Lab 2 – Estruturas de Dados, 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.

1. Binarizar uma Imagem

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

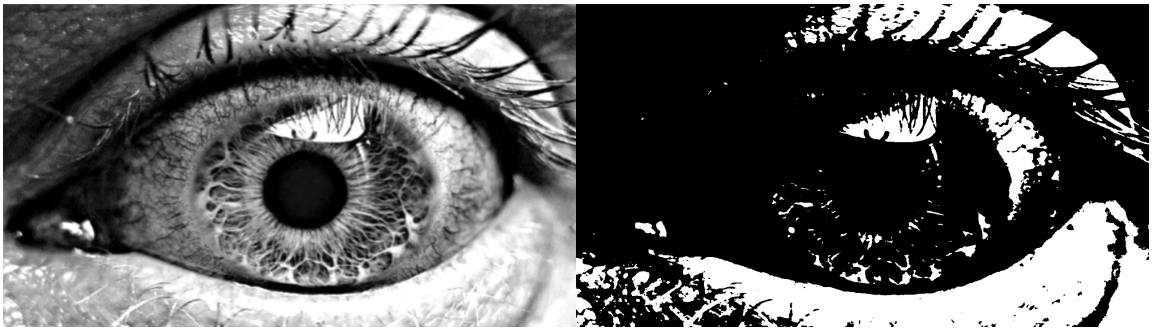

Fig. 1 – Imagem original e imagem binarizada correspondente.

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 **imagem.pgm**, chama a função de binarização, e devolve a imagem binarizada em **output.bmp**. Se o executável final for **binariza**, então a chamada da linha de comandos será:

```
./binariza imagem.pgm output.bmp.
```

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 indicada 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.

Nota: para um conversor online de imagem PGM, pode usar o site https://convertio.co/pt/. Contudo, só precisa deste conversor se criar novas imagens de entrada.

2. Introdução ao GDB

Ao trabalhar com estruturas de dados e ponteiros, por vezes torna-se mais difícil identificar erros nos programas. Um *debugger* permite correr o programa passo a passo, ver o estado das variáveis, definir pontos de paragem (*break points*), e analisar o ponto em que o programa falhou.

Deve estudar as respostas à seguinte lista de questões:

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 especifico)?
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 todas as variáveis e respetivos valores no ponto do programa onde nos encontramos?

No InfoEstudante, para além do código base para os exercícios seguintes, tem o *GDB Reference Card* que sintetiza os principais comandos do gdb, tendo a documentação completa disponível em http://www.gnu.org/software/gdb/documentation/.

3. *Debug* de um pequeno programa em C com *strings*

a) Teste do programa.

Analise, compile e teste o programa **appendTest.c**. Tente juntar várias strings, e repare que o resultado nem sempre é correto. Para sair do programa tem que fazer Ctrl-C.

b) Deteção do erro com gdb.

Vamos agora recorrer ao gdb para fazer o *debug* do programa. Para começar tem que recompilar o programa com informação para *debug*, para que o *debugger* possa 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 utilize a flag –g do gcc:

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

De seguida chame o gdb com o seu programa a partir da linha de comando (ou de dentro do GNU Emacs). Coloque um *breakpoint* na função append, e corra o programa. Quando o *debugger* parar no *breakpoint*, execute as instruções da função **append()** linha a linha, observando os valores das variáveis. Repare bem nos valores de **s1** e **s2** passados à função. Estarão corretos? Por quê este erro? (dica: como é que se representam *strings* em C?)

c) Correção do erro e novo teste.

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

4. *Segmentation faults* e *bus errors*

Os *segmentation faults* e *bus errors* são erros comuns relacionados com ponteiros em C. Geralmente são provocados por ponteiros com endereços inválidos, ou de-referenciados incorretamente (operador *). Vamos agora proceder ao debug de um programa que tem 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. Mas na versão fornecida o programa gera um *segmentation fault* depois de aceitar mais que um valor de entrada (note: ao correr numa janela DOS em Windows o erro por vezes é tolerado, só dando erro quando o primeiro valor é negativo ou positivo elevado).

b) Deteção do erro com gdb.

Certifique-se que compilou o programa com informação para *debug*, e carregue e corra o programa no gdb. Repare que o gdb para 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 de funções (*stack trace*) até ao ponto de paragem. Repare que o programa está no fundo de uma sequência de chamadas de funções do sistema. Uma vez que o código do sistema está correto (pelo menos esperamos que sim!), utilize o comando **frame n** para se deslocar para a última chamada no nosso código **average.c** que levou ao erro, sendo **n** o indicado no *stack trace* do backtrace. 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.

5. 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, ou colocar 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.