

Estrutura de Dados I

Hoje: Alocação dinâmica de memória

Prof. Dr. Rafael P. Torchelsen
rafael.torchelsen@inf.ufpel.edu.br

Alocação Dinâmica de Memória



- **Motivação:**
 - Nossos programas pré-fixavam o número de variáveis a serem utilizadas.
 - No caso de vetores e matrizes o tamanho era fixado como sendo um limitante superior previsto (constante).
 - **Desperdício de memória.**
 - Variáveis locais são armazenadas em uma parte da memória chamada pilha (ou stack), com espaço limitado.
 - **Ex: Insuficiente para armazenar vetores grandes.**
 - Memória alocada na pilha é armazenada seguindo o conceito de primeiro a entrar último a sair, ou seja a primeira alocação nessa região é a última a ser liberada. Por exemplo, a primeira variável alocada vai ficar na memória até que todas as outras armazenadas na pilha sejam liberadas
 - Existe um projeto de aprimoramento da linguagem C para alocar vetores e matrizes com tamanho dinâmico, porém as variáveis são colocadas na pilha, com todas as vantagens e desvantagens disso
 - <http://en.wikipedia.org/wiki/C99>

Alocação Dinâmica de Memória



- **Motivação:**

- Existe uma parte da memória para alocação dinâmica chamada heap que contém toda memória disponível, não reservada para outras finalidades.
- C permite alocar (**reservar**) espaço de memória de tamanho arbitrário no heap em tempo de execução tornando os programas mais flexíveis.
 - Liberar o espaço utilizado por uma variável não depende da liberação de outra, como na stack (pilha de memória)
- O acesso a memória alocada é realizada por meio de ponteiros.

Vamos simplificar por agora!

- No próximo exemplo vamos considerar que só existe uma memória e que tudo é alocado lá.
 - Na verdade é assim mesmo fisicamente, mas não logicamente! Temos que trabalhar com ambos em mente se quisermos fazer tudo da melhor maneira possível.
 - Quando chegarem em Sistemas Operacionais entenderam como a memória é gerenciada entre diferentes tipos, programas, níveis de segurança, etc.
 - Vamos usar o modelo simplificado agora para facilitar o entendimento.

Alocação Estática de Memória

Memória Principal

End. Valor

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

```
int main(){  
  
    int notaAluno[3];  
  
    ...  
}
```

A memória foi alocada antes mesmo do primeiro comando dentro do main ser executado

Essa região de memória fica reservada durante toda a execução do programa, mesmo que a variável notaAluno nunca seja utilizada

E se o número de alunos mudar durante a execução do programa? Só recompilado.

Alocação Dinâmica de Memória

Memória Principal

End.	Valor
0	
1	
2	
3	
4	
5	
6	
7	
...	...

Não é possível modificar

```
int main(){
```

```
    int notaAluno[3];
```

```
    ...
```

```
    //numero de alunos mudou
```

```
    int notaAluno[4]; // ?
```

```
}
```

Como modificar o tamanho de um vetor?

Gostaríamos que fosse assim

```
int nNumAlunos;  
scanf("%d",&nNumAlunos);  
int notaAluno[nNumAlunos];
```

Alocação Dinâmica de Memória

Memória Principal

End. Valor

0

1

2

3

4

5

6

7

...

...

```
int main(){  
  
    int nNumAlunos;  
    scanf("%d",&nNumAlunos);  
  
    int *notaAlunos;  
    ...  
}
```

Até agora não temos
onde colocar as notas,
não temos um
endereço para
notaAlunos

Alocação Dinâm

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

Memória Principal

End. Valor

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

```
int main(){  
  
    int nNumAlunos;  
    scanf("%d",&nNumAlunos);  
  
    int *notaAlunos;  
    notaAlunos = malloc(sizeof(int)*nNumAlunos);  
    ...  
}
```

Já vimos como utilizar um ponteiro que aponta para um vetor e através do ponteiro modificar o conteúdo do vetor. Isso é possível com notaAlunos

notaAlunos recebe um endereço de memória, como um ponteiro normal. Porém, esse endereço não pertence a outra variável

O endereço é referente a um espaço de memória que acabou de ser reservado

Alocação Dinâmica de Memória

Memória Principal

End. Valor

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

```
int main(){  
  
    int nNumAlunos;  
    scanf("%d",&nNumAlunos);  
  
    int *notaAlunos;  
    notaAlunos = malloc(sizeof(int)*nNumAlunos);  
    ...  
    nNumAlunos = 4;  
    notaAlunos = malloc(sizeof(int)*nNumAlunos);  
}
```

Chamar o malloc novamente reserva um outro espaço de memória

Ao colocar o novo endereço no ponteiro utilizado anteriormente, como podemos acessar o espaço de memória anterior?

Não podemos! Nunca devemos perder o endereço de memória!

Liberando

```
void free(void *pointer);
```

Memória Principal

End. Valor

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

```
int main(){  
  
    int nNumAlunos;  
    scanf("%d",&nNumAlunos);  
  
    int *notaAlunos;  
    notaAlunos = malloc(sizeof(int)*nNumAlunos);  
    free(notaAlunos);  
    ...  
    nNumAlunos = 4;  
    notaAlunos = malloc(sizeof(int)*nNumAlunos);  
}
```

Como somos nós
que alocamos a
memória, somos
nós que devemos
libera-la!

Caso a memória não seja
liberada ela fica
indisponível não somente
durante a execução do
programa, mas após o
seu fechamento,
chamamos isso de
memory leak ou
vazamento de memória

Podemos liberar memória utilizando a função
free, mas somente memória alocada
dinamicamente!

Falta de Memória

- Quando malloc retorna o endereço NULL é provável que a memória disponível não seja suficiente
- Sempre devemos testar se a função malloc retornou NULL

```
int *teste = malloc (sizeof(int));  
if (!teste) {  
    printf("Erro! Falta de memoria");  
    return -1;  
}
```

Ou seja: teste == NULL

Casting

```
int main(){  
  
    int nNumAlunos;  
    scanf("%d",&nNumAlunos);  
  
    int *notaAlunos;  
    notaAlunos = (int *)malloc(sizeof(int)*nNumAlunos);  
    free(notaAlunos);  
}
```

Devemos sempre criar um ponteiro do tipo de dado que vamos armazenar na memória alocada, e devemos sempre fazer um casting para o tipo de dado.

A função malloc aloca um espaço de memória e retorna um endereço de memória sem tipo.
void

Um ponteiro sem tipo não permite avanço e retrocesso na memória em intervalos iguais ao tipo de dado armazenado, pois não sabemos quanto pular já que não sabemos o tipo de dado.

Structs

```
typedef struct{  
    char nome[30];  
    int idade;  
    int altura;  
}Pessoa;
```

```
int main(){
```

```
    Pessoa *p;  
    p= (Pessoa *)malloc(sizeof(Pessoa));  
    // le os dados de uma pessoa  
    printf("Nome: "); scanf("%s",&(*p).nome);  
    printf("Idade: "); scanf("%d",&(*p).idade);  
    printf("Nome: "); scanf("%d",&(*p).altura);  
    // imprime os dados na tela  
    printf("Nome: %s\n", (*p).nome);  
    printf("Idade: %d\n", (*p).idade);  
    printf("Nome: %d\n", (*p).altura);  
    free(p);  
}
```

Somente uma pessoa

Acesso ao dados na struct

Outra forma:
p->nome

Não é necessário o *
pois o -> só funciona
em ponteiros.


O parênteses é necessário porque o • (ponto) tem
prioridade maior que o*.

Structs

```
typedef struct{
    char nome[30];
    int idade;
    int altura;
}Pessoa;

int main(){

    Pessoa *p;
    p= (Pessoa *)malloc(sizeof(Pessoa));
    // le os dados de uma pessoa
    printf("Nome: "); scanf("%s",&(*p).nome);
    printf("Idade: "); scanf("%d",&(*p).idade);
    printf("Nome: "); scanf("%d",&(*p).altura);
    // imprime os dados na tela
    printf("Nome: %s\n", (*p).nome);
    printf("Idade: %d\n", (*p).idade);
    printf("Nome: %d\n", (*p).altura);
    free(p);
}
```



```
scanf("%s",&p->nome);
scanf("%d",&p->idade);
scanf("%d",&p->altura);
```

Structs

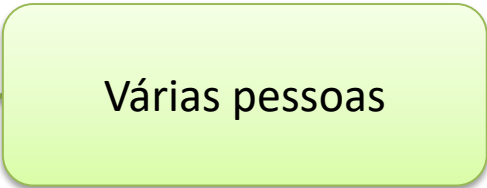
```
typedef struct{  
    char nome[30];  
    int idade;  
    int altura;  
}Pessoa;
```

```
int main(){
```

```
    int nNumPessoas;  
    printf("Quantas pessoas?");  
    scanf("%d", nNumPessoas);
```

```
    Pessoa *p;  
    p= (Pessoa *)malloc(sizeof(Pessoa) * nNumPessoas);
```

```
    ...
```



Várias pessoas

String

- Com alocação dinâmica podemos criar uma variável char com o tamanho exato do texto, não temos mais desperdício

`char nome[50];`

← E se o nome for Ana ?

`char *nome = (char *)malloc(sizeof(char) * nTamanhoDoNome);`

Alocação Dinâmica de Matrizes

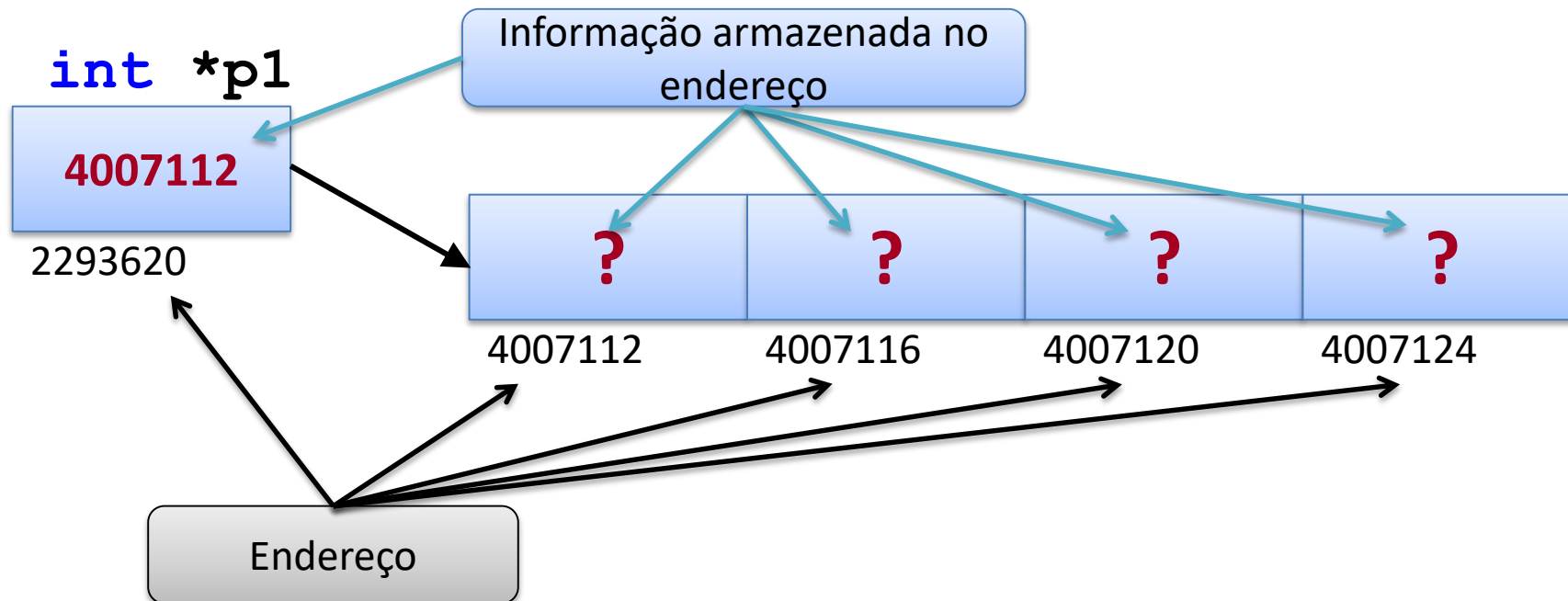
- **Introdução:**

- Vimos que para alocar um vetor de forma dinâmica usamos a função **malloc** que retorna o endereço do primeiro elemento do vetor alocado.
- Precisamos de um ponteiro para guardar o endereço retornado.
- Os elementos do vetor podem ser acessados através do ponteiro usando notação convencional de vetores **p[i]** ou através da notação de ponteiros ***(p+i)**.

Alocação Dinâmica de Matrizes

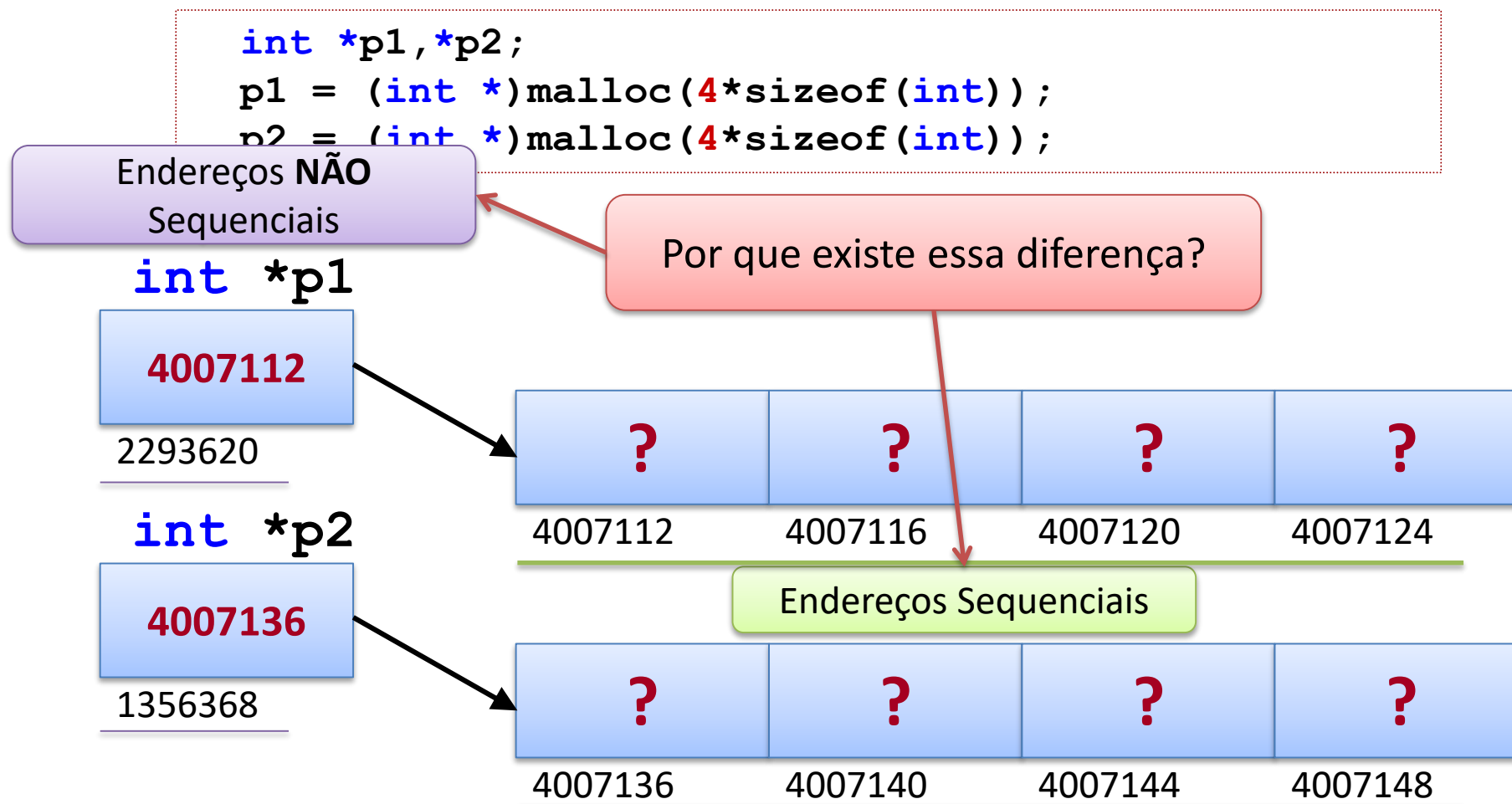
- Exemplo: Alocando um vetor de inteiros

```
int *p1;  
p1 = (int *)malloc(4*sizeof(int));
```



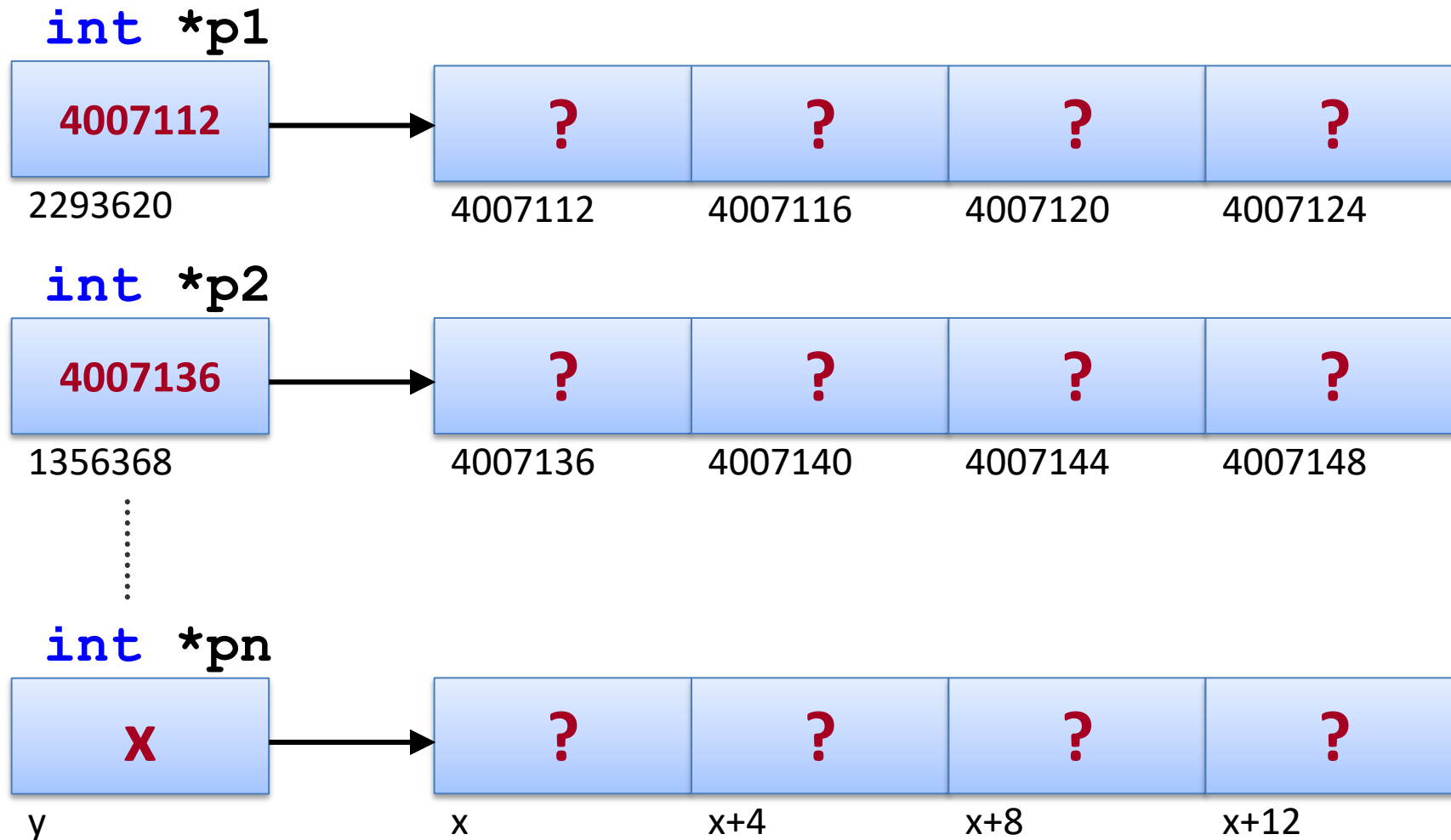
Alocação Dinâmica de Matrizes

- Exemplo: Alocando dois vetores de inteiros



Alocação Dinâmica de Matrizes

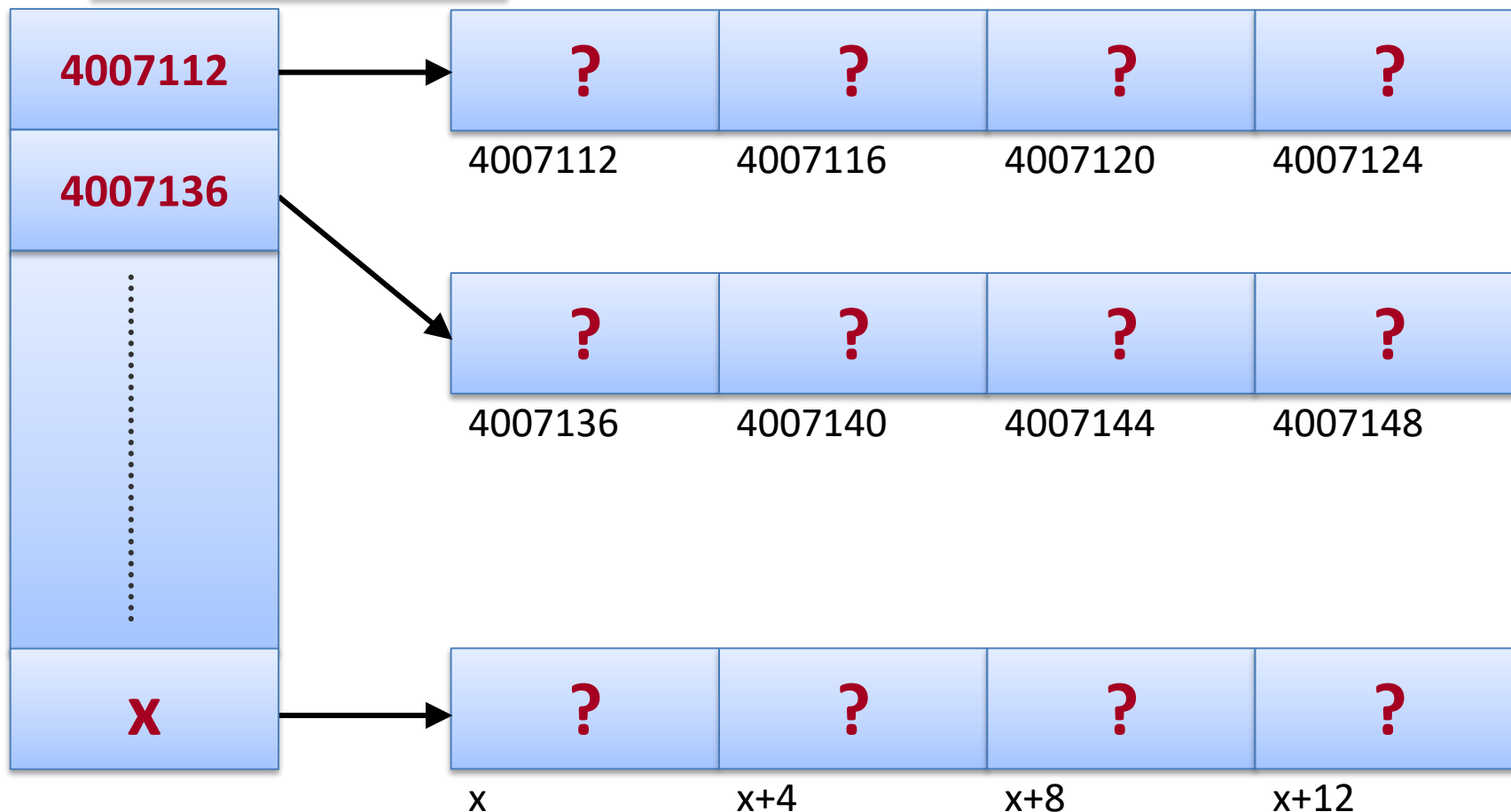
- Exemplo: Alocando N vetores de inteiros



Alocação Dinâmica de Matrizes

- N vetores = vetor de vetores

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



Alocação Dinâmica de Matrizes



- **Alocação de memória para matrizes:**
 - Matriz é um caso particular de vetor, onde os elementos são vetores (**vetor de vetores**).
 - Devemos portanto alocar um vetor de apontadores e depois um vetor de elementos para cada linha.
 - Para alocar um vetor de apontadores dinamicamente é necessário um apontador para apontadores (**ponteiro duplo**).
- **Desalocar a memória da matriz:**
 - Para desalocar devemos chamar **free** para cada linha e também para o vetor de apontadores.

Alocação Dinâmica de Matrizes

- Alocação de memória para matrizes:

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

int main() {
    int **M;
    int i, ncols=5, nrows=6;

    M = (int **)malloc(nrows*sizeof(int *));
    for(i=0; i<nrows; i++)
        M[i] = (int *)malloc(ncols*sizeof(int));

    //Agora podemos acessar M[i][j]:
    // Matriz M na linha i, coluna j.
    ...
    return 0;
}
```

Alocação Dinâmica de Matrizes

- Desalocar a memória da matriz:

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

int main() {
    int **M;
    int i, ncols=5, nrows=6;

    ...

    //Desaloca memória.
    for(i=0; i<nrows; i++)
        free(M[i]);
    free(M);

    return 0;
}
```


Outras funções: calloc

- **void *calloc(size_t num, size_t size);**
- Reserva espaço na memória para um vetor de num itens do programa. Cada item tem tamanho size e todos os bits do espaço são inicializados com 0. A função retorna um ponteiro de tipo void para o espaço reservado ou NULL no caso de algum erro ocorrer.

Outras funções: realloc

- **void *realloc(void *pont, size_t size);**
- A função altera o tamanho do objeto na memória apontado por pont para o tamanho especificado por size. O conteúdo do objeto será mantido até um tamanho igual ao menor dos dois tamanhos, novo e antigo. Se o novo tamanho requerer movimento, o espaço reservado anteriormente é liberado. Caso o novo tamanho for maior, o conteúdo da porção de memória reservada a mais ficará com um valor sem especificação.
- Se o tamanho size for igual a 0 e pont não é um ponteiro nulo o objeto previamente reservado é liberado.

Exercícios

- Faça a implementação da função realloc, você deve utilizar a função malloc para isso

`void *realloc(void *pont, size_t newSize);`

Dica: `void * memcpy (void * destination, const void * source, size_t num);`

Exercícios

```
typedef struct{
    char nome[30];
    int idade;
    int altura;
}Pessoa;
```

Imputação
Universidade Federal de Pelotas

- Faça um programa que armazene a informação de várias pessoas.
- O programa só deve sair quando o usuário disser que não deseja mais entrar com os dados de outra pessoa.
- Antes de sair o programa deve apresentar, de forma organizada, os dados de todas as pessoas.

Flags de compilação

- -Wall
 - Todos os warnings

Exercício

Implementar em C um programa que utilize uma matriz com vetor de ponteiros e que ofereça as seguintes opções para o usuário:

- 1) Criar e redimensionar uma matriz $m \times n$, onde n e m são fornecidos pelo usuário;
- 2) Realizar a leitura dos elementos da matriz;
- 3) Fornecer a soma dos elementos da matriz;
- 4) Retornar em um vetor (utilizando ponteiros) os elementos de uma determinada coluna da matriz;
- 5) Imprimir a matriz
- 6) Sair do programa

Observações:

- 1) A matriz deve ser alocada dinamicamente no programa por meio do uso da função malloc.
- 2) O programa deve ser modularizado e utilizar os seguintes protótipos de subalgoritmos:
 - a. `int ** criaMatriz(int m, int n)`
 - b. `void leiaMatriz(int **mat, int m, int n)`
 - c. `int somaMatriz(int **mat, int m, int n) { }`
 - d. `int* colunaMatriz(int ** mat, int m, int n, int ncoluna)`
 - e. `void liberaMatriz(int **mat, int ncoluna)`
 - f. `void imprimeMatriz(int **mat, int m, int n)`
 - g. `void imprimeVetor (int *vet, int n) { }`
- 3) O subalgoritmo `int* colunaMatriz(int ** mat, int m, int n, int ncoluna)` deve criar um novo vetor (ponteiro para vetor) e retornar o mesmo para o programa principal que será responsável pela impressão dos valores a partir da chamada de **`void imprimeVet (int *vet, int n) { }`**

Cuidado com os memory leaks!

Links

- https://www.youtube.com/results?search_query=stack+heap+c
- https://www.youtube.com/results?search_query=malloc+c
- <https://programacaodescomplicada.wordpress.com/>
 - <https://www.youtube.com/user/progdescomplicada/playlists>
 - Lembrem desses links durante todo o semestre!
 - Procurem na playlist e no menu do site os tópicos de interesse.