Uma pequena introdução ao gdb

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

O que é o gdb?

O **gdb** (GNU project debugger) é um depurador ("debugger") de programas para as várias linguagens compiladas pelo gcc.

Um processo de depuração ("debugging") é tipicamente iniciado para perceber a causa de um mau funcionamento de um programa. Podemos:

- Inspecionar interativamente o estado do programa (valores de variáveis, stack de execução, …) em ligação ao código fonte de um programa.
- Executar um programa passo-a-passo ou fazê-lo parar em pontos especiais de interesse.

Exemplo

Programa `pal.c` para testar se strings dadas como argumentos são ou não palíndromos (i.e. se se lêem de igual forma da esquerda para a direita ou da direita para a esquerda, ex. como em `"abcba"`):

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
```

```
bool is_palindrome(const char s[]) {
  int l = 0;
  int r = strlen(s) - 1;
  while (l != r) {
    if (s[l] != s[r])
      return false;
    l++;
    r--;
  }
  return true;
}

int main(int argc, char** argv) {
  for (int i = 1; i < argc; ++i)
    printf("Is \"%s\" a palindrome? %s\n",
      argv[i],
      is_palindrome(argv[i]) ? "yes" : "no");
   return 0;
}
```

Há um bug …

Há um “bug” no programa, `"abba"` (na 3ª linha do output) deveria ser considerado um palíndromo …

```
$ ./pal aba abc abba ziniz abcb
Is "aba" a palindrome? yes
Is "abc" a palindrome? no
Is "abba" a palindrome? no
Is "ziniz" a palindrome? yes
Is "abcb" a palindrome? no
```

Compilação de programas para debugging

Para usar o gdb devemos empregar a opção `-g` na compilação com o gcc:

```
$ gcc -g -Wall pal.c -o pal
```

Desta forma, o binário gerado incluirá informação de "debugging" que o gdb poderá usar.

Iniciando o gdb

Podemos depois iniciar o gdb para o programa gerado. Teremos uma linha de comando onde podemos executar comandos de debugging, desde logo o comando `run args`, ou de forma abreviada `r args`, para iniciar o programa com os argumentos desejados.

```
$ gdb ./pal
GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
...
Reading symbols from ./prog...done.
(gdb) run abba
Starting program: /home/runner/gdb/pal abba
Is "abba" a palindrome? no
[Inferior 1 (process 287) exited normally]
```

O programa executa até ao fim neste caso.

Definição de pontos de paragem

O nosso interesse será perceber o que acontece durante a execução `is_palindrome`.

Podemos definir um ponto de paragem - um **“breakpoint”** - no programa à entrada da função `is_palindrome` e reiniciar o programa com `run` de novo. A execução para em `is_palindrome`.

```
(gdb) br is_palindrome
Breakpoint 1 at 0x55fa74800696: file pal.c, line 6.
(gdb) run abba
Starting program: /home/runner/gdb/pal abba
Breakpoint 1, is_palindrome (
    s=0x7ffc4985c022 "abba") at pal.c:6
6     int l = 0;
```

Inspecionando o estado do programa

Podemos nesta altura listar o código fonte em contexto com `list` …

```
(gdb) list
1   #include <stdio.h>
2   #include <stdbool.h>
3   #include <string.h>
4
5   bool is_palindrome(const char s[]) {
6     int l = 0;
7     int r = strlen(s) - 1;
8     while (l != r) {
9       if (s[l] != s[r])
10        return false;
```

inspecionar a stack de execução de funções com `where` …

```
(gdb) where
#0  is_palindrome (s=0x7fff563dd022 "abba") at pal.c:9
#1  0x000056351860072a in main (argc=2, argv=0x7fff563db6d8)
    at pal.c:21
```

e inspecionar o conteúdo da variável `s` com o comando `print`:

```
(gdb) print s
$1 = 0x7ffc4985c022 "abba"
```

Mais um breakpoint …

Será interessante observar os valores de `l` e `r` à entrada do corpo do ciclo, i.e. na linha 9. Para tal definimos novo ponto de paragem fornecendo a linha de código como argumento desta vez …

```
(gdb) br 9
Breakpoint 2 at 0x6b1: file pal.c, line 9.
```

e usamos de seguida o comando `continue` para continuar até até ao próximo ponto de paragem, que ocorrerá na linha 9 como pretendido.

```
(gbd) cont
Breakpoint 2, is_palindrome (s=0x7ffd203f3022 "abba") at pal.c:9
9       if (s[l] != s[r])
```

Nesta altura podemos imprimir os valores de `l`, `r`, `s[l]` e `s[r]` com o comando `print` novamente. Em alternativa podemos usar o comando `display` que imprimirá os valores automaticamente sempre que o programa parar.

```
(gdb) display l
1: l = 0
(gdb) display r
2: r = 3
(gdb) display s[l]
3: s[l] = 97 'a'
```

```
3: s[l] = 97 'a'
(gdb) display s[r]
4: s[r] = 97 'a'
(gdb) cont
Breakpoint 2, is_palindrome (s=0x7ffd203f3022 "abba") at pal.c:9
9       if (s[l] != s[r])
1: l = 1
2: r = 2
3: s[l] = 98 'b'
4: s[r] = 98 'b'
```

Descortinando o bug …

Continuando a inspecionar a evolução das variáveis, começamos a perceber o comportamento “estranho” do programa … observamos que é que possível chegar a um estado em que `l > r`. Não era suposto o ciclo parar?

```
(gdb) cont
Breakpoint 2, is_palindrome (s=0x7ffd203f3022 "abba") at pal.c:9
9       if (s[l] != s[r])
1: l = 2
2: r = 1
3: s[l] = 98 'b'
4: s[r] = 98 'b'
```

As iterações sucessivas indicam que `l` e `r` vão tomar valores de índices inválidos sobre array `s`, i.e., temos “buffer overflows”. Abaixo podemos ver que a função irá executar até que `l = 5` e `r = -2` e que teremos `s[l] = 'L'` e `s[r] = 'l'` !!!

(Nota: como o comportamento de C é indefinido no caso de “buffer overflows”, outras execuções para o mesmo input (`"abba"`) poderão dar resultados diferentes! Experimente no seu computador … com boa probalidade isso acontecerá!)

```
(gdb) cont
Continuing.

Breakpoint 2, is_palindrome (s=0x7ffd203f3022 "abba") at pal.c:9
9       if (s[l] != s[r])
1: l = 3
2: r = 0
3: s[l] = 97 'a'
4: s[r] = 97 'a'
(gdb) cont
Continuing.

Breakpoint 2, is_palindrome (s=0x7ffd203f3022 "abba") at pal.c:9
9       if (s[l] != s[r])
1: l = 4
2: r = -1
3: s[l] = 0 '\000'
4: s[r] = 0 '\000'
(gdb) cont
Continuing.

Breakpoint 2, is_palindrome (s=0x7ffd203f3022 "abba") at pal.c:9
9       if (s[l] != s[r])
1: l = 5
2: r = -2
3: s[l] = 76 'L'
4: s[r] = 108 'l'
```

Prosseguindo para a próxima instrução com o comando `next`, percebemos que `is_palindrome` realmente retorna `false` ...

```
(gdb) next
10        return false;
```

Em conclusão, qual é o bug ?

```
while (l != r) {
  if (s[l] != s[r])
    return false;
  l++;
  r--;
}
```

Para palíndromos de tamanho par, chegamos necessariamente a uma situação em que `l > r`. A iteração cái fora do array comparando posições de memória que nada têm a ver com a string passada como argumento.

Deveríamos ter isso sim `l < r` como condição de paragem, isto é:

```
while (l < r) {
  if (s[l] != s[r])
    return false;
  l++;
  r--;
}
```

Sumário de alguns comandos gdb

Inspeção do estado do programa:

- `print expr`: imprime valor de expressão `expr`;
- `display expr`: imprime valor de `expr` sempre que o programa pára na sessão de debuggin;
- `watch expr`: monitoriza valor `expr` e pára execução após cada alteração ao seu valor;
- `list`: lista código fonte em contexto;

- `where`: mostra estado da stack de execução;

Manipulação de pontos de paragem:

- `break x`: define novo ponto de paragem na função ou linha `x`;
- `delete n`: apaga ponto de paragem `n` (um número);

Fluxo de execução:

- `run args`: inicia programa com argumentos dados;
- `next`: passa à próxima linha do programa;
- `step`: passa à próxima linha do programa, mas entra dentro de nova função se houver uma chamada a função;
- `continue`: continua até próximo ponto de paragem

Outros:

- `help`: mostra informação geral de ajuda;
- `help comando`: obtém informação de ajuda em relação a determinado comando;
- `file prog`: carrega executável `prog` para debugging;
- `quit`: sai do gdb;

Mais informação

- GDB Quick reference - referência rápida de comandos
- Vídeo: Introduction to GDB - a tutorial - Harvard CS50
- GDB User Manual