



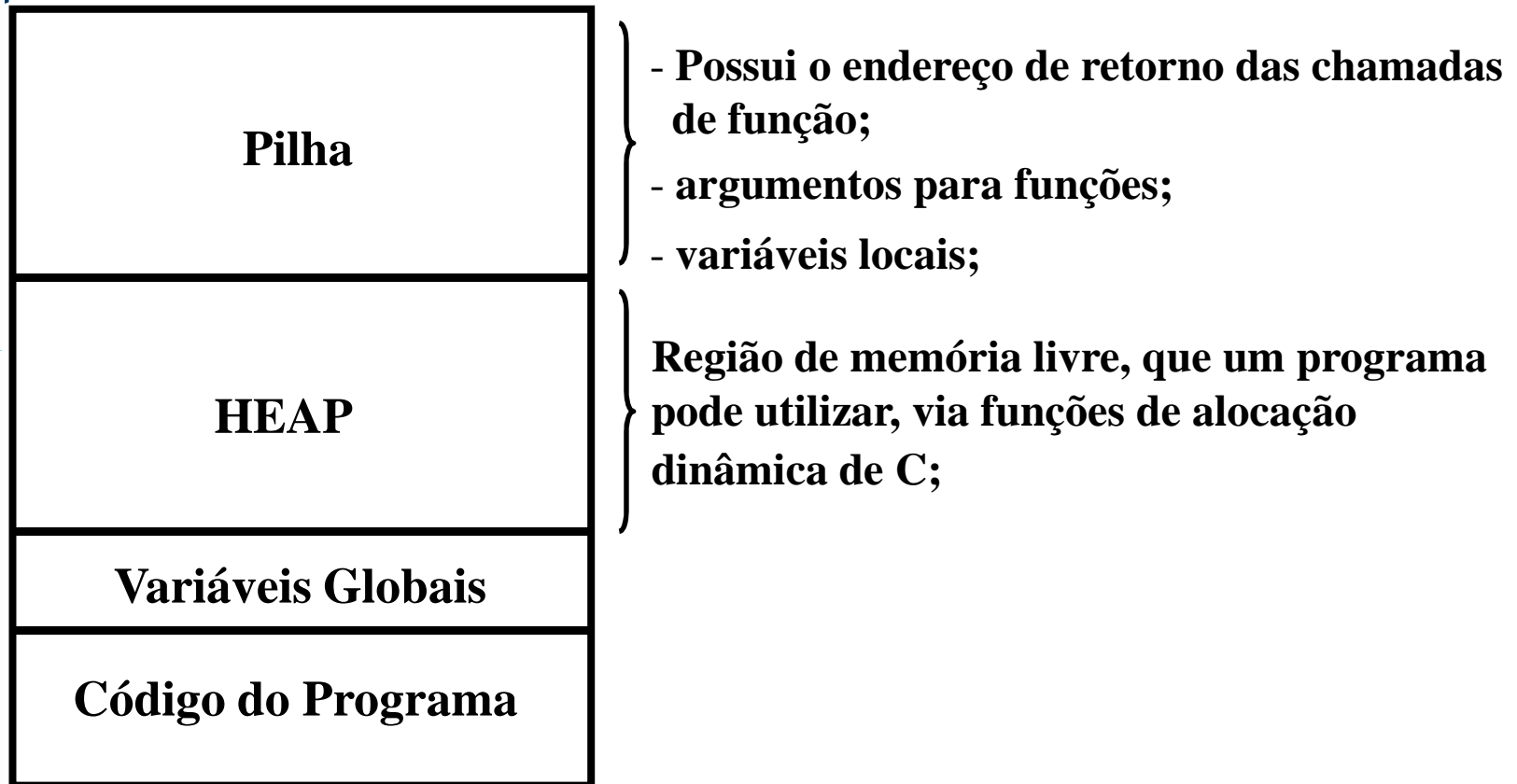
# *Alocação Dinâmica de Memória*



# Alocação de Memória

- Duas maneiras (mais comuns) de reservar memória:
  - a. Reserva estática de espaços de memória com tamanho fixo, na forma de variáveis locais :  
`int a; char nome[64];`
  - b. Reserva dinâmica de espaços de memória de tamanho arbitrário, com o auxílio de ponteiros:  
`int *a; char *nome;`
- Variáveis não podem ser acrescentadas em tempo de execução
  - Porém, um programa pode precisar de quantidade variável de memória
- A reserva só ocorre durante a execução do programa, através de requisições ao SO

# Mapa de Memória



Mapa conceitual de memória de um programa em C



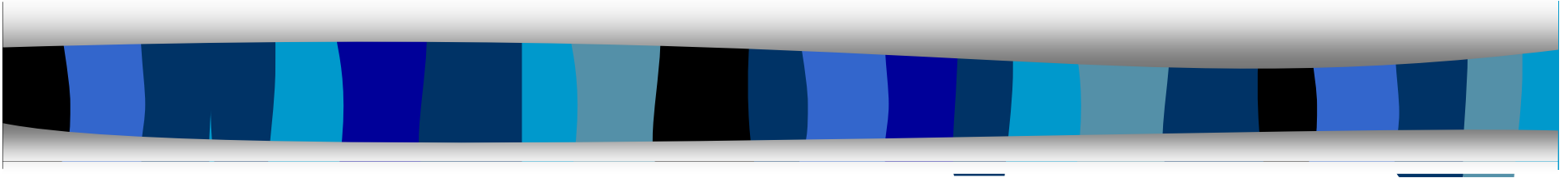
# Alocação de Memória

- Vantagens da alocação dinâmica:
  - Flexibilidade: muitas vezes não temos como prever, antecipadamente, as necessidades de uso de memória.
  - Economia: podemos reservar memória de acordo com a necessidade imediata. Evitamos super-dimensionamentos.
- Desvantagem principal:
  - Gerenciamento de Memória:
    - A alocação dinâmica atribui parte da responsabilidade de gerenciar memória ao programador. Essa tarefa, geralmente, é propensa a erros de difícil detecção e correção.
    - Gerenciar a memória envolve: reserva de memória, acesso correto às regiões alocadas, liberação de regiões não mais utilizadas



# Funções de Alocação em C

- A alocação e liberação de espaços de memória é feito por funções da biblioteca “stdlib.h” (em alguns sistemas “malloc.h”)
- As principais:
  - [malloc\(size\)](#): “*memory allocation*” Aloca espaço em memória
  - [free\(ref\)](#): Libera espaço em memória, alocado dinamicamente
- Alternativas:
  - [calloc\(n, size\)](#): “*count allocation (?)*” Aloca espaço em memória para um *array* de *n* elementos de tamanho *size*
  - [realloc\(ref, size\)](#): Modifica o tamanho de um bloco de memória previamente alocado.



## *Funções de Alocação Dinâmica: referência*



# Função malloc()

- Aloca um bloco consecutivo de bytes na memória e retorna o endereço deste bloco.
- Devemos informar o tamanho do bloco, por parâmetro, em número de bytes
  - Frequentemente, devemos usar a função “*sizeof()*”
- O espaço alocado pode ser usado para armazenar qualquer tipo de dado (*void \**).
  - Devemos converter o tipo genérico retornado (*void \**) para o tipo desejado (*cast*)

Ex:

```
Aluno *a;  
a = (Aluno *)malloc(sizeof(Aluno));
```



# Função free()

- Libera o uso de um bloco de memória, permitindo que este espaço seja reaproveitado.
- Deve ser passado para a função free() exatamente o mesmo endereço retornado por uma chamada da função malloc()
- A determinação do tamanho do bloco a ser liberado é feita automaticamente.

Ex:

```
int *p;  
p = (int*) malloc(100 * sizeof(int));  
free(p);
```





# Função calloc()

- Função equivalente a malloc(), usada para alocar espaço para um vetor de elementos  
`calloc(10, sizeof(int)) ≡ malloc(10 * sizeof(int))`
- Devemos informar, por parâmetro, o tamanho do vetor e o tamanho (em bytes) de cada elemento desse vetor.
  - O espaço alocado é iniciado com bits 0
- Esta função também retorna um ponteiro para void (void\*)
  - Devemos converter o tipo genérico retornado para o tipo desejado

Ex:

```
Aluno *a;  
a = (Aluno *)calloc(10, sizeof(Aluno));
```



# Função realloc()

- Função utilizada para modificar o tamanho ocupado por uma área de memória já alocada
- Devemos informar, por parâmetro, a referência da área a ser redimensionada, e o novo tamanho (em bytes).
  - Se a área original não puder ser redimensionada, uma nova área é criada, e a antiga é liberada.
- Esta função também retorna um ponteiro para void (void\*)

Ex: 

```
int *a, *b;  
a = (int *)malloc(sizeof(int));  
b = (int *)realloc(a, sizeof(int)*4);
```

OBS: `realloc(NULL, size) ≡ malloc(size)`

# Exemplo: Alocação de Matrizes

- No caso de vetores multidimensionais (e.g. matrizes), devemos alocar um vetor de apontadores, i.e. um apontador por linha, e depois um vetor de elementos para cada linha

Ex:

```
float **mat; /* matriz de elementos do tipo float */
int nlin = 10, ncol=10; /* numeros de linhas e colunas */
mat = (float **)calloc(nlin, sizeof(float *));
    /* aloca vetor de ponteiros para variaveis float */
if (mat != NULL){
    int i;
    for (i=0; i < nlin; i++) {
        mat[i] = (float *)calloc(ncol, sizeof(float));
            /* aloca vetor de variaveis float */
    }
}
```



# Exercício (Lab 4)

8. Implemente um programa que:
  - a. Crie uma função que receba “tam” como parâmetro um número inteiro entre 10 e 100. Então, deve criar um vetor de inteiros com números entre 0 e 50, cujo tamanho é definido por “tam”.
  - b. Crie uma função que receba dois vetores de inteiros (e seus respectivos tamanhos), como parâmetro, e retorne a concatenação dos dois como um terceiro vetor.
  - c. Crie uma função que receba como parâmetro um vetor de inteiros (e seu tamanho), e imprima seus elementos na tela.
  - d. O programa principal deve usar a função em (a.) para criar dois vetores de inteiros de tamanhos distintos (definidos pelo usuário). Deve, então, usar a função em (b.) para concatenar os dois vetores. Finalmente, deve usar a função em (c.) para imprimir os dois vetores originais e o vetor retornado.



# Exercício (Lab 4)

9. Altere o programa do Exercício 7, de forma que no lugar de variáveis do tipo Pessoa e Endereço, ponteiros para esses tipos sejam declarados.

Implemente a função

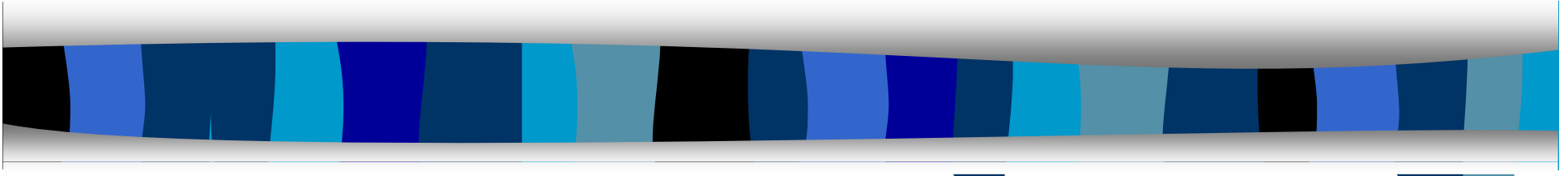
```
Pessoa criaPessoa ()
```

que deve instanciar Pessoa (e Endereço) e deve pedir ao usuário todas as informações (não usual, mas funciona como exercício)

Altere a função abaixo para que receba os dados do endereço e crie um novo objeto Endereco, para associar a “p”

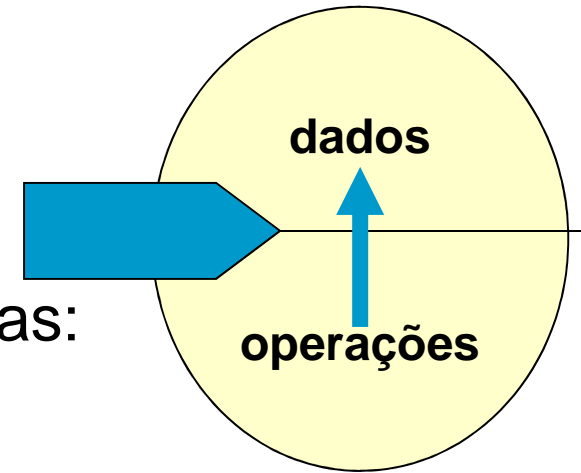
```
void alteraEndereco (Pessoa *p, <dados de endereço>)
```

Faça as alterações que achar necessárias às outras funções, para que continuem funcionando.



*Tipo Abstrato de Dados*

# Introdução:



- Programas consistem em 2 coisas:
  1. Algoritmos e
  2. Estruturas de dados (ED)
- A escolha e a implementação de uma ED é tão importante quanto as rotinas que manipulam os dados.
  - A organização da informação e a forma de acesso é normalmente determinada pela natureza do problema
- TADs são como generalizações de tipos primitivos
  - Exemplo: o conjunto dos inteiros acompanhado das operações de adição, subtração e multiplicação. Um número Complexo não possui suporte nativo: usamos TADs.



# Definição:

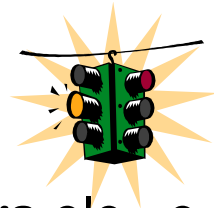
- Tipo Abstrato de Dados é uma especificação de tipo contendo um **conjunto de dados** e **operações** que podem ser executadas sobre esses dados.
  - Descreve quais dados podem ser armazenados (características) e como eles podem ser manipulados (operações)
  - Mas não descreve como isso é implementado no programa
- TADs são geralmente implementados através de tipos compostos heterogêneos (Registros), associados a um conjunto de funções que operam sobre essa estrutura.

- Exemplo:

```
struct Estudante {  
    char nome [64]; int idade; char matricula[10]  
}  
  
int maiorDeIdade(Estudante estudante); //retorna 1 se verdadeiro  
void validaMatricula(Estudante estudante); // efetua matricula
```



# Encapsulamento:

- Tipos Abstratos de Dados utilizam o conceito de **Encapsulamento**, para reduzir dependências entre as estruturas e garantir o comportamento correto das operações sobre os dados.
  - Encapsular: esconder aquilo que não deve ser manipulado diretamente.
- Importante: em um TAD, os dados armazenados são acessados sempre através das funções definidas para ele, e **nunca diretamente** 
  - Independência: é possível alterar a estrutura interna de um TAD sem afetar código cliente
  - Consistência: as funções definidas para um TAD garantem que os dados serão sempre acessados na ordem e da forma corretas

# Exemplo:

## Mundo Real



Pessoa

## Dados de Interesse

- Idade da Pessoa

## ESTRUTURA de Armazenamento

- Tipo Inteiro

## Possíveis OPERAÇÕES

- Nasce ( $\text{idade} = 0$ );
  - Aniversário ( $\text{idade} = \text{idade} + 1$ )

# Exemplo:

## Mundo Real



Fila de Espera

## Dados de Interesse

- Nome de cada pessoa e sua posição na fila

## ESTRUTURA de Armazenamento

- Tipo Fila

## Possíveis OPERAÇÕES

- Sai da Fila (o primeiro)
- Entra na Fila (no fim)

# Alocação para TADs

- Sempre que trabalhamos com tipos abstratos, devemos ter funções para cada operação sobre essa estrutura
- Essas funções, normalmente, recebem por parâmetro uma cópia do TAD que devem processar
  - Sempre que passamos um TAD por parâmetro, criamos uma cópia dessa estrutura
  - Isso é particularmente ineficiente para TADs com muitas informações
  - É mais eficiente se a criação do TAD for realizado por alocação dinâmica, e referência a essa área for passada por parâmetro

```
Lista criaLista() {  
    /* 1000 posições */  
    Lista a;  
    a.ultimo = -1;  
    return a;  
}
```

```
Lista* criaLista(int size) {  
    Lista *a = (Lista *)malloc(sizeof(Lista));  
    a->lista = (int*)calloc(size, sizeof(int));  
    a->ultimo = -1;  
    return a;  
}
```



# Tipos Comuns:

- Existem 4 tipos importantes de TADs, que estudaremos:
  1. Listas (Lineares e Encadeada)
  2. Pilha
  3. Fila
  4. Árvores Binárias
- Cada um desses TADs fornece uma solução para uma classe de problemas
- São essencialmente dispositivos que executam operações específicas de armazenamento e recuperação da informação
- Todos eles armazenam e manipulam itens de dados, onde um item é uma unidade de informação



# Exercício (Lab 5)

10. Sabemos que a linguagem C não possui o tipo String definido. Representamos textos como vetores de char e usamos funções de biblioteca para manipulá-los.

Vamos construir um TAD “String”, que usa um vetor de char para representar o texto (como normalmente fazemos).

Vamos também definir algumas operações para esse TAD:

- a. `String criaString()`: cria uma string vazia (`'\0'`)
- b. `String criaString(char c[])`: cria string contendo `c[]`
- c. `void append(String s1, String s2)`: apenda `s2` em `s1`
- d. `void addChar(String s1, char c)`: adiciona `c` no final de `s1`
- e. `String substring(String s1, int ini, int final)`:  
retorna substring delimitada por `ini` e `final`, como nova String  
(devemos usar alocação dinâmica para algumas tarefas)