INSTITUTO SUPERIOR DE CIÊNCIAS DO TRABALHO E DA EMPRESA **Sistemas Operativos** **ETI - IGE**	2º Semestre 2005/2006 Ficha 3

C/C++; biblioteca stdio

1. A biblioteca `stdio` engloba o conjunto de funções que são normalmente usadas na linguagem C para I/O. Podem-se usar, praticamente da mesma forma, em C, C++ e, de forma muito semelhante, na generalidade dos compiladores e variantes do Unix.

Para incluir as declarações associadas à biblioteca inclui-se o ficheiro `stdio.h`

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

2. A função comum de escrita no ecrã é o printf que tem a seguinte forma geral:

```
printf( formato, argumentos...)
```

O *formato* é na realidade a mensagem que vai ser escrita no ecrã quando a função for chamada.

Exemplo: O seguinte código permite escrever no ecrã a mensagem `Hello World` (terminando com uma mudança de linha, provocada pela escrita do caracter \n).

```
#include <stdio.h>
main()
{
    printf ("Hello World\n");
}
```

Exemplo: qual o efeito dos seguintes `printf` ?

```
printf ("Hello\nWorld\n");
printf ("Hello \n\n World\n");
printf ("\nHello\n World");
```

3. Na realidade, a maior parte das vezes a função será usada para escrever no ecrã valores das variáveis do programa. Para escrever o valor de uma variável com o `printf` é preciso conjugar duas coisas:

- Assinalar essa intenção na *mensagem* (primeiro argumento)
- Veicular a variável como argumento adicional

Por exemplo:

```
int i = 10;
printf( "Aqui vai aparecer o valor de i que é %d e depois segue\n", i );
```

Neste caso o printf tem dois argumentos. O que vai ser escrito é (sempre!) o primeiro argumento. Acontece que ao, escrever o primeiro argumento, se encontrar uma sequência especial como é o caso de

```
%d
```

Substitui essa sequência pelo valor da variável indicada no 2º argumento. Resultado, aparece a mensagem:

```
Aqui vai aparecer o valor de i que é 10 e depois segue
```

4. O mesmo esquema aplica-se para mais que uma variável: as sequências especiais inscritas na mensagem são substituídas, por ordem, pelas variáveis indicadas nos argumentos adicionais do `printf`. Exemplo:

```
int i = 10, k = 35
printf( "Uma valor é %d e o outro é %d\n", i, k );
```

Ou seja, é preciso que o número de sequências especiais inscritas na mensagem seja igual ao número de argumentos que são dados ao `printf` além do primeiro.

5. Na realidade os argumentos não precisam de ser variáveis simples; pode, ser, como em geral na linguagem C/C++ expressões. Por exemplo:

```
int i = 10, k = 35
printf( "%d+%d=%d", i, k, i+k );
```

6. As sequências especiais, ditas sequências de formatação, indicam o tipo da variável. Se a sequência de formatação for `%d`, é suposto que a variável correspondente seja `int`. Da mesma forma verificam-se as seguintes correspondências:

```
%d   - int
%ld  - long
%c   - caracter
%s   - string (array de caracteres)
%f   - float
%lf  - double (float "longo")
...
```

Exemplo:

```
int i = 79;
float f = 10.5;
string *str = "Hello";
printf ("<%d><%f><%s>\n", i, f, str);
printf ("<%c><%d>\n", i, i );
printf ("<%f><%d>\n", f, (int) f);
printf ("<%c><%d>\n", str[0], str[0]);
```

7. É claro que, se as sequências de formatação não estiverem de acordo com os tipos das variáveis correspondentes, podem ser escritos os maiores disparates.
Por exemplo:

```
int i = 79;
float f = 10.5;
printf ("<%d><%f>\n", f, i);
```

Ou seja, a sequências de formatação inscritas na mensagem e os correspondentes argumentos do `printf`, devem concordar quer em número quer em género.

8. As sequências de formatação podem ser enriquecidas com elementos que, justamente, servem para controlar a forma como os valores das variáveis são escritos no ecrã (ou seja, formatar).
Por exemplo:

`%3d`	escreve o valor do número em, pelo menos, 3 posições.
`%16.2f`	escreve um `float` com 16 posições, 2 para decimais;
`%10s`	escreve uma `string` em 10 posições;

Exercício: faça um programa que escreva os números de 1 a 100, 10 em cada linha, alinhados à direita em cada coluna.

9. Nada disto é segredo de estado., pelo contrário. No UNix as funções standard do C são directamente documentadas. Se quiser saber mais pormenores sobre esta outras funções pode fazer o comando:
`man 3 printf`

10. O `printf` é na realidade uma versão específica de uma função mais geral fprintf. O fprintf tem a seguinte forma geral:

```
fprintf ( canal , formato, argumentos...)
```

A diferença para o `printf` é apenas o primeiro argumento que indica o canal de escrita. No caso do printf o canal de escrita é omitido: é, por construção, o canal standard de saída do programa (normalmente o ecrã). Este canal standard de saída é descrito pela variável stdout (declarada por inclusão do `stdio.h`) que pode também ser usada no `fprintf`. Ou seja, o `printf`

```
printf ( formato, argumentos...)
```

é o mesmo que

```
fprintf(stdout, formato, argumentos...)
```

10. A função de leitura correspondente ao `printf` é o `scanf`, que tem uma estrutura de argumentos parecida:

```
scanf ( argumentos, variáveis...)
```

O primeiro argumento indica o que se pretende ler, através de sequências de formatação parecidas com o `printf`. Os restantes argumentos indicam o endereço das variáveis que vão receber os valores lidos.

11. Um exemplo típico é a leitura de um número. Exemplo:

```
main()
{
    int n;
    printf ("Diga um número: ");
    scanf ("%d", &n );
    printf ("O número seguinte é: %d \n", i);
}
```

O primeiro `scanf` indica que se está à espera de ler um número inteiro. O segundo argumento denota que o número inteiro lido deve ser colocado na variável n. Para isso é indicado como argumento o endereço da variável n (um pointer para a variável n).

O comportamento do `scanf` é o seguinte: ultrapassa os separadores (espaços, mudanças de linha), lê todos os caracteres que possam constituir um número (ou seja, os algarismos) e termina no primeiro que não seja.

Exemplo, experimente o programa anterior com as seguintes entradas:

```
12      seguido de enter[1]
12xxx
123456789
   123
x123
```

Experimente ainda mudar de linha antes de dar o número.

12. O `scanf` não garante que a entrada seja conforme ao que se espera (ou seja, não faz validações !). Por exemplo, na situação anterior a entrada x123 faz com que, na realidade, não seja lido qualquer número.

13. Além do %d aplicam-se as sequências semelhantes ao `printf`, sendo as mais comuns:

%s	palavra
%c	um caracter
%f	um número decimal

Note que o `%s` lê apenas uma palavra (ou seja, termina no primeiro espaço ou mudança de linha encontrado). Por exemplo:

```
main()
```

[1] tem, seja com for, que terminar a entrada com enter; antes disso os caracteres escritos ficam apenas no buffer de entrada - não chegam ao programa.

```
{
    int n;
    printf ("Diga o seu nome completo: ");
    scanf ("%s", s );
    printf ("Disse \n", i);
}
```

Note ainda que, neste caso, o s é dado sem &. É normal: trata-se de um array de caracteres; o s é ele próprio um endereço (o do primeiro elemento do array).

14. Raramente se usa o `scanf` com sequências de formatação mais elaboradas que as anteriores. Mas o `scanf` permite uma formatação mais geral perecida (numa óptica de leitura) com a do scanf. A formatação do `scanf` indica o padrão que é suposto obedecer a sequência de caracteres de entrada: a leitura só é bem sucedida se a entrada obedecer a esse padrão.

A indicação de um carácter comum na sequência de formatação indica que é suposto a entrada veicular esse mesmo carácter.

Exemplo: experimente o seguinte programa e indique uma sequência de entrada que permita atribuir um número à variável n.

```
main()
{
    int n;
    printf ("Diga um número: ");
    scanf ("XXX%d", &n );
    printf ("O número seguinte é: %d \n", i);
}
```

Pode também ser usado um repetidor. Exemplo: experimente o seguinte programa

```
main()
{
    int n;
    char s[100];
    printf ("Diga um número: ");
    scanf ("%3c%d", s, &n );
    printf ("O número seguinte é: %d \n", i);
}
```

e indique uma sequência de entrada que permita atribuir um número à variável n.

O símbolo * permite ultrapassar caracteres (isto é, ler mas descartar).
Exemplo: experimente o seguinte programa

```
main()
{
    int n;
    printf ("Diga um número: ");
    scanf ("%*3c%d", &n );
    printf ("O número seguinte é: %d \n", i);
}
```

15. O exemplo do `printf`, o `scanf` é uma versão da função mais geral `fscanf`. O `scanf` lê de uma canal standard de entrada (normalmente o teclado) descrito pela variável stdin. Ou seja:

```
fscanf ( formato, variáveis...)
```

é o mesmo que

```
fscanf ( stdin, formato, variáveis...)
```

16. Além do `scanf` à outras funções de leitura tanto ou mais utilizadas. Uma delas é o

```
getchar()
```
ou
```
fgetc(stdin)
```
que lê e devolve um caracter.

Exemplo:
```
#include <stdio.h>
main()
{
    printf ("Carregue em ENTER para continuar: ");
    getchar();
}
```

O caracter lido é devolvido pela função. Exemplo:
```
#include <stdio.h>
main()
{
    int c;
    printf ("Carregue em ENTER para continuar: ");
    c=getchar();
    if ( c != '\n' )
        printf ("Era ENTER não era %c\n", c);
}
```

17. Considere o seguinte exemplo:
```
#include <stdio.h>
main()
{
    int c;
    while ( getchar() != EOF )
        ;
}
```

Este lê todos os caracteres da entrada até ao "fim do ficheiro" (interactivamente o fim de ficheiro pode ser dado com CTRL-D). Neste caso o fim de ficheiro é sinalizado pela própria função getchar() que devolve a constante EOF (definida em stdio.h) nessas circunstâncias.

O mesmo efeito se pode obter através da função feof:
```
#include <stdio.h>
main()
{
    int c;
    c = getchar();
    while ( ! feof(stdin) )
    {
        c=getchar();
    }
}
```

que devolve `true` (seja 1) quando se verificar o fim de ficheiro, neste caso no stdin.

18. Exemplo: o seguinte exemplo implemente um `cat` rudimentar
```
#include <stdio.h>
main()
{
    int c;
    c = getchar();
    while ( ! feof(stdin) )
    {
```

```
            putchar(c);
            c=getchar();
        }
    }
```

Use este programa para fazer um comando que permita:
i) ver o conteúdo do ficheiro `/etc/passwd`
ii) criar, no seu directório de trabalho, um ficheiro com algumas palavras
iii) copiar o ficheiro `/etc/passwd` para o seu directório de trabalho;

19. As várias funções de leitura podem ser misturadas. Seja como for, relativamente à leitura, funciona sempre a seguinte lógica: a entrada é uma sequência de caracteres, que têm que ser consumidos por ordem; assim, cada operação de leitura consuma 0 ou mais caracteres; a próxima operação recomeça no carácter seguinte (o primeiro carácter que a operação anterior não consumiu).

Exemplo, considere o seguinte programa ao qual se dá como entrada: `12` (seguido de enter).

```
main()
{
    int n, c;
    printf ("Diga um número: ");
    scanf ("%d", &n );
    printf ("Disse %d\n", &n);
    printf ("Carregue em ENTER");
    getchar();
}
```

Contrariamente ao que a as mensagens sugeriam, após a entrada do número o programa não fica à espera. A leitura do número pelo `scanf` consome o 12 mas não o enter: logo o `getchar` não tem que esperar por um novo carácter: prossegue imediatamente lendo o enter.

20. Outra função importante de leitura é a função `gets()` ou `fgets()`. A variante `gets` tem a forma

```
gets(s)
```

Sendo s uma string (array de caracteres). A variante `fgets` tem a forma:

```
fgets(s, dimensao, stdion)
```

Sendo dimensão uma expressão que indica o número máximo de caracteres a ler.

A segunda forma é a mais adequada porque permite evitar a leitura de mais caracteres do que a dimensão do array de destino permite. Por exemplo:

```
main()
{
    int s[21];
    printf ("Diga uma frase (máx 20 caracteres): ");
    fgets(s, 21, stdin):
    printf ("Disse %sd\n", s);
}
```

Note que, neste caso, a função lê uma linha completa, até ao ENTER (a menos que o limite seja ultrapassado). É portanto a função adequada para ler strings que possam ter espaços no meio.