



REGISTOS

struct


Um **vector** agrupa dados do mesmo tipo.
Um **registo** agrupa dados de tipos diferentes.

Registos ajudam a organizar dados complexos pois permitem tratar um conjunto de variáveis relacionadas como uma unidade.

abstração

nCal, nome, ano, notas são etiquetas


```
struct Aluno {  
    int nCAL;  
    char nome[40];  
    int ano;  
    float notas[3];  
};
```



declarar
registro

nCal, nome, ano, notas são etiquetas

```
struct Aluno {  
    int nCAL;  
    char nome[40];  
    int ano;  
    float notas[3];  
};
```




declarar
registro

//dentro da main

```
struct Aluno a = {1743, "Asimov", 6, {20.0, 20.0, 20.0}};
```

nCal, nome, ano, notas são etiquetas

```
struct Aluno {  
    int nCAL;  
    char nome[40];  
    int ano;  
    float notas[3];  
};
```



declarar
registo

//dentro da main

```
struct Aluno a = {1743, "Asimov", 6, {20.0, 20.0, 20.0}};
```

1743		'A'	's'	'i'	'm'	'o'	'v'	'\0'	
1000	...	1004	1005	1006	1007	1008	1009	1010	...

	6		20.0		20.0		20.0		
...	1044	...	1048	...	1052		1056	...	1060

nCal, nome, ano, notas são etiquetas

```
struct Aluno {  
    int nCAL;  
    char nome[40];  
    int ano;  
    float notas[3];  
};
```

} declar
registo

//dentro da main

```
struct Aluno a = {1743, "Asimov", 6, {20.0, 20.0, 20.0}};  
struct Aluno *p = &a;
```

a.nCAL	1743
a.nome	1004
a.nome[1]	's'
a.notas	1048
p	1000
p->ano	6

1743		'A'	's'	'i'	'm'	'o'	'v'	'\0'	
1000	...	1004	1005	1006	1007	1008	1009	1010	...

	6		20.0		20.0		20.0		
...	1044	...	1048	...	1052		1056	...	1060

```
struct Aluno {
    int nCAL;
    char nome[40];
    int ano;
    float notas[3];
};
```

//dentro da main

```
struct Aluno a = {1743, "Asimov", 6, {20.0, 20.0, 20.0}};
```

```
struct Aluno *p = &a;
```

```
a.nCal = 1742;
```

```
a.notas[1] = 12.5;
```

} acesso a campo
do registro

a.nCAL	1743
a.nome	1004
a.nome[1]	's'
a.notas	1048
p	1000
p->ano	6

1742		'A'	's'	'i'	'm'	'o'	'v'	'\0'	
1000	...	1004	1005	1006	1007	1008	1009	1010	...

	6		20.0		12.5		20.0		
...	1044	...	1048	...	1052		1056	...	1060

Comparação

```
struct Aluno{
    int nCAL;
    char nome[40];
    int ano;
    float notas[3];
};

//dentro da main

struct Aluno a = {1743, "Asimov", 6, {20.0, 20.0, 20.0}};
struct Aluno b = {1642, "Newton", 6, {20.0, 20.0, 20.0}};

if (a == b) printf("alunos iguais");
```



Comparação de
registos tem de ser
feita campo a campo.

```
if (a.nCal == b.nCal &&
    strcmp(a.nome, b.nome) == 0 &&
    a.ano == b.ano ...)
    printf("alunos iguais");
```

Registos dentro de registos

```
struct ponto{  
    int x;  
    int y;  
};
```

```
struct rect{  
    struct ponto inf_esq;  
    struct ponto sup_dir;  
};
```

//dentro da main

```
struct ponto p1 = {2, 2};  
struct ponto p2 = {4, 3};  
struct rect r = {p1, p2};
```

```
r.inf_esq.x = 0;
```

// p1 nao alterado

// pois r.inf_esq é uma
cópia de p1, e não p1 em si

Registos e funções

```
struct ponto novoponto(int x, int y){  
    struct ponto temp;  
    temp.x = x;  
    temp.y = y;  
    return temp;  
}
```

```
struct rect novorect(struct ponto p1, struct ponto p2){  
    struct rect temp;  
    temp.inf_esq = p1;  
    temp.sup_dir = p2;  
    return temp;  
}
```

Registos e funções

```
struct rect novorect(struct ponto p1, struct ponto p2) {  
    struct rect rtemp;  
    rtemp.inf_esq = p1;  
    rtemp.sup_dir = p2;  
    return temp;  
}
```

//dentro da main

```
struct ponto pa = {2, 2};  
struct ponto pb = {4, 3};  
struct rect r = novorect(pa, pb);
```

cópias de pa e pb

cópia de temp ◀

typedef

typedef serve para dar novos nomes a tipos de dados existentes.

```
typedef <tipo existente> <novo_nome>
```

typedef

typedef serve para dar novos nomes a tipos de dados existentes.

```
typedef <tipo existente> <novo_nome>
```

O tipo existente continua a estar disponível através do seu nome original.

typedef

```
struct ponto{  
    int x;  
    int y;  
};
```



```
typedef struct{  
    int x;  
    int y;  
} Ponto;
```

```
struct rect{  
    struct ponto inf_esq;  
    struct ponto sup_dir;  
};
```



```
typedef struct{  
    Ponto inf_esq;  
    Ponto sup_dir;  
} Rect;
```

typedef

```
typedef struct{  
    int x;  
    int y;  
} Ponto;
```

```
typedef struct{  
    Ponto inf_esq;  
    Ponto sup_dir;  
} Rect;
```

```
Rect novorect(Ponto p1, Ponto * p2){  
    Rect temp;  
    temp.inf_esq = p1;  
    temp.sup_dir = *p2;  
    return temp;  
}
```

```
//dentro da main  
Ponto pa = {2, 2};  
Ponto pb = {4, 3};  
Rect r = novorect(pa, &pb);
```



```
Rect novorect(Ponto p1, Ponto * p2){  
    Rect temp;  
    temp.inf_esq = p1;  
    temp.sup_dir = *p2;  
    return temp;  
}
```

```
//dentro da main  
Ponto pa = {2, 2};  
Ponto pb = {4, 3};  
Rect r = novorect(pa, &pb);
```

O ponto `pa` é copiado para `p1` (argumento da função), que depois é copiado para `temp.inf_esq`.

O ponto `pb` é passado por referência para a função e depois é copiado para `temp.sup_dir`.

Assim, `pa` é copiado 2x, enquanto `pb` só é copiado 1x.

Usar apontadores de registos reduz significativamente a quantidade de cópias de dados e torna o programa mais rápido.

apontadores e registos

```
typedef struct {  
    char matricula[6];  
    char marca[20];  
    char modelo[30];  
    int tipo;  
} Veiculo;
```

```
typedef struct {  
    int id;  
    char descricao[500];  
    int n_veiculos;  
    Veiculo veiculos[50];  
} Frota;
```

```
void mudaMatricula(Frota * f, int i, char * m){  
    int j;  
    for(j=0; j<6; j++)  
        f->veiculos[i].matricula[j] = m[j];  
}
```

```
//dentro da main }
```

```
Frota f1;
```

```
f1.id = 1;
```

```
Veiculo v = {"31RI59", "DeLorean", "DMC-12", 1};
```

```
f1.veiculos[0] = v;
```

```
f1.n_veiculos = 1;
```



Exercício 1

Escreva um programa que guarde e mostre o registo de um aluno `Aluno` da cadeira de programação. Um aluno tem os seguintes parâmetros: nome (string), NIP (int), vector com 3 números reais para guardar as classificações das avaliações.

Escreva uma função `criar_aluno` que pede os dados do aluno ao utilizador e devolve um registo de aluno com esses dados.

Escreva uma função `mostrar_aluno` que recebe o apontador de um registo de aluno e imprime os seus valores na consola.

Exercício 1a

Expanda o programa anterior para mostrar os registos dos alunos da cadeira de Programação.

Crie um vector de `Aluno` de tamanho `MAX_ALUNOS` e uma variável `n_alunos` que recebe do utilizador quantos alunos vai receber.

Peça os dados de todos os alunos a receber e no final, escreva na consola os dados de todos os alunos.

Exercício 1b

Modifique o programa anterior por forma a que um registo `Aluno` agora contenha um vector de registos `Cadeira`. Um registo `Cadeira` tem: nome, ano, vector com `MAX_AVAL` elementos para guardar as classificações das avaliações e um inteiro `n_avaliacoes` para especificar o número de avaliações da cadeira.

O vector das cadeiras tem um máximo de `MAX_CADEIRAS` (definir constante um valor, e.g. 10). `MAX_AVAL` é uma contante, também.

Crie a função `pedirCadeira` para pedir os dados de 1 `Cadeira` ao utilizador e devolver um registo `Cadeira`.

Exercício 1b

Crie uma função `adicionarCadeira` para adicionar uma `Cadeira` a um aluno. A função recebe um apontador para um `Aluno` e um apontador para uma `Cadeira`. Deve guardar a `Cadeira` no local correcto do vetor de `Cadeira`.

Crie a função `mostrarCadeira`.

Agora a cadeira de Programação é um dos elementos do vector, cujo o número de avaliações é 3.

Modifique o programa p

Exercício 1c

Modifique o programa anterior por forma a que um registo `Curso` agora contenha um vector de registos `Aluno`, o número de alunos, o nome do curso, o ano de entrada. O vector de alunos tem um tamanho máximo de `MAX_ALUNOS`.

Escreva as funções `criarCurso`, `adicionarAluno`, `mostrarCurso`.

`criarCurso` Pede ao utilizador o nome do curso, ano de entrada e número de alunos.

`adicionarAluno` Recebe um apontador de `Curso`, um apontador de `Aluno` e introduz o aluno no vetor de `Aluno` que faz parte do `Curso`.

Exercício 1c

Escreva as funções `mediaCadeira`, `mediaAluno`, `mediaCurso`.

`mediaCadeira` Devolve a média de todas as notas da Cadeira.

`mediaAluno` Devolve a média de todas as cadeiras de um Aluno.

`mediaCurso` Devolve a média de todos os alunos de um Curso.

Exercício 1c

Use todas as funções até agora para criar um curso com todos os alunos da turma de Programação.

Adicione todos os alunos da turma, a cadeira de Programação e outra ao calhas a todos os alunos e invente notas fictícias.

As funções de média devem ser usadas para mostrar a média de cada cadeira de cada aluno, a média de todas as cadeiras de cada aluno e a média do curso.

Exercício 1c

Use todas as funções até agora para criar um programa que faz o seguinte:

1. Cria um curso (pede nome, ano de entrada e nº de alunos).
2. Cria alunos para esse curso.
3. Cria 2 cadeiras (Programação e outra ao calhas) para cada um dos alunos. Pede o nº de avaliações e todas as notas.
4. Imprime os dados de cada aluno, com a média de cada Cadeira e a media do Aluno. No final imprime a media do Curso.

Exercício 1d

Crie uma função (escolha um nome apropriado) que recebe o nome de uma Cadeira (string), o apontador para um Curso e devolve a média de todos os alunos do Curso que têm essa cadeira. A média de cada aluno é apenas para essa Cadeira em particular e não para todas as cadeiras que esse aluno tem.

Exercício 2

No exemplo das frotas e veiculos, altere a função `mudarMatricula` para receber uma `matricula` `mat` (`string`) invés do argumento `i`. A função tem de procurar em todos os veiculos qual é o veiculo que tem a `matricula` `mat` e depois substituir essa `matricula` por `nova mat`.