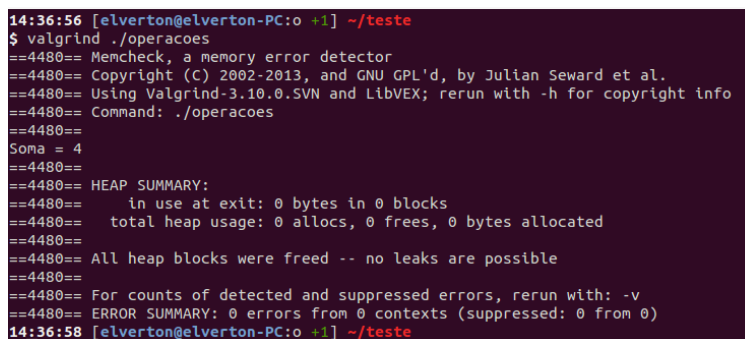


7 Utilizando o Valgrind.

Um modo interessante de verificar se seu programa contém alguns erros comuns envolvendo o uso de memória é utilizar uma ferramenta chamada Valgrind. Essa ferramenta é capaz de detectar problemas tais como leitura de variáveis que não foram devidamente inicializadas, falhas de segmentação ou vazamentos de memória — quando você esquece de chamar `free()`. Para depurar o código, é necessário incluir o parâmetro “-g” nas opções passadas para o `gcc`. Isso faz com que ele inclua informações no binário que é gerado que permitem que o valgrind mostre a linha de código em que ocorreu cada erro. Usar o valgrind é bastante fácil, bastando invocá-lo com o comando “valgrind” seguido pela forma como você executaria o trabalho prático normalmente no terminal do Linux. Por exemplo:

```
1 valgrind ./operacoes
```

Ao digitar esse comando em um terminal linux, contendo o binário “operacoes” compilado para a plataforma (Leia o Direcionamento 5), obtemos a saída mostrada pela figura 2.



```
14:36:56 [elvertont@elvertont-PC:~]$ valgrind ./operacoes
==4480== Memcheck, a memory error detector
==4480== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==4480== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==4480== Command: ./operacoes
==4480==
Soma = 4
==4480==
==4480== HEAP SUMMARY:
==4480==   in use at exit: 0 bytes in 0 blocks
==4480==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==4480==
==4480== All heap blocks were freed -- no leaks are possible
==4480==
==4480== For counts of detected and suppressed errors, rerun with: -v
==4480== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
14:36:58 [elvertont@elvertont-PC:~]$
```

Figura 2: Utilizando o Valgrind para avaliar o programa “operacoes”.

Primeiramente, o valgrind checa os possíveis erros de memória durante a execução do código. No caso do programa “operacoes”, nenhum erro foi encontrado e a saída do código aparece conforme mostrado. Ao final, ele mostra um sumário das alocações feitas pelo programa. Essas alocações se referem ao heap, que é uma área da memória criada quando um programa é invocado e é dedicada para alocações dinâmicas. Como não foi utilizada nenhuma alocação dinâmica, temos 0 “allocs” e 0 “frees”.

Entretanto, para ficar mais interessante o uso do valgrind, vamos criar um código chamado “faz_nada.c”, conforme mostrado abaixo:

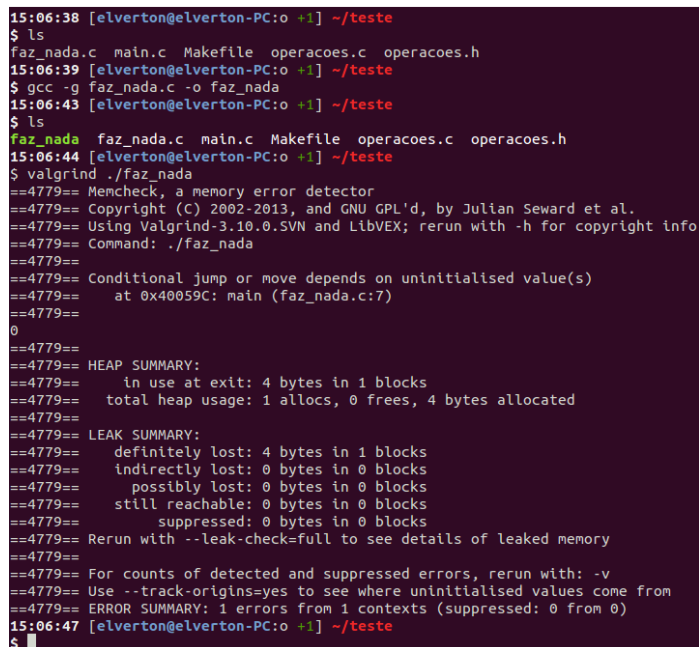
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *A;
5     A = (int*)malloc(sizeof(int));
6     if(*A == 1){
7         *A = 1;
8         printf("%d\n",*A);
9     } else{
10        *A = 0;
```

```

11     printf("%d\n",*A);
12 }
13 return 0;
14 }

```

Tente encontrar dois problemas neste código ... Ok! O primeiro é usar uma variável não inicializada na comparação. O segundo é não desalocar a memória alocada para a variável A, mais comumente chamado de vazamento de memória. A figura 3 mostra a compilação desse código e qual é a saída do Valgrind para ele.



```

15:06:38 [elvertont@elvertont-PC:o +1] ~/teste
$ ls
faz_nada.c main.c Makefile operacoes.c operacoes.h
15:06:39 [elvertont@elvertont-PC:o +1] ~/teste
$ gcc -g faz_nada.c -o faz_nada
15:06:43 [elvertont@elvertont-PC:o +1] ~/teste
$ ls
faz_nada faz_nada.c main.c Makefile operacoes.c operacoes.h
15:06:44 [elvertont@elvertont-PC:o +1] ~/teste
$ valgrind ./faz_nada
==4779== Memcheck, a memory error detector
==4779== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==4779== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==4779== Command: ./faz_nada
==4779==
==4779== Conditional jump or move depends on uninitialised value(s)
==4779==    at 0x40059C: main (faz_nada.c:7)
==4779==
0
==4779==
==4779== HEAP SUMMARY:
==4779==    in use at exit: 4 bytes in 1 blocks
==4779==    total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==4779==
==4779== LEAK SUMMARY:
==4779==    definitely lost: 4 bytes in 1 blocks
==4779==    indirectly lost: 0 bytes in 0 blocks
==4779==    possibly lost: 0 bytes in 0 blocks
==4779==    still reachable: 0 bytes in 0 blocks
==4779==    suppressed: 0 bytes in 0 blocks
==4779== Rerun with --leak-check=full to see details of leaked memory
==4779==
==4779== For counts of detected and suppressed errors, rerun with: -v
==4779== Use --track-origins=yes to see where uninitialised values come from
==4779== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
15:06:47 [elvertont@elvertont-PC:o +1] ~/teste
$

```

Figura 3: Utilizando o Valgrind para avaliar o programa “faz_nada”.

A saída do Valgrind mostra alguns avisos que ajudam a identificar os dois erros citados. O primeiro aviso é “Conditional jump ...”, que significa que um desvio condicional depende de uma variável não inicializada. Observe que o valgrind informa a linha do erro, que é justamente a linha da comparação. Após o término do programa, o Valgrind emite um sumário do uso do heap. Veja que foram alocados 4 bytes (ou 32 bits que é o tamanho do inteiro A alocado) mas eles não foram desalocados, segundo a linha do “total heap usage”. Para desalocar um endereço de memória alocado em “C”, você pode usar a função free(), conforme mostrado abaixo:

```

1 ...
2 A = (int*)malloc(sizeof(int));
3 ...
4 free(A);
5 ...

```

Lembre-se sempre de usar o Valgrind nos trabalhos práticos pois ele é utilizado pelos monitores para verificar o quão bom está sua alocação dinâmica de memória.