Memória RAM e suas segmentações [Bloco 2 - 02]

Universidade de São Paulo – ICMC [Instituto de Ciências Matemáticas e Computação] – São Carlos

Introdução à Ciência de Computação I [SCC0221] – Professor Fernando Pereira dos Santos

Danielle Modesti – Bacharelado em Ciências de Computação – 1° semestre 2021

No USP: 12543544

Pesquisa da Aula 17 (alocação dinâmica I)

Aula de *17/06/2021*

25/06/2021

Como a RAM funciona ao compilar e executar um código?

- Executável → carregado pelo S.O. até a memória
  - é o S.O. que faz o gerenciamento de memória, controlando alocações, protegendo, limpando e controlando o acesso de programas à memória principal.
- Antes da execução do programa, é reservada uma área de memória para a Stack e a Heap começa a ser formada (se necessário, e quase sempre é)
  - Em C: o compilador aloca uma região da memória para ser usada pelo programa;
  - A alocação pode ser estática ou dinâmica;
  - Na **execução** de códigos de programas, a memória RAM é acionada/acessada e alocada;

    - Quanto mais processos foram executados, mais capacidade de processamento é necessária para a RAM;
    - Quando o programa está em memória, ele acessa diretamente as rotinas, processos e registradores por ciclos de processamento, controlados pelo processador (que verifica quais processos estão sendo chamados na leitura do programa).

- Porém, há um limite de capacidade: é por isso que é bom saber trabalhar tanto com a Stack (para execuções mais rápidas e que lidam com menores quantidades de dados mais diretamente) quanto com a Heap (pode lidar com vários processos e programas complexos: ao fim da execução de uma tarefa, podemos desalocar a memória ali utilizada e reutilizar o espaço para outras finalidades, otimizando programas).

- Em C, ao contrário do que ocorre em outras linguagens de alto nível (como Python e Java, que têm 'coletor de lixo' ou *garbage collector*), precisamos nos preocupar em alocar (ocupar) e desalocar (desocupar - retirar as informações para que não fique 'lixo' de alocações anteriores) a memória de forma correta e fazer a manutenção do seu uso para evitar problemas no código.

C e Assembly em relação à memória

- É muito comum, nessas linguagens, o contato com gerenciamento de memória.
    - Alocação e desalocação do Heap feitas **manualmente** pela API (interface de programação de aplicações - conjunto de rotinas e padrões estabelecidos de um software) do S.O. (garbage collector);
    - os S.O.s mais recentes fazem uma limpeza de endereços de memória alocados e não mais utilizados (evitar preencher a RAM e precisar reiniciar o computador caso esqueçamos de liberar memória dinamicamente alocada);
- Assembly - comum manipulação da Stack quase que diretamente;
- C - Stack gerenciada pelo compilador;

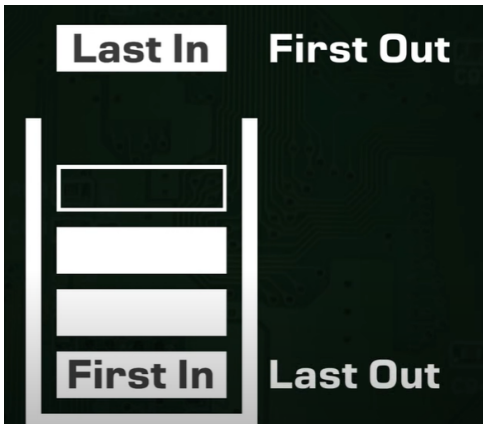
- Quanto mais alto nível é a linguagem, menos é necessário gerenciar a Heap e a Stack (por exemplo, em Java, há garbage collection automática).

Divisões da RAM

- A RAM (a qual é acionada quando rodamos nossos programas), virtualmente, é dividida em diversos segmentos que possuem diferentes usos e funções. Entretanto, aprofundar-se-á nos segmentos de **memória Stack** e **memória Heap;**
- Não existe fisicamente uma área da memória específica para a Stack e nem uma área fisicamente reservada para a Heap;
- Usamos o conceito de pilha para entender o funcionamento dessas memórias.

Memória Stack

- Estrutura de dados
- Divisão da RAM que lida com dados de uma maneira **"Último a entrar, primeiro a sair";**
  - Não posso **adicionar nem remover dados no meio dela**: apenas sobre o elemento anterior, de forma sequencial



- Memória-pilha: A Stack é comparada, para entendimento visual, com uma pilha de papéis sobre uma mesa. Ao adicionar papéis uns sobre os outros, o primeiro a ser visualizado é o último a ser adicionado à pilha;

💡 Adicionar/remover dados → a estrutura de dados Stack precisa percorrer sequencialmente com o 'last in, first out' (como ela armazena novas informações)
Para alterar valores → se for um vetor (implementação sequencial) - posso acessar índices específicos e alterá-los
Se for outra implementação de Stack: depende, pode ser ineficiente fazer isso.

> Stack/Pilha → forma otimizada de organizar dados na memória alocados em sequência e abandonados (não são desalocados da memória). Há empilhamento.

- A Stack está localizada em algum lugar na memória principal. Não existe como uma entidade própria e especial: todos os dados armazenados na Stack estão em algum lugar da memória RAM.
- Todos os elementos na Stack podem ser acessados a qualquer momento, mas o tamanho da Stack não muda a não ser que ocorra uma operação de "`pop`" (desempilhar/remover um elemento do topo da pilha) ou "`push`" (empilhar o último elemento ao topo da pilha)

Usando a Stack:

1. Quando o programa inicia, ele começa lendo a função main(). A Stack acessa a main e reserva os espaços de memória pedidos no programa

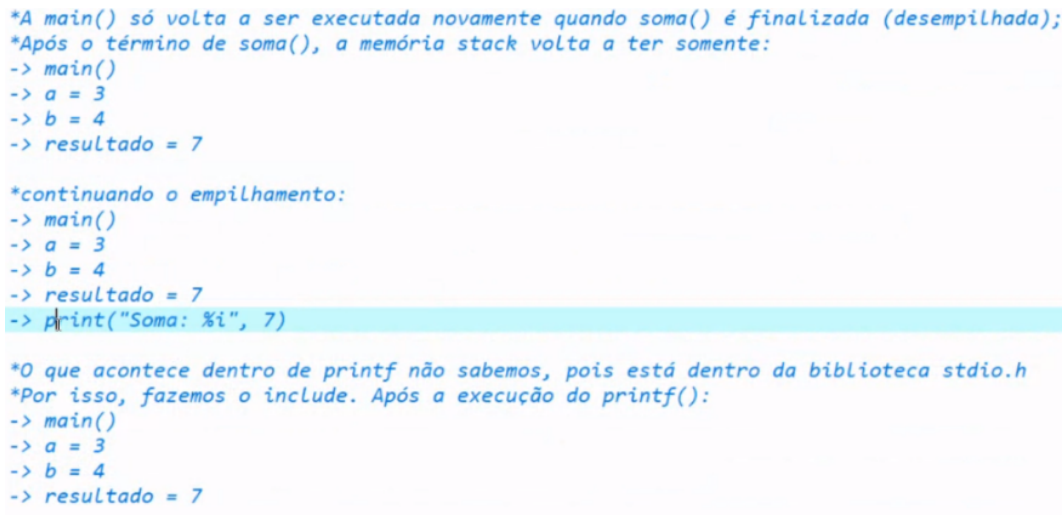
```
/*Todo programa começa pela função principal. Na prática:
    - o programa inicia e empilha a função principal na Memória Stack (pilha);
    - e continua o processo com funções e variáveis
```

```c
int soma(int x, int y){
    return x + y;
}

int main(){
    int a=3, b=4, resultado;
    resultado = soma(a,b);
    printf("Soma: %i", resultado);

    return 0;
}
```

2. Quando ele encontra uma outra função soma(), por exemplo, com seus próprios parâmetros de entrada, com passagem por valor, ele faz uma cópia dos valores desses parâmetros, envia para a função soma(), a qual vai para o topo da pilha de execução
3. Dessa forma, a execução da função main() (que está embaixo na pilha) é interrompida → enquanto executamos a função soma
4. Quando a função soma() termina, ela é desempilhada da Stack → isso inclui todos os seus elementos (variáveis e processos de escopo local ativos). Se a função fosse executada novamente, somente as variáveis 'a', 'b' e 'resultado' (variáveis da main()) estariam alocadas na memória e seria necessário novo empilhamento de variáveis locais da função soma(). As variáveis da main() só são desempilhadas da Stack quando termina a sua execução.

   Portanto, só há desempilhamento dos dados de uma função da Stack quando ela TERMINA.
5. A main(), que agora está no topo, volta a ser executada. Quando termina sua execução, a memória Stack fica vazia (a main é a first in, last out).

```
Memória Stack para o nosso exemplo:
-> main()
-> a = 3
-> b = 4
-> resultado = ? (sem inicialização; algum lixo de memória)
-> soma(3,4)
-> x = 3
-> y = 4
```



O empilhamento ocorre conforme a execução é feita

- Alocar/Desalocar → processo **automático**, não manual (é mais fácil, mas não dá a liberdade ao programador de alocar sua própria memória como desejar)
- Ao compilar o programa, há instruções de máquina próprias para alocar/desalocar memória (o compilador faz isso para nós)
- Ao declarar vetor[10], o programa reserva, sozinho, 10 espaços de memória, iguais entre si (depende do tipo de dado - um vetor de 10 inteiros alocaria 10 x 4 endereços de memória de 1 byte)

- Seus dados são colocados e abandonados da memória conforme o uso (terminou a função: desaloca os endereços de memória ocupados por seus processos. Se ela só estiver suspensa e chamando outra função, NÃO desaloca - isso somente ocorre ao fim da execução da função)

- A maioria das arquiteturas de computador não tem facilidade de manipular a Stack da memória (costuma ter só o registrador de ponteiro de pilha - Stack pointer).

Como é organizado o armazenamento na Stack?

- Apesar de ser capaz de acessar qualquer endereço da memória RAM, precisa adicionar ou remover um por vez numa pilha sequencial, em um

esquema de `"Last In, first out`" (quando for alterar a quantidade de informações lá armazenadas)

  - Empilhar os objetos um em cima do outro e acessar o que está no topo (para alterar a quantidade de elementos na pilha, é preciso percorrer elemento por elemento na pilha, ir 'tirando os papéis empilhados' que estão acima)

- Não é possível colocar dados na Stack sem saber seu tamanho na hora de execução (ela é limitada, não suporta elevada quantidade de dados)
- As linguagens de alto nível determinam a quantidade de espaço reservada para a Stack

Cada necessidade de alocação é um trecho da *stack* que vai sendo usado sempre em sequência determinado por um marcador, ou seja, um apontador, um **ponteiro**, se "movimenta" para indicar que uma nova parte na sequência desta porção reservada está comprometida.

Quando algo reservado para um segmento não é mais necessário, este marcador se movimenta em direção contrária a sequência de dados indicando que alguns desses dados podem ser descartados (sobrepostos com novos dados).

A alocação de cada trecho da memória não existe na *stack*, é apenas o movimento deste ponteiro indicando que aquela área será usada por algum dado.

A grosso modo podemos dizer que a aplicação tem total controle sobre a *stack, exceto quando acaba o espaço disponível nele.

Existem recursos para alterar manualmente o tamanho da *stack*, mas isso é incomum.

- Cada thread tem sua Stack (novas funções empilham novos frames de Stack enquanto o frame anterior fica suspenso e alocado na memória).

- Quando funções são chamadas, ponteiros e parâmetros são gravados no topo da pilha para que a função chamada tenha acesso aos parâmetros para executar e depois o programa possa continuar executando do ponto onde houve chamada de função.

Em que situações é aconselhável o uso da memória Stack?

- Armazenar valores temporários quando temos muitos dados para utilizar.
    - Muitos sistemas antigos tinham poucos registradores para processar dados. Se fosse preciso utilizar mais registradores, era preciso armazenar valores temporários em algum lugar da memória. Com a Stack, retiramos valores dos quais não precisamos e depois os retornamos na pilha
    - Não há necessidade de rastrear quais endereços de memória específicos guardam valores temporários
    - Entretanto, isso só é aconselhável quando estamos guardando, relativamente, poucos dados, ou se planejarmos com antecedência em qual ordem retiraremos (usaremos "pop") os valores (lembrando que a Stack só recupera dados de forma sequencial).

- Trabalhando com relativamente poucos dados, requeridos apenas enquanto uma função estiver ativa. Usamos a Stack por ser mais rápida e fácil de manusear (ter processos mais automáticos).

- Passagem por parâmetros
    - Os frames de Stack ajudam a estruturas os dados melhor
    - Parâmetro ou argumento: valor necessário para uma sub-rotina para que ela funcione corretamente
    - Se os parâmetros da sub-rotina são colocados ('pushed') na Stack antes da rotina/função ser chamada, esta é capaz de encontrar os parâmetros em um frame abaixo do frame atual (o acesso aos parâmetros fica abaixo do acesso ao endereço do retorno)
    - Como a Stack está na memória, todos os elementos podem ser acessados a qualquer hora
    - A sub-rotina acessará corretamente seus parâmetros, a qualquer momento da execução
    - Útil para retornos múltiplos que não podem ser armazenados ao mesmo tempo (serão retirados da Stack)

    - Stack: usada por funções, métodos e procedures

- Usar **Stack** quando se deseja (e é possível, é claro) uma execução mais rápida do programa, visto que ela **é mais veloz que a Heap**.
    - **Quando acessamos a Heap, é preciso passar por duas camadas de abstração: a Heap e a Stack**
    - Quando acessamos a Stack, só percorremos ela.

Como funciona isso:

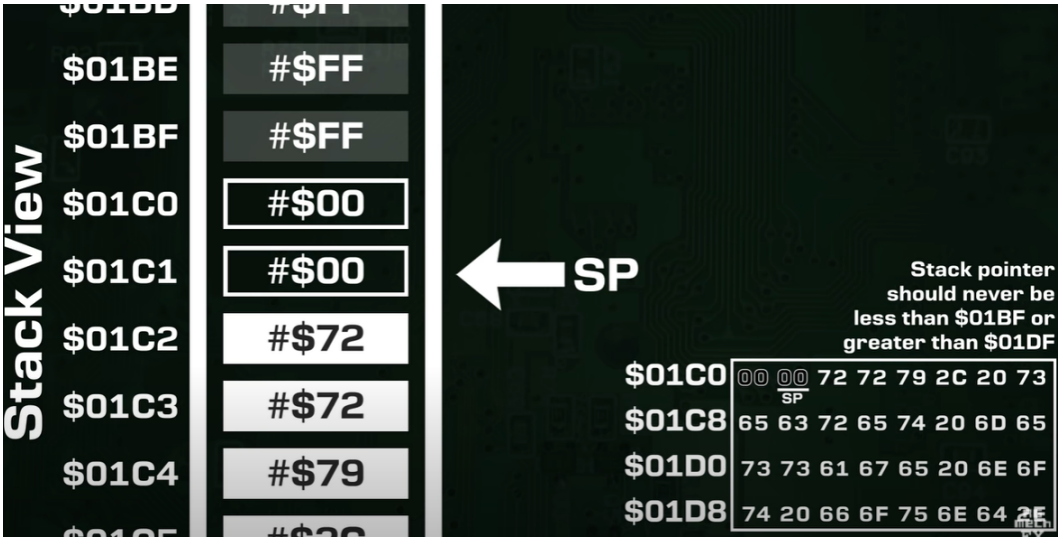
- Quando alocamos coisas na memória Heap, o endereço da região de memória alocada passa para a Stack. Assim, ao acessar elementos na Heap, encontramos o ponteiro/endereço e só aí o enviamos para a pilha da Stack;
- Por outro lado, ao acessar algo na Stack, já estamos dentro dela. Ou seja, a velocidade se deve ao fato do elemento que queremos acessar já estar naquela camada de abstração (na própria 'pilha')
- Quando damos o malloc(), uma função de alocação dinâmica cujo resultado é passado para um ponteiro, o SO vai na memória Heap (maior, dinâmica, desorganizada) e reserva um pedaço de memória que alocamos. Depois, passa o ponteiro dessa alocação para o programa, para indicar algo como: "o espaço reservado está e começa aqui"
- Na memória Heap, o SO acessa e reserva diretamente os espaços que pedimos para alocar e devolve para o programa o endereço exato (em qual posição de memória - para isso, retorna um ponteiro, um endereço). Depois disso, precisa passar pela Stack.

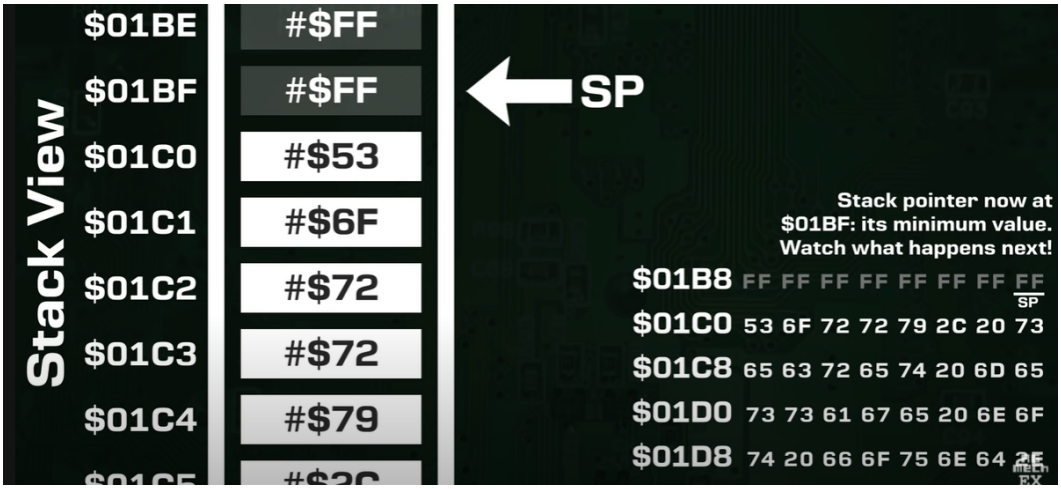
Existem prejuízos em abusar da memória Stack?

- Sim. Abrigar muitas sub-rotinas (por exemplo, chamar uma função dentro de outra múltiplas vezes → '*nesting subroutines results in more stack frames and in a taller stack*') aumenta demasiadamente o tamanho da pilha Stack, a qual é limitada. Ultrapassar esse limite resulta em *Stack Overflow*.

- Como funciona: chamamos uma função, com todo o seu processamento, parâmetros de entrada e de saída (este é o retorno)
- Os bytes com os endereços para o retorno da função estão na base do frame da stack
- Quando a sub-rotina (uma nova função) é chamada, o frame atual da Stack é suspenso e um novo frame é criado acima (aumenta o que está armazenado na Stack)

- Stack Overflow - quando o 'ponteiro' da Stack (stack pointer) ultrapassa a capacidade máxima definida
  - Stack Pointer (SP) → aponta para algum lugar na memória (algum endereço) que marca o limite entre memória livre para alocação (mais dinâmica) e os valores para a Stack (memória estática)
  - Se sabemos que o código executado não fará a Stack crescer muito, definir uma pequena porção para a alocação e uso da Stack é suficiente
  - O ponteiro da Stack jamais pode apontar para um endereço fora do seu alcance. Se isso ocorrer, há problemas de execução
  - Hoje, é raro ocorrer stack overflow, pois os compiladores reservam como default uma grande quantidade de memória stack no projeto





Overflow: valores 8C e 14 fora do máximo da Stack

SO avisa: 'out of bounds'

- Stack Overflow → ocorre quando tentamos colocar algo na stack e não há espaço reservado disponível
- Execuções recursivas descontroladas causam stack overflow: quando a stack está no topo, quando adicionamos um elemento a mais, não há mais endereço para acessar (a pilha acabou)
    - Quando tentamos empilhar mais uma coisa, dá erro

- No Stack underflow (retirar valores/'pop values' da Stack quando ela está completamente vazia), também há problemas.

- Tentamos resgatar ou remover algo da pilha
- Tentamos retornar um elemento e não tem nada.

Memória Heap

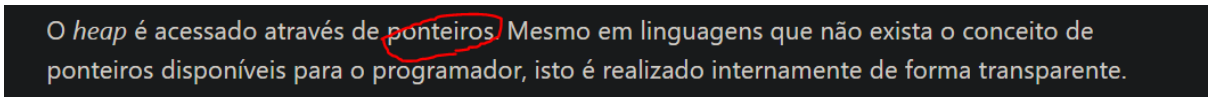
> Heap/Monte → organização da memória mais flexível, permite o uso de qualquer área lógica disponível. Organização mais dinâmica e "jogada", livre.

- Alocação dinâmica de memória: com malloc(), realloc(), calloc() e free()
- O programador é quem aloca e desaloca espaços de memória para utilizar → memória bem flexível e complexa de trabalhar
- Quando não desejamos mais manipular aquele endereço de memória, precisamos desalocar (free())
- Elementos alocados na Heap podem ser manuseados e desalocados a qualquer momento e em qualquer ordem; acesso mais lento que Stack; pode levar à fragmentação de memória (espaços perdidos entre regiões de memória utilizada).

Como é organizado o armazenamento na Heap?

- Ao contrário da Stack, não impõe um modelo, é um padrão dinâmico de alocação de memória
- Não é muito eficiente, mas flexível → é porque precisa passar pela Stack de qualquer maneira, para acessar os dados
- Alocamos e desalocamos pequenos trechos de memória, só para a necessidade do dado

- A alocação pode ocorrer em qualquer parte livre da memória disponível para o processo
- Desalocação do Heap
  - Pode ocorrer manualmente, com risco de bugs
  - Pode ocorrer por meio do garbage collector do SO, que identifica quando uma parte do Heap não é mais necessária
  - Quando uma aplicação se encerra
- O Heap não é uma área da memória, mas um conjunto de pequenas áreas de memória (é fragmentado por toda a memória)
  - alocam-se os endereços disponíveis, os quais, geralmente, não são sequenciais
- O conteúdo só é apagado manualmente ou quando a área disponível for escrita novamente
- A alocação no Heap 'custa caro'; o SO deve fazer muitas tarefas para garantir a perfeita alocação de uma área para um de seus trechos



- Com a Heap: mesmo após encerrar a execução de uma função, a memória não é desalocada. O programador precisa dar free() no espaço de memória que alocou (trabalhando com linguagem C ou Assembly)

Quais as principais diferenças entre Stack e Heap?

- Quanto ao tipo de estrutura de dados
  Stack - estrutura de dados linear
  Heap - estrutura de dados hierárquica (é dividida em níveis pelos seus elementos)
- Quanto à alterações na quantidade de elementos
  Stack - precisa vasculhá-la sequencialmente, no esquema de 'last in, first

out'
Heap - encontra e altera diretamente no endereço.

- Quanto à alocação e desalocação de memória (space management)
  Stack - ocorre automaticamente, por ação do compilador e do SO. A desalocação ocorre quando a função termina. Nunca ocorre fragmentação. É alocada sequencialmente.
  Heap - é manual, o programador precisa se atentar a isso, explicitando desalocações. Pode causar fragmentações de memória. Aloca espaços onde for possível

- Quanto ao armazenamento

  Stack - o tamanho é definido pelo compilador. Não pode ultrapassar essa delimitação
  Heap - o tamanho é mais 'flexível': é qualquer espaço lógico disponível, ou seja, de uma capacidade bem maior que a que a Stack provê.

- Quanto ao acesso e à manipulação de variáveis
  Stack - só acessa variáveis locais (de cada função) e não pode redimensioná-las
  Heap - acessa variáveis globais e pode redimensioná-las

- Quanto à velocidade de acesso de dados
  Stack - uma camada de abstração - mais rápida
  Heap - duas camadas de abstração - mais lenta

- Desafios
  Stack - pouca capacidade
  Heap - fragmentação de memória

As duas segmentações da memória diferem muito em tamanho (ou capacidade de armazenamento)?

Sim. Normalmente, a Stack, para a linguagem C, reserva cerca de 8 MB de memória. A Heap permite alocar qualquer região lógica disponível.

Em que situações é aconselhável o uso da memória Heap?

- Normalmente na criação de objetos ou ponteiros para estruturas de dados.
- Se o tamanho dos dados for indeterminado ou possivelmente grande, provavelmente a alocação deva ser feita na Heap
    - exemplo: criando um array muito grande e manter variáveis armazenadas por um longo período de tempo

- Quando não sei, *a priori,* o tamanho do que vou utilizar;
- Para manipulação de dados mais 'específicos', evitando declarar espaços na memória desnecessariamente (eu aloco a quantidade que quero)
- A Heap permite alocar espaços não sequenciais para posições dos arrays, algo que a Stack não permite, pois aloca sequencialmente;
- Ex.: Fila de Impressão → quando o elemento na posição [0] passa pela fila e é executado, o próximo deve ocupar esta posição para disponibilizar espaço aos próximos. Se a alocação fosse estática, a fila ficaria com espaços vazios (suspensos) e rapidamente seria preenchida e teria sua capacidade excedida, parando a execução (isso é inviável).

Bibliografia

https://pt.stackoverflow.com/questions/3797/o-que-são-e-onde-estão-a-stack-e-heap → Um pouco sobre as tais "pilhas", um conceito abstrato

https://www.youtube.com/watch?v=IWQ74f2ot7E&ab_channel=RetroGameMechanicsExplained → sobre a Stack: "Last in, first out"

https://www.sanfoundry.com/c-question-run-time-stack-usage/ → Memória e linguagem C

https://www.youtube.com/watch?v=XETZoRYdtkw&ab_channel=Computerphile → Como a memória do computador funciona?

https://www.techtudo.com.br/artigos/noticia/2012/02/o-que-e-memoria-ram-e-qual-sua-funcao.html → O que é memória RAM?

https://pt.stackoverflow.com/questions/209542/como-um-programa-é-carregado-na-memória-e-depois-executado → Como partir do código para a memória?

https://canaltech.com.br/produtos/o-que-e-thread/ → Sobre threads

https://www.guru99.com/stack-vs-heap.html → Informações sobre Stack e Heap