

# Alocação Dinâmica & Listas

Prof. Denio Duarte

[duarte@uffs.edu.br](mailto:duarte@uffs.edu.br)

Prof.

# Alocação Estática

```
int main()
{
    int a;
    float b;
    int v[3];
    a=10;
    b=3.1416;
    v[0]=3;v[1]=4;v[2]=5;
    :
    return 0;
}
```

Memória RAM

?			
?			
?	?	?	

# Alocação Estática

```
int main()
{
    int a;
    float b;
    int v[3];
    a=10;
    b=3.1416;
    v[0]=3;v[1]=4;v[2]=5;
    :
    return 0;
}
```

Memória RAM

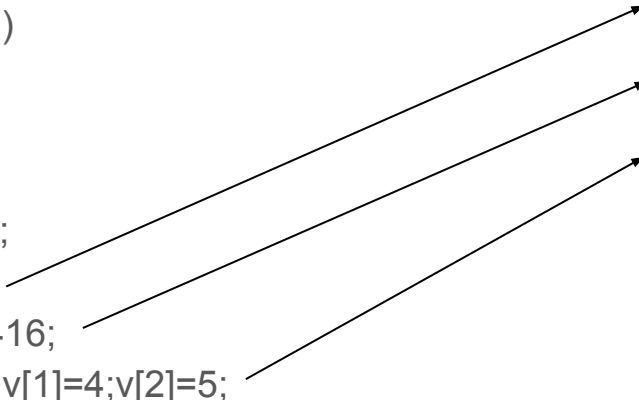
10			
3.1416			
3	4	5	

Os espaços para as variáveis  
são alocados durante o carga do  
programa para a memória

# Alocação Estática

```
int main()
{
    int a;
    float b;
    int v[3];
    a=10;
    b=3.1416;
    v[0]=3;v[1]=4;v[2]=5;
    :
    return 0;
}
```

Memória RAM



10			
3.1416			
3	4	5	

**Problema:** se tivermos que carregar muitos valores para a memória mas não sabemos o máximo?

# Alocação Estática

```
int main()
{
    int a;
    float b;
    int v[3000];
    a=10;
    b=3.1416;
    v[0]=3;v[1]=4;v[2]=5;
    :
    return 0;
}
```

Memória RAM

10					
3.1416					
3	4	5	?	?	...

**Problema:** se tivermos que carregar muitos valores para a memória mas não sabemos o máximo?

1. Desperdício de memória
2. Falta de memória caso existam mais valores do que o planejado

# Antes de continuarmos ...

- Recapitulação ponteiros

int a;

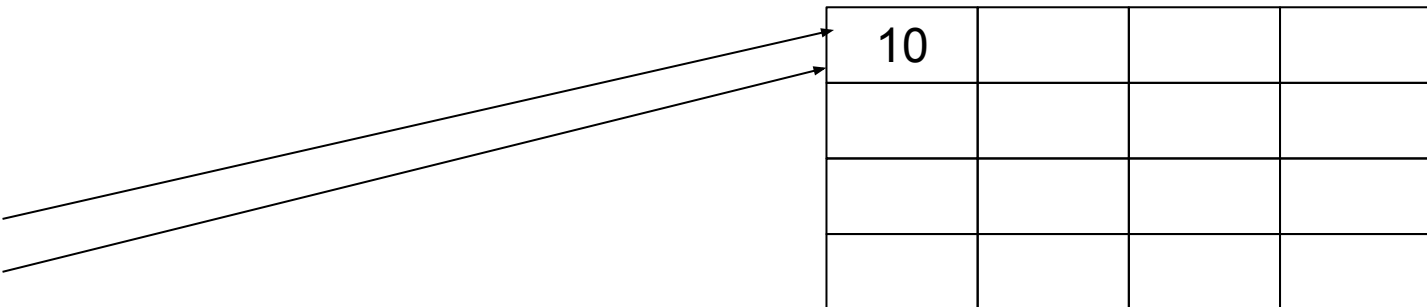
int \*p;

?			

# Antes de continuarmos ...

- Recapitulação ponteiros

```
int a;  
int *p;  
a=10;  
p=&a;
```

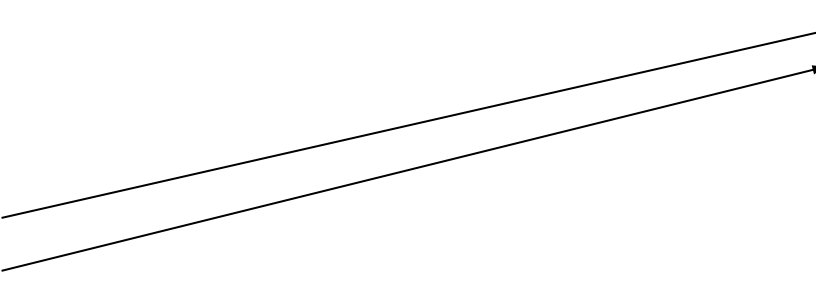


10			

# Antes de continuarmos ...

- Recapitulação ponteiros

```
int a;  
int *p;  
a=10;  
p=&a;  
*p=20;
```



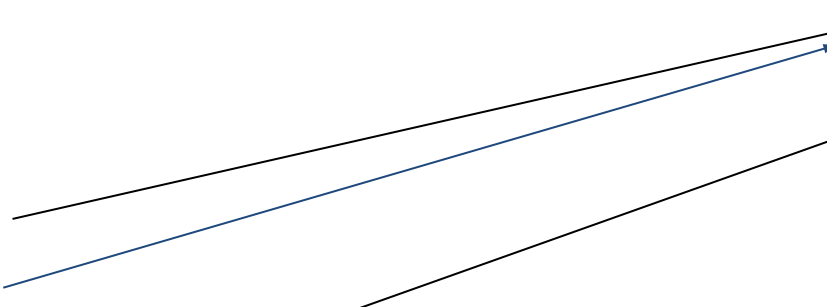
20			



# Antes de continuarmos ...

- Recapitulação ponteiros

```
int a;  
int *p;  
a=10;  
p=&a;  
*p=20;  
int b=11;
```



The diagram illustrates the state of memory after the provided code. It features a 4x4 grid of cells. The first row's first cell contains the value 20, and the second row's first cell contains the value 11. Three arrows originate from the code on the left: a black arrow from 'a=10;' points to the cell containing 20; a blue arrow from 'p=&a;' points to the cell containing 11; and a black arrow from 'int b=11;' points to the first cell of the third row. All other cells in the grid are empty.

20			
11			

# Antes de continuarmos ...

- Recapitulação ponteiros

```
int a;
```

```
int *p;
```

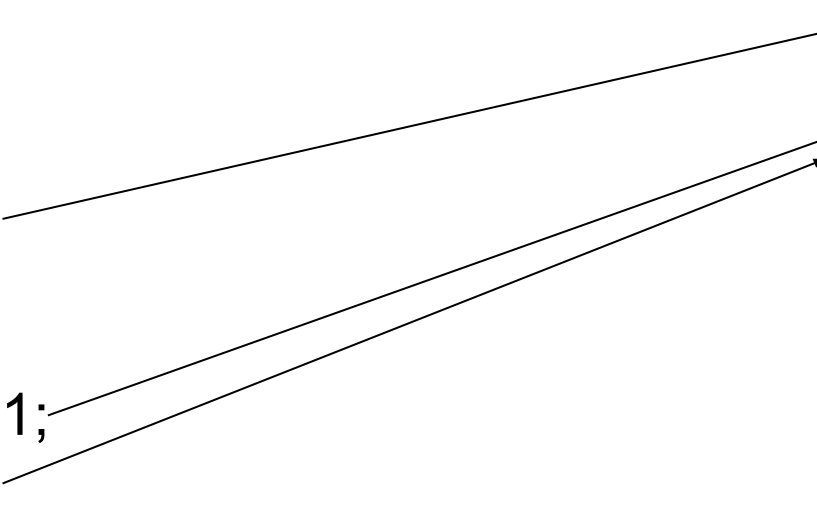
```
a=10;
```

```
p=&a;
```

```
*p=20;
```

```
int b=11;
```

```
p=&b;
```




The diagram illustrates the state of memory after the provided code. It consists of a 4x4 grid of cells. The first row has the value '20' in the first column. The second row has the value '11' in the first column. The remaining cells are empty. Three arrows originate from the code on the left: one from 'a=10;' pointing to the first cell of the first row, one from '\*p=20;' pointing to the first cell of the second row, and one from 'p=&b;' pointing to the first cell of the third row. This indicates that the value 10 was stored at the address of 'a', but was then overwritten by 20 through the pointer 'p'. The value 11 is stored at the address of 'b'.

20			
11			

## Antes de continuarmos ...

- Lembrando: vetores são ponteiros disfarçados :)

```
int v[4]={2,4,6,8};
```



2	4	6	8

## Antes de continuarmos ...

- Lembrando: vetores são ponteiros disfarçados :)

```
int v[4]={2,4,6,8};
```

```
*v=10;
```

```
*(v+1)=12;
```

2	4	6	8

10	12	6	8

## Antes de continuarmos ...

- Lembrando: vetores são ponteiros disfarçados :)
  - Por isso não se coloca & no scanf para strings (vetores de caracteres):

```
char st[10];  
scanf("%s",st); // e não scanf("%s",&st)
```

# Alocação Dinâmica

- Os espaços para as variáveis são alocados **durante a execução do programa**.
  - O espaço deve ser alocado e uma variável deve apontar para ele (**variável ponteiro**)
    - A variável que aponta (o ponteiro) e o espaço alocado devem ser do mesmo tipo
  - A biblioteca **stdlib.h** deve ser incluída para usarmos a função de alocação de memória

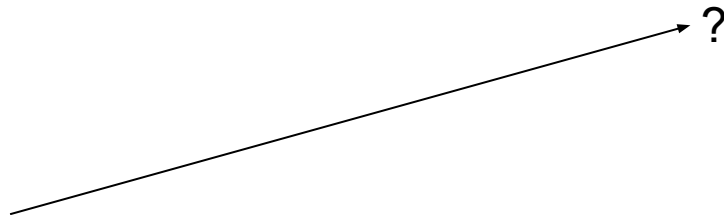
# Alocação Dinâmica

- Como utilizar?
  - Criar uma variável com o tipo desejado colocando \* antes do nome da variável: `int *p;`
  - Para armazenar algo no local que a variável está apontando: `*p=10;` // só podemos armazenar um valor se p apontar para um local válido
  - Para alterar o endereço que a variável aponta faz-se: `p=<novo endereço> => p=&a;` // & indica endereço de algo
  - Para alocar um espaço de memória e retornar o endereço do local: `malloc(qtdade de bytes) => p=(int *)malloc(sizeof(int));`

# Alocação Dinâmica

- Exemplo:

⋮  
int \*p;






# Alocação Dinâmica

- Exemplo:

```
:  
int *p=NULL;
```

Lembrando:

- Para inicializar um ponteiro para um lugar “seguro”, usamos o valor **NULL**

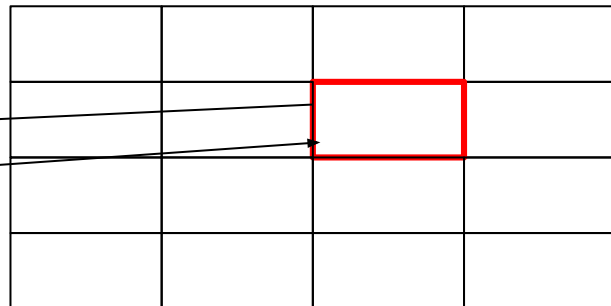

# Alocação Dinâmica

- Exemplo:

:

```
int *p;
```

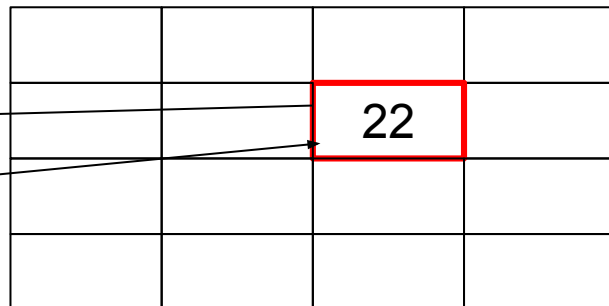
```
p=(int *)malloc(sizeof(int));
```



# Alocação Dinâmica

- Exemplo:

```
:  
int *p;  
p=(int *)malloc(sizeof(int));  
*p=22;  
printf("Conteúdo do p: %d",*p);  
printf("Local de apontamento: %p",p);  
// interessante, veja como fica o scanf  
scanf("%d",p); // e não scanf("%d",&p)
```



		22	

# Listas

1. Até agora, vimos uma estrutura de dados: **vetores**
2. Propriedades importantes de um **vetor**:
  - a. São uma área de memória contígua
3. Em aula anterior vimos uma segunda estrutura: **structs**
4. Propriedades importantes de uma **struct**:
  - a. Os elementos (**membros**) de uma **struct** podem ser de tipos diferentes
  - b. Para selecionar um elemento de uma **struct**, especificamos o nome do elemento

# Relembrando Struct

1. Para declarar variáveis que são **structs**, podemos escrever

```
struct date {  
    int day;  
    int month;  
    int year;  
};  
struct date date1, date2;
```

```
struct employee{  
    int id;  
    char name[TAM_NOME+1];  
    double income;  
};  
struct employee e1, e2;
```

2. Representação de **date1** na memória do computador:



3. Os nomes dos membros de uma **struct** não conflitam com outros nomes de fora da struct

# Relembrando Struct

## 1. Definindo um tipo com typedef

```
struct date {  
    int day;  
    int month;  
    int year;  
};  
typedef struct date Date;
```

```
typedef struct {  
    int id;  
    char name[TAM_NOME+1];  
    double income;  
} Employee;
```

```
Date dt1, dt2;  
Employee emp1, emp2;
```

# Lista

- Uma lista, de maneira geral, é um conjunto de elementos do mesmo tipo.
- Uma lista pode ser implementada de diversas formas.
- Podemos considerar isso como uma lista?

```
Employee emp[25];
```

```
typedef struct{  
    int id;  
    char name[TAM_NOME+1];  
    double income;  
} Employee;
```

# Lista

- Podemos considerar isso como uma lista?
  - Sim. Vetores criam listas estáticas
  - Já fizemos nos exercícios anteriores.
  - Quais os problemas que ela apresenta?

```
typedef struct{  
    int id;  
    char name[TAM_NOME+1];  
    double income;  
} Employee;
```

```
Employee emp[25];
```



# Lista

- Lista com vetor
  - Quais os problemas que ela apresenta?
    - Tamanho fixo da quantidade máxima de elementos
    - Possível fonte de desperdício de memória

```
typedef struct{  
    int id;  
    char name[TAM_NOME+1];  
    double income;  
} Employee;
```

```
Employee emp[25];
```

# Saudade do Python :)

- Cria-se uma **struct** (classe)
- Uma variável do tipo lista (**lista\_f**)
- Adicionava-se o elemento da classe na lista
- Repetia até o infinito ...

```
1 class func:
2     · id=0
3     · name= ''
4     · income=0.0
5     #
6     list_f=[]
7
8     f=func()
9     f.id=15
10    f.name='Fifteen'
11    f.income=15
12    list_f.append(f)
13
14    f=func()
15    f.id=16
16    f.name='Sixteen'
17    f.income=16
18    list_f.append(f)
19
20    #Imprimir a lista
21
22    for func in range(len(list_f)):
23        ··· print("Id: " + str(list_f[func].id)
24        ··· |····· + "\nFuncionário: " + list_f[func].name
25        ··· |····· + "\nSalário: " + str(list_f[func].income))
26        ··· print("\n")
```

# Back to reality

1. Implementando lista dinamicamente em C
2. Para declarar variáveis que são ponteiros para **structs**, podemos escrever

```
struct date {  
    int day;  
    int month;  
    int year;  
};  
typedef struct Date;
```

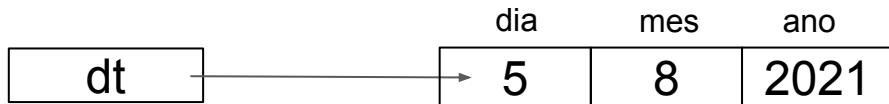
```
typedef struct employee{  
    int id;  
    char name[TAM_NOME+1];  
    double income;  
} Employee;
```

```
Date *dt;  
Employee *emp1, *emp2;  
  
dt = (Date *) malloc(sizeof(Date));  
dt->day=5 // não pode ser dt.day  
dt->month=8  
dt->year=2021
```



```
// Cuidado  
Date dte; //alocação estática de memória  
  
dte.day=5  
dte.month=8  
dte.year=2021
```

3. Representação de **dt** na memória do computador:



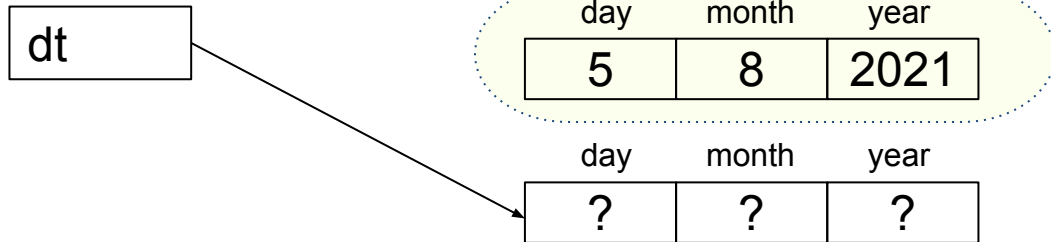
# Back to reality

```
struct date {  
    int day;  
    int month;  
    int year;  
};  
typedef struct Date;
```

```
typedef struct employee{  
    int id;  
    char name[TAM_NOME+1];  
    double income;  
} Employee;
```

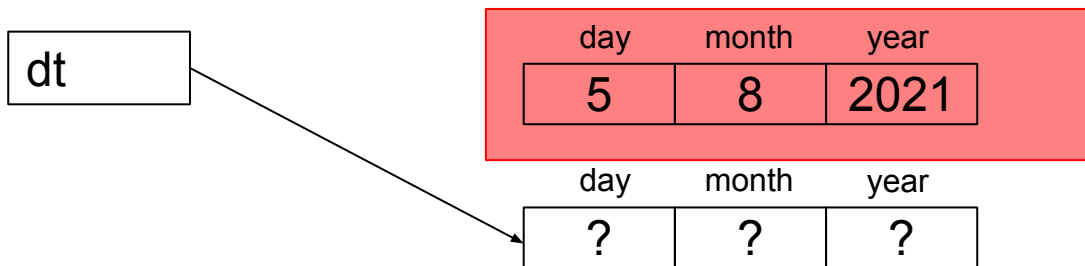
```
Date *dt;  
Employee *emp1, *emp2;  
  
dt = (Date *) malloc(sizeof(Date));  
dt->day=5 // não pode ser dt.day  
dt->month=8  
dt->year=2021  
dt = (Date *) malloc(sizeof(Date));
```

Ao alocarmos um novo espaço, a **localização** do anterior é **perdida**



# Back to reality

1. Quando alocamos um novo espaço na memória perdemos o endereço do anterior :(



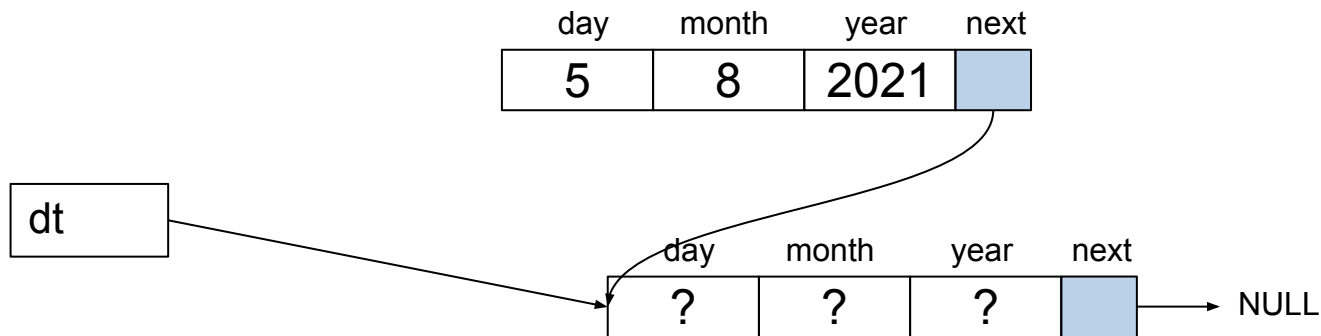
2. Em Python a lista era acessada com o índice: lista[0], lista[1], ...
3. Em C só daria certo se alocarmos tudo de uma vez só, mas cairíamos no problema de “o que é tudo?”
  - `dt = (Date *) malloc(sizeof(Date)*100);` // igual a `Date dt[100];`
  - Agora poderíamos fazer: `dt[0], ..., dt[99]`

# Lista encadeada

- Uma lista encadeada representa uma sequência de objetos, do mesmo tipo, na memória.
  - Os espaços alocados não estão, necessariamente, organizados de forma contígua (espaços de memória adjacentes)
- Cada elemento da sequência armazena seu valor e o endereço do próximo elemento
  - Ou seja, junto a cada um dos elementos da lista, explicitamente armazenamos o endereço para o próximo elemento da lista

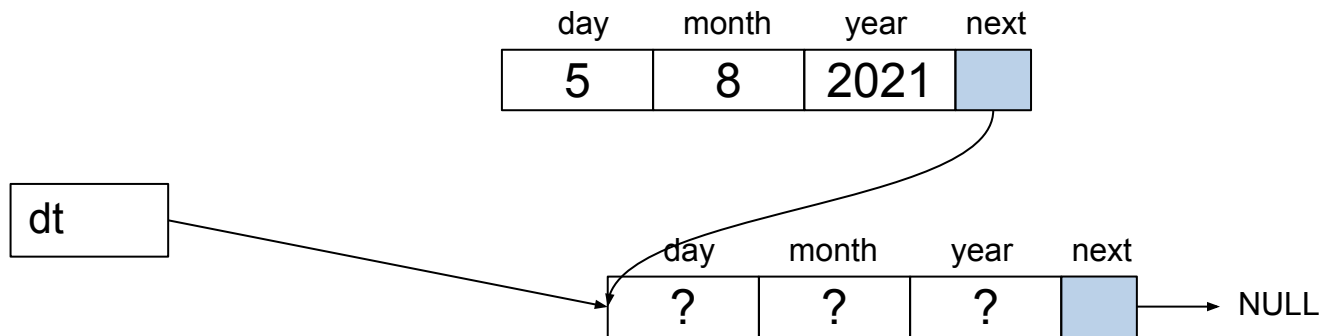
# Lista encadeada

- A lista é chamada de encadeada pois dentro do espaço alocado tem um ponteiro que aponta para o próximo
  - Isso se faz necessário para os espaços não ficarem perdidos



# Lista encadeada

- A lista é chamada de encadeada pois dentro do espaço alocado tem um ponteiro que aponta para o próximo
  - Isso se faz necessário para os espaços não ficarem perdidos

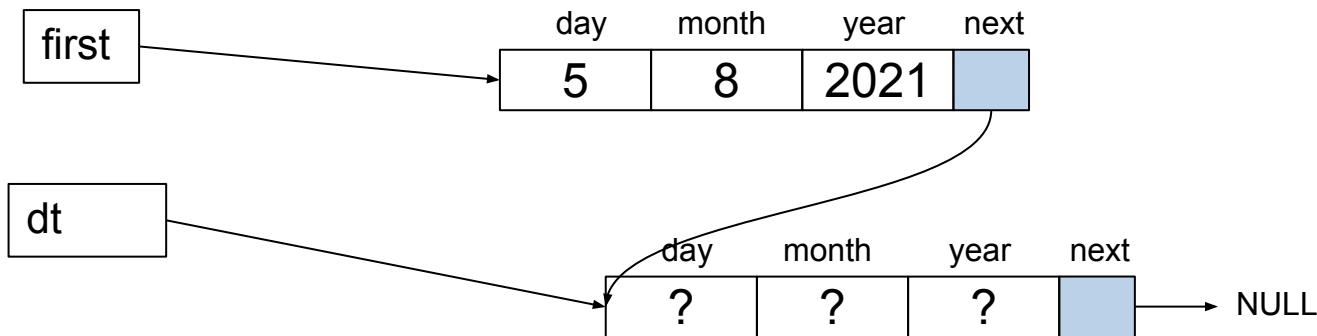


Ok. Agora conseguimos “caminhar”  
pelas regiões alocadas, mas como  
começar a caminhada?



# Lista encadeada

- A lista é chamada de encadeada pois dentro do espaço alocado tem um ponteiro que aponta para o próximo
  - Isso se faz necessário para os espaços não ficarem perdidos

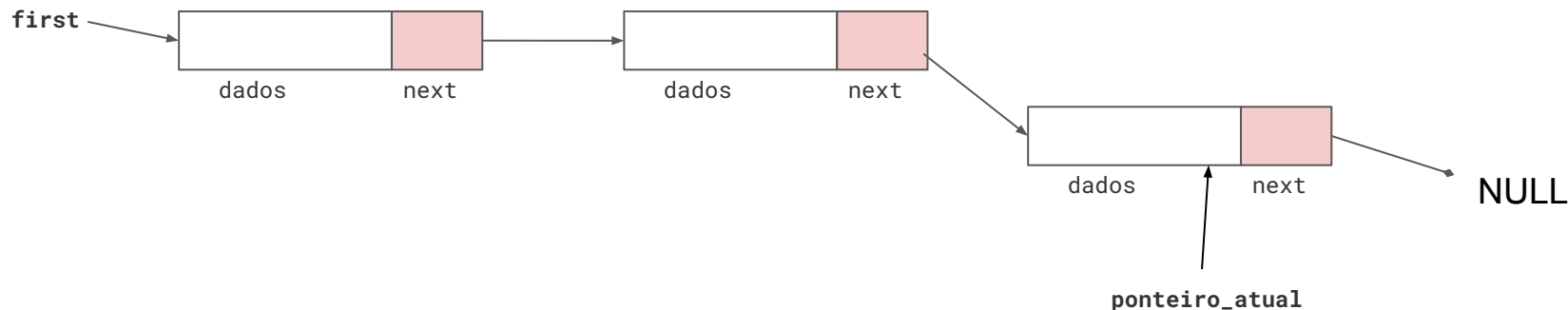


Ok. Agora conseguimos “caminhar”  
pelas regiões alocadas, mas como  
começar a caminhada?

Basta criar um ponteiro auxiliar que  
vai nos dizer quem é o primeiro :)

# Lista encadeada

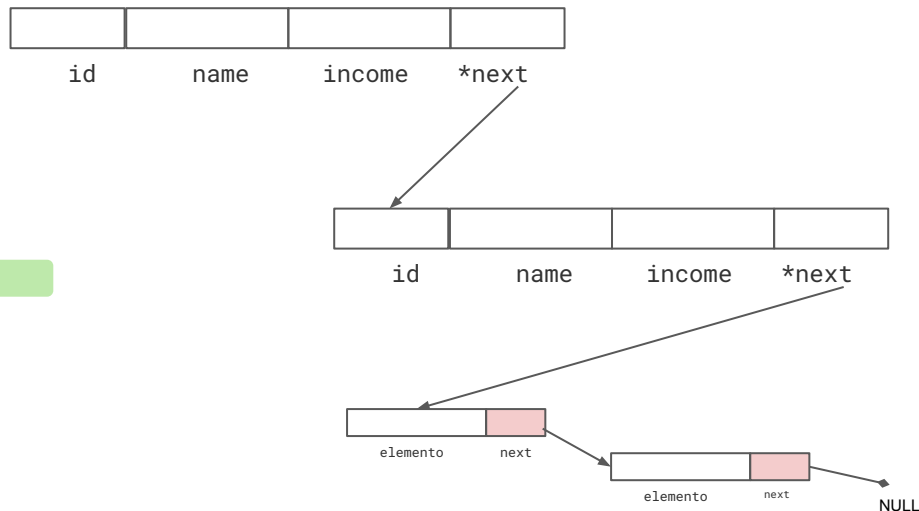
- A lista é chamada de encadeada pois dentro do espaço alocado tem um ponteiro que aponta para o próximo
  - Esquema



# Lista encadeada

- Uma lista encadeada representa uma sequência de objetos, do mesmo tipo, na memória. Cada elemento da sequência armazena seu valor e o endereço do próximo elemento
  - Ou seja, junto a cada um dos elementos da lista, explicitamente armazenamos o endereço para o próximo elemento da lista

```
struct employee{  
    int id;  
    char name[TAM_NOME+1];  
    double income;  
    struct employee *next;  
};  
typedef struct employee Employee;
```



# Lista encadeada

- Uma lista encadeada representa uma sequência de objetos, do mesmo tipo, na memória. Cada elemento da sequência armazena seu valor e o endereço do próximo elemento
- Os elementos de uma lista não ocupam uma área contígua de memória (como os vetores), o que não permite acesso direto aos elementos.
- Para acessar um elemento, é necessário que todos os elementos estejam encadeados.

```
struct employee{  
    int id;  
    char name[TAM_NOME+1];  
    double income;  
    struct employee *next;  
};  
typedef struct employee Employee;
```

# Lista encadeada

- Como criamos a lista

```
Employee *first; //head é legal também :)

first = (Employee *) malloc(sizeof(Employee));

first->id = 1;
strcpy(first->name, "Pafuncio");
first->income = 3000.0;
first->next = NULL;
```

```
struct employee{
    int id;
    char name[TAM_NOME+1];
    double income;
    struct employee *next;
}

typedef struct employee Employee;
```

OU

```
typedef struct employee{
    int id;
    char name[TAM_NOME+1];
    double income;
    struct employee *next;
} Employee;
```

# Memória

first

?


```
Employee *first; //head
```

```
first = (Employee *) malloc(sizeof(Employee));
```

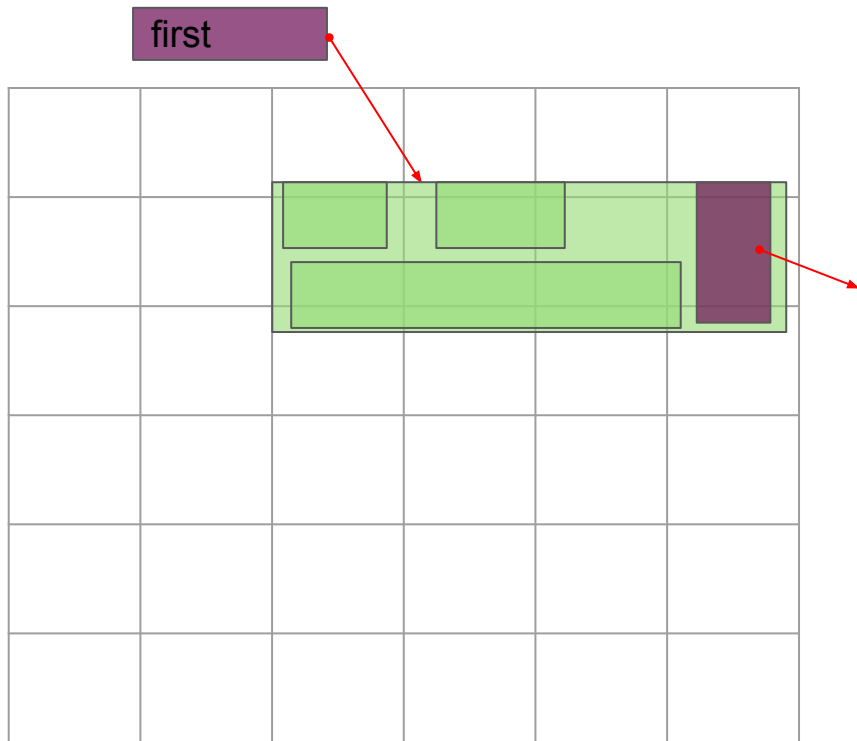
```
first->id = 1;
```

```
strcpy(first->name, "Pafuncio");
```

```
first->income = 3000.0;
```

```
first->next = NULL;
```

# Memória



```
Employee *first; //head
```

```
first = (Employee *) malloc(sizeof(Employee));
```

```
first->id = 1;
```

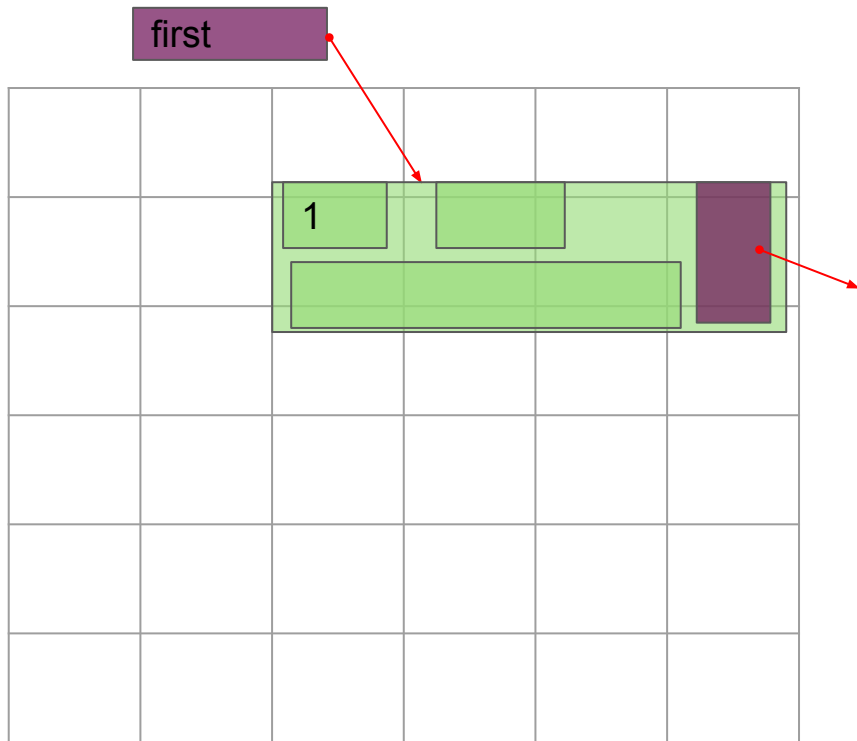
```
strcpy(first->name, "Pafuncio");
```

```
first->income = 3000.0;
```

```
first->next = NULL;
```



# Memória



```
Employee *first; //head
```

```
first = (Employee *) malloc(sizeof(Employee));
```

```
first->id = 1;
```

```
strcpy(first->name, "Pafuncio");
```

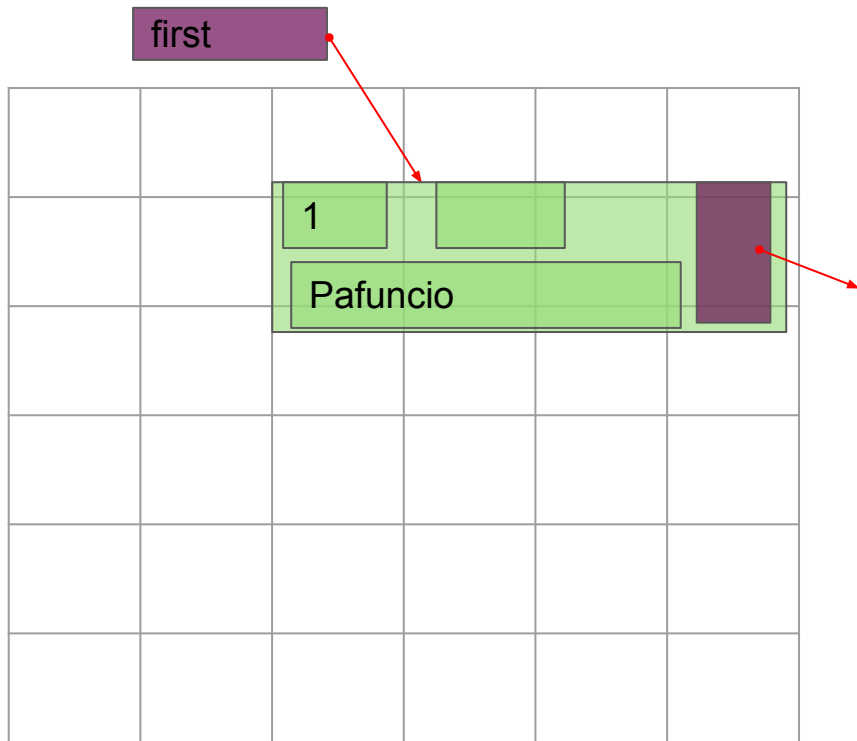
```
first->income = 3000.0;
```

```
first->next = NULL;
```





# Memória



```
Employee *first; //head
```

```
first = (Employee *) malloc(sizeof(Employee));
```

```
first->id = 1;
```

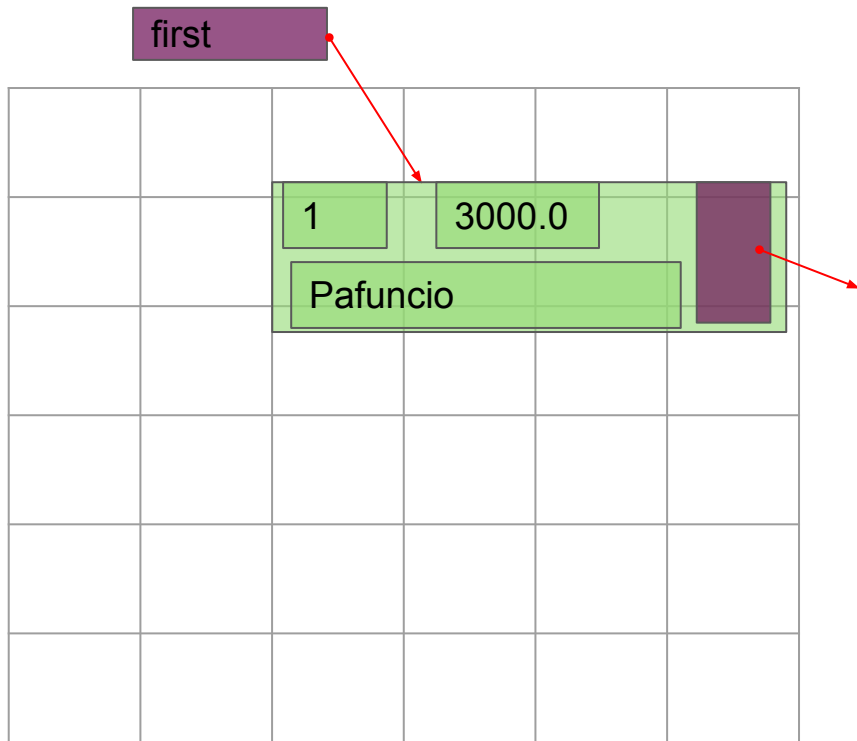
```
strcpy(first->name, "Pafuncio");
```

```
first->income = 3000.0;
```

```
first->next = NULL;
```



# Memória



```
Employee *first; //head
```

```
first = (Employee *) malloc(sizeof(Employee));
```

```
first->id = 1;
```

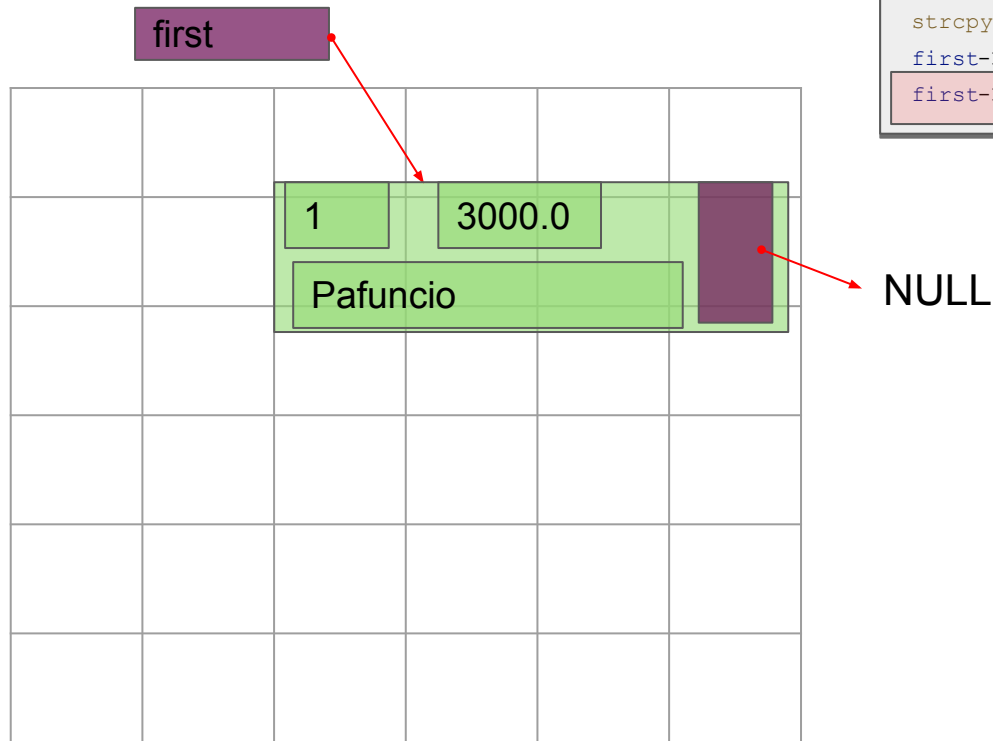
```
strcpy(first->name, "Pafuncio");
```

```
first->income = 3000.0;
```

```
first->next = NULL;
```



# Memória



```
Employee *first; //head
```

```
first = (Employee *) malloc(sizeof(Employee));
```

```
first->id = 1;
```

```
strcpy(first->name, "Pafuncio");
```

```
first->income = 3000.0;
```

```
first->next = NULL;
```

# Lista encadeada

- Adicionando outros elementos
  - Preciso pensar no encadeamento da lista

```
struct employee{
    int id;
    char name[TAM_NOME+1];
    double income;
    struct employee *next;
}
typedef struct employee Employee;
```

```
Employee *f, *aux, *first=NULL;
for (/*QUANTOS EU QUISEI*/){
    f = (Employee *)malloc(sizeof(Employee));
    f->id = contador; //contador é uma variável qualquer
    scanf("%s", f->name);
    f->income = 3000.0;
    f->next = NULL;
    if (first == NULL){
        first = f; aux=f; // inicialmente, todos apontam para a primeira
    } else {
        // região alocada
        aux->next=f; // aux deve apontar sempre para a região
        aux=f;      // anterior à nova alocada
    }
}
```

# Lista encadeada

- Como imprimimos os elementos
  - Para imprimir devemos iterar sobre todos os elementos partindo do primeiro

```
struct employee{
    int id;
    char name[TAM_NOME+1];
    double income;
    struct employee *next;
}
typedef struct employee Employee;
```

```
Employee *aux; //vai ser nosso 'contador'
for (aux = first; aux != NULL; aux = aux->next){
    //aqui aux vale o elemento atual na lista.
    printf("Employee id: %d, name: %s, income: %lf\n", aux->id, aux->name, aux->income);
}
```

# Lista encadeada

- Como criamos a lista
  - Existe várias formas
  - Outra possibilidade

```
struct employee {  
    int id;  
    char name[TAM_NOME+1];  
    double income;  
};  
typedef struct employee Employee
```

```
struct list {  
    Employee *element;  
    struct list *next;  
};  
typedef struct list List;
```

# Lista encadeada

- Como criamos a lista
  - Existe várias formas
  - Outra possibilidade


```
struct employee {  
    int id;  
    char name[TAM_NOME+1];  
    double income;  
};  
typedef struct employee Employee;
```

```
struct list {  
    Employee *element;  
    struct list *next;  
};  
typedef struct list List;
```

```
List *first;
```

# Atenção! Achtung! Attento! Watch out! Attention!

```
struct employee{
    int id;
    char name[41];
    double income;
    Date dbirth;
    struct employee *next;
};
typedef struct employee Employee;
```



```
typedef struct {
    int day;
    int month;
    int year;
} Date;
```

```
Employee emp;
emp.id=10;
strcpy(emp.name, "Your Name");
emp.income=4500.45;
emp.dbirth.day=10;
emp.dbirth.month=8;
emp.dbirth.year=2000;
emp.next=NULL;
```

```
Employee *emp;
emp=(Employee *)malloc(sizeof(Employee))
emp->id=10;
strcpy(emp->name, "Your Name");
emp->income=4500.45;
emp->dbirth.day=10;
emp->dbirth.month=8;
emp->dbirth.year=2000;
emp->next=NULL;
```



# Importante

- Toda a memória alocada deve ser liberada se não os espaços ficarão perdidos
  - Algumas linguagens possuem um sistema de limpeza de memória: **garbage collector**
  - O C padrão não!
    - `free()`

```
Employee *first; //head é legal também :)

first = (Employee *) malloc(sizeof(Employee));
first->id = 1;
strcpy(first->name, "Pafuncio");
first->income = 3000.0;
first->next = NULL;

:

free(first);
```

# Exercícios

1. Considerando as definições a seguir, faça o que é pedido nos itens abaixo:

```
struct date {  
    int dia;  
    int mes;  
    int ano;  
};  
typedef struct date Date;
```

```
struct employee{  
    int id;  
    char name[41];  
    double income;  
    Date dbirth;  
    struct employee *next;  
};  
typedef struct employee Employee;
```

- Crie as estruturas indicadas, e crie o primeiro funcionário da lista encadeada;
- Adicione um segundo funcionário mantendo o encadeamento;
- Crie uma função que receba o ponteiro inicial da lista e imprima todos os elementos (funcionários)

# Exercícios

2. Considerando a estrutura proposta no exercício anterior, faça as seguintes adaptações em seu programa:
  - a. O programa deve ler (do teclado) vários registros de funcionários (quando `id` for igual a 0 a entrada é finalizada).
  - b. Use a mesma função implementada anteriormente e imprima a lista para ver se todos os elementos estão presentes
  - c. Crie uma função para desalocar a memória de todas as instâncias de funcionário (nós da lista).
  - d. **Super desafio:** crie uma nova função que imprime a lista na ordem inversa
    - Dica: vimos uma possível técnica em aulas antes da A1 :)