

# Sistemas Hardware-Software

## Aula 11 – Alocação de memória

2021 – Engenharia

Maciel Vidal  
Igor Montagner  
Fábio Ayres

# Alocação de memória

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define MAXW 512
5  #define MAXH 512
6
7  int main(int argc, char *argv[]) {
8      int mat[MAXH][MAXW];
9
10     /*
11        trabalhar com arquivo PGM
12     */
13
14     return 0;
15 }
```

# Alocação de memória

- Qual o consumo de memória do programa anterior?
- Ele varia conforme o tamanho da matriz?
- E se precisarmos de matrizes maiores?

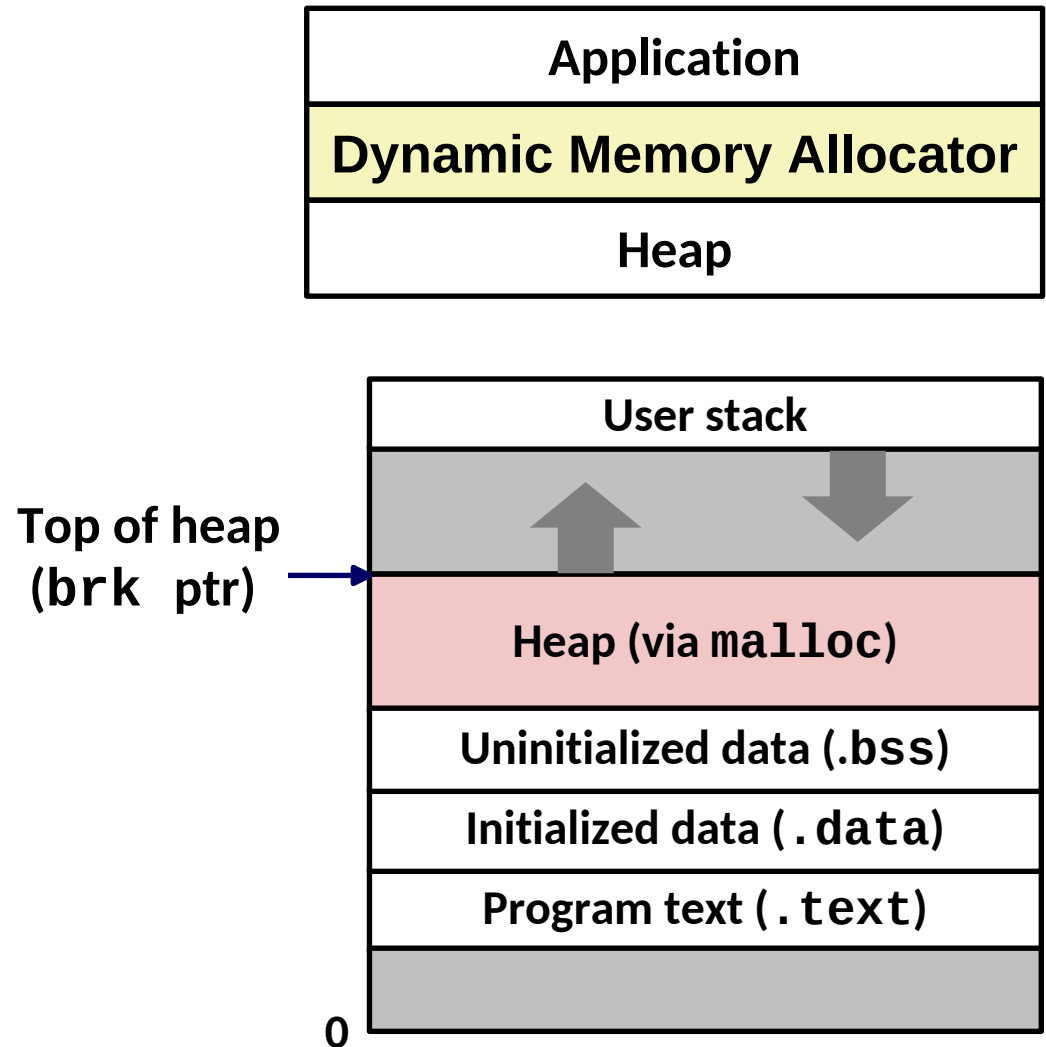
# Alocação estática

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define MAXW 512
5  #define MAXH 512
6
7  int main(int argc, char *argv[])
8  {
9      int mat[MAXH][MAXW];
10
11      /*
12       * trabalhar com arquivo PGM
13       */
14
15      return 0;
16 }
```

- Tamanho definido no momento da compilação
- Armazenado na pilha (se for variável local) ou no arquivo executável diretamente (se for global)

# Alocação dinâmica de memória

- Programas usam **alocadores de memória dinâmica** para criar e gerenciar novos espaços de memória virtual
  - C: malloc, free
  - C++: new, delete
- A área do espaço de memória virtual gerenciada por estes alocadores é chamada de **heap**



# Alocação dinâmica de memória

- Alocadores organizam o heap como uma coleção de blocos de memória que estão **alocados** ou **disponíveis**
- Tipos de alocadores
  - Explícitos: usuário é responsável por **alocar** e **dealocar** (ou liberar) a memória. Exemplo: malloc, new
  - Implícitos: usuário não precisa se preocupar com a liberação da memória. Exemplo: **garbage collector** em Java

# malloc

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

```
void *malloc(size_t size)
```

sizeof(int) \* n

**Se bem sucedido:** retorna ponteiro para bloco de memória com pelo menos **size** bytes reservados, e com alinhamento de 16 bytes (em x86-64). Se **size** for zero, retorna **NULL**.

**Se falhou:** retorna **NULL** e preenche **errno**

# free

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

```
void free(void *p)
```

```
int calc() {  
    int* vec = malloc  
    return;  
}
```

SEM FREE

Devolve o bloco apontado por **p** para o *pool* de memória disponível

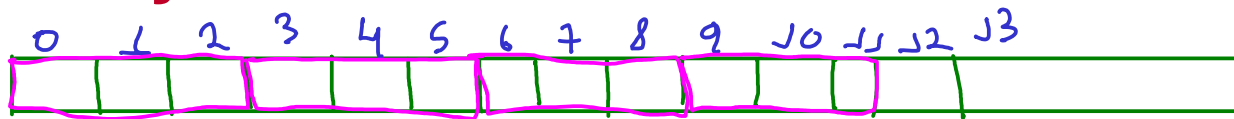


# Alocação dinâmica

$h=4, w=3$

$VEC[12]$

$VEC[h][w]$



```
int main() {  
    int w = 512;  
    int h = 512;  
    int *mat = malloc(sizeof(int) * w * h);  
    // iremos considerar uma linha colocada atrás da outra  
    mat[i * w + j] // acesso ao elemento [i][j].  
}
```

$FOR(i=0; i<3; i++)\{$   
 $FOR(j=0; j<4; j++)\{$   
 $VEC[i*w + j]$

$w$		
0	1	2
3	4	5
6	7	8
9	10	11

$FREE(MAT)$

$i=1 \quad j*3+2=$   
 $j=2$   
 $0*3+2=2$

- Tamanho definido em tempo de execução pelas variáveis  $w$  e  $h$
- Representadas por um apontador para o começo do bloco alocado
- Permanece alocado até ser liberado com `free`

# Alocação dinâmica

200  
300

- Vantagens
  - Controle feito em tempo de execução
  - Economia de memória
  - Expandir / diminuir / liberar conforme necessário
- Desvantagens
  - Riscos da gerência
    - Liberar espaços não mais necessários
    - Não acessar espaços já liberados
    - Acessar apenas a quantidade requisitada
    - Etc.

# Exemplo

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

```
void foo(int n) {
    int i, *p;
```

ALOCA();

LOAD();

```
/* Allocate a block of n ints */
```

```
p = (int *) malloc(n * sizeof(int));
```

```
if (p == NULL) {
    perror("malloc");
    exit(0);
}
```

PROC();

RESULT();

LIBERAR();

FREE()

```
/* Initialize allocated block */
```

```
for (i = 0; i < n; i++) {
    p[i] = i;
}
```

```
/* Return allocated block to the heap */
```

```
free(p);
```

```
}
```



# Atividade prática

## **Exercícios básicos (30 minutos)**

1. Analisar manualmente programas buscando por erros de memória

# Atividade prática

## Verificação de programas (30 minutos)

1. Usar ferramentas de checagem de integridade de memória

# Ler memória não-inicializada

Bug clássico: assumir que dados no heap são pré-inicializados com zero

```
/* return y = Ax */
int *matvec(int **A, int *x) {
    int *y = malloc(N*sizeof(int));
    int i, j;

    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            y[i] += A[i][j]*x[j];
    return y;
}
```

# Sobrescrever memória

Bug clássico: off-by-one!

```
int **p;  
  
p = malloc(N*sizeof(int *));  
  
for (i=0; i<=N; i++) {  
    p[i] = malloc(M*sizeof(int));  
}
```

# Memory leaks

```
foo() {  
    int *x = malloc(N*sizeof(int));  
    ...  
    return;  
}
```

C++ tem uma boa solução para esse problema: smart pointers!



# Outras funções

**calloc:** Versão de malloc que inicializa bloco alocado com zeros.

**realloc:** “Re-aloca” um bloco – muda o tamanho do bloco garantindo a integridade dos dados. Note que o bloco realocado pode mudar de lugar na memória!

**sbrk:** usado internamente pelos alocadores para aumentar ou diminuir o heap

# Atividade prática

## Implementação de programas que alocam memória

1. Revisão de strings
2. Implementando programas que alocam memória

# Insper

[www.insper.edu.br](http://www.insper.edu.br)