Tratamento de erros e depuração

Tópicos abordados:
– Funções de depuração e tratamento de erros
– O depurador GDB

1. Funções de depuração e tratamento de erros

Sempre que se cria um programa existe a possibilidade de este conter erros (designados em inglês por *bugs*). Este anexo é dedicado à apresentação de algumas técnicas para auxiliar os programadores a detetarem os erros no código.

Uma das técnicas mais vulgares é o recurso à função '*printf()*' para imprimir o valor de variáveis ou simplesmente para detetar até onde é que a execução do programa correu sem problemas. Embora esta abordagem seja um pouco limitada, podemos aumentar o seu potencial se a informação que for mostrada tiver conteúdo importante, em vez de por exemplo "estou aqui!". Outro aspeto importante é o tratamento de erros, cuja inexistência é uma das causas mais comuns de problemas.

Assim, de forma a sistematizar a depuração e o tratamento de erros decorridos da execução de chamadas ao sistema, criaram-se três funções, cujos protótipos se apresentam de seguida:

```
void debug(const char *file, const int line, char *fmt, ...);
void warning(const char *file, const int line, char *fmt, ...);
void error(const char *file, const int line, int exitCode, char
*fmt, ...);
```

Figura 1 - Protótipos das funções de depuração e tratamento de erros

A função ***debug()*** mostra o nome e a linha do ficheiro onde foi chamada seguida de uma mensagem para o canal de erros *stderr*. A função ***warning()*** é semelhante à anterior, mas acrescenta a descrição do erro que se encontra na variável global *errno*. Por fim a função ***error()*** além de mostrar a mesma informação que a ***warning()*** para o *stderr* termina a

execução do programa com uma chamada à função *exit()* que tem como parâmetro de entrada o valor de *exitCode*.

Nos parâmetros *file* e *line*, devem especificar-se, respetivamente, o nome do ficheiro e a linha onde é efetuada a chamada à função. O valor destes parâmetros deve ser obtido através das macros do pré-processador __FILE__ e __LINE__ (escrevem-se com dois '_' antes e depois no nome). No parâmetro, *fmt*, pode passar-se uma mensagem juntamente com o valor de zero ou mais variáveis. Isto é possível porque se pode especificar uma *string* de formatação como a que se usa na função *printf()*. Desta forma, o número total de parâmetros das funções apresentadas é variável. Seguem-se alguns exemplos:

```
debug(__FILE__, __LINE__, "Valor do contador: %d", i);
warning(__FILE__, __LINE__, "%s só recebeu %d parâmetros",
argv[0], argc);
error(__FILE__, __LINE__, 1,"problemas …");
```

Figura 2 - Exemplos da chamada das funções debug, warning e error

Como se pode constatar nos exemplos, as macros do pré-processador __FILE__ e __LINE__ são constantemente usadas. Então porque é que estas macros não fazem parte das próprias funções? A explicação é simples, se tal fosse feito os valores dessas macros passariam a conter o nome e a linha do ficheiro onde as funções foram definidas em vez do nome e a linha do ficheiro de onde foram chamadas. Assim, as funções não devem ser chamadas diretamente, mas através de macros (cujo nome é igual ao das funções, mas em maiúsculas) que evitam a repetição de __LINE__ e __FILE__. Eis um exemplo da macro para a função ***debug()***:

```
#define DEBUG(...)  debug(__FILE__, __LINE__, __VA_ARGS__)
```

Figura 3 - Definição da macro DEBUG

O parâmetro __VA_ARGS__ permite que esta macro também suporte um número variável de parâmetros. Estes tipos de macros foram introduzidos na norma ISO para a linguagem de programação C em 1999, o gcc 2.96 e versões superiores já suportam esta característica.

O valor da variável *errno* (deve ser incluído o ficheiro *errno.h* - para mais informações consultar '*man errno*') é atribuído pelas **chamadas ao sistema** para indicar que algo correu mal nessa chamada. O seu valor só tem significado quando a chamada devolve erro, normalmente o valor -1. Se o seu valor for zero, significa que a chamada foi bem-sucedida. No entanto, a grande maioria das chamadas ao sistema não atribuem o valor

zero ao *errno* quando são bem-sucedidas. Isto pode criar situações que nos induzam em erro, por exemplo, se fizermos duas chamadas ao sistema, em que a primeira devolve erro e a segunda é bem-sucedida, mas só consultamos o valor de *errno* após a segunda chamada. Neste caso o valor de *errno* vai indicar-nos um erro, apesar de a segunda chamada ter sido bem-sucedida.

1.1. Código fonte dos ficheiros debug.c e debug.h

O código das funções apresentadas encontra-se nos ficheiros fonte *debug.c* e *debug.h* que a seguir se transcrevem (o código fonte destes ficheiros encontra-se na página da disciplina). Estas funções serão sempre incluídas nos programas desenvolvidos ao longo do semestre.

```
/**
* @file debug.h
* @brief Macros das funções de depuração
*/
#ifndef DEBUG_H
#define DEBUG_H

void debug(const char *file, const int line, char *fmt, ...);
void warning(const char *file, const int line, char *fmt, ...);
void error(const char *file, const int line,
     int exitCode, char *fmt, ...);

#define DEBUG(...) debug(__FILE__, __LINE__, __VA_ARGS__)
#define WARNING(...) warning(__FILE__, __LINE__, __VA_ARGS__)
#define ERROR(exitCode, ...) \
  error(__FILE__, __LINE__, (exitCode), __VA_ARGS__)

#endif      /* DEBUG_H */
```

Figura 4 - Listagem do debug.h

```
/**
* @file debug.c
* @brief Funções de depuração
*/
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <netdb.h>

#include "debug.h"
```

```
void debug(const char *file, const int line, char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    fprintf(stderr, "[%s@%d] DEBUG - ", file, line);
    vfprintf(stderr, fmt, ap);
    va_end(ap);
    fprintf(stderr, "\n");
    fflush(stderr);
}


void warning(const char *file, const int line, char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    fprintf(stderr, "[%s@%d] WARNING - ", file, line);
    vfprintf(stderr, fmt, ap);
    va_end(ap);
    fprintf(stderr, ": %s\n", strerror(errno));
    fflush(stderr);
}


void error(const char *file, const int line, int exitCode, char
*fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    fprintf(stderr, "[%s@%d] ERROR - ", file, line);
    vfprintf(stderr, fmt, ap);
    va_end(ap);
    fprintf(stderr, ": %s\n", strerror(errno));
    fflush(stderr);
    exit(exitCode);
}
```

Figura 5 - Listagem do *debug.c*

1.2. Como incluir código de depuração nos programas

Normalmente, quando um trabalho de programação é dado como concluído, o programador vai eliminar todas as linhas de código que usou para fazer depuração. Esta atitude não é correta! Duplica o trabalho do programador, corre o risco de eliminar linhas a mais e além disso pode vir a necessitar novamente dessa informação. Em vez disso o programador deve fazer uso das diretivas do pré-compilador: *#ifdef* e *#endif*, que lhe permitem definir partes do código que devem ser, ou não, incluídas na compilação. Segue-se um exemplo:

```
...
#ifdef SHOW_DEBUG
DEBUG (“o valor é %d”, contador);
```

```
...
#endif
...
```

Figura 6 - Exemplo do uso #ifdef ... #endif

Assim, sempre que se encontrar definido SHOW_DEBUG (poderia ser outro nome à escolha do programador) o código que se encontra entre o *#ifdef* e o *#endif* é incluído na compilação. Para facilitar a definição de SHOW_DEBUG podemos fazer uso da opção -D do gcc, por exemplo:

```
gcc –Wall –D SHOW_DEBUG programa.c –o executavel
```

Desta forma, e com uma programação cuidada, não é necessário "limpar" o código quando o programa estiver concluído. A opção -D do *gcc* irá criar uma macro com o nome fornecido e com valor 1.

1.3. Depuração de memória dinâmica

Quando se usa memória dinâmica (através de *malloc()*, *realloc()* e *free()*) podem facilmente serem cometidos erros, tais como a utilização de memória que já foi libertada ou o ultrapassar do tamanho alojado. A biblioteca C do Linux (designada *glibc*), permite a ativação de um mecanismo rudimentar de depuração através da definição da variável do ambiente MALLOC_CHECK_. Este mecanismo permite detetar uma grande quantidade de problemas relacionados com a utilização incorreta da memória dinâmica. O seu comportamento varia consoante o valor atribuído a MALLOC_CHECK_. Se o valor for 1 mostra uma mensagem, se for 2 aborta o programa, se for 3 combina o comportamento de 1 e 2. Para desligar este mecanismo deve atribuir-se o valor de 0 (zero). Segue-se um exemplo da linha de comando a usar para definir um valor para MALLOC_CHECK_:

```
export MALLOC_CHECK_=1
```

Figura 7 - Ativação da depuração de memória dinâmica

Se a execução da aplicação induzir uma incorreta utilização da memória dinâmica, serão afixadas mensagens de aviso.

```
$ ./teste_malloc
malloc: using debugging hooks
free(): invalid pointer 0xbffff750!
```

2. O depurador GDB

As técnicas anteriores nem sempre são suficientes ou eficazes na deteção de alguns problemas. Nesse caso será necessário recorrer ao depurador da GNU, o *gdb*. Embora a sua interface de modo texto afaste o programador do seu uso, o *gdb* é uma poderosa ferramenta, desenvolvida inicialmente por um dos gurus do código aberto, *Richard Stallman*.

Para ilustrar as potencialidades do *gdb*, foi criado o programa *teste_gdb* que contém erros e é terminado pelo sistema operativo sempre que é executado. O código fonte encontra-se listado de seguida, sendo de referir que para tirar partido do depurador, o código deve ser compilado com a **opção -g**, que leva a que o compilador inclua informação adicional no código gerado.

```
/**
* @file teste_gdb.c
* @brief Programa para exemplificar o uso do GDB
*/
#include <stdio.h>
#include <stdlib.h>
#include "debug.h"
#define C_NUM_STRINGS 2

void IniciaString(char *str)
{
    int i;

    for (i = 1; i <= 10; i++){
      *(str++) = 'x';
    }
    *str = '\0';
}

int main(void)
{
    int j, i;
    char *strs[C_NUM_STRINGS];
    size_t Tam;

    Tam = sizeof(char) * 11;
    for (i = 1; i <= C_NUM_STRINGS; i++) {
      strs[i] = malloc(Tam);
      if (strs[i] == NULL)
        ERROR(1, "Impossível alojar %d bytes", Tam);
    }

    for (i = 1; i <= C_NUM_STRINGS; i++) {
      IniciaString(strs[i]);
    }

    for (j = 1; j <= C_NUM_STRINGS; j++) {
      printf("string nº%d - '%s'\n", j, strs[j]);
    }
```

```
    return 0;
}
```

Figura 8 - Listagem do programa teste_gdb

Quando se corre o programa ele é terminado abruptamente pelo Sistema Operativo e resulta um *coredump* (se essa opção se encontrar ativa). O *coredump* é um ficheiro designado *core* que contém uma imagem da memória do processo no momento em que ocorreu o erro. Consoante a configuração do sistema Linux, o nome do ficheiro *core* pode ter como sufixo o PID do processo (e.g., *core*.1875). Para que esse ficheiro seja criado é necessário configurar a *shell* que está a ser usada. No caso da *bash* tal é feito através do comando:

```
$ ulimit –c unlimited
```

Nota: para mais informações consultar *man bash*.

Para tentar descobrir o problema do programa executa-se o *gdb* do seguinte modo:

```
gdb teste_gdb NOME_DO_FICHEIRO_CORE
```

Em que *teste_gdb* é o nome do ficheiro executável e NOME_DO_FICHEIRO_CORE é o nome do ficheiro *coredump*. Após algumas mensagens e avisos, o *gdb* exibe as seguintes linhas:

```
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x080486df in main () at teste_gdb.c:36
36              if (strs[i] == NULL)
```

A expressão *Segmentation fault* indica que provavelmente a terminação abrupta do programa se deveu a um acesso ilegal à memória. Além desta informação, existe também a transcrição da linha 36 do código do *teste_gdb*, que indica o local onde se encontrava a execução do programa na altura em que ocorreu o erro. A informação obtida até agora ainda não é suficiente para determinar a razão do erro. O *gdb* encontra-se à espera de comandos na sua linha de comandos. Pode listar-se o código fonte afetado através do comando *list*.

```
(gdb) list
31          size_t Tam;
32
33          Tam = sizeof(char) * 11;
34          for (i = 1; i <= C_NUM_STRINGS; i++) {
35              strs[i] = malloc(Tam);
36              if (strs[i] == NULL)
37                  ERROR(1, "Impossível alojar %d bytes", Tam);
38          }
39          for (i = 1; i <= C_NUM_STRINGS; i++) {
```

Este comando lista 10 linhas, 5 antes da linha onde foi detetado o erro (linha 36), e 4 depois. Dado o elevado número de comandos e respetivas opções, serão referidos apenas alguns comandos. O leitor interessado em explorar as potencialidades do *gdb* deve consultar o seu manual (http://www.gnu.org/software/gdb/documentation/). No entanto todos os comandos do *gdb* possuem uma sintética documentação, acedida através do comando *help* seguido do nome do comando, por exemplo:

```
(gdb) help list
```

O passo seguinte será analisar o valor das variáveis existentes no fragmento de código considerado. Através do comando *print* acede-se ao valor das variáveis:

```
(gdb) print Tam
$1 = 11
(gdb) print i
$2 = 134519968
(gdb)
```

O valor de i encontra-se claramente fora da gama que definimos no ciclo (1 até C_NUM_STRINGS). O valor de i (variável local) foi corrompido, muito possivelmente pelo uso indevido de espaço na pilha (i é uma variável local). Algo se passou que afetou a pilha, para obter mais informação recorre-se ao comando *frame* que exibe as *frames* lógicas existentes na pilha (por *frame* lógicas entende-se as zonas da pilha afetas a cada chamada de função):

```
(gdb) frame
#0  0x080486df in main () at teste_gdb.c:36
36              if (strs[i] == NULL)
```

É possível aceder a uma informação mais vasta sobre a *frame* corrente através do comando *info frame*:

```
(gdb) info frame
Stack level 0, frame at 0xbffffab8:
 eip = 0x80486df in main (teste_gdb.c:36); saved eip 0x42017499
 called by frame at 0xbffffaf8
 source language c.
 Arglist at 0xbffffab8, args:
 Locals at 0xbffffab8, Previous frame's sp is 0x0
 Saved registers:
  ebp at 0xbffffab8, eip at 0xbffffabc
```

Contudo, neste caso concreto essa informação não ajuda a resolver o problema. O comando *info* é um dos mais poderosos do *gdb*, possuindo muitas opções. Uma dessas opções é a *locals* que possibilita a visualização do valor de todas as variáveis locais:

```
(gdb) info locals
j = 134519884
i = 134519968
strs = {0xbffffac8 ",ûÿ¿r\204\004\bP\212\004\b", 0x8049c90 ""}
Tam = 11
```

A variável i apresenta um valor estranho (que já se havia detetado), mas repare-se no conteúdo do vetor de ponteiros para caracteres *strs*. O primeiro elemento apresenta um endereço (0xbffffac8) substancialmente diferente do seguinte. Muito possivelmente esse primeiro elemento não foi iniciado. Para clarificar a situação, procede se ao *print* dos elementos individuais do vetor, i.e. strs[0], strs[1] e [2] (este último já não pertence ao vetor):

```
(gdb) print strs[0]
$4 = 0xbffffac8 ",ûÿ¿r\204\004\bP\212\004\b"
(gdb) print strs[1]
$5 = 0x8049c90 ""
(gdb) print strs[2]
$6 = 0x8049ca0 ""
```

Confirma-se que o elemento strs[0] pertence a uma gama diferente dos restantes. Mas, repare se em strs[2]. O valor pertence à gama de endereços do strs[1]! Isto permite concluir o que se passou de errado: em vez de se tratar o vetor de 0 até 2 exclusive, está

a ser tratado de 1 a 2 inclusive. Este tipo de erro é muito frequente quando se transita da linguagem Pascal (em que os vetores são indexados a partir de 1) para a linguagem C. Assim, a escrita na zona de memória strs[2], para a qual não foi reservado espaço, corrompeu um valor adjacente, neste caso a variável i. Ao visualizar o endereço de i em hexadecimal (através da opção '/x') obtém-se:

```
(gdb) print /x i
$9 = 0x8049ca0
```

Desta forma verifica-se que o endereço de i e strs[2] é o mesmo: 0x8049ca0. Ou seja, ao ser atribuído (indevidamente) um valor a strs[2] na realidade estava a escrever-se no espaço da pilha afeto à variável i, corrompendo o seu valor. Note-se que essa corrupção da variável de iteração que ocorreu na linha 36 levou a que o valor de i empregue na comparação da linha seguinte conduzisse ao acesso de zona de memória que sai fora do espaço de endereçamento do nosso processo. Isto levou ao término do referido processo por parte do sistema operativo.

Como já foi referido, o *gdb* é uma ferramenta extremamente poderosa e de indiscutível utilidade. Assim, convida-se o leitor a explorar mais as suas potencialidades através do comando *help* (dentro do próprio *gdb*) ou através de *info gdb* para obter uma informação mais detalhada. É de referir ainda que existem algumas interfaces gráficas para esta ferramenta, nomeadamente o *ddd* ('Data Display Debugger' que é possível consultar no endereço: http://www.gnu.org/software/ddd/) que apresenta uma interessante potencialidade gráfica, assim como o *gnome nemiver*.