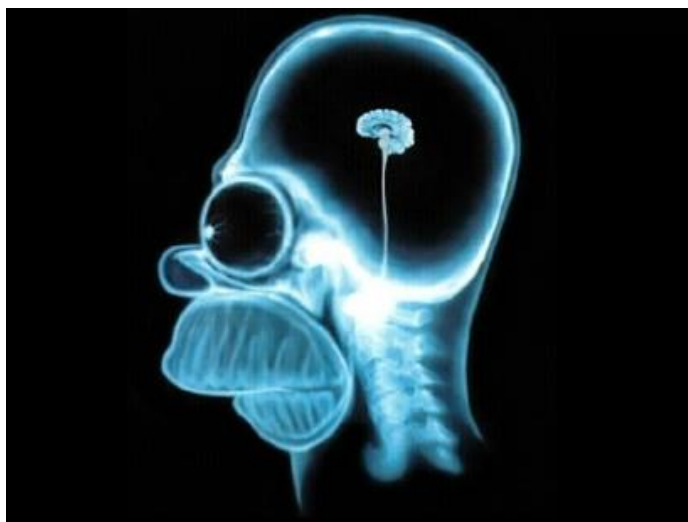




**DEI**  
DEPARTAMENTO  
DE ENGENHARIA INFORMÁTICA  
TÉCNICO LISBOA

# Alocação dinâmica de memória



IAED

# Alocação Dinâmica de Memória

- Alocação estática

```
int tab[100];
```

- Memória alocada durante o *scope* da variável
  - Não é possível libertar quando não necessária
  - Não é possível utilizar fora do *scope*
- Solução: **alocação dinâmica** !

# Alocação Dinâmica de Memória

- Função `malloc`

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

- Recebe como argumento o número de *bytes*
  - Tipo `size_t` representa uma dimensão em *bytes*
- Devolve um ponteiro (endereço) para o primeiro *byte* do bloco de memória contígua alocada
  - `void *` indica um ponteiro para um tipo não especificado
  - permite utilização com qualquer tipo de dados
  - posteriormente faz-se conversão para o tipo correcto por *type cast*

```
int *vec;
```

```
vec = (int*) malloc(sizeof(int)*100);
```

# Alocação Dinâmica de Memória

- Libertação de memória é efectuada com a função `free`

```
void free(void *ptr) ;
```

- Recebe como argumento o ponteiro para a primeira posição do bloco de memória contígua a libertar
- Não devolve nada
- Como libertar a memória reservada com o malloc anterior?

```
free(vec) ;
```

- Tanto `malloc` como `free` estão definidas em `stdlib.h`
  - Necessário `#include <stdlib.h>`
  - Sugestão: Usem o `valgrind`

# Exemplo

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

int* CriaVectorInt(int tamanho) {
    int *v, i;

    v = (int*) malloc(tamanho * sizeof(int));
    for (i = 0; i < tamanho; i++)
        v[i] = 0;
    return v;
}
```

# Exemplo

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

int* CriaVectorInt(
    int *v,i;
    v = (int*)mallo
    for (i=0;i<tama
        v[i]=0;
    return v;
}
```

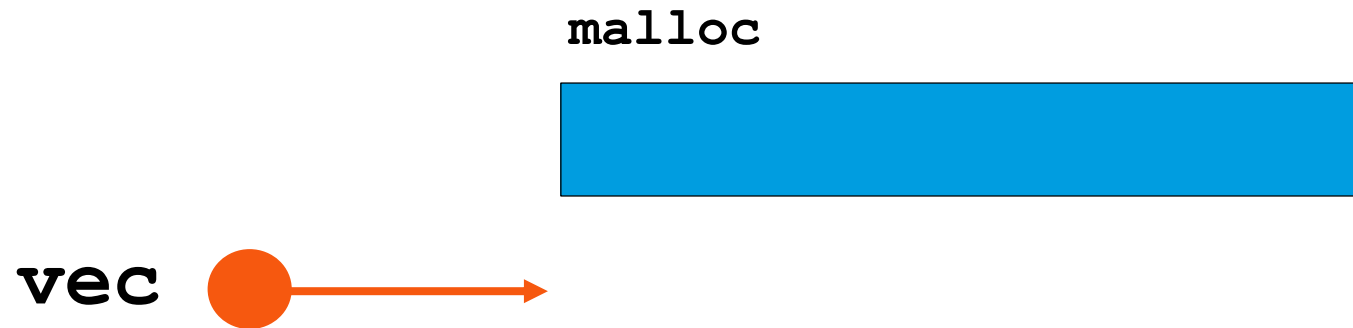
```
int main() {
    int *vec, t;

    puts("introduza o numero de elementos\n");
    scanf("%d",&t);
    vec = CriaVectorInt(t);
    /* faço qq coisa com o vec */
    free(vec);

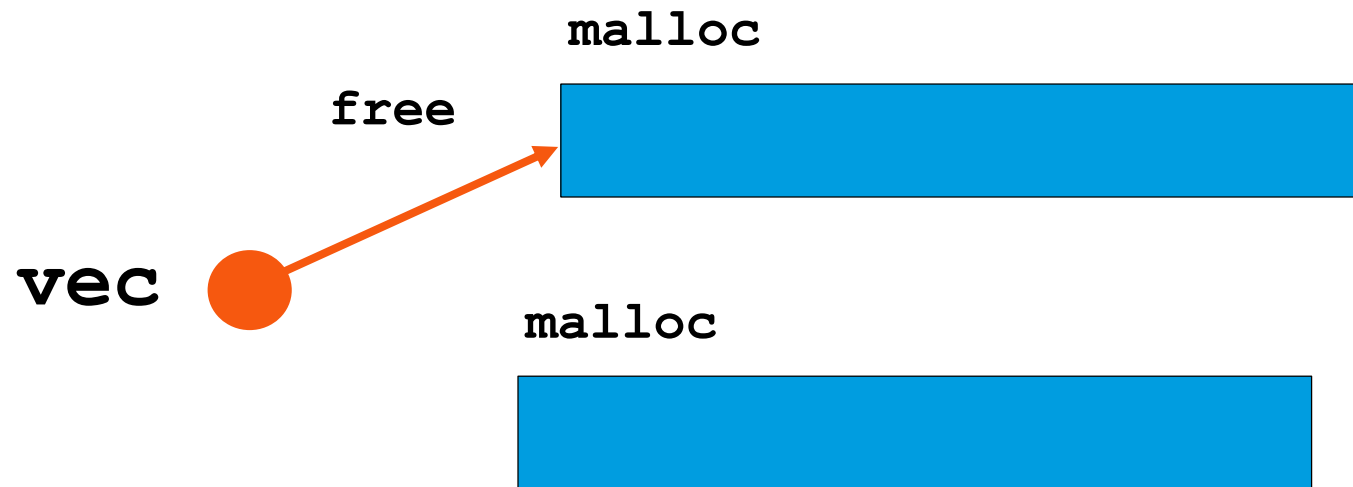
    puts("introduza o numero de elementos\n");
    scanf("%d",&t);
    vec = CriaVectorInt(t);
    /* faço qq coisa com o vec */
    free(vec);

    return 0;
}
```

# Exemplo



# Exemplo





# Exemplo



# Alocação Dinâmica de Memória

- Função `realloc`

```
void *realloc(void *ptr, size_t size);
```

- Recebe ponteiro `ptr` para bloco de memória antigo e dimensão `size` do novo bloco de memória
- Devolve ponteiro para novo bloco de memória
- Copia valores do bloco antigo para o novo
  - Se novo for mais pequeno, só copia até caber
  - Se novo for maior, copia tudo e deixa o resto sem ser inicializado

```
vec = (int*) realloc(vec, sizeof(int)*250);
```

# Outras funções úteis

Existem outras funções úteis para, por exemplo, inicializar a memória alocada ou para copiar segmentos de memória.

```
void *calloc(size_t nmemb, size_t size);
```

permite tal como a função **malloc** alocar memória, neste caso para um vector com **nmemb** elementos em que cada elemento tem **size** bytes, mas em que a memória é inicializada a zero.

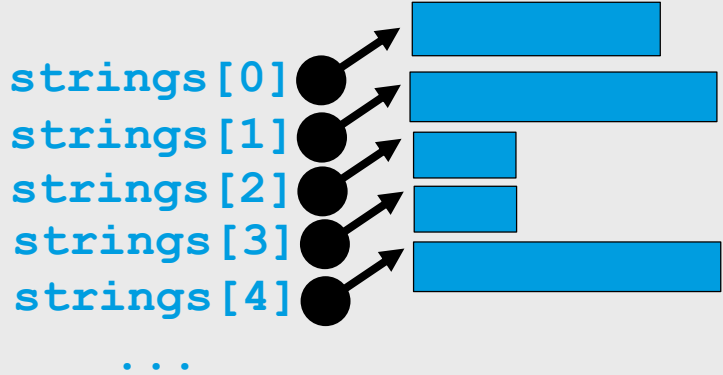
Ex: Reserva memória para um bloco de 100 inteiros

```
vec = (int*) calloc (100, sizeof(int));
```

# Exemplo: matriz de 10 linhas de tamanhos variáveis

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

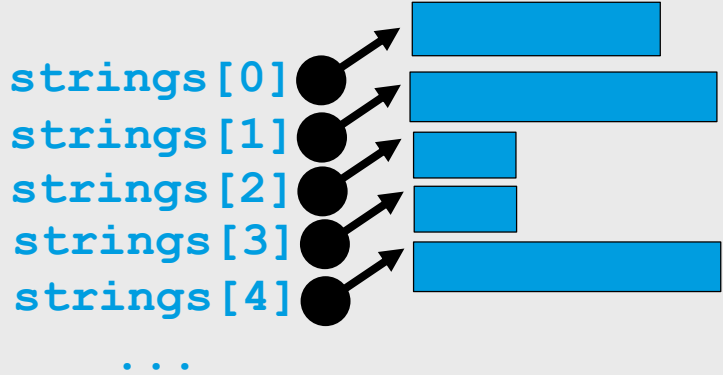
int main() {
    char buffer[256];
    char* strings[10];
    int i;
    for(i=0;i<10;i++){
        printf("introduza uma palavra\n");
        scanf("%s",buffer);
        strings[i]=(char*)malloc(sizeof(char)*(strlen(buffer)+1));
        strcpy(strings[i],buffer);
    }
    for(i=0;i<10;i++)
        printf("%s\n", strings[i]);
    return 0;
}
```



# Mesmo exemplo... Tudo alocado dinamicamente

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

int main() {
    char buffer[256];
    char** strings;
    int i;
    strings = (char**)malloc(sizeof(char*)*10);
    for(i=0;i<10;i++){
        printf("introduza uma palavra\n");
        scanf("%s",buffer);
        strings[i]=(char*)malloc(sizeof(char)*(strlen(buffer)+1));
        strcpy(strings[i],buffer);
    }
    for(i=0;i<10;i++)
        printf("%s\n", strings[i]);
    return 0;
}
```





**DEI**  
DEPARTAMENTO  
DE ENGENHARIA INFORMÁTICA  
TÉCNICO LISBOA

# Ponteiros para estruturas

IAED

# Ponto de Situação

- *Ponteiros e Tabelas*
- *Alocação Dinâmica de Memória*
- Estruturas, Funções e Apontadores
- Estruturas Auto-Referenciadas
- Exemplo de aplicação:
  - Listas ligadas
  - Pilhas

# Declaração de Estruturas (recapitulação)

- Declaração de variável do tipo estrutura:

```
typedef struct ponto {  
    double x;  
    double y;  
} Ponto
```

```
Ponto centro;
```

- Manipulação : `<variavel>.<membro>`

```
centro.x = 12.3;  
centro.y = 5.2;
```



# Declaração de Estruturas

- Declaração de variável do tipo estrutura:

```
typedef struct ponto {  
    double x;  
    double y;  
} Ponto  
  
Ponto centro;  
Ponto *pcentro = &centro;
```

- Manipulação : `(*<variavel>) .<membro>`

```
(*pcentro) .x = 12.3;  
(*pcentro) .y = 5.2;
```

*Mas isto é  
aborrecido!*

# Declaração de Estruturas (operador ->)

- Declaração de variável do tipo estrutura:

```
typedef struct ponto {  
    double x;  
    double y;  
} Ponto  
  
Ponto centro;  
Ponto *pcentro = &centro;
```

- Manipulação :  $(*\text{<ponteiro>}) . \text{<membro>}$  é equiv. a  $\text{<ponteiro>} \text{->} \text{<membro>}$

```
pcentro->x = 12.3;  
pcentro->y = 5.2;
```

# Ponteiros para Estruturas

- Declaração de ponteiro para uma estrutura:

```
Ponto *pcentro;
```

- A declaração de um ponteiro **não aloca memória!!**
  - Se quisermos alocar memória de forma explícita fazemos:

```
pcentro = (Ponto*) malloc(sizeof(Ponto));
```

# Estruturas e Funções

- Funções podem receber e retornar estruturas

```
Ponto adicionaPonto(Ponto p1, Ponto p2) {  
    Ponto res;  
    res.x = p1.x + p2.x;  
    res.y = p1.y + p2.y;  
    return res;  
}
```

- Função retorna *cópia* da estrutura `res`
- Passagem de argumentos feita por valor
  - Chamada `adicionaPonto(pa, pb)` não altera valores de `pa` nem `pb`.

Adicionaponto cria  
cópias de `pa` e `pb` que só  
existem dentro da função!

Será eficiente?

# Estruturas, Funções e Ponteiros

- Passagem de estruturas grandes como parâmetros é **ineficiente**
  - É necessário efectuar a cópia de todos os campos
- Utilizam-se normalmente ponteiros para estruturas
- Podemos alterar o conteúdo dos argumentos!

```
void adicionaPonto(Ponto *p1, Ponto *p2, Ponto *res) {  
    res->x = p1->x + p2->x;  
    res->y = p1->y + p2->y;  
}
```

# Estruturas, Funções e Ponteiros

- Também podemos reservar memória para uma estrutura que será utilizada fora da função
  - Reservamos memória com o malloc e retornamos o pointer para a memória alocada.

```
Ponto* adicionaPonto(Ponto *p1, Ponto *p2) {  
    Ponto *res;  
  
    res = (Ponto *) malloc(sizeof(Ponto));  
    res->x = p1->x + p2->x;  
    res->y = p1->y + p2->y;  
  
    return res;  
}
```

# Estruturas, Funções e Ponteiros

- Também podemos reservar memória para uma estrutura que será utilizada fora da função
  - Reservamos memória com o malloc e retornamos o ponteiro para a memória alocada.

```
Ponto* adicionaPonto(Ponto *p1, Ponto *p2) {  
    Ponto *res;  
  
    res = (Ponto *) malloc(sizeof(Ponto));  
    res->x = p1->x + p2->x;  
    res->y = p1->y + p2->y;  
  
    return res;  
}
```

**ATENÇÃO:**  
em geral, para cada  
malloc tem de haver um  
free!!

Uma “*memory leak*”  
ocorre sempre que  
“perdemos” o endereço  
de memória do objecto  
alocado.

# Estruturas, Funções e Ponteiros

- Exemplo de fuga de memória (**ERRO!!**):

```
void printSOMA(Ponto *p1, Ponto *p2) {  
    Ponto *res;  
  
    res = (Ponto *) malloc(sizeof(Ponto));  
    res->x = p1->x + p2->x;  
    res->y = p1->y + p2->y;  
  
    printf("( %d, %d) \n", res->x, res->y);  
}
```

**ATENÇÃO:**  
em geral, para cada  
**malloc** tem de haver um  
**free!!**

Uma “*memory leak*”  
ocorre sempre que  
“perdemos” o endereço  
de memória do objecto  
alocado.



# Estruturas, Funções e Ponteiros

- Exemplo (corrigido):

```
void printSOMA(Ponto *p1, Ponto *p2) {  
    Ponto *res;  
  
    res = (Ponto *) malloc(sizeof(Ponto));  
    res->x = p1->x + p2->x;  
    res->y = p1->y + p2->y;  
  
    printf("( %d, %d) \n", res->x, res->y);  
    free (res);  
}
```

**ATENÇÃO:**  
em geral, para cada  
malloc tem de haver um  
free!!

Uma “*memory leak*”  
ocorre sempre que  
“perdemos” o endereço  
de memória do objecto  
alocado.