USP - UNIVERSIDADE DE SÃO PAULO

INSTITUTO DE CIÊNCIAS MATEMÁTICAS E COMPUTAÇÃO (ICMC)

DISCIPLINA INTRODUÇÃO À CIÊNCIA DA COMPUTAÇÃO

Memória RAM e suas Segmentações

Carlos Filipe de Castro Lemos

Número USP: 12542630

SÃO CARLOS

2021

1. Introdução

Nas áreas da informática, é muito comum falar-se em memória, mas, pelo menos em um primeiro momento, não é dos temas mais largamente explorados, o que pode causar muitas dificuldades no aprendizado de jovens acadêmicos ou problemas de difícil solução na vida dos programadores profissionais.

Nesse contexto, pretende-se com o presente trabalho fazer uma exposição objetiva sobre a Memória RAM e suas Segmentações, focando em aspectos conceituais, formas de funcionamento, limitações, bem como sobre pontos positivos ou negativos. Ressalte-se que as abordagens oferecidas serão especialmente em relação às memórias *stack* e *heap*.

2. A Memória RAM e as suas Alocações

1. Em primeiro lugar, é preciso especificar de qual memória trataremos ao longo do texto a fim de sintonizar a comunicação e evitar equívocos. De modo bastante superficial, memória é todo dispositivo que guarda ou armazena dados, sendo certo que podem ser divididos em memórias principal e secundárias. As memórias principais são essenciais para o funcionamento da máquina e operam com dados de forma temporária (como a memória RAM "*Random Acess Memory*"), enquanto que as memórias secundárias armazenam dados de forma permanente (como pen drives, SSD ou discos rígidos). Assim, é imperioso mencionar que abordaremos o tema com foco na memória principal.

2. A conexão entre a memória lógica (virtual dos *softwares*) e a memória física (espaços físicos do *hardware*) é realizada pelo gerenciador de memória chamado de MMU (*Memory Management Unity*). Dessa forma, quando um programa é executado, as informações vindas do processador serão armazenadas na memória principal através da atuação do gerenciador de memória. Esse processo é chamado de alocação de memória, sendo certo que o caminho inverso (de liberação da memória) é nomeado de desalocação.

Vale ressaltar que, quando o programa é executado em um sistema operacional, a alocação de memória acontecerá de forma organizada, uma vez que haverá segmentação das informações lógicas e consequente agrupamento dos dados em blocos de memória.

Nesse contexto, costuma-se abstrair conceitos para facilitar o entendimento, bem como categorizá-los. Desse modo, foram criadas várias expressões, sendo as mais importantes, relacionadas à memória principal, as chamadas de *text, data, stack e heap*:

- ***Text***: trata-se do pedaço de memória principal que armazena todo o código-fonte do programa e as suas constantes, sendo certo que possui um tamanho permanente (podendo sofrer alterações somente por meio de mudanças no código fonte).

- ***Data***: trata-se da parcela da memória principal que irá guardar as variáveis globais inicializadas e não inicializadas. É de se notar que essas variáveis são criadas pelo programador com a intenção de terem tamanho fixo ao longo de toda a execução do programa (como, por exemplo, uma *string* que armazena até 50 caracteres).

- ***Stack****:* trata-se da parcela da memória principal que armazenará variáveis locais, parâmetros e retornos que foram inicializados ou processados pelo programa. Essa porção poderá mudar de tamanho ao longo da execução do programa.

- ***Heap****:* trata-se do quinhão de memória principal que será usado para guardar blocos de memória que são alocados dinamicamente. Além de ter tamanho variável ao longo da execução do programa, é necessário que o programador faça a alocação de forma manual e explícita no código-fonte.

  Ademais, é interessante notar que a alocação pode se dar de três formas:

- ***Alocação estática***: quando da confecção dos programas, os programadores realizam declarações de variáveis globais (indicadas fora das funções) ou locais (indicadas dentro das funções), as quais, em tempo de execução do programa, serão armazenadas de forma estática na porção de memória *Text, Data ou Stack.*

- ***Alocação automática****:* têm esse nome porque são iniciadas automaticamente, ou seja, independente de o programador escrever códigos para essa finalidade. Normalmente são derivadas de bibliotecas (*packages*) e estão ligadas ao escopo local e aos parâmetros de funções, ficando guardadas na região *Stack*.

- ***Alocação dinâmica***: o controle de alocação de memória deve ocorrer por meio de ação explícita do programador, o qual deverá fazer a alocação e a desalocação de memória principal de forma manual ou semiautomática.

Assim, podemos visualizar o processo de alocação de memória de forma visual (Figura 1) ou de forma contextualizada (Programa 1):

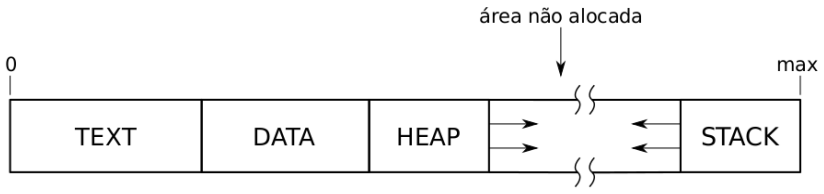


Figura 1. Representação visual da memória usada por um programa. Notar que a memória Stack e a Heap são variáveis

```
#include <stdio.h>

int a = 0; // Variável global de alocação estática na memória data.
const float PI = 3,14; // Constante de valor global e alocação estática na memória data.
```

```
void soma(int a, int b){  // Função auxiliar com parâmetros de escopo local (“a” e “b”),
                          // cuja alocação automática ocorrerá na stack.
        return a+b; // Procedimento de retorno para a função main(), o qual é armazenado
                    // na Stack. Sua execução promove desalocação das variáveis “a” e “b”.
}

int main(){ // Função principal sem parâmetros, mas com variáveis locais.
int *ponteiro; // Variável de escopo local e alocação na memória stack

ponteiro = (int*)malloc(40); // Alocação dinâmica de memória feita manualmente para a
                             // reserva de 40 bits de espaço na memória heap.

soma();

free(ponteiro); // Desalocação manual de 40 bits alocados na memória da heap.

return 0; // Desalocação automática de toda a memória alocada para o programa
}
```

Programa 1. Todo o código fonte acima é armazenado na memória Text.

3. Memória Stack

1. Etimologicamente, *stack* significa “pilhas”, ou seja, um amontoado de objetos que são colocados uns sobre os outros. Isso está relacionado com a forma como os processos ou procedimentos de execução do programa são operacionalizados. Além disso, o funcionamento da memória *stack* também é conhecido pela expressão “*last in, first out queues*”, o que, em tradução livre, significa que o “último a entrar é o primeiro a sair da fila.”

2. No entanto, como mencionado anteriormente, a *stack* consiste conceitualmente em uma porção da memória principal que é utilizada para armazenar variáveis automáticas de escopo local, o que acontece depois que a função é chamada. Isto é, cada uma das variáveis de escopo local, que têm valores fixos, serão empilhadas umas em cima das outras, e isso refletirá em eficiência, pois haverá ganho de *performance*, de forma que, para entendê-lo, é necessário conhecer seu mecanismo, que, embora complexo, é interessante.

3. Nesse passo, quando um programa é executado, teremos funções em andamento, as quais irão instruir a máquina na ordem prevista no seu algoritmo. A alocação de memória, com efeito, será essencial para virtualizar o programa e lhe dar meios de funcionamento. É de se anotar também que isso acontece de forma automática, sendo certo que uma das primeiras porções de memória reservadas será a “*stack frame*”, cujo tamanho

é determinado pela capacidade do sistema. Esclareça-se ainda que esse valor será desalocado quando o programa for encerrado, liberando memória para outras funções.

Aprofundando-se um pouco no assunto, é interessante notar que, dependendo da máquina, o valor disponível de memória principal poderá sofrer limitações de tamanho, pois arquiteturas de 32-bits conseguem acessar no máximo 4GB de endereços de memória, enquanto que aquelas outras de 64-bits ou superiores possuem capacidades mais robustas.

Em outras palavras, isso significa que teremos trilhões de números hexadecimais representando cada espaço físico da memória principal, os quais são iniciados em 0000000 e terminam em FFFFFFFF. Esses números, por exemplo, podem ser combinados dois a dois ou quatro a quatro hexadecimais, gerando 2.147.483.647 espaços de memória em uma arquitetura de 32-bits. Todavia, nem todos esses valores estarão disponíveis para o funcionamento do programa, uma vez que o tamanho da *stack frame* reservada é pequeno, atingindo 8 mb ou menos. Por outro lado, depois de reservado o *stack frame*, o programa estará apto para usufruir da possibilidade de armazenar dados na memória *stack*.

4. Ou seja, essa é uma realidade potencialmente inconveniente ao programador, a qual deve ser observada com atenção, uma vez que a alocação descontrolada da memória *stack* poderá gerar erros de compilação (quando se aloca variáveis muito grandes - “Segmentation Fault”) ou de execução do programa (quando, por exemplo, há excessivos empilhamentos que causam o estouro da capacidade armazenamento - *“Stack Overflow”)*.

Convém mencionar que esses problemas não são frequentes em programas de somenos importância, mas podem ser graves quando afetam servidores de sistemas que proporcionam o uso de compartilhado de um mesmo servidor.

5. Seja como for, depois de reservado a *stack frame*, o programa tem a possibilidade de armazenar dados na memória *stack* e isso pode ser traduzido no aprovisionamento de *valores* ou de *referências*. A primeira modalidade é mais comum, visto que associa valores a determinados endereços de memória (armazenamento de dados), enquanto que a segunda utiliza o mecanismo de referenciamento de dados por meio de ponteiros (armazenamento de endereços de memória).

Com o intuito de esmiuçar um pouco mais o assunto, podemos retomar a explicação acima de que cada valor é associado a endereços de memória diferentes, os quais são representados por um número hexadecimal. Assim, cada bit terá um número próprio, como, por exemplo, 7F, 7E, 7D, 7C, 7B, 7A, 79 e 78. Desdobrando o raciocínio, esses oito números hexadecimais, em conjunto, formarão um byte, cujo endereçamento poderia ser, ilustrativamente, 2FH. Indo além, podemos verificar que a combinação de dois ou mais bytes poderia formar, por exemplo, caracteres, palavras ou números.

	Bits								Resultado
	7F	**7E**	**7D**	**7C**	**7B**	**7A**	**79**	**78**	
Byte 2FH	0	0	0	0	0	0	0	1	O binário 00000001 armazenado refere-se ao número 1 decimal

Tabela 1. Exemplifica o armazenamento do número 1 binário.

Aliás, em linguagens de alto nível, podemos criar variáveis do tipo valores usando esses valores de endereços de memória para armazenar números binários que representarão caracteres e números. Tanto é assim que, em linguagem C, por exemplo, podemos fazer a seguinte correspondência entre variáveis e memória física:

a. ***Char***: guarda um número da tabela ASCII que corresponde a um caractere, sendo certo que cada caractere ocupa o tamanho é de 4 bits da memória (1 byte);

b. ***Int***: guarda um número inteiro, cujo tamanho é fornecido pelo barramento do processador. Vale dizer que, com a evolução dos processadores, o valor passou de 16 bits (4 bytes) para 32 bits (8 bytes);

c. ***Float***: guarda um número real com certa precisão (por meio de um número em vírgula flutuante - notação científica), cujo tamanho é de 16 bits (4 bytes).

d. ***Double***: guarda um número real com precisão mais apurada que o float, cujo tamanho é de 32 bits (8 bytes).

e. ***Booleano***: trata-se do tipo lógico que retorna valores verdadeiros (1) ou falsos (0).

f. ***Vetores***: são formados por um conjunto de variáveis que são alocadas em blocos contíguos (do tipo char, int, float, double), cujo tamanho irá oscilar de acordo com o tipo das variáveis. Por exemplo: um vetor unidimensional dotado de dez posições (vetor[10]) ocupará um espaço de memória de 40 bytes se for do tipo *int*, mas poderá ocupar 80 bytes se for do tipo *double*.

Por fim, ressalte-se que o armazenamento de valores ainda poderá sofrer variações de tamanho por meio de escolhas arbitrárias na confecção do programa, pois pode-se optar por modificar os tipos acima mencionados usando, por exemplo, números sem sinal (*unsigned*) ou ampliar ou reduzir o domínio de números reais (por meio do *short* e *long*).

6. Noutro giro, mencionamos acima que a memória *stack* serve para fazer o referenciamento de dados, armazenando, por exemplo, endereços de memória. De fato, isso pode acontecer por meio de ponteiros e, efetivamente, apresentam utilidade quando se

deseja fazer alterações de valores alocados na memória, mas são muito mais utilizados quando se deseja fazer a alocação de memória *heap*.

7. Assim, tendo em vista todas as informações mencionadas acima, podemos entender como o funcionamento da memória *stack* acontece por meio do programa abaixo:

```
#include <stdio.h>

int a = 0; // Na execução do programa, será empilhada em na memória stack

float soma (int f, float g){  // Parâmetros de escopo local (“a” e “b”) de alocação automática
                              // ocorrerá na Stack quando a função for chamada.
        return f+g; // Procedimento de retorno para a função main(), o qual é armazenado
                    // na Stack. Sua execução promove desalocação das variáveis “a” e “b”.
}

int main(){ // Função principal sem parâmetros, mas com variáveis locais.

int b = 0;  // Variável local que armazena valores do tipo inteira.
char c = ‘w’;
float d = 5,32;
double e = 5,3214234;
int *ponteiro;

ponteiro = &b; // O ponteiro recebeu o endereço de memória da variável “a”.

soma(b,d);
return 0; // Desalocação automática de toda a memória alocada para o programa
}
```

Programa 2. Exemplifica como é feita a atribuição de valores ou de referenciamento na memória stack.

Para facilitar a explicação, usaremos endereços de memória mais simples. Dessa forma, temos que, quando o programa for executado, a variável global “a” será criada e alocada na porção de memória data. Portanto, não aparecerá na memória *stack*.

Em seguida, a função main() é chamada e faz-se a inicialização das variáveis locais, o que, consequentemente, exigirá a alocação da memória *stack*. Nesse caminho, conforme as alocações acontecerem, e na ordem com que aparecerem, seus valores serão empilhados da seguinte forma:

Endereço de Memória	Nome da Variável	Valor Armazenado
4	ponteiro	0 (endereço de memória)

3	e	5,3214234
2	d	5,32
1	c	99
0	b	0

Tabela 2. Exemplifica a alocação de memória da função main()

Na sequência, haveremos de perceber que a função soma() será chamada, de modo que o fluxo de informações da função main() será interrompido e a execução do programa será transferida para a parte do código onde estão descritas as suas instruções da soma().

Nesse passo, perceberemos que os valores das variáveis “b” e “d” serão copiados para os parâmetros da função soma() (“f” e “g”). Veja-se bem: os valores das variáveis locais de main(), “b” e “d”, permanecerão alocados nos endereços de memória 0 e 2 e, por isso, a função soma() fará nova alocação de memória *stack* para armazenar os valores recebidos. Portanto, teremos um novo empilhamento de dados:

Endereço de Memória	**Nome da Variável**	**Valor Armazenado**
6	g	5,32 (igual ao “d”)
5	f	0 (igual ao “b”)
4	ponteiro	0 (endereço de memória)
3	e	5,3214234
2	d	5,32
1	c	99
0	b	0

Tabela 3. Exemplifica a alocação de memória da função main() e da função soma().

Quando a execução do programa chegar na instrução return da função soma(), ela irá retornar o valor da soma entre “f” e “g”, sendo certo que, na sequência, encerrará sua sua execução e os valores alocados nos endereços de memória 5 e 6 serão desalocados automaticamente pelo gerenciador de memória, liberando espaço para outras utilizações. Em ato contínuo, a execução da função main() será retomada no ponto que foi interrompida.

No próximo passo, a instrução return da função main() será lida e executada. Dessa forma, haverá o encerramento do programa, o que, automaticamente, provocará a desalocação de memória referente aos endereços 0, 1, 2, 3 e 4.

8. Por isso, a memória *stack* é chamada de pilha e seu procedimento é conhecido por "*last in, first out queues*". Afinal, conforme a necessidade, as variáveis podem ser alocadas em formato de pilha e, conforme são usadas, serão descartadas por meio da desalocação, sendo que os últimos valores empilhados são os primeiros a serem retirados.

9. Por fim, é necessário listar pontos positivos e negativos do uso da memória *stack*.

Em comparação com a *heap*, a *stack* é muito menor. Isso favorece o desempenho da máquina e também traz segurança de alocação, pois, se por um lado abrevia o tempo de acesso aos dados (porque o *stack frame* já possui todos os endereços de memória catalogados), por outro ele é reservado especificamente para o programa. Assim, pode-se chegar no valor da variável sem extensas varreduras e com mais segurança.

No entanto, apresenta como limitação o fato de que os seus valores ou que os seus objetos são de escopo local e, por esse motivo, somente podem ser acessados por meio de um método ou de uma função específica, que fora previamente acionada (diferente da *heap*, que pode ser acessada por qualquer parte do código através de um ponteiro que aponta para a sua posição da memória).

Além disso, o uso da memória *stack* pode ser um fator de ineficiência do programa e do próprio sistema operacional, pois, como não se sabe exatamente a demanda de uso da memória, a utilização de valores arbitrários pode alocar valores excessivos (ocupando espaço desnecessariamente) ou insuficientes (não atendendo às necessidades e às finalidades para as quais o programa foi feito).

4. Memória Heap

1. Etimologicamente, *heap* é proveniente do inglês e significa "amontoar". Isso guarda uma certa relação com as funções e finalidades da memória *heap*.

2. Conceitualmente, a memória *heap* pode ser vista como a porção da memória randômica onde valores ou referências são alocados dinamicamente pelos programas e, por isso, podem apresentar variações na localização (em espaços contínuos ou em piscinas ou ilhas de dados) ou em tamanho de alocação (seja porque foram criados por meio de variáveis que possuem valor desconhecido ao programador, seja em razão de eventos não previstos durante a execução do programa).

3. Em primeiro lugar, é interessante notar que, diferente da memória *stack*, a *heap* não tem valores estáticos, não é carregada no início da execução do programa e não apresenta alocação automática quando as funções são chamadas. Por isso, sua reserva de memória deverá ocorrer durante a execução do programa, lembrando que, nesse caso, será

necessário que o programador construa instruções específicas e explícitas para gerenciar a memória. Além disso, é oportuno ressaltar que algumas linguagens, como o C#, podem dispensar a desalocação, mas continuam a exigir a alocação.

4. Na linguagem C, utiliza-se funções presentes na biblioteca *stdlib.h* para gerenciar o uso de memória, sendo certo que *malloc()* faz a alocação, *free()* promove a desalocação e a *realloc()* redimensiona o tamanho do bloco de memória previamente alocado.

Anote-se que, diferente da memória *stack*, que possui o *stack frame*, a memória *heap* não possui uma lista finita de números hexadecimais reservados que indicam os endereços de memória para se fazer a alocação. Por isso, essas funções necessariamente precisam ser combinadas com o uso de ponteiros (variáveis da *stack* que armazenam endereços de memória), o que, embora irrisório, confere um caráter de pausa no acesso à informação. Aliás, essa é uma das razões que a memória *stack* é considerada mais ágil ou rápida em comparação com a *heap*.

5. Nesse ponto, é interessante fazer uma observação, afinal, quando a função malloc() for chamada, ela irá buscar espaços de memória *heap* e irá reservar um bloco de endereços de memória do tamanho indicado como parâmetro pelo programador. Importa mencionar também que, nesse conjunto de endereços, haverá um deles que será tomado como referência de acesso ao bloco de memória. Em condições normais de funcionamento, esse valor de referência será fixado na primeira posição do bloco. Ou seja, se usarmos vetores, o endereço de memória do vetor[0] servirá para essa finalidade; mas, caso sejam variáveis simpĺes, o próprio endereço da variável será suficiente.

6. Para demonstrar seu funcionamento, é possível supor, por exemplo, que uma professora precise fazer a média aritmética de 8 notas de trabalhos e provas (valores inteiros) para calcular a nota final do aluno. O programador poderia resolver a necessidade da educadora por meio de um código fonte que utilizasse 8 variáveis estáticas (do tipo inteiras) ou um vetor de 8 espaços (do tipo inteiros); aliás, ainda existiria uma terceira via de implementação que consistiria na alocação dinâmica de 8 espaços na memória *heap*.

Optamos por analisar a terceira via (referente a alocação dinâmica de 8 espaços de memória *heap* para armazenar valores inteiros), já que as outras duas utilizam espaço da memória *stack*. Nesse caso, poderíamos ter o seguinte código:

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

int main(){

   int *ponteiroHeap; // O ponteiro é empilhado na memória stack.
   ponteiroHeap = malloc(8*4); // Faz-se a alocação de espaço de memória para 8 números
                               // Cada inteiro ocupa 4 bytes, logo são reservados 32 bytes.
                               // A atribuição indica o primeiro endereço de memória do
```

```
                                    // bloco e toda vez que o ponteiro for acionado, ele apontará
                                    // inicialmente para esse endereço de memória.
    mediaAritmetica(); // Função que calcula a média aritmética.
    free(ponteiroHeap); // Libera a memória alocada dinamicamente.

    return 0;
}
```

Programa 3. Exemplifica como é feita a alocação dinâmica de memória heap.

Por outro lado, poderia acontecer de o programador ter sido contratado para desenvolver um programa que pudesse ser utilizado em outros anos letivos. Dessa forma, ela não poderia supor a utilização de 8 variáveis estáticas, pois a real demanda de espaço de memória, dependendo do contexto, poderia envolver diferentes quantidades de trabalhos ou de provas. Assim, em uma análise mais detida, a declaração de valores estáticos não seria eficiente, pois das duas uma: ou poder-se-ia estabelecer um valor muito grande (redundando em desperdício de memória), ou poder-se-ia usar um valor muito pequeno (frustrando os propósitos do programa). Desse modo, a solução mais adequada para o programador seria fazer uso da alocação dinâmica da memória *heap*, afinal poder-se-ia reservar um espaço de memória a qual correspondesse ao tamanho da necessidade. Nesse caso, o programador poderia fazer a alocação de memória da seguinte forma:

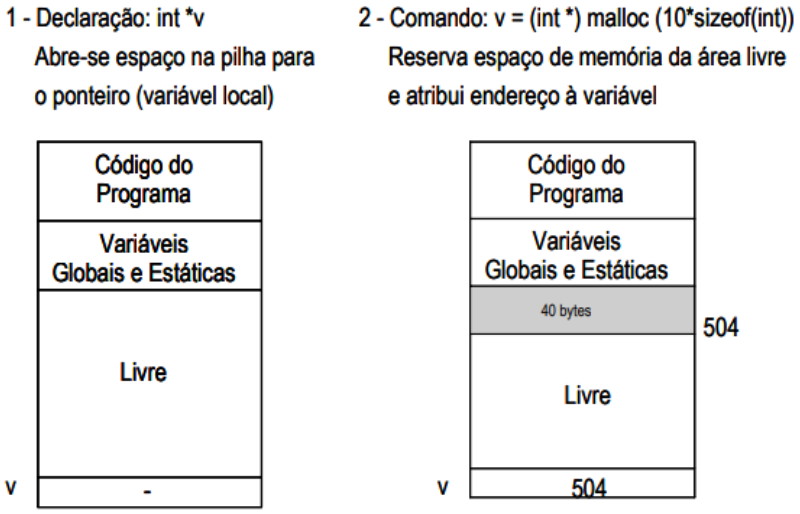
```
#include <stdio.h>
#include <stdlib.h>

int main(){
    int quantidadeNotas; // Variável alocada na memória stack.
    int *ponteiroHeap; // O ponteiro é empilhado na memória stack.
    scanf("%d", &quantidadeNotas); // Faz a leitura da quantidade de notas
    ponteiroHeap = malloc(quantidadeNotas*sizeof(int));
                            // Faz-se a alocação personaliza do espaço de memória.
    mediaAritmetica(); // Função que calcula a média aritmética.
    free(ponteiroHeap); // Libera a memória alocada dinamicamente.

    return 0;
}
```

Programa 4. Exemplifica como é feita a alocação dinâmica de memória heap *por meio de valor indefinido..*

Nesse momento, convém acrescentar uma observação, pois, em abstrações para facilitar o entendimento, a maior parte da doutrina sobre o assunto costuma dizer que a memória *heap* é alocada no extremo oposto da memória *stack*, embora essa informação pareça variar de acordo com a linguagem utilizada. Seja como for, assumindo que esteja correta, percebe-se que, conforme valores são empilhados na *stack*, e, ao mesmo tempo, conforme se amontoa dados na *heap*, o espaço de memória disponível para alocação entre elas diminui, o que pode ser visualmente verificado no exemplo abaixo:



7. Assim, se o gerenciamento da memória não for feito com cautela e eficiência, esse esgotamento da memória *heap* pode proporcionar erros de memória, tal como o *memory leak*. A causa mais frequente para este tipo de problema reside em erros de programação (ausência de boas práticas), pois, por exemplo, se não houver a liberação de memória, pode ocorrer seu consumo excessivo (lentidão), sendo esse problema mais perceptível, principalmente, em servidores de longa duração.

Em alguns outros casos, o programador também pode instruir a utilização de um ponteiro pendente (*dangling pointer*), ou seja, um ponteiro que não carrega endereço de memória (causando erros de execução) ou que aponta para diferentes regiões da memória (podendo provocar o encerramento prematuro do programa - "*crash"*). Em casos extremos, a utilização de *dangling pointers* pode danificar irremediavelmente o sistema operacional por alterar informações que lhe são essenciais (por exemplo, dados da kernel que estão na memória principal passando por algum processamento).

Outro ponto negativo da memória *heap* está no fato de que ela é mais lenta que a memória *stack*. Isso acontece porque, para acessar seus valores, é necessária a intermediação de um ponteiro e, às vezes, a busca pelo valor demandará uma varredura em longos trechos de memória.

8. Como pontos positivos, podemos mencionar que a memória *heap* é mais versátil que a memória *stack*, pois, como visto acima, ela otimiza a utilização da memória principal evitando desperdícios ou falta. Além disso, ela é muito maior que a *stack*, podendo, portanto, armazenar maior quantidade de dados. Por fim, podemos mencionar que, diferente da memória *stack*, a *heap* dá acesso ao seu conteúdo por qualquer função (independente da variável ter sido declarada); ou seja, não há restrição de escopo para usá-la, pois alterações de memória são feitas por meio de referenciamento de endereços.

Bibliografia

https://docs.oracle.com/cd/E19253-01/820-0446/chp-typeopexpr-2/index.html

https://homepages.dcc.ufmg.br/~hbarbosa/teaching/ufmg/2020-1/lp/notes/11-memory.pdf>

https://www.inf.ufpr.br/roberto/ci067/10_aloc.html

http://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/the-stack-and-the-heap.html

https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/
http://www.macoratti.net/vbn_conc.htm

https://www.eximiaco.tech/entendendo-ponteiros-e-a-heap-free-store/

https://www.inf.pucrs.br/~pinho/LaproI/Vetores/Vetores.htm

http://www.ic.uff.br/~cbraga/ed/apostila/ed05-vetores.pdf

https://www.treinaweb.com.br/blog/gerenciamento-de-memoria-no-c-stack-heap-value-types-e-reference-types

http://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/the-stack-and-the-heap.html

https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/

https://courses.engr.illinois.edu/cs225/sp2021/resources/stack-heap/

https://pt.wikipedia.org/wiki/Vazamento_de_mem%C3%B3ria