, portanto um deles par,
34             quando i=3 já três números foram multiplicados, portanto um
35             deles multiplo de 3, e assim sucessivamente */
36        }
37        printf("Resultado: %d\n", combinacoes);
38    }
39 }

1 /* euler.c */
2 #define ITERACOES 20
3
4 int main()
5 {
6     int i = 0;
7     /* atendendo a que se pretende alguma precisão, utilizar reais
8     de precisão dupla: double
9     ao criar uma variável para o factorial do passo i,
10    evita-se ter de calcular de i em cada passo, basta multiplicar
11    a variável factorial por i */
12    double resultado = 0, factorial = 1;
13
14    while(i <= ITERACOES)
15    {
16        /* graças à variável factorial, o número de operações a realizar é
17         apenas um produto, uma divisão e uma soma por cada iteração */
18        if(i > 1)
19            factorial *= i;
20        resultado += 1 / factorial;
21        /* mostrar com precisão 16 em notação científica: %.16g */
22        printf(" %d: %.5g\n", i, resultado);
23        i++;
24    }
25    printf("Resultado: %.16g", resultado);
26 }

1 /* trocos.c */
2 int main()
3 {
4     float montante;
5     int count;
6     printf("Introduza um montante em euros, podendo ter centimos: ");
7     /* receber um número real do utilizador, com %f */
8     scanf("%f", &montante);
9     /* arredondar para o cêntimo mais próximo */
10    montante+=0.005;
11    /* contar o número de vezes que cabe a moeda maior no remanescente */
12    count=(int)(montante/2.);
13    montante-=2*count;
14    if(count>0)
15        printf("2 euros: %d\n", count);
16    /* efectuar com o montante restante o mesmo, para as restantes moedas */
17    count=(int)(montante);
18    montante-=count;
19    if(count>0)
20        printf("1 euro: %d\n", count);
21    count=(int)(montante/.5);
22    montante-=0.5*count;

```



```

23 if(count>0)
24     printf("50 centimos: %d\n",count);
25 count=(int)(montante/.2);
26 montante-=0.2*count;
27 if(count>0)
28     printf("20 centimos: %d\n",count);
29 count=(int)(montante/.1);
30 montante-=0.1*count;
31 if(count>0)
32     printf("10 centimos: %d\n",count);
33 count=(int)(montante/.05);
34 montante-=0.05*count;
35 if(count>0)
36     printf("5 centimos: %d\n",count);
37 count=(int)(montante/.02);
38 montante-=0.02*count;
39 if(count>0)
40     printf("2 centimos: %d\n",count);
41 count=(int)(montante/.01);
42 montante-=0.01*count;
43 if(count>0)
44     printf("1 centimo: %d\n",count);
45 /* o código acima é algo repetido, resultando desnecessariamente
46 em código muito extenso para a função implementada. No módulo
47 seguinte este exercício será pedido novamente, de forma a que
48 possa ser resolvido sem repetições, fazendo uso das as novas
49 ferramentas leccionadas. */
50 }

```

```

1  /* primo.c */
2  int main()
3  {
4      int divisor, n;
5
6      printf("Funcao que verifica se um numero N e' primo:\nIndique N:");
7      scanf("%d", & n);
8
9      if(n < 1)
10         printf("Erro: o numero tem de ser maior que zero.\n");
11     else
12     {
13         /* a variável divisor vai iterar até que o seu quadrado seja
14         maior que o número (evitando a utilização da raiz quadrada) */
15         divisor = 2;
16         while(divisor *divisor < n)
17         {
18             /* o número é divisível se o resto da divisão for nula */
19             if(n % divisor == 0)
20             {
21                 printf("\nNumero divisivel por %d\n", divisor);
22                 return;
23             }
24             /* mostrar interação */
25             printf("%d ", divisor);
26             divisor++;
27         }
28         printf("\nNumero primo!\n");
29     }
30 }

```

```

1  /* triplasoma.c */
2  int main()
3  {
4      int i, j, n, count;
5
6      printf("Escreva um numero para decompor em somas de tres parcelas.\nNumero:");
7      scanf("%d", & n);
8      count = 0;
9
10     /* percorrer todas as possíveis permutações (apenas dois ciclos aninhados),
11     assumindo que os números mais altos aparecem sempre primeiro */
12     i = n - 2;
13     while((n - i) <= i *2)
14     {
15         /* este condicional garante que há parcelas para além desta,
16         mais pequenas ou iguais, pelo menos duas iguais a i */
17         j = (n - i) - 1;
18         if(j > i)
19             j = i;
20         while((n - i - j) <= j)
21         {
22             /* encontrado i+j+(n-i-j), imprimir */
23             printf(" %d+%d+%d\n", i, j, n - i - j);
24             count++;
25             j--;
26         }
27         i--;
28     }
29     printf("Numero de somas: %d\n", count);
30 }

```

```

1  /* pi.c */
2  #define ITERACOES 3
3

```

```

4 int main()
5 {
6     int i = 0, j;
7     double resultado, factor = 0, parcela;
8
9     resultado = 2. *sqrt(2.) / 9801.;
10
11     while(i < ITERACOES)
12     {
13         parcela = 1103 + 26390 *i;
14         /* (4k)! */
15         j = 4 *i;
16         while(j > 0)
17         {
18             parcela *= j;
19             j--;
20         }
21         /* /(k!)^4 */
22         j = i;
23         while(j > 0)
24         {
25             parcela /= j *j *j *j;
26             j--;
27         }
28         /* /(396)^(4k) */
29         j = 4 *i;
30         while(j > 0)
31         {
32             parcela /= 396;
33             j--;
34         }
35         i++;
36         factor += parcela;
37     }
38     resultado *= factor;
39     resultado = 1 / resultado;
40     printf("\nvalor de PI (%d iteracoes): %.17g\n", ITERACOES, resultado);
41 }

```

```

1 /* formularesolvente.c */
2 /* listar os coeficientes de funções inteiras, entre -4 e 4,
3  que têm apenas raízes inteiras */
4
5 int main()
6 {
7     /* variáveis são agora reais, do tipo float */
8     float a, b, c, delta;
9     /* versão inteira para resposta à pergunta */
10    int ia, ib, ic, idelta, isqrt, k, count;
11    printf("Equacao do segundo grau a*x^2+b*x+c=0.\nIndique a b c: ");
12    /* leitura dos três argumentos em conjunto */
13    scanf("%f %f %f", &a, &b, &c);
14
15    /* calcular primeiramente o interior da raiz quadrada, dependente
16    do sinal deste valor, haverá número de raízes distintas */
17    delta = b *b - 4 *a *c;
18    /* mostrar parcial */
19    printf("Delta: %f\n", delta);
20    if(delta < 0)
21        printf("A equacao nao tem raizes reais.\n");
22    else if(delta == 0)
23        printf("A equacao tem uma raiz unica, x=%f\n", - b / (2 *a));
24    else
25        /* pode-se colocar logo o resto da expressão nos argumentos
26        do printf */
27        printf("o valor de x pode ser %f ou %f\n",
28        (- b + sqrt(delta)) / (2 *a), (- b - sqrt(delta)) / (2 *a));
29
30    /* resposta à pergunta */
31    printf("\nCalculo de coeficientes entre -K e K inteiros nao nulos, com raizes inteiras.");
32    printf("\nIntroduza K:");
33    scanf("%d", &k);
34
35    printf("\nCoeficientes de -%d a %d inteiros nao nulos, com raizes inteiras:\n",
36    k, k);
37    count = 0;
38    for(ia = - k; ia <= k; ia++)
39        for(ib = - k; ib <= k; ib++)
40            for(ic = - k; ic <= k; ic++)
41            {
42                if(ia == 0 || ib == 0 || ic == 0)
43                    continue;
44                idelta = ib *ib - 4 *ia *ic;
45                if(idelta == 0 && ia != 0 && ib % (2 *ia) == 0)
46                {
47                    count++;
48                    printf(" [%d %d %d]", ia, ib, ic);
49                }
50                else if(idelta > 0)
51                {
52                    /* verificar se é um quadrado perfeito */
53                    isqrt = sqrt(idelta);
54                    if(isqrt *isqrt == idelta && ia != 0 &&

```

```

55         ((- ib + isqrt) % (2 *ia) == 0 &&
56         (- ib - isqrt) % (2 *ia) == 0))
57     {
58         count++;
59         printf(" (%d %d %d)", ia, ib, ic);
60     }
61 }
62 }
63 printf("\nTotal: %d", count);
64 }

```

```

1  /* argumentos.c */
2  int main(int argc, char **argv)
3  {
4      /* argumentos da função main: argc - número de argumentos;
5       argv - vector de strings com os argumentos */
6      int i;
7      for(i = 0; i < argc; i++)
8          /* acesso à string na posição i do vector argv: argv[i] */
9          printf("Argumento %d: %s\n", i, argv[i]);
10 }

```

```

1  /* inverte.c */
2  /* definir o tamanho máximo de uma string em macros, para assim ser
3  fácil de alterar este valor sem ter de ver todo o código */
4  #define MAXSTR 255
5
6  /* funções sobre strings, mantendo a norma utilizada para os nomes
7  em string.h o nome da função fica com três letras str, seguido de
8  três letras indicando a operação, neste caso a inversão: strinv */
9  void strinv(char *str)
10 {
11     int len, i;
12     char aux;
13     len = strlen(str);
14     /* é necessário percorrer apenas metade do tamanho da string */
15     for(i = 0; i < len / 2; i++)
16     {
17         /* trocar este caracter pelo seu simétrico */
18         aux = str[i];
19         str[i] = str[len - 1 - i];
20         str[len - 1 - i] = aux;
21         /* mostrar a iteração */
22         printf(" %d: %s\n", i, str);
23     }
24 }
25
26 int main()
27 {
28     char str[MAXSTR];
29     printf("Inversao de um texto.\nTexto: ");
30     /* ler a string através da função gets */
31     gets(str);
32     /* inverter a string */
33     strinv(str);
34     /* após a execução da função anterior, a string ficou invertida */
35     printf("Texto invertido: %s", str);
36 }

```

```

1  /* rand.c */
2  /* as constantes utilizadas, podem ser definidas como macros,
3  de forma a aparecerem no início do programa. */
4  #define RAND_A 231533
5  #define RAND_B 82571
6  #define RAND_M 428573
7
8  /* variável global, com a semente/valor da última geração */
9  unsigned int seed = 0;
10
11 /* definição de uma função rand. Já existe uma função rand
12 nas bibliotecas, que pode ser utilizada nos exercícios seguintes. */
13 unsigned int rand()
14 {
15     seed = (RAND_A *seed + RAND_B) % RAND_M;
16     return seed;
17 }
18
19 int main()
20 {
21     int n, i;
22     printf("Gerador de numeros aleatorios inteiros.\nvalor maximo: ");
23     scanf("%d", &n);
24     n++;
25     /* inicializar a semente com o tempo */
26     seed = time(NULL);
27     /*seed=1;*/
28     printf("seed=%d: ", seed);
29     for(i = 0; i < 10; i++)
30         printf("%d ", rand() % n);
31     printf("\n");
32 }

```

```

1  /* find.c */
2  /* na declaração do vector v, não se deve colocar o número de
3  elementos, para funcionar para qualquer dimensão */
4  int Find(int v[], int n, int elemento)

```

```

5 {
6     int i;
7     for(i = 0; i < n; i++)
8         if(v[i] == elemento)
9             /* caso se encontre o valor correcto, retornar da
10              função mesmo com o ciclo a meio */
11             return i;
12     /* percorreu-se todos os elementos e não se encontrou
13     nenhum igual, retornar -1 */
14     return -1;
15 }
16
17 /* função que mostra no ecrã os primeiros elementos de um vector */
18 void PrintInts(int v[], int n, char *nome)
19 {
20     int i;
21     printf("%s", nome);
22     for(i = 0; i < n; i++)
23         printf("%d ", v[i]);
24 }
25
26 int main()
27 {
28     /* declaração do vector v com 1000 elementos (de 0 a 999) */
29     int v[1000], i;
30
31     /* fixar a semente a 1 */
32     srand(1);
33
34     /* inicializar o vector, com valores aleatórios de 0 a 999 */
35     for(i = 0; i < 1000; i++)
36         /* acesso à posição i do vector v: v[i] */
37         v[i] = rand() % 1000;
38
39     /* mostrar os 10 primeiros elementos */
40     PrintInts(v, 10, "Vector: ");
41
42     /* chamada da função find no próprio argumento do printf */
43     printf("\nPosicao de 2: %d", Find(v, 1000, 2));
44 }

```

```

1 /* maximo.c */
2 float Maximo(float v[], int n)
3 {
4     int i;
5     float resultado = 0;
6     for(i = 0; i < n; i++)
7         /* actualizar o resultado se for o primeiro elemento,
8          ou o valor registado for inferior ao elemento actual */
9         if(i == 0 || v[i] > resultado)
10         {
11             resultado = v[i];
12             /* mostrar iteração */
13             printf("\n %d: %.2f", i, resultado);
14         }
15     return resultado;
16 }

```

```

17 void PrintFloats(float v[], int n, char *nome)
18 {
19     int i;
20     printf("%s", nome);
21     for(i = 0; i < n; i++)
22         printf("%.2f ", v[i]);
23 }
24
25
26 int main()
27 {
28     int i;
29     float v[1000];
30
31     srand(1);
32
33     for(i = 0; i < 1000; i++)
34         v[i] = - log((1 + rand() % 10000) / 10000.);
35
36     PrintFloats(v, 10, "Vector: ");
37
38     printf("\nValor maximo: %.6g", Maximo(v, 1000));
39 }
40

```

```

1 /* mdc.c */
2 int mdc(int x, int y)
3 {
4     /* caso y=0, retornar x */
5     if(y == 0)
6         return x;
7     /* c.c. retornar mdc(y, mod(x,y)), exactamente o comando seguinte
8     após substituir o mod pelo comando do resto da divisão */
9     return mdc(y, x % y);
10 }
11
12 int main()

```

```

13 {
14     int x, y, i, j, soma;
15     printf("Calculo do maximo divisor comum entre dois numeros.\nIndique x e y:");
16     scanf("%d %d", &x, &y);
17     printf("MDC: %d\n", mdc(x, y));
18
19     /* calcular o valor da resposta pedido no exercício */
20     soma = 0;
21     for(i = 1; i < 100; i++)
22         for(j = i + 1; j <= 100; j++)
23             soma += mdc(i, j);
24     printf("A soma dos MDC dos pares de numeros de 1 a 100: %d\n", soma);
25 }

```

```

1 /* sort.c */
2 void Sort(int v[], int n)
3 {
4     /* duas variáveis iteradoras, já que se tem de percorrer todos
5     os pares */
6     int i, j, aux;
7     for(i = 0; i < n; i++)
8         /* como a ordem é indiferente, a segunda variável começa
9         logo com o valor acima da primeira, desta forma apenas é
10        analisado o par i=10, j=35, e nunca o par i=35, j=10, dado
11        que quando i=35, o j começa logo em 36 */
12        for(j = i + 1; j < n; j++)
13            if(v[i] > v[j])
14            {
15                /* pares pela ordem incorrecta, trocar recorrendo
16                a uma variável auxiliar */
17                aux = v[i];
18                v[i] = v[j];
19                v[j] = aux;
20            }
21 }
22
23 void PrintInts(int v[], int n, char *nome)
24 {
25     int i;
26     printf("%s", nome);
27     for(i = 0; i < n; i++)
28         printf("%d ", v[i]);
29 }
30
31 int main()
32 {
33     int v[1000], i;
34
35     srand(1);
36
37     for(i = 0; i < 1000; i++)
38         v[i] = rand() % 1000;
39
40     PrintInts(v, 10, "\nVector antes de ser ordenado: ");
41
42     Sort(v, 1000);
43
44     PrintInts(v, 10, "\nVector depois de ser ordenado: ");
45
46     printf("\nQuartil 25: %d\nQuartil 50: %d\nQuartil 75: %d\n",
47           v[250], v[500], v[750]);
48 }

```

```

1 /* removedups.c */
2 int RemoveDups(int v[], int n)
3 {
4     int i, j;
5     /* como o vector está ordenado, os duplicados estão seguidos,
6     pelo que é necessário saber se o elemento seguinte é igual */
7     for(i = 0; i < n - 1; i++)
8         /* enquanto o elemento seguinte for igual, apagá-lo */
9         while(i < n - 1 && v[i] == v[i + 1])
10        {
11            /* para apagar o elemento actual, tem de se copiar
12            todos os elementos seguintes uma posição para trás. */
13            for(j = i; j < n - 1; j++)
14                v[j] = v[j + 1];
15            /* não esquecer de actualizar a dimensão do vector */
16            n--;
17        }
18     return n;
19 }
20
21 /* Nota: numa versão mais eficiente, mas requerendo matéria do bloco
22 seguinte, poderia-se criar um vector novo e ir colocando os elementos
23 distintos nesse vector. Esse método faz uma só passagem pelo vector,
24 independentemente do número de elementos duplicados, enquanto que a
25 função acima percorre o resto do vector por cada elemento duplicado.
26 */
27 void Sort(int v[], int n)
28 {
29     int i, j, aux;
30     for(i = 0; i < n; i++)
31         for(j = i + 1; j < n; j++)

```

```

32         if(v[i] > v[j])
33         {
34             aux = v[i];
35             v[i] = v[j];
36             v[j] = aux;
37         }
38     }
39
40 void PrintInts(int v[], int n, char *nome)
41 {
42     int i;
43     printf("%s", nome);
44     for(i = 0; i < n; i++)
45         printf("%d ", v[i]);
46 }
47
48 int main()
49 {
50     int v[1000], i;
51
52     srand(1);
53
54     for(i = 0; i < 1000; i++)
55         v[i] = rand() % 1000;
56
57     PrintInts(v, 10, "\nVector inicial: ");
58
59     Sort(v, 1000);
60
61     printf("\nElementos que permanecem no vector: %d",
62           RemoveDups(v, 1000));
63
64     PrintInts(v, 10, "\nVector final: ");
65 }

```

```

1  /* somanumeros.c */
2  #define MAXSTR 255
3
4  int main()
5  {
6      char numeros[MAXSTR], *pt;
7      double soma = 0;
8
9      printf("Introduza um conjunto de numeros reais separados por espacos.\n");
10     gets(numeros);
11
12     /* pt fica a apontar para a primeira token (neste caso número real)*/
13     pt = (char *) strtok(numeros, " ");
14     while(pt != NULL)
15     {
16         /* converter a string para o seu valor numérico */
17         soma += atof(pt);
18         /* mostrar parcial */
19         printf(" %.2f (%s)\n", soma, pt);
20         /* passar para a próxima token */
21         pt = (char *) strtok(NULL, " ");
22     }
23     printf("A soma dos numeros e' %.15g.\n", soma);
24 }

```

```

1  /* trocos2.c */
2  int trocos(float montante, int mostrar)
3  {
4      int count, i, total;
5      /* vectores com informação estática */
6      float moedas[] =
7      {
8          2, 1, 0.5, 0.2, 0.1, 0.05, 0.02, 0.01, 0
9      };
10     char *smoedas[] =
11     {
12         "2 euros",
13         "1 euro",
14         "50 centimos",
15         "20 centimos",
16         "10 centimos",
17         "5 centimos",
18         "2 centimos",
19         "1 centimo"
20     };
21     total = 0;
22     montante += 0.005;
23     /* arredondar para o cêntimo mais próximo */
24     /* processar a informação estática, em vez de duplicar o código
25     utilizando constantes para cada possível valor do vector. */
26     for(i = 0; moedas[i] > 0; i++)
27     {
28         count = (int)(montante / moedas[i]);
29         montante -= moedas[i] * count;
30         if(count > 0 && mostrar)
31             printf("%s: %d\n", smoedas[i], count);
32         total += count;
33     }
34     return total;

```

```

35 }
36
37 int main()
38 {
39     float montante;
40     int total = 0;
41     printf("Introduza um montante em euros, podendo ter centimos: ");
42     scanf("%f", &montante);
43     trocos(montante, 1);
44     /* resposta à pergunta */
45     for(montante = 0.01; montante < 20; montante = montante * 2 + 0.01)
46         total += trocos(montante, 0);
47     printf("\nTotal moedas: %d", total);
48 }

```

```

1 /* jogodados.c */
2 #define ITERACOES 100000
3 #define MAXLANCAMENTOS 10
4
5 /* simulação de um jogo, dado um número fixo de lançamentos */
6 int JogoDados(int lancamentos, int mostraDado)
7 {
8     int dado,i,anterior=-1,pontos=0;
9     /* executar o número de lançamentos pedido, assumindo que o
10     dado anterior é -1, de forma a não dar derrota no primeiro
11     lançamento, nem ser necessário fazer um teste especial. */
12     for(i=0;i<lancamentos;i++) {
13         dado=1+rand()%6;
14         /* mostrar o dado */
15         if(mostraDado)
16             printf("%d ",dado);
17         pontos+=dado;
18         if(dado==anterior)
19             /* caso de derrota retorna logo, com a pontuação negativa */
20             return -pontos;
21         /* actualizar a variável anterior, com o valor correcto
22         para o próximo ciclo */
23         anterior=dado;
24     }
25     /* foi possível efectuar todos os lançamentos, retornar os
26     pontos positivos. */
27     return pontos;
28 }
29
30 int main()
31 {
32     int n,i,j,pontos;
33     srand(time(NULL));
34     printf("Jogo do lançamento de dados.\nIndique numero de lancamentos: ");
35     scanf("%d",&n);
36     printf("Pontos: %d\n\nvalores medios dos pontos:\n",JogoDados(n,1));
37     /* simulação de vários jogos */
38     for(i=1;i<MAXLANCAMENTOS;i++) {
39         /* contabilizar os pontos ao longo de todos os jogos*/
40         pontos=0;
41         for(j=0;j<ITERACOES;j++)
42             pontos+=JogoDados(i,0);
43         /* fornecer a pontuação média */
44         printf("Com %d lançamento(s): %.2f\n",
45             i,(float)pontos/ITERACOES);
46     }
47 }

```

```

1 /* baralhar.c */
2 void Baralhar(int v[], int n)
3 {
4     int i, j, aux;
5     /* processar todos os elementos */
6     for(i = 0; i < n - 1; i++)
7     {
8         /* gerar um valor aleatório para sortear o elemento do
9         vector a ficar na posição i (entre i e n-1). */
10         j = i + rand() % (n - i);
11         aux = v[i];
12         v[i] = v[j];
13         v[j] = aux;
14     }
15 }
16
17 void PrintInts(int v[], int n, char *nome)
18 {
19     int i;
20     printf("%s", nome);
21     for(i = 0; i < n; i++)
22         printf("%d ", v[i]);
23 }
24
25 int main()
26 {
27     int v[1000], i;
28     srand(1);
29     /* desta vez inicializar o vector com a função identidade */
30     for(i = 0; i < 1000; i++)
31         v[i] = i;

```

```

32 PrintInts(v, 10, "Vector identidade: ");
33
34 Baralhar(v, 1000);
35
36 PrintInts(v, 10, "\nVector baralhado: ");
37
38 printf("\nNa posicao 250, 500, 750: %d %d %d",
39 v[250], v[500], v[750]);
40
41 }

```

```

1 /* mergersort.c */
2 #define MAXVECTOR 1000000
3
4 /* função auxiliar para juntar duas partes do vector ordenadas */
5 void MergeSort2(int v[], int a, int meio, int b)
6 {
7     /* juntar ambas as metades no vector auxiliar */
8     static int aux[MAXVECTOR];
9     int i, j, k;
10    /* i primeira parte do vector,
11       j segunda parte,
12       k vector ordenado */
13    i = a;
14    j = meio;
15    k = a;
16    /* k avança de a a b */
17    while(k < b)
18        /* vector auxiliar fica com o menor dos valores, a não
19           ser que uma das partes já tenha acabado, e ao mesmo tempo
20           incrementa as respectivas variáveis iteradoras */
21        if(i < meio && (j >= b || v[i] <= v[j]))
22            aux[k++] = v[i++];
23        else
24            aux[k++] = v[j++];
25
26    /* copiar o vector auxiliar para o vector em ordenação */
27    for(k = a; k < b; k++)
28        v[k] = aux[k];
29
30    /* mostrar indicação que já está ordenado o vector de a a b */
31    if(b - a > 100000)
32        printf(" Ordenado de %d a %d\n", a, b - 1);
33 }
34
35 /* função auxiliar para ordenar de a até b */
36 void MergeSort1(int v[], int a, int b)
37 {
38     if(b - a < 2)
39         /* um ou zero elementos para ordenar, retornar */
40         return;
41
42     /* dividir em dois grupos, ordenar separadamente */
43     MergeSort1(v, a, (a + b) / 2);
44     MergeSort1(v, (a + b) / 2, b);
45
46     /* juntar os dois grupos ordenados */
47     MergeSort2(v, a, (a + b) / 2, b);
48 }
49
50 void MergeSort(int v[], int n)
51 {
52     MergeSort1(v, 0, n);
53 }
54
55
56 int main()
57 {
58     int i;
59     /* keyword static força a que esta variável seja na realidade global,
60        e mantem os valores entre chamadas da função. Neste caso é a função
61        main, que não será chamada segunda vez, mas a keyword static permite
62        que o compilador utilize outra zona de memória que não o stack. */
63     static int v[MAXVECTOR];
64     srand(1);
65     for(i = 0; i < MAXVECTOR; i++)
66         v[i] = rand() % 1000;
67     MergeSort(v, MAXVECTOR);
68     printf("Quantis: %d %d %d\n", v[250000], v[500000], v[750000]);
69 }

```

```

1 /* binarysearch.c */
2 /* função auxiliar, procura num intervalo de a a b (inclusivé) */
3 int BinarySearch1(int v[], int a, int b, int elemento)
4 {
5     /* mostrar informação */
6     printf(" Procurar %d entre %d e %d\n", elemento, a, b);
7
8     /* caso a seja já maior que b, retornar fracasso */
9     if(a > b)
10         return - 1;
11
12     /* testar o elemento do meio, e se for igual, retornar esta posição */
13     if(v[(a + b) / 2] == elemento)
14         return(a + b) / 2;

```



```

14  /* caso seja menor, chamar recursivamente esta função para o segmento
15  da esquerda, c.c. chamar para o segmento da direita. */
16  if(v[(a + b) / 2] < elemento)
17      return BinarySearch1(v, (a + b) / 2 + 1, b, elemento);
18  /* notar que não há necessidade de else nos ifs, dado que se está
19  a retornar logo da função dentro dos ifs */
20  return BinarySearch1(v, a, (a + b) / 2 - 1, elemento);
21 }
22
23 int BinarySearch(int v[], int n, int elemento)
24 {
25     return BinarySearch1(v, 0, n - 1, elemento);
26 }
27
28 #define MAXVECTOR 1000000
29
30 void MergeSort2(int v[], int a, int meio, int b)
31 {
32     static int aux[MAXVECTOR];
33     int i, j, k;
34     i = a;
35     j = meio;
36     k = a;
37     while(k < b)
38         if(i < meio && (j >= b || v[i] <= v[j]))
39             aux[k++] = v[i++];
40         else
41             aux[k++] = v[j++];
42     for(k = a; k < b; k++)
43         v[k] = aux[k];
44 }
45
46 void MergeSort1(int v[], int a, int b)
47 {
48     if(b - a < 2)
49         return;
50     MergeSort1(v, a, (a + b) / 2);
51     MergeSort1(v, (a + b) / 2, b);
52     MergeSort2(v, a, (a + b) / 2, b);
53 }
54
55 void MergeSort(int v[], int n)
56 {
57     MergeSort1(v, 0, n);
58 }
59
60 int main()
61 {
62     int i;
63     static int v[MAXVECTOR];
64     srand(1);
65     for(i = 0; i < MAXVECTOR; i++)
66         v[i] = rand() % 1000;
67     MergeSort(v, MAXVECTOR);
68
69     printf("Posicao de 250, 500 e 750: %d %d %d\n",
70           BinarySearch(v, MAXVECTOR, 250),
71           BinarySearch(v, MAXVECTOR, 500),
72           BinarySearch(v, MAXVECTOR, 750));
73 }

```

```

1  /* numeracaoromana.c */
2  #define MAXROMANA 24
3
4  /* Este exercício tem também a questão da repetição de código do exercício
5  dos trocos */
6  void ArabeParaRomana(int arabe, char romana[], int mostrar)
7  {
8      static int valor[] =
9      {
10         1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1, 0
11     };
12     static char *texto[] =
13     {
14         "M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I", ""
15     };
16     int i;
17     strcpy(romana, "");
18     /* processar todos os valores, do maior para o menor, e ir
19     concatenando as respectivas strings, idêntico ao exercício dos trocos */
20     for(i = 0; valor[i] > 0; i++)
21         while(arabe >= valor[i])
22         {
23             strcat(romana, texto[i]);
24             arabe -= valor[i];
25             /* mostrar esta operação */
26             if(mostrar)
27                 printf(" %d %s\n", arabe, romana);
28         }
29 }
30
31 int RomanaParaArabe(char romana[], int mostrar)
32 {

```

```

33  /* estas variáveis estáticas estão declaradas em duas funções,
34  poderiam ser globais, mas como são de interesse apenas para duas
35  funções optou-se pela sua duplicação localmente. */
36  static int valor[] =
37  {
38      1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1, 0
39  };
40  static char *texto[] =
41  {
42      "M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I", ""
43  };
44  int arabe = 0, i;
45  char *pt;
46  pt = romana;
47  while(pt[0] != 0)
48  for(i = 0; valor[i] > 0; i++)
49  /* tentar encontrar na string as letras romanas (1 e 2 caracteres) */
50  if(texto[i][1] == 0 && pt[0] == texto[i][0])
51  {
52      /* 1 caracter, match perfeito */
53      pt++;
54      arabe += valor[i];
55      /* mostrar esta operação */
56      if(mostrar)
57          printf(" %d %s\n", arabe, pt);
58      /* o valor árabe é atualizado, avançar para o próximo
59      caracter e voltar a procurar por match com as letras romanas*/
60      break;
61  }
62  else if(texto[i][1] != 0 &&
63  pt[0] == texto[i][0] &&
64  pt[1] == texto[i][1])
65  {
66      /* 2 caracteres, match perfeito */
67      pt += 2;
68      arabe += valor[i];
69      /* mostrar esta operação */
70      if(mostrar)
71          printf(" %d %s\n", arabe, pt);
72      /* idêntico, mas avança dois caracteres */
73      break;
74  }
75  return arabe;
76  }
77
78  int main()
79  {
80      int numero, i, soma;
81      char romana[MAXROMANA];
82      printf("Conversao de numeracao arabe para romana e vice-versa.\n");
83      printf("Numero arabe: ");
84      gets(romana);
85      numero = atoi(romana);
86      ArabeParaRomana(numero, romana, 1);
87      printf("Resultado: %s\nNumero romano: ", romana);
88      gets(romana);
89      printf("Resultado: %d\n", RomanaParaArabe(romana, 1));
90      /* resposta à pergunta */
91      soma = 0;
92      for(i = 1; i < 1000; i++)
93      {
94          ArabeParaRomana(i, romana, 0);
95          soma += strlen(romana);
96      }
97      printf("Caracteres dos numeros de 1 a 1000, em numeracao romana: %d",
98      soma);
99  }

```

```

1  /* pokerdados.c */
2  void PrintInts(int v[], int n, char *nome)
3  {
4      int i;
5      printf("%s", nome);
6      for(i = 0; i < n; i++)
7          printf("%d ", v[i]);
8  }
9
10 char *poker5dados(int dados[5], int mostra)
11 {
12     int count[6], i, j, max;
13     /* contar quantos dados existem de cada tipo */
14     max = 0;
15     for(i = 0; i < 6; i++)
16         count[i] = 0;
17     for(i = 0; i < 5; i++)
18     {
19         count[dados[i] - 1]++;
20         if(max < count[dados[i] - 1])
21             max = count[dados[i] - 1];
22     }
23     /* mostra contagem */
24     if(mostra)
25         PrintInts(count, 6, "\n Frequencia dos valores: ");

```

```

26  /* verificar primeiro as situações de maior valor */
27  if(max == 5)
28      return "poker real";
29  if(max == 4)
30      return "poker quadruplo";
31  if(max == 3)
32  {
33      /* se existir um número com contagem 2, então temos um fullen */
34      for(i = 0; i < 6; i++)
35          if(count[i] == 2)
36              return "fullen";
37  }
38  if(max == 1)
39  {
40      /* verificar a sequência */
41      if(count[0] == 0 || count[5] == 0)
42          return "sequencia";
43  }
44  if(max == 3)
45      /* agora a única hipótese é o trio */
46      return "trio";
47  if(max == 2)
48  {
49      /* se existir dois números com contagem 2, temos dois pares */
50      for(i = 0, j = 0; i < 6; i++)
51          if(count[i] == 2)
52              j++;
53      if(j == 2)
54          return "duplo par";
55      else
56          return "par";
57  }
58  return "nada";
59 }
60 }
61
62 int main()
63 {
64     int dados[5], i, j, nada, par, duplopar;
65     char reter[80], *resposta;
66     /* simulação de um jogo, lançando 5 dados */
67     srand(time(NULL));
68     for(i = 0; i < 5; i++)
69         dados[i] = 1 + rand() % 6;
70
71     PrintInts(dados, 5, "Valor dos dados lançados: ");
72     printf("(%)s\n", reter (10100 reter o primeiro e o dado central): ",
73     poker5dados(dados, 1));
74     gets(reter);
75     /* lançar os dados não retidos, e apresentar resultado final */
76     for(i = 0; i < 5 && reter[i]; i++)
77         if(reter[i] != '1')
78             dados[i] = 1 + rand() % 6;
79     PrintInts(dados, 5, "Valor dos dados finais: ");
80     printf("(%)s\n", poker5dados(dados, 1));
81     /* código para responder à pergunta */
82     srand(1);
83     nada = par = duplopar = 0;
84     for(i = 0; i < 1000; i++)
85     {
86         for(j = 0; j < 5; j++)
87             dados[j] = 1 + rand() % 6;
88         resposta = poker5dados(dados, 0);
89         if(strcmp(resposta, "nada") == 0)
90             nada++;
91         else if(strcmp(resposta, "par") == 0)
92             par++;
93         else if(strcmp(resposta, "duplo par") == 0)
94             duplopar++;
95     }
96     printf("nada: %d\npar: %d\nduplo par: %d\n", nada, par, duplopar);
97 }

```

```

1  /* strreplace.c */
2  #define MAXSTR 1024
3
4  int strreplace(char *str, char *find, char *replace)
5  {
6      char *pt;
7      int szfind, szreplace, i, count = 0;
8
9      szfind = strlen(find);
10
11     if(szfind == 0)
12         return;
13
14     szreplace = strlen(replace);
15
16     pt = str;
17     while((pt = strstr(pt, find)) != NULL)
18     {
19         count++;
20         /* substituir find por replace */

```

```

21     if(szfind >= szreplace)
22     {
23         /* há espaço */
24         for(i = 0; i < szreplace; i++)
25             pt[i] = replace[i];
26         if(szfind > szreplace)
27         {
28             /* reposicionar o resto da string */
29             for(i = szreplace; pt[i + szfind - szreplace]; i++)
30                 pt[i] = pt[i + szfind - szreplace];
31             pt[i] = 0;
32         }
33     }
34     else
35     {
36         /* não há espaço, reposicionar o resto da string */
37         for(i = strlen(pt); i > 0; i--)
38             pt[i - szfind + szreplace] = pt[i];
39
40         /* copiar o replace */
41         for(i = 0; i < szreplace; i++)
42             pt[i] = replace[i];
43     }
44     /* é importante que pt avance o tamanho do replace, para
45     não efectuar substituições na própria substituição */
46     pt += szreplace;
47     /* mostrar operação */
48     printf(" %s\n", str);
49 }
50 return count;
51 }
52
53 int main()
54 {
55     char str[MAXSTR], find[MAXSTR], replace[MAXSTR];
56     int count;
57     printf("Substituir num texto, as ocorrencias de uma string A \
58     por uma string B.\nTexto: ");
59     gets(str);
60     printf("String A: ");
61     gets(find);
62     printf("String B: ");
63     gets(replace);
64
65     count = strreplace(str, find, replace);
66     printf("Resultado (%d trocas): %s\n", count, str);
67 }

```

```

1  /* ndamas.c */
2  #define MAXDAMAS 16
3
4  int total;
5
6  void MostraDamas(int linhasDamas[MAXDAMAS], int n)
7  {
8      int i, j;
9      static int mostrar = 1;
10     /* contar as posições e mostrar apenas a primeira vez */
11     total++;
12     if(mostrar == 0)
13         return;
14     /* como a variável mostrar é estática, da próxima vez
15     que esta função for chamada será 0, e esta parte do código
16     não será executada */
17     mostrar = 0;
18     /* barra horizontal superior */
19     printf("\n+");
20     for(i = 0; i < n; i++)
21         printf("---");
22     printf("+");
23     for(i = 0; i < n; i++)
24     {
25         /* barra vertical esquerda */
26         printf("\n|");
27         /* casas vazias à esquerda da dama desta linha */
28         for(j = 0; j < linhasDamas[i]; j++)
29             printf(" ");
30         /* dama nesta linha */
31         printf("# ");
32         /* casas vazias à direita da dama desta linha */
33         for(j = linhasDamas[i] + 1; j < n; j++)
34             printf(" ");
35         /* barra vertical direita */
36         printf("|");
37     }
38     /*barra horizontal inferior */
39     printf("\n+");
40     for(i = 0; i < n; i++)
41         printf("---");
42     printf("+");
43 }
44
45 /* com base nas linhas das damas já colocadas, colocar uma dama em coluna */

```

```

46 void ColocarDama(int linhasDamas[MAXDAMAS], int coluna, int n)
47 {
48     int i, j;
49     /* percorrer todas as linhas para tentar colocar a dama */
50     for(i = 0; i < n; i++)
51     {
52         for(j = 0; j < coluna; j++)
53             if(linhasDamas[j] == i ||
54                /* mesma linha */
55                linhasDamas[j] + coluna - j == i ||
56                /* mesma diagonal */
57                linhasDamas[j] + j - coluna == i)
58                 break;
59         /* se não houve uma paragem é porque esta linha i é possível */
60         if(j == coluna)
61         {
62             linhasDamas[coluna] = i;
63             /* mostrar as damas se esta é a última coluna,
64             c.c. chamar recursivamente esta função para a coluna seguinte */
65             if(coluna + 1 == n)
66                 MostraDamas(linhasDamas, n);
67             else
68                 ColocarDama(linhasDamas, coluna + 1, n);
69         }
70         /* notar que mesmo após ter colocado uma dama na linha i,
71         o algoritmo continua a explorar o i+1, e prossegue até n-1,
72         explorando todas as hipóteses possíveis. */
73     }
74 }
75
76 int main()
77 {
78     int n;
79     int linhasDamas[MAXDAMAS];
80     printf("Conta quantas posicoes existem, num tabuleiro de NxN, ");
81     printf("para colocacao de N damas sem que estas se ataquem ");
82     printf("mutuamente.\nIndique N (maximo %d): ", MAXDAMAS);
83     scanf("%d", &n);
84     total = 0;
85     ColocarDama(linhasDamas, 0, n);
86     printf("\nTotal: %d\n", total);
87 }

```

```

1  /* doisdados.c */
2  #define ITERACOES 10000
3  #define MAXELEMENTOS 100
4  #define MAXVALORES 22
5
6  /* mostra um gráfico com tantas colunas quantos os elementos,
7  e 16 linhas sendo os valores re-escalados. Retorna o número
8  de cardinais impressos (para responder à pergunta) */
9  int GraficoDeBarras(int serie[], int n)
10 {
11     char grafico[MAXVALORES][MAXELEMENTOS];
12     int i, j, maximo, count = 0;
13     float factor;
14     if(n > MAXELEMENTOS)
15         return;
16     /* determinar valor máximo da série */
17     for(i = 0; i < n; i++)
18         if(i == 0 || maximo < serie[i])
19             maximo = serie[i];
20     /* obter o factor para re-escalar os valores para 16 linhas */
21     factor = 15. / maximo;
22     /* desenhar os eixos do gráfico */
23     sprintf(grafico[0], "");
24     sprintf(grafico[1], "  +");
25     /* barra horizontal */
26     for(i = 0; i < n; i++)
27         strcat(grafico[1], "-");
28     /* barra vertical */
29     for(i = 2; i < 18; i++)
30         sprintf(grafico[i], "%5d|*s ", (int)((i - 2) / factor + 0.5), n,
31                 "");
32     /* desenhar a série para o array */
33     for(i = 0; i < n; i++)
34         for(j = 0; j < 16; j++)
35             if(serie[i] * factor >= j)
36             {
37                 grafico[j + 2][i + 6] = '#';
38                 count++;
39             }
40     /* imprimir todo o array (aqui considerado um vector de strings) */
41     for(i = 17; i >= 0; i--)
42         printf("\n%s", grafico[i]);
43     return count;
44 }
45
46 void PrintInts(int v[], int n, char *nome)
47 {
48     int i;
49     printf("%s", nome);
50     for(i = 0; i < n; i++)

```

```

51     printf("%d ", v[i]);
52 }
53
54 int main()
55 {
56     int soma[11], i, count;
57     srand(1);
58     printf("simulacao de %d lancamento de dois dados:\n", ITERACOES);
59     for(i = 0; i < 11; i++)
60         soma[i] = 0;
61     for(i = 0; i < ITERACOES; i++)
62         soma[(1 + rand() % 6) + (1 + rand() % 6) - 2]++;
63     count = GraficoDeBarras(soma, 11);
64     PrintInts(soma, 11, "23456789p0p0\n\nvalores: ");
65     printf("\nnúmero de # presentes no histograma: %d", count);
66 }
67
68 /* calendario.c */
69 /* teste de ano bissexto */
70 int bissexto(int ano)
71 {
72     if(ano % 400 == 0)
73         return 1;
74     if(ano % 100 == 0)
75         return 0;
76     if(ano % 4 == 0)
77         return 1;
78     return 0;
79 }
80
81 /* retorna o número de dias do mês, a contar com os anos bissextos */
82 int DiasDoMes(int ano, int mes)
83 {
84     static int diasDoMes[] =
85     {
86         31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
87     };
88     if(mes == 2)
89         return diasDoMes[1] + bissexto(ano);
90     else if(mes >= 1 && mes <= 12)
91         return diasDoMes[mes - 1];
92     return 0;
93 }
94
95 /* retorna a diferença entre duas datas*/
96 int DiasDeDiferenca(int ano1, int mes1, int dia1, int ano2, int mes2, int
97 dia2)
98 {
99     int dias = 0, i;
100
101     /* somar os dias de diferença por anos */
102     for(i = ano1; i < ano2; i++)
103         dias += 365 + bissexto(i);
104
105     /* subtrair os dias desde o início do ano até mes1/dia1 */
106     for(i = 0; i < mes1 - 1; i++)
107         dias -= DiasDoMes(ano1, i + 1);
108     dias -= dia1 - 1;
109
110     /* somar os dias desde o início do ano até mes2/dia2 */
111     for(i = 0; i < mes2 - 1; i++)
112         dias += DiasDoMes(ano2, i + 1);
113     dias += dia2 - 1;
114
115     return dias;
116 }
117
118 /* retorna o dia da semana de uma data */
119 int DiaDaSemana(int ano, int mes, int dia)
120 {
121     /* ver os dias de diferença entre um dia conhecido, e fazer o
122     resto da divisão por 7 */
123     return(6 + DiasDeDiferenca(2000, 1, 1, ano, mes, dia)) % 7;
124 }
125
126 /* determinação do mês/dia da Páscoa */
127 void Pascoa(int ano, int *mes, int *dia)
128 {
129     /* código de acordo com o algoritmo */
130     int a, b, c, d, e, f, g, h, i, k, l, M;
131     a = ano % 19;
132     b = ano / 100;
133     c = ano % 100;
134     d = b / 4;
135     e = b % 4;
136     f = (b + 8) / 25;
137     g = (b - f + 1) / 3;
138     h = (19 * a + b - d - g + 15) % 30;
139     i = c / 4;
140     k = c % 4;
141     l = (32 + 2 * e + 2 * i - h - k) % 7;
142     M = (a + 11 * h + 22 * l) / 451;
143     *mes = (h + l - 7 * M + 114) / 31;

```

```

77     *dia = ((h + 1 - 7 * M + 114) % 31) + 1;
78 }
79
80 /* esta função adiciona/subtrai dias apenas no mesmo ano */
81 void AdicionarDias(int *ano, int *mes, int *dia, int ndias)
82 {
83     /* se há mais dias a subtrair que o mês, então subtrair
84     um mês inteiro */
85     while(ndias > 0 && ndias > DiasDoMes(*ano, *mes) -*dia)
86     {
87         ndias -= DiasDoMes(*ano, *mes);
88         (*mes)++;
89     }
90     /* situação inversa */
91     while(ndias < 0 && -ndias > *dia - 1)
92     {
93         ndias += DiasDoMes(*ano, *mes - 1);
94         (*mes)--;
95     }
96     /* já há dias suficientes, subtrair apenas os dias*/
97     *dia += ndias;
98 }
99
100 /* coloca os dias de feriado para um dado ano/mês */
101 void Feriados(int ano, int mes, int dias[32])
102 {
103     int i, pmes, pdia, fmes, fdia;
104     static int deltaPascoa[] =
105     {
106         - 48, - 47, - 2, + 60
107     };
108     static int mesFeriadoFixo[] =
109     {
110         1, 4, 5, 6, 10, 12, 12
111     };
112     static int diaFeriadoFixo[] =
113     {
114         1, 25, 1, 10, 5, 1, 25
115     };
116
117     for(i = 0; i < 32; i++)
118         dias[i] = 0;
119     Pascoa(ano, & pmes, & pdia);
120     if(pmes == mes)
121         dias[pdia] = 1;
122
123     for(i = 0; i < 4; i++)
124     {
125         fmes = pmes;
126         fdia = pdia;
127         AdicionarDias(& ano, & fmes, & fdia, deltaPascoa[i]);
128         if(fmes == mes)
129             dias[fdia] = 1;
130     }
131
132     for(i = 0; i < 7; i++)
133         if(mesFeriadoFixo[i] == mes)
134             dias[diaFeriadoFixo[i]] = 1;
135 }
136
137 /* imprime um mês formatadamente */
138 void PrintMes(int ano, int mes)
139 {
140     static char *nomesDosMeses[] =
141     {
142         "Janeiro", "Fevereiro", "Marco", "Abril", "Maio", "Junho",
143         "Julho", "Agosto", "Setembro", "Outubro", "Novembro", "Dezembro"
144     };
145     int diasDoMes, i = 1, j;
146     int dias[32];
147     Feriados(ano, mes, dias);
148     diasDoMes = DiasDoMes(ano, mes);
149     printf("\n%s de %d:", nomesDosMeses[mes - 1], ano);
150     printf("\n#####");
151     printf("\n# D S T Q Q S S #");
152     printf("\n#####");
153
154     /* processar todos os dias, de semana a semana */
155     while(diasDoMes > 0)
156     {
157         /* início de uma nova semana */
158         printf("\n#");
159         j = DiaDaSemana(ano, mes, i);
160         /* na primeira semana pode não começar logo no domingo */
161         if(j > 0)
162             printf("%*s", 3 * j, " ");
163         do
164         {
165             if(dias[i] == 1)
166                 printf(" F ");
167             else
168                 printf("%2d ", i);

```

```

169         i++;
170         diasDoMes--;
171         j = DiaDaSemana(ano, mes, i);
172     }
173     while(j > 0 && diasDoMes > 0);
174     /* se a semana acabou e não foi ao sábado
175     (última semana), completar */
176     if(j > 0)
177         printf("%s", 3 * (7 - j), " ");
178     printf("#");
179 }
180
181 printf("\n#####");
182 }
183
184 int main()
185 {
186     int ano, mes, resultado, i;
187     int dias[32];
188     printf("Calendario mensal, com indicacao de feriados.\nIndique ano mes:");
189     scanf("%d %d", &ano, &mes);
190     PrintMes(ano, mes);
191     /* código para resposta à pergunta */
192     resultado = 0;
193     for(ano = 2000; ano < 2100; ano++)
194     {
195         /* feriados */
196         Feriados(ano, 2, dias);
197         for(i = 0; i < 32; i++)
198             resultado += dias[i];
199         /* dias do mês */
200         resultado += DiasDoMes(ano, 2);
201         /* dia da semana do dia 1 */
202         resultado += DiaDaSemana(ano, 2, 1);
203     }
204     printf("\nResposta: %d", resultado);
205 }

```

```

1 /* editdistance.c */
2 #define MAXSTR 256
3
4 void PrintInts(int v[], int n, char *nome)
5 {
6     int i;
7     printf("%s", nome);
8     for(i = 0; i < n; i++)
9         printf("%d ", v[i]);
10 }
11
12 void MostraMatriz(int[MAXSTR][MAXSTR], char *, char *, int, int);
13 void MostraOperacoes(int[MAXSTR][MAXSTR], char *, char *, int, int);
14
15 /* ver http://en.wikipedia.org/wiki/Levenshtein_distance */
16 int EditDistance(char *s1, char *s2)
17 {
18     int szs1, szs2, i, j, aux;
19     /* vector bidimensional, para suporte ao algoritmo fornecido */
20     int m[MAXSTR][MAXSTR];
21     /* calcular o tamanho das strings */
22     szs1 = strlen(s1);
23     szs2 = strlen(s2);
24     /* caso fora dos limites, retornar */
25     if(szs1 >= MAXSTR || szs2 >= MAXSTR)
26         return -1;
27     /* inicializações */
28     for(i = 0; i <= szs1; i++)
29         m[i][0] = i;
30     for(j = 0; j <= szs2; j++)
31         m[0][j] = j;
32     /* calcular a matriz por ordem, até szs1, szs2
33     isto pode ser feito uma vez que a expressão fornecida
34     calcula i,j com base em índices inferiores, portanto
35     já calculados. */
36     for(i = 1; i <= szs1; i++)
37         for(j = 1; j <= szs2; j++)
38             if(s1[i - 1] == s2[j - 1])
39                 m[i][j] = m[i - 1][j - 1];
40             else
41             {
42                 aux = m[i - 1][j] + 1; // apagar
43                 if(m[i][j - 1] + 1 < aux)
44                     // inserir
45                     aux = m[i][j - 1] + 1;
46                 if(m[i - 1][j - 1] + 1 < aux)
47                     // substituir
48                     aux = m[i - 1][j - 1] + 1;
49                 m[i][j] = aux;
50             }
51     MostraMatriz(m, s1, s2, szs1, szs2);
52     MostraOperacoes(m, s1, s2, szs1, szs2);
53     /* retornar a distância de edição, na última posição do vector */
54     return m[szs1][szs2];
55 }

```



```

56 }
57
58 void MostraMatriz(
59 int m[MAXSTR][MAXSTR],
60 char *s1,
61 char *s2,
62 int szs1,
63 int szs2)
64 {
65     int i;
66     char str[MAXSTR];
67
68     /* mostrar a matriz de base à comparação */
69     printf(" Matriz de distancias:\n");
70     for(i = 0; i < szs2; i++)
71         printf("%c ", s2[i]);
72     for(i = 0; i <= szs1; i++)
73     {
74         if(i > 0)
75             sprintf(str, "\n %c ", s1[i - 1]);
76         else
77             strcpy(str, "\n ");
78         PrintInts(m[i], szs2 + 1, str);
79     }
80 }
81
82 void MostraOperacoes(
83 int m[MAXSTR][MAXSTR],
84 char *s1,
85 char *s2,
86 int szs1,
87 int szs2)
88 {
89     int i, j, k, aux, apagar, substituir, inserir;
90     char str[MAXSTR];
91     printf("\n Operacoes:");
92     strcpy(str, s1);
93     apagar = substituir = inserir = 0;
94     for(i = szs1, j = szs2; i > 0 || j > 0;)
95     {
96         printf("\n %s ", str);
97         if(i > 0 && j > 0 && s1[i - 1] == s2[j - 1])
98         {
99             printf("(%) %s", s1[i - 1], str);
100             i--;
101             j--;
102         }
103         else
104         {
105             if(i > 0)
106                 aux = m[i - 1][j]; // apagar
107             else
108                 aux = m[i][j];
109             if(j > 0 && m[i][j - 1] < aux)
110                 // inserir
111                 aux = m[i][j - 1];
112             if(j > 0 && i > 0 && m[i - 1][j - 1] < aux)
113                 // substituir
114                 aux = m[i - 1][j - 1];
115
116             if(i > 0 && aux == m[i - 1][j])
117                 // apagar
118             {
119                 i--;
120                 k = i - 1;
121                 while(str[k++] != '\0')
122                     str[k] = str[k + 1];
123                 printf("[-%c] %s", s1[i], str);
124                 apagar++;
125             }
126             else if(j > 0 && aux == m[i][j - 1])
127                 /* inserir */
128             {
129                 j--;
130                 k = i;
131                 while(str[k++] != '\0')
132                     while(k > i)
133                     {
134                         str[k] = str[k - 1];
135                         k--;
136                     }
137                 str[i] = s2[j];
138                 printf("[+%c] %s", s2[j], str);
139                 inserir++;
140             }
141             else if(i > 0 && j > 0)
142             {
143                 /* substituir */
144                 str[--i] = s2[--j];
145                 printf("[%c/%c] %s", s1[i], s2[j], str);
146                 substituir++;
147             }
148         }
149     }

```

```

148         else
149             break;
150     }
151 }
152 printf("\nApagar: %d, Inserir: %d, Substituir: %d\n",
153        apagar,
154        inserir,
155        substituir);
156 }
157
158 int main()
159 {
160     char s1[MAXSTR], s2[MAXSTR];
161     printf("Calculo da distancia de edicao entre duas strings.\nIndique s1: ");
162     gets(s1);
163     printf("Indique s2: ");
164     gets(s2);
165     printf("Resultado: %d", EditDistance(s1, s2));
166 }

```

```

1 /* adiciona.c */
2 /* declaração da estrutura SLista, com typedef para Lista */
3 typedef struct SLista
4 {
5     /* variáveis pertencentes à estrutura */
6     int valor;
7     /* apontador para o elemento seguinte da lista,
8     ou NULL se não existirem mais elementos */
9     struct SLista *seg;
10 }
11 Lista;
12
13 /* função adiciona um elemento ao topo da lista, retornando
14 o novo início da lista */
15 Lista *Adiciona(Lista *lista, int valor)
16 {
17     /* alocar espaço para um novo elemento */
18     Lista *elemento = (Lista *) malloc(sizeof(Lista));
19     if(elemento != NULL)
20     {
21         /* atribuir o valor do novo elemento */
22         elemento->valor = valor;
23         /* o elemento seguinte é o actual primeiro elemento da lista */
24         elemento->seg = lista;
25     }
26     /* em caso de falha, retorna NULL, c.c. retorna o novo início
27 da lista, que é o elemento criado */
28     return elemento;
29 }
30
31 /* remover o primeiro elemento da lista, retornando para o segundo,
32 agora primeiro elemento da lista */
33 Lista *Remove(Lista *lista)
34 {
35     Lista *aux;
36     if(lista != NULL)
37     {
38         /* guarda o elemento a retornar */
39         aux = lista->seg;
40         /* liberta o espaço alocado por este elemento */
41         free(lista);
42         /* retornar o segundo, agora primeiro elemento da lista */
43         return aux;
44     }
45     return NULL;
46 }
47
48 /* contar o número de elementos na lista */
49 int Elementos(Lista *lista)
50 {
51     int count = 0;
52     /* pode-se utilizar o próprio parâmetro para iterar pela lista */
53     while(lista != NULL)
54     {
55         lista = lista->seg;
56         count++;
57     }
58     return count;
59 }
60
61 /* versão de PrintInts para listas */
62 void PrintListaInts(Lista *lista, int n, char *nome)
63 {
64     printf("%s", nome);
65     while(lista != NULL && n-->0)
66     {
67         printf("%d ", lista->valor);
68         lista = lista->seg;
69     }
70 }
71
72 int main()
73 {

```

```

74     int i;
75     Lista *lista = NULL;
76     srand(1);
77     /* adicionar 1000 elementos à lista */
78     for(i = 0; i < 1000; i++)
79         lista = Adiciona(lista, rand() % 1000);
80     /* retornar os valores pedidos */
81     printf("Numero de elementos: %d\n",
82           Elementos(lista));
83     PrintListaInts(lista, 10, "Elementos: ");
84     /* libertar toda a lista antes de sair */
85     while(lista != NULL)
86         lista = Remove(lista);
87 }

```

```

1  /* more.c */
2  #define MAXSTR 1024
3
4  int main(int argc, char **argv)
5  {
6      FILE *f;
7      char str[MAXSTR], *pt, c;
8      int count = 0;
9
10     /* verificar se foi dado um argumento */
11     if(argc <= 1)
12     {
13         printf("Utilização: more <ficheiro>\n");
14         return;
15     }
16
17     /* abrir o ficheiro em modo de texto */
18     f = fopen(argv[1], "rt");
19     if(f == NULL)
20     {
21         printf("Ficheiro %s não existe.\n", argv[1]);
22         return;
23     }
24
25     while(! feof(f))
26     {
27         count++;
28         /* contador da linha */
29         if(fgets(str, MAXSTR, f) != NULL)
30         {
31             /* caso a string tenha menos de 75 caracteres,
32              arrumar a questão */
33             if(strlen(str) <= 75)
34                 printf("%4d:%s", count, str);
35             else
36             {
37                 /* strings com mais de uma linha, utilizar um apontador */
38                 pt = str;
39                 while(strlen(pt) > 75)
40                 {
41                     /* finalizar a string para ter 75 caracteres */
42                     c = pt[75];
43                     pt[75] = 0;
44                     /* na primeira linha, colocar o número da linha
45                     do ficheiro */
46                     if(pt == str)
47                         printf("%4d:%s", count, pt);
48                     else
49                         printf("      %s", pt);
50                     /* repôr o caracter retirado ao cortar a string */
51                     pt[75] = c;
52                     /* avançar 75 caracteres */
53                     pt += 75;
54                 }
55                 /* colocar o resto da string */
56                 printf("      %s", pt);
57             }
58         }
59     }
60 }

```

```

1  /* insere.c */
2  typedef struct SLista
3  {
4      int valor;
5      struct SLista *seg;
6  } Lista;
7
8  Lista *Adiciona(Lista *lista, int valor)
9  {
10     Lista *elemento = (Lista *) malloc(sizeof(Lista));
11     if(elemento != NULL)
12     {
13         elemento->valor = valor;
14         elemento->seg = lista;
15     }
16     return elemento;
17 }
18 }

```

```

19
20 Lista *Remove(Lista *lista)
21 {
22     Lista *aux;
23     if(lista != NULL)
24     {
25         aux = lista->seg;
26         free(lista);
27         return aux;
28     }
29     return NULL;
30 }
31
32 int Elementos(Lista *lista)
33 {
34     int count = 0;
35     while(lista != NULL)
36     {
37         lista = lista->seg;
38         count++;
39     }
40     return count;
41 }
42
43 Lista *Insere(Lista *lista, int valor)
44 {
45     /* caso a lista esteja vazia, ou
46     o primeiro elemento é igual ou maior,
47     adicionar o elemento no início */
48     if(lista == NULL || lista->valor >= valor)
49         return Adiciona(lista, valor);
50
51     /* caso a lista não seja vazia e o primeiro
52     elemento é menor, então inserir a partir
53     do segundo elemento */
54     lista->seg = Insere(lista->seg, valor);
55
56     /* tendo sido inserido o elemento a partir
57     do segundo elemento, o valor da lista
58     não se altera */
59     return lista;
60 }
61
62 Lista *Apaga(Lista *lista, int valor)
63 {
64     /* apagar todos os elementos iniciais iguais ao valor */
65     while(lista != NULL && lista->valor == valor)
66         lista = Remove(lista);
67
68     /* se a lista ficou vazia, retornar */
69     if(lista == NULL)
70         return NULL;
71
72     /* repetir para os elementos seguintes */
73     lista->seg = Apaga(lista->seg, valor);
74
75     return lista;
76 }
77
78 void PrintListaInts(Lista *lista, int n, char *nome)
79 {
80     printf("%s", nome);
81     while(lista != NULL && n-->0)
82     {
83         printf("%d ", lista->valor);
84         lista = lista->seg;
85     }
86 }
87
88 int main()
89 {
90     int i;
91     Lista *lista = NULL;
92     srand(1);
93     for(i = 0; i < 1000; i++)
94         lista = Insere(lista, rand() % 1000);
95     PrintListaInts(lista, 10, "Elementos apos inserir 1000 elementos: ");
96     for(i = 0; i < 1000; i += 2)
97         lista = Apaga(lista, i);
98     printf("\nNumero de elementos na lista apos apagar: %d\n",
99     Elementos(lista));
100     PrintListaInts(lista, 10, "Elementos apos apagar: ");
101     while(lista)
102         lista = Remove(lista);
103 }
104
105 /* insertsort.c */
106 typedef struct SLista
107 {
108     int valor;
109     struct SLista *seg;
110 }
111 Lista;

```

```

8
9 void PrintListaInts(Lista *lista, int n, char *nome)
10 {
11     printf("%s", nome);
12     while(lista != NULL && n-->0)
13     {
14         printf("%d ", lista->valor);
15         lista = lista->seg;
16     }
17 }
18
19 Lista *Adiciona(Lista *lista, int valor)
20 {
21     Lista *elemento = (Lista *) malloc(sizeof(Lista));
22     if(elemento != NULL)
23     {
24         elemento->valor = valor;
25         elemento->seg = lista;
26     }
27     return elemento;
28 }
29
30 Lista *Remove(Lista *lista)
31 {
32     Lista *aux;
33     if(lista != NULL)
34     {
35         aux = lista->seg;
36         free(lista);
37         return aux;
38     }
39     return NULL;
40 }
41
42 Lista *Insere(Lista *lista, int valor)
43 {
44     if(lista == NULL || lista->valor >= valor)
45         return Adiciona(lista, valor);
46     lista->seg = Insere(lista->seg, valor);
47     return lista;
48 }
49
50 /* ordenar a lista inserindo os elementos de forma ordenada */
51 Lista *InsertSort(Lista *lista)
52 {
53     int count = 0;
54     /* criar uma lista vazia onde se vai colocando os elementos */
55     Lista *aux = NULL;
56     while(lista != NULL)
57     {
58         /* inserir o novo elemento */
59         aux = Insere(aux, lista->valor);
60         /* apagar o elemento da lista antiga, e avançar para o próximo */
61         lista = Remove(lista);
62         /* mostrar este passo durante as primeiras iterações */
63         if(count++ < 10)
64             PrintListaInts(aux, 10, "\n Lista: ");
65     }
66     /* retornar a nova lista, a antiga já não existe */
67     return aux;
68 }
69
70
71 int main()
72 {
73     int i;
74     Lista *lista = NULL;
75     srand(1);
76     for(i = 0; i < 1000; i++)
77         lista = Adiciona(lista, rand() % 1000);
78     PrintListaInts(lista, 10, "Lista nao ordenada: ");
79     lista = InsertSort(lista);
80     PrintListaInts(lista, 10, "\nLista ordenada: ");
81     while(lista != NULL)
82         lista = Remove(lista);
83 }
84
85 /* enesimo.c */
86 typedef struct SLista
87 {
88     int valor;
89     struct SLista *seg;
90 } Lista;
91
92 Lista *Adiciona(Lista *lista, int valor)
93 {
94     Lista *elemento = (Lista *) malloc(sizeof(Lista));
95     if(elemento != NULL)
96     {
97         elemento->valor = valor;
98         elemento->seg = lista;
99     }
100 }

```

```

17     return elemento;
18 }
19
20 Lista *Remove(Lista *lista)
21 {
22     Lista *aux;
23     if(lista != NULL)
24     {
25         aux = lista->seg;
26         free(lista);
27         return aux;
28     }
29     return NULL;
30 }
31
32 Lista *Insere(Lista *lista, int valor)
33 {
34     if(lista == NULL || lista->valor >= valor)
35         return Adiciona(lista, valor);
36     lista->seg = Insere(lista->seg, valor);
37     return lista;
38 }
39
40 Lista *InsertSort(Lista *lista)
41 {
42     Lista *aux = NULL;
43     while(lista != NULL)
44     {
45         aux = Insere(aux, lista->valor);
46         lista = Remove(lista);
47     }
48     return aux;
49 }
50
51 /* retorna o valor em uma determinada posição, ou 0 se não existirem
52    elementos suficientes */
53 int Valor(Lista *lista, int posicao)
54 {
55     /* avançar na lista até atingir a posição especificada */
56     while(posicao > 1 && lista != NULL)
57     {
58         posicao--;
59         lista = lista->seg;
60     }
61
62     /* se a lista não acabou antes de se obter a posição pretendida,
63        retornar o valor actual */
64     if(lista != NULL)
65         return lista->valor;
66
67     return 0;
68 }
69
70 int main()
71 {
72     int i;
73     Lista *lista = NULL;
74     srand(1);
75     for(i = 0; i < 1000; i++)
76         lista = Adiciona(lista, rand() % 1000);
77     lista = InsertSort(lista);
78     printf("elementos 250, 500 e 750: %d %d %d",
79         Valor(lista, 250),
80         Valor(lista, 500),
81         Valor(lista, 750));
82     while(lista != NULL)
83         lista = Remove(lista);
84 }

```

```

1  /* palavras.c */
2  #define MAXSTR 1024
3
4  /* função que dada uma string retorna o número de palavras */
5  int Palavras(char *str)
6  {
7      char strBackup[MAXSTR];
8      char *pt = str;
9      int count = 0;
10     strcpy(strBackup, str);
11     pt = strtok(pt, " \t\n\r");
12     while(pt != NULL)
13     {
14         count++;
15         /* é uma palavra só com letras à partida */
16         while(pt[0] != 0)
17             if(! isalpha(pt[0]))
18             {
19                 /* afinal a palavra não tem apenas letras */
20                 count--;
21                 break;
22             }
23         else pt++;
24     pt = strtok(NULL, " \t\n\r");

```

```

25     }
26     /* mostrar resultado se existirem palavras */
27     if(count > 0)
28         printf(" [%d] %s", count, strBackup);
29     return count;
30 }
31
32
33 int main(int argc, char **argv)
34 {
35     FILE *f;
36     char str[MAXSTR], *pt, c;
37     int count = 0;
38
39     setlocale(LC_ALL, "");
40
41     /* verificar se foi dado um argumento */
42     if(argc <= 1)
43     {
44         printf("Utilização: palavras <ficheiro>\n");
45         return;
46     }
47
48     /* abrir o ficheiro em modo de texto */
49     f = fopen(argv[1], "rt");
50     if(f == NULL)
51     {
52         printf("Ficheiro %s não existe.\n", argv[1]);
53         return;
54     }
55
56     while(! feof(f))
57         if(fgets(str, MAXSTR, f) != NULL)
58             count += Palavras(str);
59
60     printf("\no ficheiro %s contem %d palavras.\n", argv[1], count);
61 }

```

```

1  /* cifracesar.c */
2  #define MAXSTR 255
3
4  void CifraDeCesar(char *str, int chave)
5  {
6      int i, chaved;
7      chaved = chave;
8      /* chave negativa, somar a diferença até ser positiva */
9      while(chave < 0)
10         chave += 'z' - 'a' + 1;
11     while(chaved < 0)
12         chaved += '9' - '0' + 1;
13     /* processar todos os caracteres */
14     for(i = 0; str[i] != 0; i++)
15         if(str[i] >= 'a' && str[i] <= 'z')
16             /* novo valor será: a + (k-a + chave mod z-a+1) */
17             str[i] = 'a' + (str[i] - 'a' + chaved) % ('z' - 'a' + 1);
18         else if(str[i] >= 'A' && str[i] <= 'Z')
19             str[i] = 'A' + (str[i] - 'A' + chaved) % ('Z' - 'A' + 1);
20         else if(str[i] >= '0' && str[i] <= '9')
21             str[i] = '0' + (str[i] - '0' + chaved) % ('9' - '0' + 1);
22     }
23
24
25 int main(int argc, char **argv)
26 {
27     FILE *f;
28     int chave;
29     char str[MAXSTR];
30
31     if(argc != 3)
32     {
33         printf("Indicar o nome do ficheiro e chave.\n");
34         return;
35     }
36
37     chave = atoi(argv[2]);
38
39     f = fopen(argv[1], "rt");
40     if(f == NULL)
41     {
42         printf("Não foi possível abrir para leitura o ficheiro %s.\n",
43             argv[1]);
44         return;
45     }
46
47     while(fgets(str, MAXSTR, f) != NULL)
48     {
49         CifraDeCesar(str, chave);
50         printf("%s", str);
51     }
52     fclose(f);
53 }

```

```

1  /* more.c */
2  #define MAXSTR 1024

```

```

3
4 int main(int argc, char **argv)
5 {
6     FILE *f;
7     char *pt;
8     int i, j, tamanho;
9
10    /* verificar se foi dado um argumento */
11    if(argc <= 1)
12    {
13        printf("Utilizacao: more <ficheiro>\n");
14        return;
15    }
16
17    /* abrir o ficheiro em modo de texto */
18    f = fopen(argv[1], "rb");
19    if(f == NULL)
20    {
21        printf("Ficheiro %s nao existe.\n", argv[1]);
22        return;
23    }
24
25    fseek(f, 0, SEEK_END);
26    tamanho = ftell(f);
27
28    pt = (char *) malloc(tamanho);
29    if(pt == NULL)
30        return;
31
32    fseek(f, 0, SEEK_SET);
33    fread(pt, tamanho, 1, f);
34    fclose(f);
35
36    /* processar todos os caracteres do ficheiro */
37    for(i = 0; i < tamanho; i += 16)
38    {
39        printf("\n%6d", i);
40        /* colocar os caracteres em hexadecimal */
41        for(j = 0; j < 16 && i + j < tamanho; j++)
42            printf("%02x ", (unsigned char) pt[i + j]);
43        /* colocar espaços no final da linha */
44        while(j++ < 16)
45            printf(" ");
46        /* colocar os caracteres em texto, se possível */
47        printf("|");
48        for(j = 0; j < 16 && i + j < tamanho; j++)
49            if(isprint(pt[i + j]))
50                printf("%c", pt[i + j]);
51            else
52                printf(" ");
53    }
54    free(pt);
55 }

```

```

1 /* basededados.c */
2 #define MAXSTR 255
3
4 /* nome do ficheiro da BD fixo, para não ter de o estar a introduzir
5  ao entrar na aplicação */
6 #define NOME_FICHEIRO "basededados.dat"
7
8 /* estrutura definida com os campos necessários para cada registo */
9 typedef struct
10 {
11     int id;
12     char nome[MAXSTR];
13     char telefone[MAXSTR];
14     char cidade[MAXSTR];
15     char descricao[MAXSTR];
16 }
17 SContacto;
18
19 /* função para gravar este novo registo num no ficheiro */
20 void Gravar(SContacto *registo)
21 {
22     FILE *f = fopen(NOME_FICHEIRO, "r+b");
23     /* se não abrir, tentar criar o ficheiro de raiz */
24     if(f == NULL)
25         f = fopen(NOME_FICHEIRO, "w+b");
26     if(f != NULL)
27     {
28         /* posicionar o ficheiro na posição do registo */
29         fseek(f, (long)(registo->id) * sizeof(SContacto), SEEK_SET);
30         /* gravar sobrepondo ou gravando um novo registo */
31         fwrite(registo, sizeof(SContacto), 1, f);
32         fclose(f);
33     }
34     else
35         printf("\nErro: nao foi possível abrir o ficheiro.");
36 }
37
38 /* ler um registo do ficheiro */
39 void Ler(SContacto *registo)

```



```

40 {
41     FILE *f = fopen(NOME_FICHEIRO, "rb");
42     if(f != NULL)
43     {
44         /* posiciona e lê o registo com base no id */
45         fseek(f, (long)(registo->id) * sizeof(SContacto), SEEK_SET);
46         fread(registo, sizeof(SContacto), 1, f);
47         fclose(f);
48     }
49     else
50         printf("\nErro: nao foi possivel abrir o ficheiro.");
51 }
52
53 /* retorna o número de registos, obtido através do ficheiro */
54 int Registos()
55 {
56     int n;
57     FILE *f = fopen(NOME_FICHEIRO, "rb");
58     if(f != NULL)
59     {
60         fseek(f, 0, SEEK_END);
61         n = ftell(f);
62         fclose(f);
63         return n / sizeof(SContacto);
64     }
65     return 0;
66 }
67
68 /* pedir ao utilizador para introduzir os dados de um registo */
69 void Carregar(SContacto *registo)
70 {
71     printf("Nome: ");
72     gets(registo->nome);
73     printf("Telefone: ");
74     gets(registo->telefone);
75     printf("Cidade: ");
76     gets(registo->cidade);
77     printf("Descrição: ");
78     gets(registo->descricao);
79 }
80
81 int Menu()
82 {
83     char str[MAXSTR];
84     printf("\n#####");
85     printf("\n# Menu: #");
86     printf("\n#####");
87     printf("\n# 1 | LISTAR registos #");
88     printf("\n# 2 | ADICIONAR registo #");
89     printf("\n# 3 | VER registo #");
90     printf("\n# 4 | EDITAR registo #");
91     printf("\n#####");
92     printf("\nOpção: ");
93     gets(str);
94     return atoi(str);
95 }
96
97 void ListarRegistos()
98 {
99     int i, total;
100     SContacto registo;
101     total = Registos();
102     printf("\n#####");
103     printf("\n# Lista: ", total);
104     printf("\n#####");
105     for(i = 0; i < total; i++)
106     {
107         registo.id = i;
108         Ler(& registo);
109         printf("\n# %4d | %s", registo.id, registo.nome);
110     }
111     printf("\n#####\n");
112 }
113
114 void VerRegisto()
115 {
116     char str[MAXSTR];
117     SContacto registo;
118     printf("ID: ");
119     gets(str);
120     registo.id = atoi(str);
121     Ler(& registo);
122     printf("\n#####");
123     printf("\n# Nome | %s", registo.nome);
124     printf("\n# Telefone | %s", registo.telefone);
125     printf("\n# Cidade | %s", registo.cidade);
126     printf("\n# Descrição | %s", registo.descricao);
127     printf("\n#####\n");
128 }
129
130 int main()
131 {

```

```

132 int opcao, i;
133 char str[MAXSTR];
134 SContato registro;
135
136 while(1)
137 {
138     opcao = Menu();
139
140     if(opcao == 1)
141     {
142         ListarRegistros();
143     }
144     else if(opcao == 2)
145     {
146         /* adicionar registro */
147         registro.id = Registros();
148         Carregar(& registro);
149         Gravar(& registro);
150     }
151     else if(opcao == 3)
152     {
153         VerRegistro();
154     }
155     else if(opcao == 4)
156     {
157         /* editar registro */
158         printf("ID: ");
159         gets(str);
160         registro.id = atoi(str);
161         Carregar(& registro);
162         Gravar(& registro);
163     }
164     else
165         break;
166 }
167 }

```

```

1 /* mergesort.c */
2 typedef struct SLista
3 {
4     int valor;
5     struct SLista *seg;
6 }
7 Lista;
8
9 Lista *Adiciona(Lista *lista, int valor)
10 {
11     Lista *elemento = (Lista *) malloc(sizeof(Lista));
12     if(elemento != NULL)
13     {
14         elemento->valor = valor;
15         elemento->seg = lista;
16     }
17     return elemento;
18 }
19
20 Lista *Remove(Lista *lista)
21 {
22     Lista *aux;
23     if(lista != NULL)
24     {
25         aux = lista->seg;
26         free(lista);
27         return aux;
28     }
29     return NULL;
30 }
31
32 Lista *MergeSort(Lista *lista)
33 {
34     Lista *lista1 = NULL, *lista2 = NULL;
35
36     /* se a lista tiver apenas um elemento, retornar */
37     if(lista == NULL || lista->seg == NULL)
38         return lista;
39
40     /* dividir a lista a meio */
41     while(lista != NULL)
42     {
43         lista1 = Adiciona(lista1, lista->valor);
44         lista = Remove(lista);
45         /* elementos pares para a lista 2 */
46         if(lista != NULL)
47         {
48             lista2 = Adiciona(lista2, lista->valor);
49             lista = Remove(lista);
50         }
51     }
52     /* neste momento já não há lista original, apenas lista1 e lista2 */
53
54     /* ordenar ambas as metades */
55     lista1 = MergeSort(lista1);
56     lista2 = MergeSort(lista2);

```

```

57
58 /* juntar ambas as listas ordenadas */
59 while(lista1 != NULL && lista2 != NULL)
60 {
61     if(lista1->valor < lista2->valor)
62     {
63         lista = Adiciona(lista, lista1->valor);
64         lista1 = Remove(lista1);
65     }
66     else
67     {
68         lista = Adiciona(lista, lista2->valor);
69         lista2 = Remove(lista2);
70     }
71 }
72
73 /* adicionar os restantes elementos */
74 if(lista1 == NULL)
75     lista1 = lista2;
76 while(lista1 != NULL)
77 {
78     lista = Adiciona(lista, lista1->valor);
79     lista1 = Remove(lista1);
80 }
81
82 /* inverter a lista, já que os elementos mais baixos
83 são colocados primeiro, mas ficam em último */
84 while(lista != NULL)
85 {
86     lista1 = Adiciona(lista1, lista->valor);
87     lista = Remove(lista);
88 }
89
90 return lista1;
91 }
92
93 int main()
94 {
95     int i;
96     Lista *lista = NULL;
97     srand(1);
98     for(i = 0; i < 100000; i++)
99         lista = Adiciona(lista, rand() % 1000);
100 lista = MergeSort(lista);
101 /* remover os primeiros 999 elementos, para aceder à posição 1000 */
102 for(i = 0; i < 999; i++)
103     lista = Remove(lista);
104 printf("Elemento na posição 1000: %d\n", lista->valor);
105 /* remover os primeiros 9000 elementos, para aceder à posição 10000 */
106 for(i = 0; i < 9000; i++)
107     lista = Remove(lista);
108 printf("Elemento na posição 10000: %d", lista->valor);
109 while(lista != NULL)
110     lista = Remove(lista);
111 }

```

```

1 /* sort.c */
2 #define MAXSTR 1024
3
4 typedef struct SListaSTR
5 {
6     char *str;
7     struct SListaSTR *seg;
8 }
9 ListaSTR;
10
11 ListaSTR *Adiciona(ListaSTR *lista, char *str)
12 {
13     ListaSTR *elemento = (ListaSTR *) malloc(sizeof(ListaSTR));
14     if(elemento != NULL)
15     {
16         /* alocar espaço para a string, e copiar a string */
17         elemento->str = (char *) malloc(strlen(str) + 1);
18         strcpy(elemento->str, str);
19         elemento->seg = lista;
20     }
21     return elemento;
22 }
23
24 ListaSTR *Remove(ListaSTR *lista)
25 {
26     ListaSTR *aux;
27     if(lista != NULL)
28     {
29         aux = lista->seg;
30         /* libertar não só a estrutura como a string */
31         free(lista->str);
32         free(lista);
33         return aux;
34     }
35     return NULL;
36 }
37

```

```

38 ListaSTR *MergeSort(ListaSTR *lista)
39 {
40     ListaSTR *lista1 = NULL, *lista2 = NULL;
41
42     if(lista == NULL || lista->seg == NULL)
43         return lista;
44
45     while(lista != NULL)
46     {
47         lista1 = Adiciona(lista1, lista->str);
48         lista = Remove(lista);
49         if(lista != NULL)
50         {
51             lista2 = Adiciona(lista2, lista->str);
52             lista = Remove(lista);
53         }
54     }
55
56     lista1 = MergeSort(lista1);
57     lista2 = MergeSort(lista2);
58
59     /* tudo igual até aqui, excepto utilizar str em vez de valor.
60     Agora, ao juntar ambas as listas ordenadas, utilização de strcmp */
61     while(lista1 != NULL && lista2 != NULL)
62     {
63         if(strcmp(lista1->str, lista2->str) < 0)
64         {
65             lista = Adiciona(lista, lista1->str);
66             lista1 = Remove(lista1);
67         }
68         else
69         {
70             lista = Adiciona(lista, lista2->str);
71             lista2 = Remove(lista2);
72         }
73     }
74
75     if(lista1 == NULL)
76         lista1 = lista2;
77     while(lista1 != NULL)
78     {
79         lista = Adiciona(lista, lista1->str);
80         lista1 = Remove(lista1);
81     }
82
83     while(lista != NULL)
84     {
85         lista1 = Adiciona(lista1, lista->str);
86         lista = Remove(lista);
87     }
88
89     return lista1;
90 }
91
92
93 int main(int argc, char **argv)
94 {
95     FILE *f;
96     char str[MAXSTR], *pt, c;
97     ListaSTR *lista = NULL, *aux;
98
99     /* verificar se foi dado um argumento */
100    if(argc <= 1)
101    {
102        printf("Utilização: sort <ficheiro>\n");
103        return;
104    }
105
106    /* abrir o ficheiro em modo de texto*/
107    f = fopen(argv[1], "rt");
108    if(f == NULL)
109    {
110        printf("Ficheiro %s não existe.\n", argv[1]);
111        return;
112    }
113
114    while(! feof(f))
115        if(fgets(str, MAXSTR, f) != NULL)
116            lista = Adiciona(lista, str);
117
118    lista = MergeSort(lista);
119
120    /* mostrar a lista ordenada */
121    aux = lista;
122    while(aux != NULL)
123    {
124        printf("%s", aux->str);
125        aux = aux->seg;
126    }
127
128    /* libertar a lista */
129    while(lista != NULL)

```

```

130     lista = Remove(lista);
131 }
132
133 /* find.c */
134 #define MAXSTR 1024
135
136 int Find(char *str, char *find)
137 {
138     int ifind, istr, i;
139     /* verificar se a string em find é possível de corresponder
140     à string de str, desde i */
141     for(i = 0; str[i] != 0; i++)
142     {
143         ifind = 0;
144         istr = i;
145         do
146         {
147             if(find[ifind] == 0)
148                 return i;
149             /* se se pretende ignorar um caracter, avançar ifind e istr */
150             if(find[ifind] == '?')
151             {
152                 ifind++;
153                 istr++;
154             }
155             else if(find[ifind] == '*')
156             {
157                 /* pretende-se ignorar zero ou mais caracteres:
158                 verificar se o resto da string pode-se obter com
159                 o resto da string de procura*/
160                 if(Find(str + istr, find + ifind + 1) >= 0)
161                     return i;
162                 break;
163             }
164             else if(find[ifind] == str[istr])
165             {
166                 /* match de dois caracteres, avançar */
167                 ifind++;
168                 istr++;
169             }
170             else
171                 /* caracteres distintos, parar */
172                 break;
173         } while(find[ifind] == 0 || str[istr] != 0);
174     }
175     return - 1;
176 }
177
178 int main(int argc, char **argv)
179 {
180     FILE *f;
181     char str[MAXSTR], *pt, c;
182     int count = 0;
183
184     /* verificar se foi dado um argumento */
185     if(argc <= 2)
186     {
187         printf("Utilização: find <ficheiro> <procura>\n");
188         return;
189     }
190
191     /* abrir o ficheiro em modo de texto*/
192     f = fopen(argv[1], "rt");
193     if(f == NULL)
194     {
195         printf("Ficheiro %s não existe.\n", argv[1]);
196         return;
197     }
198
199     /* adaptado de more.c */
200     while(! feof(f))
201     {
202         count++;
203         /* apresenta a linha apenas se há match */
204         if(fgets(str, MAXSTR, f) != NULL && Find(str, argv[2]) >= 0)
205         {
206             if(strlen(str) <= 75)
207                 printf("%4d:%s", count, str);
208             else
209             {
210                 pt = str;
211                 while(strlen(pt) > 75)
212                 {
213                     c = pt[75];
214                     pt[75] = 0;
215                     if(pt == str)
216                         printf("%4d:%s", count, pt);
217                     else
218                         printf("      %s", pt);
219                     pt[75] = c;
220                     pt += 75;
221                 }
222             }
223         }
224     }
225 }

```

```

91     }
92     printf("    %s", pt);
93 }
94 }
95 }
96 }
1 /* histograma.c */
2 #define MAXELEMENTOS 100
3 #define MAXVALORES 22
4 #define MAXSTR 255
5
6 /* função reutilizada do exercício doisdados.c da AF7 */
7 int GraficoDeBarras(int serie[], int n)
8 {
9     char grafico[MAXVALORES][MAXELEMENTOS];
10    int i, j, maximo, count = 0;
11    float factor;
12    if(n > MAXELEMENTOS)
13        return;
14    for(i = 0; i < n; i++)
15        if(i == 0 || maximo < serie[i])
16            maximo = serie[i];
17    factor = 15. / maximo;
18    sprintf(grafico[0], "");
19    sprintf(grafico[1], "    +");
20    for(i = 0; i < n; i++)
21        if(i % 5 == 0)
22            strcat(grafico[1], "+");
23        else
24            strcat(grafico[1], "-");
25    for(i = 2; i < 18; i++)
26        sprintf(grafico[i], "%5d%s", (int)((i - 2) / factor + 0.5), n,
27            "");
28    for(i = 0; i < n; i++)
29        for(j = 0; j < 16; j++)
30            if(serie[i] * factor >= j)
31            {
32                grafico[j + 2][i + 6] = '#';
33                count++;
34            }
35    for(i = 17; i > 0; i--)
36        printf("%s\n", grafico[i]);
37    return count;
38 }
39
40 int main(int argc, char **argv)
41 {
42     int histograma[256], i, j, count;
43     char str[MAXSTR], schar[256];
44     FILE *f;
45
46     if(argc != 2)
47     {
48         printf("Deve indicar um ficheiro para fazer o histograma.\n");
49         return;
50     }
51     f = fopen(argv[1], "r");
52     if(f == NULL)
53     {
54         printf("Nao foi possivel abrir para leitura o ficheiro %s.\n",
55             argv[1]);
56         return;
57     }
58
59     /* primeiro fazer o histograma completo */
60     for(i = 0; i < 256; i++)
61         histograma[i] = 0;
62     while(fgets(str, MAXSTR, f) != NULL)
63         for(i = 0; str[i] != 0; i++)
64             histograma[(unsigned char) str[i]]++;
65     /* concatenar e construir a string de valores */
66     for(i = 0; i < 256; i++)
67         schar[i] = i;
68     /* utilizar apenas os caracteres alfa-numéricos */
69     for(i = 0, j = 0; i < 256; i++)
70         if(schar[i] >= 'a' && schar[i] <= 'z' ||
71            schar[i] >= 'A' && schar[i] <= 'Z' ||
72            schar[i] >= '0' && schar[i] <= '9')
73         {
74             histograma[j] = histograma[i];
75             schar[j] = schar[i];
76             j++;
77         }
78     schar[j] = 0;
79     count = GraficoDeBarras(histograma, j);
80     printf("    %s\nNumero de cardinais no histograma: %d", schar,
81         count);
82 }
1 /* indicador.c */
2 #define MAXSTR 1024
3
4 typedef struct SListaSTR

```

```

5 {
6     char *str;
7     struct SListaSTR *seg;
8 }
9 ListaSTR;
10
11 ListaSTR *Adiciona(ListaSTR *lista, char *str)
12 {
13     ListaSTR *elemento = (ListaSTR *) malloc(sizeof(ListaSTR));
14     if(elemento != NULL)
15     {
16         elemento->str = (char *) malloc(strlen(str) + 1);
17         strcpy(elemento->str, str);
18         elemento->seg = lista;
19     }
20     /* em caso de falha, retorna NULL */
21     return elemento;
22 }
23
24 ListaSTR *Remove(ListaSTR *lista)
25 {
26     if(lista != NULL)
27     {
28         ListaSTR *aux = lista->seg;
29         free(lista->str);
30         free(lista);
31         return aux;
32     }
33     return NULL;
34 }
35
36 ListaSTR *Inverte(ListaSTR *lista)
37 {
38     ListaSTR *aux = NULL;
39     while(lista != NULL)
40     {
41         aux = Adiciona(aux, lista->str);
42         lista = Remove(lista);
43     }
44     return aux;
45 }
46
47 char *EspacosCodigo(char *codigo)
48 {
49     char *pt;
50     /* colocar todos os caracteres brancos em espaços */
51     while((pt = strchr(codigo, '\n')) != NULL)
52         *pt = ' ';
53     while((pt = strchr(codigo, '\r')) != NULL)
54         *pt = ' ';
55     while((pt = strchr(codigo, '\t')) != NULL)
56         *pt = ' ';
57
58     return codigo;
59 }
60
61 void RemoverStrings(ListaSTR *lista)
62 {
63     char *pt, tipo = ' ', lastchar = ' ', llastchar = ' ';
64
65     while(lista != NULL)
66     {
67         /* apagar as macros */
68         if(lista->str[0] == '#')
69         {
70             pt = (char *) malloc(1);
71             *pt = '\0';
72             free(lista->str);
73             lista->str = pt;
74             continue;
75         }
76         pt = lista->str;
77         while(*pt != '\0')
78         {
79             /* verificar se há início de string entre
80             aspas/plícas/comentários */
81             if(tipo != ' ' || pt[0] == '\\' || pt[0] == '"' ||
82                pt[0] == '/' && pt[1] == '*')
83             {
84                 if(tipo == ' ')
85                 {
86                     tipo = pt[0];
87                     pt++;
88                 }
89
90                 while(*pt != '\0' &&
91                      (pt[0] != tipo ||
92                       tipo == '/' && lastchar != '*' ||
93                       (*pt == '\\' || *pt == '"') &&
94                       lastchar == '\\' && llastchar != '\\'))
95                 {
96                     llastchar = lastchar;

```

```

97         lastchar = pt[0];
98         /* o conteúdo é anulado e colocado c, de forma a
99         não influenciar o indicador */
100         *pt = 'c';
101         pt++;
102     }
103     if(*pt != '\0')
104     {
105         tipo = ' ';
106         pt++;
107     }
108 }
109 else
110     pt++;
111 }
112 lista = lista->seg;
113 }
114 }
115
116 void InicioFimBlocos(ListaSTR *lista)
117 {
118     static char *inicioFimBlocos = "{}";
119     char *pt;
120     int i;
121     while(lista != NULL)
122     {
123         for(i = 0; inicioFimBlocos[i] != '\0'; i++)
124         {
125             /* não pode estar nada antes do início/fim do bloco */
126             if((pt = strchr(lista->str, inicioFimBlocos[i])) != NULL &&
127                pt != lista->str)
128             {
129                 lista->seg = Adiciona(lista->seg, pt);
130                 *pt = 0;
131             }
132             /* não pode estar nada depois do início/fim do bloco */
133             if(lista->str[0] == inicioFimBlocos[i] &&
134                strlen(lista->str) > 1)
135             {
136                 lista->seg = Adiciona(lista->seg, lista->str + 1);
137                 lista->str[1] = 0;
138             }
139         }
140         lista = lista->seg;
141     }
142 }
143
144
145 int Indicador(ListaSTR *lista)
146 {
147     int chavetasAbertas = 0, i;
148     int resultado = 0, conds = 0, ciclos = 0, comandos = 0;
149     char *pt;
150     static char *condsSTR[] =
151     {
152         "if", "case", NULL
153     };
154     static char *ciclosSTR[] =
155     {
156         "for", "while", NULL
157     };
158     /* processar todos os blocos globais */
159     printf("\n conds ciclos comandos parcial");
160     while(lista != NULL)
161     {
162         /* contar os blocos */
163         if(lista->str[0] == '{')
164             chavetasAbertas++;
165         else if(lista->str[0] == '}')
166         {
167             /* fim de bloco */
168             chavetasAbertas--;
169             if(chavetasAbertas == 0)
170             {
171                 /* contabilizar o bloco global */
172                 resultado += (conds + 2 *ciclos + 1) *(conds + 2 *ciclos +
173                    1) + comandos;
174                 /* mostrar parcial */
175                 printf("\n%8d %8d %8d %8d",
176                    conds, ciclos, comandos, resultado);
177                 /* zerar os valores para o próximo bloco */
178                 conds = ciclos = comandos = 0;
179             }
180             else if(chavetasAbertas < 0)
181                 chavetasAbertas = 0;
182         }
183         else
184         {
185             /* número de comandos */
186             pt = lista->str;
187             while((pt = strchr(pt + 1, ';')) != NULL)
188                 comandos++;

```



```

189
190     /* número de conds */
191     for(i = 0; condsSTR[i] != NULL; i++)
192     {
193         pt = lista->str;
194         while((pt = strstr(pt, condsSTR[i])) != NULL)
195         {
196             conds++;
197             pt++;
198         }
199     }
200     /* número de ciclos */
201     for(i = 0; ciclosSTR[i] != NULL; i++)
202     {
203         pt = lista->str;
204         while((pt = strstr(pt, ciclosSTR[i])) != NULL)
205         {
206             ciclos++;
207             pt++;
208         }
209     }
210 }
211
212 lista = lista->seg;
213 }
214 return resultado;
215 }
216
217 int main(int argc, char **argv)
218 {
219     FILE *f;
220     int i;
221     char str[MAXSTR], *pt, c;
222     ListaSTR *lista = NULL, *aux, *buffer;
223
224     /* verificar se foi dado um argumento */
225     if(argc <= 1)
226     {
227         printf("Utilizacao: indicador <ficheiro>\n");
228         return;
229     }
230
231     /* abrir o ficheiro em modo de texto */
232     f = fopen(argv[1], "rt");
233     if(f == NULL)
234     {
235         printf("Ficheiro %s nao existe.\n", argv[1]);
236         return;
237     }
238
239     while(! feof(f))
240     {
241         if(fgets(str, MAXSTR, f) != NULL)
242             lista = Adiciona(lista, str);
243
244         lista = Inverte(lista);
245
246         /* acertar os espaços nas linhas */
247         aux = lista;
248         while(aux != NULL)
249         {
250             aux->str = EspacosCodigo(aux->str);
251             aux = aux->seg;
252         }
253
254         /* congelar strings de comentários, strings e plicas */
255         RemoverStrings(lista);
256
257         /* início/fim de blocos em linhas isoladas */
258         InicioFimBlocos(lista);
259
260         /* calcular indicador */
261         printf("\nIndicador: %d", Indicador(lista));
262
263         /* libertar a lista */
264         while(lista != NULL)
265             lista = Remove(lista);
266     }

```

ÍNDICE DE PROGRAMAS

Programa 1-1 Olá Mundo	7
Programa 2-1 Troca o valor de duas variáveis.....	9
Programa 2-2 Troca o valor de duas variáveis sem variável auxiliar.....	10
Programa 2-3 Imprime o número de bytes que um inteiro ocupa	11
Programa 2-4 Imprime diferentes tipos de variáveis	12
Programa 2-5 Pede valores de variáveis de diferentes tipos que depois imprime.....	13
Programa 3-1 Determina se um número é par ou impar.....	16
Programa 3-2 Determina se um ano é normal ou bissexto.....	18
Programa 3-3 Determina o número de dias de um mês/ano.....	19
Programa 4-1 Soma dos primeiros 4 quadrados naturais	23
Programa 4-2 Soma dos primeiros 4 quadrados naturais utilizando um ciclo	24
Programa 4-3 Imprime o código numérico correspondente a cada letra.....	25
Programa 4-4 Calculo do número de pares de inteiros que respeitam uma determinada condição	26
Programa 5-1 Escolha de opções num menu.....	40
Programa 5-2 Escolha de opções num menu, utilizando uma função	42
Programa 5-3 Determina se um ano é normal ou bissexto utilizando funções	44
Programa 5-4 Determina o número de dias de um mês/ano utilizando funções.....	45
Programa 6-1 Réplica do Programa 4-2.....	49
Programa 6-2 Soma dos primeiros 4 quadrados naturais com o ciclo “for”	49
Programa 6-3 Utilização alternativa do ciclo “for” na soma dos 4 quadrados naturais.....	50
Programa 6-4 Versão com ciclos “for” do Programa 4-4	50
Programa 6-5 Versão com ciclos “do-while” do Programa 5-2	52
Programa 6-6 Versão removendo a função Menu do Programa 6-5.....	53
Programa 6-7 Versão com “switch” do Programa 6-6	54
Programa 6-8 Função DiasDoMes com “switch” do Programa 5-4.....	55
Programa 7-1 Calculo do total e média de um indicador por cada dia da semana	57
Programa 7-2 Calculo do total e média, versão com vectores	58
Programa 7-3 Confirmação que uma string é um vector de caracteres	60
Programa 7-4 Calcular o tamanho de uma string	61
Programa 7-5 Função DiasDoMes com um vector inicializado na declaração.....	62
Programa 7-6 Exemplificação das diferentes maneiras de se inicializar strings.....	62
Programa 7-7 Calculo do total e média, sem código repetido, utilizando vectores	63
Programa 7-8 Leitura de argumentos passados ao programa	64
Programa 7-9 Soma de duas matrizes	64
Programa 8-1 Tentativa de implementação de uma função para trocar o valor de duas variáveis....	69
Programa 8-2 Função Troca que realmente troca o valor de duas variáveis.....	70
Programa 8-3 Versão final da função de troca de duas variáveis.....	71
Programa 8-4 Exemplo de utilização de variáveis globais.....	72
Programa 8-5 Exemplo de utilização de variáveis estáticas (“static”)	73
Programa 8-6 Procura a primeira ocorrência de uma string em outra (strstr)	74
Programa 9-1 Factorial, versão iterativa e recursiva	78
Programa 9-2 Torres de Hanoi	80
Programa 10-1 Generalização do Programa 7-2, calculando a média e o desvio padrão	101
Programa 10-2 Réplica do Programa 9-1.....	103
Programa 10-3 Calculo da média e desvio padrão, alocando apenas a memória necessária	104
Programa 10-4 Concatena as strings introduzidas, utilizando apenas a memória necessária	106

Programa 10-5 Geração de um vector aleatório.....	107
Programa 11-1 Réplica do Programa 5-4.....	113
Programa 11-2 Cálculo dos dias de um mês/ano, utilizando estruturas	115
Programa 11-3 Registo de uma estrutura com dados de diferentes tipos de pessoas.....	116
Programa 11-4 Versão do registo de pessoas, mais legível e eficiente	118
Programa 11-5 Estrutura de um vector com os valores e a sua dimensão	119
Programa 11-6 Tipo abstracto de dados TVectorInt	121
Programa 11-7 Versão de TVectorInt com apontadores, e controle do tempo	123
Programa 11-8 Ajustes feitos no tipo abstracto de dados TVectorInt, para melhoria da eficiência	125
Programa 11-9 Tipo abstracto de dados TListaInt.....	128
Programa 12-1 Número de dias dado mês/ano, lendo dados a partir do ficheiro “in.txt”	134
Programa 12-2 Número de dias do mês dado mês/ano lendo dados de “in.txt” e escrevendo para “out.txt”	135
Programa 12-3 Ficha de pessoa, com gravação e leitura do ficheiro.....	138
Programa 12-4 Actualização do tipo abstracto TVectorInt para ler/gravar para ficheiro, em modo binário	141
Programa 13-1 Versão condensada do número de dias de um dado mês/ano	143
Programa 13-2 Versão ainda mais condensada do número de dias de um dado mês/ano.....	144
Programa 13-3 Instrução de shift vs multiplicação/divisão por 2	145
Programa 13-4 Representação binária de um número real	146
Programa 13-5 Versão do Programa 13-4 utilizando a estrutura “union”	147
Programa 13-6 Utilização do formato do número real para aumentar a eficiência de algumas operações.....	148
Programa 13-7 Exemplo de utilização de várias variáveis booleanas numa só variável inteira	149
Programa 15-1 Código incompreensível	166
Programa 15-2 Programa incompreensível, preparado para debug	167

ÍNDICE REMISSIVO

"(null)"	75, 108
#define	65
#include	7

A

abrir um ficheiro e não chegar a fechar	142
abrir um ficheiro e não testar se foi bem aberto	142
abstracção de dados	120
abstracção funcional	43
adiciona	151
alocar memória e não testar se a operação foi bem sucedida	112
apontadores	74
argumentos	83
argv	64
arranjos	31
atribuições	9
atribuições entre variáveis de tipos distintos	112

B

baralhar	89
basededados	155
binarysearch	91
bits num número real	146
bloco de código	19
break	55

C

calendario	94
cast	105
char	11
ciclo dentro de outro ciclo	26
Ciclos	23
ciclos contendo apenas uma variável lógica no teste de paragem	55
cifracesar	154
combinacoes	33
comentários ao código	9
comentários explicam a linguagem C	14
comentários inexistentes em partes complexas	150
compilador	7
compilador e Editor de C	161
compile time	100
Condicinais	16

conjunto de conjuntos de caracteres	63
conjunto de strings	64
conjuntos de variáveis	57
continue	55
copy / paste	41

D

declaração de variáveis	9
declarações de variáveis fora do início das funções	47
declarações ou atribuições a variáveis nunca utilizadas	13
doisdados	94
double	11
do-while	51
duas instruções na mesma linha ..	21

E

editdistance	96
else	16
esimo	153
erros comuns	13
estrutura de dados	128
Estruturas	113
euler	33
execução alternativa	16
execução do programa	13
execução passo-a-passo	10
Exemplo de código incompreensível	165
expressão lógica	16, 20

F

fibonacci	32
ficheiros	130
find	84, 157
float	11
fopen / fprintf / fscanf / fgets / feof / fclose	133
for	49
formularesolvente	36
fread / fwrite	140
free	104
fseek / ftell	140
função	40
funções com muitos argumentos ..	77
funções com parâmetros no nome	46
funções específicas	47

funções muito grandes	76
Funções standard mais utilizadas	169

G

gets	60
------------	----

H

Heap	104
histograma	158
hms	30

I

if	16
indentação variável	20
indicador	159
inicialização de strings	62
inicialização de um vector	62
insere	152
insertsort	153
instruções parecidas não seguidas	46
instruções parecidas seguidas	27
int	11
inverte	83

J

jogodados	88
-----------------	----

L

linguagem C	7
linguagem de programação	7
linguagem imperativa	7
linhas de código nunca executadas	21
listas	126
long	11
long long	11

M

má utilização da recursão	82
má utilização do switch ou cadeia if-else	66
macro	65
macros em todas as constantes ..	67

main	8
mais ciclos e condicionais	49
malloc	104
matrizes	64
maximo	85
mdc.....	85
Memória	100
mergesort	90, 156
modo binário	139
modo de texto	139
more	151, 155

N

Não consigo perceber o resultado do programa	164
não libertar memória após alocar	112
ndamas.....	93
nomes sem significado.....	76
Notepad++	163
NULL.....	75, 105
numeracaoromana	91

O

olá mundo.....	7
olamundosizeof	29
operações binárias lógicas	146
operador "."	116
operador &	71
operador *	71, 116
operador ?	143
operador ->	116
operador >> >= << <=	145
operador de linha de comando 	132
operador de linha de comando >>	132
operadores & ^ ~	146
operadores && !	17
operadores + - * / %	17
operadores ++ --	24
operadores += -= *= /= %=	24
operadores == != > >= < <=	17
operadores binários	145
operadores de linha de comando < e >	130

P

palavras	154
passagem por referência.....	71
pi.....	35
pokerdados	92
pré-processamento	7
Primeiro Programa	7
primo	34
printf	8
Procedimentos	69
produto.....	31
programa.....	7
programar	7

R

rand	84
Recursão.....	78
removedups	87
run time	100

S

scanf	12
short.....	11
situação de desespero	164
sizeof	11
soma	30
somadigitos.....	32
somanumeros	87
sort	86, 157
srand	84
Stack.....	102
static.....	72
stdin	135
stdlib.h	84
stdout.....	136
strcat	74
strcpy	74
string	10, 59
string de formatação.....	11
string.h	74
strlen	60
strreplace	93
strstr.....	74
strtok.....	87
struct	114

switch	53
--------------	----

T

TCC: Tiny C Compiler	161
tempo de compilação	100
tempo de corrida.....	100
time(NULL);	84
time.h	84
tipos abstractos de dados.....	120
tipos elementares.....	11
torres de Hanoi.....	79
triplasoma	35
trocos	34
trocos2	88
truques	143
typedef	117

U

uma atribuição, uma leitura	13
union	147
utilização do goto	27
utilizar atribuições como expressões	144

V

Variáveis	9
variáveis com nomes quase iguais	66
variáveis desnecessárias.....	14
variáveis estáticas.....	72
variáveis globais.....	72, 76
variável	9
variável iteradora	24
vector de caracteres	59
Vectores	57
vectores de vectores	64
vectores multi-dimensionais	64

W

while	23
-------------	----