

Algoritmia e Estruturas de Dados

Módulo II Erros de execução e Debugging



1. Erros de sintaxe
2. Erros de lógica
3. Erros de execução
4. Tratamento de exceções: a estrutura Try-Catch
5. Debugging

Erros de sintaxe



Ocorrem na fase de escrita de código do programa

Resultam de instruções escritas de forma incorrecta/incompleta

Impedem a execução/teste do programa

```
static void adicionar_dados()
{
    Console.Cler();
    Console.Write("numero:");
    strin numero = Console.ReadLine();
    Console.Write("nome:");
    string nome = Console.ReadLine();
    string linha_texto = numero + ";" + nome;
    GlobalVars.array_linhas[GlobalVars.pos] = linha_texto;
    GlobalVars.pos = GlobalVars.pos + 1;
}
```

100 %

Error List

2 Errors 0 Warnings 0 Messages

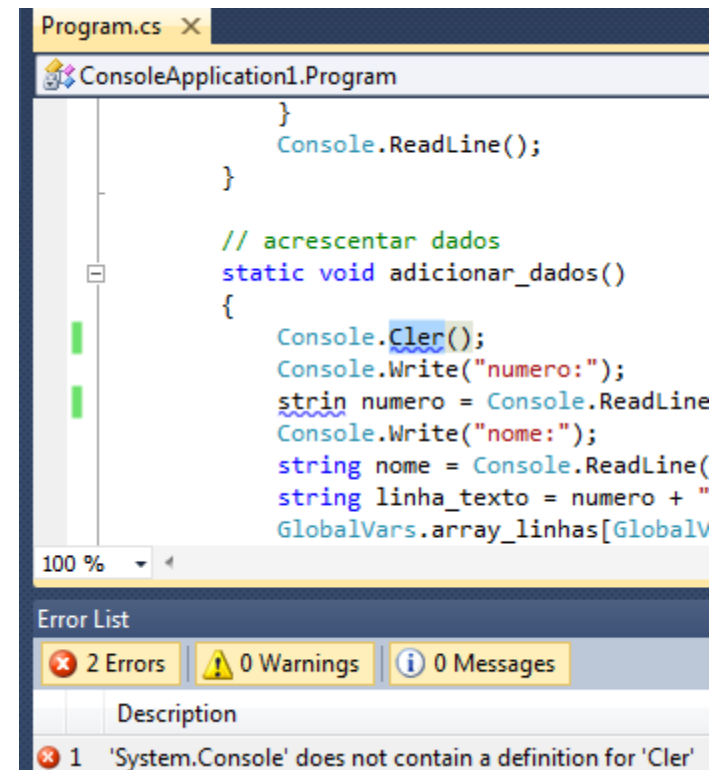
	Description	File	Line	Column
1	'System.Console' does not contain a definition for 'Cler'	Program.cs	100	21
2	The type or namespace name 'strin' could not be found (are you missing a using directive or an assembly reference?)	Program.cs	102	13

Erros de sintaxe



Nestes casos, o compilador não consegue converter as instruções que contêm erros de sintaxe em linguagem intermédia (MSIL, *Microsoft Intermediate Language*), pelo que a sua execução não é possível.

Duplo clique numa linha da *Error List* posiciona-o na linha onde o erro ocorre, na sua folha de programação .cs.



Erros de lógica



Ocorrem durante a execução do programa, quando o programa não se comporta exactamente da forma que seria de esperar.

Também designados por erros de semântica.

Resultam de um comportamento ou de um resultado inesperado do programa.

Alguns erros de lógica podem ser mais ou menos evidentes (um ciclo while que nunca termina; uma operação aritmética executada com a variável errada, etc.).

Erros de lógica



Outros erros de lógica podem ser muito difíceis de descobrir, permanecendo ocultos durante dias/meses ou anos.

Ou até podem nunca ser descobertos. pois podem ocorrer apenas em situações muito específicas ou particulares.

Exemplo:

Programa que deve dar um determinado alerta quando a temperatura ultrapassa os 40°C. Imaginemos que em determinada altura essas condições se verificam e o programa afinal não dá o tal alerta! - **erro de lógica**.

O **Debugging** (abordado mais à frente) é uma ferramenta muito útil para a detecção de erros de lógica.

Erros de execução



Ocorrem durante a execução do programa, quando o programa gera mensagens de erros inesperadas ou indesejadas.

É frequente este tipo de erros levar a que o programa deixe de funcionar, podendo crachar!

Podem ser provocados por **factores externos**

- um ficheiro que não existe,
- uma pasta que está com uma path diferente,
- falta de permissões de acesso, etc...

ou **factores internos** ao programa

- uma divisão por zero
- Uma operação aritmética com uma variável que não tem qualquer valor
- Etc...

Erros de execução



O programador pode/deve tentar antecipar a possibilidade deste tipo de erros ocorrer. E implementar código no programa que possa prever a sua ocorrência e respectiva resolução.

O objectivo não é impedir que este tipo de erros de execução ocorra! É antes o de tratar estas eventuais situações da forma mais adequada.

Por exemplo: se tentamos abrir um ficheiro que não existe: o programa deve prever essa situação, dando uma mensagem apropriada ao utilizador, indicando qual o erro que ocorreu.

Erros de execução



Em C# um erro de execução é designado por **Exception** (Excepção).

Todas as exceções são representadas por classes derivadas da classe *Exception*.

Quando uma exceção é "apanhada" (caught, / catch), o programa deve incluir o código que a permite tratar.

Erros de execução



Exceções *Exception* podem ser tratadas com a declaração **try... catch**

```
try
{
    // código que pode causar a exceção
}
catch (TipoExceçãoA varA)
{
    //tratamento da exceção
}
catch (TipoExceçãoB varB)
{
    //tratamento da exceção
}

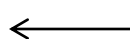
...

finally
{
    // código que é executado incondicionalmente
}
```

try

Código executado até ocorrer uma exceção.

No caso de não ocorrer nenhuma exceção, quando este bloco de código terminar passa à clausula **finally** (se existir, pois é opcional).



```
try
{
    // código que pode causar a exceção
}
catch (TipoExceçãoA varA)
{
    //tratamento da exceção
}
catch (TipoExceçãoB varB)
{
    //tratamento da exceção
}
...
finally
{
    // código que é executado incondicionalmente
}
```

Catch

Pode ter ou não argumentos.

Sem argumentos: trata qualquer exceção (erro) que possa ocorrer.

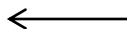
Com argumentos: trata a exceção específica, definida como argumento: essa exceção deve ser um objecto da Classe System.Exception

```
try
{
    // código que pode causar a exceção
}
catch (TipoExcepçãoA varA)
{
    //tratamento da exceção
}
catch (TipoExcepçãoB varB)
{
    //tratamento da exceção
}
...
finally
{
    // código que é executado incondicionalmente
}
```

Catch

Se existirem vários blocos catch, a sua ordem é importante. Eles são testados pela ordem que aparecem no código.

Assim, devemos descrever exceções da mais específica para a mais genérica.



```
try
{
    // código que pode causar a exceção
}
catch (TipoExceçãoA varA)
{
    //tratamento da exceção
}
catch (TipoExceçãoB varB)
{
    //tratamento da exceção
}
...
finally
{
    // código que é executado incondicionalmente
}
```

Finally

Bloco opcional.

Se existir, o código associado à clausula **finally** é executado de forma incondicional, isto é, quer seja ou não lançada uma exceção.

```
try
{
    // código que pode causar a exceção
}
catch (TipoExcepçãoA varA)
{
    //tratamento da exceção
}
catch (TipoExcepçãoB varB)
{
    //tratamento da exceção
}
...
← finally
{
    // código que é executado incondicionalmente
}
```

Exemplo

Exceção de formato

Execução do programa



```
file:///C:/Users/Beta/Desktop/Try-Catch/ConsoleA
Numero: g
valor incorrecto
```

```
file:///C:/Users/Beta/Desktop/Try-Catch/ConsoleA
Numero: 3
programa continua por aqui...
```

```
static bool ler_numero()
{
    try
    {
        Console.SetCursorPosition(9, 3);
        int num = int.Parse(Console.ReadLine());
    }
    catch (System.FormatException)
    {
        return false;
    }
    return true;
}

static void Main(string[] args)
{
    Console.SetCursorPosition(1, 3);
    Console.Write("Numero:");
    while (ler_numero() == false)
    {
        Console.WriteLine("valor incorrecto");
    }

    Console.WriteLine("programa continua por aqui...");
    Console.ReadLine();
}
```

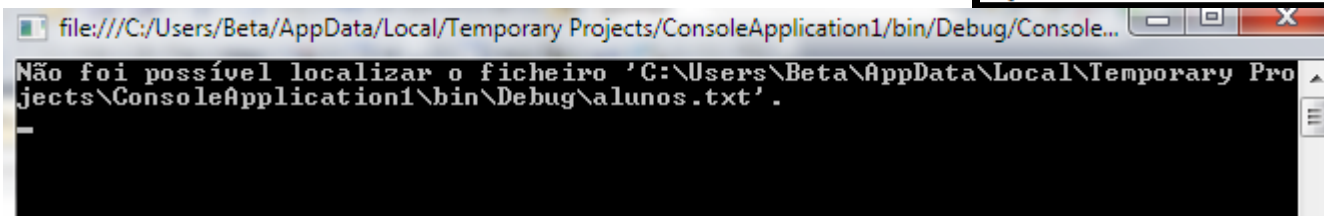
Exemplo

Capturar o erro que ocorreu

Mostra a mensagem de erro
capturada

```
static void ler_ficheiro()
{
    StreamReader sr;
    try
    {
        sr = File.OpenText("alunos.txt");
    }
    catch (System.Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

static void Main(string[] args)
{
    ler_ficheiro();
    Console.ReadLine();
}
```



Exemplo

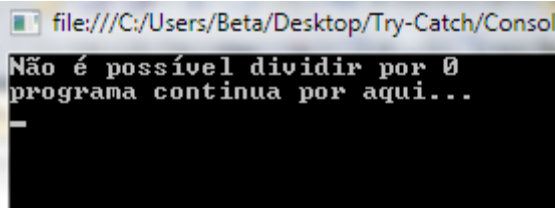
Descreve dois blocos catch.
O primeiro mais específico
(divisão por zero), o
segundo mais genérico



```
static void calcula_percentagem(int valor, int total)
{
    try
    {
        int percentagem = valor / total;
    }
    catch (System.DivideByZeroException)
    {
        Console.WriteLine("Não é possível dividir por 0");
    }
    catch (System.Exception)
    {
        Console.WriteLine("impossível calcular a percentagem");
    }
}

static void Main(string[] args)
{
    int valor = 10;
    // total inicializado a 0 propositadamente, de forma a ocorrer um erro!
    int total=0;
    // procura o primeiro ; numa linha de texto
    calcula_percentagem(valor, total);

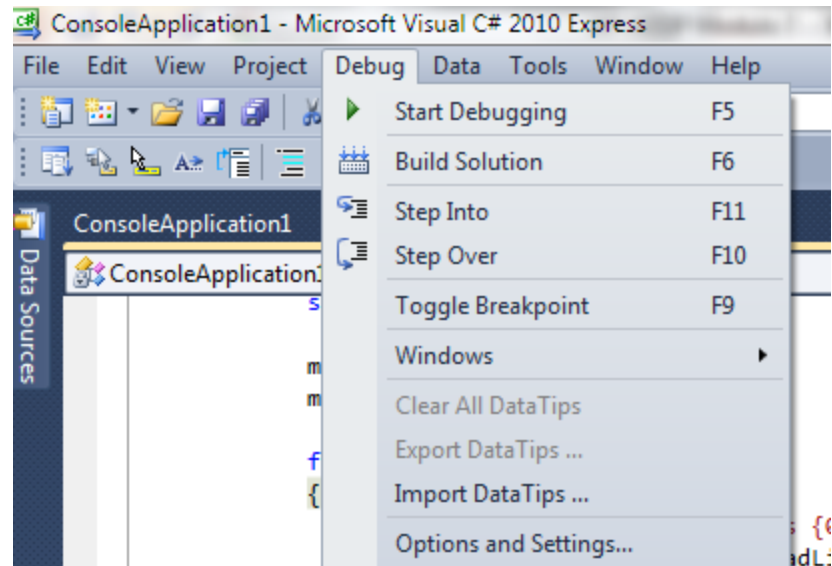
    Console.WriteLine("programa continua por aqui...");
    Console.ReadLine();
}
```



file:///C:/Users/Beta/Desktop/Try-Catch/Consol
Não é possível dividir por 0
programa continua por aqui...
-

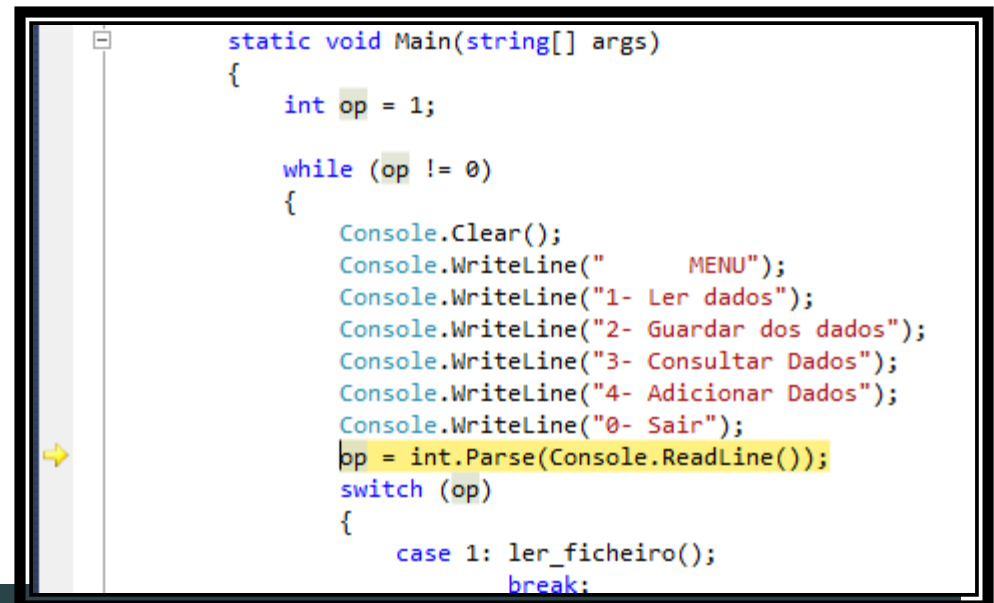
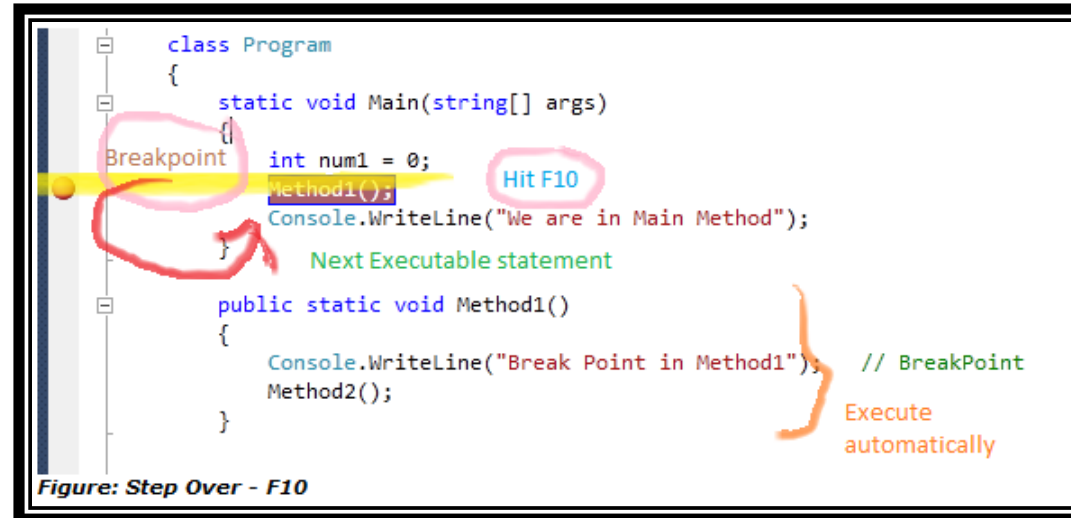
Debugging

Ferramenta de debugging permite monitorizar a execução do programa, linha a linha, Verificando o estado das variáveis e a sequência lógica das instruções executadas.



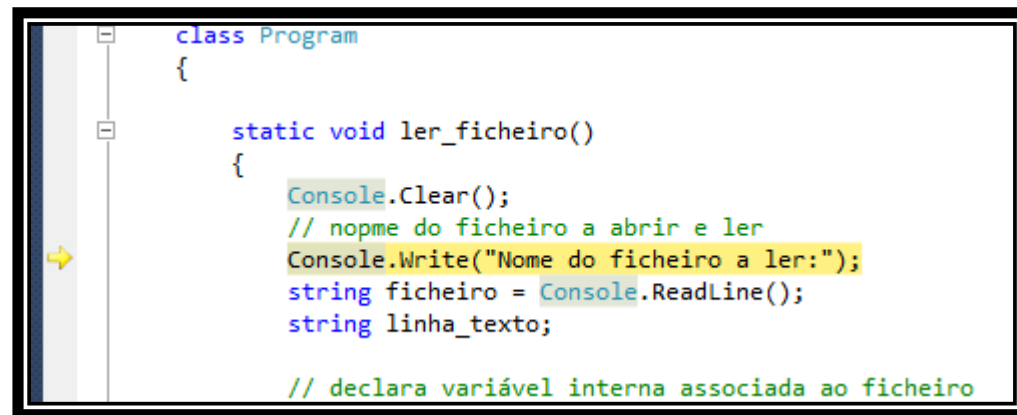
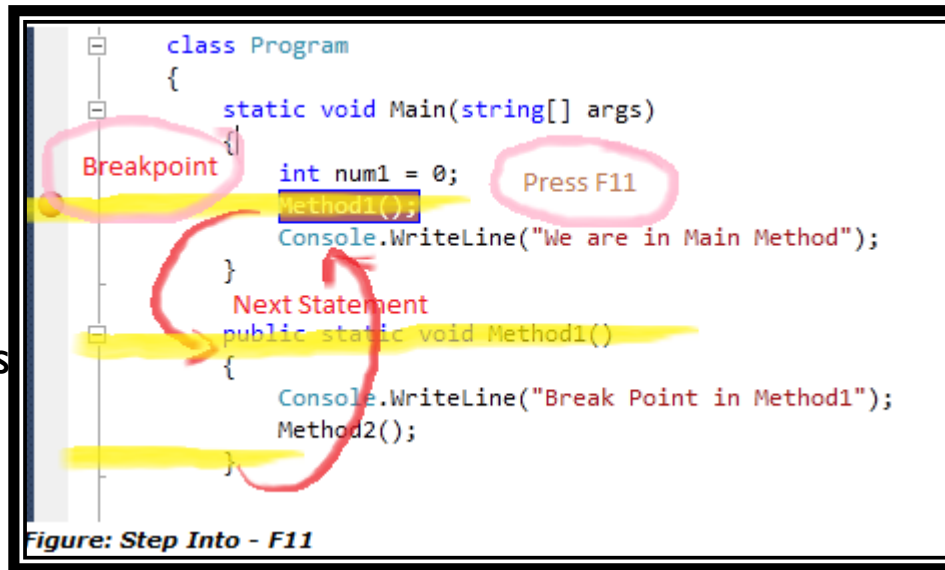
Debugging

- **F5 - Start debugging**
inicia a execução do programa
- **F10 – Step Over:**
Clicando sucessivamente em F10, executa o código fazendo o *debugging* linha a linha



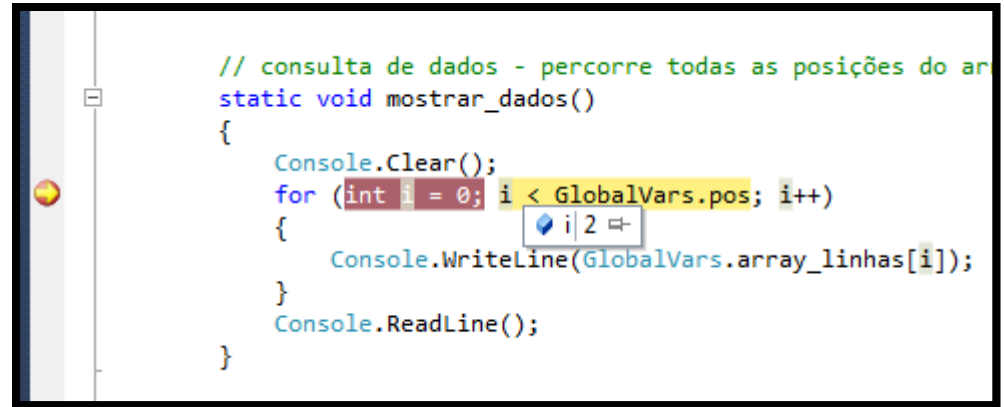
Debugging

- **F11 – Setp Into**
Semelhante ao F10, mas também faz o debugging linha a linha nos procedimentos entretanto invocados no código



Debugging

- Criar **BreakPoints**
Tecla **F9** ou menu
Debug > Toggle Breakpoint

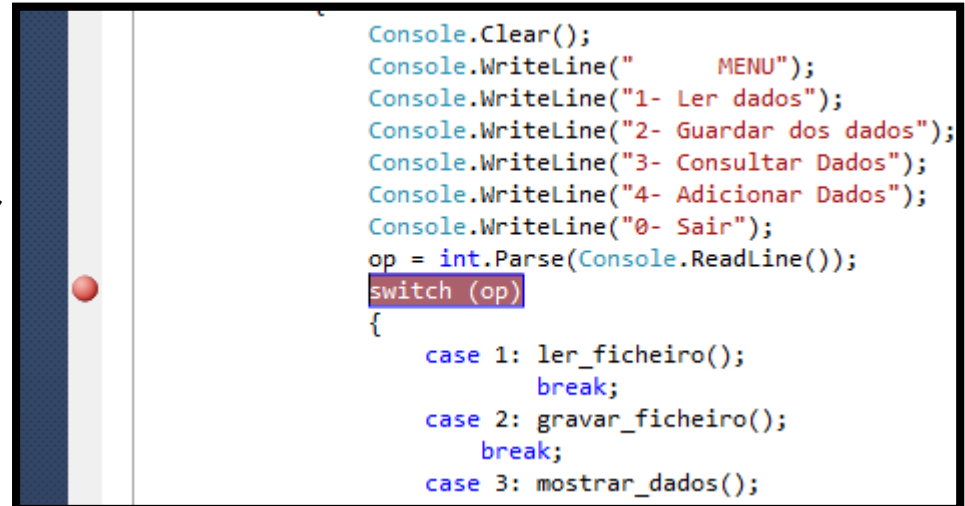


Um breakpoint indica uma linha de código onde a execução do programa vai parar, de forma a que possamos analisar os conteúdos das variáveis.

A partir de um breakpoint podemos avançar na execução do código com F10 ou F11

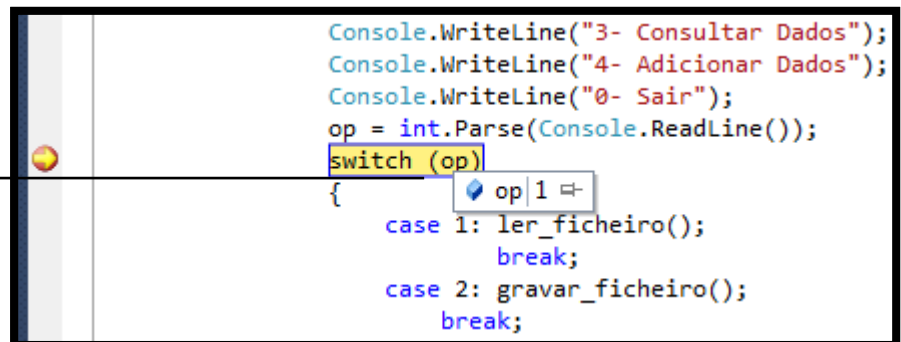
Debugging

Indico uma posição onde posso parar a execução do programa e verificar qual o valor da variável `op`.



```
Console.Clear();
Console.WriteLine("    MENU");
Console.WriteLine("1- Ler dados");
Console.WriteLine("2- Guardar dos dados");
Console.WriteLine("3- Consultar Dados");
Console.WriteLine("4- Adicionar Dados");
Console.WriteLine("0- Sair");
op = int.Parse(Console.ReadLine());
switch (op)
{
    case 1: ler_ficheiro();
           break;
    case 2: gravar_ficheiro();
           break;
    case 3: mostrar_dados();
```

Valor que a variável `op` está a assumir em determinado momento da execução do programa



```
Console.WriteLine("3- Consultar Dados");
Console.WriteLine("4- Adicionar Dados");
Console.WriteLine("0- Sair");
op = int.Parse(Console.ReadLine());
switch (op)
{
    case 1: ler_ficheiro();
           break;
    case 2: gravar_ficheiro();
           break;
```