

Ponteiros

Os tipos básicos são representados em memória por grupos de células de memória.

Um ponteiro é um grupo de células capaz de guardar um endereço de memória.

Na arquitetura X86-64 um ponteiro tem a dimensão de 8 bytes.

O **operador &** aplica-se a uma variável e serve para obter o ponteiro para essa variável (que é o seu endereço de memória).

```
char c;  
char * p = &c;
```

O **operador *** aplica-se a um ponteiro e acede ao valor apontado (conteúdo de). Operador desreferenciação.

Sendo **p** um ponteiro para inteiro, ***p** representa um inteiro e pode aparecer no lugar de um inteiro.

& - ponteiro para

* - conteúdo de

Ponteiros como argumento de funções

A linguagem C só tem passagem de parâmetros por valor.

O tipo ponteiro permite passar, como valor de um parâmetro, a referência de uma variável (endereço).

Através desse ponteiro pode-se aceder ao conteúdo da variável, simulando-se assim uma passagem por referência.

```
void swap(int *pa, int *pb) {  
    int aux = *pa;  
    *pa = *pb;  
    *pb = aux;  
}  
  
int a = 22, b = 33;  
int main() {  
    swap(&a, &b);  
}
```

```
void swap(int pa, int pb) {  
    int aux = pa;  
    pa = pb;  
    pb = aux;  
}  
  
int a = 22, b = 33;  
int main() {  
    swap(a, b);  
}
```

Ponteiros e arrays

```
int a[10], *p;
```

O identificador do array – **a** é equivalente ao ponteiro para a primeira posição - **&a[0]**

p = a é equivalente a **p = &a[0]**

As operações sobre *arrays* com operador indexação podem ser escritas com notação de ponteiros.

	acesso a conteúdo	ponteiros para posições do <i>array</i>
operador indexação	<code>a[0]</code> <code>a[1]</code> <code>a[i]</code>	<code>&a[0]</code> <code>&a[1]</code> <code>&a[i]</code>
notação de ponteiro	<code>*a</code> <code>*(a + 1)</code> <code>*(a + i)</code>	<code>a</code> <code>a + 1</code> <code>a + i</code>

a não é uma variável, é um valor constante do tipo ponteiro. Não é possível **a++** ou **a = p**.

Exemplo

Ordenar um *array* de inteiros

```
void sort(int array[], size_t size) {
    for (size_t i = 0; i < size - 1; ++i)
        for (size_t j = 0; j < size - i - 1; ++j)
            if (array[j] > array[j + 1])
                swap(&array[j], &array[j + 1]);
}

void print(int array[], size_t size) {
    putchar('\n');
    for (size_t i = 0; i < size; ++i)
        printf("%d ", array[i]);
    putchar('\n');
}

int array[] = {1, 2, 20, 4, 5, 30, 10, 34, 22};

int main() {
    print(array, ARRAY_SIZE(array));
    sort(array, ARRAY_SIZE(array));
    print(array, ARRAY_SIZE(array));
}
```

Programação alternativa usando notação de ponteiro. (A utilização de ponteiros pode, eventualmente, ser melhor).

```
void sort(int *array, size_t size) {
    for (size_t i = 0; i < size - 1; ++i)
        for (int *p = array; p < array + size - i - 1; ++p) {
            if (*p > *(p + 1))
                swap(p, p + 1);
        }
}
```

A cópia de *arrays* não pode ser feita assim:

```
int values[] = {34, 40, 36, 36, 37, 33, 33, 32};
int buffer[100];
buffer = values;
```

buffer e **values**, sem o operador indexação, equivalem aos ponteiros para a primeira posição dos *arrays*.

A cópia tem que ser feita elemento a elemento.

Ponteiros para caracteres

`char message[] = "texto para teste";` *array* de caracteres
`char * pmessage = "texto para teste";` ponteiro para array de caracteres com atributo de constante
`message[i]` dá acesso ao mesmo caracter que `pmessage[i]`

String

Na linguagem C, *string* é equivalente a um *array* de caracteres terminados por 0.

A biblioteca da linguagem normaliza funções para manipulação de *strings*.

Array como argumento de funções

Quando se passa um *array* como argumento de função efetivamente está-se a passar o ponteiro para a primeira posição. Internamente à função, o parâmetro é suportado numa variável local do tipo ponteiro.

`void to_upper(char str[])` é equivalente a `void to_upper(char *str)`
`to_upper("uma string constante");` *string* constante
`to_upper(array);` *array* de caracteres - `char array[10]`
`to_upper(ptr);` variável do tipo ponteiro - `char *ptr`

Aritmética de ponteiros

Princípio básico: se **p** é um ponteiro para um elemento de um array então **p + 1** é o ponteiro para o elemento seguinte.

Admitindo que a unidade de endereçamento da arquitetura de processador é o *byte*.

Se o tipo **char** ocupar um *byte* em memória **p + 1** incrementa o endereço de uma unidade.

Se o tipo **float** ocupar quatro *bytes* em memória **p + 1** incrementa o endereço de quatro unidades.

p++ pós-incrementa **p** do número de *bytes* igual à dimensão de um elemento.

++*p pré-incrementa o valor apontado por **p**.

***++p** pré-incrementa o ponteiro e depois desreferencia-o acedendo ao valor apontado.

***p++** desreferencia o ponteiro acedendo ao elemento apontado e depois incrementa o ponteiro.

p + n aponta para o elemento **n** posições à frente do elemento apontado por **p**. Se **n** tiver o valor 4, **p** será aumentado de `4 * sizeof(*p)` *bytes*.

p - q representa o número de elementos entre os ponteiros **p** e **q**.

Não é possível realizar outras operações sobre ponteiros. Por exemplo, não é possível somar dois ponteiros.

Ponteiro tipo void

Um ponteiro para **void** não pode ser desreferenciado, somado ou subtraído, porque o elemento

apontado é indefinido.

Por omissão, o compilador gcc trata o ponteiro para void como ponteiro para char para efeito de aritmética de ponteiros (não está segundo a norma).

Uma variável do tipo ponteiro para void pode receber ponteiros de qualquer tipo. Assim como afetar ponteiros de qualquer tipo.

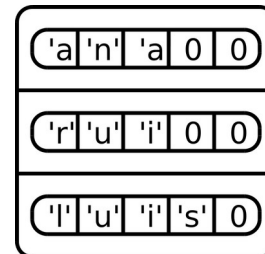
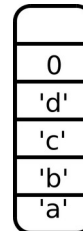
Array bidimensional

```
char a[6] = "abcd";
```

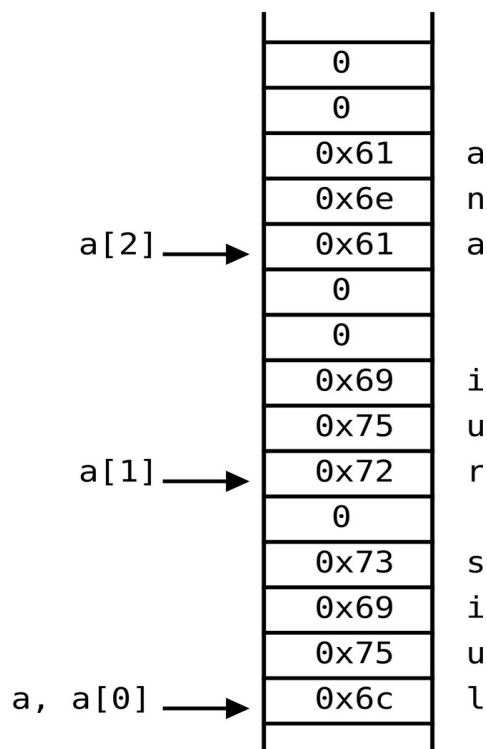
a é um *array*; os elementos do *array* são caracteres

```
char a[][5] = {"luis", "rui", "ana"};
```

a é um *array*; os elementos do *array* são *arrays* de caracteres



Visualização em memória



Exemplo

Ordenar uma sequência de nomes de pessoas, representados num *array* bidimensional de caracteres.

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

void swap(char a[], char b[]) {
    size_t i;
```

```

    for (i = 0; a[i] != 0 && b[i] != 0; ++i) {
        char tmp = a[i];
        a[i] = b[i];
        b[i] = tmp;
    }
    if (a[i] == 0) {
        for (; b[i] != 0 ; ++i)
            a[i] = b[i];
        a[i] = 0;
    }
    else {
        for (; a[i] != 0 ; ++i)
            b[i] = a[i];
        b[i] = 0;
    }
}

void sort(char array[][100], int size) {
    for (size_t i = 0; i < size - 1; ++i)
        for (size_t j = 0; j < size - i - 1; ++j)
            if (strcmp(array[j], array[j + 1]) > 0)
                swap(array[j], array[j + 1]);
}

void print(char array[][100], size_t size) {
    putchar('\n');
    for (size_t i = 0; i < size; ++i)
        printf("%s\n", array[i]);
}

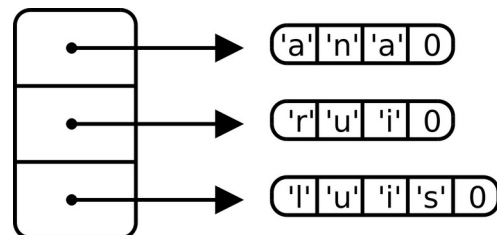
char names[][100] = {
    "antonio manuel",
    "joaquim antunes",
    "manuel francisco",
    "luis alfredo"
};

int main() {
    int i;
    print(names, ARRAY_SIZE(names));
    sort(names, ARRAY_SIZE(names));
    print(names, ARRAY_SIZE(names));
}

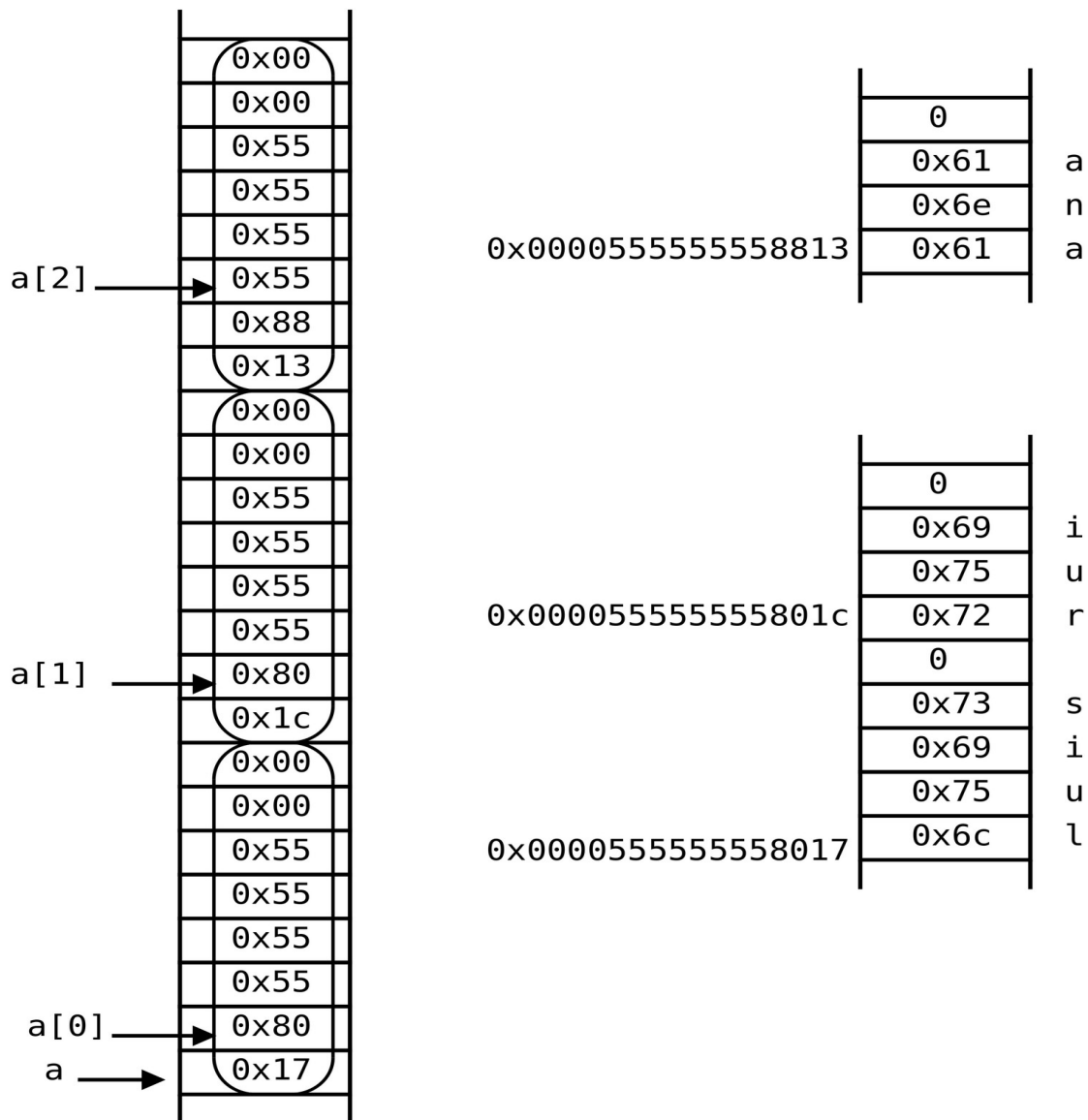
```

Array de ponteiros

```
char *a[] = {"luis", "rui", "ana"};
```



Visualização em memória



Exemplo

Ordenar uma sequência de nomes, representada como um *array* de ponteiros para *strings*.

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

void swap(char **a, char **b) {
    char *aux = *a;
    *a = *b;
    *b = aux;
}

void sort(char *array[], int size) {
    for (size_t i = 0; i < size - 1; ++i)
        for (size_t j = 0; j < size - i - 1; ++j)
            if (strcmp(array[j], array[j + 1]) > 0)
```

```

        swap(&array[j], &array[j + 1]);
    }

void print(char *array[], size_t size) {
    putchar('\n');
    for (size_t i = 0; i < size; ++i)
        printf("%s\n", array[i]);
}

char *nomes[] = {
    "antonio manuel",
    "joaquim antunes",
    "manuel francisco",
    "luis alfredo"
};

int main() {
    print(nomes, ARRAY_SIZE(nomes));
    sort(nomes, ARRAY_SIZE(nomes));
    print(nomes, ARRAY_SIZE(nomes));
}

```

Argumentos na linha de comando.

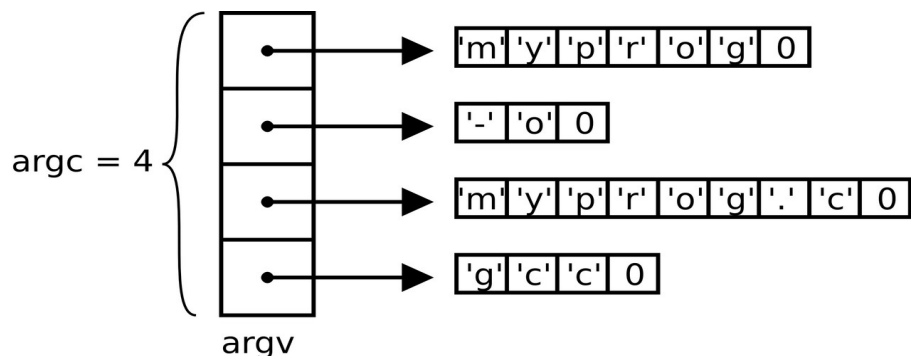
As palavras escritas na linha de comando são enviadas como argumentos para o programa. São passadas como argumento da função **main**, na forma de um *array* de ponteiros para caracteres, apontando cada ponteiro para o início de cada palavra.

```
int main(int argc, char *argv[]);
```

Por exemplo, na invocação do comando:

```
$ gcc myprog.c -o myprog
```

o programa gcc vai receber os seguintes argumentos:



Exercícios

1. Fazer um programa que imprima os argumentos na consola.
2. Fazer uma função para separar as palavras numa linha de texto.

```
size_t string_split(char *text_line, char *words[], size_t words_size);
```

3. Fazer um programa para listar as últimas **n** linhas do texto de entrada.

```
$ tail -n
```

Structs

O tipo *struct* agrega variáveis de tipos diferentes.

Declaração do tipo:

```
struct pessoa {  
    char nome[100];  
    int idade;  
    int peso;  
    float altura;  
};
```

Definição de variável do tipo *struct*:

```
struct pessoa utente;
```

Para simplificar a escrita pode-se usar:

```
typedef struct pessoa Pessoa;
```

e escrever *Pessoa* em vez de *struct pessoa*.

```
Pessoa eu;
```

Acesso a membro (<nome da *struct*>.<membro>):

```
eu.idade;
```

As *struct* podem-se copiar com o operador afetação:

```
Pessoa a, b;  
a = b;
```

A passagem de *struct* como argumento de função é feita por valor.

```
int imc(struct pessoa p) {  
    return p.peso / (p.altura * p.altura);  
}
```

Retorno de *struct* como valor de uma função.

```
struct pessoa pessoa_nova(char n[], int i, int p, int a) {  
    struct pessoa temp;  
    strcpy(temp.nome, n);  
    temp.idade = i;  
    temp.peso = p;  
    temp.altura = a;  
    return temp;  
}
```

Inicialização na definição

```
struct pessoa pessoa = {"António", 53, 80, 1.76};  
struct pessoa pessoa = {  
    .nome = "António",  
    .altura = 1.76
```



```
};
```

Ponteiros e *structs*

Se for necessário passar uma estrutura muito grande para uma função, deve-se considerar a passagem por ponteiro.

```
int imc(struct pessoa *p) {  
    return (*p).peso / ((*p).altura * (*p).altura);  
}
```

A linguagem C dispõe de um operador alternativo para aceder ao membro de uma *struct* baseado em ponteiro: `->`

```
return p->peso / (p->altura * p->altura);
```

Os operadores `.` e `->` em conjunto com `()` e `[]` são os mais prioritários.

```
struct pessoa {  
    char nome[100];  
    int idade;  
    int peso;  
    int altura;  
} *p;
```

`++p->idade` - incrementa o membro `idade`.

`(p++)->idade` - incrementa `p` depois de aceder a `idade`. Só faz sentido num *array* de pessoas.

`*p->nome` - dá acesso ao primeiro carácter de `nome`.

`*p++->nome` - incrementa `p` depois de aceder ao primeiro carácter de `nome`.

`(*p->nome)++` - incrementa o código do primeiro carácter de `nome`.

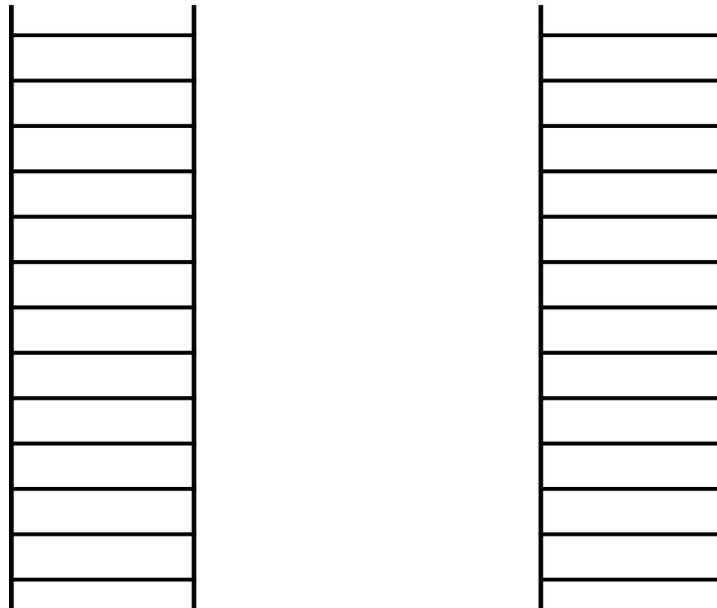
Exercício

1. Fazer um programa que ordene um *array* de *struct* do tipo Pessoa.
2. Fazer um programa que ordene um *array* de ponteiros para *struct* do tipo Pessoa.

Alojamento de *struct* em memória

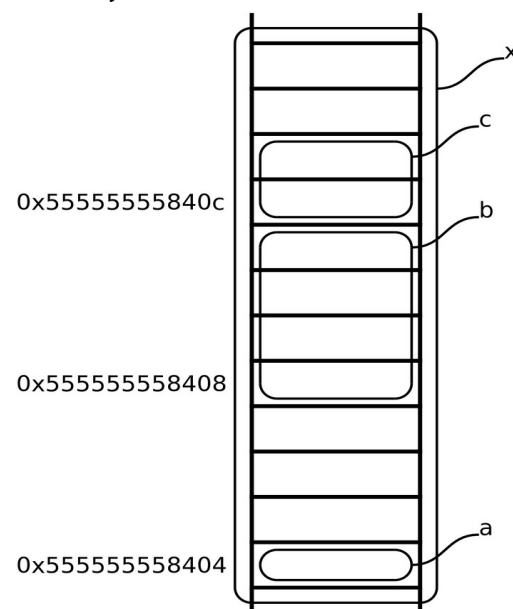
Consideremos a seguinte definição da variável **x**:

```
struct y {
    char a;
    int b;
    short c;
} x;
```



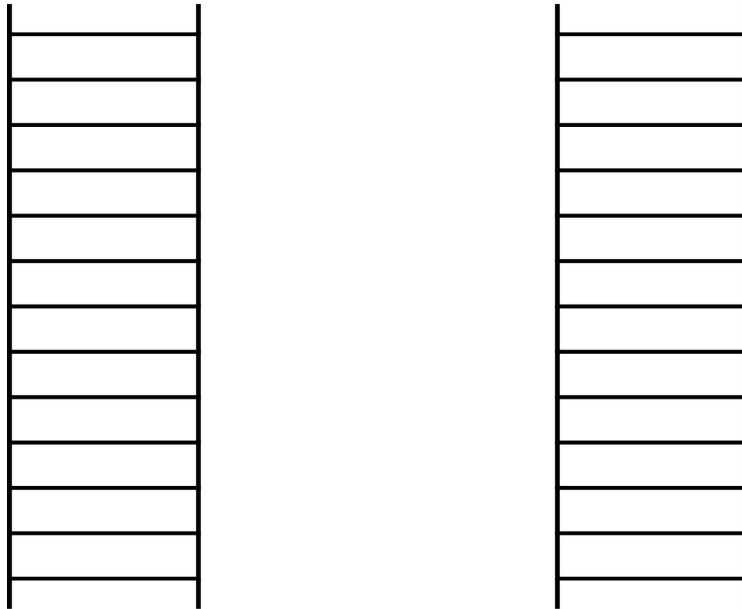
- Os membros de uma *struct* são dispostos em memória seguindo as regras de alinhamento do tipo a que pertencem, mesmo que para isso seja necessário inutilizar posições de memória entre campos consecutivos.
- O endereço de início da *struct* é alinhado no maior alinhamento necessário a um dos seus campos.
- A dimensão de uma *struct* é múltipla do seu alinhamento.

Num processador a 32 ou 64 bits, com compilador GNU, para alinhar o campo **b**, é necessário avançar três posições de memória. A dimensão total desta *struct* é de 12 *bytes*.



A dimensão de uma *struct* depende da ordem e do tipo dos campos. Consideremos a definição alternativa da variável **x**:

```
struct y {
    char a;
    short c;
    int b;
} x;
```



Na arquitetura Intel é possível alojar variáveis desalinhadas. Um acesso desalinhado consome duas operações de acesso à memória. Um acesso alinhado consome apenas uma operação de acesso à memória.

Na arquitetura ARM, por definição, acessos desalinhados são interditos. A implementação *hardware* não contempla essa operação. Em algumas implementações, se isso acontecer, o processador interrompe o programa, noutras o resultado é indefinido.

Exercícios

1. Desenhar a ocupação de memória para um *array* de duas *struct* do tipo **struct y**.
2. Tente desenhar a ocupação de memória para um *array* de duas *struct* do tipo **struct y**, colocando o campo **a** da *struct* da primeira posição num endereço ímpar e mantendo o critério de alinhamento para todos os campos.
3. Procure uma definição alternativa para **struct y** de modo a ocupar menos espaço de memória.

Arrays de structs e de ponteiros para struct

Exemplo

Realizar um programa para contar as palavras de um texto e imprimir as dez mais frequentes. Registrar a informação de cada palavra numa *struct* com dois campos, um *array* de caracteres para armazenar a palavra e um campo do tipo inteiro para acumular a contagem. Implementar a coleção de palavras num *array* dessas *structs*.

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

```
#define WORD_MAX_SIZE 30

#define WORDS_MAX 40000

typedef struct {
    char word[WORD_MAX_SIZE];
    int counter;
} Word;

Word words[WORDS_MAX];
int words_counter;

void word_insert(char *word) {
    int i;
    for (i = 0; i < words_counter; ++i) {
        if (strcmp(words[i].word, word) == 0) {
            words[i].counter++;
            break;
        }
    }
    if (i == words_counter) {
        strcpy(words[words_counter++].word, word);
        words[i].counter = 1;
    }
}

void sort(Word words[], size_t n) {
    int i, j;
    for (i = 0; i < n - 1; ++i)
        for (j = 0; j < n - i - 1; ++j)
            if (words[j].counter < words[j + 1].counter) {
                Word tmp = words[j];
                words[j] = words[j + 1];
                words[j + 1] = tmp;
            }
}

void word_print(int n) ;

int word_read(char buffer[], size_t size);

int main() {
    char word_buffer[ WORD_MAX_SIZE];

    while (word_read(word_buffer, sizeof word_buffer) != EOF)
        word_insert(word_buffer);

    sort(words, words_counter);

    word_print(10);
}

void word_print(int n) {
    for (int i = 0; i < n; ++i)
        printf("%s - %d\n", words[i].word, words[i].counter);
}
```

```
int word_read(char buffer[], size_t size) {
    char c = getchar();
    while (isspace(c) && c != EOF)
        c = getchar();
    if (c == EOF)
        return EOF;
    int i = 0;
    while ( ! isspace(c) && c != EOF && i < size - 1) {
        buffer[i++] = c;
        c = getchar();
    }
    buffer[i] = 0;
    return i;
}
```

Exercício

Realizar uma versão do programa anterior em que a coleção de palavras é suportada num *array* de ponteiros para *struct*.

Utilizar instrumentação de medida de tempo de execução em ambas as versões e comparar o desempenho.

Structs com campos baseados em bits

Justificação:

- casos em que se pretenda reduzir a memória ocupada;
- acesso a bits de forma simplificada.

Exemplo 1

Representar uma data de forma compactada.

```
struct date {
    short day: 5;
    short month: 4;
    short year: 7;
};

struct date date_pack(int year, int month, int day) {
    struct date tmp = {year - 2000, month, day };
    return tmp;
}
```

`sizeof (struct date)` é igual a dois, o mesmo que `sizeof (short)`

Exemplo 2

Acesso a registo de periférico mapeado no espaço de memória

```
struct register_status {
    char counter_enable:1;
    char counter_reset: 1;
};

struct register_status *status = 0xe0008004;

status->counter_enable = 1;
```

Como a unidade mínima de acesso à memória é a palavra de memória – *byte*, para afetar um número de bits inferior é necessário ler, alterar os bits desejados e voltar a escrever. Resultando em dois acessos à memória, um de leitura e um de escrita.

Apesar da aparentemente vantagem em relação à utilização de operadores lógicos bit-a-bit, nem sempre é conveniente a sua utilização:

1. Se, no acesso a um registo de periférico, este for só de escrita a operação de leitura é inútil.
2. Se o operação de leitura sobre um registo de periférico provocar a alteração de estado do periférico, ela é certamente indesejada.

Union

Uma ***union*** é uma variável que pode armazenar, em tempos diferentes, valores de diferentes tipo e tamanhos.

Através de uma *union* é possível encarar um dado conteúdo de memória na perspectiva de diferentes tipos.

Exemplo:

```
struct symbol {  
    char *name;  
    int flags;  
    union {  
        int value_int;  
        float value_float;  
        char *value_string;  
    };  
};
```

Avaliação de desempenho

Medida de tempo

```
#include <time.h>

unsigned int get_time() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec * 1000 + (ts.tv_nsec / 1000000);
}

unsigned initial = get_time();

unsigned elapsed = get_time() - initial;
```