

Arquitetura de Computadores 2021/2022

Ficha 4

Tópicos: 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 e ilustrando o uso 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. No caso deste guião foi usado o compilador gcc 7.5.0 (outros ambientes ou versões podem ter comportamentos ligeiramente diferentes).

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 avisos (não os colocamos todos aqui) sobre a formatação no `printf` (linha 54) e sobre uma função não declarada (linha 55):

```
simples.c:54:41: warning: format '%c' expects argument of type 'int', but argument 3 has type 'char *' [-Wformat=]
```

```
printf("Original = '%s'\n      Cópia = '%c'\n", s1, s2);
```

```
simples.c:55:3: warning: implicit declaration of function 'transforma'; did you mean 'transform'? [-Wimplicit-function-declaration]
transforma( s2 );
```

e depois aparece um erro de **ligação** (ou *link*, indicado pelo utilitário `ld`):

```
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 apenas este erro, passando a usar o mesmo nome e compile! Mantem-se o primeiro aviso mas é produzido um executável. Ao executar o programa deve confirmar que há problemas ao tentar escrever a segunda *string*. Se o programa bloquear interrompa premindo **Ctrl-C** (Ctrl e C em simultâneo). A *string* `s2` deve ser impressa com `%s` (como acontece com a *string* `s1`) e a sua impressão já deve correr bem, mas fica bloqueado.

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 do código fonte 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
```

Deve aparecer o aviso sobre uma variável não inicializada:

```
simples.c:58:3: warning: 'x' is used uninitialized in this function
```

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
```

Esta ferramenta avisa que várias situações que podem ser erros:

- Na linha 56 é apontado o erro de que o vetor `z`, está a ser acedido fora dos seus limites (`z[2]` não existe) e, em consequência, o mesmo vai acontecer dentro da função chamada na linha 57;
- Na linha 57 é também indicado que o *array* `z` não tem a dimensão esperada pela função `soma`;
- Na linha 58 está a ler a variável `x` que nunca foi inicializada (visto antes);

Afinal existem muitos erros no nosso programa!; `x` era para 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 mas continua a bloquear (interrompa com Ctrl-C). Como descobrir o problema?

Compilação de um programa em C e sua execução em modo *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. Continuando com o mesmo programa `simples.c`, segue-se a descrição dos comandos mais comuns a utilizar no `gdb`. Para consolidar/completar o seu estudo, realize posteriormente também o guião disponível no capítulo 3 do livro *Dive into Systems*, secções 3.1 até à 3.3: https://diveintosystems.org/book/C3-C_debug/index.html

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
```

† Na imagem Linux fornecida pode não estar instalado o utilitário `cppcheck`; nesse caso, execute:
sudo apt install cppcheck.

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 (Nota: se a “caixa” estiver desenhada com linhas tracejadas em vez de contínuas execute \$ TERM=linux gdbtui simples):

```

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 *ddd*:



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.

```

simples.c
36 char maior( char c ) {
37     return c+ ('A'-'a'); // passa minuscula a maiuscula
38 }
39
40 void transform( 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     }
45 }
46
47
48 int main (int argc, char *argv[]) {
49     char *s1 = "abcde0";
50     char *s2 = dupstr (s1);
51     int z[3];
52     int x;

```

```

native process 2305 In: transform                                L43    PC: 0x4006ba
(gdb) run
Starting program: /mnt/Aulas/AC/Praticas/ficha3/simples

Program received signal SIGINT, Interrupt.
0x004006ba in transform (str=0x403008 "abcde0") at simples.c:43
(gdb) █

```

Execução passo-a-passo e *breakpoints*

Com a execução suspensa, podemos executar passo-a-passo usando o comando **next**. 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 streprint 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
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[]) {
50         char *s1 = "abcde0";

```

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**.

```

simples.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, abreviando, **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 informações. Repare nas primeiras linhas:

```
==2152==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xb5f007b6 at
pc 0x004cdb46 bp 0xbfc59038 sp 0xbfc59028
WRITE of size 1 at 0xb5f007b6 thread T0
#0 0x4cdb45 in dupstr /home/ac/Aulas/21-22/2oSem/AC/Ficha03/simples.c:29
#1 0x4cddb1 in main /home/ac/Aulas/21-22/2oSem/AC/Ficha03/simples.c:51
#2 0xb77cefa0 in __libc_start_main (/lib/i386-linux-gnu/libc.so.6+0x18fa0)
#3 0x4cd820 (/home/ac/Aulas/21-22/2oSem/AC/Ficha03/simples+0x820)
...
```

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 com o *array* da 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*[‡]. 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 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, como habitualmente, só com a opção -g:

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

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

```
valgrind --leak-check=full ./simples
```

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

Deverá observar que ao executar o programa este mostra agora algo semelhante ao seguinte:

```

==2138== Memcheck, a memory error detector
==2138== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2138== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright
info
==2138== Command: ./simples
==2138==
==2138== Invalid write of size 1
==2138==    at 0x1086E8: dupstr (simples.c:29)
==2138==    by 0x1087C8: main (simples.c:51)
==2138== Address 0x51d4a5e is 0 bytes after a block of size 6 alloc'd
==2138==    at 0x483021B: malloc (in /usr/lib/valgrind/vgpreload_mem-
check-x86-linux.so)
==2138==    by 0x1086AF: dupstr (simples.c:22)
==2138==    by 0x1087C8: main (simples.c:51)
==2138==
Original = 'abcde0'
==2138== Invalid read of size 1
==2138==    at 0x48333C3: __GI_strlen (in /usr/lib/valgrind/vgpre-
load_memcheck-x86-linux.so)
==2138==    by 0x4D660C7: vfprintf (vfprintf.c:1643)
==2138==    by 0x4D6C545: printf (printf.c:33)
==2138==    by 0x1087E3: main (simples.c:55)
==2138== Address 0x51d4a5e is 0 bytes after a block of size 6 alloc'd
==2138==    at 0x483021B: malloc (in /usr/lib/valgrind/...linux.so)
==2138==    by 0x1086AF: dupstr (simples.c:22)
==2138==    by 0x1087C8: main (simples.c:51)
==2138==
Copia = 'abcde0'
==2138== Invalid read of size 1
==2138==    at 0x108778: transform (simples.c:42)
==2138==    by 0x1087F1: main (simples.c:56)
==2138== Address 0x51d4a5e is 0 bytes after a block of size 6 alloc'd
==2138==    at 0x483021B: malloc (in /usr/lib/valgrind/...linux.so)
==2138==    by 0x1086AF: dupstr (simples.c:22)
==2138==    by 0x1087C8: main (simples.c:51)
==2138==
==2138== Invalid read of size 1
==2138==    at 0x48333C3: __GI_strlen (in /usr/lib/valgrind/...linux.so)
==2138==    by 0x4D660C7: vfprintf (vfprintf.c:1643)
==2138==    by 0x4D6C545: printf (printf.c:33)
==2138==    by 0x10882E: main (simples.c:59)
==2138== Address 0x51d4a5e is 0 bytes after a block of size 6 alloc'd
==2138==    at 0x483021B: malloc (in /usr/lib/valgrind/...linux.so)
==2138==    by 0x1086AF: dupstr (simples.c:22)
==2138==    by 0x1087C8: main (simples.c:51)
==2138==
nova= ABCDE0, X=3

```



```

==2138==
==2138== HEAP SUMMARY:
==2138==      in use at exit: 6 bytes in 1 blocks
==2138==    total heap usage: 3 allocs, 2 frees, 19,974 bytes allocated
==2138==
==2138== 6 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2138==    at 0x483021B: malloc (in /usr/lib/valgrind/vgpreload_mem-
check-x86-linux.so)
==2138==    by 0x1086AF: dupstr (simples.c:22)
==2138==    by 0x1087C8: main (simples.c:51)
==2138==
==2138== LEAK SUMMARY:
==2138==    definitely lost: 6 bytes in 1 blocks
==2138==    indirectly lost: 0 bytes in 0 blocks
==2138==    possibly lost: 0 bytes in 0 blocks
==2138==    still reachable: 0 bytes in 0 blocks
==2138==    suppressed: 0 bytes in 0 blocks
==2138==
==2138== For counts of detected and suppressed errors, rerun with: -v
==2138== ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 0 from 0)

```

São reportados vários *incidentes* (dois dos quais marcados a vermelho), mas note que um único erro pode gerar mais que um incidente. O primeiro incidente diz que estamos a escrever um byte 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';`. Confirma-se que se trata do problema que já vimos antes.

O segundo incidente diz que há uma leitura fora do array. Ora este erro (tal como os restantes incidentes) 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 os restantes.

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