Insper

Sistemas Hardware-Software

Aula 10 – Alocação de memória

Engenharia
Fabio Lubacheski
Maciel C. Vidal
Igor Montagner
Fábio Ayres

Alocação de memória

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

#define MAXW 512
#define MAXH 512

int main(int argc, char *argv[]) {
    int mat[MAXH][MAXW];

    /*
        trabalhar com arquivo PGM
    */

    return 0;
}
```

Alocação de memória

- Qual o consumo de memória do programa anterior?
- E se precisarmos de matrizes maiores ?
- Ou se precisarmos definir o tamanho da matriz só em tempo de execução ?
- Onde é alocada a matriz ?

Alocação estática

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

#define MAXW 512
#define MAXH 512

int main(int argc
    int mat[MAXH]

    /*
        trabalhar com arquivo PGM
    */

    return 0;
}
```

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

Alocação estática

Existe problema no código abaixo ? – **ex1_slide.c**

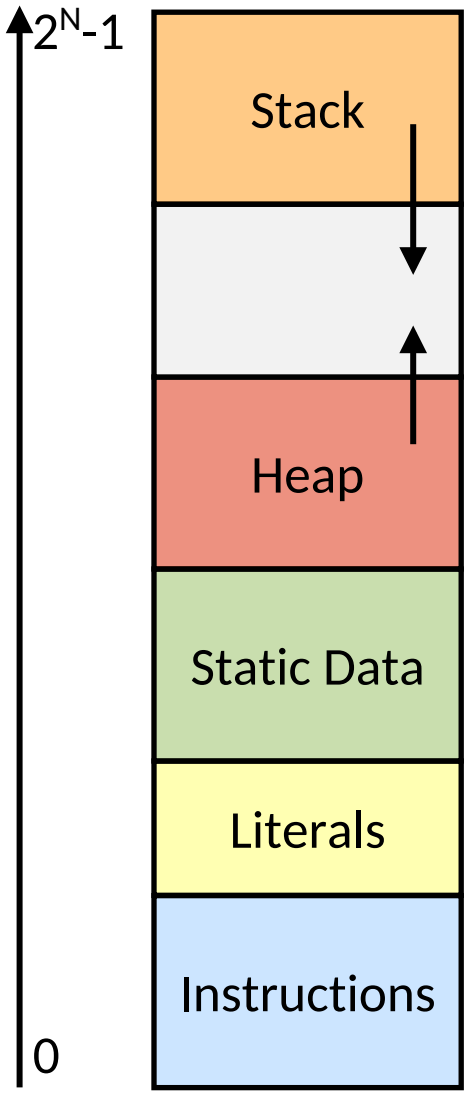
Como corrigir o problema ?

```c
char* retorna_str() {
  char str_vetor[32];
  str_vetor[0]='a';
  str_vetor[1]='b';
  str_vetor[2]='c';
  str_vetor[3]='d';
  str_vetor[4]='\x0';
  return str_vetor;
}
```

```c
int main(){
    char *resp;
    resp = retorna_str();
    printf("%s",resp);
    return 0;
}
```

Executável na memória

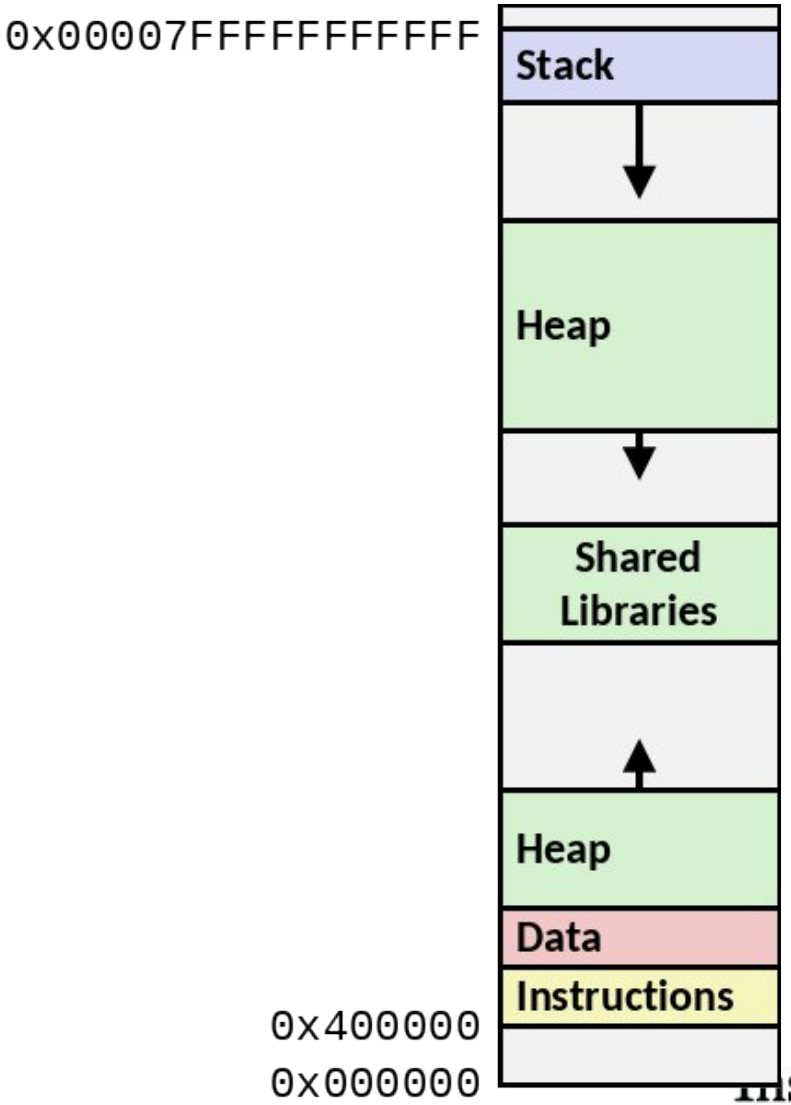
- **Stack** (tempo de compilação)
  - Variáveis locais
- Alocação estática (tempo de compilação)
  - **Static Data**: variáveis globais
  - **Literals**: contantes string
- **Instructions**
  - Instruções de máquina
- **Heap** (tempo de execução)
  - Alocação dinâmica



Alocação dinâmica de memória

- Programas usam **alocadores de memória dinâmica** para criar e gerenciar novos espaços de memória em **tempo de execução**.
  - C: malloc, free
  - C++: new, delete

- A área do espaço de memória 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)

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)

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

Alocação dinâmica

```c
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].
```

- 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

- 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

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

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

  /* Allocate a block of n ints */
  p = (int *) malloc(n * sizeof(int));
  if (p == NULL) {
    perror("malloc");
    exit(0);
  }

  /* Initialize allocated block */
  for (i = 0; i < n; i++) {
    p[i] = i;
  }

  /* Return allocated block to the heap */
  free(p);
}
```

Para onde vai cada parte no código ?

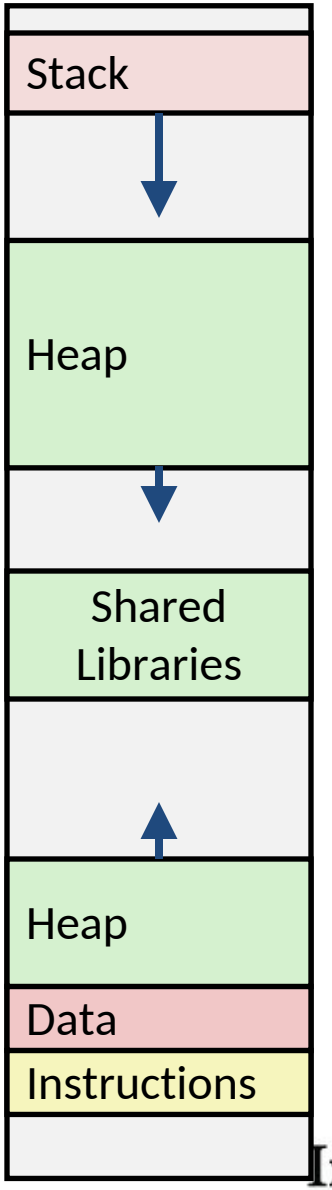
```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
    /* Some print statements ... */
}
```

ex2_slide.c



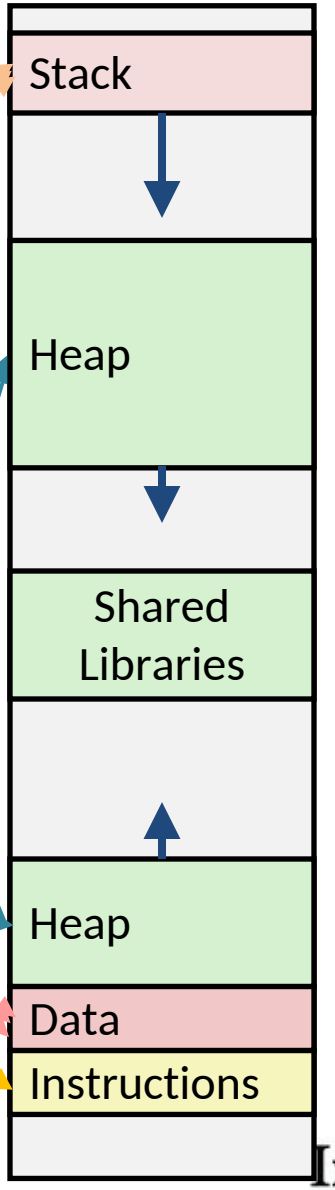
Para onde vai cada parte no código ?

```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
    /* Some print statements ... */
}
```



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