



# Arquitetura de Computadores

LIC. EM ENG.<sup>a</sup> INFORMÁTICA  
FACULDADE DE CIÊNCIAS E TECNOLOGIA  
UNIVERSIDADE DE COIMBRA



## Lab 7 – Ponteiros e o *Debugger* GDB

***Neste trabalho de laboratório pretende-se ver como os ponteiros permitem uma maneira flexível de trabalhar com estruturas de dados, e como o debugger gdb pode ser útil para detetar erros (bugs) e falhas dos programas relacionadas com a má utilização dos ponteiros. Esta ferramenta de debug poderá ser inclusive utilizada para depurar erros nas nossas funções em assembly.***

### 1. Introdução

Ao trabalhar com ponteiros, por vezes torna-se difícil identificar os erros que tipicamente ocorrem nos nossos programas. Alguns deles podem ter como origem uma utilização indevida de ponteiros. Um *debugger* permite, em caso de falha do nosso programa, identificar o local onde ocorreu o problema. Permite também correr o programa passo a passo, ver o estado das variáveis e definir pontos de paragem (*breakpoints*).

No final deste trabalho deverá que ser capaz de responder à seguinte lista de questões relacionadas com a utilização de um *debugger*:

1. Como correr um programa no gdb?
2. Como deve ser feita a compilação para ter o máximo de informação sobre o programa ao correr dentro do gdb?
3. Como colocar um *breakpoint* (ponto de paragem) num programa?
4. Como colocar um *breakpoint* que só ocorra quando um conjunto de condições for verdadeiro (por exemplo, quando determinadas variáveis têm um valor específico)?
5. Como executar a linha seguinte do programa em C depois de um *break*?
6. Se a linha seguinte for uma chamada a uma função, a função é executada num passo único. Como é que se consegue executar o código dentro da função linha a linha?
7. Como continuar a correr o resto do programa depois de um *break*?
8. Como ver o valor de uma variável (ou mesmo de uma expressão) no gdb?
9. Como é que se pode configurar o gdb para escrever sempre o valor de uma determinada variável ao executar o programa passo a passo?
10. Como escrever uma lista com todos os variáveis e respetivos valores no ponto do programa onde nos encontramos?
11. Como sair do gdb?

No arquivo distribuído com o trabalho, para além do código base para os exercícios seguintes, pode encontrar o *GDB Reference Card* que sintetiza os principais comandos que pode utilizar no ***gdb***. A documentação completa sobre todos os comandos disponíveis pode ser acedida em <http://www.gnu.org/software/gdb/documentation/>.

## 2. *Debug* de um pequeno programa em C com *strings*

### a) Teste do programa.

Analise, compile e teste o programa **appendTest.c**, distribuído com o enunciado do trabalho. Este programa procura juntar três *strings* introduzidas pelo utilizador numa única *string*. Repare que o programa não funciona corretamente.

### b) Detecção do erro com ***gdb***.

Vamos recorrer ao ***gdb***, um debugger *Open Source* disponível na máquina MIPS que estamos a utilizar, para fazer o *debug* do programa. Para começar a utilizar o ***gdb*** tem de recompilar o programa para adicionar informação adicional que permita ao *debugger* associar as instruções em código máquina existentes no executável com as linhas do programa em C, bem como as zonas de dados a variáveis. Para tal deve utilizar a flag “-g” do gcc quando compilar:

```
gcc -g appendTest.c -o appendTest
```

De seguida chame o ***gdb*** com o seu programa a partir da linha de comando:

```
gdb appendTest
```

Ao abrir o ***gdb*** pode correr o programa para identificar o local onde o erro ocorreu. Repare que o ***gdb*** irá parar no *segmentation fault*, permitindo fazer o *debug* neste ponto de paragem. Primeiro deve verificar onde se encontra no programa. O comando a utilizar é o **backtrace** (ou **bt**), que imprime uma lista com o “rasto” de chamadas a funções (*stack trace*) até ao ponto de paragem. Com este comando irá conseguir identificar a linha de código que está a gerar o problema. Olhe bem para essa linha e tente identificar a má utilização de ponteiros que deu origem ao problema. Saia do ***gdb*** (comando **quit**) e corrija este problema.

### c) Correção do erro e novo teste.

Depois de corrigir o problema anterior, compile e execute novamente o programa. Utilize por exemplo os seguintes valores para as três strings: str1 = “AAA”, str2 = “BBB” e str3 = “CCC”. O programa continua a funcionar corretamente? Onde está o problema? Utilize o **Ctrl+C** para sair do programa.

Para tentarmos descobrir o problema volte a compilar o código com a opção -g e corra o ***gdb***. Coloque um *breakpoint* na função **append**, e corra o programa. Utilize o comando **break** ou **b** para definir o breakpoint (**b append**) e o comando **run** ou **r** para executar o programa (veja na folha de ajuda como estes comandos funcionam). Quando o *debugger* parar no *breakpoint*, execute as instruções da função **append()** linha a linha (comando **n**), observando os valores das várias variáveis (comando **print** ou **p**, por exemplo **p s1**). Repare bem nos valores das variáveis **s1**, **s2** e **s3** passados à função. Estarão corretos? Então porque é que este erro ocorre?

Corrija o *bug* no programa, compile e teste novamente até funcionar corretamente.

Dica: como é que se representam *strings* em linguagem C?

### 3. *Segmentation faults e bus errors*

Os *segmentation faults* e *bus errors* são erros comuns em C e estão normalmente relacionados com ponteiros. Geralmente são provocados por ponteiros com endereços inválidos, ou referenciados incorretamente (operador `*`). Vamos agora proceder ao *debug* de mais um programa que contém este tipo de erro.

#### a) Teste do programa.

Compile e teste o programa **average.c**. Tal como o nome sugere, o programa devia calcular a média de um conjunto de números inteiros. Mas na versão fornecida o programa gera um *segmentation fault* depois de aceitar mais do que um valor de entrada.

#### b) Detecção do erro com gdb.

Certifique-se que compilou o programa com informação para *debug*, carregue e corra o programa no gdb. Repare que o gdb irá parar no *segmentation fault*, permitindo fazer o *debug* neste ponto de paragem. Tal como no exercício anterior deverá identificar o local onde o programa está a falhar utilizando o comando `bt`. Repare que o erro está no fundo de uma sequência de chamadas a funções do sistema (`scanf`). Uma vez que o código do sistema está correto (pelo menos esperamos que sim!), utilize o comando `frame n` tentar identificar o código que conduziu ao erro. O valor de `n` deverá ser o indicado no *stack trace* do backtrace e deverá corresponder à última entrada relativa ao nosso código. Por exemplo neste caso, tente `frame 3`. O gdb escreve a linha do programa onde ocorreu o *segmentation fault*. Examine cuidadosamente o código para detetar o erro.

#### c) Correção do erro e novo teste.

Corrija o erro que provocava o *segmentation fault*, recompile e teste o programa.

### 4. Passagem por valor e por referência com ponteiros

Se corrigiu o *bug* no exercício anterior, o programa já lê os valores corretamente, mas devolve um valor de média errado.

Utilize o gdb para detetar e corrigir o erro, observando os valores de saída da função **read\_values()**. Para tal pode colocar um *breakpoint* indicando o número da linha do programa onde a função é chamada. Pode em alternativa colocar o *breakpoint* dentro da função e continuar a execução até ao final da função e ver os valores devolvidos. Para correr o programa até ao final da função onde nos encontramos, utilize o comando `finish` do gdb.

O programador que escreveu **average.c** tentou passar uma variável por referência. Em C++ é possível passar variáveis por referência, mas não em C. Explique porque é que os ponteiros podem dar a ilusão de uma linguagem de programação permitir passagem por referência.

Corrija o programa para que este passe a ter o funcionamento desejado.

## 5. Binarizar uma Imagem

Uma imagem  $w \times h$  pode ser guardada como uma tabela unidimensional do tipo 'unsigned char'. Se **img** é a referida tabela, então o byte **img[i\*w+j]**, em que  $0 \leq i < h$  e  $0 \leq j < w$ , guarda o nível de cinzento do pixel situado na linha **i** e coluna **j**. Pretende-se que escreva uma função em C que faça a binarização de uma imagem. O seu código deverá percorrer todos os píxeis da imagem, colocando a zero aqueles que estão abaixo de um limiar predefinido, e escrevendo 255 nos restantes.



**Fig. 1 – Imagem original e imagem binarizada correspondente.**

Para esta parte do trabalho vai necessitar dos ficheiros **main.c** e **bin\_img.c** fornecidos em anexo. O código do ficheiro **main.c** lê uma imagem **input.pgm** (fornecida com o trabalho), chama a função de binarização, e devolve a imagem binarizada em **output.pgm**. **Não é necessário alterar o código do ficheiro main.c!**

Se o executável final for chamado **binariza**, então para correr o programa deverá utilizar a seguinte linha de comandos:

```
./binariza input.pgm output.pgm
```

Deverá completar o código da função **bin\_img()** em **bin\_img.c** de forma a esta binarizar a imagem passada em memória. Note que o endereço de memória onde a imagem está guardada é indicado pelo ponteiro **dp**, e a largura e altura da imagem são passadas diretamente em **width** e **height**. Para obter o executável final deverá criar os códigos objeto **main.o** e **bin\_img.o** e ligá-los convenientemente, recorrendo a um *makefile*.

Como exercício extra, altere o programa anterior para aceitar o valor do limiar na linha de comando.

Notas:

1. Como a implementação do gcc na máquina MIPS por defeito assume uma implementação mais clássica da linguagem C, algumas funcionalidades disponíveis em alguns compiladores de C mais modernos podem não estar disponíveis. Por exemplo, definir o tipo das variáveis **inline** com as instruções não é permitido por defeito (ex.: `for(int i=0;i<10;i++)...`). Se pretender manter esta funcionalidade deverá adicionar a flag `-std=c99` na compilação.

2. *Para testar o programa no servidor MIPS, vai ter de transferir para a sua conta no servidor o ficheiro com a imagem de teste (**input.pgm**). Para visualizar a imagem resultante da execução do programa, terá de a transferir primeiro para o seu computador local e visualizá-la utilizando a ferramenta disponibilizada na pasta **Tools** (**OpenSeeIt.exe**) se estiver a utilizar o Windows. Para outros sistemas operativos poderá utilizar um visualizador de imagens compatível (que leia imagens PGM) ou converte-las para um dos formatos de imagem mais usuais (.bmp, .jpg, etc...). Para isso poderá usar o conversor online disponível no endereço <https://convertio.co/pt/>.*
3. *Para conseguir fazer com que o programa aceite o valor de limiar através da linha de comando, considere usar a função `atoi()`, que converte um número inteiro representado por uma string numa variável do tipo inteiro. Para mais informações sobre esta função, consulte o web site <http://linux.die.net/man/3/atoi>.*