

Depuração de código

Aprendendo a depurar código em nível de Assembly

O termo depuração (*debugging*) é usado na área da computação para se referir ao ato de procurar e corrigir falhas (*bugs*) em softwares. A principal ferramenta, embora não única, usada para essa tarefa é um tipo de software conhecido como depurador (*debugger*). Essa ferramenta basicamente dá ao programador a possibilidade de controlar a execução de um programa enquanto ele pode ver informações sobre o processo em tempo de execução.

Existem depuradores que meramente exibem o código-fonte do programa e o programador acompanha a execução do código vendo o código-fonte do projeto. Mas existe uma categoria de depuradores que exibem o *disassembly* do código do programa e o programador é capaz de ver a execução do código acompanhando as instruções em Assembly.

Este capítulo tem por objetivo dar uma noção básica de como depuradores funcionam e ensinar a usar algumas ferramentas para depuração de código.

Ferramentas

O conteúdo será principalmente baseado em ferramentas sendo utilizadas em ambiente Linux, porém a maior parte do conteúdo é reaproveitável no Windows.

Códigos de exemplo serão escritos em C e compilados com o GCC, bem como alguns serão escritos diretamente em Assembly e usando o *assembler* NASM.


Entendendo os depuradores

Entendendo os conceitos principais sobre um depurador e como eles funcionam.

Depuradores (*debuggers*) são ferramentas que atuam se conectando (*attaching*) em processos para controlar e monitorar a execução dos mesmos. Isso é possível por meio de recursos que o próprio sistema operacional provém, no caso do Linux por meio da `syscall ptrace`.

O processo que se conecta é chamado de *tracer* e o processo conectado é chamado de *tracee*. Essa conexão é chamada de *attach* e é feita em uma *thread* individual do processo. Quando o depurador faz *attach* em um processo ele na verdade está fazendo *attach* na *thread* principal do processo.

Processos

 As *threads* são tarefas individuais em um processo. Cada *thread* de um processo executa um código diferente de maneira concorrente em relação as outras *threads* do mesmo processo.

Um processo é basicamente a imagem de um programa em execução. Uma parte do sistema operacional conhecida como *loader* (ou *dynamic linker*) é a responsável por ler o arquivo executável, mapear seus códigos e dados na memória, carregar dependências (bibliotecas) resolvendo seus símbolos e iniciar a execução da *thread* principal do processo no código que está no endereço do *entry point* do executável. Onde *entry point* se trata de um endereço armazenado dentro do arquivo executável e é o endereço onde a *thread* principal inicia a execução.

O depurador tem acesso a memória de um processo e pode controlar a execução das *threads* do processo. Ele também tem acesso a outras informações sobre o processo, como o valor dos registradores em uma *thread* por exemplo.

Context switch


Do ponto de vista de cada *thread* de um processo ela tem exclusividade na execução de código no processador e no acesso a seus recursos. Inclusive em Assembly usamos registradores do processador diretamente sem nos preocuparmos com outras *threads* (do mesmo processo ou de outros) usando os mesmos registradores "ao mesmo tempo".

Cada núcleo (*core*) do processador têm um conjunto individual de registradores, mas é comum em um sistema operacional moderno diversas tarefas estarem concorrendo para executar em um mesmo núcleo.

Uma parte do sistema operacional chamada de *scheduler* é responsável por gerenciar quando e qual tarefa será executada em um determinado núcleo do processador. Isso é chamado de escalonamento de processos (*scheduling*) e quando o *scheduler* suspende a execução de uma tarefa para executar outra isso é chamado de **troca de contexto** ou **troca de tarefa** (*context switch* ou *task switch*).

Quando há a troca de contexto o *scheduler* se encarrega de salvar na memória RAM o estado atual do processo, e isso inclui o valor dos registradores. Quando a tarefa volta a ser executada o estado é restaurado do ponto onde ele parou, e isso inclui restaurar o valor de seus registradores.

É assim que cada *thread* tem valores distintos em seus registradores. É assim também que depuradores são capazes de ler e modificar o valor de registradores em uma determinada *thread* do processo, o sistema operacional dá a capacidade de acessar esses valores no contexto da tarefa e permite fazer a modificação. Quando o *scheduler* executar a tarefa o valor dos registradores serão atualizados com o valor armazenado no contexto.

 Processadores Intel mais modernos têm uma tecnologia chamada **Hyper-Threading**. Essa tecnologia permite que um mesmo núcleo atue como se fosse dois permitindo que duas *threads* sejam executadas paralelamente no mesmo núcleo.

Cada "parte" independente é chamada de processador lógico (*logical processor*) e cada processador lógico no núcleo tem seu conjunto individual de registradores. Com exceção de alguns registradores "obscuros" que são compartilhados pelos processadores lógicos do núcleo. Esses registradores não foram abordados no livro, mas caso esteja curioso pesquise por *Model-specific register* (MSR) e MTRRs. Apenas alguns MSR são compartilhados pelos processadores lógicos.

Sinais

Os sinais é um mecanismo de comunicação entre processos (*Inter-Process Communication* - IPC). Existem determinados sinais em cada sistema operacional e quando um sinal é enviado para um processo ele é temporariamente suspenso e um tratador (*handler*) do sinal é executado.

A maioria dos sinais podem ter o tratador personalizado pelo programador mas alguns têm um tratador padrão e não podem ser alterados. É o caso por exemplo no Linux do sinal SIGKILL, que é o sinal enviado para um processo quando você tenta forçar a finalização dele (com o comando `kill -9` por exemplo). O tratador desse sinal é exclusivamente controlado pelo sistema operacional e o processo não é capaz de personalizar ele.

Exemplo de personalização do tratador de um sinal:

```
main.c
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void termination(int signum)
{
    puts("Goodbye!");
    _Exit(EXIT_SUCCESS);
}

int main(void)
{
    struct sigaction action = {
        .sa_handler = termination,
    };

    sigaction(SIGTERM, &action, NULL);
    int pid = getpid();

    printf("My PID: %d\nPlease, send-me SIGTERM... ", pid);
    while (1)
    {
        getchar();
    }

    return 0;
}
```

Experimente compilar e executar esse programa. No Linux você pode enviar o sinal SIGTERM para o processo com o comando `kill`, como em:

```
$ kill 155541
```

```
# Onde 155541 seria o PID do processo.
```

O sinal SIGTERM seria o jeito "educado" de finalizar um processo. Porém como pode ser observado é possível que o processo personalize o tratador desse sinal, que por padrão finaliza o programa. Nesse código de exemplo se removermos a chamada para a função `_Exit()` o processo não irá mais finalizar ao receber SIGTERM. É por isso que existe o sinal mais "invasivo" SIGKILL que foi feito para ser usado quando o processo não está mais respondendo.

Um processo que está sendo depurado (o *tracee*) para toda vez que recebe um sinal e o depurador toma o controle da execução. Exceto no caso de SIGKILL que funciona normalmente sem a intervenção do depurador.

Exceções

Quando um processo dispara uma [exceção](#), um tratador (*handler*) configurado pelo sistema operacional envia um sinal para o processo tratar aquela exceção. Depuradores são capazes de identificar (e ignorar) exceções intervindo no processo de *handling* desse sinal.

Depuradores

Agora que já entendemos um pouco sobre processos vai ficar mais fácil entender como depuradores funcionam. Afinal de contas depuradores depuram processos. 😊


Depuradores têm a capacidade de controlar a execução das *threads* de um processo, tratar os sinais enviados para o processo, acessar sua memória e ver/editar dados relacionados ao contexto de cada *thread* (como os registradores, por exemplo). Todo esse poder é dado para os usuários do depurador por meio de alguns recursos que serão descritos abaixo.

Breakpoint

Um ponto de parada (*breakpoint*) é um ponto no código onde a execução do programa será interrompida e o depurador irá manter o programa em pausa para que o usuário possa controlar a execução em seguida.

Os *breakpoints* são implementados na prática (na arquitetura x86-64) como uma instrução `int3` que dispara [a exceção](#) #BP. Quando um depurador insere um *breakpoint* em um determinado ponto do código ele está simplesmente modificando o primeiro byte da instrução para o byte **0xCC**, que é o byte da instrução `int3`. Quando a exceção é disparada o sinal SIGTRAP é enviado para o processo e o depurador se encarrega de dar o controle da execução para o usuário. Quando o usuário continua a execução o

depurador restaura o byte original da instrução, executa ela e coloca o byte **0xCC** novamente.

 Em arquiteturas que não têm uma exceção específica para disparar *breakpoints* os depuradores substituem a instrução por alguma outra instrução que disparará alguma exceção. Como uma instrução de divisão ilegal por exemplo.

Podemos comprovar isso com o seguinte código:

```
#include <stdio.h>
#include <signal.h>

void breakpoint(int signum)
{
    puts("Breakpoint!");
}

int main(void)
{
    struct sigaction action = {
        .sa_handler = breakpoint,
    };

    sigaction(SIGTRAP, &action, NULL);

    asm("int3");
    puts("...");

    return 0;
}
```

Ao executar a instrução `int3` inserida com [inline Assembly](#) na linha 17, o processo recebe o sinal SIGTRAP e nosso tratador é executado. Experimente comentar a chamada para **sigaction** na linha 15 para ver o resultado do tratador padrão.

Hardware/Software breakpoint

O termo *software breakpoint* é usado para se referir a um *breakpoint* que é definido e configurado por software (o depurador), como o que já foi descrito acima. Por exemplo *breakpoints* podem ter uma condição de parada e isso é implementado pelo próprio

depurador. Ele faz o tratamento do *breakpoint* normalmente mas antes verifica a condição, se a condição não for atendida ele continua a execução do código como se o *breakpoint* nunca tivesse acontecido.

Já o termo *hardware breakpoint* é usado para se referir a um *breakpoint* que é suportado pelo próprio processador. A arquitetura x86-64 tem 8 registradores de depuração (*debug registers*) onde 4 deles podem ser usados para indicar *breakpoints*.

Os registradores DR0, DR1, DR2 e DR3 armazenam o endereço onde irá ocorrer o *breakpoint*. Já o registrador DR7 habilita ou desabilita esses *breakpoints* e configura uma condição para eles. Onde a condição determina em qual ocasião o *breakpoint* será disparado, como por exemplo ao ler/escrever naquele endereço ou ao executar a instrução no endereço.

Quando a condição do *breakpoint* é atendida o processador dispara [uma exceção](#) #BP.



Os *debug registers* não podem ser lidos/modificados sem privilégios de kernel. Rodando sobre um sistema operacional um processo comum não é capaz de manipulá-los diretamente.

Esse mesmo recurso (com até mais recursos ainda) poderia ser implementado pelo depurador com um *software breakpoint*. Por exemplo caso o depurador queira que um *breakpoint* seja disparado ao ler/escrever em um determinado endereço o depurador pode simplesmente modificar as permissões de acesso daquele endereço e, quando o processo fosse acessar os dados naquele endereço, uma exceção #GP seria disparada e o depurador poderia retomar o controle da execução.

Execução passo a passo

Depuradores não são apenas capazes de executar o software e esperar por um *breakpoint* para retomar o controle. Eles podem também executar apenas uma instrução da *thread* por vez e permanecer controlando a execução. Isso é chamado de execução passo a passo (*step by step*), onde o "passo" é uma única instrução. O usuário do depurador pode clicar em um botão ou executar um comando e apenas uma instrução do processo será executada, e o usuário pode ver o resultado da instrução e optar pelo que fazer em seguida.

Isso é implementado na arquitetura x86-64 usando a *trap flag* (TF) no registrador [EFLAGS](#). Quando a TF está ligada cada instrução executada dispara [uma exceção](#) #BP, permitindo assim que o depurador retome o controle após executar uma instrução.

Existe também o conceito de *step over* que é quando o depurador executa apenas "uma instrução" porém passando todas as instruções da rotina chamada pelo CALL. O que ele faz na prática é definir um *breakpoint* temporário para a instrução seguinte ao CALL, como na ilustração:

```
mov rdi, 5      ; Última instrução executada
--> call anything ; CALL que iremos "passar por cima"
test rax, rax   ; Instrução onde o breakpoint será definido
```

Se o depurador estiver parado no CALL e executamos um *step over*, o depurador coloca o *breakpoint* temporário na instrução TEST e então irá executar o processo. Quando o *breakpoint* na instrução TEST for alcançado ele será removido e o controle será dado para o usuário.

Repare no "defeito" desse mecanismo. O *step over* só funciona apropriadamente se a instrução seguinte ao CALL realmente for executada, senão o processo continuará a execução normalmente. Experimente rodar o seguinte código em um depurador:

```
testing.asm
```

```
bits 64
default rel

SYS_EXIT equ 60

section .text

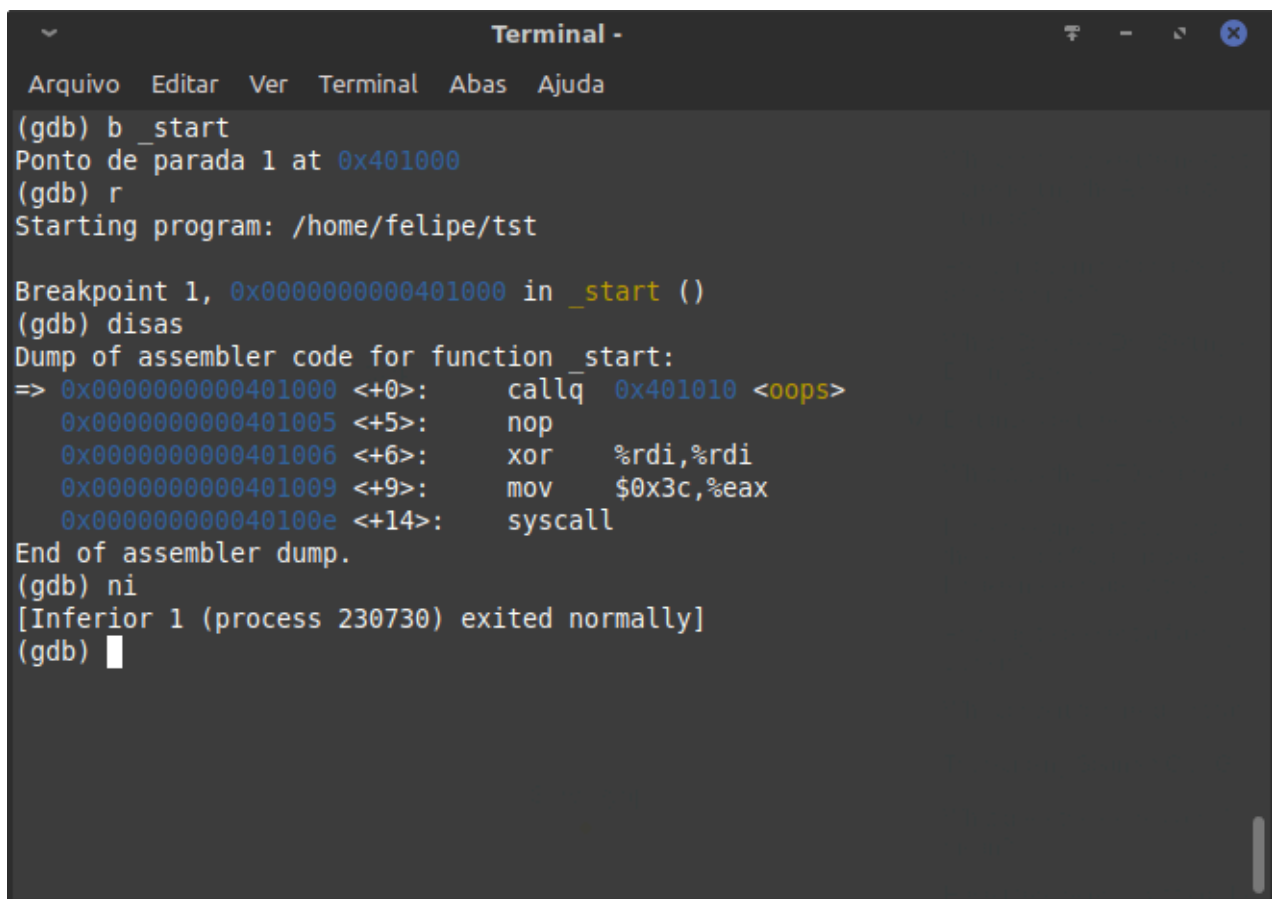
global _start
_start:
    call oops
    nop
    xor rdi, rdi
    mov rax, SYS_EXIT
    syscall

oops:
    add qword [rsp], 1
    ret
```

Compile com:

```
$ nasm testing.asm -o testing.o -felf64
$ ld testing.o -o testing
```

Ao dar um *step over* na chamada `call oops` um comportamento inesperado ocorre, o programa irá finalizar sem parar após o retorno da chamada. Isso é demonstrado na imagem abaixo com o depurador GDB:



```
Terminal -
Arquivo  Editar  Ver  Terminal  Abas  Ajuda
(gdb) b _start
Ponto de parada 1 at 0x401000
(gdb) r
Starting program: /home/felipe/tst

Breakpoint 1, 0x0000000000401000 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x0000000000401000 <+0>:      callq  0x401010 <oops>
    0x0000000000401005 <+5>:      nop
    0x0000000000401006 <+6>:      xor     %rdi,%rdi
    0x0000000000401009 <+9>:      mov     $0x3c,%eax
    0x000000000040100e <+14>:     syscall
End of assembler dump.
(gdb) ni
[Inferior 1 (process 230730) exited normally]
(gdb) █
```

Saída do depurador GDB

Informações de depuração do executável

Muitos depuradores voltados para desenvolvedores leem informações de depuração à respeito do executável produzidas pelo próprio compilador. O compilador pode, por exemplo, dar informações para que o depurador seja capaz de identificar de qual arquivo e linha do código-fonte uma instrução pertence.

É assim que funcionam os depuradores que exibem o código-fonte (ao invés de apenas as instruções em Assembly) enquanto executam o processo.

No caso do GCC ele armazena essas informações dentro do próprio executável na tabela de símbolos. Já o compilador da Microsoft, usado no Visual Studio, atualmente gera um arquivo `.pdb` contendo todas as informações de depuração.

Vale ressaltar aqui que o GCC (e qualquer outro compilador) **não** armazena o código-fonte do projeto dentro do executável. Ele meramente armazena o endereço do arquivo lá.

É comum também que depuradores apresentem algum erro ao não encontrar o arquivo-fonte indicado no endereço armazenado nas informações de depuração. Isso acontece quando ele tenta apresentar uma linha de código naquele arquivo mas o mesmo não foi encontrado na sua máquina.

Depurando com o GDB

Aprendendo a usar o depurador GDB do projeto GNU.

O GDB é um depurador de linha de comando que faz parte do projeto GNU. O [Mingw-w64](#) já instala o GDB junto com o GCC, e no Linux ele pode ser instalado pelo pacote `gdb`:


```
$ sudo apt install gdb
```

O GDB pode ser usado para depurar código tanto visualizando o Assembly como também o código-fonte. Para isso é necessário compilar o binário adicionando informações de depuração, com o GCC basta adicionar a opção `-g3` ao compilar. Exemplo:

```
$ gcc -g3 test.c -o test
```

E pode rodar o GDB passando o caminho do binário assim:

```
$ gdb ./test
```

 O caminho do binário é opcional. Caso especificado o GDB já inicia com esse binário como alvo para depuração, mas existem comandos do GDB que podem ser usados para escolher um alvo conforme será explicado mais abaixo.

O GDB funciona com comandos, quando você o inicia ele te apresenta um prompt onde você pode ir inserindo comandos para executar determinadas ações. Mais abaixo irei apresentar os principais comandos e como utilizá-los.

Esse depurador suporta depurar código de diversas linguagens de programação (incluindo C++, Go e Rust), mas aqui será demonstrado seu uso somente em um código escrito em C. O seguinte código será usado para demonstração:

```
test.c
```

```
#include <stdio.h>

#define DEFINED_VALUE 12345


int add(int a, int b)
{
    return a + b;
}

int main(int argc, char **argv)
{
    char str[] = "a =";
    int x = 8;

    printf("%s %d\n", str, add(x, 3));
    return 0;
}
```

E será compilado da seguinte forma:

```
$ gcc -g3 test.c -o test
```

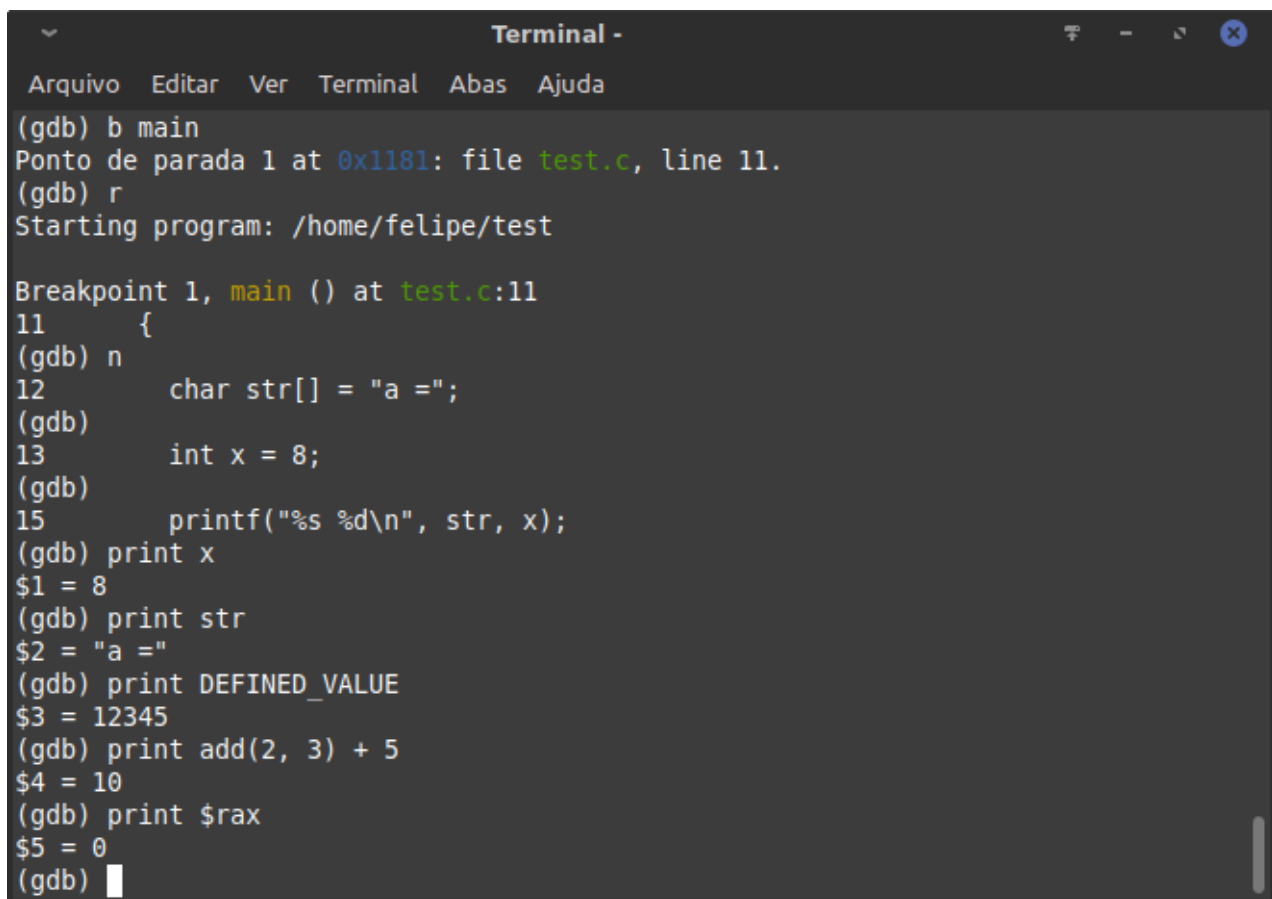
 A opção `-g` é usada para adicionar informações de depuração ao executável. Esse `3` seria o nível de informações que serão adicionadas, onde 3 é o maior nível.

Para mais informações consulte a [documentação do GCC](#).

Expressões

Determinadas instruções do GDB recebem uma expressão como argumento onde é possível usar qualquer tipo de constante, variável ou operador da linguagem que está sendo depurada (neste caso C). Isso inclui *casts*, *strings* literais, macros e até mesmo chamadas de funções. Logo a expressão interpretada é quase idêntica a uma expressão que você escreveria na linguagem que está sendo depurada (no nosso caso C).

Também é possível referenciar o valor de algum registrador na expressão usando o prefixo `$`, como `$rax` por exemplo. Na imagem abaixo é uma demonstração usando o comando `print`:



```
Terminal -
Arquivo  Editar  Ver  Terminal  Abas  Ajuda
(gdb) b main
Ponto de parada 1 at 0x1181: file test.c, line 11.
(gdb) r
Starting program: /home/felipe/test

Breakpoint 1, main () at test.c:11
11      {
(gdb) n
12      char str[] = "a =";
(gdb)
13      int x = 8;
(gdb)
15      printf("%s %d\n", str, x);
(gdb) print x
$1 = 8
(gdb) print str
$2 = "a ="
(gdb) print DEFINED_VALUE
$3 = 12345
(gdb) print add(2, 3) + 5
$4 = 10
(gdb) print $rax
$5 = 0
(gdb) 
```

Saída do GDB ao usar o comando `print`

Comandos

O GDB aceita abreviações dos comandos, onde ele identifica o comando a ser executado de acordo com suas primeiras letras ou abreviações definidas pelo depurador. Por exemplo o comando `breakpoint` pode ser executado também como `break`, `br` ou apenas `b`.

Ao apertar *enter* sem digitar nenhum comando o GDB irá reexecutar o último comando que você executou.

quit

```
quit [EXPR]
```

Finaliza o GDB. A expressão opcional é avaliada e o resultado dela é usado como código de saída. Se a expressão não for passada o GDB sai com código `0`.

file

```
file FILE
```

Usa o arquivo binário especificado como alvo para depuração. O programa é procurado no diretório atual ou em qualquer caminho registrado na variável de ambiente PATH.

attach e detach

```
attach <process-id>  
detach
```

O comando `attach` faz o [attach](#) no processo de ID especificado. Já o comando `detach` desfaz o *attach* no processo que está atualmente conectado.

Você também pode iniciar a execução do GDB com a opção `-p` para ele já inicializar fazendo *attach* em um processo, como em:

```
$ gdb -p 12345
```

breakpoint

```
break [PROBE_MODIFIER] [LOCATION] [thread THREADNUM] [if CONDITION]
```

Se o comando for executado sem qualquer argumento o *breakpoint* será adicionado na instrução atual.

LOCATION é a posição onde o *breakpoint* deve ser inserido e pode ser o número de uma linha, endereço ou posição explícita.

Ao especificar o número da linha, o nome do arquivo e o número da linha são separados por `:`. Se não especificar o nome do arquivo o *breakpoint* será adicionado a linha do arquivo atual. Exemplos:

```
(gdb) b 15
(gdb) b test.c:17
```

Onde o primeiro adicionaria o *breakpoint* na linha 15 do arquivo atual, e o segundo adicionaria na linha 17 do arquivo `test.c`.

O endereço pode ser simplesmente o nome de uma função ou então uma expressão, onde nesse caso é necessário usar `*` como prefixo ao símbolo ou endereço de memória. Como em:


```
(gdb) b main
(gdb) b *main + 8
(gdb) b *0x12345
```

No primeiro caso um *breakpoint* seria adicionado a função **main**. No segundo caso o endereço da primeira instrução da função **main** seria somado com 8, e o endereço resultante seria onde o *breakpoint* seria inserido. Já no terceiro caso o *breakpoint* seria inserido no endereço `0x12345`.

Também é possível especificar para qual *thread* o *breakpoint* deve ser inserido, onde por padrão o *breakpoint* é válido para todas as *threads*. Exemplo:

```
(gdb) b add thread 2
```

Isso adicionaria o *breakpoint* somente para a *thread* de ID 2.

 É possível usar o comando `info threads` para obter a lista de *threads* e seus números de identificação.

E por fim dá para adicionar uma condição de parada ao *breakpoint*. Onde **CONDITION** é [uma expressão](#) booleana. Exemplo:

```
(gdb) b 7 if a == 8
```

Onde no contexto do nosso código de exemplo, `a` seria o primeiro parâmetro da função `add`.

clear

```
clear [LOCATION]
```

Remove um *breakpoint* no local especificado. LOCATION funciona da mesma forma que no comando `breakpoint`.

Caso LOCATION não seja especificado remove o *breakpoint* na posição atual.

run

```
run [arg1, arg2, arg3...]
```

O comando `run` inicia (ou reinicia) a execução do programa alvo. Opcionalmente pode-se passar argumentos de linha de comando para o programa. Caso os argumentos não sejam especificados, os mesmos argumentos utilizados na última execução de `run` serão utilizados.

Nos argumentos é possível usar o caractere curinga `*`, ele será expandido pela shell do sistema. Também é possível usar os redirecionadores `<`, `>` ou `>>`.

kill

Finaliza a execução do programa que está sendo depurado.

start, starti

```
start [arg1, arg2, arg3...]  
starti [arg1, arg2, arg3...]
```

O uso desses dois comandos é idêntico ao uso de `run`. Porém o comando `start` inicia a execução do programa parando no começo da função **main**. Já o `starti` inicia parando na primeira instrução do programa.

next, nexti

```
next [N]  
nexti [N]
```

O comando `next` (ou apenas `n`) executa uma linha de código. Se N for especificado ele executa N linhas de código. Já o comando `nexti` (ou apenas `ni`) executa uma ou N instruções Assembly.

Os dois comandos atuam como um [step over](#), ou seja, não entram em chamadas de procedimentos.

step, stepi

```
step [N]  
stepi [N]
```

O `step` (ou `s`) executa uma ou N linhas de código. Já o `stepi` (ou `si`) executa uma ou N instruções Assembly. Os dois comandos entram em chamadas de procedimentos.

jump

```
jump LOCATION
```

Salta (modifica RIP) para o ponto do código especificado. Onde LOCATION é idêntico ao caso do comando *breakpoint* onde é possível especificar um número de linha ou endereço.

advance

```
advance LOCATION
```

Esse comando continua a execução do programa até o ponto do código especificado, daí para a execução lá. Assim como na instrução `jump`, o comando `advance` (ou `adv`) recebe um `LOCATION` como argumento.

O comando `advance` também para quando a função atual retorna.

finish

Executa até o retorno da função atual. Quando a função retorna é criada uma variável (como no caso do comando `print`) com o valor de retorno da função.

continue

Continua a execução normal do programa.

record e reverse-*

Imagine que mágico seria se o depurador pudesse voltar no tempo e desfazer as instruções executadas no programa, fazendo ele executar de maneira reversa parecido com rebobinar uma fita. Bom, o GDB pode fazer isso. 😎

Quando o programa já está em execução você pode executar o comando `record full` para iniciar a gravação das instruções executadas e `record stop` para parar de gravar.

Quando há a gravação é possível executar o programa em ordem reversa usando os comandos: `reverse-step` (`rs`), `reverse-stepi` (`rsi`), `reverse-next` (`rn`), `reverse-nexti` (`rni`) e `reverse-continue` (`rc`).

Esses comandos fazem a mesma coisa que os comandos normais, porém executando o programa ao reverso. Cada instrução revertida tem suas modificações na memória ou registradores desfeitas. Conforme demonstra a imagem abaixo.

GDB usando execução reversa.

Outros subcomandos de `record` são:

record goto

Salta para uma determinada instrução que foi gravada. Pode-se usar `record goto begin` para voltar ao início da gravação (desfazendo todas as instruções), `record goto end` para ir para o final da gravação ou `record goto N` onde N seria o número da instrução na gravação para saltar para ela.

record save <filename>

Salva os logs de execução no arquivo.

record restore <filename>

Restaura os logs de execução a partir do arquivo.

thread

```
thread <thread-id>
thread apply <thread-id> <command>
thread find <regex>
thread name <thread-name>
```

O comando `thread` pode ser usado para trocar entre *threads* do processo. Você pode usar o comando `info threads` para listar as *threads* do processo e obter seus ID.

Exemplo:

```
(gdb) thread 2
```

Isso trocaria para a *thread* de ID 2. Esse comando também tem os seguintes subcomandos:

thread apply

Executa um comando na *thread* especificada.

thread name

Define um nome para a *thread* atual, facilitando a identificação dela.

thread find

Recebe uma expressão regular como argumento que é usada para listar as *threads* cujo o nome coincida com a expressão regular. O comando exibe o ID das *threads* listadas.

print

```
print[/FMT] [EXPR]
```

O comando `print` (ou `p`) exibe no terminal o resultado da expressão passada como argumento. Opcionalmente pode-se especificar o formato de saída, onde os formatos são os mesmos utilizados no [comando x](#). Exemplo:

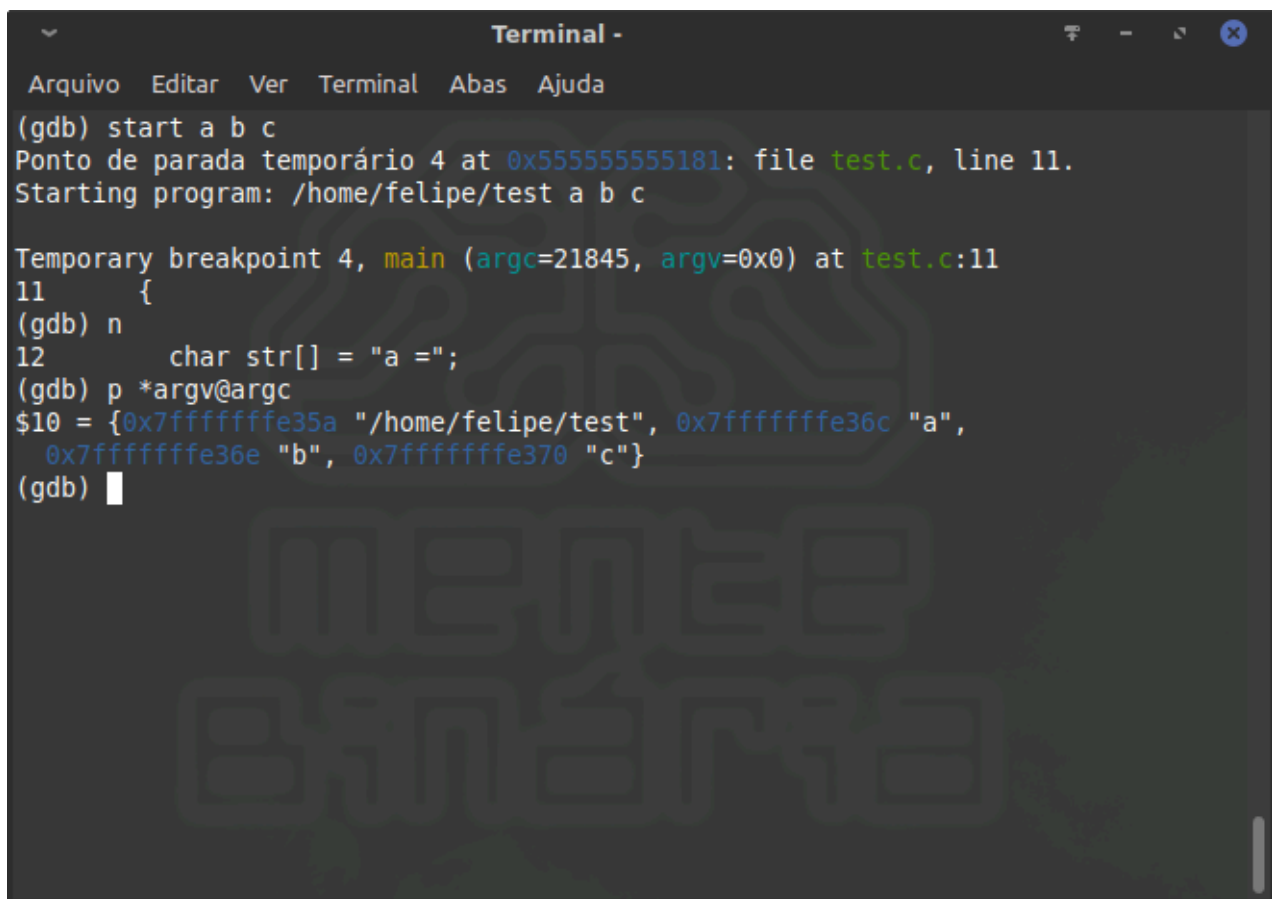
```
(gdb) p/x 15  
$1 = 0xf
```

Repare que a cada execução do comando `print` ele define uma variável (`$1`, `$2` etc.) que armazena o resultado da expressão do comando. Você também pode usar o valor dessas variáveis em uma expressão e assim reaproveitar o resultado de uma execução anterior do comando. Os símbolos `$` e `$$` se referem aos valores da última e penúltima execução do comando, respectivamente. Exemplo:

```
(gdb) p x + $3
```

Existe também o operador binário `@` que pode ser usado para tratar o valor no endereço especificado como uma *array*. O formato do uso desse operador é `array@size`, passando à esquerda o primeiro elemento da *array*.

Onde o tipo de cada elemento da *array* é definido de acordo com o tipo do objeto que está sendo referenciado. Na imagem abaixo é demonstrado o uso desse operador para visualizar todo o conteúdo da *array* **argv**.



```
Terminal -
Arquivo  Editar  Ver  Terminal  Abas  Ajuda
(gdb) start a b c
Ponto de parada temporário 4 at 0x55555555181: file test.c, line 11.
Starting program: /home/felipe/test a b c

Temporary breakpoint 4, main (argc=21845, argv=0x0) at test.c:11
11      {
(gdb) n
12      char str[] = "a =";
(gdb) p *argv@argc
$10 = {0x7fffffff35a "/home/felipe/test", 0x7fffffff36c "a",
      0x7fffffff36e "b", 0x7fffffff370 "c"}
(gdb) █
```

Saída do GDB ao usar Artificial Array

printf

```
printf "format string", ARG1, ARG2, ARG3, ..., ARG
```

Esse comando pode ser usado de maneira semelhante a função **printf** da libc. Cada argumento é separado por vírgula e o primeiro argumento é a *format string* que suporta quase todos os formatos suportados pela função **printf**. Os demais argumentos são [expressões](#).

Exemplo de uso:

```
(gdb) printf "%p\n", $rsp
0x7fffffffdf20
```

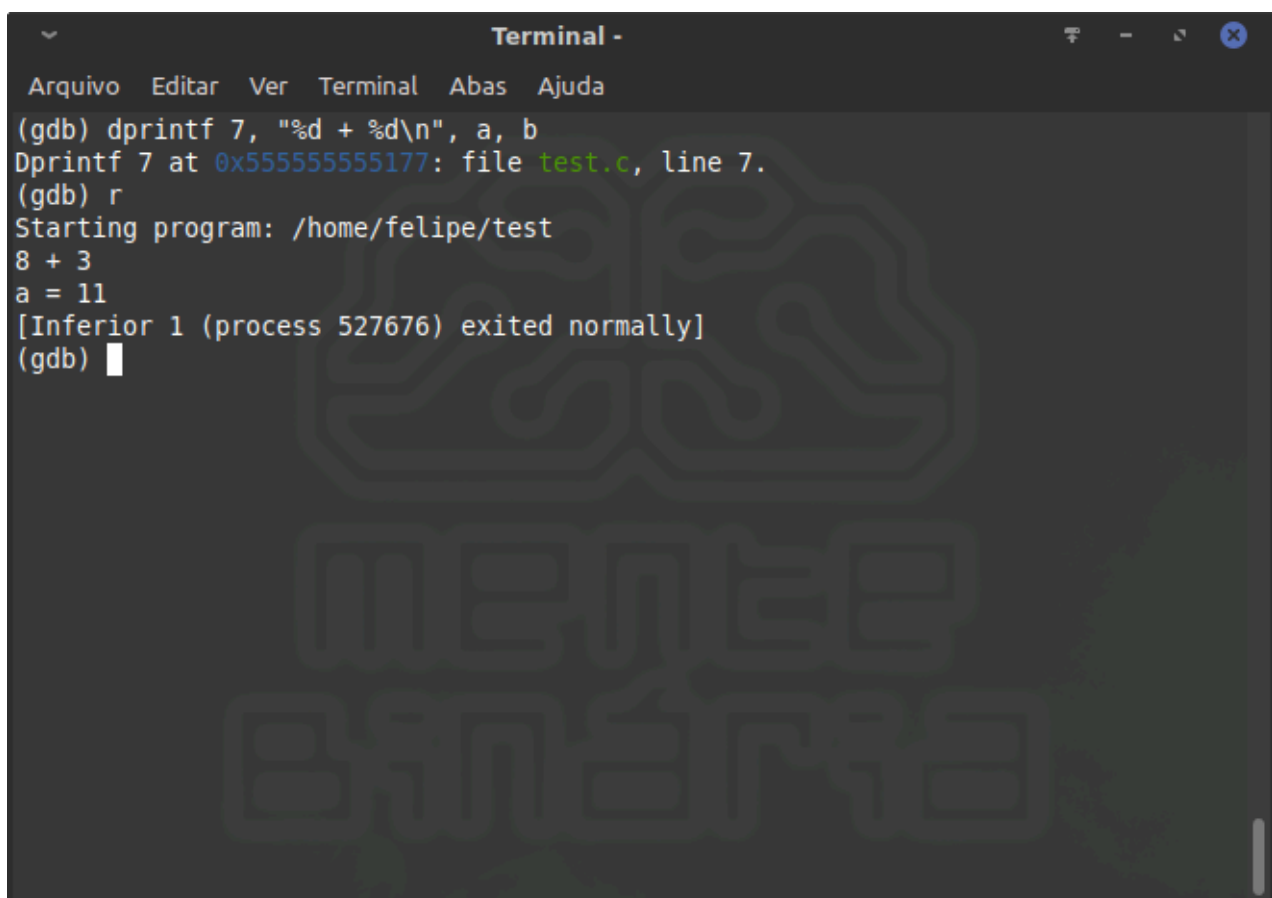
dprintf


```
dprintf LOCATION, "format string", ARG1, ARG2, ARG3, ..., ARG
```

Esse comando insere um *breakpoint* no código onde, toda vez que ele é alcançado, o comando `printf` é executado e depois a execução continua. O uso desse comando é semelhante ao do comando `printf`. Exemplo:

```
(gdb) dprintf 7, "%d + %d\n", a, b
```

No nosso código de exemplo, isso inseria o *dynamic printf* na linha 7 que está dentro da função **add**. Conforme a imagem abaixo demonstra:

A screenshot of a terminal window titled "Terminal -". The window has a menu bar with "Arquivo", "Editar", "Ver", "Terminal", "Abas", and "Ajuda". The terminal output shows the following sequence of commands and responses: (gdb) dprintf 7, "%d + %d\n", a, b; Dprintf 7 at 0x55555555177: file test.c, line 7.; (gdb) r; Starting program: /home/felipe/test; 8 + 3; a = 11; [Inferior 1 (process 527676) exited normally]; (gdb) [cursor].

```
Terminal -
Arquivo  Editar  Ver  Terminal  Abas  Ajuda
(gdb) dprintf 7, "%d + %d\n", a, b
Dprintf 7 at 0x55555555177: file test.c, line 7.
(gdb) r
Starting program: /home/felipe/test
8 + 3
a = 11
[Inferior 1 (process 527676) exited normally]
(gdb) █
```

Demonstração de uso do dprintf no GDB

X

```
x[/FMT] ADDRESS
```

O comando `x` serve para ver valores na memória. O argumento FMT (opcional) é o número de valores a serem exibidos, seguido de uma letra indicando o formato do valor seguido de uma letra que indica o tamanho do valor. Por padrão exibe apenas um valor caso o número não seja especificado. O formato e tamanho padrão é o mesmo utilizado na última execução do comando `x`.

As letras de formato são: `o` (octal), `x` (hexadecimal), `d` (decimal), `u` (decimal não-sinalizado), `t` (binário), `f` (*float*), `a` (endereço), `i` (instrução), `c` (caractere de 1 byte), `s` (*string*) e `z` (hexadecimal com zeros à esquerda).

Ao usar o formato `i` será feito o *disassembly* do código no endereço. O número de valores é usado para especificar o número de instruções para fazer o *disassembly*.

Exemplo:

```
(gdb) x/x 0x7fffffffdf64
0x7fffffffdf64: 0x003d2061
```

As letras de tamanho são: `b` (byte), `h` (metade de uma palavra), `w` (palavra) e `g` (*giant*, 8 bytes). Na arquitetura x86-64 uma palavra é 32-bit (4 bytes).

Exemplos:

```
(gdb) x/b 0x7fffffffdf64
0x7fffffffdf64: 0x61
(gdb) x/4b 0x7fffffffdf64
0x7fffffffdf64: 0x61    0x20    0x3d    0x00
```

disassembly

```
disassembly[/MODIFIER] [ADDRESS]
disassembly[/MODIFIER] start,end
disassembly[/MODIFIER] start,+length
```

O comando `disassembly` (ou `disas`) pode ser usado para exibir o *disassembly* de uma função ou *range* de endereço. O argumento ADDRESS (opcional) é uma expressão, sem esse argumento ele faz o *disassembly* na posição ou função atual.

Também é possível especificar um *range* de endereços para exibir o *dissassembly* das instruções, separando o endereço inicial e final por vírgula. Se usar o `+` no segundo argumento separado por vírgula, ele é considerado como o tamanho em bytes do *range* iniciado em **start**.

Exemplos:


```
(gdb) disas 0x0000555555551b4,0x0000555555551b9
Dump of assembler code from 0x555555551b4 to 0x555555551b9:
    0x0000555555551b4 <main+51>:    mov     $0x3,%esi
End of assembler dump.
(gdb) disas 0x0000555555551b4,+5
Dump of assembler code from 0x555555551b4 to 0x555555551b9:
    0x0000555555551b4 <main+51>:    mov     $0x3,%esi
End of assembler dump.
```

O argumento MODIFIER é uma (ou mais) das seguintes letras:

- `s` - Exibe também as linhas de código correspondentes as instruções em Assembly.
- `r` - Também exibe o código de máquina em hexadecimal.

Exemplo:

```
(gdb) disas/rs 0x0000555555551b4,+5
Dump of assembler code from 0x555555551b4 to 0x555555551b9:
test.c:
15    printf("%s %d\n", str, add(x, 3));
    0x0000555555551b4 <main+51>:    be 03 00 00 00    mov     $0x3,%esi
End of assembler dump.
```

 Por padrão o *dissassembly* é feito em sintaxe AT&T, mas você pode modificar para sintaxe Intel com o comando: `set disassembly-flavor intel`

list

```
list
list LINENUM
list FILE:LINENUM
list FUNCTION
list FILE:FUNCTION
list *ADDRESS
```

Exibe a listagem de código na linha ou início da função especificada. Um endereço também pode ser especificado usando um `*` como prefixo, as linhas de código correspondentes ao endereço serão exibidas.

Caso `list` seja executado sem argumentos mais linhas são exibidas a partir da última linha exibida pela última execução de `list`.

O número de linhas exibido é por padrão 10, mas esse valor pode ser alterado com o comando `set listsize <number-of-lines>`.

backtrace

```
backtrace [COUNT]
```

O comando `backtrace` (ou `bt`) exibe o *stack backtrace* atual. O argumento COUNT é o número máximo de *stack frames* que serão exibidos. Se for um número negativo exibe os primeiros *stack frames*.

Exemplo:

```
(gdb) bt
#0  add (a=8, b=3) at test.c:7
#1  0x000055555555551c0 in main (argc=1, argv=0x7fffffff068) at test.c:15
```

frame

```
frame [FRAME_NUMBER]
```

Sem argumentos exibe o *stack frame* selecionado. Caso seja especificado um número como argumento, seleciona e exibe o *stack frame* indicado pelo número. Esse número pode ser consultado com o comando `backtrace`.

Esse comando tem os seguintes subcomandos:

frame address

```
frame address STACK_ADDRESS
```

Exibe o *stack frame* no endereço especificado.

frame apply

```
frame apply COUNT COMMAND  
frame apply all COMMAND  
frame apply level FRAME_NUMBER COMMAND
```

O comando `frame apply` executa o mesmo comando em um ou mais *stack frames*. Esse subcomando é útil, por exemplo, para ver o valor das variáveis locais que estão em uma função de outro *stack frame* além do atual.

COUNT é o número de *frames* onde o comando será executado. Por exemplo `frame apply 2 p x` executaria o comando `print` nos últimos 2 *frames* (o atual e o anterior).

O `frame apply all` executa o comando em todos os *frames*. Já o `frame apply level` executa o comando em um *frame* específico. exemplo:

```
(gdb) frame apply level 1 info locals  
#1  0x000055555555551c0 in main (argc=1, argv=0x7fffffffe068) at test.c:15  
str = "a ="  
x = 8
```

frame function

```
frame function FUNCTION_NAME
```

Exibe o *stack frame* da função especificada.

frame level

```
frame level FRAME_NUMBER
```

Exibe o *stack frame* do número especificado.

info

O comando `info` contém diversos subcomandos para exibir informações sobre o programa que está sendo depurado. Abaixo será listado apenas os subcomandos principais.

info registers

Exibe os valores dos registradores. Pode-se passar como argumento uma lista (separada por espaço) dos registradores para exibir. Sem argumentos exibe o valor de todos os [registradores de propósito geral](#), [registradores de segmento](#) e [EFLAGS](#). Exemplo:

```
(gdb) info reg rax rbx
```

info frame

O uso desse subcomando é semelhante ao uso do comando [frame](#) e contém os mesmos subcomandos. A diferença é que ele exibe todas as informações relacionadas ao *stack frame*. Enquanto o comando `frame` apenas exibe informações de um ponto de vista de alto-nível.

info args

```
info args [NAMEREGEXP]
```

Exibe os argumentos passados para a função do *stack frame* atual. Se NAMEREGEXP for especificado exibe apenas os argumentos cujo o nome coincida com a expressão regular.

info locals

```
info locals [NAMEREGEXP]
```

Uso idêntico ao de `info args` só que exibe o valor das variáveis locais.

info functions

```
info functions [NAMEREGEXP]
```

Exibe todas as funções cujo o nome coincida com a expressão regular. Se o argumento não for especificado lista todas as funções.

info breakpoints

Exibe os *breakpoints* definidos no programa.

info source

Exibe informações sobre o código-fonte atual.

info threads

Lista as *threads* do processo.

display e undisplay

```
display[/FMT] EXPRESSION  
undisplay [NUM]
```

Esse comando pode ser usado da mesma maneira que o comando [print](#). Ele registra uma expressão para ser exibida a cada vez que a execução do processo faz uma parada.

Exemplo:

```
(gdb) display/7i $rip
```

Isso exibiria o *disassembly* de 7 instruções a partir de RIP a cada passo executado.


Se `display` for executado sem argumentos ele exibe todas as expressões registradas para auto-display.

Enquanto o comando `undisplay` remove a expressão com o número especificado. Sem argumentos remove todas as expressões registradas por `display`.

source

```
source FILE
```

Carrega o arquivo especificado e executa os comandos no arquivo como um script.

 Quando o GDB inicia ele faz o *source* automático do script de nome `.gdbinit` presente na sua pasta *home*. Exceto se o GDB for iniciado com a *flag* `--nh`.

help

O comando `help`, sem argumentos, lista as classes de comandos. É possível rodar `help CLASS` para obter a lista de comandos daquela classe.

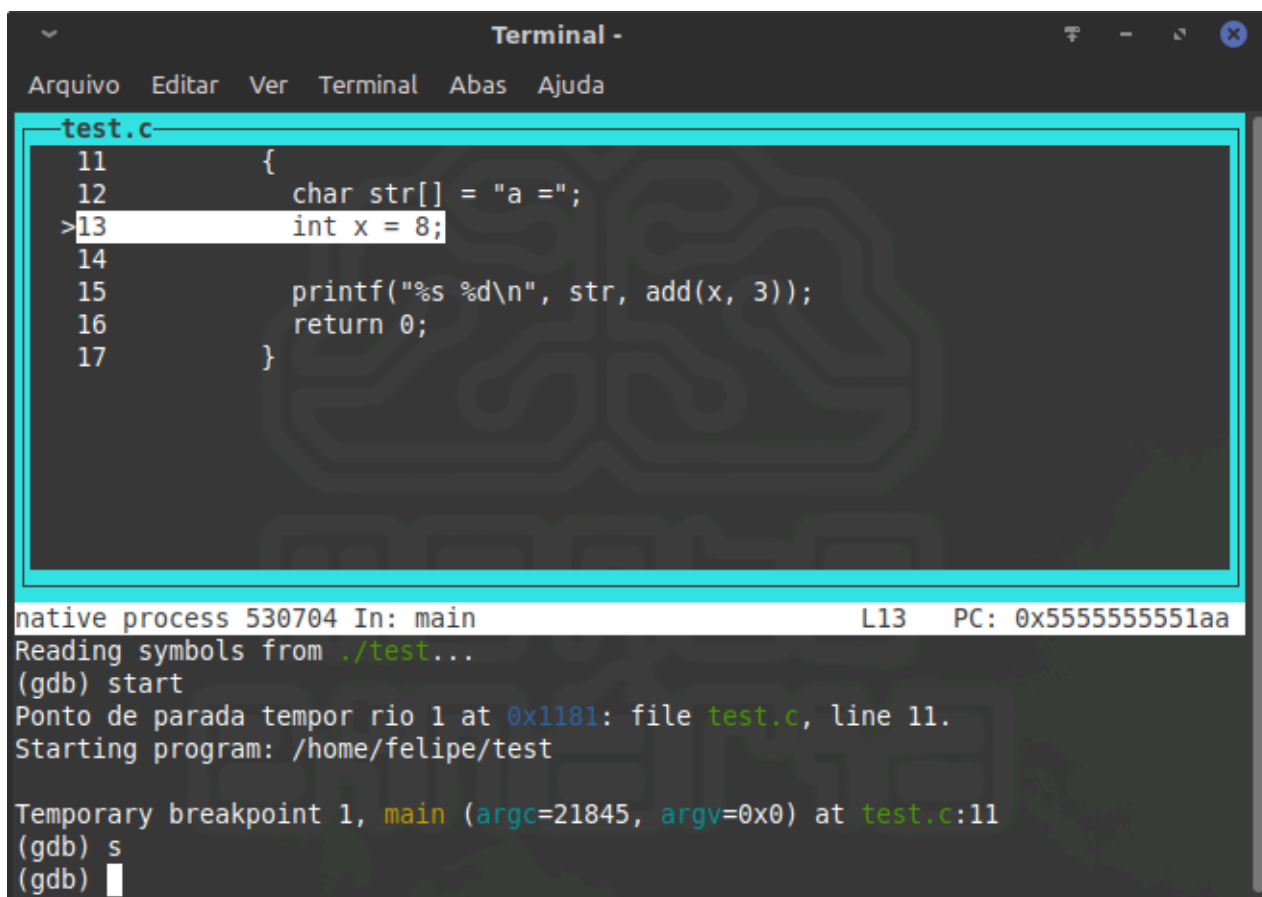
Também é possível rodar `help COMMAND` para obter ajuda para um comando específico, pode-se inclusive usar abreviações. E também é possível obter ajuda para subcomandos, conforme exemplos:


```
(gdb) help ni  
(gdb) help info reg  
(gdb) help frame apply level
```

Text User Interface (TUI)

É possível usar o GDB com uma interface textual permitindo que seja mais agradável acompanhar a execução enquanto observa o código-fonte. Para isso basta iniciar o GDB com a *flag* `-tui`, como em:

```
$ gdb -tui ./test
```



The screenshot shows a terminal window titled "Terminal -" with a menu bar containing "Arquivo", "Editar", "Ver", "Terminal", "Abas", and "Ajuda". The main window displays the source code of a file named "test.c". The code is as follows:

```
11      {  
12      char str[] = "a =";  
>13      int x = 8;  
14        
15      printf("%s %d\n", str, add(x, 3));  
16      return 0;  
17      }
```

Below the code editor, the GDB status bar shows: "native process 530704 In: main L13 PC: 0x555555551aa". The output of the GDB commands is shown in the terminal:

```
Reading symbols from ./test...  
(gdb) start  
Ponto de parada tempor rio 1 at 0x1181: file test.c, line 11.  
Starting program: /home/felipe/test  
  
Temporary breakpoint 1, main (argc=21845, argv=0x0) at test.c:11  
(gdb) s  
(gdb) █
```

GDB Text User Interface

Atalhos de teclado

Atalho de teclado	Descrição
Ctrl+x a	O atalho <code>Ctrl+x a</code> (Ctrl+x seguido da tecla <code>a</code>) alterna para o modo TUI caso tenha iniciado o GDB normalmente.
Ctrl+x 1	Alterna para o <i>layout</i> de janela única.
Ctrl+x 2	Alterna para o <i>layout</i> de janela dupla. Quando já está no <i>layout</i> de janela dupla o próximo <i>layout</i> com duas janelas é selecionado. Onde é possível exibir código-fonte+Assembly, registradores+Assembly e registradores+código-fonte.
Ctrl+x o	Muda a janela ativa.
Ctrl+x s	Muda para o modo Single Key Mode .
PgUp	Rola a janela ativa uma página para cima.
PgDn	Rola a janela ativa uma página para baixo.
↑ (Up)	Rola a janela ativa uma linha para cima.
↓ (Down)	Rola a janela ativa uma linha para baixo.
← (Left)	Rola a janela ativa uma coluna para a esquerda.
→ (Right)	Rola a janela ativa uma coluna para a direita.
Ctrl+L	Redesenha a tela.

Single Key Mode

Quando se está no modo **Single Key** é possível executar alguns comandos pressionando uma única tecla, conforme tabela abaixo:

Tecla	Comando	Nota
c	continue	
d	down	

f	finish	
n	next	
o	nexti	"o" de <i>step over</i> .
q	-	Sai do modo <i>Single Key</i> .
r	run	
s	step	
i	stepi	
u	up	
v	info locals	"v" de <i>variables</i> .
w	where	Alias para o comando backtrace.

Qualquer outra tecla alterna temporariamente para o modo de comandos. Após um comando ser executado ele retorna para o modo *Single Key*.

Depurando com o Dosbox

Aprendendo a usar o depurador do Dosbox

O emulador Dosbox tem um depurador embutido que facilita bastante na hora de programar alguma coisa para o MS-DOS