

# Arquitetura de Computadores 2018/19

## Ficha 3

**Tópicos:** Programação em C. Desenvolvimento e depuração de programas.

### Desenvolvimento de um programa em C e *debug*

Esta ficha está na forma de um guião que deve seguir e que ilustra um pouco do processo de desenvolvimento com auxílio de várias ferramentas. Cada linguagem e cada ambiente de desenvolvimento tem as suas ferramentas, mas aqui vamos ilustrar com algumas que habitualmente se utilizam em Unix/Linux para programação em C.

Copie o programa C de nome “`simples.c`”, disponibilizado no sistema CLIP, e veja-o num editor.

Este programa cria em `s2` uma *string* que é uma cópia de `s1` e depois imprime ambas as *strings* (para verificarmos que temos uma cópia). Depois transforma `s2` para passar todas as letras minúsculas a maiúsculas e usa a função `soma` para somar todos os inteiros no vetor `z`. Pode descobrir alguns erros apenas ao olhar para o programa, mas faça de conta que não os viu.

#### Compilação de um programa em C e verificações extra

Ao compilar este programa, não são detetados erros sintáticos, mas há um aviso sobre uma função não declarada:

```
warning: implicit declaration of function 'transforma'
```

e depois aparece um erro de **ligação**:

```
undefined reference to `transforma'
```

Olhando para os identificadores das funções pode-se concluir que nos enganámos no nome da função: declaramos “`transform`” mas usamos “`transforma`”. Corrija, passando a usar o mesmo nome e compile!

Deve agora tentar executar o programa. Note que depois de imprimir a *string* `s2`, o processo fica “bloqueado”. Interrompa premindo **Ctrl-C**. Mesmo que um compilador seja capaz de compilar um programa sem reportar erros e produzir um executável, não quer dizer que o programa esteja correto! Como proceder para localizar a origem do erro?

Podemos pedir a alguns compiladores para fazerem verificações extra para identificar possíveis erros ou potenciais erros causados por um mau uso da linguagem. Também podemos usar ferramentas para análise de código que nos podem avisar de alguns problemas. Vamos começar por pedir ao compilador que seja “implicante” e que nos avise de todos os possíveis problemas que conseguir identificar. Para isso, compile o programa com a opção “`-Wall`”:

```
cc -Wall -o simples simples.c
```

Use também a ferramenta `cppcheck` para fazer uma análise do código. Esta é particularmente útil quando o compilador de C não é capaz deste tipo de análise†. Experimente:

```
cppcheck simples.c
```

e analise o resultado (mensagens de aviso e de erro). Pode obter ainda mais informação com:

```
cppcheck --enable=warning simples.c
```

Cada um destes comandos avisa que várias situações que podem ser erros:

- Na linha 54 está a tentar imprimir uma *string* (`char*`) indicando ao `printf` apenas `char` (`%c`);
- Na linha 58 está a usar a variável `x` que nunca foi inicializada;

† Na imagem Linux fornecida deve ter todos os comandos necessários. No editor `medit`, no menu `Tools`, esses comandos devem estar configurados para executar sobre o ficheiro que estiver a editar.

- O programa `cppcheck` ainda avisa que o vetor `z`, na linha 56, está a ser acedido numa posição que não existe (`z[2]` não existe) e que está a ser passado para a função `soma` (linha 57) que espera um vetor maior do que `z`.

Afinal existem muitos mais erros no nosso programa! A *string* deve ser impressa com `%s` (como acontece com a impressão da *string* `s1`); `x` deve ser inicializado com o resultado de `soma()` na linha 57 e o vetor `z` devia ter 3 posições em vez de duas. Corrija!

Volte a compilar com `-Wall` (pode também correr o `cppcheck`). Pelo menos os problemas anteriores devem estar todos corrigidos e o programa agora afixa a *string* `s2` corretamente (igual a `s1`) mas continua a bloquear (interrompa com `Ctrl-C`). Como descobrir o problema?

### Compilação de um programa em C e *debugging*

Um *debugger* é uma ferramenta que permite a execução controlada do programa e a inspeção detalhada do seu estado durante a execução (neste caso, usamos o `gdb`, o *GNU debugger*). Quando pretendido, a execução do programa “alvo” é suspensa (mas sem terminar) e, nessa altura, podemos analisar o estado do programa alvo e, em particular, o valor das variáveis. Podemos assim saber onde vai a execução e ir executando as instruções do programa alvo passo-a-passo (leia-se “linha-a-linha”) e verificar o valor das variáveis em cada um destes passos.

Para facilitar o uso do *debugger*, comece por compilar o programa com a seguinte linha de comando (atenção à opção `-g`; pode continuar a usar `-Wall`):

```
cc -g -o simples simples.c
```

A opção `“-g”` serve para indicar ao compilador para adicionar informação de *debug* ao programa executável “`simples`”. Essa informação permitirá ao *debugger* correlacionar as linhas de código fonte C com as instruções em código máquina geradas pelo compilador.

Para carregar o seu programa “`simples`” sob controlo do *gdb*, execute a seguinte linha de comando

```
$ gdbtui simples ou $ gdb -tui simples
```

Prima RETURN. Deverá observar algo semelhante à figura seguinte:

```

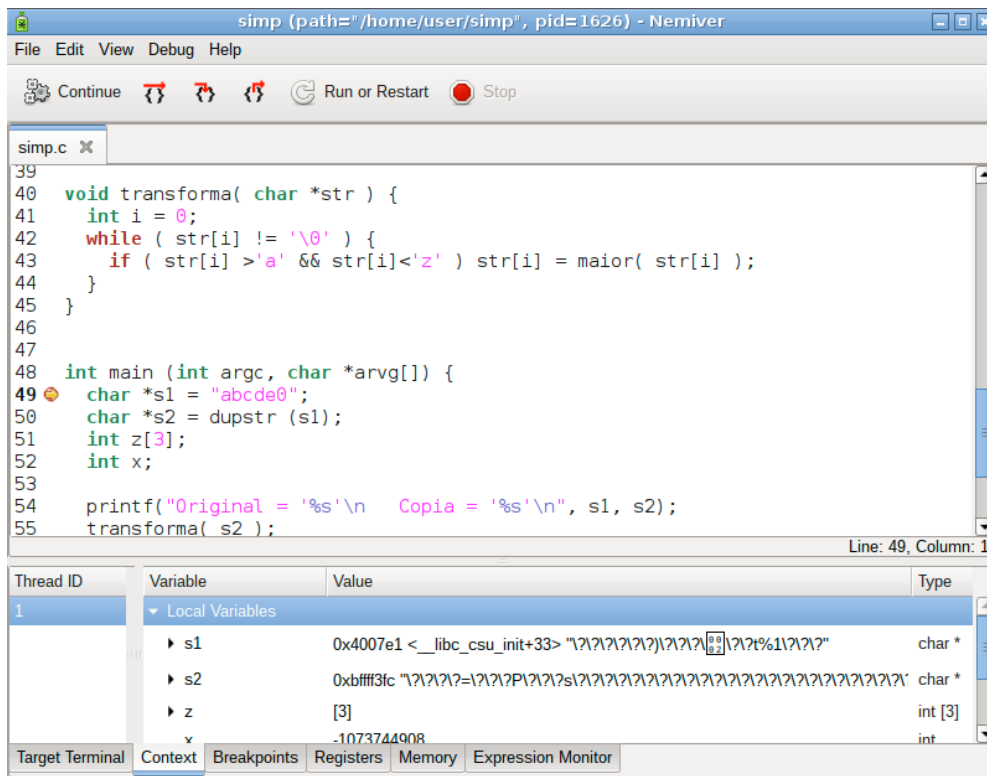
simples.c
48 int main (int argc, char *argv[]) {
49     char *s1 = "abcde0";
50     char *s2 = dupstr (s1);
51     int z[3];
52     int x;
53
54     printf("Original = '%s'\n    Cópia = '%s'\n", s1, s2);
55     transform( s2 );
56     z[0]=z[1]=z[2]=1;
57     x=soma( z );
58     printf( "nova= %s, x=%d\n", s2, x );
59     return 0;
60 }
61
62
63

exec No process in:                               L??  PC: ??
---Type <return> to continue, or q <return> to quit---
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from simples...done.
(gdb)

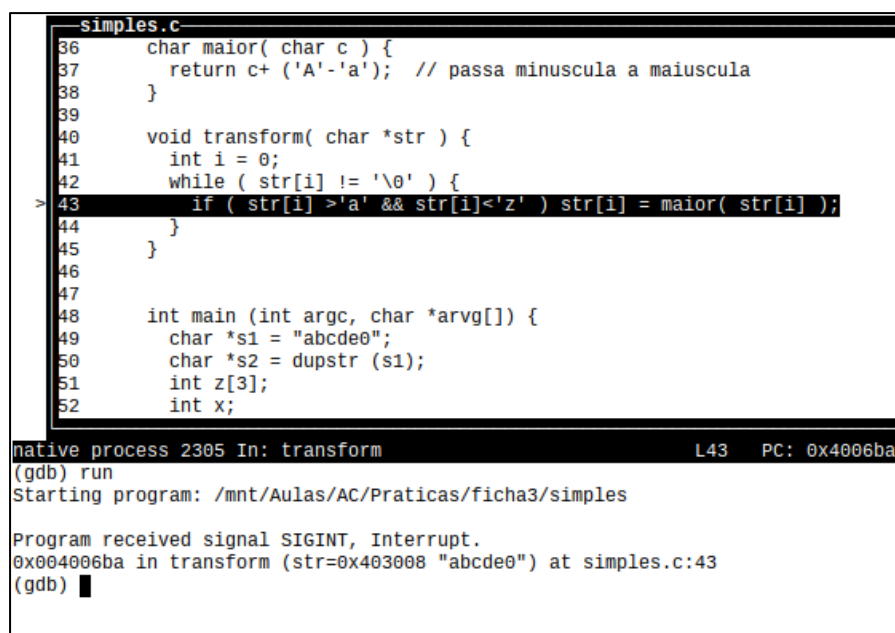
```

Na metade superior da janela poderá visualizar o código fonte (neste caso o corpo da função `main`). Na metade inferior tem a consola do *gdb* onde pode introduzir comandos.


Também pode executar o *gdb* com uma interface gráfica como o *nemiver*:



Para executar o programa sob controlo do *debugger*, deverá introduzir o comando **“run”** na consola do *gdb*. Se o fizer neste caso, o programa irá bloquear como antes. Interrompa com Ctrl-C. Desta vez o *debugger* mostra onde o seu programa está, o que parece ser dentro da função *transform*. *Sempre que o terminal parecer ficar “baralhado” prima Ctrl-L para atualizar o ecrã.*



### Execução passo-a-passo e *breakpoints*

Com a execução suspensa, podemos executar passo-a-passo usando o comando **next** (ou  no *nemiver*). Verifique que o programa está num ciclo infinito. Veja agora o conteúdo das variáveis com o comando **“print x”**, onde “x” é o nome de uma variável (ou mesmo, uma expressão aritmética que pode incluir as variáveis válidas do programa naquele ponto). Experimente `print str` e `print i`. Deve confirmar que a *string* não foi alterada e que *i* continua com o valor 0. Já sabe qual é o erro? Termine o *debugger* (comando **quit**) e corrija o erro (falta `i=i+1`).

Volte a compilar e executar. Agora tudo parece correr bem, mas a *string* *s2* após a transformação para maiúsculas ainda apresenta o 'a' minúsculo. O programador falhou e não está a converter o 'a'! O problema deve estar na função *transforma* ou na *maior*.

Volte a executar o *gdb* mas agora, antes de fazer **run** vamos indicar ao *debugger* um ponto de paragem (já sabemos onde parar e sem isto o programa corria até ao fim e não podíamos fazer nada). Para tal vamos colocar um *breakpoint* (ponto de paragem). Isso faz-se com o comando *break x*, onde *x* é um número de linha ou o nome de uma função. Por exemplo, *break transforma* colocará um *breakpoint* na primeira instrução da função *transforma*. Note que se pedir para listar a função (*list transforma*) na linha 41 apareceu o "b+", que identifica que há um *breakpoint* naquela linha. Nas interfaces gráficas, tipicamente, os *breakpoints* são colocados selecionando a linha com o rato e aparece uma marca a vermelho indicando o *breakpoint* activo.

Pode-se listar quais são os *breakpoints* existentes com o comando *info break*.

```

simples.c
35  char maior( char c ) {
36      return c+ ('A'-'a'); // passa minuscula a maiuscula
37  }
38
39  void transforma( char *str ) {
40      int i = 0;
41      while ( str[i] != '\0' ) {
42          if ( str[i] >'a' && str[i]<'z' ) str[i] = maior( str[i] );
43          i = i+1;
44      }
45  }
46
47
48  int main (int argc, char *argv[]) {
49      char *s1 = "abcde0";
50
exec No process in: L?? PC: ??
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from simples...done.
(gdb) br transforma
Breakpoint 1 at 0x6a6: file simples.c, line 41.
(gdb) list trabsforma
Function "trabsforma" not defined.
(gdb) list transforma
(gdb)

```

Execute com **run**. O programa vai parar quando chegar ao *breakpoint* e **parará antes de executar as instruções nessa linha** (prima Ctrl-L se necessitar de redesenhar o ecrã).

Pode agora usar **next** e imprimir as variáveis com *print*, procurando saber o que está mal. Deve ver que "print str[i]" mostra que está na letra 'a' quando i==0, mas a função *maior* não é chamada. Pode mesmo fazer "print str[i]>'a'" e verificar que devolve 0 (ou seja str[i] não é maior que 'a' daí não chamar a função *maior*. Claramente a condição devia ser "str[i]>='a' && str[i]<='z'".

Pode também seguir todas as alterações a variáveis. Use o comando *watch i*, para ver o novo valor de cada vez que a variável *i* for alterada. Execute vários passos com **next**. Para entrar dentro da função *maior* e ver a sua execução interna, use o comando **step** em vez de **next**.

```

simple.c
34
35
36 char maior( char c ) {
37     return c+ ('A'-'a'); // passa minuscula a maiuscula
38 }
39
40 void transforma( char *str ) {
41     int i = 0;
42     while ( str[i] != '\0' ) {
43         if ( str[i] >'a' && str[i]<'z' ) str[i] = maior( str[i] );
44         i = i+1;
45     }
46 }
47
48
49 int main (int argc, char *argv[]) {

```

native process 2489 In: transforma L42 PC: 0x4006f3

Hardware watchpoint 2: i

Old value = 2  
New value = 3

Hardware watchpoint 4: i

---Type <return> to continue, or q <return> to quit---

Vamos agora executar até ao fim dando o comando *continue* (ou só *cont*).

### Execução de um programa com um verificador de memória

Apesar de todas as correções o programa continua com erros. Estes não causaram problemas durante a execução, mas podem mais tarde, em determinadas condições, dar origem a resultados errados ou ao programa abortar durante a execução. São normalmente erros relacionados com a gestão de memória, acesso a *arrays* ou ao uso de apontadores.

Vamos por isso usar uma funcionalidade de alguns compiladores para verificar a correta utilização da memória. Na compilação podemos pedir a introdução de código extra que verifica vários dos erros anteriores. No entanto esta funcionalidade torna o programa mais lento e não garante que encontra todos os erros. **Nunca assuma que um programa não tem erros só porque não foram encontrados.**

Compile o programa, agora com o seguinte comando:

```
cc -g -fsanitize=address -o simples simples.c
```

Ao executar, o programa vai abortar dando uma série de informação. Repare nas primeiras linhas:

```

==2534==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xb59007f6 at pc
0x004f3b29 bp 0xbfd85b68 sp 0xbfd85b5c
WRITE of size 1 at 0xb59007f6 thread T0
#0 0x4f3b28 in dupstr /mnt/Aulas/AC/Praticas/ficha3/simples.c:29
#1 0x4f3d8b in main /mnt/Aulas/AC/Praticas/ficha3/simples.c:51

```

Tal indica que acedeu para além dos limites de uma variável ou *array* e que o erro ocorreu no *dupstr*, linha 29 (que foi chamado na linha 51 no *main*). Olhando para essa linha vê:

```
newstr[size] = '\0';
```

O que se passa? Pode correr o *debugger* para confirmar, mas *size* vai para além do seu *array*, porque se no *malloc* pediu a criação de um *array* de dimensão *size*, logo só existem posições de 0 a *size-1*! Não corrija ainda.

Outra ferramenta que pode usar é o *valgrind*<sup>‡</sup>. Esta é independente do compilador. O *valgrind* faz também uma execução controlada de um programa alvo e pode realizar vários tipos de verificações. Entre elas, acessos indevidos à memória (excelente para ajudar na deteção de escritas para além das

<sup>‡</sup> Se não tiver este comando use a aplicação do seu Linux para instalar *software* ou, num terminal, dê o comando:  
`apt-get install valgrind`

dimensões dos arrays) e erros na gestão de memória (e.g., fazer *malloc* e não fazer o *free* correspondente, ou tentar fazer *free* duas vezes do mesmo bloco de memória).

Compile o programa de novo, só com a opção -g:

```
cc -g -o simples simples.c
```

Para executar o programa *simples* sob controlo do *valgrind* execute o comando:

```
valgrind simples ou alleyoop simples (este é uma interface para o valgrind)
```

Deverá observar algo semelhante à figura seguinte:

```
user@linux32AC:/mnt/Aulas/AC/Praticas/ficha3$ valgrind simples
==2553== Memcheck, a memory error detector
==2553== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==2553== Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for copyright info
==2553== Command: simples
==2553==
==2553== Invalid write of size 1
==2553==   at 0x10866B: dupstr (simples.c:29)
==2553==   by 0x108738: main (simples.c:51)
==2553== Address 0x4a0102e is 0 bytes after a block of size 6 alloc'd
==2553==   at 0x482E27C: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-li
nux.so)
==2553==   by 0x108632: dupstr (simples.c:20)
==2553==   by 0x108738: main (simples.c:51)
==2553==
Original = 'abcde0'
==2553== Invalid read of size 1
==2553==   at 0x48313C3: __GI_strlen (in /usr/lib/valgrind/vgpreload_memcheck-x
86-linux.so)
==2553==   by 0x488B17F: vfprintf (vfprintf.c:1637)
==2553==   by 0x4890955: printf (printf.c:33)
==2553==   by 0x108753: main (simples.c:55)
==2553== Address 0x4a0102e is 0 bytes after a block of size 6 alloc'd
==2553==   at 0x482E27C: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-li
nux.so)
==2553==   by 0x108632: dupstr (simples.c:20)
==2553==   by 0x108738: main (simples.c:51)
==2553==
```

Repare que o programa é integralmente executado. São reportados vários *incidentes* (dois nas caixas de destaque), mas note que um único erro pode gerar mais que um incidente.

O primeiro incidente diz que há um “*Invalid write of size 1, 0 bytes after a block of size 6 alloc'd*”. Isto significa, portanto, que estamos a escrever para além da zona que foi alocada (no byte imediatamente a seguir ao final da zona alocada). Mais, diz-nos que este erro ocorre na linha 29 (dentro da função *dupstr*). Esta linha contém “*newstr[size] = '\0';*”. Trata-se do problema que já vimos antes.

O segundo incidente diz que há um “*Invalid read of size 1, 0 bytes after a block of size 6 alloc'd*”. Ora este erro está relacionado com o anterior. No anterior era uma escrita. Agora é na leitura da variável “*s2*” aquando do *printf*. Corrigindo o primeiro incidente, deveremos corrigir também o segundo.

Corrija no *malloc* para pedir um *array* de *size+1* posições.

## Programação em C

1. Copie o programa C de nome “*args.c*” a partir do sistema CLIP. Este programa recebe um conjunto de argumentos na linha de comando e, para cada um dos argumentos, tenta adivinhar o seu tipo e depois invoca a função *printf*. No caso das *strings*, também usa a função *maiusculas*. Complete o programa implementando a função *maiusculas* que deve devolver a *string* em parâmetro com todas as letras passadas a maiúsculas. Veja a função *toupper* da biblioteca de C (use por exemplo o comando: *man 3 toupper*).

Recomenda-se agora que execute, passo-a-passo, o programa e experimente analisar o estado do programa (incluindo as variáveis locais e parâmetros) em diferentes fases da execução do programa (não se esqueça de usar *breakpoints*). Irá com certeza observar que quando executar “*next*”



o programa irá terminar sem parecer fazer nada. Isso porque deve executar o comando com argumentos na linha de comando! Para executar o programa com argumentos, no *debugger* deverá executar o comando “**run** *arg<sub>1</sub>* *arg<sub>2</sub>* ... *arg<sub>n</sub>*”. Por exemplo: *run 2015 3.1416 asdfg*.

Experimente ver o estado do `argv` (`print argv[0]`, `print argv[1]`, etc.). Irá ainda observar que quando executa “*next*” na linha que chama a função *maiusculas*, esta é executada de uma vez como se fosse uma única instrução, e não passo-a-passo. Para “entrar” dentro de uma função deverá utilizar o comando “*step*” (ou “*s*”) e não o comando “*next*”. Uma vez dentro da função, recomenda-se que utilize o comando *next*.

Note que quando o controlo está “dentro” da função pode facilmente imprimir o valor do parâmetro e variáveis e assim verificar se estão de acordo com o esperado e ver as alterações efetuadas à *string* `str`.

2. Faça um programa que fica em ciclo a ler uma *string* do teclado e a escrevê-la no ecrã prefixada com “->”. O programa só terminará quando o utilizador escrever a string “*fim!*”. Veja o seguinte exemplo (a **negrito** o que o utilizador escreveu):

```
$ ./echostring
ola
->ola
boa tarde
->boa tarde
fim
->fim
termina!!!
->termina!!!
fim!
->fim!
$
```

Para realizar este programa necessita de utilizar as funções: `fgets`, `printf`, e `strcmp`. Para saber como se usam estas funções leia as páginas de manual correspondentes executando no terminal o comando “*man 3 função*”, onde *função* deverá ser o nome da função de que pretende obter informação.