Guia Rápido: GCC, Makefile e Valgrind.

Alexandro Ramos

10 de setembro de 2015

Sumário

1 Introdução **3**

2 GCC **3**

2.1 Outros parâmetros úteis no GCC 4

3 Makefiles **6**

3.1 Makefile 1 9

3.1.1 Observações Importantes 10

3.2 Makefile 2 12

4 Valgrind **13**

4.1 Vazamento de Memória (*Memory leak*) 13

4.2 Leitura e escrita inválida 16

Referências Bibliográficas **18**

1 Introdução

A programação envolve a escrita, teste e manutenção de programas de computador. De acordo com Celes et al. (2004), o conhecimento de técnicas de programação adequadas para a elaboração de programas de computador tornou-se indispensável para profissionais que atuam nas áreas técnico-científicas. Por essa razão, o ensino de programação tornou-se um requisito básico para a formação desses profissionais. Um programa pode ser escrito em uma linguagem de programação, ou em um nível mais baixo, diretamente em linguagem de máquina.

Ao escrever um programa, é importante que o conjunto de dados e as operações que serão realizadas sobre eles estejam organizados, a fim de proporcionar o reuso de código e melhorar o desempenho. Nesse cenário surgem os TADs (Tipos Abstratos de Dados), que especificam os tipos de dados (domínio e operações) sem referência a detalhes da implementação (Dockhorn Costa, 2015).

A implementação dos programas de exemplo contidos neste documento foi realizada utilizando linguagem C e, na maioria dos casos, empregando o conceito de TAD. A construção de um programa que utiliza essa linguagem envolve duas fases independentes: compilação e execução. Sendo assim serão apresentadas ferramentas para compilar os programas, e durante a execução verificar erros e o uso de memória pelos programas.

ATENÇÃO: Para a execução dos exemplos e ferramentas demonstrados neste tutorial, assume-se que o sistema operacional utilizado seja do tipo Unix.

2 GCC

Os compiladores são programas responsáveis pelo processo de tradução das instruções escritas em uma determinada linguagem para a linguagem de máquina. Neste tutorial, será utilizado o GCC, incluído por padrão em diversas distribuições Linux.

Vamos considerar que um programa em C foi criado para imprimir na saída padrão a string "Hello World!", e armazenado em um arquivo chamado **hello.c**. Para compilar um programa em C, o processo é bem simples.

```c
#include <stdio.h>

int main(void) {
  printf("Hello World!\n");
  return 0;
}
```

No terminal, navegue até o diretório onde está o arquivo **hello.c** e digite o seguinte comando:

```
$ gcc -c hello.c
```

onde:

- **gcc**: é o programa para realizar a compilação.
- **-c**: argumento utilizado para gerar o arquivo-objeto.
- **hello.c**: o arquivo-fonte a ser compilado.

Ao terminar a compilação, um arquivo chamado **hello.o** é gerado. Agora é necessário realizar o processo de ligação para um executável, através do comando:

```
$ gcc -o prog hello.o
```

onde:

- **gcc**: é o programa para realizar a compilação.
- **-o**: liga o objeto gerado para um executável.
- **prog**: nome do arquivo executável especificado pelo programador.
- **hello.o**: o arquivo-objeto.

Também é possível compilar e fazer a ligação através de um único comando.

```
$ gcc hello.c -o prog
```

onde:

- **gcc**: é o compilador utilizado.
- **hello.c**: o arquivo-fonte a ser compilado.
- **-o**: liga o objeto gerado para um executável.
- **prog**: nome do arquivo executável especificado pelo programador.

Para executar o programa, basta digitar o seguinte comando no terminal:

```
$ ./prog
```

2.1 Outros parâmetros úteis no GCC

Além dos parâmetros `-c` e `-o`, o gcc oferece algumas opções importantes que permitem verificar erros de sintaxe e outros problemas no código, bem como opções para otimização do programa.

Considere o código do programa **warning.c** a seguir:

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

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

int main(void) {
  int num1 = 3;
  int num2 = 2;
  int s, p;

  s = soma(num1, num2);

  return 0;
}
```

Se esse programa for compilado com o padrão do gcc, nenhum alerta será exibido, conforme exibido na Figura 1.

```
Terminal
aramos@Labotim-07:~$ gcc warning.c -o prog
aramos@Labotim-07:~$
```

Figura 1: Compilando o programa **warning.c** com a configuração padrão do gcc.

A princípio nenhum problema foi identificado pelo compilador. Mas analisando melhor o programa, podemos observar que na linha **11** a variável inteira **p** foi declarada mas nunca foi utilizada. Ao incluir a opção **`-Wunused-variable`** o compilador exibirá uma mensagem alertando sobre esse problema, conforme exibido na Figura 2.

```
Terminal
aramos@Labotim-07:~$ gcc warning.c -Wunused-variable -o prog
warning.c: In function ‘main’:
warning.c:11:9: warning: unused variable ‘p’ [-Wunused-variable]
  int s, p;
         ^
aramos@Labotim-07:~$
```

Figura 2: Compilando o programa **warning.c** incluindo a opção **`-Wunused-variable`**.

O gcc inclui diversos parâmetros para verificar problemas no código durante a compilação, que podem ser consultados no manual do mesmo. Para exibir todos os alertas,

basta executar o comando `-Wall`. Observe na Figura 3 que outro aviso foi encontrado, informando que a variável **s** foi inicializada na linha 13, mas não está sendo utilizada no seu programa.

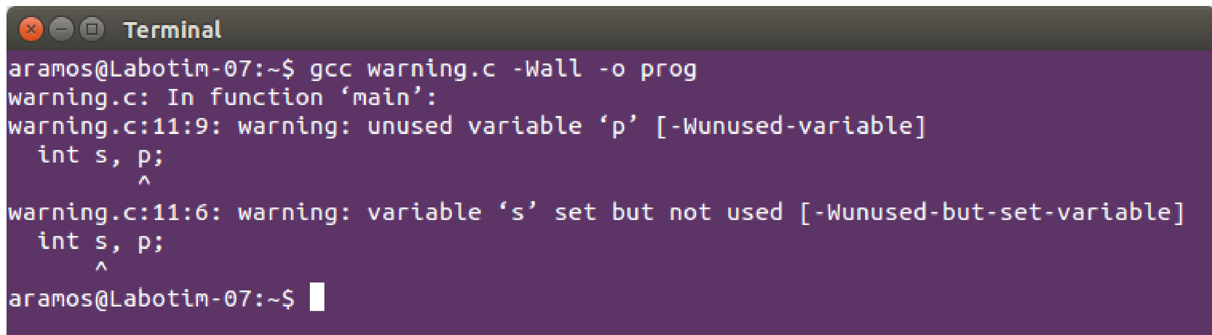


Figura 3: Compilando o programa **warning.c** incluindo a opção **`-Wall`**.

Outros parâmetros importantes são:

- **`-g`**: inclui no arquivo executável elementos utilizados pelos programas de depuração, como o gdb ou ddd.
- **`-o1, -o2, -o3`**: esses parâmetros realizam otimizações para melhorar o desempenho da execução do código. Variam de 1 a 3, sendo 3 o melhor processo de otimização.

3 Makefiles

Makefiles são arquivos que permitem organizar os comandos da compilação nos projetos de software, tais como a ligação e montagem de arquivos dos projetos. Este tutorial é um pequeno guia para compilar pequenos e médios projetos.

Em nosso exemplo, vamos considerar a implementação usando uma técnica de programação baseada na definição de tipos estruturados, conhecida como tipos abstratos de dados (TAD). A seguir é apresentado o TAD Ponto. O arquivo ***ponto.c*** contém a implementação do módulo e o arquivo ***ponto.h*** a interface do módulo. Ambos estão disponíveis em Celes et al. (2004).

Interface do **Módulo Ponto** (*ponto.h*)

```c
#ifndef PONTO_H_
#define PONTO_H_

/* TAD: Ponto (x,y) */

/* Tipo exportado */
typedef struct ponto Ponto;

/* Funções exportadas */

/* Função cria
 * Aloca e retorna um ponto com coordenadas (x,y).
 */
Ponto* pto_cria (float x, float y);

/* Função libera
 * Libera a memória de um ponto previamente criado.
 */
void pto_libera (Ponto* p);

/* Função acessa
 * Retorna os valores das coordenadas de um ponto.
 */
void pto_acessa (Ponto *p, float* x, float *y);

/* Função atribui
 * Atribui novos valores ás coordenadas de um ponto.
 */
void pto_atribui (Ponto* p, float x, float y);

/* Função distância
 * Retorna a distância entre dois pontos.
 */
float pto_distancia (Ponto *p1, Ponto* p2);

#endif /*PONTO_H_*/
```

Implementação do **Módulo Ponto** (*ponto.c*)

```
#include <stdlib.h>    /* malloc, free, exit */
#include <stdio.h>     /* printf */
#include <math.h>    /* sqrt */
#include "ponto.h"

struct ponto {
  float x;
  float y;
};

Ponto* pto_cria (float x, float y)
{
  Ponto* p = (Ponto*) malloc(sizeof(Ponto));
  if (p == NULL) {
    printf("Memória insuficiente!\n");
    exit(1);
  }
  p->x = x;
  p->y = y;
  return p;
}

void pto_libera (Ponto* p)
{
  free(p);
}

void pto_acessa (Ponto* p, float* x, float* y)
{
  *x = p->x;
  *y = p->y;
}

void pto_atribui (Ponto* p, float x, float y)
{
  p->x = x;
  p->y = y;
}

float pto_distancia (Ponto* p1, Ponto* p2)
{
  float dx = p2->x - p1->x;
  float dy = p2->y - p1->y;
  return sqrt(dx*dx + dy*dy);
}
```

Exemplo de um **programa cliente** (*progponto.c*)

```
#include <stdio.h>
#include "ponto.h"

int main (void)
{
  Ponto* p = pto_cria(2.0,1.0);
  Ponto* q = pto_cria(3.4,2.1);
  float d = pto_distancia(p,q);
  printf("Distância entre dois pontos: %f\n", d);
  pto_libera(q);
  pto_libera(p);
  return 0;
}
```

Em um Makefile, a escrita do código é feita em blocos que possuem três elementos importantes:

- **"alvo"**: arquivo a ser construído. Exemplos de alvos são arquivos-objeto ou executáveis.
- **"pré-requisito"**: arquivo necessário para gerar o "alvo". Um alvo pode ter nenhum, um ou mais pré-requisitos.
- **"comando"**: ação aplicada ao(s) pré-requisito(s) para gerar o "alvo".

Nota: Vários blocos podem ser encadeados para criar relações mais complexas de dependência, bastando que uma das dependências seja o alvo de um outro bloco.

```
### Makefile ###

alvo: pré-requisito(s)
<TAB>   comando
```

O make atua de forma recursiva nas relações de dependência, e verifica se o alvo precisa ser recriado. Se as dependências não foram modificadas, o alvo não é recriado.

A seguir são apresentadas diferentes versões do makefile para compilação do exemplo.

3.1 Makefile 1

Depois que os arquivos-fonte do módulo e do programa cliente foram implementados, podemos gerar um executável compilando cada um dos arquivos separadamente e depois realizar o processo de ligação, conforme visto na seção 2.

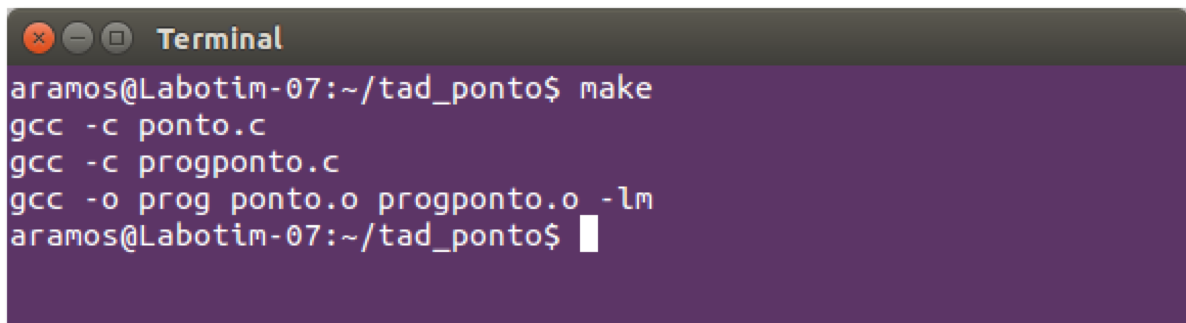
```
$ gcc -c ponto.c
$ gcc -c progponto.c
$ gcc -o prog ponto.o progponto.o -lm
```

> **Nota:** A opção **`-lm`** foi adicionada para realizar a ligação com a biblioteca **math**, utilizada na implementação do arquivo **ponto.c**.

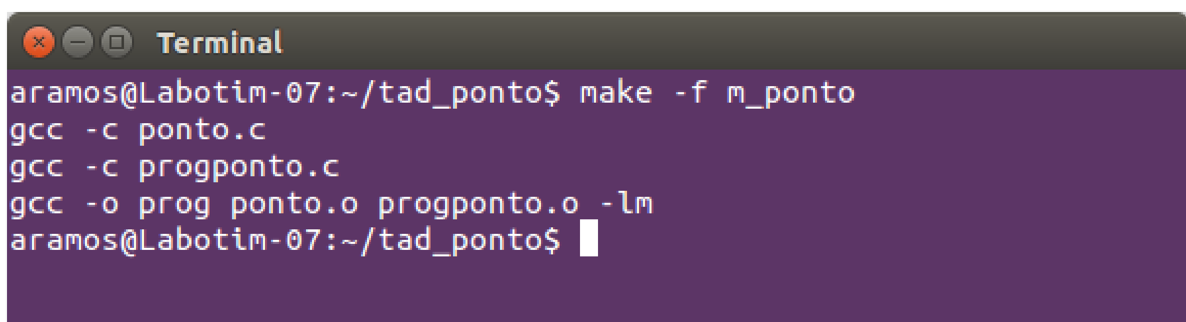
Agora vamos organizar os comandos acima de acordo com a estrutura do Makefile. O resultado é o seguinte:

```
1  ### Makefile ###
2
3  all: prog
4
5  prog: ponto.o progponto.o
6      gcc -o prog ponto.o progponto.o -lm
7
8  progponto.o: progponto.c
9      gcc -c progponto.c
10
11 ponto.o: ponto.c
12     gcc -c ponto.c
```

Depois disso, o arquivo normalmente é salvo no mesmo diretório onde está o código-fonte do programa. Se você nomear o arquivo com "**Makefile**" ou "**makefile**", basta apenas executar o comando **`make`**.



Caso você nomeie um makefile de forma diferente, é necessário usar a opção **`-f`** e em seguida o nome do arquivo.



3.1.1 Observações Importantes

Existem algumas convenções para nomear os alvos, uma vez que não existe um padrão formal. Geralmente, os seguintes nomes são adotados:

- **all**: nome do primeiro alvo.
- **clean**: usado para apagar arquivos-objeto (*.o) e outros arquivos temporários.
- **rmproper**: usado para apagar tudo que precisa ser modificado.

No exemplo anterior utilizamos apenas o alvo **all**. Vamos incluir os alvos **clean** e **rmproper**. Você pode criar outros alvos, por exemplo **run** para executar o programa.

```
1  ### Makefile ###
2
3  all: prog
4
5  prog: ponto.o progponto.o
6      gcc -o prog ponto.o progponto.o -lm
7
8  progponto.o: progponto.c
9      gcc -c progponto.c
10
11 ponto.o: ponto.c
12     gcc -c ponto.c
13
14 clean:
15     rm -rf *.o
16
17 rmproper: clean
18     rm -rf prog
19
20 run: prog
21     ./prog
```

Ao executar o comando make, o primeiro alvo do arquivo é executado, neste caso **all**, conforme demonstrado na Figura 4.

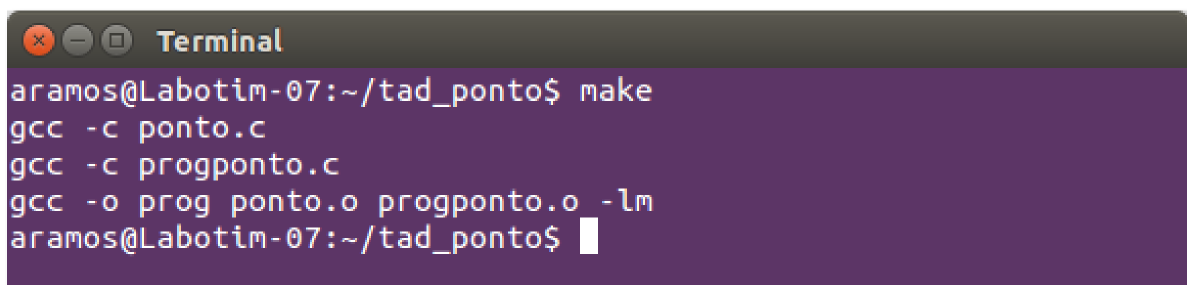


Figura 4: Executando o utilitário `make` sem especificar o alvo.

Porém, você pode executar explicitamente o alvo desejado, por exemplo, o **clean** para limpar os arquivos-objeto ou temporários ou o **run** para compilar, ligar e executar (Figura 5).

```
aramos@Labotim-07:~/tad_ponto$ make run
gcc -c ponto.c
gcc -c progponto.c
gcc -o prog ponto.o progponto.o -lm
./prog
Distância entre dois pontos: 1.780449
aramos@Labotim-07:~/tad_ponto$
```

Figura 5: Executando o utilitário **make** especificando o alvo **run**.

3.2 Makefile 2

Ao manipular uma grande quantidade de arquivos, a elaboração e manutenção do Makefile pode ser bem complicada. O arquivo abaixo generaliza o modelo, podendo ser utilizado para compilar e ligar diferentes projetos modularizados (usando TADs), como o do exemplo anterior.

```
1  ### Makefile ###
2
3  # Nome do compilador
4  CC = gcc
5
6  # Opções de compilação
7  CFLAGS = -Wall -g
8
9  # Ligação com as bibliotecas
10 LDFLAGS = -lm
11
12 # wildcard é utilizado para listar os arquivos-fonte no diretório atual
13 FONTES = $(wildcard *.c)
14
15 # gera a listar dos arquivos-objeto usando a lista de arquivos-fonte
16 OBJETOS = $(FONTES:.c=.o)
17
18 # nome do arquivo executável
19 EXECUTAVEL = prog
20
21 # all: prog
22 all: $(EXECUTAVEL)
23
24 # prog: ponto.o progponto.o
25 #    gcc -o prog ponto.o progponto.o -lm
26 $(EXECUTAVEL): $(OBJETOS)
27     $(CC) -o $@ $^ $(LDFLAGS)
28
29 # progponto.o: progponto.c
30 #    gcc -c progponto.c
31 #
32 # ponto.o: ponto.c
33 #    gcc -c ponto.c
34 %.o: %.c
35     $(CC) -c $(CFLAGS) $^
36
37 clean:
38     rm -rf *.o
39
40 # rmproper: clean
41 #    rm -rf prog
42 rmproper: clean
43     rm -rf $(EXECUTAVEL)
44
45 # run: prog
46 #    ./prog
47 run: $(EXECUTAVEL)
48     ./$(EXECUTAVEL)
```

4 Valgrind

Alguns dos erros mais difíceis de resolver em programas escritos usando a linguagem C são oriundos da má gestão da memória. Erros como: alocar blocos de memória e esquecer de desalocar após o uso, tentar escrever ou ler em um bloco de memória que não foi reservado, não inicializar variáveis de controle ou ponteiros.

Existem vários programas para auxiliar na detecção desses erros. Um deles é o Valgrind, que fornece uma série de ferramentas de depuração e análise do seu programa. A mais popular dessas ferramentas é chamada **Memcheck**, que pode detectar erros relacionados à memória.

4.1 Vazamento de Memória (*Memory leak*)

Memory leak, ou vazamento de memória, ocorre quando blocos de memória alocados por um programa para realizar uma determinada operação não são liberados após o uso. O programa a seguir aloca e libera memória para uma matriz.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int** alocaMatriz(int linhas, int colunas) {
5    int i;
6    int** mat = (int**)malloc(linhas*sizeof(int*));
7    for (i = 0; i < linhas; i++)
8      mat[i] = (int*)malloc(colunas*sizeof(int));
9
10   return mat;
11 }
12
13 void liberaMatriz(int** mat) {
14   free(mat);
15 }
16
17 int main (void) {
18   int** mat = alocaMatriz(2,3);
19   liberaMatriz(mat);
20
21   return 0;
22 }
```

Ao compilar e executar este programa, nenhuma mensagem de erro será exibida. Porém, se analisarmos a função que libera a memória, podemos detectar um problema: somente a alocação feita para o vetor de ponteiros está sendo liberada, mas os vetores-linha alocados não são liberados. Ao executar o programa usando Valgrind, esse problema será detectado.

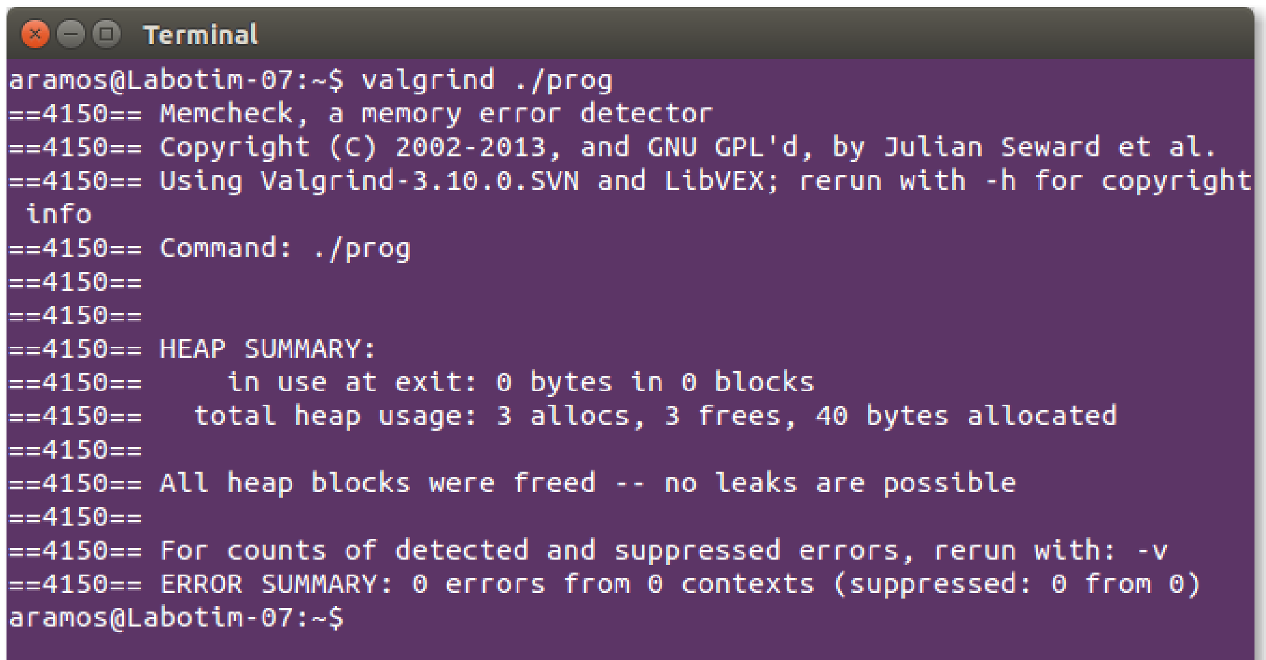
```
aramos@Labotim-07:~$ valgrind ./prog
==3934== Memcheck, a memory error detector
==3934== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==3934== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright
 info
==3934== Command: ./prog
==3934==
==3934==
==3934== HEAP SUMMARY:
==3934==     in use at exit: 24 bytes in 2 blocks
==3934==   total heap usage: 3 allocs, 1 frees, 40 bytes allocated
==3934==
==3934== LEAK SUMMARY:
==3934==    definitely lost: 24 bytes in 2 blocks
==3934==    indirectly lost: 0 bytes in 0 blocks
==3934==      possibly lost: 0 bytes in 0 blocks
==3934==    still reachable: 0 bytes in 0 blocks
==3934==         suppressed: 0 bytes in 0 blocks
==3934== Rerun with --leak-check=full to see details of leaked memory
==3934==
==3934== For counts of detected and suppressed errors, rerun with: -v
==3934== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
aramos@Labotim-07:~$
```

No relatório do Valgrind, é informado que 3 blocos de memória foram alocados e somente um foi liberado. Para obter mais detalhes sobre o vazamento de memória, basta executar o Valgrind com a opção `-leak-check=full`.

O código a seguir apresenta a função liberaMatriz com a liberação dos vetores-linha e depois do vetor de ponteiros.

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

int** alocaMatriz(int linhas, int colunas) {
  int i;
  int** mat = (int**)malloc(linhas*sizeof(int*));
  for (i = 0; i < linhas; i++)
    mat[i] = (int*)malloc(colunas*sizeof(int));

  return mat;
}

void liberaMatriz(int** mat, int linhas) {
  int i;
  for (i = 0; i < linhas; i++)
    free(mat[i]);
  free(mat);
}

int main (void) {
  int** mat = alocaMatriz(2,3);
  liberaMatriz(mat, 2);

  return 0;
}
```

Agora o relatório do Valgrind informa que todos os blocos de memória alocados foram devidamente liberados.



ATENÇÃO: Ao manipular arquivos em C, é importante ficar atento à abertura e fechamento dos mesmos, pois na chamada da função **fopen()** é reservada memória para realizar as operações com o arquivo e a função **fclose()** libera a memória reservada.

```
        ...

        FILE* arquivo;
        arquivo = fopen("entrada.txt", "rt");

        ...

        fclose(arquivo);
```

4.2 Leitura e escrita inválida

Um outro problema muito comum é tentar escrever ou ler em blocos de memória que não foram reservados pelo programa. Considere o programa a seguir.

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

#define MAX 5

int* f (int n) {
  int i;
  int* vet = (int*)malloc(n*sizeof(int));
  for (i = 0; i <= n; i++) {
    vet[i] = i;
  }
  return vet;
}

int main (void) {
  int *v;
  v = f(MAX);

  printf("O útimo elemento do vetor é: %d\n", v[MAX]);

  free(v);
}
```

Observamos na figura abaixo que o programa compilou e executou sem apresentar nenhum tipo de erro.

```
Terminal
aramos@Labotim-07:~$ gcc invalid.c -o prog
aramos@Labotim-07:~$ ./prog
O útimo elemento do vetor é: 5
aramos@Labotim-07:~$
```

Agora vamos executar o programa utilizando **Valgrind**.

```
Terminal
aramos@Labotim-07:~$ valgrind ./prog
==5260== Memcheck, a memory error detector
==5260== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==5260== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==5260== Command: ./prog
==5260==
==5260== Invalid write of size 4
==5260==    at 0x4005FD: f (in /home/aramos/prog)
==5260==    by 0x400622: main (in /home/aramos/prog)
==5260==  Address 0x51fc054 is 0 bytes after a block of size 20 alloc'd
==5260==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==5260==    by 0x4005D8: f (in /home/aramos/prog)
==5260==    by 0x400622: main (in /home/aramos/prog)
==5260==
==5260== Invalid read of size 4
==5260==    at 0x40062F: main (in /home/aramos/prog)
==5260==  Address 0x51fc054 is 0 bytes after a block of size 20 alloc'd
==5260==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==5260==    by 0x4005D8: f (in /home/aramos/prog)
==5260==    by 0x400622: main (in /home/aramos/prog)
==5260==
O útimo elemento do vetor é: 5
==5260==
==5260== HEAP SUMMARY:
==5260==     in use at exit: 0 bytes in 0 blocks
==5260==   total heap usage: 1 allocs, 1 frees, 20 bytes allocated
==5260==
==5260== All heap blocks were freed -- no leaks are possible
==5260==
==5260== For counts of detected and suppressed errors, rerun with: -v
==5260== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
aramos@Labotim-07:~$
```

Podemos observar que o relatório do Valgrind aponta dois erros. Vamos entender melhor cada um deles.

O primeiro é um erro de escrita no bloco de memória, que ocorre na função **f**, chamada pela função **main**.

```
==5260== Invalid write of size 4
==5260==    at 0x4005FD: f (in /home/aramos/prog)
==5260==    by 0x400622: main (in /home/aramos/prog)
==5260==  Address 0x51fc054 is 0 bytes after a block of size 20 alloc'd
==5260==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==5260==    by 0x4005D8: f (in /home/aramos/prog)
==5260==    by 0x400622: main (in /home/aramos/prog)
```

Ao analisar o código, podemos observar que, na **linha 8**, foi alocado um bloco de memória para **n** inteiros (n = 5). O laço de repetição usado para preencher o vetor, na **linha 9**, caminha da posição 0 até n, tentando gravar em uma área de memória que não foi reservada previamente.

```
6       int* f (int n) {
7         int i;
8         int* vet = (int*)malloc(n*sizeof(int));
9         for (i = 0; i <= n; i++) {
10          vet[i] = i;
11        }
12        return vet;
13      }
```

O segundo é um erro de leitura do bloco de memória, que ocorre na função **main**, ao chamar a função **printf()**.

```
==5260== Invalid read of size 4
==5260==    at 0x40062F: main (in /home/aramos/prog)
==5260==  Address 0x51fc054 is 0 bytes after a block of size 20 alloc'd
==5260==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==5260==    by 0x4005D8: f (in /home/aramos/prog)
==5260==    by 0x400622: main (in /home/aramos/prog)
==5260==
O útimo elemento do vetor é: 5
```

Analisando o código, podemos observar que, na **linha 19**, a função **printf()** recebe um argumento (v[MAX] = v[5]) que lê um bloco de memória não reservado pelo programa, pois a última posição válida do vetor é a 4.

```
19      printf("O útimo elemento do vetor é: %d\n", v[MAX]);
```

Referências Bibliográficas

Celes, W., Cequeira, R., e Rangel, J. L. (2004). *Introdução a Estrutura de Dados*. Campus.

Dockhorn Costa, P. (2015). *Estrutura de Dados I - Notas de Aula*.