Programação em C/C++ - Formas Avançadas

Sérgio Lopes

Programação em C/C++ - Formas Avançadas

Sérgio Lopes

Índice

Lista de Tabelas

Sobre o Manual

Público Alvo

Este manual foi criado, especificamente, para o módulo *0785 - Programação em C/C++ - Formas Avançadas*, ministrado em formações profissionais como indicado pelo catálogo da ANQ, www.catalogo.anq.gov.pt. Tem como base os módulos 782 - Programação em C/C++ - Estrutura Básica e Conceitos Fundamentais, 783 - Programação em C/C++ - Ciclos e Decisões e 784 - Programação em C/C++ - Funções e Estruturas.

Estrutura e Conteúdo

Esta manual começa por se afastar do código da linguagem de programção C, introduzindo os conceitos de pré-processamento, das suas directivas e do seu impacto na compilação, passando depois para a apresetação de outros sistemas numéricos tais como o sistema binário ou o sistema hexadecimal. Voltando ao código, o manual pega nos conceitos iniciais para introduzir campos de bits, acesso a ficheiros de texto e ficheiros binários.

Licença

Capítulo 1. Pré-Processador de C

Em linguagem de programação C todas as ferramentas de desenvolvimento de programas (onde se incluem compiladores e *linkers*) possuem uma ferramenta dedicada ao pré-processamento dos ficheiros de código, esta ferramenta, a que comummente chamamos **pré-processador** é responsável por fazer o primeiro processamento do nosso código, alterando o mesmo de acordo com as várias directivas que se encontram nos ficheiros.

Deste processo resulta um código modificado que é depois entregue ao compilador onde é, efectivamente, compilado. Assim, é possível que o código que é escrito pelo programador seja modificado, em alguns casos com poucas alterações noutros com alterações profundas, antes de ser compilado e transformado no programa final, pronto a usar. Naturalmente, estas modificações são sempre ordenadas pelo programador.

As directivas de pré-processamento, que informam o pré-processador das alterações a fazer, são colocadas depois do carácter **#** e embora possa aparecer em qualquer parte do código, o mais comum é surgirem no início, antes de qualquer definição de estrutura, declaração de função ou da função *main*.

Directivas de Pré-Processamento

De seguida serão apresentadas as directivas e macros usadas pelo pré-processador. Podem existir directivas diferentes ou até macros diferentes, as apresentadas são as consideradas padrão e que devem estar disponíveis em qualquer conjunto de ferramentas de desenvolvimento.

Porque estas directivas não são instruções da linguagem C mas sim instruções a fornecer a uma ferramenta especial de apoio ao compilador, não são terminadas com o **;**. Nenhuma directiva de pré-processamento termina com **;**.

#include

Esta directiva permite a inclusão de ficheiros externos. É a directiva que usamos para incluir os ficheiros onde estão definidas funções da biblioteca padrão ou funções definidas por nós mas que não foram criadas no ficheiro de código onde estamos a trabalhar. O exemplo mais comum desta situação é a inclusão dos ficheiros *stdio.h* e *stdlib.h* que possuem as funções mais usadas na criação dos nosso programas, como a função *printf* ou a função *scanf*.

Ao encontrar esta directiva, o pré-processador copia para o local da directiva todo o conteúdo do ficheiro indicado. A indicação do ficheiro pode ser feita de duas maneiras ligeiramente diferentes, usando os caracteres < e > ou usando o carácter ".

As duas formas resultam no mesmo a diferença está na forma como o pré-processador procura os ficheiros. No primeiro caso, usando os caracteres < e >, o pré-processador procura o ficheiro a incluir na localização padrão [1], no segundo caso, usando o carácter ", o pré-processador procura o ficheiro na mesma localização que o ficheiro de código que está a processar. Esta última forma é a mais comum quando estamos a tentar incluir ficheiros que nós próprios criámos.

Exemplo.

[1] A localização padrão depende da instalação do compilador e restantes ferramentas

```
#include <stdio.h>
#include "minhasfuncoes.h"
```

#define

A directiva **#define** permite a criação de macros, no formato *chave-valor*, que o pré-processador irá substituir ao longo do nosso código, sempre que encontrar o nome da macro. Assim, ao definirmos uma macro com o código **#define MAX 50**, o pré-processador irá substituir o nome **MAX**, sempre que o encontrar no código, pelo valor **50**.

O uso desta directiva permite simular a criação de constates [2], facilitando assim a construção de código que precise de usar valores que, sendo constantes na maioria das situações, podem vir a ser alterados mais tarde. Por exemplo, o uso de uma macro para o valor do IVA, que é constante durante um ano, ou o uso de uma macro para o valor máximo que um vector de caracteres pode ter.

Exemplo.

```
#define IVA 0.21
#define PI 3.1514
#definem MAX_NOME 251
```

O nome das macros segue a mesma regra que os nomes de variáveis ou funções e é comum que sejam escritos em maiúsculas, no entanto, o valor da macro, que corresponde ao texto que vem depois do nome e que é separado do mesmo por um espaço, pode conter qualquer texto, expressão ou instrução de C já que o pré-processador irá substituir esse valor em todas as ocorrências do nome.

As macros podem ainda ter parâmetros na sua definição, de tal forma que são usadas quase como as funções. As regras para a escrita de parâmetros em macros são diferentes das regras das funções, e nas macros não há definição de tipos de dados. A sintaxe é:

```
#define <nome da macro>(<lista de parâmetros separados por vírgulas>) <expressao>
```

Exemplo.

```
#define DOBRO(x) (2 * (x))
```

A macro criada no exemplo permite receber um valor que é depois multiplicado por dois. Repare como não existe qualquer indicação do tipo do parâmetro **x**, o que implica que possamos passar para dentro da macro qualquer valor numérico.

Para usarmos a macro criada basta usar o nome e colocar dentro de parêntesis o valor que pretendemos dar ao parâmetros **x**:

```
#define DOBRO(x) (2 * (x))

int main()
{
    printf("Resultado: %d", DOBRO(30));
}
```

No entanto, este tipo de construção não é comum e deve ser evitado. É para isto que servem as funções além de que usar a directiva **#define** para substituir ou simular funções torna-se facilmente complexo,

[2] Vimos no primeiro módulo que estas macros não são verdadeiras constantes.

difícil de gerir e impossibilita a correcta utilização de algumas ferramentas como as ferramentas de *debug* para a detecção de erros.

Esta é também a primeira directiva que estamos a estudar do conjunto de directivas de **Compilação Condicional**, processo no qual escolhemos qual o código que é compilado de acordo com determinadas regras ou condições. Com este processo o pré-processador irá remover código do ficheiro quando o mesmo não for para compilar e deixar apenas o que for necessário. Todas as directivas que vão ser vistas nas secções seguintes são directivas que permitem controlar a compilação de código.

#undef

A directiva **#undef** opõe a directiva **#difine**. Se uma permite definir uma macro, a outra permite remover a definição de qualquer macro, fazendo com que, a partir do momento em que é invocada, a macro deixe de existir e deixe de ser alterada pelo pré-processador.

#if, #elif, #else e #endif

Estas directivas permitem controlar a compilação de código incluíndo ou excluíndo blocos de código de acordo com as condições verificadas. São em tudo similares às estruturas de controlo **if**, **if… else**.

A directiva **#if** permite avaliar uma condição, a directiva **#elif** é igual à estrutura **else if**, aparecendo sempre depois da directiva **#if**. A directiva **#else** é homónima da estrura **else**, também sendo colocada depois da directiva **#if**.

A única directiva que não tem um par com as estruturas de controlo é a directiva **#endif**. Esta directiva permite terminar uma directiva **#if**, correspondendo grosseiramente à chaveta que termina o bloco de código da estrutura de controlo **if**.

Exemplo.

```
#define DEBUG 1
#define PRODUCAO 0

int main()
{
#if DEBUG
    printf("Informação de DEBUG: ...\n");
#elif PRODUCAO
    printf("Programa em estado de produção\n");
#else
    printf("Estado de compilação indefinido\n");
#endif
}
```

Como vemos no exemplo, a estrura formada pelas directivas **#if**, **#elif**, **#else** e **#endif** é semelhante à estrutura de controlo **if… else***. As duas directivas **#elif** e **#else** são opcionais, mas as directivas **#if** e **#endif** são necessárias, esta última é sempre colocada no fim para indicar que a construção terminou.

#ifdef e #ifndef

Duas directivas que permitem testar se determinada macro está definida. A primeira directiva devolve verdadeiro caso a macro esteja definida, a segunda directiva devolve falso se a macro não estiver definida.

Exemplo.

```
#define TESTE 1

int main()
{
    #ifdef TESTE
       printf("Macro TESTE definida\n");
    #endif

    #ifndef EXE
        printf("Macro EXE não definida\n");
    #endif

    return 0;
}
```

Se avaliarmos o exemplo à luz do que sabemos sobre a directiva **#if** podemos verificar que as directivas **#ifdef** e **#ifndef** não são mais que atalhos para facilitar a avaliação de que determinada macro está definida. Daí a necessidade da directiva **#endif** a indicar o fim da construção.

#error

A directiva **#error** irá emitir um erro de compilação, terminando a compilação do nosso código e mostrando o erro na consola, ou na secção de erros do IDE se estivermos a usar algum. A directiva pode ser usada para testar alguma condição necessária à correcta compilação do código, como a existência de um ficheiro particular ou de uma *flag* de compilação.

Exemplo.

```
#ifndef FLAG_IMPORTANTE
    #error Não foi definida a FLAG_IMPORTANTE
#endif
```

A directiva **#error** não precisa estar dentro de uma directiva **#if**, mas nesse caso é sempre executada o que implica que o nosso código nunca é compilado.

Comparações Com Directivas

É possível usar operadores de comparação com directivas de pré-processamento, no entanto estes operadores apenas estão definidos para números inteiros, isto é, não irão funcionar com outros dados que não sejam número. Os operadores disponíveis são os mesmos que para a linguagem C, entre outros, ==, >, >=, <=, <, **!**=.

Estão também disponíveis operadores *boleanos*, que permitem avaliar logicamente uma condição, e que são também iguais aos da linguagem C.

Outras Directivas

Existem outras directivas, além as indicadas. A tabela seguinte pretende oferecer uma visão geral de todas as directivas existentes.

#include	Permite incluir o conteúdo de outros ficheiros, tipicamente ficheiros com definições de funções. Ex: *#include <stdio.h>*.
#define	Permite definir um símbolo, vulgarmente chamado de macro no formato *chave-valor*. Ex: *#define IVA 0.21*.
#undef	Remove uma definição anteriormente feita com o *#define*. Ex: *#undef IVA*.
#if	Testa uma condição.
#elif	Cláusula alternativa à directiva **#if** que incluí o teste de uma nova condição.
#else	Cláusula alternativa à directiva **#if**.
#endif	Termina uma construção iniciada com **#if**, **#ifdef** ou **#ifndef**.
#ifdef	Permite testar se uma macro está definida.
#ifndef	Permite testar se determinada macro não está definida.
#line	Altera a numeração da linha, renumerando a linha de código onde é invocada.
#pragma	Permite o uso de características específicas do compilador. Estas características serão diferentes de compilador para compilador e podem incluir optimizações ou outras funcionalidades extra.
#error	Termina a compilação e mostra uma mensagem de erro.
#	Não é uma directiva, mas sim um operador que permite criar *Strings* a serem usadas nas directivas.
##	Permite juntar valores das macros.
defined	Permite verificar se uma macro está definida. Funciona como a directiva **#ifdef**. Ex: *defined(MACRO)*.

Macros Pré-Definidas

Todas as ferramentas de desenvolvimento de programas em linguagem C oferecem algumas macros já definidas e que podemos usar nos nosso programas, essas macros não podem ser redefinidas não podendo ser alteradas, removidas ou criadas novas macros com os mesmos nomes. Embora possam existir mais macros oferecidas pelo pré-processador que estejamos a usar, as macros padrão que podem ser encontras em qualquer pré-processador estão indicadas na tabela seguinte.

__LINE__	Devolve o número da linha onde é invocada.
__FILE__	Devolve o nome do ficheiro onde está a ser usada.
__DATE__	Indica a data actual no formato Mmm dd yyyy. Esta data é dependente da configuração do pré-processador e se não puder se determinada são mostrados vários caracteres **?**. O resultado exacto desta macro depende também da definições regionais se o pré-processador estiver configurado para as usar. Ex: *Set 29 2010*.
__TIME__	Indica a hora, no formato hh:mm:ss. Ex: *14:23:35*.
__STDC__	Indica se a compilação é feita em modo padrão, devolvendo 1 em caso afirmativo.

Capítulo 2. Outros Sistemas Numéricos

O sistema numérico que usamos é o sistema decimal. Este sistema, embora extremamente útil para as pessoas, não é o melhor para ser usado em computadores que funcionam à base de sistemas digitais, onde o sinal positivo ou zero da corrente que passa pelos circuitos é usado para registar a informação. Num ambiente como este, é necessário que existam sistemas numéricos adequados e que permitam representar, com alguma fiabilidade, a informação necessária ao correcto funcionamento dos sistemas (programas gerais, sistema operativo, periféricos, etc.).

Surgem então outros sistemas numéricos, mais adequados a serem usados por computadores, como o **sistema binário**, o **sistema octal** ou o **sistema hexadecimal**.

Binário

O sistema binário faz uso de apenas dois digitos, 0 e 1, para a representação de todos os valores, cada número é representado com um conjunto de *zeros* e *uns*. Por usar apenas dois dígitos, esta notação é designada como base 2. Um número binário poderia ser representado por qualquer combinação de elementos que possuam dois estados exclusivos, embora o uso do símbolo "0" e "1" seja o mais comum.

Exemplo Número 37: 100101

As operações de soma, multiplicação, subtração ou divisão seguem as mesmas regras que na notação decimal. A diferença está em que no caso da notação decimal existiam dez símbolos possíveis para representar os números e no caso da notação binária apenas existem dois.

Octal

O sistema octal tem como base oito símbolos, tipicamente de "0" a "7", que são usados para representar os vários números. Este tipo de notação é usada principalmente quando se pretendem representar conjuntos de três bits, por exemplo nos sistemas de permissões de ficheiros.

Exemplo Número 74: 112

Hexadecimal

Hexadecimal usa como representação dezasseis símbolos, os dez primeiros são comuns ao sistema decimal e os seis últimos são representados pelas letras entre "A" e "F". Este sistema é usado em grande parte para facilitar a representação de números binários a ler pelo utilizador, como é o caso de endereços de memória, ou para representar caracteres que normalmente não podem ser representados, como é o caso de espaços em URLs, que são representados pelo número hexadecimal **20**, são ainda muito usados na criação de páginas WEB para representar cores nas componentes RGB, Red(vermelho), Green(verde), Blue(azul), por exemplo FF0000 representa vermelho.

Exemplo Número 255: FF

Conversão Entre Sistemas

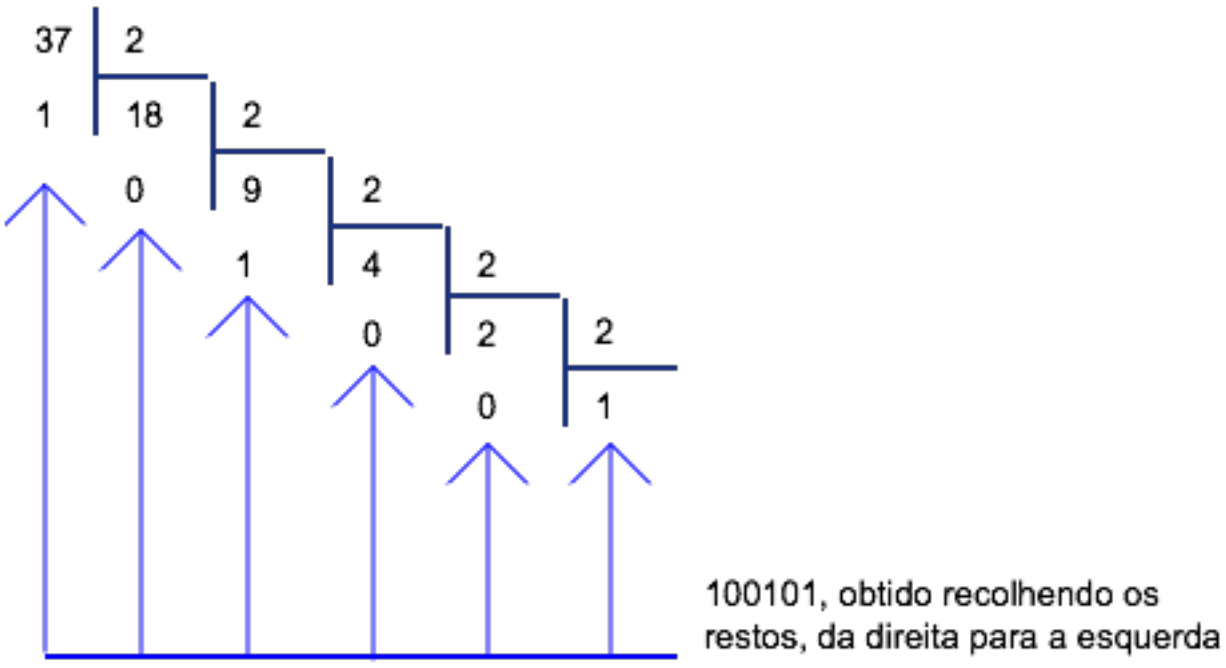
As regras seguintes demonstram como fazer a conversão entre os vários sistemas apresentados anteriormente. Estas regras são similares entre os vários sistemas alterando apenas o valor da base que identifica o sistema numérico.

Binário e Decimal

Converter de binário para decimal:

```
10010100101
(1 x 2^5) + (0 x 2^4) + (0 x 2^3) + (1 x 2^2) + (0 x 2^1) + (1 x 2^0) =
(1 x 32) + (0 × 16) + (0 × 8) + (1 × 4) + (0 × 2) + (1 × 1) = 37
```

Converter de decimal para binário:

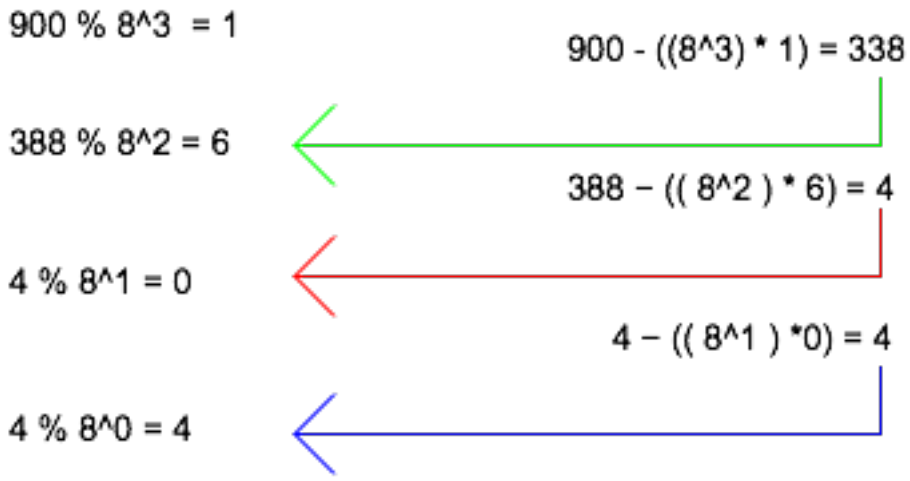


Octal e Decimal

Converter de octal para decimal:

```
112
(1 x 8^2) + (1 x 8^1) + (2 x 8^0) = 74
```

Converter decimal para octal:



Hexadécimal e Decimal

Converter de decimal para hexadecimal:

A conversão entre decimal e hexadecimal é mais fácil se for feita indirectamente, isto é, se convertermos primeiro para binário e depois para hexadecimal.

Converter de hexadecimal para decimal:

```
C0E716
C0E716 = (12 x 16^5) + (0 x 16^4) + (14 x 16^3) + (7 x 16^2) + (1 x 16^1) + (6 x 16^0) =
(12 x 4096) + (0 x 256) + (14 x 16) + (7 x 1) = 49 383
```

Binário e Hexadecimal

Converter de binário para hexadecimal:

Separe o número binário em conjuntos de quatro dígitos, adicione zeros à esquerda se for necessário, depois basta substituir os conjuntos binários pela sua representação.

```
1100101
1100101 = 0110 0101 = 6 5 = 65
```

Converter de hexadecimal para binário:

Basta fazer a operação inversa à anterior, separando os dígitos e substituindo pela sua representação binária.

```
E7
E7  = E 7 = 1110 0111 = 11100111
```

Binário e Octal

A conversão de binário para octal segue a mesma regra da conversão para hexadecimal com a diferença de que os bits são agrupados três a três em vez de quatro a quatro.

Converter binário para octal:

```
1100101
1100101 = 001 100 101 = 1 4 5 = 145
```

Converter octal para binário:

```
65
65 = 110 101
```

Capítulo 3. Operações sobre Bits

Operações sobre bits são operações que afectam o valor das variáveis no seu formato binário. Sempre que for aplicada uma operação sobre os bits de uma variável, o valor dessa variável é convertido para binário e a operação é aplicada a esse valor binário. O resultado da operação é depois convertido para o valor decimal e a variável é actualizada [1].

NOT

Operação de negação. Com esta operação todos os bits da variável são negados e, uma vez que em binário só existem duas representações possíveis, os **zeros são transformados em uns** e os **uns são transformados em zeros**.

```
int main()
{
    int x = 4;

    printf("Valor original de X: %d", x);

    x = ~x;

    printf("Valor negado de X: %d", x);

    return 0;
}
```

Resultado:

```
Valor original de X: 4
Valor negado de X: -5
```

OR

Operação de *OU*. Esta operação aplica um *OU*, ou *soma lógica* a cada um dos bits da variável.

```
int main()
{
    int x = 4, y = 5;

    printf("Valor da soma lógica: %d", (x | y));

    return 0;
}
```

Resultado:

```
Valor da soma lógica: 5
```

AND

Operação de *E*. *Multiplicação lógica* dos bits da variável.

```
int main()
```

[1] Esta explicação não corresponde ao processo interno correcto, e pretende apenas ajudar a explicar as operações sobre bits.

```
{
    int x = 4, y = 5;

    printf("Valor da multiplicação lógica: %d", (x & y));

    return 0;
}
```

Resultado:

```
Valor da multiplicação lógica: 4
```

XOR

Uma operação de *XOR*, ou de *ou exclusivo*, usa dois números binários de tamanho igual (se os números não tiverem a mesma quantidade de dígitos então são adicionados zeros à esquerda do número com menos dígitos) e, para cada bit ou dígito do número, coloca um **zero se os bits forem iguais** e um **um se os bits forem diferentes**.

Exemplo de Operação XOR.

```
    0111 (igual a 7 decimal)
XOR 0011 (igual a 3 decimal)
  = 0100 (igual a 4 decimal)
```

Como regra simples, podemos considerar que uma operação de *ou exclusivo* só aceita que os dígitos sejam diferentes, devolvendo um zero na posição onde dois dígitos são iguais.

Na linguagem de programação C, a operação de *ou exclusivo* é aplicada através do uso do carácter ^.

Capítulo 4. Campos Bit

Os campos bit, do inglês *bit-fields*, membros de estruturas com comportamento especial. Estes membros, que podem existir misturados com membros de outros tipos de dados, registam os valores que lhes são atribuídos de forma binária, isto é, se num campo bit colocarmos o valor 7, esse valor vai ser convertido para binário e vai preencher o campo bit. Esta característica implica que um campo bit seja dependente da implementação interna da linguagem, e assim dependente do sistema operativo, e só conseguem guardar valores cuja representação binária caiba no número de bit com que o campo foi declarado.

O campos bit estão limitados aos tipos de dados inteiros, especificamente o *int*, *unsigned int* e *signed int* [1].

Quando presentes numa estrutura, a sintaxe de declaração é diferente da dos restantes membros, e se existirem campos bit misturados com membros de outros tipos, os campos bit devem ser colocados antes de todos os restantes membros.

Exemplo Estrutura com Dois Campos Bit.

```
struct ex {
    //campos bit
    int b1: 1;
    int b2: 3;

    //restantes membros
    char descricao[100];
};
```

Como podemos ver no exemplo, a estrutura de nome **ex** possuí três membros em que os dois primeiros, de nome **b1** e **b2**, são campos bit e o terceiro, de nome **descricao**, é um vector de caracteres com 100 posições.

A sintaxe dos campos bit implica que, depois do nome da variável, seja colocado o carácter **:** seguido do número de bits a serem usados pelo campo. Este número de bits define o tamanho máximo de valores que podem ser guardados, uma vez que todos os valores são convertidos para a sua representação binária. No exemplo, o campo **b1** tem apenas *1 bit* pelo que poderá guardar no máximo dois valores, 0 e 1, enquanto que o campo **b2** tem *3 bits* pelo que poderá guardar 8 valores diferentes, de 0 a 7.

Este tipo de dados é usado apenas quando a quantidade de memória é muito limitada, por exemplo em dispositivos móveis, telecomandos ou controladores industriais.

[1] Por omissão, um *int* é um *unsigned int*

Capítulo 5. Ficheiros

Todos os programas que temos feito até agora perdem os dados assim que terminam. A informação que gerem é registada apenas em memória *RAM*, e apenas durante a execução do programa. Para podermos persistir a informação, e garantir que lhe conseguimos aceder entre execuções do programa é necessário que a mesma seja transferida para um tipo de memória que seja persistente. É nessa situação que entram os **ficheiro**.

Ficheiros, embora estruturas de dados que são usadas em memória, têm a capacidade para que a informação que contém seja guardada num suporte como o disco rígido do computador, um CD ou *PEN* de modo a ficarem registados de forma permanente.

Embora a estrutura interna dos ficheiros não seja exactamente assim, podemos considerar que todos os ficheiros possuem um nome, que incluí o caminho completo para o ficheiro e qualquer extensão que seja definida, e uma zona de memória onde os dados do ficheiros são carregados e com os quais trabalhamos. Esta zona de memória pode conter mais informação, ou menos informação, que a presente no disco onde o ficheiro foi guardado. Pode também existir sem que tenha ainda sido gravado para um disco.

Assim, um ficheiro, em linguagem de programação C, corresponde sempre a duas secções: a de memória *RAM* que está a ser usada pelo programa que abriu o ficheiro e a zona do disco onde o ficheiro está guardado. Um ficheiro pode existir apenas em memória e nunca ser guardado num disco, exemplo disso são as *streams padrão* [1].

As *streams padrão* são três ficheiros que são abertos pelo sistema operativo e que existem apenas enquanto o mesmo estiver em execução, e que oferecem a capacidade de ler dados do teclado, escrever dados para o ecrã escrever erros (que tipicamente são enviados para o ecrã). Estas *streams* possuem o nome de stdin, stdout e stderr para a *stream de entrada*, a *stream de saída* e a *stream de erros*, correspondendo ao teclado no caso da primeira e ao ecrã no caso das duas últimas.

Tipos de Ficheiros: Binários / Texto

Os ficheiros podem guardar a informação codificada de dois modos: texto e binário. No primeiro, o conteúdo do ficheiro é um conjunto de bytes agrupados de modo a criar texto, isto é, todos os bytes estão agrupados de modo a que correspondam a letras de texto e possam ser abertos por programas de edição de texto; no segundo caso o conteúdo de ficheiro tem uma organização que só é conhecida pelo programador, e consequentemente pelo programa, que criou o ficheiro o que implica que só pode ser aberto por quem criou o ficheiro.

O modo de texto é o modo que permite guardar informação de uma forma que facilita a leitura por outros programas ou por pessoas, por outro lado, o modo binário implica que a organização dos bytes não é conhecida o que torna a leitura do conteúdo mais difícil.

Para o programador, embora usando funções diferentes, o processo de escrita e leitura de um ficheiro é igual, quer no formato binário quer no formato de texto.

[1] *stream* é um termo inglês que pode ser traduzido para "fluxo", neste caso, um fluxo de dados.

Modos de Abertura de Ficheiros

Sempre que abrimos um ficheiro, quer o guardemos em disco ou não, é necessário indicar um o modo de abertura (leitura, escrita, actualização ou uma combinação das anteriores) bem como o formato no qual vamos escrever a nossa informação (texto ou binário).

r	Abertura para leitura. Não será possível escrever para este ficheiro nem guardar qualquer alteração. O ficheiro tem de existir, caso contrário será emitido um erro.
w	Abertura para escrita. Se o ficheiro existir, qualquer informação que contenha será destruída, caso contrário o ficheiro será criado vazio. Esta modo só permite a escrita e não a leitura dos dados no ficheiro.
a	Abertura para actualizar. Permite que seja adicionado conteúdo ao fim de um ficheiro existente. Se o ficheiro não existir será criado completamente vazio. Este modo não permite a leitura do conteúdo do ficheiro.
r+	Abre o ficheiro para leitura e escrita. Se o ficheiro não existir será emitido um erro.
w+	Abre o ficheiro para leitura e escrita. Se o ficheiro não existir será criado completamente vazio.
a+	Abertura para actualizar. Permite a leitura e escrita, se o ficheiro não existir será criado.
b	Este modo não pode ser usado isoladamente, apenas em conjunto com um dos seis modos anteriores. Ao ser aplicado a algum dos modos anteriores, altera a abertura do ficheiro de modo a que o mesmo seja aberto em formato binário, em oposição a uma abertura em modo de texto que é a opção por omissão.

Ao terminarmos a nossa utilização do ficheiro é importante que o mesmo seja fechado. *Nunca devemos deixar um ficheiro aberto depois do nosso programa terminar*. Se o fizermos, as estruturas necessárias para gerir o ficheiro serão mantidas em memória consumindo processamento e memória que é necessária ao normal funcionamento do sistema operativo e dos restantes programas. Existe também um limite para o número máximo de ficheiros que podem ser abertos no sistema operativo e deixarmos os nosso ficheiros abertos depois de terminarmos de os utilizar, esse número máximo pode ser atingido e impedir o funcionamento correcto do sistema operativo ou de outros programas.

Tratamento de Erros

A utilização de ficheiros está sempre sujeita à ocorrência de erros que afectam o programa, quer na abertura quer na restante manipulação dos dados. Dada a natureza dos ficheiros e a sua relação com o sistema operativo, um erro que não seja tratado correctamente poderá ter algumas consequências nefastas para o normal funcionamento do sistema.

Na abertura de ficheiros a função usada, *fopen* irá devolver o valor *NULL* se ocorrer algum erro. Se tal acontecer é necessário que o programa tome as medidas adequadas para garantir que o restante código funciona correctamente. Sendo que, quando ocorre um erro, não nos é devolvido um ponteiro para um ficheiro válido, não podemos executar qualquer código que faça uso do ficheiro.

As situações onde podem ocorrer erros a abrir ficheiros são: - Nome de ficheiro inválido, que contenha caracteres que não são possíveis de usar ou cujo nome resulte em alguma situação ilegal. - Abrir um ficheiro não fechado, e consequentemente em uso por outro programa. - Abrir um ficheiro que não

existe, esta situação é diferente do primeiro caso uma vez que o nome pode estar correcto mas o ficheiro não existir.

Durante a escrita de dados é possível que, devido a condições que tenham mudado após a abertura, surja erros. Entre os erros possíveis alguns dos mais comuns serão: - A quota de espaço do utilizador foi esgotada. - Foi feita uma tentativa para escrever uma quantidade de informação superior à permitida para o processo - A escrita foi interrompida por um sinal do sistema operativo. - Ocorreu um erro de I/O durante a escrita, por exemplo, o utilizador removeu o disco onde o ficheiro estava. - Não há mais espaço livre no disco.

O fecho de um ficheiro pode falhar, principalmente, se ocorrer algum erro na operação de escrita final.

Quase todas as funções que permitem trabalhar com ficheiros devolvem um valor de erro e todas elas, devolvam ou não um valor de erro, alteram um variável global, que já se encontra definida para nosso uso chamada *errno*. Esta variável pode ser usada para testar condições de erro em qualquer altura pois ela contém sempre o valor numérico do último erro que surgiu. Deve ser notado que a variável só regista o último erro detectado e que se ocorrerem vários erros na mesma operação ou se operações seguidas resultarem em erro, apenas o erro que ocorreu em último estará registado.

Um valor de zero na variável *errno* significa que nenhum erro foi detectado.

Memória Buffer

Como vimos na secção anterior, a escrita e leitura de ficheiros está muito dependente do sistema operativo e de condições externas ao nosso programa. Essas condições implicam também que a manipulação de ficheiros afecta o sistema operativo, forçando o mesmo a dedicar recursos cada vez que tentamos manipular os ficheiros. De forma a minimizar os vários impactos que a manipulação de ficheiros tem nos recursos da máquina, e de modo a que o processo possa ser optimizado, existem estruturas de memória chamadas *buffers*.

Um *buffer* [2] é uma zona de memória onde os dados que escrevemos são mantidos de forma temporária, isto é, quando escrevemos dados para um ficheiro o local onde os dados são guardados é, primeiro, num *buffer*. O sistema operativo só irá registar os dados em disco em intervalos periódicos e sempre pegando nos dados que estão em *buffer*.

Isto implica que a escrita de dados nunca é directa, apesar de usarmos funções que enviam os dados para o ficheiro, internamente o que o nosso sistema operativo faz é guardar esses dados em memória durante um período de tempo pré-determinado para evitar estar a aceder constantemente ao disco onde o ficheiro está guardado. Dado que o acesso a disco, seja um disco rígido, uma disquete, CD/DVD, *PEN* ou qualquer outro dispositivo de armazenamento permanente, é sempre mais lento que o acesso a memória, se não existissem *buffers* onde guardar a informação temporariamente, o processo de escrita ou leitura de dados de ficheiros seria extremamente lento.

A desvantagem é que existe um período de tempo em que, apesar de termos escrito os dados, a informação não está guardada no ficheiro e pode ser perdida devido a falhas no programa, falhas de corrente, ou qualquer outra falha que faça com que a informação em memória desapareça. Como ainda não foi escrita para um suporte permanente está num estado vulnerável.

Buffers de leitura fazem o mesmo efeito de optimização, permitindo ao sistema operativo obter blocos de dados de uma só vez e guardar esses dados em memória, mesmo que o programa só queira ler

[2] A tradução portuguesa de *buffer* não é usada. *Buffer* traduziria para *tampão*

alguns caracteres. Desta forma o acesso ao dispositivo de armazenamento permanente é minimizado e o processo de leitura torna-se mais amigável para os recursos do sistema.

Com *buffers* de leitura não existe o problema de se perderem dados como existe nos *buffers* de escrita já que a informação que neles é colocada é apenas uma cópia da informação que está disponível no ficheiro.

Funções de Manipulação de Ficheiros

A lista seguinte pretende apresentar as funções de manipulação de ficheiros que são fornecidas com a biblioteca padrão da linguagem de programação C. Estas funções podem ser substituídas por funções vindas de outras bibliotecas mas o funcionamento base será similar.

Tabela 5.1. Manipulação Geral

fopen	Permite abrir o ficheiro no modo especificado.
freopen	Fecha um ficheiro anteriormente aberto.
fclose	
fflush	Força a gravação da informação, esvaziando o *buffer* e gravando os dados no suporte físico do ficheiro.
setbuf	Define um *buffer* a ser usado pelas operações de leitura e escrita.
sevbuf	Define um *buffer* a ser usado pelas operações de leitura e escrita.

Tabela 5.2. Posição de Cursor

fgetpos	Obtém a posição actual do cursor.
fsetpos	Define a posição actual do cursor.
fseek	Movimenta o cursor dentro do ficheiro. Podem ser usadas algumas constantes como *SEEK_SET*, *SEEK_CUR* e *SEEK_END*.
ftell	Indique a posição inicial para escrita ou leitura.
rewind	Coloca o cursor no início do ficheiro.

Tabela 5.3. Escrita e Litura de Dados

fwrite	Escreve um conjunto de informação no ficheiro.
fread	Lê um conjunto de informação do ficheiro.

Tabela 5.4. Escrita e Leitura Formatada de Texto

fscanf	Efectua a leitura formatada de dados num ficheiro aberto em modo de texto.
fprintf	Efectua a escrita formatada de dados num ficheiro aberto em modo de texto.

Tabela 5.5. Escrita e Leitura de Caracteres

getc	Permite ler um carácter. Esta função é genérica e pode ser usada com ficheiros ou com o teclado.

fgetc	Permite a leitura de um carácter a partir de um ficheiro.
fgets	Permite a leitura de uma *String* a partir de um ficheiro.
putc	Escreve um carácter para o ficheiro. Tal como a função *getc* é genérica e pode ser usada com mais coisas que apenas ficheiros.
fputc	Escreve um carácter para um ficheiro.
fputs	Escreve uma string para um ficheiro.

Tabela 5.6. Tratamento de Erros

feof	Verifica se atingimos o fim do ficheiro.
clearerr	Limpa a variável de controlo de erros.
ferror	Verifica se a última leitura ou escrita provocou um erro.

O uso de um ficheiro segue quatro passos genéricos: abrir o ficheiro, escrever a informação necessária, garantir o registo da informação periodicamente e fechar o ficheiro. Estes passos são, naturalmente, expandidos para se incluir a validação de erros, a detecção de situações anómalas, e qualquer outro código necessário à correcta gravação dos dados.

Todas as funções de leitura e escrita fazem uso de um *cursor* que indica a posição onde estamos a ler ou a escrever dados. Uma vez que todas as operações são sequenciais, provocando o avanço do *cursor*, caso seja necessário voltar a ler algum dado que já foi lido, ou escrever sobre algum dado que já tenha sido escrito, é preciso usar funções para recolocar o cursor no local certo.

O uso de ficheiros requer também a utilização de estruturas próprias, constantes e tipos de dados que estão disponíveis no ficheiro *header stdio.h*. Após inclusão deste ficheiro teremos à nossa disposição todas estruturas necessárias.

Como em muitas outras situações, poderão existir mais funções de leitura e escrita de ficheiros ou mesmo funções de manipulação geral que não estão listadas nem são apresentadas neste manual. Em regra geral, essas funções extra são dependentes do compilador e não fazem parte do *standard* da linguagem de programação C, como tal estarão fora do âmbito deste manual.

fopen

Função usada para abrir um ficheiro. A função recebe dois parâmetros: o **nome do ficheiro** e o **modo de abertura**; e devolve um ponteiro, do tipo *FILE*, que pode depois ser usado em outras funções. Em caso de erro devolve o valor *NULL*.

```
FILE * fopen(const char *ficheiro, const char *modo);
```

Por omissão, todos os ficheiros são abertos em modo de texto, se precisarmos indicar outro modo devemos adicionar a letra **b** como vímos na secção referente aos modos de abertura.

freopen

A função efectua um fecho e uma nova abertura do ficheiro. Primeiro quaisquer dados que existam no *buffer* são escritos para o ficheiro, algum erro que ocorra na escrita é ignorado, mesmo que disso resultem perdas de dados. Se o nome do ficheiro não for uma *String* válida a função tenta alterar o modo passado de forma a criar o ficheiro, no entanto esta opção é dependente do sistema operativo.

FILE *freopen(const char *nome_ficheiro, const char *modo, FILE *ficheiro);

fclose

Permite fechar um ficheiro que tenha sido aberto com sucesso. A tentativa de fecho de um ficheiro irá provocar um acesso de escrita para gravar os dados que ainda existam no *buffer*, por este motivo a função de fecho pode devolver erros que estão associadas à escrita de dados.

```
int fclose(FILE *ficheiro)
```

fflush

Força a gravação de dados para o ficheiro, esvaziando o *buffer* e pedindo ao sistema operativo que grave os dados no suporte físico onde o ficheiro está guardado.

```
fflush(FILE *ficheiro)
```

Após uma operação de *flush* o ficheiro continua aberto e pronto a ser usado.

setbuf

Permite definir um *buffer* a ser usado pelas funções de leitura e escrita seguintes. Se em vez de um *buffer* for passado o valor *NULL* então as funções de escrita e leitura usadas a seguir deixam de fazer uso de qualquer *buffer*.

```
void setbuf(FILE *ficheiro, char *buffer)
```

Esta função deve ser usada antes de ser feita qualquer escrita ou acesso ao ficheiro.

sevbuf

Tal como a função anterior, esta função permite definir um buffer a ser usado pelas funções de escrita e leitura. Oferece mais opções que a função *setbuf* e pode fazer uso de um *buffer* previamente configurado pela função *setbuf*.

- {*IOFBF} fará com que as operações de escrita e leitura façam uso completo do _buffer*;
- {*IOLBF} fará com que as operações de escrita e leitura façam _buffer* de linhas de informação;
- {*IONBF} fará com que as operações de escrita e leitura deixem de usar _buffer*;

```
int setvbuf(FILE *ficheiro, char *buffer, int tipo, size_t tamanho)
```

fgetpos

Obtém a posição do cursor de escrita ou leitura dentro do ficheiro. Depois de obtermos a posição do cursor podemos usar esta posição em conjunto com outras funções que permitam recolocar o cursor, permitindo assim navegar no ficheiro.

```
int fgetpos(FILE *ficheiro, fpos_t *pos)
```

fsetpos

Com esta posição podemos colocar o cursor numa posição previamente guardada e obtida através da função *fgetpos*. Usando estas duas funções podemos ir guardando posições para as quais pretendemos voltar e navegar no ficheiro, no entanto, para se conseguir usar a função *fsetpos* é necessário ter uma posição válida obtida com a função *fgetpos* o que obriga a percorrer o ficheiro de forma sequencial, pelo menos uma vez.

```
int setpos(FILE *ficheiro, fpos_t *pos)
```

fseek

Tal como a função anterior, a função *fseek* permite posicionar o cursor do ficheiro numa posição definida por nós mas ao contrário da função *fsetpos* não fazemos uso de posições previamente obtidas. Esta função usa um valor de deslocamento a partir de um ponto que fornecemos. Podemos também usar 3 macros que estão definidas e que correspondem ao início do ficheiro, à posição actual e ao fim do ficheiro, respectivamente SEEK_SET, SEEK_CUR e SEEK_END.

```
int fseek(FILE *ficheiro, long deslocamento, int ponto_inicial)
```

Se, por exemplo, quisermos deslocar o cursor para trás partindo da posição actual, podemos usar um deslocamento negativo e usando como posição inicial a macro SEEK_CUR.

ftell

Esta função é similar à função *fgetpos* mas devolve a posição como um valor do tipo *long*. É por isso útil em conjunto com a função *fseek*, podendo o valor devolvido por esta função ser usado como posição inicial a partir da qual se faz um deslocamento.

```
long ftell(FILE *ficheiro)
```

rewind

Permite colocar o cursor do ficheiro na sua posição inicial, ficando a apontar para o princípio do ficheiro. Esta função não devolve qualquer valor de erro e por isso torna-se mais difícil detectar algum erro que surja durante a operação.

```
void rewind(FILE *ficheiro)
```

O processo mais comum para detectar um erro provocado durante a operação de reabertura é definir a variável *errno* a zero, invocar a função e testar a variável *errno* novamente para ver se contém um valor diferente de zero.

fwrite

Permite escrever um conjunto de elementos de tamanho igual para um ficheiro. Esta é a primeira de duas funções de escrita e leitura de dados binários, e como é possível ver não permite especificar qualquer formato, posição ou outra opção que altere o aspecto do que é escrito.

A função escreve, em binário, os elementos indicados baseando-se no seu tamanho, daí terem todos de ser de tamanho igual.

```
size_t fwrite(void *dados, size_t tamanho, size_t quantidade, FILE *fp)
```

O cursor vai avançar uma distância correspondente a **tamanho x quantidade**.

A estrutura *size_t* é usada sempre que nos referimos ao tamanho de dados em memória e corresponde a um *inteiro sem sinal*.

fread

Permite a leitura de um bloco de dados a partir de um ficheiro. Esta função, tal como a função anterior, apenas lida com dados em binário e converte os dados a partir do formato binário do ficheiro. Tal como na função *fwrite* é preciso indicar quantos dados se vão ler e o seu tamanho, e todos os dados lidos têm de ser do mesmo tamanho.

```
size_t fread(void *dados, size_t tamanho, size_t quantidade, FILE *fp)
```

fscanf

Permite ler dados formatados a partir do ficheiro. Esta função é igual à função *scanf*, acrescentando apenas um parâmetro novo onde se indica o ponteiro para o ficheiro que vai ser usado. Tal como a função *scanf* permite a leitura formatada de informação e partilha com esta função os mesmos especificadores.

```
int fscanf (FILE *ficheiro, const char *formato, ... )
```

fprintf

Permite a escrita formatada de dados para um ficheiro. Esta função é igual à função *printf* e acrescenta, tal como aconteceu na função anterior, um parâmetro que é usado para especificar o ficheiro para onde vão ser escritos os dados.

```
int fprintf (FILE *ficheiro, const char *formato, ... )
```

getc

Obtém o próximo carácter que se encontra na ficheiro. Esta função recebe um ponteiro para ficheiro e devolve apenas um carácter, que corresponde ao próximo carácter presente no ficheiro a seguir ao cursor. Esta função pode ser implementada como uma macro.

```
int getc (FILE *ficheiro)
```

fgetc

Obtém o próximo carácter que se encontra na ficheiro. A função é igual à função *getc* mas, ao contrário da função *getc*, não pode ser implementada como uma macro.

```
int fgetc (FILE * ficheiro)
```

fgets

Permite ler uma *String* de um ficheiro. Esta função lê todos os caracteres até encontrar um *\n* ou ter atingido o número máximo de caracteres a ler. O *\n* é incluído na *String*.

```
char * fgets (char *texto, int máximo, FILE *ficheiro)
```

putc

Escreve um carácter para o ficheiro. Tal como a função *getc*, a função *putc* pode ser implementada como uma macro.

```
int putc (int carácter, FILE *ficheiro)
```

fputc

Escreve um carácter para o ficheiro. Igual à função *putc* mas não podendo ser implementada como uma macro.

```
int fputc (int carácter, FILE *ficheiro)
```

fputs

Escreve uma *String* para o ficheiro. Esta função faz para com a função *fgets* e permite escrever todos os caracteres que constituem a *String* excepto o carácter terminador, *\0*, que não é gravado no ficheiro.

```
int fputs (const char *texto, FILE *ficheiro)
```

feof

Indica se chegámos ao fim do ficheiro, devolvendo um valor diferente de zero em caso afirmativo. Esta função tem uma particularidade: só detecta o fim do ficheiro depois de ter sido feita uma leitura falhada, isto é, só depois de passarmos o fim do ficheiro é que a função pode ser usada e confirmar que, efectivamente, estamos no fim.

```
int feof(FILE *ficheiro)
```

Embora esta utilização possa parecer inútil, afinal a função de leitura já nos indica um erro quando tentamos ler após o fim do ficheiro, a verdade é que a função de leitura pode falhar por várias razões. O usdo desta função de teste permite confirmar se a função de leitura emitiu um erro porque chegámos ao fim do ficheiro e não porque ocorreu outra situação inesperada.

clearerr

Caso ocorra um erro, esta função limpa a variável que regista o número de erro.

```
void clearerr(FILE *ficheiro)
```

Todas as funções que emitam erros fazem-no de duas formas: devolvendo um valor e alterando uma variável chamada *errno*. Com esta função podemos limpar o valor desta variável.

ferror

Esta função permite verificar se a última instrução de litura ou escrita provocou um erro. O valor devolvido é zero se não existiu erro nenhum, e um inteiro diferente de zero se existiu algum erro.

```
int ferror(FILE *ficheiro)
```

Streams

Todos os conceitos que foram introduzidos neste capítulo são transversais a todas os *Streams*.

Stream, que em português poderia ser traduzido para *fluxo*, representam fluxos de dados, quer origens quer destinos, onde podemos ler e escrever informação. Se pensarmos no conceito de ficheiro, não interessa se estamos a escrever o ficheiro num CD, num DVD, numa *PEN*, numa disquete, etc. Não é importante o destino físico onde os dados vão ser armazenados já que as formas de leitura e escrita são as mesmas.

De forma similar, a escrita de dados através de uma placa de rede, ou através de USB, ou mesmo através de um *MODEM* de acesso à *Internet* não depende do suporte físico (cabos de cobre, wireless, etc) nem do percurso que os dados fazem. Se tornarmos a forma de escrever e ler informação um mecanismo que siga regras comuns podemos tornar todo o processo de leitura e escrita de dados algo simples e fácil de adaptar.

Foi nesse sentido que surgiu o conceito de *Streams*, através do qual não pensamos na implementação física do suporte onde os dados são guardados mas apenas nas operações básicas que se lhes podem aplicar: abrir *stream*, escrever informação, garantir registo periódico da informação, fechar *stream*.

Sabendo então que estamos a lidar com um conceito comum, podemos reler as secções deste capítulo e substituir todas as ocorrências da palavra *ficheiro* por *stream*. As regras, funções e métodos usados são os mesmos, quer estejamos a trabalhar com ficheiros em disco localmente, quer estejamos a trabalhar com ficheiros em memória ou até com *streams* puras que não existem em qualquer disco e que representam apenas a comunicação entre dois programas, ou de forma genérica a comunicação (o fluxo de dados) entre dois pontos.

Bibliografia

- Richard L. Petersen. *Introductory C, Second Edition: Pointers, Functions, and Files*. Morgan Kaufmann. 7 de Novembro, 1996. ISBN 978-0125521420.