Depuração

Videos desta aula:

Parte 1: GDB

,

Parte 2: memória

e

Parte 3: tracing e outros

.

Depurar significa “purificar” ou “limpar”. Ao programar em C, frequentemente temos necessidade de depurar a lógica de um programa, para resolver erros de execução, de alocação/liberação de memória ou de desempenho. Esta página descreve algumas ferramentas disponíveis para tal propósito.

Preparação

O primeiro passo para a depuração de um programa executável é compilá-lo de forma a **incluir no executável os símbolos** necessários ao processo de depuração (como os nomes das variáveis e funções, referências às linhas do código fonte, etc). Isso é feito adicionando a opção `-g` ao comando de compilação.

O comando de compilação do arquivo

fatorial.c

, por exemplo, seria:

```
$ gcc -g -o fatorial fatorial.c
```

Feito isso, o programa está pronto para ser depurado com as ferramentas apresentadas a seguir.

Depuração de execução: GDB

O depurador padrão para a linguagem C no Linux é o GDB (GNU Debugger). O GDB é um depurador em modo texto com muitas funcionalidades, mas relativamente complexo de usar para os iniciantes.

Para iniciar uma depuração, basta invocar o GDB com o executável (compilado com a opção `-g`):

```
$ gdb fatorial
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
... (msgs diversas)
Lendo símbolos de fatorial...concluído.
(gdb) run
Starting program: /home/prof/maziero/fatorial
O fatorial de 4 é 24
O fatorial de 10 é 3628800
```

```
[Inferior 1 (process 24652) exited normally]
(gdb) quit
```

O *prompt* do GDB aceita diversos comandos, entre eles o *run* e o *quit*, ilustrados acima. Os comandos mais básicos disponíveis são:

comando	**ação**	**exemplos**

`run`
`r`
	inicia a execução (*run*)	`run`
		`r < dados.dat`
		`r > saida.txt`

`list` `l`	lista linhas do código-fonte	`l 23`
`b`	cria um ponto de parada (*breakpoint*)	`b 17` `b main`
`c`	continua após o ponto de parada	`c`
`s`	avança para a próxima linha de código (*step*)	`s`
`n`	avança para a próxima linha de código (*next*), sem parar dentro das funções	`n`
`p`	imprime o valor de uma variável ou expressão	`p soma` `p /x soma`
`watch`	avisa quando uma variável muda de valor	`watch i`
`disp`	mostra o valor de uma variável ou expressão a cada pausa (*display*)	`disp soma`
`set variable`	Ajusta o valor de uma variável do programa em análise	`set variable soma = 100`
`bt`	Mostra a posição atual do programa (*backtrace*), incluindo as funções ativas no momento	`bt`
`frame`	Seleciona o frame de execução a analisar (o nível de chamada de função)	`frame 3`
`^x^a`	alterna entre as interfaces padrão e NCurses	

Uma relação mais extensa de comandos pode ser encontrada neste GDB Reference Card.

O GDB proporciona uma forma fácil de localizar erros fatais de memória, como *Segmentation Fault* e outros. Basta executar o programa no GDB (usando *run*) até ocorrer o erro. Quando este ocorrer, o número da linha será informado e os valores das variáveis envolvidas podem ser inspecionados para encontrar o erro.

Para testar, use o GDB para encontrar os erros de acesso à memória neste programa: memerror.c.

Vários guias de uso do GDB podem ser encontrados nos links abaixo:

- Beej's Quick Guide to GDB
- A GDB tutorial
- Debugging with GDB
- Guide to faster, less frustrating debugging
- Using GNU's GDB debugger
- Tutorial de GDB (em português)

Como interfaces alternativas para o GDB, existe o modo *NCurses*, que pode ser ativado invocando o GDB com a opção `-tui` ou pelo comando `^x^a` (*ctrl-x ctrl-a*). Outras interfaces disponíveis em modo texto são o `cgdb`, que usa comandos similares aos do VI, e o modo GDB do EMACS.

O GDB está integrado em IDEs (*Integrated Development Environments*) gráficos como *Visual Studio Code* e

Eclipse. Além disso, existem interfaces gráficas para o GDB, como o Data Display Debugger (ddd), Nemiver e Gede.

Despejo de memória

A maioria dos sistemas UNIX permite salvar em um arquivo o conteúdo da memória de um programa em execução (processo), quando este é interrompido por um erro. Esse arquivo se chama `core file` (despejo de memória) e pode ser aberto por depuradores, para auxiliar na compreensão da causa do erro.

O despejo de memória é útil para identificar a causa de erros fatais em programas, sobretudo para **erros esporádicos ou difíceis de reproduzir**.

Para usar esse recurso deve-se seguir os seguintes passos (usando como exemplo o arquivo memerror.c):

1. Habilitar a geração de arquivo *core* no terminal atual:

```
$ ulimit -c unlimited
```

2. Compilar o programa com o flag de depuração (`-g`):

```
$ cc -g memerror.c -o memerror
```

3. Lançar o programa:

```
$ memerror
```

4. Quando o programa abortar por erro, será gerado um arquivo *core* no diretório corrente:

5.
```
$ ls -l
-rw-------  1 maziero maziero 262144 Nov 30 14:57 core
-rwxrwxr-x  1 maziero maziero  11104 Nov 30 14:56 memerror
-rw-rw-r--  1 maziero maziero    839 Nov 30 14:56 memerror.c
```

6. Alternativamente, pode-se **forçar o encerramento** (e a consequente geração do arquivo *core*) através de um sinal `SIGQUIT` (sinal n° 3). Esse sinal deve ser enviado ao processo através do comando `kill` (a partir de outro terminal, se necessário):

```
$ kill -3 PID
```

   onde PID é o identificador numérico do processo a ser encerrado.

7. Uma vez obtido o arquivo *core*, basta abri-lo através do depurador (GDB) e analisar seu estado:

```
$ gdb memerror core
```

Em algumas versões de Linux (Ubuntu, etc) o serviço *Apport* trata os erros fatais e impede a geração de arquivos *core*. Pode-se desligar temporariamente esse serviço através do comando `sudo service apport stop`, para obter os arquivos *core*.

Depuração de memória

Um dos problemas mais frequentes (e de depuração mais difícil) na programação em C é o uso incorreto da memória. Situações como acesso a posições inválidas de vetores ou matrizes (*buffer overflow*), uso de ponteiros não inicializados, não-liberação de memória dinâmica (*memory leaks*) podem gerar comportamentos erráticos difíceis de depurar.

Há várias ferramentas para auxiliar na depuração de problemas de memória, vistas na sequência.

Análise estática

Ferramentas de análise estática examinam o código-fonte de uma forma mais detalhada que o compilador, permitindo encontrar diversos erros que podem passar despercebidos, como índices de vetores fora da faixa válida.

Algumas ferramentas disponíveis para análise estática de código:

- `cppcheck`: verificador estático de código C/C++ (recomenda-se usar flag `--enable=all`)
- `splint`: idem

Flags do compilador

Opções do GCC para depuração de memória:

- `-fsanitize=address`: ativa o AddressSanitizer, um detector de erros de memória em tempo de execução. O código do executável é instrumentado (são adicionadas instruções) para verificar erros de acesso a posições inválidas de memória.
- `-fcheck-pointer-bounds`: ativa a verificação de limites de ponteiros.
- `-fstack-protector`: gera código adicional para verificar a integridade da pilha (*flag* habilitado por default).

As verificações adicionadas por esses *flags* são efetuadas a cada acesso à memória, por isso têm um forte impacto no desempenho e no uso de memória do executável. Então, só devem ser usadas durante o processo de desenvolvimento e nunca no produto final.

Bibliotecas de depuração

São bibliotecas que instrumentam as rotinas de alocação/liberação de memória, permitindo depurar erros relacionados ao uso de memória dinâmica, como *memory leaks*, *double free* e *use after free*.

- DMalloc
- Electric Fence

Depuradores de memória

Ferramentas como Mtrace e Valgrind realizam a análise dinâmica do programa (ou seja, durante sua execução) para encontrar erros relacionados à memória.

A ferramenta mais usada e popular é sem dúvida o **Valgrind**. Ele é particularmente útil para encontrar

problemas de vazamento de memória (*memory leaks*), mas pode realizar diversas análises envolvendo memória, caches e *profiling* da execução.

Algumas opções usuais do Valgrind são:

- `--tool=toolname` : escolhe a ferramenta de análise, que pode ser:
  - `memcheck` : análise de acesso à memória (default)
  - `cachegrind` : análise de uso dos caches
  - `exp-sgcheck` : análise mais detalhada de variáveis globais e do stack
  - ...
- `--leak-check=full` : relatório detalhado sobre *memory leaks*

Eis abaixo um programa com alguns erros de memória frequentes:

errors.c

```c
#include <stdio.h>
#include <stdlib.h>

#define VETSIZE 100

int main()
{
  int *vet1, *vet2 ;
  int x ;

  vet1 = malloc(VETSIZE * sizeof (int)) ;
  vet2 = malloc(VETSIZE * sizeof (int)) ;

  // erro 1: acesso a uma posição fora do vetor (buffer overflow)
  vet1[VETSIZE] = 0 ;

  // erro 2: leitura de uma variável não inicializada
  if (x == 0)
    printf ("x vale zero\n") ;

  free (vet2) ;

  // erro 3: liberar duas vezes a mesma área (double free)
  free (vet2) ;

  // erro 4: usar uma área após tê-la liberado (use after free)
  vet2[0] = 0 ;

  // erro 5: a área de vet1 não foi liberada (memory leak)
}
```

Primeiro, deve-se compilar o código com a opção `-g`. Em seguida, executar o Valgrind com as opções de depuração de memória:

```
$ gcc -Wall -g errors.c -o errors
$ valgrind --leak-check=full ./errors
```

O relatório gerado pelo Valgrind pode ser extenso e deve ser analisado com atenção. A seguir destacamos os trechos do relatório relativos aos quatro erros de memória do código acima.

O trecho abaixo diz respeito ao **erro 1**:

```
==29519== Invalid write of size 4
==29519==    at 0x1086F8: main (errors.c:15)
==29519==  Address 0x522d1d0 is 0 bytes after a block of size 400 alloc'd
==29519==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==29519==    by 0x1086DB: main (errors.c:11)
```

- escrita de 4 bytes inválida, na função `main`, na linha 15 de `errors.c`
- essa escrita ocorreu depois do bloco de 400 bytes que foi alocado na linha 11

O trecho abaixo diz respeito ao **erro 2**:

```
==29519== Conditional jump or move depends on uninitialised value(s)
==29519==    at 0x108702: main (errors.c:18)
```

- a condicional na linha 18 depende de um valor não inicializado (pode conter lixo)

O trecho abaixo diz respeito ao **erro 3**:

```
==29519== Invalid free() / delete / delete[] / realloc()
==29519==    at 0x4C30D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==29519==    by 0x108727: main (errors.c:24)
==29519==  Address 0x522d210 is 0 bytes inside a block of size 400 free'd
==29519==    at 0x4C30D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==29519==    by 0x10871B: main (errors.c:21)
==29519==  Block was alloc'd at
==29519==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==29519==    by 0x1086E9: main (errors.c:12)
```

- `free()` inválido na linha 24
- a área de 400 bytes em questão já foi liberada na linha 21
- essa área havia sido alocada na linha 12.

O trecho abaixo diz respeito ao **erro 4**:

```
==29519== Invalid write of size 4
==29519==    at 0x10872C: main (errors.c:27)
==29519==  Address 0x522d210 is 0 bytes inside a block of size 400 free'd
==29519==    at 0x4C30D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==29519==    by 0x10871B: main (errors.c:21)
==29519==  Block was alloc'd at
==29519==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==29519==    by 0x1086E9: main (errors.c:12)
```

- escrita de 4 bytes inválida na linha 27
- essa área de memória de 400 bytes já foi liberada na linha 21
- essa área havia sido alocada na linha 12.

O trecho abaixo diz respeito ao **erro 5**:

```
==29519== 400 bytes in 1 blocks are definitely lost in loss record 1 of 1
==29519==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==29519==    by 0x1086DB: main (errors.c:11)
```

- *memory leak*: a área de memória alocada na linha 11 não foi liberada ao encerrar o programa.

Como exercício, analise o programa memerror.c usando o Valgrind.

Tracing

Uma ferramenta útil para auxiliar na depuração de programas é o utilitário `strace`, que permite listar as chamadas de sistema efetuadas por um executável qualquer. Por não precisar de opções especiais de compilação, nem do código-fonte do executável, é uma ferramenta útil para investigar o comportamento de executáveis de terceiros. Além disso, o `strace` pode analisar processos já em execução (através da opção `-p`).

Eis um exemplo (abreviado) da execução de `strace` sobre o programa fatorial.c:

```
$ strace ./fatorial
execve("./fatorial", ["./fatorial"], 0x7fff761a4a10 /* 63 vars */) = 0
brk(NULL)                               = 0x5628852a9000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=141702, ...}) = 0
mmap(NULL, 141702, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f39dc9c3000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
...
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
brk(NULL)                               = 0x5628852a9000
brk(0x5628852ca000)                     = 0x5628852ca000
write(1, "O fatorial de 4 \303\251 24\n", 22O fatorial de 4 é 24
) = 22
write(1, "O fatorial de 10 \303\251 3628800\n", 28O fatorial de 10 é 3628800
) = 28
exit_group(0)                           = ?
+++ exited with 0 +++
```

De forma similar, o comando `ltrace` gera uma listagem sequencial de todas as chamadas de biblioteca geradas durante a execução de um programa:

```
$ ltrace ./fatorial
printf("O fatorial de %d \303\251 %ld\n", 4, 24O fatorial de 4 é 24
) = 22
printf("O fatorial de %d \303\251 %ld\n", 10, 3628800O fatorial de 10 é 3628800
) = 28
+++ exited (status 0) +++
```

Performance profiling

Além da depuração propriamente dita, em busca de erros, por vezes torna-se necessário analisar o comportamento temporal do programa. Para realizar essa análise pode-se utilizar o *GNU Profiler* (`gprof`). O `gprof` permite verificar:

- o tempo gasto em cada função
- o grafo de chamadas (call graph)
    - que funções são chamadas por que funções
    - que funções chamam outras funções
- quantas vezes cada função é chamada
- ...

Para realizar o *profiling* de um executável, é necessário inicialmente compilá-lo com o flag adequado (`-pg`) e em seguida executá-lo:

```
$ gcc -pg -g -o fatorial.c
$ ./fatorial
```

A execução irá gerar um arquivo binário `gmon.out`, que contém os dados de *profiling*. Esse arquivo é usado pelo utilitário `gprof` para gerar as estatísticas desejadas:

```
$ gprof fatorial gmon.out
```

O relatório de *profilling* do programa

demo-profile.c

obtido com o `gprof` pode ser encontrado

neste arquivo

. O grafo de chamadas (*call graph*) pode ser visualizado de forma gráfica através da ferramenta Gprof2dot.

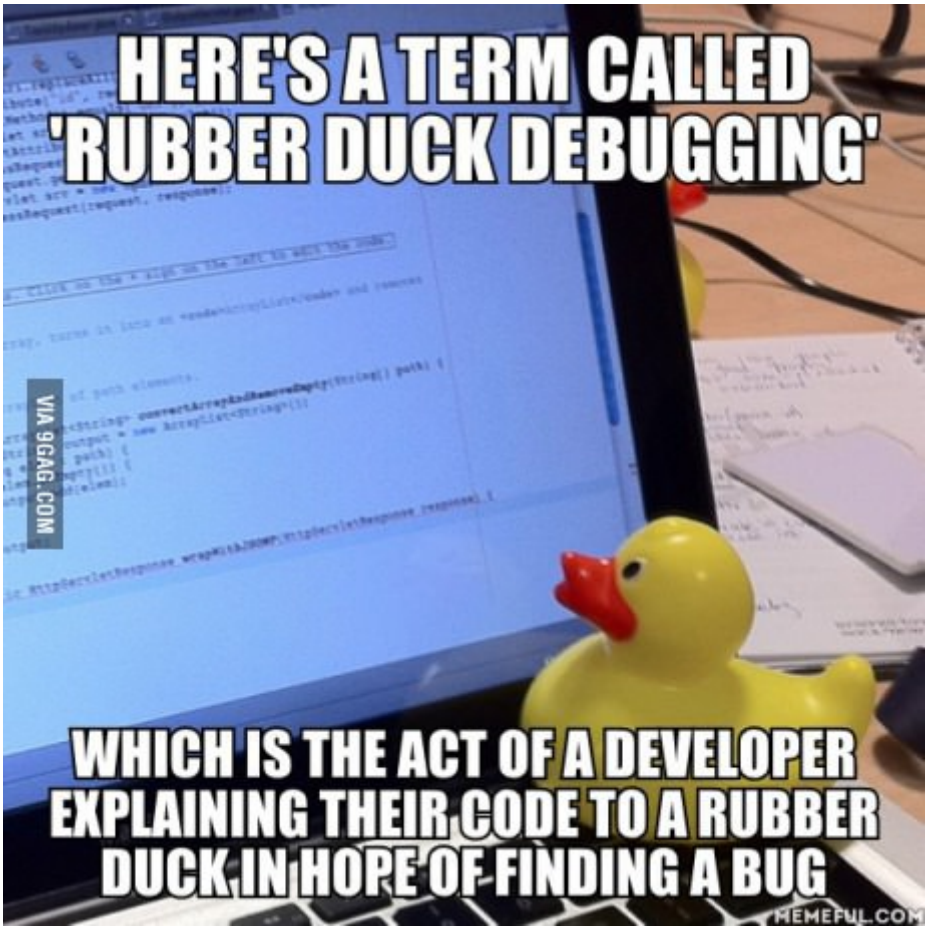
Mais detalhes e opções de relatório podem ser obtidas no manual GNU gprof.

O Valgrind também permite realizar *profiling*, através de sua ferramenta interna `callgrind` e do visualizador externo KCachegrind.

Rubber duck debugging

Se nada mais funcionar, **explique seu código a alguém** !

> *Outra técnica efetiva [de depuração] é explicar seu código a alguém. Isto vai geralmente fazer você explicar o bug para si mesmo. Às vezes bastam algumas frases, seguidas de um envergonhado "Esqueça, eu estou vendo o que está errado. Desculpe incomodá-lo". Isto funciona notavelmente bem; você pode até mesmo usar não-programadores como ouvintes. O centro de computação de uma universidade mantinha um ursinho de pelúcia perto do "help desk". Alunos com bugs misteriosos deviam explicá-los ao ursinho antes de poder falar com um assistente humano.*
>
> *The Practice of Programming*, Brian Kernighan and Rob Pike, 1999.



Outras ferramentas

- `strip`: permite remover os símbolos e código não usado de um executável ou arquivo-objeto, diminuindo consideravelmente seu tamanho (mas impedindo futuras depurações no mesmo).
- `diff`: permite comparar dois arquivos ou diretórios (recursivamente), apontando as diferenças entre seus conteúdos. Muito útil para comparar diferentes versões de árvores de código fonte.
- `patch` : permite aplicar um arquivo de diferenças (gerado pelo comando `diff`) sobre uma árvore de arquivos, modificando os arquivos originais de forma a obter uma nova árvore. Muito usado para divulgar novas versões de códigos-fonte muito grandes.
- `grep`: permite encontrar linhas em arquivos contendo uma determinada string ou expressão regular. Pode ser muito útil para encontrar trechos de código específicos em grandes volumes de código.
- `indent`: reformatador de código-fonte, aceita diversas opções de endentação automática.