Problemas com Ponteiros

Camilla Hollmann, André Grégio

Neste capítulo, serão discutidos alguns problemas com ponteiros, focando no gerenciamento e acesso incorretos à memória (em geral inválida ou não-inicializada).

# Ponteiro não-inicializado (*wild pointer*)

Um ponteiro não-inicializado aponta para uma região de memória qualquer, que pode estar inclusive fora do escopo de memória do seu programa.

| // Declaração de ponteiro para inteiro.  **int \*ptri;** |
| --- |

No exemplo, **ptri** é um ponteiro para inteiro não-inicializado, ou seja, apesar de poder armazenar um endereço, não sabemos o que ele armazena (pode ser NULL ou algum valor considerado lixo). A seguir, um exemplo de uso incorreto do ponteiro declarado acima:

| // Atribuição INCORRETA.  **\*ptri = 99;** |
| --- |

Por não ter sido inicializado, o ponteiro **ptri** ainda não armazena um endereço de memória válido. Dessa forma, o programa irá tentar armazenar o valor 99 em um endereço de memória qualquer para a qual **ptri** apontava, causando um comportamento indefinido ou um *crash*…

É importante lembrar que, se você inicializar o ponteiro com NULL, ele não pode ser desreferenciado, pois não há um endereço de memória válido no ponteiro. Veja exemplos:

| // Declaração de ponteiro para inteiro.  **int \*ptri;**  **/\*** Atribuição de valor nulo para o ponteiro e  inicialização **INCORRETA** do ponteiro. **\*/**  ptri = NULL;  \*ptri = 99; |
| --- |

| // Declaração de ponteiro para inteiro.  **int \*ptri;**  // Inicialização **CORRETA** do ponteiro.  **ptri = malloc(sizeof(int));** |
| --- |

# Ponteiro pendente (*dangling pointer*)

Um ponteiro pendente ou *dangling pointer* é um ponteiro que aponta para uma área da memória que foi excluída ou liberada.

| // Declaração e inicialização de ponteiro para inteiro.  **int \*ptri = malloc(sizeof(int));**  // Liberação do bloco alocado para ptri, tornando-o pendurado.  **free(ptri);** |
| --- |

No exemplo abaixo, o ponteiro pendente será apontado para NULL, assim deixando de ser *dangling pointer*. Pode-se chamar essa medida de “aterramento” do ponteiro.

| // Declaração e inicialização de ponteiro para inteiro.  **int \*ptri = malloc(sizeof(int));**  // Liberação do bloco alocado para ptri, tornando-o pendurado.  **free(ptri);**  // Ponteiro não mais aponta para bloco de memória inválido.  **ptri = NULL;** |
| --- |

Um dos problemas que podem ser causados por *dangling* pointers é o chamado *user-after-free*, no qual o valor contido em um endereço de memória pode ser lido após sua liberação.

| // Declaração e inicialização do ponteiro para inteiro com 32 Bytes.  **int \*ptrc = malloc(sizeof(int)\*8);**  // Atribuição de valor para o vetor e impressão.  **for(int i=0; i < N; i++) {**  **ptri[i] = i;**  **printf("%d ", ptri[i]);**  **}**  // Liberação do bloco alocado para o vetor, tornando-o pendurado.  **free(ptri);**  **printf(“\nDepois do free:\n“);**  // Uso da variável após liberação, gerando comportamento indefinido.  **for(int i=0; i < N; i++)**  **printf("%d ", \*(ptri+i));** |
| --- |

**SAÍDA:**

**0 1 2 3 4 5 6 7**

**Depois do free:**

**1582040296 5 -242498083 -842059238 4 5 6 7**

Ler a memória que foi liberada é comportamento indefinido e não há como saber o que será acessado. Além disso, tal prática também pode corromper a *heap*, permitindo a execução arbitrária de código.

# Vazamento de memória (*memory leak*)

O *memory leak* ocorre quando se perde a referência para uma memória previamente alocada (perda de ponteiro) e não se libera essa memória depois do uso.

| // Função que remove elemento de uma pilha dinâmica  **void pop(pilha \*P) {**  **if(P->topo)**  **P->topo = P->topo->proximo;**  /\* Ao ajustar o topo sem guardar o endereço do elemento a ser  \* removido, o elemento que ocupava o topo anteriormente continua  \* ocupando memória e não pode mais ser liberado… \*/ |
| --- |

No exemplo a seguir, supõe-se o uso de ponteiros e alocação dinâmica dentro do escopo de uma função qualquer:

| // Declaração e alocação do ponteiro para inteiro.  **int \*ptri = malloc(100 \* sizeof(int))**  // Retorno da função sem dar free, gerando um vazamento de memória.  **if (ptri != condicao)**  **return 0;** |
| --- |

Para evitar que isso ocorra, é importante liberar a memória alocada após o uso da variável, utilizando a função *free ( )*:

| // Declaração e alocação do ponteiro para inteiro.  **int \*ptri = malloc(sizeof(int))**  // Libera a memória alocada após seu uso, evitando o vazamento.  **if (ptri != condicao)**  **free(ptri);**  **return 0;** |
| --- |

O vazamento de memória pode gerar uma série de problemas, desde o mau funcionamento do programa por consumo excessivo de memória, até vulnerabilidades que podem ser exploradas por terceiros, como *denial of service* (negação de serviço), quebrando o programa em execução, até a leitura do intervalo de endereços de memória de um programa visando vazamento de informações. Segue um exemplo de programa com consumo excessivo de memória:

| // Declaração do ponteiro para inteiro.  **int \*ptri;**  **/\*** Alocação repetitiva de memória sem haver a liberação, levando ao esgotamento de memória e falha do programa. **\*/**  **while (1)**  **ptri = malloc(sizeof(int));** |
| --- |

# Identificando vazamentos com Valgrind

Uma boa prática para aqueles que desejam evitar problemas com alocação de memória em seus programas é o uso do software *Valgrind,* que detecta os erros decorrentes do uso incorreto da alocação dinâmica de memória.

Depois de feita a compilação, use o seguinte comando para executar seu programa exibindo o relatório de erros do Valgrind:

| **valgrind --leak-check=full ./PROGRAMA < entrada\*\*** |
| --- |

A Figura 1 mostra-se um exemplo de uso do Valgrind para detectar um vazamento de memória:

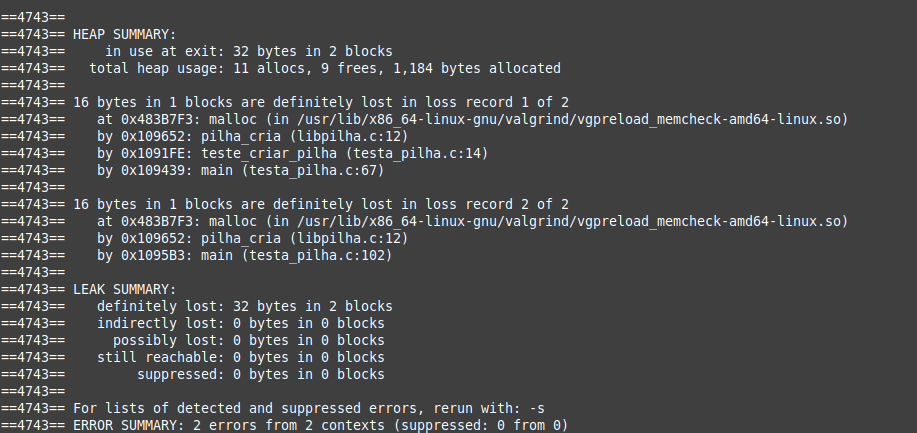


Figura 1: Saída do Valgrind com detecção de vazamento de memória

É interessante rodar o comando com a *flag* **-s**, que faz com que os erros detectados e suprimidos sejam exibidos de modo mais detalhado.

Note no início da Figura 1 (*HEAP SUMMARY*) que o Valgrind informa que, ao fim do programa, ainda havia 32 bytes em uso, isto é, alocações cuja memória não foi liberada (linha: ***in use at exit: 32 bytes in 2 blocks***). Na linha subsequente, é mostrado o uso total da *heap*, que ajuda a ver quantos *mallocs* não tiveram seus respectivos *frees* (a conta deve bater!): ***total heap usage: 11 allocs, 9 frees, 1184 bytes allocated***, indicando que foram feitos 11 mallocs e apenas 9 frees, emais de 1Kbyte foi alocado durante a execução do programa monitorado pelo Valgrind.

A seguir, mostra-se cada um dos 2 blocos ***in use at exit*.** No primeiro bloco, onde é indicado que 16 bytes foram perdidos, note que o Valgrind exibe a função, o arquivo do código e a linha de código que geraram o alerta:

* O *main()* do arquivo *testa\_pilha.c*, na **linha 67** (Figura 2), chama a função *teste\_criar\_pilha()*, a qual faz uma chamada de *pilha\_cria()* na **linha 14** (Figura 3)do mesmo arquivo:

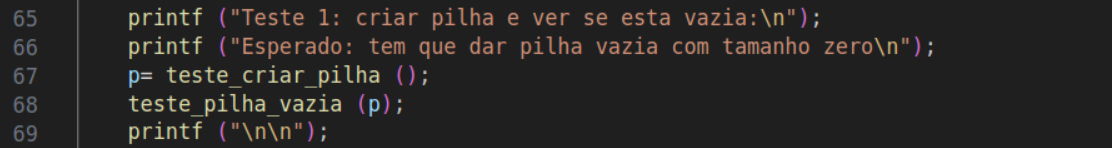


Figura 2: função *main()* do arquivo *testa\_pilha.c*

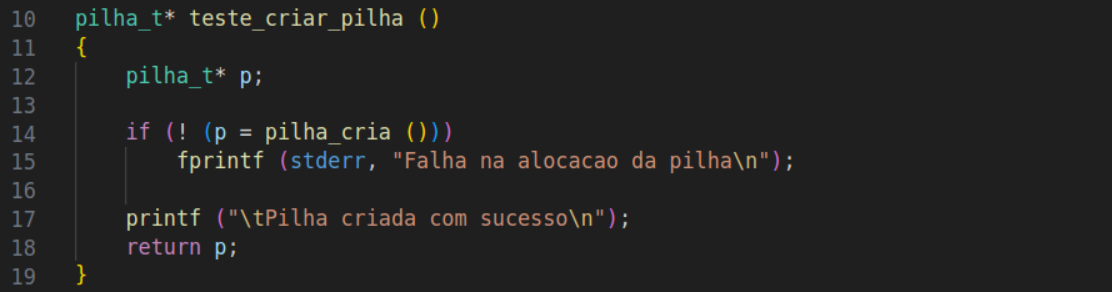


Figura 3: definição da função *teste\_criar\_pilha()* no início do arquivo *testa\_pilha.c*

* A função *teste\_criar\_pilha()* por sua vez chama *pilha\_cria()*, que faz um mallloc na **linha 12** do arquivo *libpilha.c* (Figura 4).

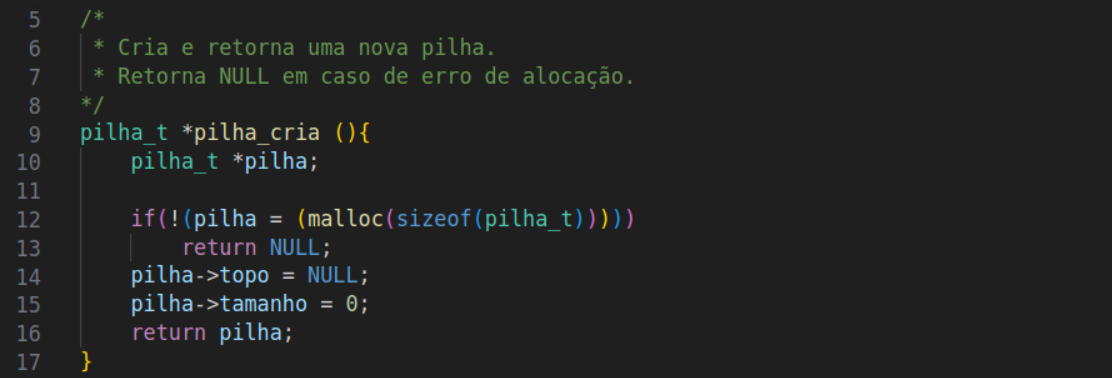


Figura 4: definição da função *pilha\_cria()* no arquivo *libpilha.c*

* No segundo bloco do relatório mostrado na Figura 1, o Valgrind informa que mais 16 bytes foram perdidos em alocações não liberadas, desta vez devido a algo iniciado na **linha 102** da função *main()* do arquivo *testa\_pilha.c* (Figura 5).

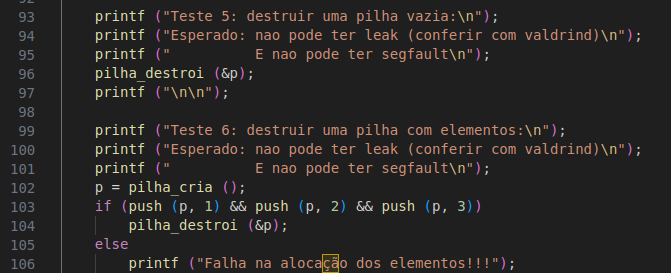


Figura 5: chamada da função *pilha\_cria()* no *main()* do arquivo *testa\_pilha.c*

O que ocorre é que após a criação do espaço de memória para a pilha “p” e seu uso, não é feita a liberação da memória após o tempo de vida esperado de “p”.

Isto ocorre devido a um problema na função *pilha\_destroi()*, que libera a memória alocada para cada *nodo\_t*, mas não a da *pilha\_t*… Para corrigir o erro, basta adicionar a chamada de *free* na **linha 27** da função responsável pela liberação, *pilha\_destroi()*, como mostrado na Figura 6.

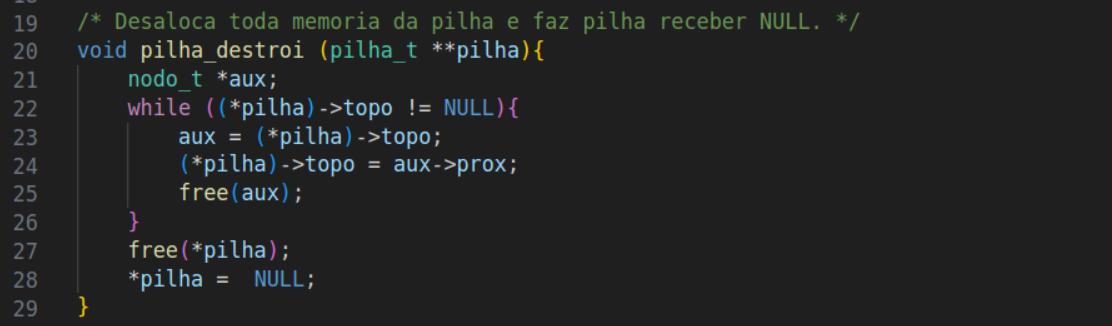


Figura 6: Função *pilha\_destroi()* corrigida no arquivo *libpilha.c*

Ao adicionar a chamada faltante ao *free*, os 32 bytes “vazados” da memória referentes às chamadas à função *pilha\_cria()*, nas linhas 67 e 102 do arquivo *testa\_pilha.c* desaparecem. Após a correção e reexecução do Valgrind, a saída esperada é que não se tenha nenhum byte/bloco em uso ao finalizar a execução do programa, como ilustrado na Figura 7.

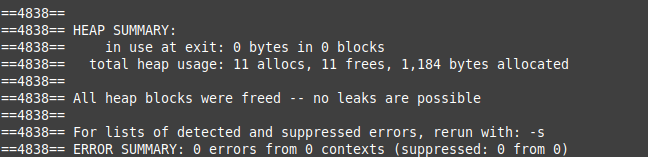


Figura 7: Saída do Valgrind sem erros!

A mensagem **“*All heap blocks were freed - - no leaks are possible*”** nos informa que não há vazamentos possíveis. Na última linha, é exibido um sumário de erros - no caso, 0.

O comando a seguir permite observar a origem de todos os valores não inicializados, porém ressalta-se que o custo de execução com o parâmetro *yes* no argumento *--track-origins* é alto.

| **valgrind --leak-check=full --track-origins=yes ./PROGRAMA** |
| --- |

A *flag* ***-v*** significa *verbose* e permite que o relatório seja mais detalhado de forma compreensível para o usuário. É possível obter um relatório mais detalhado do que pode ser feito usando o *Valgrind* com o comando:

| **valgrind --help** |
| --- |

# Pontos importantes!

Em resumo:

* Um ponteiro não-inicializado pode armazenar um valor indeterminado, enquanto que um ponteiro nulo armazena o valor determinado “NULL“ (que não é o endereço válido para qualquer objeto do programa ao qual o ponteiro pertence).
* Um ponteiro pendente é o ponteiro que é liberado de forma incorreta (sem ser apontado para NULL), podendo causar seu uso posterior com comportamento indeterminado.
* O vazamento de memória ocorre quando um ponteiro não é liberado após sua vida útil, tornando aquela parte da memória inacessível e desperdiçando recursos do sistema.
* A ferramenta Valgrind ajuda a identificar vazamentos de memória em programas, contabilizando bytes perdidos e chamadas à *malloc* e free, e exibindo as funções envolvidas no vazamento (indicando a linha e o arquivo do código-fonte).

Referências:

* <https://wiki.sei.cmu.edu/confluence/display/c/MEM31-C.+Free+dynamically+allocated+memory+when+no+longer+needed> - Carnegie Mellon University Software Engineering Institute - SEI CERT C Coding Standard Wiki.
* <https://www.geeksforgeeks.org/dangling-void-null-wild-pointers/>
* <https://cwe.mitre.org/data/definitions/401.html> - Common Weakness Enumeration - CWE 401 - Missing Release of Memory after Effective Lifetime.
* <https://encyclopedia.kaspersky.com/glossary/use-after-free/> - Encyclopedia by Kaspersky - Use-After-Free.
* <https://owasp.org/www-community/vulnerabilities/Memory_leak> - OWASP - Memory Leak.
* <https://web.stanford.edu/class/archive/cs/cs107/cs107.1174/guide_valgrind.html> - Nate Hardison & Julie Zelenski - Guide to Valgrind.