Trabalho Dirigido nº 2

Nesta ficha vamos entender os vetores dinâmicos que exploramos na ficha anterior. O aspecto que nos interessa aqui é a manipulação genérica dos elementos de um vetor dinâmico, independentemente dos valores exactos que ali estão arquivados e das operações que pretendemos realizar sobre uma coleção de objetos arquivados. Vamos falar de apontadores de função, de combinadores ou iteradores, de pesquisa e ordenação, mas também de string e ficheiros. Na sequência, veremos como usar na prática para resolver problemas de programação.

O que vão aprender

- Apontadores de função;
- Modelo de memória dos programas C;
- Iteradores para vetores dinâmicos;
- Pesquisa linear e binária;
- Ordenação e tipos de ordenação;
- Cadeias de caracteres (strings);
- Ficheiros.

Apontadores de função

Comecemos a nossa exploração do conceito de apontadores de função.

Do TD1, aprendemos que podemos ***generalizar*** uma estrutura de dados de tipo "contentor" (como os vetores, que contêm valores de um determinado tipo) a custa de apontadores para `void`. Ou seja, em `C`, sabemos ***generalizar dados***.

Mas será que podemos ***generalizar funções*** num programa `C`?

Sim.

Vamos, para esse efeito, redescobrir esta ideia de apontador para `void` e estender o conceito para as funções.

Um apontador de função, numa primeira abordagem, não é nada mais do que um apontador regular mas não para valores alocados na memória, mas sim para funções.

Assim, na declaração `int (*ptr_fun)(int, int)`, `ptr_fun` é uma variável do tipo apontador de função que pode armazenar o endereço de qualquer função com a assinatura `int (int, int)` (função binária de inteiros que devolve um inteiro).

Reparemos no detalhe dos parêntesis.

`int *ptr_fun ...` define `prt_fun` como um apontador para `int`.

`int (* ptr_fun) ...` define `ptr_fun` como uma apontador de função que devolve um inteiro.

Considere assim as funções

```c
int soma(int a, int b) {
    return a + b;
}

int produto(int a, int b) {
    return a * b;
}
```

Podemos utilizar esta declaração desta forma:

```c

```

```c
int main(void) {
    // Declaração do apontador de função, aponta para soma
    int (*ptr_fun)(int, int) = soma;

    // Chamada da função através do apontador
    int resultado = ptr_fun(10, 5);
    printf("Resultado: %d\n", resultado);
    // Mostra 15

    ptr_fun = produto; //prt_fun aponta agora para produto
    // Chamada da função através do apontador
    resultado = ptr_fun(10, 5);
    printf("Resultado: %d\n", resultado);
    // Mostra 50

    return 0;
}
```

Repare que neste cenário, funções de alocação de memória como `malloc` não são aqui de grande utilidade. O uso que fazemos dos apontadores de funções é apontar para funções existentes, definidas de forma tradicional.

Tecnicamente, funções como `malloc` permitam alocar memória numa zona designada de ***heap***. É uma zona dedicada a dados e não a funções ou código. É no entanto tecnicamente possível, embora totalmente fora do alcance deste trabalho dirigido, criar código na *heap* (um programa é também uma sequência de bit), transferi-lo para a zona dedicada aos programas executáveis e ali transforma-lo em código regular executável. Com isso, acabamos de injectar código num computador onde este poderá causar catástrofes. Matéria para peritos que sabem muito bem o que fazem.

Voltemos ao domínio dos mortais. É possível definir um apontador de funções que seja genérico tanto em relação ao tipo dos valores de retorno quanto aos parâmetros envolvidos? Sim, utilizando `void`!

```c
#include <stdio.h>

int   soma_int     (int a, int b)     {return a + b;}
int   produto_int  (int a, int b)     {return a * b;}
```

```c
float soma_float   (float a, float b) {return a + b;}
float produto_float(float a, float b) {return a * b;}

int main(void) {
    // Declaração de um apontador genérico para funções.
    // Este apontador aponta para funções que retornam void
    // e não especifica parâmetros.
    // Note que isso **não é seguro** em termos de tipagem.
    void (*ptr_fun)();

    // Usando o apontador para funções que operam com inteiros.
    // Faz-se cast da função soma_int para o tipo genérico
    ptr_fun = (void (*)()) soma_int;

    // Chama a função, fazendo cast para o protótipo correto
    int resultado_int = ((int (*)(int, int)) ptr_fun)(10, 5);
    printf("Resultado int (soma): %d\n", resultado_int);
    // Imprime 15

    ptr_fun = (void (*)()) produto_int;
    resultado_int = ((int (*)(int, int)) ptr_fun)(10, 5);
    printf("Resultado int (produto): %d\n", resultado_int);
    // Imprime 50

    // Usando o mesmo apontador para funções que operam com float:
    ptr_fun = (void (*)()) soma_float;
    float resultado_float = ((float (*)(float, float)) ptr_fun)
(10.0f, 5.0f);
    printf("Resultado float (soma): %f\n", resultado_float);
    // Imprime 15.000000

    ptr_fun = (void (*)()) produto_float;
    resultado_float = ((float (*)(float, float)) ptr_fun)(10.0f,
5.0f);
    printf("Resultado float (produto): %f\n", resultado_float);
    // Imprime 50.000000

    return 0;
}
```

É possível, mas nem é muito seguro (há haver um erro, pode pagar-se caro), nem é de todo elegante ou leve...

Note que para usar esta solução genérica, somos obrigado a abusar da conversão de tipos em duplicado, **de** e **para** `ptr_fun`.

```
...
// para atribuir a prt_fun o endereço de produto_int, que não são naturalmente compatíveis em termos de assinatura
ptr_fun = (void (*)()) produto_int;

// e para chamar a função ptr_fun como se fosse uma função binária de inteiros para inteiro
resultado_int = ((int (*)(int, int)) ptr_fun)(10, 5);
...
```

Em resumo, este exemplo introduz um programa que faz uso de um apontador de função (`ptr_fun`) para apontar consecutivamente para várias funções (`soma_int`, `produto_int`, `soma_float` e `produto_float`).

Uma nota sobre o perigo dos apontadores de funções.

O mecanismo de cast de apontadores é muito pouco fiscalizado pelo sistema de tipos do C. A responsabilidade é do programador e, como bem sabemos, o programador sabe sempre muito bem o que faz... Veja:

```
#include <stdio.h>

// Função que recebe dois inteiros e retorna um inteiro
int soma(int a, int b) {
    return a + b;
}
```

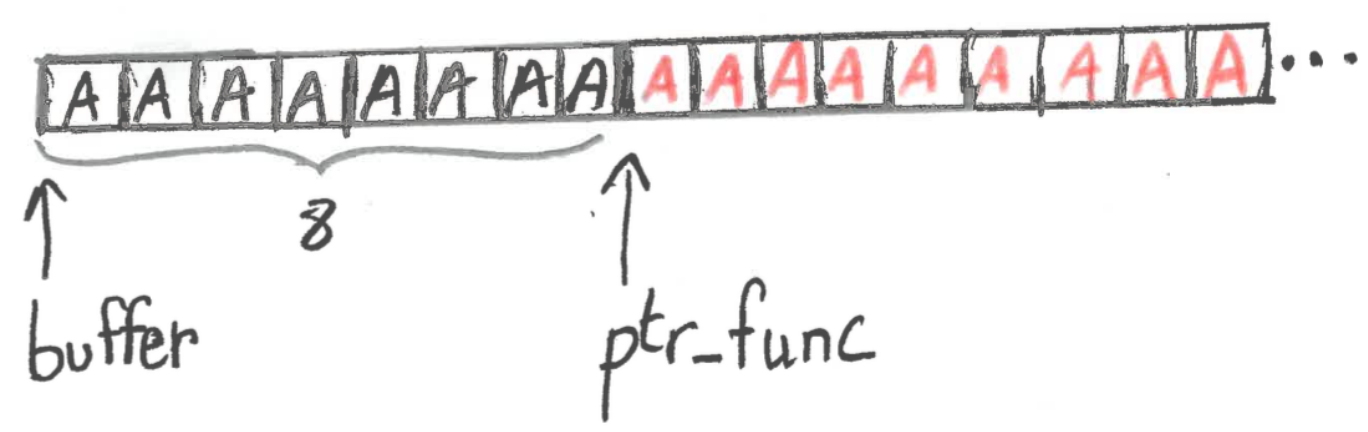
```c
int main(void) {
    // Apontador de função com assinatura errada (deveria ser int (*)(int, int))
    int (*ptr)(int) = (int (*)(int)) soma;  // Casting inadequado! e o compilador não se queixa!

    // Chamamos a função incorretamente e o compilador não se queixa
    printf("%d\n", ptr(10)); // Comportamento indefinido!

    return 0;
}
```

Como exposto nas aulas teóricas, podemos tirar proveito da fraca tipagem do C no que diz respeito à conversão de tipos e da fraca verificação de consistência no uso dos vetores/strings para provocar um *buffer overflow* potencialmente catastrófico.

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

// Função legítima
void funcao_normal() {
    printf("Função normal executada.\n");
}

// Função maliciosa
void funcao_maliciosa() {
    printf(" Código malicioso executado! \n");
}

int main(void) {
    void (*ptr_func)() = funcao_normal; // Apontador de função legítimo
    char buffer[8];

    // Simula um buffer overflow que sobrescreve o apontador de função (definido "ao lado" de buffer)
```

```
    strcpy(buffer, "AAAAAAAAAAAAAAAAAA");
    // note que estamos a copiar para buffer dados em excesso, daí
o Overflow
    // maliciosamente bem definido, este Overflow pode modificar
ptr_func!

    printf("Chamando a função:\n");
    ptr_func(); // Se o overflow modificar ptr_func, pode chamar
funcao_maliciosa!

    return 0;
}
```

Analisemos a linha `strcpy(buffer, "AAAAAAAAAAAAAAAA");`. Esta é a linha realmente problemática deste exemplo. O que fazemos ali é copiar para a zona apontada por `buffer` que supostamente aponta para uma cadeia de caracteres de tamanho 8, uma string bem maior. Esta string será copiada e vai sobrepor o seu conteúdo ao que está nas 8 posições apontadas por `buffer`, mas também no espaço adjacente. Este espaço é o espaço onde podemos encontrar `ptr_func`!



Assim, bem escolhido, podemos alterar maliciosamente (ou por acidente) o valor de `ptr_func` e conseguir correr o código que o programador não entendia, em primeira mão, executar.

Diabólico...

Voltemos para o lado do bem e do útil. Continuemos a nossa exploração deste conceito. Podemos arquivar estes apontadores em vetores? Sim.

Vejamos este exemplo:

```c
#include <stdio.h>


int soma      (int a, int b) {return a + b;}
int subtrai   (int a, int b) {return a - b;}
int multiplica(int a, int b) {return a * b;}
int divide    (int a, int b) {return (b)? a / b : 0;}

int main(void) {
    // Vetor de apontadores para funções. Cada função tem a assinatura int (int, int)
    int (*operacoes[4])(int, int) = { soma, subtrai, multiplica, divide };

    int a = 20, b = 5;

    // Usamos o vetor de funções para executar as operações
    printf("soma(%d, %d) = %d\n", a, b, operacoes[0](a, b));
    printf("subtrai(%d, %d) = %d\n", a, b, operacoes[1](a, b));
    printf("multiplica(%d, %d) = %d\n", a, b, operacoes[2](a, b));
    printf("divide(%d, %d) = %d\n", a, b, operacoes[3](a, b));

    return 0;
}
```

Exercício 1

Crie um programa que lê um vetor de inteiros e aplique uma transformação em cada elemento usando um vetor de apontadores de função. O programa deve:

- Oferecer um menu para o utilizador escolher uma transformação:

        - Dobrar o valor.
        - Triplicar o valor.
        - Elevar o valor ao quadrado.
    - Implementar cada transformação na forma de uma função com a assinatura `int (int)` (i.e. de `int` para `int`).
    - Armazenar as funções num vetor de apontadores de função do tipo `int (*)(int)`.
    - Aplicar a função escolhida a cada elemento do vetor e exibir o vetor transformado.

Dicas:

- Para cada elemento, a chamada será algo como:

```
vetor[i] = vetor_funcoes[opcao](vetor[i]);
```

- Certifique-se de validar a entrada do utilizador para evitar índices inválidos.

Exercício 2

Vamos usar vetores de apontadores de função para facilitar o teste de uma biblioteca de pilhas (como a que fizemos no TD1).

Considere a biblioteca seguinte, que oferece o serviço simplificado de uma pilha (`stack.h` e `stack.c`)

```
#ifndef STACK_H
#define STACK_H

#include <stdbool.h>

#define STACK_SIZE 1000

typedef struct {
    int data[STACK_SIZE];
```

```
    int top;
} Stack;

void init(Stack *s);
void push(Stack *s, int value);
int  pop(Stack *s);
int  top(Stack *s);
bool is_empty(Stack *s);

#endif
```

```
#include <stdio.h>
#include "stack.h"

void init(Stack *s) {
    s->top = -1;
}

void push(Stack *s, int value) {
    if (s->top == STACK_SIZE - 1) {
        printf("Error: Stack Overflow!\n");
        return;
    }
    s->data[++(s->top)] = value;
    printf("Pushed: %d\n", value);
}

int pop(Stack *s) {
    if (is_empty(s)) {
        printf("Error: Stack Underflow!\n");
        return -1;  // valor para indcar um erro
    }
    int value = s->data[(s->top)--];
    printf("Popped: %d\n", value);
    return value;
}
```

```c
int top(Stack *s) {
    if (is_empty(s)) {
        printf("Error: Stack is empty!\n");
        return -1;
    }
    printf("Top: %d\n", s->data[s->top]);
    return s->data[s->top];
}

bool is_empty(Stack *s) {
    return s->top == -1;
}
```

O objetivo é criar um mecanismo de teste da biblioteca que leia os comandos a partir da entrada padrão (`stdin`), execute as operações correspondentes na pilha e exiba os resultados. O programa deve usar **um vetor de apontadores de função** para mapear dinamicamente os comandos para suas respectivas funções.

Assim com a entrada seguinte:

```
push 10
push 20
top
pop
push 30
pop
pop
pop
```

A saída deverá ser semelhante a:

```
Pushed: 10
Pushed: 20
Top: 20
Popped: 20
Pushed: 30
Popped: 30
Popped: 10
Error: Stack Underflow!
```

Requisitos:

1. **Usar um vetor de apontadores de função** para associar cada comando a sua função correspondente.
2. **Ler e processar comandos da entrada padrão**, extraindo argumentos quando necessário (`push X` deve extrair `X` e empilhá-lo).
3. **Lidar com operações inválidas**, como tentar remover um elemento de uma pilha vazia.

Dicas:

1. Criar **um vetor de apontadores de função** para associar cada comando (`push`, `pop`, `top`) à função correta.
2. Criar um **interpretador de comandos** que:
    - Lê a entrada do usuário.
    - Divide o comando e seus argumentos.
    - Encontra a função correspondente no vetor de apontadores e executa-a.

Vamos aqui dar um esqueleto da solução. Espera-se que complete o ficheiro. Ou seja, que proponha código nos locais onde aparece o comentário `// COMPLETAR`.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "stack.h"

#define MAX_COMMAND_LEN 30
```

```c
// Estrutura para mapar o nome os comandos com funções
typedef struct {
    char *name;
    void (*func)(Stack *, char *); // repare na assinatura!
} Command;

// protótipo das funções
void handle_push(Stack *s, char *arg);
void handle_pop(Stack *s);
void handle_top(Stack *s);

// Implementação das funções
void handle_push(Stack *s, char *arg) {
    // se nenhum argumento foi extraido, assinalar o erro
    if (arg == NULL) {
        printf("Error: No value provided for push.\n");
        return;
    }
    // transformar o argumento de uma string para um inteiro
    int value = atoi(arg);
    push(s, value);
}

void handle_pop(Stack *s) {
    // nada por preprocessar, chamamos directamente
    // a função associada
    pop(s);
}

void handle_top(Stack *s) {
    // nada por preprocessar, chamamos directamente
    // a função associada
    top(s);
}

// vetor que mapeia comandos a funções concretas
Command commands[] = {
    {"push", handle_push},
```

```
    // COMPLETAR!
};

// uma macro para assinalar o número de elementos no vector
commands: tamanho do vetor na sua globalidade dividida pelo tamanho
dos seus elementos (aqui command[0]). Temos assim o número de
elementos.
#define COMMAND_COUNT (sizeof(commands) / sizeof(commands[0]))

int main() {
    Stack stack;
    init(&stack);

    char input[MAX_COMMAND_LEN];

    // ciclo principal: lê até encontrar EOF
    while (fgets(input, MAX_COMMAND_LEN, stdin) != NULL) {

        // Remove o caractere de nova linha
        input[strcspn(input, "\n")] = '\0';

        // Divide a entrada em comando e argumento
        char *command = strtok(input, " ");
        char *arg = strtok(NULL, " ");

        // Ignora linhas vazias
        if (command == NULL) {
            continue;
        }

        // Procura o comando na lista de funções possíveis
        int found = 0;
        for (int i = 0; i < COMMAND_COUNT; i++) {
            if (strcmp(command, commands[i].name) == 0) {
              // Encontrei! O comando está na lista!
              // COMPLETAR!
              // ---> executar a respectiva função com o rspectivo
argumento (args)
              // ....
```

```
                found = 1; // assinalar que o comando foi
encontrado
                break;
            }
        }

        if (!found) { // o comando assinalado não existe!
            printf("Comando desconhecido: %s\n", command);
        }
    }

    return 0;
}
```

Modelo de memória dos programas C.

Vamos agora aos detalhes. Esta parte do trabalho dirigido será mais técnica e agrega, unifica e detalha as diferentes exposições que fizemos até ao momento sobre a forma como a memória é gerida pelos programas C e os executáveis gerados pelo compilador.

Os programas escritos em C seguem um **modelo de memória** estruturado, onde diferentes segmentos são usados para armazenar diferentes tipos de dados. Compreender esse modelo é essencial para evitar erros como **acessos inválidos, vazamentos de memória e corrupção de dados**. Os conhecimentos adquiridos em *Arquitetura de Computadores*, ou de programação em *Assembly* vão ajudar.

Principais Regiões de Memória

A memória de um programa em C é dividida em **cinco regiões principais**:

1. Segmento de Código (Text Segment)
2. Segmento de Dados Estático (Data Segment)
3. Segmento de Heap (Memória Dinâmica)

4. Segmento de Pilha (Stack)
5. Regiões de Memória Mapeada (opcional)

Segmento de Código (`Text Segment`)

- **Armazena:** O código binário do programa, ou seja, as instruções compiladas.
    - **Acessível somente para leitura:** Proteger essa área evita que o código seja modificado durante a execução.
    - **Contém:**
        - Código das funções, incluíndo `main()`
        - Literais de strings constantes (`"hello world"` está armazenado aqui)

Segmento de Dados Estático (`Data Segment`)

São, na verdade, dois segmentos adjacentes. O segmento das variáveis com inicialização e o segmento das variáveis sem inicialização.

- **Armazena:** Variáveis **globais** e **estáticas** (`static`), divididas em:
    - **Segmento `.data`**: Variáveis globais/estáticas **inicializadas**.
    - **Segmento `.bss`**: Variáveis globais/estáticas **não inicializadas**.
- **Exemplo:**

```
int global_var = 42;      // Armazenado no segmento `.data`
static int static_var;    // Armazenado no segmento `.bss`
```

Heap (Memória Dinâmica)

- **Armazena:** Dados alocados dinamicamente com `malloc()`, `calloc()`, `realloc()`.
    - Cresce para endereços **superiores** na memória.
    - Após uso, o programador **deve libertar** a memória (`free()`) para evitar fugas de memória.
- **Exemplo:**

```
int *p = (int *)malloc(sizeof(int) * 10);  // Aloca 10 inteiros na heap, em espaços adjacentes
free(p);  // Liberta a memória
```

Stack (Pilha)

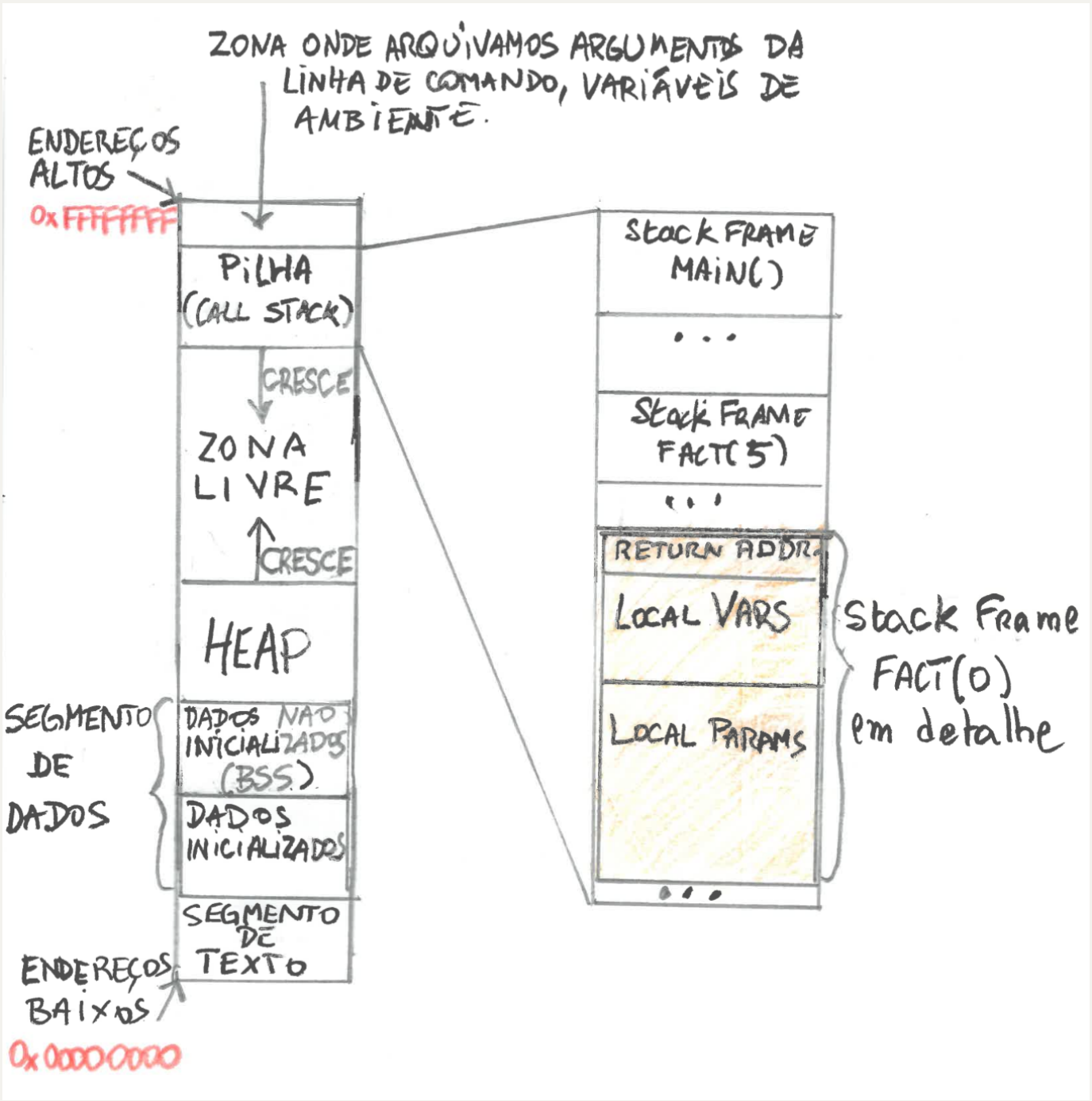
Espaço memória dedicada a gestão das chamadas de funções. É a zona de trabalho para as funções em execução. A zona dedicada a uma função f é *empilhada* quando f é chamada, e é retirada da pilha (política *LIFO*) quando a execução de f termina.

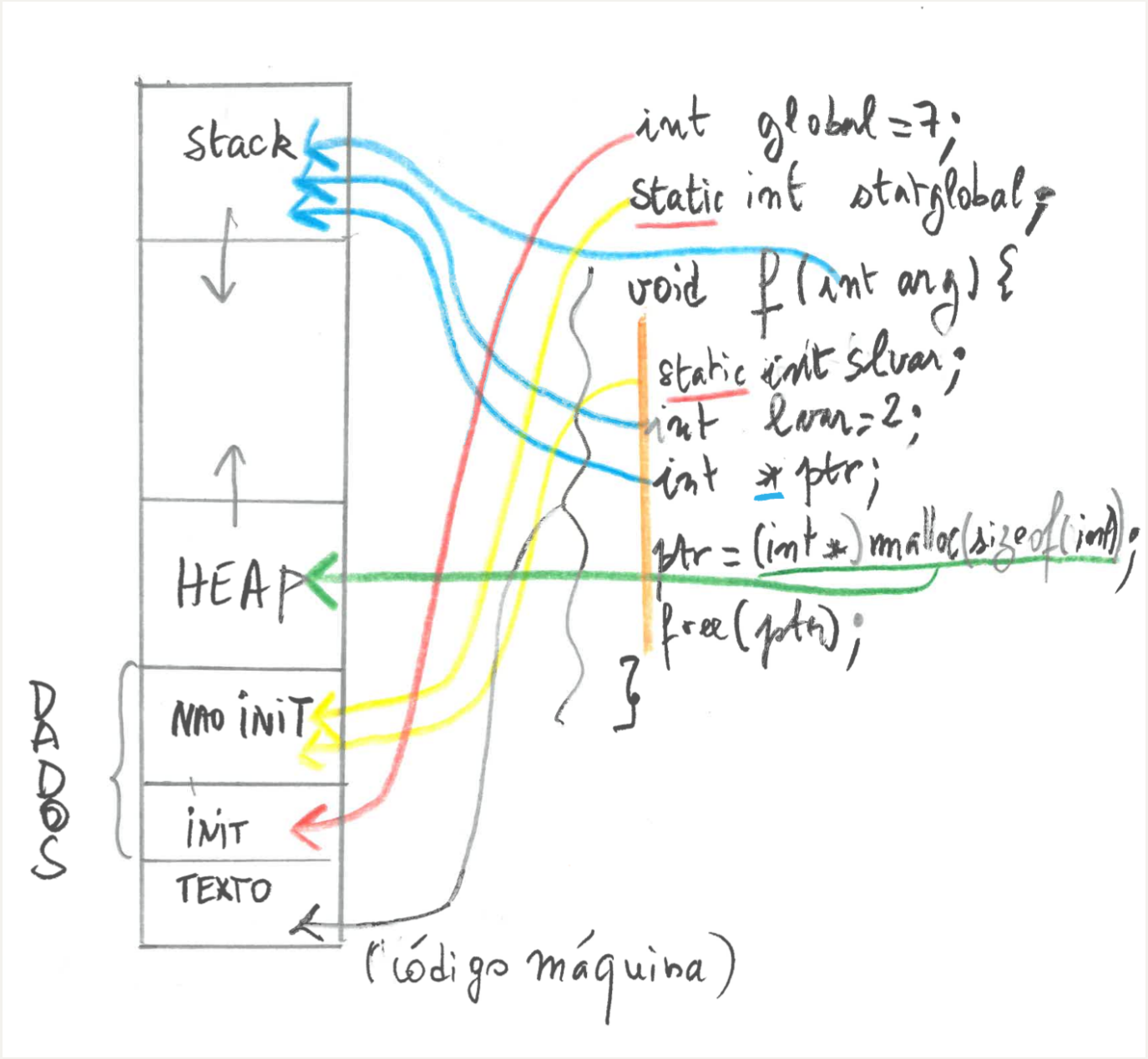
- **Armazena:**
    - Variáveis locais
    - Parâmetros de funções
    - Endereços de retorno (`return address`) (no fim da execução da presente função, onde ir? Onde está a próxima instrução por executar?)
- Cresce para endereços *inferiores* (para baixo na memória).
- Gerida automaticamente pelo compilador (não é o programador que tem de tratar dos detalhes da sua criação, gestão e afins).
- Pode causa um `Stack Overflow`. Esta situação ocorre se a pilha crescer muito, por exemplo, quando se loca um vetor de tamanho estático enorme, ou se houver recursões em número não razoável.

Exemplo:

```c
void funcao(int x) { // cópia "x" do parâmetro efetivo armazenado na pilha
    int local_var = 10;  // Armazenado na pilha
}
```

Em imagens.





Organização da Memória de uma *stack frame* na Stack

Vamos olhar mais em detalhe para a organização do modelo memória provocado por uma chamada de função.

Relembremos que quando uma função é chamada durante a execução de um programa, o sistema operativo aloca espaço na **call stack (pilha de chamadas)**, para armazenar todas as informações necessárias para sua execução. Essa organização é conhecida como **stack frame**.

Estrutura de um Stack Frame

Cada chamada de função cria um **stack frame** na pilha, contendo (entre outros) os seguintes elementos:

1. **Endereço de Retorno (`Return Address`)**
    - Guarda o endereço para onde a execução deve retornar após o término da função.
2. **Parâmetros da Função**
    - Valores passados para a função, armazenados na stack (para chamadas que não usam registradores).
3. **Espaço para Variáveis Locais**
    - As variáveis locais da função são armazenadas no stack frame.

Sem entrar nos detalhes, há mais algumas informações relevantes guardadas numa *stack frame*, mas não são relevantes para o propósito desta ficha.

Exemplo Prático

Vamos considerar o seguinte código:

```c
#include <stdio.h>

void funcao(int a, int b) {
    int x = a + b;  // Variável local
    printf("Soma: %d\n", x);
}

int main() {
     funcao(3, 4);
/**/ return 0;
}
```

Quando `funcao(3, 4)` é chamada, um novo **stack frame** é criado na pilha. A estrutura da memória fica assim:

```
+-------------------------+
| Stack Frame de main()   |
| (anterior na pilha)     |
+-------------------------+
| Endereço de Retorno     |  <- endereço para onde retornar
+-------------------------+     (para main() em /**/)
| Parâmetro `a = 3`       |
+-------------------------+
| Parâmetro `b = 4`       |
+-------------------------+ <-- Separação entre a zona dos
| Variável local `x`      |     parâmetros e das variáveis
+-------------------------+     locais.
| resto da Stack Frame de |
|      funcao()           |
+-------------------------+
```

Uma situação embaraçosa recorrente

Se pretendemos calcular num valor numa função (via uma variável local, por exemplo) e devolver este valor via o seu endereço, isso vai resultar num erro embaraçoso.

No fim da execução da função, a *stack frame* é eliminada da pilha (`pop`) e todo o contexto de memória local arquivado na stack frame é descartado. O valor de retorno é um endereço que agora é lixo (ou potencialmente algo de sensível , já que pertencerá a uma possível futura *stack frame*).

Um exemplo:

```
int* funcao(int x) {
  int t [100];
  for (i=0;i<100;i++) t[i]=i+x;
  return t;  // ERRO!!! quando a stack frame de funcao
             // é descartada o endereço devolvido não é válido
}
```

Como uma variável (local) é armazenada na *Stack* ou na *Heap* em C?

Em **C**, as variáveis podem ser armazenadas na **stack (pilha)** ou na **heap (zona de memória dinâmica)** dependendo de **como** e **onde** elas são declaradas.

Variáveis na Stack

Esta parte já sabemos. As variáveis **locais** e os **parâmetros de função** são armazenados na **stack frame**. Importa realçar que o compilador ***tem de saber*** que **tamanho** tem os dados que pretende ali colocar. É essencial para esta informação ser colocada na *stack frame*.

Como uma variável é colocada na stack?

- Quando uma função é chamada, uma **stack frame** é criado na pilha.
- O compilador reserva **automaticamente** espaço para as **variáveis locais** e o valor dos **parâmetros passados... por valor** dentro dessa *stack frame*.
- Quando a função termina, esse espaço é **automaticamente descartado**.

Variáveis na Heap

A **heap** é usada quando um valor precisa de **persistir após** a saída de uma função, ou quando o tamanho da variável em causa só é conhecido **em tempo de execução**.

Como um valor é colocada na heap?

- A alocação de memória na *heap* é feita pelo programador com `malloc()`, `calloc()` ou `realloc()`.
- A variável local que armazena o endereço do espaço memória alocado é, ela, arquivada na *stack frame*. A função pode retornar este endereço (o valor da variável local que é um apontador). A variável local é descartada à semelhança da stack frame que a contém, no fim da execução da função. No entanto o espaço alocado cujo endeço foi devolvido permanece ativo. Os valores na *heap* Sobrevi
- O programador precisa **liberar a memória manualmente** com `free()` no fim do uso do espaço alocado.

Exemplo

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

int *funcao(int v) {
  int *p = (int *)malloc(v*sizeof(int));
  // p é uma variável local, mas o espaço
  // alocado para o qual aponta, está na heap
  if (p == NULL) {
    printf("Erro ao alocar memória!\n");
    return NULL;
  }
  p[0] = 10;
  p[1] = 20;
  printf("Valores na heap: %d %d\n", *p, *(p + 1));
  return p;
}
```

```c
int main() {
  int *t = NULL;
  t = funcao(20);
  printf("Valores na heap: %d %d\n", *t, *(t + 1);
  free (t);
  return 0;
}
```

Explicação:

- A variável `p` é um **apontador armazenado na stack**, mas os valores apontados (`*p`) **estão na heap**.
- Os valores alocados na ***heap***, via uma variável local armazenada na ***stack frame***, sobrevivam ao fecho da *stack frame* e podem ser utilizada fora da função.
- A memória não é devolvida automaticamente, então **é necessário usar `free()`**.

Diferença Stack vs Heap

CARACTERÍSTICA	STACK (PILHA)	HEAP (MEMÓRIA DINÂMICA)
Alocação	Automática	Manual (`malloc()`, `calloc()`, `realloc()`)
Libertação	Automática quando a função retorna	Manual (`free()`)
Velocidade	Muito rápida	Mais lenta
Tamanho	Limitado (pode causar Stack Overflow)	Maior, depende da RAM

Problemas comuns relacionados com o modelo de memória

Relembramos aqui algumas das situações habituais na gestão de memória dinâmica e que conduzem a erros.

Uso de apontadors para Memória Inexistente

```
int *p;
*p = 42;  // ERRO: apontador não inicializado
```

Correção: adotar um estilo defensivo, nomeadamente inicializar sempre as declarações de apontadors e assegurar que apontam para uma zona válida de memória ante de uso.

```
int *p = NULL;
...
if (p) *p=42; // se p não for o apontador NULL então ...
...
p = (int *) malloc (sinzeof(int));
...
if (p) *p=42;
...
```

Uso de Memória Após `free()`

Também conhecido por `Use-After-Free`.

```
int *p = (int *)malloc(sizeof(int) * 10);
free(p);
p[0] = 42;  // ERRO: Acessando memória liberada
```

Correção: Após `free()`, seguir os mesmos cuidados que no caso anterior.

Fugas de memória

Os "famosos" ***memory leaks*** são ocasionados por falta da devida libertação após uso. Quando uma zona de memória deixa de ter uso, é necessário devolvê-la (*desaloca*-lá) para que possa ser reutilizada.

```
void leaky_function() {
    int *p = (int *)malloc(100 * sizeof(int)); // Alocação sem
free() antes do retorno da função
}
```

Neste exemplo, de cada vez que a função é chamada, é atribuído e inicializado (_ alocado_) um espaço memória de tamanho `100*sizeof(int)`. Este espaço não é devolvido. Logo após n chamadas da função, teremos `n*100*sizeof(int)` blocos de memória desperdiçados sem poderem ser usados por outros programas/funções.

Correção: Assegurar que a função `free()` é chamada sobre os recursos alocados anteriormente (com `malloc`, `calloc` ou `realloc`) para libera-los em fim de uso.

O famigerado `Segmentation Fault`

Estas três primeiras situações podem provocar o famoso erro de segmentação `segmentation fault`. Basicamente, esta situação ocorre quando é detectada um acesso ilegal à memória pelo programa executado. Esta deteção é na origem feita pelo *hardware*, nomeadamente pelo `MMU` (*Memory Managment Unit*) e é transmitida (na forma de uma *interrupção hardware*) à camada superior até ao sistema operativo. O sistema operativo recebe a interrupção e envia um sinal (o sinal `SIGSEGV`) ao *processo* (aqui, para simplificar, o programa executado) que causou a violação detectada. Se o programa não tratar do sinal enviado, o sistema operativo encerra o programa.

Neste trabalho dirigido, omitimos como programas podem tratar de sinais enviados pelo sistema operativo. Mas fica o conhecimento de que é possível um programa dialogar com o sistema operativo e um exemplo básico.

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

```c
#include <stdlib.h>
#include <signal.h> //biblioteca que oferece o serviço de gestão dos sinais/interupções

// Manipulador do sinal SIGSEGV
void sigsegv_handler(int signum) {
    printf("Erro: Segmentation Fault detectado (Sinal %d)!\n", signum);
    printf("Encerrando o programa de forma segura...\n");
    exit(EXIT_FAILURE);  // Finaliza o programa de maneira controlada
}

int main() {
    // Configura o manipulador de sinal para SIGSEGV
    signal(SIGSEGV, sigsegv_handler);

    printf("Programa iniciado. Vamos forçar um Segmentation Fault...\n");

    // Forçando um segmentation fault: acesso a um apontador NULL
    int *p = NULL;
    *p = 42;  // BAM!

    printf("Isso nunca será impresso!\n");

    return 0;
}
```

Terminamos esta secção com a informação de que ferramentas como `valgrind` podem ajudar a detectar e localizar tais erros de memória.

Para além das já conhecidas fugas de memória (*memory leak*) ou do problema do uso de apontadores após libertação (*aka*, *use-atfer-free*), assinalemos o seguinte problema: o uso um apontador para variável local (`Dangling Pointer`)

```c
int *funcao() {
    int x = 10;
    return &x;  // ERRO: Retorna endereço de variável que será destruída!
}
```

Problemas com a capacidade máxima da pilha de chamadas

Trata-se do famoso `Stack Overflow`. Essencialmente trata-se de situações em que o programa esgota o espaço disponível para a pilha de chamadas.

```c
void recursiva() {
  int p[10000]; // vetor "enorme" arquivado na pilha
  recursiva();  // Recursão infinita!
}
```

Correção: A correção não é simples. Primeiro tem-se de diagnosticar a razão do problema. A origem é recursividade infinita? Recursividade finita mas em demasia? É do tamanho exorbitante da *frame* de uma função?

Como vimos em aulas anteriores, se o problema for uma recursividade infinita, trata-se de um erro de programação. **É um bug!** É preciso corrigí-lo.

Se se trata de chamadas recursivas em demasia, poderá ser necessário transformar a função em causa e propor uma reformulação (usando recursividade terminal, ou reescrever o algoritmo e o código correspondente por forma a que usa ciclos - estas duas alternativas são em muito semelhantes).

Se for o último caso, podemos tratar de alocar dinamicamente as variáveis volumosas (e de as libertar com `free()` no fim).

Por fim, convém realçar que é possível indicar o tamanho da `call stack` aquando da execução de um programa:

```
ulimit -s 65536 && ./meu_programa
```

Explicação: esse comando disponível no **Linux/macOS** tem duas partes:

1. **`ulimit -s 65536`**
    - Define o tamanho máximo da stack para **65536 KB** (ou **64 MB**).
    - `-s` indica **stack size**.
2. **`&& ./meu_programa`**
    - **Executa `./meu_programa` somente se o comando anterior (`ulimit -s 65536`) for bem-sucedido.**
    - O programa `meu_programa` será executado **com uma `call stack` maior**.

Uma digressão sobre as diferentes formas de conversão de tipos

---

Em C, há diferentes formas de converter um tipo de dado para outro, conhecidas como **casting**. Este tutorial compara duas abordagens específicas e discute outras formas de conversão de tipos em C.

Comparação entre tipos de casting

Considere os seguintes exemplos:

Exemplo 1: Casting via apontador (Type Punning)

```
float f = 1.5f;
int x = *(int *)(&f);
```

Explicação:

- `&f` é o endereço da variável `f`.
- `(int *)` converte esse endereço para um apontador do tipo `int`.
- `*(int *)` acede à memória interpretando o valor como um `int`.
- **Este método não converte o valor, mas reinterpreta os bits da memória de `float` como `int`!**

Exemplo 2: Casting Explícito

```
float f = 1.5f;
int x = (int) f;
```

Explicação:

- `(int) f` converte o valor armazenado em `f` para um inteiro.
- **Aqui ocorre uma conversão real:** `1.5f` é arredondado para `1`.
- Esta conversão usa as regras de truncamento padrão de ponto flutuante para inteiro que exploramos na UC de **Programação Imperativa**.

Exemplo 3: Casting Implícita

```
float f = 3.9f;
int x = f;  // Implicit cast: x recebe 3 (truncado)
```

O compilador faz automaticamente a conversão de valores quando necessário e possível.

Exemplo 4: Casting via `union`

A definição de tipos `union` podem ser úteis para a manipulação de Bits. Vejam só!

```c
#include <stdio.h>
union FloatInt {
    float f;
    int i;
};

int main() {
    union FloatInt u;
    u.f = 1.5f;
    printf("Bits de 1.5 interpretados como int: %d\n", u.i);
    return 0;
}
```

Este código usa um valor de um tipo definido por `union` para **reinterpretar os bits de um `float` como um `int`**. Este exemplo é mais um exemplo da técnica ***type punning***, muito usada em manipulação de bits e sistemas embarcados (***embeded systems***).

De uma forma geral, percebemos que os tipos servem para enquadrar a forma como consideramos os valores concretos manipulados (os bits que formam os valores). Estas conversões, quando pouco cuidadosas, são perigosas.

O que significa converter um valor de um determinado tamanho ((por exemplo um valor de tipo float, de tamanhos de 4 bytes) para um char (de tamanho de 2 bytes)? É arriscado, sem não sabemos bem o que fazemos.

Relembremos que os tipos servem um propósito disciplinar: os valores de um determinado tipo respeitam as mesmas regras. É como arrumar a roupa nas gavetas do roupeiro do quarto. Se colocamos a roupa pela seu tipo em gavetas específicas, encontramos uma camisola de lã de forma muito rápido, está na gaveta das camisolas de lã. Sabemos também que qualquer roupa da gaveta das camisolas de lã é uma camisola de lã e como tal não a pomos na máquina de lavar num programa acima de 30 graus. Os tipos disciplinam, asseguram um tratamento uniforme e evitam erros.

Levamos o conceito a um extremo com a questão que segue.

O que acontece ao converter um apontador de função para um inteiro em C?

Primeiro, é possível. Mas para que servirá? Na prática, só para quem percebe. Para os comuns dos programadores mortais, não há cenários razoáveis para usar tal coisa.

Em C, fazer o **casting de um apontador de função para um inteiro** pode gerar **comportamento indefinido**. Basta perceber que os apontadores são representados em certas arquitecturas com 8 bytes enquanto os inteiros o são com 4. Vamos analisar passo a passo o que acontece e como evitar problemas.

Código Exemplo

```c
#include <stdio.h>

void f() {
    printf("Executando a função f()\n");
}

int main() {
    void (*g)() = f;  // g aponta para a função f
    int i = (int)(void *)g;  // Converte g para um inteiro

    printf("Endereço de f interpretado como inteiro: %d\n", i);

    return 0;
}
```

Explicação passo a passo:

1. Definição da Função `f()`
    - `f()` é uma função simples, sem parâmetros e sem retorno.
2. Definição do apontador de função `g`
    - `void (*g)()` define `g` como um apontador para uma função que retorna `void`.

- `g = f;` faz `g` apontar para `f()`, armazenando o endereço da função.

3. Casting do apontador de Função para um Inteiro (`int`)
    - `g` é um apondador de função, de `void` para `void`, que queremos converter para `int`. É inicializado com o endereço de `f` . O cast necessário para poder atribuir este valor para variável `i` é:

        ```
        int i = (int)(void *)g;
        ```

    - São duas conversões de tipos consecutivas no lugar de uma conversão direta! Porquê?

      Porque uma conversão directa desta natureza (nomeadamente `(int) g`) **pode causar um erro**, dependendo da arquitectura do computador que executa o programa.
    - A conversão para `(void *)` primeiro é uma prática mais segura, pois em algumas arquiteturas `void *` tem um formato adequado para armazenar endereços de função.

Exercícios

Exercício 1

1. Escreva um programa que defina uma função `soma()` que recebe dois inteiros, calcula sua soma e retorna o resultado.
2. Dentro da função `main()`, declare duas variáveis inteiras e passe seus valores para `soma()`.
3. Dentro de `soma()`, imprima os **endereços das variáveis locais** e dos parâmetros (usar o padrão `%p` para o printf).
4. Dentro de `main()`, imprima os **endereços das variáveis globais e locais** (usar o padrão `%p` para o printf).

Objetivo

- **Observar a organização do stack frame**.
- **Compreender onde os parâmetros e variáveis locais são armazenados**.

Exemplo de Saída Esperada

```
Endereço de a em soma(): 0x7ffeefbff618
Endereço de b em soma(): 0x7ffeefbff61c
Endereço de resultado em soma(): 0x7ffeefbff610

Endereço de var_local em main(): 0x7ffeefbff640
Endereço de var_global: 0x601048  (Segmento de Dados)
```

5. (opcional) Modifique a função `soma` de forma a que consiga modificar a variável `a` recorrendo exclusivamente à variável `b` e à aritmética de apontador. A função `soma` deverá mostrar os valores de `a` e de `b` no início e no fim da função, para testemunhar da mudança de valores.

Exercício 2

Manipulação de Heap vs. Stack

1. No `main()`, declare uma variável local `int a = 10;` e imprima seu endereço.
2. Aloque dinamicamente um inteiro (`malloc(sizeof(int))`), atribua-lhe um valor e imprima seu endereço.
3. Crie uma função `aloca_memoria()` que retorna um ponteiro para um inteiro alocado dinamicamente e compare o endereço retornado com o da variável local.

Objetivo

- Diferenciar `heap` e `stack`.
- Observar que a `heap` cresce num um espaço diferente da `stack`.

Exemplo de Saída Esperada

```
Endereço da variável local: 0x7ffeefbff648
Endereço do inteiro alocado na heap: 0x55555575c260
```

Exercício 3

Analisar um `Stack Overflow`

1. Escreva duas funções recursivas que causam um `Stack Overflow`.
    a. A primeira porque declara variáveis locais de grande tamanho (um vetor de inteiros de 10000 elementos, por exemplo), e faz chamadas para ela própria um bom milhar de vezes (controlada pelo parâmetro inteiro `n`). Se chamarmos este vetor local por `t`, a função poderá imprimir o endereço de `t` desta forma

    ```
    printf("Chamada %d - Endereço de t: %p\n", n, (void *)&t);`
    ```

    b. A segunda função padece de uma recursão infinita. Ela declara uma variável local inteira `x`, imprime o endereço de `x` e chama-se recursivamente sem uma guarda para o caso de paragem.

    ```
    printf("Chamada %d - Endereço de x: %p\n", n, (void *)&x);
    ```

2. Escreva dois programas C tais que o primeiro chama a primeira função e o segundo chama a segunda função. Verifique que estes programas terminam abruptamente com o erro `Segmentation Fault`.

Objetivo

- Observar que a `stack` tem um limite de memória.
- Entender como um `stack overflow` acontece.

Exemplo de Saída Esperada

```
Chamada 1 - Endereço de x: 0x7ffeefbff620
Chamada 2 - Endereço de x: 0x7ffeefbfe620
...
Segmentation fault (core dumped)
```

Exercício 4

Neste exercício vamos observar o comportamento de um apontador para uma variável local.

1. Escreva uma função `int* retorna_ponteiro()` que declara uma variável inteira local, atribui um valor e retorna o **endereço** dessa variável.
2. No `main()`, chame `retorna_ponteiro()`, armazene o ponteiro retornado e tente aceder o valor.
3. Imprima o endereço retornado e observe o que acontece. Recorde-se que o comportamento constatado é considerado **indefinido** no C padrão.

Objetivo

- Observar o problema do uso de ponteiros para variáveis locais (`dangling pointer`).
- Compreender que o `stack frame` é destruído após a função terminar.

Exercício 5

Vamos comparar os endereços localizados na `heap` e na `stack`

1. No `main()`, crie três variáveis locais e imprima seus endereços.
2. Aloque dinamicamente três inteiros (`malloc(sizeof(int))`) e imprima seus endereços.

3. Compare os endereços da **`stack`** e da **`heap`**. O que observe? É consistente com a organização da memória descrita mais acima?

Objetivo

- Observar que variáveis locais são alocadas em endereços próximos (e que a `stack` "cresce" para baixo).
- Observar que a heap cresce para cima (os endereços alocados crescem conforme novas alocações).

Iteradores para vetores dinâmicos

Nesta secção vamos dar uso programático tanto às estruturas genéricas como aos apontadores de função.

Interessamo-nos aqui por novos serviços que queremos juntar aos serviços oferecidos aos vetores dinâmicos que implementamos no trabalho dirigido anterior.

Como somar 2 a todos os elementos de um `intdynvec`? Como multiplicar todos os elemento de um `intdynvec` por 3? Como inverter um `intdynvec`? Como criar um `intdynvec` que agrupa todos os valores pares de um outro `intdynvec`? Como detectar se um `intdynvec` está ordenado de forma crescente? Como detectar se um `intdynvec` só contém valores acima de 10? Como somar todos os valores de um `intdynvec`? Como fazer a média de todos so valores de um `intdynvec`?

Mais interessante ainda, como adaptar estas questões para um `dynvec` que virtualmente pode contar elementos de qualquer tipo?

One map to map them all

Nesta altura do nosso conhecimento programação `C`, nenhuma destas questões sobre os `intdynvec` oferece dificuldades particulares. Mas arriscamo-nos a duplicar muito código. Por exemplos somar 2 a todos os elementos de um `intdynvec` ou multiplicar todos os elemento de um `intdynvec` por 3 são duas operações algoritmicamente muito

semelhantes:

- `Para todo o elemento x do intdynvec t, atualizar com x + 2`
- `Para todo o elemento x do intdynvec t, atualizar com x * 3`

Como já sabemos, um programador é um preguiçoso eficiente. Ver código duplicado provoca-lhe uma queda de tensão. Por isso, prefere perder duas horas otimizando um processo de meia hora.

A sorte deste programador é a linguagem `C` permitir-lhe fazer tal tarefa em tempo record!

Se observamos as duas linhas de pseudo-código acima reproduzidas, reparamos que podemos reescrever-las como:

`Para todo o elemento x do intdynvec t, atualizar com processo(x)`

Em que `processo` é a função $x \mapsto x + 2$ ou a função $x \mapsto x \times 3$, conforme o caso.

Mas será possível escrever uma só vez o programa `C` subjacente a esta análise e especializar processo para os diferentes casos de interesse para o programador.

Sim! Basta considerar `processo` como um parâmetro do programa. Em `C`, tal se faz com apontadores de função.

Esquematicamente queremos construir a função seguinte:

$$(t, processo) \mapsto \forall x \in t,\ x \leftarrow processo(x) \tag{1}$$

Mas o que é esta fórmula?

É a função que a um dynvec `t` e a uma função `processo` associa o cálculo que consiste em atualizar todo o elemento `x` de `t` por `processo(x)`.

Explicamos este processo de mapeamento (`map`) numa imagem onde `f` tem o papel de `processo` e `t`é o `dynvec` considerado.

Em C, podemos implementar tal fórmula por uma função que vamos chamar `dynvec_map` que aceita uma referência `v` para um dynvec e uma função genérica `processo` dada por referência. Como tal `processo` tem por assinatura `void (*)(void *)`. Porquê? Para poder ter representar **qualquer função** que pretendemos aplicar a um `dynvec`.

A função `dynvec_map` procede por um ciclo que vai do primeiro elemento do vetor interno até ao ultimo e aplica a função `processo`. Simples.

Implementação

```c
void dynvec_map(dynvec *v, void (*processo)(void *)) {
    if (!v || !processo) return;
    for (size_t i = 0; i < v->length; i++) {
        processo((char *)v->data + (i * v->elem_size));
    }
}
```

- `dynvec_map` recebe um ponteiro para um `dynvec` (`v`) e um ponteiro para uma função (`processo`) que recebe um elemento (`void *`) e o processa.
- Verifica se o `dynvec` e a função são válidos. Se `v == NULL` ou `processo == NULL`, a função retorna imediatamente (evita acessos inválidos à memória).
- Percorre todos os elementos do vetor dinâmico (`dynvec`).
    - Converte `data` para `char *`, pois em `C` **não podemos utilizar `void *` para fazer aritmética de apontadores**. Porquê `(char *)`? Porque `char` tem por tamanho **1 byte**, logo a unidade. Assim podemos calcular **exatamente** onde cada elemento começa.
    - Calcula a posição do elemento `i` multiplicando `i * v->elem_size`. O posicionamento é realizado por aritmética de apontadores.
    - Chama `processo()` passando o endereço do elemento.

Exemplos de uso

Aplicar a função `print_int` a todos os elemento de um `dynvec` de inteiros.

A função `print_int` recebe um **ponteiro genérico (`void *`)**, converte (`cast`) o ponteiro para `int *` e acessa seu valor com `*(int *)elem` e imprime o valor inteiro seguido de um espaço (`%d`).

```c
#include <stdio.h>
#include "dynvec.h"

void print_int(void *elem) {
    printf("%d ", *(int *)elem);
}
```

```c
int main() {
    // criar para criar um dynvec que armazena inteiros
    // (sizeof(int) especifica o tamanho do elemento)
    dynvec *v = dynvec_create(sizeof(int));

    // juntamos os valores 10, 20 e 30 no dynvec
    int x = 10, y = 20, z = 30;
    dynvec_push(v, &x);
    dynvec_push(v, &y);
    dynvec_push(v, &z);

    // procedemos à aplicação de print_int
    // a todos os elemento do dynvec v
    dynvec_map(v, print_int);

    dynvec_free(v);
    return 0;
}
```

```
% ./test_map
10 20 30
```

Duplicar todos os elementos de um `dynvec` de inteiros. Para tal, usamos a função `double_value`. A função recebe

- **`void_value`** → O nome da função.
- **`void *elem`** → Recebe um **ponteiro genérico** (`void *`), pois a função pode ser usada em estruturas genéricas como `dynvec_map`.
- **`(int \*)elem`** → Converte (`cast`) `elem` para um ponteiro para `int`.
- **`\* (int \*)elem`** → Acessa o valor do inteiro armazenado no endereço.
- **Multiplica esse valor por `2` (`\*= 2`)** e armazena o resultado no mesmo local.

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

```c
#include "dynvec.h"

// Função para dobrar o valor de um número inteiro
void double_value(void *elem) {
    *(int *)elem *= 2;
}

// Função para imprimir um inteiro
void print_int(void *elem) {
    printf("%d ", *(int *)elem);
}

int main() {
    dynvec *v = dynvec_create(sizeof(int));

    int nums[] = {1, 2, 3, 4, 5};
    for (size_t i = 0; i < 5; i++) {
        dynvec_push(v, &nums[i]);
    }

    printf("Antes de map:\n");
    dynvec_map(v, print_int);
    printf("\n");

    // Aplicar dynvec_map para dobrar os valores
    dynvec_map(v, double_value);

    printf("Depois de map:\n");
    dynvec_map(v, print_int);
    printf("\n");

    dynvec_free(v);
    return 0;
}
```

```
Antes de map:
1 2 3 4 5
Depois de map:
2 4 6 8 10
```

Qual é o interesse de tal função?

Participa do princípio da abstração funcional que já descrevemos em ocasiões anteriores.

Primeiro, permite uma abstração da forma como implementamos os `dynvec`. Se mudamos a representação dos `dynvec`, basta adaptar o código da função `dynvec_map` sem por isso alterar a interface com os utilizadores da biblioteca. Para todos os efeitos e para os utilizadores, `dynvec_map` não mudou.

Segundo, e não menos importante, a abstração funcional permite ao programador pensar em termos algorítmicos sem considerar os detalhes da implementação. A função `map` sobre uma coleção de valores é uma função bem clássica que representa um **processo algorítmico bem estabelecido**, quer seja sobre um vetor, uma árvore, um grafo, um conjunto etc. Pouco importa os seus detalhes de implementação que foram fixados no momento da sua programação. Um programador pode esquecer-los e pensar alto nível sobre o problema que pretende resolver.

Exercício

Implementação de Funções para um Vetor Dinâmico Genérico (`dynvec`)

Objetivo

O objetivo deste exercício é implementar funções avançadas para um **vetor dinâmico genérico (`dynvec`)** em C. As funções permitirão a manipulação e a busca de elementos usando ponteiros para funções, tornando o `dynvec` altamente flexível e reutilizável.

Descrição

Você deve implementar as seguintes funções na biblioteca `dynvec.c`, garantindo que sejam compatíveis com a estrutura **genérica** de `dynvec` definida no arquivo de cabeçalho `dynvec.h`.

Cada função deve ser implementada de forma a suportar **qualquer tipo de dado**, utilizando `void *` e operadores de memória apropriados (`memcpy`, `memcmp`, etc.).

1. Funções de Busca

1.1. `dynvec_contains`

```
bool dynvec_contains(dynvec *v, void *elem, int (*cmp)(const void
*, const void *));
```

- **Objetivo**: Retorna `true` se `elem` existir no vetor `v`, `false` caso contrário. Para constatar que o elemento `elem` está no vetor, é necessário usar a função `cmp` passada por referência. Porquê? Porque não sabemos que tipo de elemento está no dynvec, com tal não é certo que a igualdade `==` consiga funcionar (pense no caso de um `dynvec` de uma estrutura complexa, já não se pode comparar dois elementos de forma tão simples com `==`).
- **Parâmetros**:
    - `v`: Ponteiro para um `dynvec`.
    - `elem`: Ponteiro para o elemento procurado.
    - `cmp`: Função de comparação.

1.2. `dynvec_index`

```
size_t dynvec_index(dynvec *v, void *elem, int (*cmp)(const void *,
const void *));
```

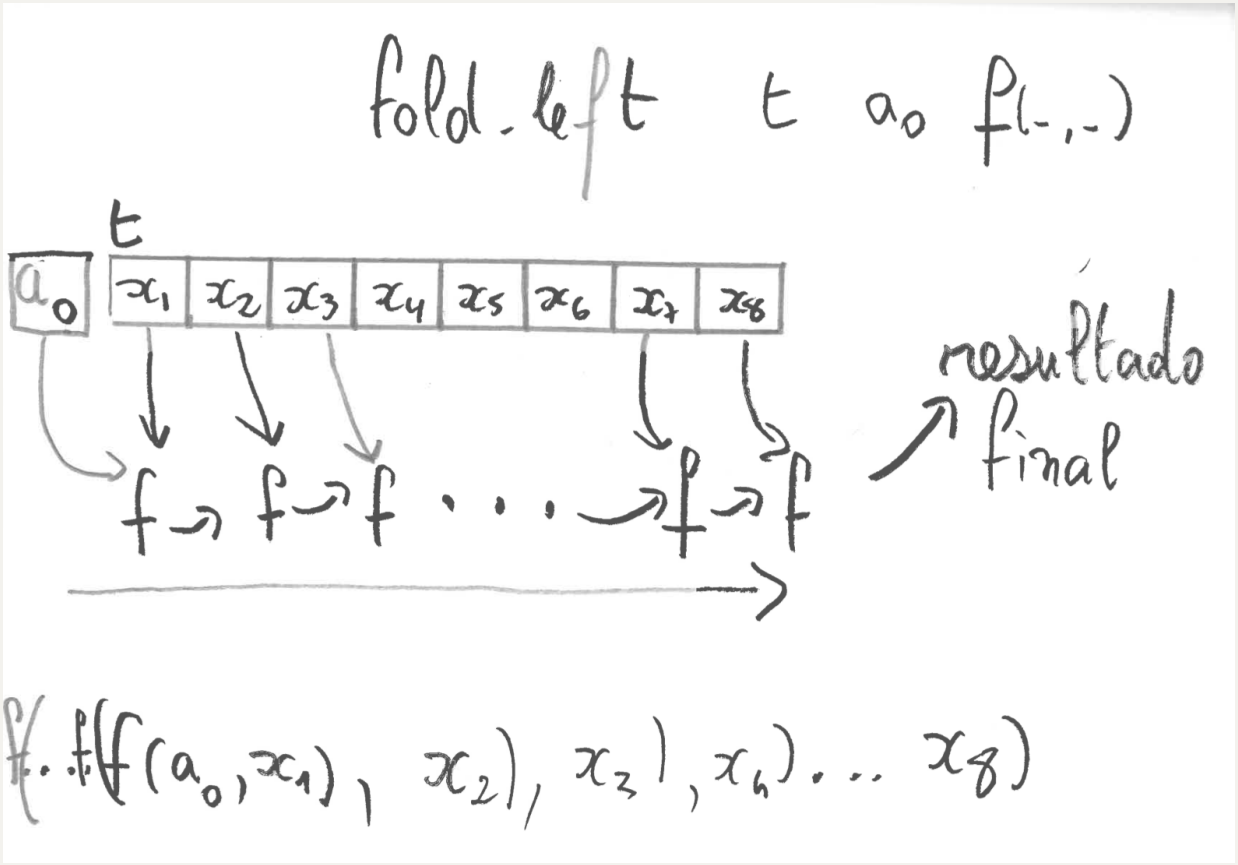
- **Objetivo**: Retorna o índice do primeiro elemento igual a `elem`, ou `-1` se não encontrado. Aqui também devemos recorrer a uma função de comparação `cmp` passada por referência.

2. Iteradores

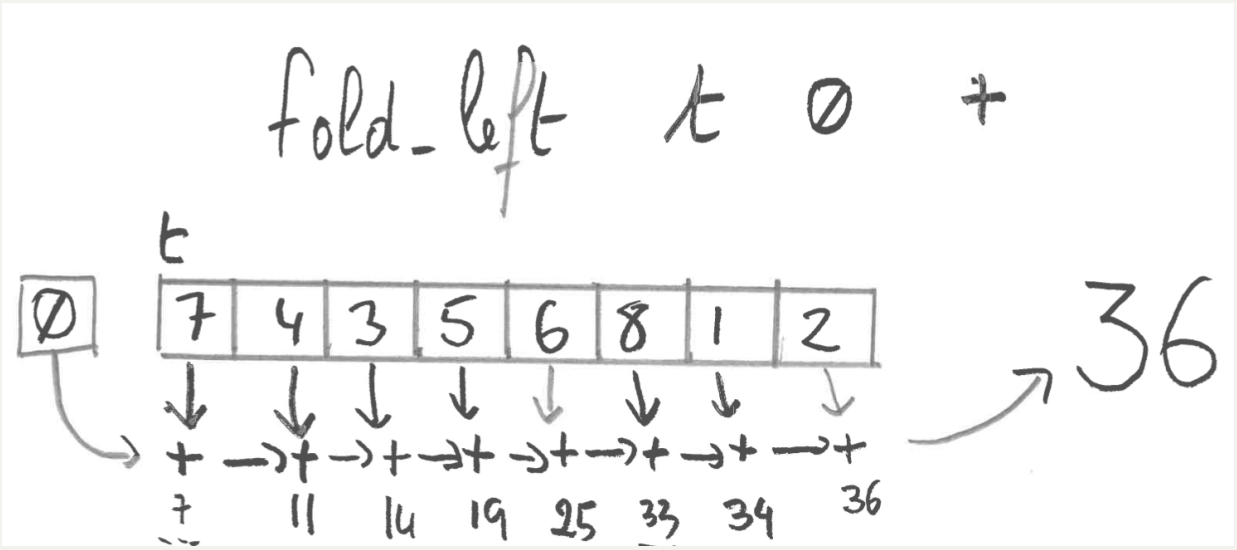
2.1. `dynvec_fold_left`

```
void dynvec_fold_left(dynvec *v, void *acc, void (*func)(void *acc,
void *elem));
```

- **Objetivo**: Reduz todos os elementos do vetor a um único valor acumulado (`acc`).



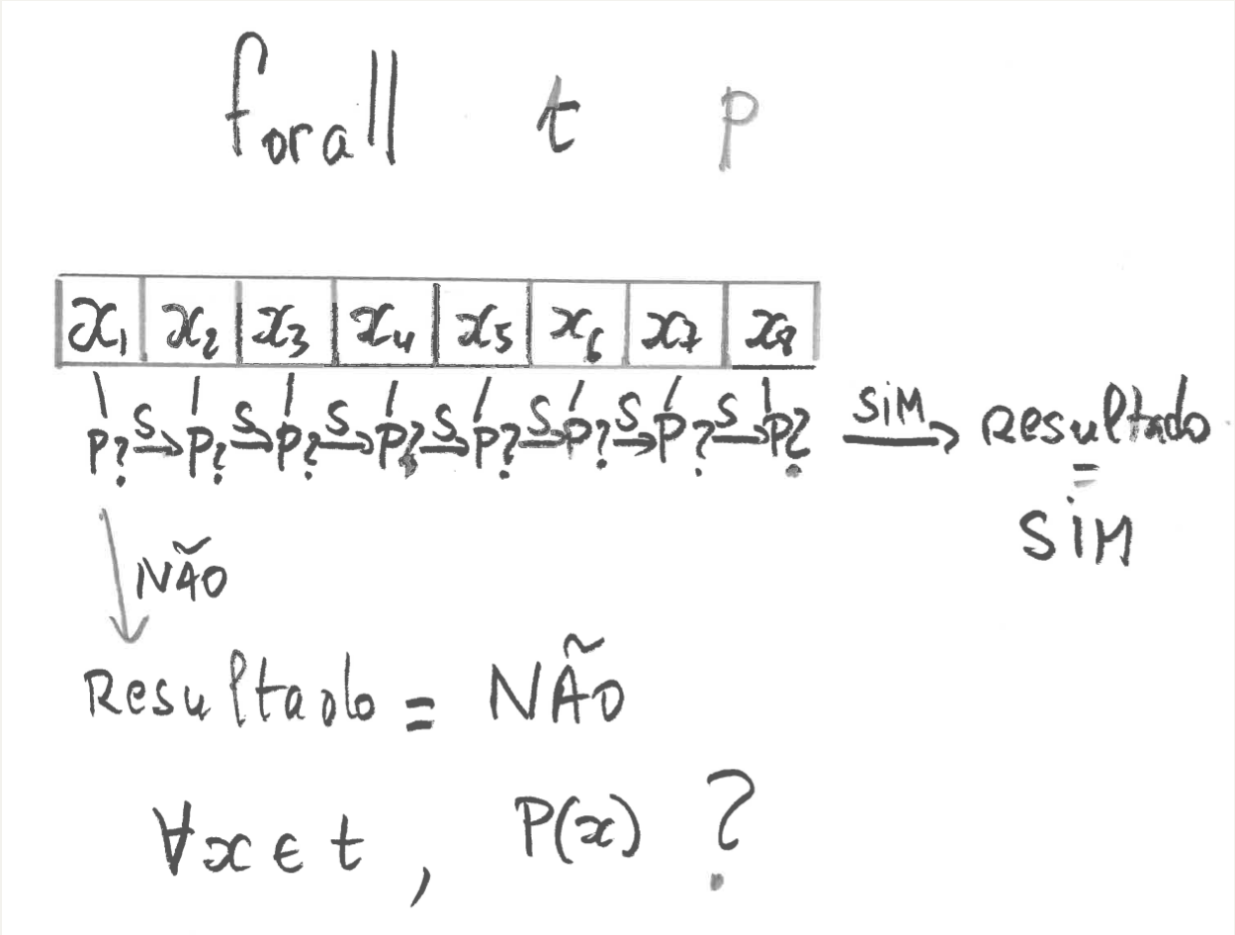
Um exemplo gráfico: a soma de um `dynvec` de inteiros



2.2. `dynvec_forall`

```
bool dynvec_forall(dynvec *v, bool (*predicate)(void *));
```

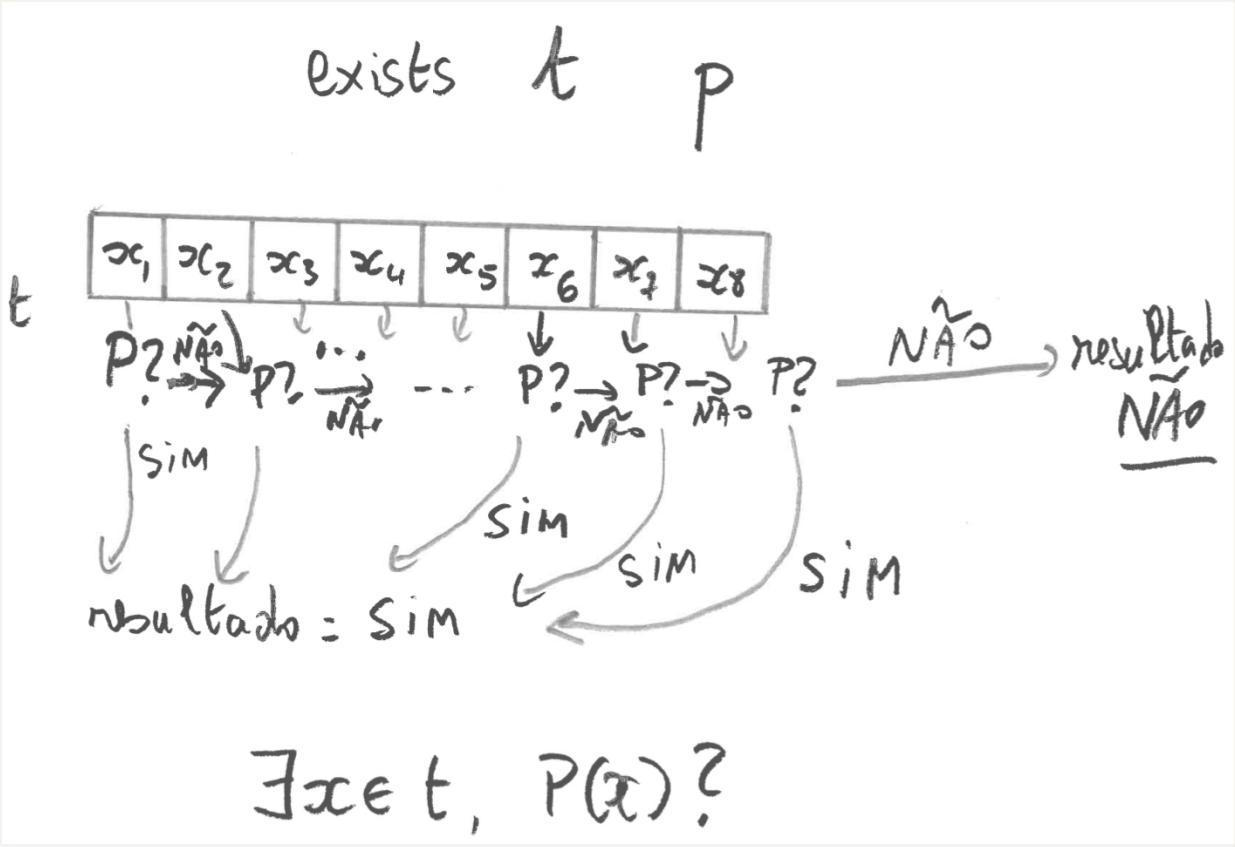
- **Objetivo**: Retorna `true` se **todos os elementos** do vetor satisfizerem `predicate`, `false` caso contrário.



2.3. `dynvec_exists`

```
bool dynvec_exists(dynvec *v, bool (*predicate)(void *));
```

- **Objetivo**: Retorna `true` se **pelo menos um** elemento satisfizer `predicate`, `false` caso contrário.



2.4. `dynvec_exists_index`

```
size_t dynvec_exists_index(dynvec *v, bool (*predicate)(void *));
```

- **Objetivo**: Retorna o índice do primeiro elemento que satisfaz `predicate`, ou `-1` se nenhum for encontrado.

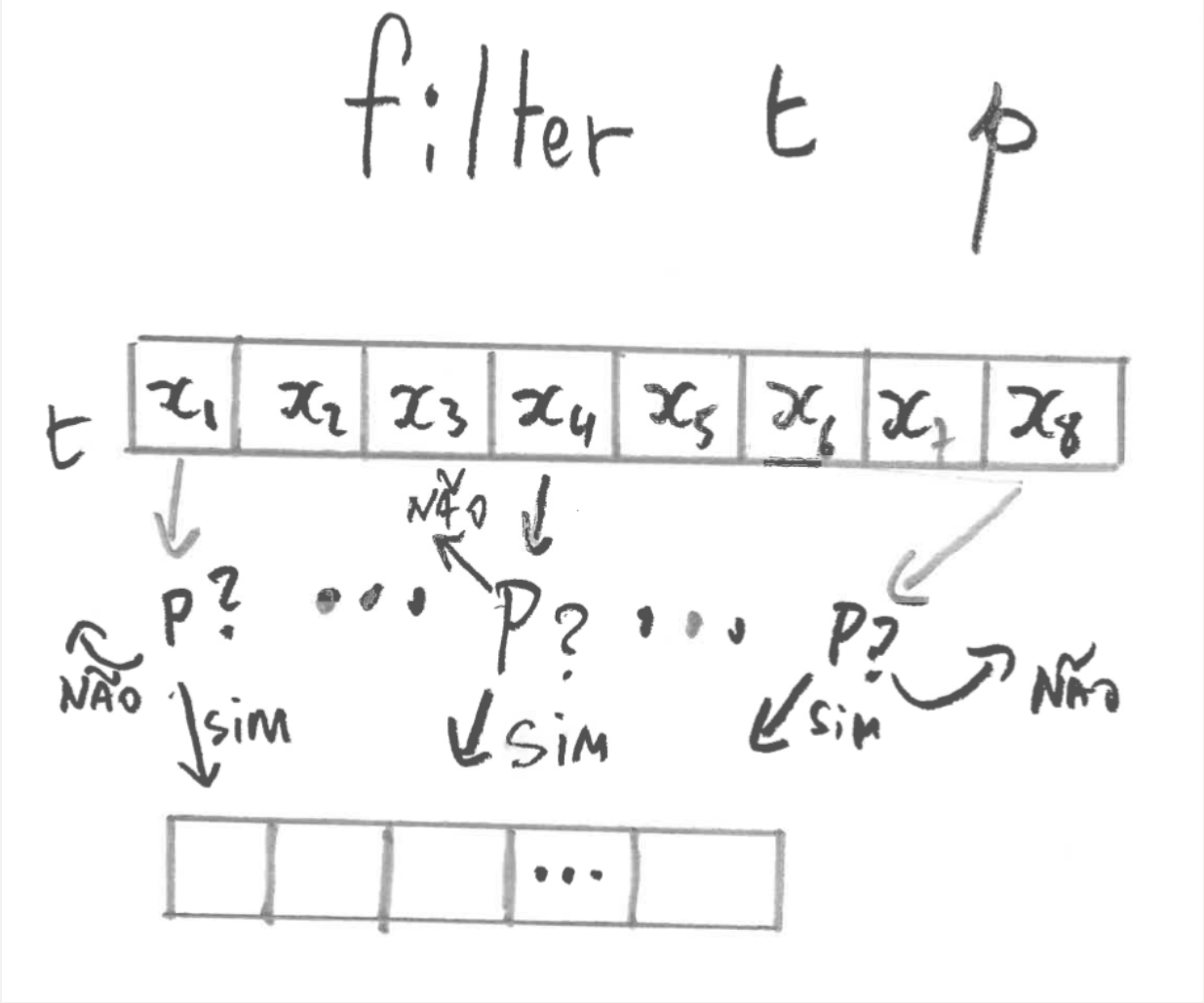
---

3. Filtragem

3.1. `dynvec_filter`

```
dynvec *dynvec_filter(dynvec *v, bool (*predicate)(void *));
```

- **Objetivo**: Cria um novo `dynvec` contendo apenas os elementos para os quais `predicate` retorna `true`.



Critérios a tomar em conta:

- **Correção**: As funções devem operar corretamente para qualquer tipo de dado.
- **Uso correto de ponteiros**: Especialmente o uso de `void *` e `char *`.
- **Eficiência**: O código deve evitar cópias desnecessárias e operações custosas.
- **Segurança de memória**: Nenhuma das funções deve acessar memória inválida ou causar vazamento.

Pesquisa Linear e Pesquisa Binária em C

Vamos agora explorar o tema da procura de valores num vector. É um algoritmo fundamental que pertence à caixa de ferramentas de qualquer programador sério.

Há duas técnicas clássicas que se aplicam em cenários diferentes. A pesquisa **linear** e a pesquisa **binária** (dita também **dicotómica**).

De forma geral estes processos de pesquisa são usados para saber se um dado valor está, ou não, numa coleção de elementos (quer seja esse conjunto codificado na forma de um vector ou noutro tipo de *contentor*) .

$$Dado\ um\ x\ de\ A,\ será\ x \in T_A? \tag{2}$$

Sendo A o tipo de dado a que x pertence (e.g. `int`) e T_A uma coleção de elementos de tipo A (e.g um vetor de elementos de tipo A). O valor procurado x é commumente designado de **chave**.

A resposta pode ser binária (sim, ou não), ou então ser o índice do valor procurado na dita coleção.

Vamos no texto que segue assumir que vamos pesquisar em vetores.

Pesquisa Linear

Conceito

A **pesquisa linear** assume a coleção de elementos como estando organizada em sequência sem demais informação sobre o seu conteúdo. Como tal percorre a coleção **elemento por elemento** até encontrar o valor desejado ou esgotar a coleção.

Implementação para Vetor de Inteiros

```
#include <stdio.h>
// arr: vector dos elementos
// tamanho: numero dos elementos no vector arr
```

```c
// chave: o valor por procurar no vector arr
int pesquisa_linear(int arr[], int tamanho, int chave) {
    // para todos os elementos arr[i] de arr,
    // do primeiro ao último, verificar se
    // arr[i] é igual a chave
    for (int i = 0; i < tamanho; i++) {
        if (arr[i] == chave) {
            return i; // Retorna o índice do elemento encontrado
        }
    }
    return -1; // Retorna -1 se não encontrar
}

int main() {
    int numeros[] = {10, 20, 30, 40, 50};
    int chave = 30;
    int tamanho = sizeof(numeros) / sizeof(numeros[0]);

    int resultado = pesquisa_linear(numeros, tamanho, chave);
    if (resultado != -1) {
        printf("Elemento encontrado no índice %d\n", resultado);
    } else {
        printf("Elemento não encontrado\n");
    }
    return 0;
}
```

Complexidade

Nesta secção vamos abordar considerações de eficiência de forma informal. Verão mais em detalhes estes tópicos em unidades curriculares que seguem.

- **Melhor caso:** Em que situações esta pesquisa é ótimo? Quando não tem nada por fazer! Este caso é o caso em que a chave por procurar está na primeira posição. A resposta é então dada imediatamente, **em tempo constante**. Dizemos que no melhor caso o cálculo é feito em **O(1)** (pronunciado *"Oh de um"*).
- **Pior caso:** Em que situação esta pesquisa dá a sua pior performance? Em que situação o maior número de cálculos são realizados?

Quando a chave não está na coleção ou é encontrada na ultima posição. Neste caso, todos os elementos da coleção são vistoriados, sendo a conclusão conhecida só no fim. Digamos que a coleção tem n elementos. A pesquisa vai realizar neste caso n comparações (`arr[i] == chave`). Ou seja, esta pesquisa vai realizar um número total de operação comparável ao número de comparações ou seja na **ordem de** nDizemos que no pior caso o cálculo é feto em **O(n)**, em **tempo linear**.

- **Caso médio:** E num caso qualquer que não seja exclusivamente extremo (nem sempre melhor, nem sempre pior)? Nos casos todos por considerar, há duas classes de situação: a configuração em que a chave não está na coleção e a configuração contrária em que a chave está na coleção. Todos os casos na primeira configuração são de complexidade linear, como já sabemos. Os casos da segunda configuração são os casos em que a chave está na coleção, numa posição arbitrária. Se está na primeira posição custa na ordem de 1, se estiver na posição 2, custa na ordem de 2, se estiver na posição 3, custa na ordem de 3, etc. até considerar o caso n. Se somarmos os casos todos para fazer a média desta configuração, temos ($\frac{1+2+3...+n}{n}$), ou seja $\frac{\frac{n\times(n+1)}{2}}{n}$, ou ainda $\frac{n\times(n+1)}{2n}$. O que dá $\frac{n+1}{2}$, uma expressão linear. Em suma e em média, o tempo de cálculo da pesquisa linear é ... **linear**.

Exercício

1. Modifique a função `pesquisa_linear` para contar o número de comparações feitas antes de encontrar o elemento.
2. Adapte a pesquisa linear para retornar um array de índices quando há múltiplas ocorrências do valor.
3. Nesta última variante, como avalia o desempenho no melhor, no pior e no caso médio?

Pesquisa Binária

Conceito

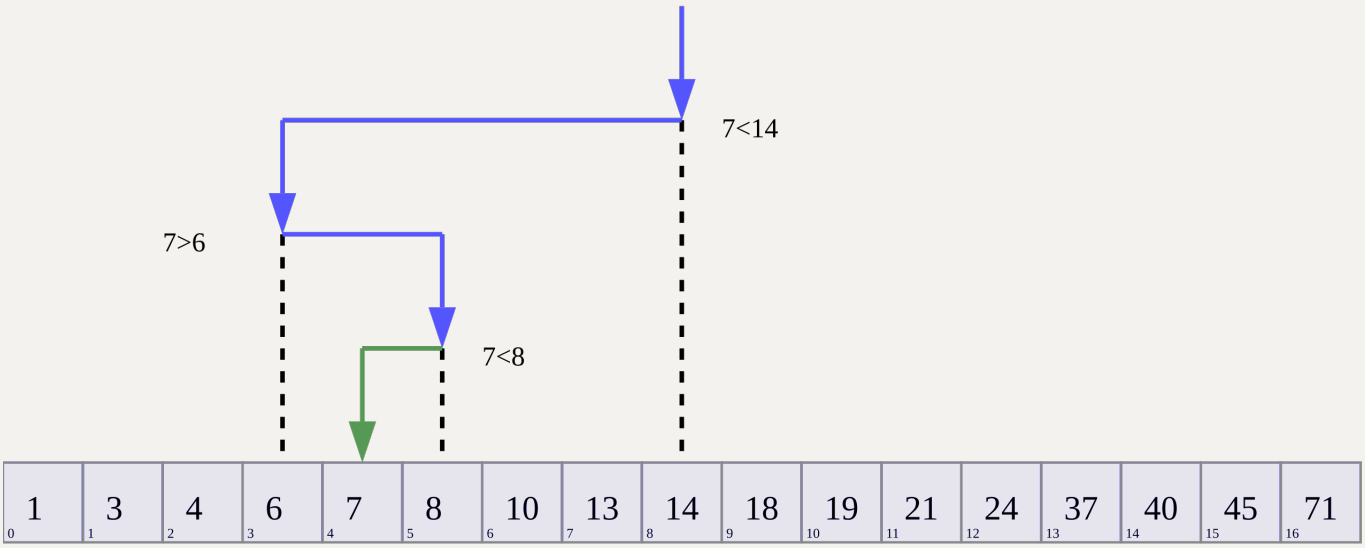
Se a coleção considerada para a pesquisa tiver **mais estrutura** que no caso geral, então podemos aproveitar este ganho de estrutura para ser mais eficiente na procura de elementos. É precisamente o que fazemos com a pesquisa binária: se a coleção considerada estiver ordenada, então podemos ganhar com isso!

A **pesquisa binária** que aqui introduzimos **só funciona, por princípio, em vetores ordenados**. O princípio de base é semelhante o tal jogo do "adivinha que número escolhi de 1 a 100". A melhor estratégia é escolher o numero no meio dos candidatos possíveis. Ou seja, no início, começar por 50. Se a resposta é "acima", restamos escolher ente 51 e 100. Aqui convém escolher 75, etc. Se a resposta for "abaixo", então convém escolher entre 1 e 49. Finalmente, a terceira resposta possível é a tão aguardada "acertaste"! Neste processo, sei que vou ganhar porque a cada tentativa errada, o intervalo que resta para adivinhar é cada vez menor, aliás divide-se este intervalo por dois. Logo haverei de ficar só com um valor na mão. E ali ou ganho com este valor ou o oponente fez batota!

De volta ao problema da **pesquisa binária** numa **coleção ordenada**, o algoritmo de pesquisa binária procede por divisões do vetor ao meio do intervalo que ausculta, repetidamente até encontrar o elemento ou determinar que ele não está presente.

Como Funciona?

1. Definir dois limites: **esquerda** (início do intervalo do vetor onde nos importa procurar) e **direita** (fim do intervalo do vetor que nos importa analisar).
2. Encontrar o valor no **meio** deste intervalo do vetor.
3. Comparar o valor do meio com a chave:
    - **Se for igual**, retorna o índice.
    - **Se for menor**, busca na metade direita.
    - **Se for maior**, busca na metade esquerda.
4. Repetir até encontrar o elemento ou esgotar o vetor (quando o intervalo que resta fica vazio).



(Procurar 7. Fonte: wikipedia)

Implementação para Vetor de Inteiros

```c
#include <stdio.h>

int pesquisa_binaria(int arr[], int esquerda, int direita, int
chave) {
    while (esquerda <= direita) {
      // calcular o indice no meio, a média, sem ariscar
      // um integer overflow!
        int meio = esquerda + (direita - esquerda) / 2;

        if (arr[meio] == chave) {
            return meio; // encontramos!
        }
        if (arr[meio] < chave) {
            esquerda = meio + 1; // ajustar o intervalo,
                                 // olhando para a direita
        } else {
            direita = meio - 1;  // ajustar o intervalo,
                                 // olhando para a esquerda
        }
    }
```

```
    return -1;
}

int main() {
    int numeros[] = {10, 20, 30, 40, 50, 60, 70};
    int chave = 30;
    int tamanho = sizeof(numeros) / sizeof(numeros[0]);

    int resultado = pesquisa_binaria(numeros, 0, tamanho - 1,
chave);
    if (resultado != -1) {
        printf("Elemento encontrado no índice %d\n", resultado);
    } else {
        printf("Elemento não encontrado\n");
    }
    return 0;
}
```

Complexidade

Abordar a performance deste método de forma informal sem atraiçoar a precisão do argumento é complicado. Vamos só dar as linhas gerais que justificam os valores que vamos aqui anunciar.

- **Melhor caso:** é o caso em que a chave está no meio. Neste caso encontrar a chave é imediato. A performance é *constante*, independentemente do tamanho da coleção, ou seja na ordem de **O(1)**.
- **Pior caso:** é o caso em que é preciso chegar ao intervalo vazio ou então ao intervalo unitário para terminar. Ou seja, é o caso em que a chave não está na coleção ou então está na última tentativa possível. Para poder quantificar estes cenários, é preciso perceber que a cada escolha negativa (a chave não está no índice do meio), o espaço de procura é dividido por 2. Após 3 tentativas (ou seja, 3 comparações), o espaço de procura foi dividido por 4 (i.e. 2^{3-1}). Em suma, o número de comparações está na ordem de **O(log n)**. Adivinhar um valor entre 1 e 100 leva no máximo, por dicotomia, $\lceil log_2(100) \rceil = 7$ tentativas. Procurar uma chave numa coleção ordenada de 100 elementos custará a volta de 7 comparações.

- **Espaço adicional:** O algoritmo de pesquisa binária iterativa utiliza uma quantidade **constante** de memória, ou seja na ordem de **O(1)**. A versão recursiva não terminal usa memória na ordem **O(log n)** por causa da pilha de chamadas (e.g. a quantidade de *stack frames* criadas). Obviamente, a versão recursiva terminal assemelha-se à versão iterativa no que diz respeito ao consumo de memória, ou seja, é constante, é **O(1)**.

Exemplo de Execução

Suponha que `arr[] = {10, 20, 30, 40, 50, 60, 70}` e `chave = 30`:

ITERAÇÃO	ESQUERDA	DIREITA	MEIO	`ARR[MEIO]`	COMPARAÇÃO
1	0	6	3	40	Não encontrado
2	0	2	1	20	Não encontrado
3	3	3	3	30	Encontrado 🎯

Neste caso, a procura terminou em apenas 3 iterações!

Uma anedota digna de se conhecer!

A primeira publicação do algoritmo de pesquisa binária numa forma completa teve lugar em **1946**, numa sebenta da autoria de John Mauchly, embora o método já era informalmente conhecido desde a Babilónia antiga.

Em **1957** William Peterson estudou mais em detalhe (isto é, formalmente) as propriedades da pesquisa binária e alertou para fragilidades das concretizações existentes deste algoritmo.

Em **1962**, no seu famoso livro **The Art Of Compter Programming** (conhecido como TAOCP), a bíblia de todo o programador corajoso e exímio (também conhecido como o livro que muitos conhecem e que muito poucos leram. O Bill Gates dizia deste livro que quem o tinha realmente lido teria contrato laboral na empresa dele), o Donald Knuth explica que apesar do algoritmo ser simples na sua formulação, era espantosamente difícil acerta numa boa implementação. Por boa entendia ele **correcta** (que faz em todos as situações o que é esperado que ele faça) e **eficiente**. É, aliás, nesta obra que podemos

encontrar o que se pode reconhecer como a primeira implementação correcta e eficiente do algoritmo de pesquisa binária.

No excelente livro **Programming Pearls** de **1986** (1^{ra} edição, 2^{da} edição do 2000), Jon Bentley reitera as armadilhas que um algoritmo como a pesquisa binária detém, propondo uma explicação concisa e clara da situação como as respectivas soluções.

E mesmo assim, em **2006**. um bug embaraçoso foi detectado na implementação da pesquisa binária da biblioteca padrão de Java. Esteve presente nesta biblioteca por quase uma década! Cf. **Joshua Bloch, Google Research Blog, "Nearly All Binary Searches and Mergesorts are Broken"**.

Em 2006, tinham passado 60 anos desde a definição completa do algoritmo, e tinham passado 44 anos desde que se conheciam as armadilhas da implementação e os seus respetivos correctivos!

Resumidamente o bug em questão é

```
...
  int meio = (esquerda + direita) / 2;
  int val = arr[meio];
...
```

O cálculo do índice meio pode exceder a capacidade do tipo int: ***integer overflow***.

Logo provocar um acesso fora do âmbito do vector (***array out of bound***)

A forma segura para calcular este índice é:

```
int meio = esquerda + (direita - esquerda) / 2;
```

Ou seja, somar o indice à esquerda metade da distância entre o indice esquerdo e o indice direito. O cálculo desta distância e a sua soma não tem forma de provocar um *overflow*.

O contexto da descoberta deste bug na biblioteca Java é interessante. Este bug esteve na biblioteca alguns anos sem ser detetado, em parte porque nos anos 2000 as memórias a que os computadores tinham acesso eram modestas (a luz do que temos hoje), logo alocar memória (para os vectores, por exemplo) resultavam em espaços de endereços que eram compatíveis para o tipo inteiro primitivo (`int`) do Java. No fundo, não era expectável que um vetor em Java tivesse tamanho tal que desafiasse os limites do tipo `int`. Foi nesta altura (no fim da primeira metade da primeira década de 2000) que o tamanho padrão das memórias dos computadores aumentou para além deste limite. Foi este facto material que proporcionou o contexto para o bug acontecer e ser detectado.

Experiências de computação começaram a dar respostas anormais e o problema foi diagnosticado e localizado a esta questão do cálculo do indice do meio, por culpa deste ganho de capacidade de memória.

Exercício

- Implemente a versão recursiva da pesquisa binária.
- Modifique a pesquisa binária para encontrar o primeiro e o último índice de um valor repetido no vetor.
- Adapte a pesquisa binária para contar quantas comparações foram feitas antes de encontrar o elemento ou concluir que ele não está presente.
- Modifique a pesquisa binária para buscar o menor elemento maior ou igual a um dado valor (pesquisa com limite inferior).
- Implemente uma pesquisa binária adaptada para encontrar o maior elemento menor ou igual a um dado valor (pesquisa com limite superior).

Pesquisa Linear e Binária para Tipos Genéricos

Em C, podemos criar versões genéricas das funções de pesquisa utilizando **apontadores `void *`.

Pesquisa Linear Genérica

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

void* pesquisa_linear_generica(void* base, size_t n, size_t
tamanho_elemento, void* chave, int (*cmp)(const void*, const
void*)) {
    for (size_t i = 0; i < n; i++) {
        void* elemento = (char*)base + i * tamanho_elemento;
        if (cmp(elemento, chave) == 0) {
            return elemento;
        }
    }
    return NULL;
}

int compara_int(const void* a, const void* b) {
    return (*(int*)a - *(int*)b);
}

int main() {
    int numeros[] = {10, 20, 30, 40, 50};
    int chave = 30;
    int tamanho = sizeof(numeros) / sizeof(numeros[0]);

    int* resultado = (int*)pesquisa_linear_generica(numeros,
tamanho, sizeof(int), &chave, compara_int);
    if (resultado) {
        printf("Elemento encontrado: %d\n", *resultado);
    } else {
        printf("Elemento não encontrado\n");
    }
    return 0;
}
```

Como Funciona?

- `void* base` é um apontador genérico para os elementos do vetor.
- `size_t n` é o número de elementos presentes no vetor apontado por `base`.
- `void* chave` é um apontador para o valor da chave. Como `base`, é considerado como um valor de um tipo genérico.
- `size_t tamanho_elemento` define o tamanho de cada elemento.
- `int (*cmp)(const void*, const void*)` é um apontador para a função de comparação.

  Este parâmetro é essencial para a versão genérica. Do ponto de visto da função de pesquisa, o vetor é genérico. A função não sabe que tipo de elementos ele contém, então **como comparar os elementos para saber se encontramos a chave**? Precisamos de informar a função de pesquisa sobre esta operação. É o papel deste parâmetro. Espera-se que esta função `cmp` devolva -1 se o primeiro elemento comparado é menor do que o segundo, 0 se é igual, e $+1$ se é maior.
- `void* elemento = (char*)base + i * tamanho_elemento;` Com esta instrução recuperamos o elemento por comparar com a chave no vetor base. Relembramos o "truque" da conversão para `(char *)` para poder realizar alguma aritmética de apontadores (que é proibida pelo compilador sobre apontadores para `void`). É pouco elegante, mas é uma técnica standard. Esta linha pretende ser equivalente a `base[i]` se não houvesse a questão do vetor genérico.

Pesquisa Binária Genérica

Em vez de implementar manualmente a função de pesquisa binária, podemos usar `bsearch()` de `<stdlib.h>`:

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

int compara_int(const void* a, const void* b) {
    return (*(int*)a - *(int*)b);
}

int main() {
    int numeros[] = {10, 20, 30, 40, 50};
```

```
    int chave = 30;
    int tamanho = sizeof(numeros) / sizeof(numeros[0]);

    int* resultado = (int*)bsearch(&chave, numeros, tamanho,
sizeof(int), compara_int);
    if (resultado) {
        printf("Elemento encontrado: %d\n", *resultado);
    } else {
        printf("Elemento não encontrado\n");
    }
    return 0;
}
```

Exercício

- Modifique o código acima para pesquisar strings (`char *`) num vetor ordenado de `char *`.
- Modifique o código acima para pesquisar estruturas (`struct`) num vetor ordenada com base num campo específico. Por exemplo, define um tipo de estrutura `Pessoa`, com os campos `nome`, `idade` e `altura`. Procure num vetor ordenado de `Pessoas` pelo "Pedro".
- Proponha uma implementação de `lsreach` (pesquisa linear) e de `bsearch` (pesquisa binária) para os `dynvec` genéricos..

Ordenação de elementos

Nesta secção vamos abordar o tema principal desta ficha. Como ordenar.

(TO BE COMPLETED! - ordenação, string, string.h, I/O e ficheiros)

Trabalho final

Quais são as 30 palavras mais usadas no "Padre Amaro" de Eça de Queirós?

Ato 1: a informação

Consultar o magnífico site "projecto gutenberg" https://www.gutenberg.org

Ato 2: a magia scriptal

Podemos enfim responder à custa de um script de uma só linha:

```
cat padre_amaro.txt | tr -cs '[:alpha:]' '[\n*]' | sort | uniq -c | sort -rn
```

Explicação :

1. passar o ficheiro para a saída stantard

   `cat padre_amaro.txt`

2. substitui-se os caracteres não alfanuméricos num fim de linha

   `| tr -cs '[:alpha:]' '[\n*]'`

para melhor tomar conta dos acentos poderá usar em alternativa:

```
| sed 's/[^ [:alpha:]]\+/\n/g'
```

3. *ordenar* todas as linhas por ordem alfabética

   ```
   | sort
   ```

4. agrupar todas as linhas idênticas pedindo contagem (opção -c)

   ```
   | uniq -c
   ```

5. *ordenar* numericamente (opção -n) na ordem inversa (opção -r)

   ```
   | sort -rn
   ```

Ato 3: a iluminação

Assim, as 30 palavras mais usadas na obra "Padre Amaro":

```
10035 o
 6567 a
 3992 de
 3735 que
 2891 e
 2351 se
 2003 da
 1938 com
 1879 n
 1867 um
 1803 do
 1675 os
 1560 uma
 1465 d
 1384 lhe
 1285 as
 1112 para
  984 E
  931 na
  888 s
  887 em
  829 ao
```

749 Amaro
745 no
729 O
641 como
636 sua
586 por
578 ra
577 senhor

Claro, podemos refinar este exemplo, retirando por exemplo os artigos da listagem. De realçar: usamos para este exemplo duas fases de ordenação

Ato 4: o desfecho

Objectivo

Escreva um programa C que lê um inteiro n numa primeira linha. Este inteiro representa o número de palavras por considerar no ranking. A obra por analisar está nas linhas seguintes. A obra é dada no standard input linha por linha até chegar ao fim de ficheiro (`EOF`).

Este programa deve apresentar o ranking das n palavras mais utilizadas na obra lida, no formato apresentado acima.

```
o p
```

Em que `p` é a palavra e `o` é o seu número de ocorrências na obra lida.

Critérios e restrições

Deverá necessariamente usar duas funções de ordenação diferentes para a resolução deste problema.