MC102 – Algoritmos e Programação de Computadores

Instituto de Computação

UNICAMP

Primeiro Semestre de 2015

Roteiro

1. Tipos enumerados
2. Registros
3. Vetor de registros
4. Redefinição de tipos
5. Turma de alunos
6. Ponteiros para registros
7. Controle de estoque
8. Exercícios

Tipos enumerados

- Para criar uma variável para armazenar um determinado mês de um ano (de janeiro a dezembro), uma das soluções possíveis é utilizar o tipo inteiro e armazenar um número associado àquele mês. Assim, janeiro seria o mês número 1, fevereiro o mês número 2, e assim sucessivamente.
- Entretanto, o código poderia ser mais claro se pudéssemos escrever algo como:

```
mes = jan;
```

Tipos enumerados

- O comando `enum` cria um tipo enumerado. Com ele podemos usar nomes/identificadores para um conjunto finito de valores inteiros.
- A sintaxe do comando `enum` é a seguinte:

```
enum nome_do_tipo {identificador_1, ..., identificador_n};
```

- Exemplo:

```
enum mes {jan, fev, mar, abr, mai, jun,
          jul, ago, set, out, nov, dez};
```

Usando um tipo enumerado

- O compilador associa o número 0 ao primeiro identificador, o número 1 ao segundo identificador, e assim por diante.
- Variáveis do novo tipo criado são, na realidade, variáveis inteiras.
- Tipos enumerados são usados para deixar o código mais legível.

```
#include <stdio.h>

enum mes {jan, fev, mar, abr, mai, jun,
          jul, ago, set, out, nov, dez};

int main() {
  enum mes a = fev, b = jun;

  printf("a = %d\n", a);
  printf("b = %d\n", b);

  return 0;
}
```

Usando um tipo enumerado

- O compilador associa o número 0 ao primeiro identificador, o número 1 ao segundo identificador, e assim por diante.
- Variáveis do novo tipo criado são, na realidade, variáveis inteiras.
- Tipos enumerados são usados para deixar o código mais legível.

```
#include <stdio.h>

enum mes {jan, fev, mar, abr, mai, jun,
          jul, ago, set, out, nov, dez};

int main() {
  enum mes a = fev, b = jun;

  printf("a = %d\n", a); /* a = 1 */
  printf("b = %d\n", b); /* b = 5 */

  return 0;
}
```

Usando um tipo enumerado

- Note que o primeiro identificador recebeu o valor zero, enquanto os demais identificadores receberam valores em sequência (1, 2, etc).
- Podemos alterar o valor do primeiro identificador e, assim, os valores de todos os demais identificadores.

```
#include <stdio.h>

enum mes {jan = 1, fev, mar, abr, mai, jun,
          jul, ago, set, out, nov, dez};

int main() {
  enum mes a = fev, b = jun;

  printf("a = %d\n", a);
  printf("b = %d\n", b);

  return 0;
}
```

Usando um tipo enumerado

- Note que o primeiro identificador recebeu o valor zero, enquanto os demais identificadores receberam valores em sequência (1, 2, etc).
- Podemos alterar o valor do primeiro identificador e, assim, os valores de todos os demais identificadores.

```
#include <stdio.h>

enum mes {jan = 1, fev, mar, abr, mai, jun,
          jul, ago, set, out, nov, dez};

int main() {
  enum mes a = fev, b = jun;

  printf("a = %d\n", a); /* a = 2 */
  printf("b = %d\n", b); /* b = 6 */

  return 0;
}
```

Usando um tipo enumerado

```
#include <stdio.h>

enum mes {jan = 1, fev, mar, abr, mai, jun,
          jul, ago, set, out, nov, dez};

int main() {
  enum mes x;

  for (x = jan; x <= dez; x++)
    printf("Mes %d\n", x);

  printf("%d: mes invalido\n", x);

  return 0;
}
```

Usando um tipo enumerado

```
#include <stdio.h>

enum mes {jan = 1, fev, mar, abr, mai, jun,
          jul, ago, set, out, nov, dez};

int main() {
  enum mes x;

  for (x = jan; x <= dez; x++)
    printf("Mes %d\n", x);

  printf("%d: mes invalido\n", x); /* 13: mes invalido */

  return 0;
}
```

Outros exemplos de tipos enumerados

- Tipo para armazenar respostas binárias:

```
enum resposta {falsa, verdadeira};
```

- Tipo para armazenar as estações do ano:

```
enum estacao {primavera, verao, outono, inverno};
```

- Tipo para armazenar o sexo de uma pessoa:

```
enum sexo {masculino, feminino};
```

- Tipo para armazenar o naipe de uma carta de baralho:

```
enum naipe {ouros, espadas, copas, paus};
```

- Tipo para armazenar a direção de navegação:

```
enum direcao {N, NE, E, SE, S, SO, O, NO};
```

Registros

- Um registro é um mecanismo da linguagem C para agrupar diversas variáveis, que inclusive podem ser de tipos diferentes, mas que, dentro de um contexto, fazem sentido serem tratadas conjuntamente.
- Exemplos de usos de registros:
  - Registro de alunos para guardar dados como nome, RA, médias de provas e médias de laboratórios.
  - Registro de pacientes para guardar dados como nome, endereço e histórico de doenças.
  - Registro de clientes para guardar dados como nome, endereço, CPF, telefone e número do cartão de crédito.

Definindo um tipo registro

- Para definirmos um tipo de registro, usamos a palavra reservada `struct` da seguinte forma:

```
struct nome_do_tipo {
  tipo_1 nome_do_campo_1;
  tipo_2 nome_do_campo_2;
  ...
  tipo_n nome_do_campo_n;
};
```

- Cada `nome_do_campo_i` é um identificador de um campo do registro, que será do tipo `tipo_i`.
- Exemplo:

```
struct Aluno {
  char nome[81];
  int idade;
  char sexo;
};
```

Definindo um tipo registro

- A definição de um tipo registro pode ser feita dentro de uma função (como `main`) ou fora dela. Usualmente, ela é feita fora de qualquer função, como no exemplo abaixo:

```
#include <stdio.h>

/* Definindo os tipos de registros */
...

void test() {
  ...
}

int main() {
  ...
}
```

Declarando uma variável registro

- O próximo passo é declarar uma variável registro do tipo definido, que será usada pelo seu programa, como no exemplo abaixo:

```
#include <stdio.h>

struct Aluno {
  char nome[81];
  int idade;
  char sexo;
};

int main() {
  /* Declarando variaveis registros do tipo Aluno */
  struct Aluno a, b;
  ...
  return 0;
}
```

Utilizando os campos de um registro

- Podemos acessar individualmente os campos de uma determinada variável registro como se fossem variáveis comuns.
- Neste caso, a sintaxe é a seguinte:

```
variavel_registro.nome_do_campo
```

- Os campos individuais de um variável registro têm o mesmo comportamento de qualquer variável do tipo do campo.
- Ou seja, todas operações válidas para variáveis de um tipo são válidas para um campo do mesmo tipo.

Utilizando os campos de um registro

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

struct Aluno {
  char nome[81];
  int idade;
  char sexo;
};

int main() {
  struct Aluno a, b;

  strcpy(a.nome, "Helen");
  a.idade = 18;
  a.sexo = 'F';
  strcpy(b.nome, "Dilbert");
  b.idade = 34;
  b.sexo = 'M';

  printf("a.nome = %s, a.idade = %d, a.sexo = %c\n", a.nome, a.idade, a.sexo);
  printf("b.nome = %s, b.idade = %d, b.sexo = %c\n", b.nome, b.idade, b.sexo);
  return 0;
}
```

Lendo e escrevendo registros

- A leitura dos campos de um registro deve ser feita campo a campo, como se fossem variáveis independentes.
- O mesmo vale para a escrita, que deve ser feita campo a campo.

Lendo e escrevendo registros

```
#include <stdio.h>

struct Aluno {
  char nome[81];
  int idade;
  char sexo;
};

int main() {
  struct Aluno a;

  printf("Digite o nome: ");
  fgets(a.nome, 81, stdin);
  printf("Digite a idade: ");
  scanf("%d ", &a.idade);
  printf("Digite o sexo: ");
  scanf("%c", &a.sexo);

  printf("a.nome = %s, a.idade = %d, a.sexo = %c\n", a.nome, a.idade, a.sexo);

  return 0;
}
```

Atribuição de registros

- Podemos atribuir (copiar) um registro para outro diretamente:

```
variavel_registro_1 = variavel_registro_2;
```

- Neste caso é feita uma cópia de todos os campos do registro `variavel_registro_2` para o registro `variavel_registro_1`.

Atribuição de registros

```
#include <stdio.h>

struct Aluno {
  char nome[81];
  int idade;
  char sexo;
};

int main() {
  struct Aluno a, b;

  printf("Digite o nome: ");
  fgets(a.nome, 81, stdin);
  printf("Digite a idade: ");
  scanf("%d ", &a.idade);
  printf("Digite o sexo: ");
  scanf("%c", &a.sexo);

  b = a;

  printf("b.nome = %s, b.idade = %d, b.sexo = %c\n", b.nome, b.idade, b.sexo);
  return 0;
}
```

Vetor de registros

- Podemos declarar um vetor de registros quando necessitamos de várias instâncias de um mesmo tipo de registro, por exemplo, para cadastrar todos os alunos de uma mesma turma.
- Como declarar um vetor de registros:

```
struct tipo_do_registro nome_do_vetor[MAX];
```

- Como acessar um campo de registro de um vetor:

```
nome_do_vetor[indice].campo
```

Vetor de registros

```
#include <stdio.h>

struct Aluno {
  int ra;
  double nota;
};

int main () {
  struct Aluno turma[10];
  int i;
  double nota = 0.0;

  for (i = 0; i < 10; i++) {
    printf("Digite o RA do aluno: ");
    scanf("%d", &turma[i].ra);
    printf("Digite a nota do aluno: ");
    scanf("%lf", &turma[i].nota);
  }
```

Vetor de registros

```
  ...

  /* Calcula a nota media da turma */
  for (i = 0; i < 10; i++)
    nota = nota + turma[i].nota;

  printf("Media da turma: %f\n", nota / 10);

  return 0;
}
```

Redefinido um tipo

- Para facilitar a compreensão e melhorar a organização dos códigos, podemos declarar um tipo próprio, que possui a mesma função de outro tipo já existente.
- Por exemplo, em um programa onde manipulamos médias de alunos, podemos declarar as variáveis relacionadas com o campo nota como `nota` e não `double`.

O comando `typedef`

- A forma de se fazer isso é utilizando o comando `typedef`, seguindo a sintaxe abaixo:

```
typedef tipo_existente tipo_novo;
```

- Usualmente, fazemos essa declaração fora da função `main()`, embora também seja permitido declarar um novo tipo dentro de uma função.
- Exemplo:

```
typedef double nota;
```

- O exemplo acima cria um novo tipo, chamado `nota`, cujas variáveis desse tipo serão pontos flutuantes (`double`).

Exemplo de uso do typedef

```
#include <stdio.h>

typedef double nota;

int main() {
  nota prova;

  printf("Digite a nota da prova: ");
  scanf("%lf", &prova);

  printf("Nota da prova: %f\n", prova);

  return 0;
}
```

O comando `typedef`

- Os usos mais comuns para o comando `typedef` são as redefinições de tipos enumerados e registros.
- Podemos redefinir o tipo `enum mes` como simplesmente `mes`:

```
typedef enum mes mes;
```

- Assim como podemos redefinir o tipo `struct Aluno` como simplesmente `Aluno`:

```
typedef struct Aluno Aluno;
```

O comando typedef

- O comando `typedef` pode ser usado junto com a definição de `enum` ou de `struct`, determinando o nome para o novo tipo.
- Podemos definir o tipo `mes` da seguinte forma:

```
typedef enum {jan, fev, mar, abr, mai, jun,
              jul, ago, set, out, nov, dez} mes;
```

- Assim como podemos definir o tipo `Aluno` da seguinte forma:

```
typedef struct {
  int ra;
  double nota;
} Aluno;
```

Exemplo de uso do typedef

```c
#include <stdio.h>

/* Definindo o tipo "enum mes" */
enum mes {jan, fev, mar, abr, mai, jun,
          jul, ago, set, out, nov, dez};




int main() {
  enum mes a = fev, b = jun;

  printf("a = %d\n", a);
  printf("b = %d\n", b);

  return 0;
}
```

Exemplo de uso do typedef

```
#include <stdio.h>

/* Definindo o tipo "enum mes" */
enum mes {jan, fev, mar, abr, mai, jun,
          jul, ago, set, out, nov, dez};

/* Redefinindo o tipo "enum mes" como "mes" */
typedef enum mes mes;

int main() {
  mes a = fev, b = jun;

  printf("a = %d\n", a);
  printf("b = %d\n", b);

  return 0;
}
```

Exemplo de uso do typedef

```
#include <stdio.h>

/* Definindo o tipo "mes" */
typedef enum {jan, fev, mar, abr, mai, jun,
              jul, ago, set, out, nov, dez} mes;




int main() {
  mes a = fev, b = jun;

  printf("a = %d\n", a);
  printf("b = %d\n", b);

  return 0;
}
```

Exemplo de uso do typedef

```c
#include <stdio.h>

/* Definindo o tipo "struct Aluno" */
struct Aluno {
  int ra;
  double nota;
};

/* Redefinindo o tipo "struct Aluno" como "Aluno" */
typedef struct Aluno Aluno;
```

Exemplo de uso do typedef

```
#include <stdio.h>

/* Definindo o tipo "Aluno" */
typedef struct {
  int ra;
  double nota;
} Aluno;
```

Exemplo de uso do typedef

```
int main () {
  Aluno turma[10];
  int i;
  double nota = 0.0;

  for (i = 0; i < 10; i++) {
    printf("Digite o RA do aluno: ");
    scanf("%d", &turma[i].ra);
    printf("Digite a nota do aluno: ");
    scanf("%lf", &turma[i].nota);
  }

  for (i = 0; i < 10; i++)
    nota = nota + turma[i].nota;

  printf("Media da turma: %f\n", nota / 10);
  return 0;
}
```

Turma de alunos

- Suponha que queremos imprimir as informações dos alunos de uma turma (RA e nome) formada por n alunos, onde o valor de n não é conhecido a priori.
- Além da ordem original dos alunos, deseja-se também imprimir as informações ordenadas de outras duas formas:
  - Em ordem crescente dos números dos RAs.
  - Em ordem lexicográfica dos nomes.

Turma de alunos

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

/* Definicao do tipo Aluno */
typedef struct {
  int ra;
  char nome[52];
} Aluno;

void imprime_turma(Aluno turma[], int n) {
  int i;

  printf("*** Turma ***\n");
  for (i = 0; i < n; i++)
    printf("%06d %s", turma[i].ra, turma[i].nome);
  printf("\n");
}
```

Turma de alunos

```
void ordena_por_ra(Aluno turma[], int n) {
  int i, j;
  Aluno aux;

  for (i = 1; i < n; i++) {
    aux = turma[i];
    j = i - 1;

    while ((j >= 0) && (turma[j].ra > aux.ra)) {
      turma[j + 1] = turma[j];
      j--;
    }

    turma[j + 1] = aux;
  }
}
```

Turma de alunos

```
void ordena_por_nome(Aluno turma[], int n) {
  int i, j;
  Aluno aux;

  for (i = 1; i < n; i++) {
    aux = turma[i];
    j = i - 1;

    while ((j >= 0) && (strcmp(turma[j].nome, aux.nome) > 0)) {
      turma[j + 1] = turma[j];
      j--;
    }

    turma[j + 1] = aux;
  }
}
```

Turma de alunos

```
int main () {
  Aluno *turma;
  int n, i;

  printf("Quantos alunos? ");
  scanf("%d", &n);

  turma = malloc(sizeof(Aluno) * n);

  for (i = 0; i < n; i++) {
    printf("Aluno numero %d:\n", i + 1);
    printf("RA: ");
    scanf("%d ", &turma[i].ra);
    printf("Nome: ");
    fgets(turma[i].nome, 52, stdin);
  }
  ...
```

Turma de alunos

```
  ...
  /* Turma na ordem inicial */
  imprime_turma(turma, n);

  ordena_por_ra(turma, n);

  /* Turma ordenada por ra */
  imprime_turma(turma, n);

  ordena_por_nome(turma, n);

  /* Turma ordenada por nome */
  imprime_turma(turma, n);

  free(turma);

  return 0;
}
```

Turma de alunos

```
*** Turma ***
152034 Ronald Bilius Weasley
161212 Hermione Jean Granger
149826 Harry James Potter

*** Turma ***
149826 Harry James Potter
152034 Ronald Bilius Weasley
161212 Hermione Jean Granger

*** Turma ***
149826 Harry James Potter
161212 Hermione Jean Granger
152034 Ronald Bilius Weasley
```

Ponteiros para registros

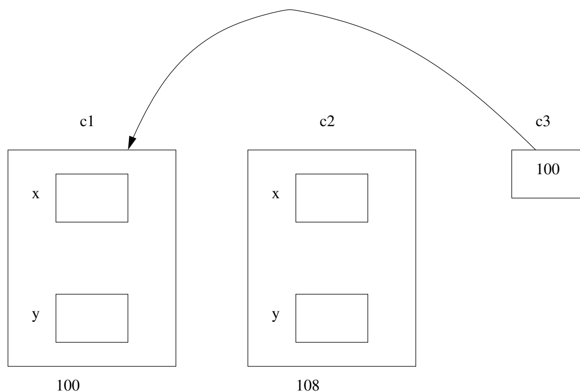
- Ao criarmos uma variável do tipo struct, esta é armazenada na memória como qualquer outra variável e, portanto, possui um endereço.
- Sendo assim, é possível criar um ponteiro para uma variável de um tipo struct.

```
#include <stdio.h>

typedef struct {
  float x, y;
} Coordenadas;

int main() {
  Coordenadas c1, c2, *c3;
  c3 = &c1;
  ...
```

Ponteiros para registros



Ponteiros para registros

```
#include <stdio.h>

typedef struct {
  float x, y;
} Coordenadas;

int main() {
  Coordenadas c1, c2, *c3;

  c3 = &c1;
  c1.x = -1;
  c1.y = -1.5;
  c2.x = 2.5;
  c2.y = -5;
  *c3 = c2;

  printf("Coordenadas: (%0.1f, %0.1f)\n", c1.x, c1.y);

  return 0;
}
```

Ponteiros para registros

```
#include <stdio.h>

typedef struct {
  float x, y;
} Coordenadas;

int main() {
  Coordenadas c1, c2, *c3;

  c3 = &c1;
  c1.x = -1;
  c1.y = -1.5;
  c2.x = 2.5;
  c2.y = -5;
  *c3 = c2;

  printf("Coordenadas: (%0.1f, %0.1f)\n", c1.x, c1.y); /* (2.5, -5.0) */

  return 0;
}
```

Ponteiros para registros

- Para se ter acesso aos campos de um registro (`struct`) através de um ponteiro, podemos utilizar o operador '*' juntamente com o operador '.', como usualmente:

```
Coordenadas c1, *c3;
c3 = &c1;
(*c3).x = 1.5;
(*c3).y = 1.5;
```

- Também podemos usar o operador '->', que permite acesso aos campos de um registro através de um ponteiro.
- Podemos reescrever o exemplo anterior da seguinte forma:

```
Coordenadas c1, *c3;
c3 = &c1;
c3->x = 1.5;
c3->y = 1.5;
```

Ponteiros para registros

- Resumindo: para ter acesso a um campo de um registro através de um ponteiro, podemos usar uma destas duas formas equivalentes:
  - Usando os operadores '*' e '.':

```
(*ponteiroRegistro).campo
```

  - Usando o operador '->':

```
ponteiroRegistro->campo
```

Ponteiros para registros

```
#include <stdio.h>

typedef struct {
  float x, y;
} Coordenadas;

int main() {
  Coordenadas c1, c2, *c3, *c4;

  c3 = &c1;
  c4 = &c2;

  c1.x = -1;
  c1.y = -1.5;

  c2.x = 2.5;
  c2.y = -5;

  ...
```

Ponteiros para registros

```
  ...

  (*c3).x = 1.5;
  (*c3).y = 1.5;

  c4->x = 2;
  c4->y = 3;

  printf("Coordenadas: (%0.1f, %0.1f)\n", c1.x, c1.y);
  printf("Coordenadas: (%0.1f, %0.1f)\n", c2.x, c2.y);

  return 0;
}
```

Ponteiros para registros

```
  ...

  (*c3).x = 1.5;
  (*c3).y = 1.5;

  c4->x = 2;
  c4->y = 3;

  printf("Coordenadas: (%0.1f, %0.1f)\n", c1.x, c1.y); /* (1.5, 1.5) */
  printf("Coordenadas: (%0.1f, %0.1f)\n", c2.x, c2.y); /* (2.0, 3.0) */

  return 0;
}
```

Controle de estoque

- Vamos criar uma pequena aplicação para manter um controle de estoque com as seguintes informações para cada item:
  - Nome
  - Quantidade
  - Preço
- Além disso, nosso programa deverá ter opções para incluir/excluir um item do estoque.

Controle de estoque

- Usaremos o seguinte registro para representar um item do estoque:

```
typedef struct {
  char nome[52];
  int quantidade;
  double preco;
  int emUso;
} Item;
```

- Usaremos um vetor para armazenar o estoque.
- O campo `emUso` de Item servirá para indicar se no vetor uma determinada posição está em uso (1) ou não (0).

Controle de estoque

Vamos criar as seguintes funções:

- Inicializa o vetor usado para armazenar os itens do estoque:
  ```
  void inicializaEstoque(Item estoque[], int tam);
  ```
- Cadastra um item para posteriormente ser incluído no estoque:
  ```
  void cadastraItem(Item *item);
  ```
- Imprime os dados de um item:
  ```
  void imprimeItem(Item item);
  ```
- Imprime os dados de todos os itens do estoque:
  ```
  void imprimeEstoque(Item estoque[], int tam);
  ```
- Insere um novo item no estoque, se houver espaço disponível:
  ```
  int insereItem(Item estoque[], int tam, Item item);
  ```
- Remove um item de nome dado, se ele estiver no estoque.
  ```
  int removeItem(Item estoque[], int tam, char nome[]);
  ```

Controle de estoque

```
void inicializaEstoque(Item estoque[], int tam) {
  int i;

  /* Inicialmente nenhuma posicao do estoque esta em uso */
  for (i = 0; i < tam; i++)
    estoque[i].emUso = 0;
}
```

Controle de estoque

```
void cadastraItem(Item *item) {
  printf("----- Cadastrando um Item -----\n");

  printf("Digite o nome do item: ");
  scanf(" "); /* Descarta espacos em branco */
  fgets(item->nome, 52, stdin);

  printf("Digite a quantidade do item: ");
  scanf("%d", &(item->quantidade));

  printf("Digite o preco do item: ");
  scanf("%lf", &(item->preco));
}
```

Controle de estoque

```
void imprimeItem(Item item) {
  printf("----- Imprimindo Item -----\n");
  printf("Nome: %s\n", item.nome);
  printf("Quantidade: %d\n", item.quantidade);
  printf("Preco: R$%0.2f\n", item.preco);
}

void imprimeEstoque(Item estoque[], int tam) {
  int i;

  for (i = 0; i < tam; i++)
    /* Se a posicao i estiver em uso, imprime o item */
    if (estoque[i].emUso)
      imprimeItem(estoque[i]);
}
```

Controle de estoque

```
int insereItem(Item estoque[], int tam, Item item) {
  int i;

  for (i = 0; i < tam; i++)
    /* Se a posicao i nao estiver em uso ... */
    if (estoque[i].emUso == 0) {
      estoque[i] = item;
      /* ... a posicao i passa a estar em uso */
      estoque[i].emUso = 1;

      /* Item inserido com sucesso */
      return 1;
    }

  /* Nao foi possivel inserir o item */
  return 0;
}
```

Controle de estoque

```
int removeItem(Item estoque[], int tam, char nome[]) {
  int i;

  for (i = 0; i < tam; i++)
    /* Se achou o item no estoque... */
    if ((estoque[i].emUso) &&
        (strcmp(estoque[i].nome, nome) == 0)) {
      /* ... remove o item do estoque */
      estoque[i].emUso = 0;

      /* Item removido com sucesso */
      return 1;
    }

  /* Nao foi possivel remover o item */
  return 0;
}
```

Exercícios

- Implemente uma função como o seguinte protótipo:
  `int buscaItem(Item estoque[], int tam, char nome[]);`
  Sua função deve verificar se existe um item cadastrado com o nome dado. Se existir, deve retornar o índice do item no vetor que representa o estoque. Caso contrário, deve retornar `-1`.
- Altere a função `insereItem` de tal forma a garantir que nunca existirão dois itens no estoque com o mesmo nome.
- Altere as funções `insereItem` e `imprimeItem` (e por consequência, também a função `imprimeEstoque`) de tal forma que estas funções recebam um ponteiro para o item de interesse e não uma cópia do mesmo, como nas implementações atuais destas funções.
- Implemente uma função que dado o vetor que representa o estoque, seu tamanho, um nome e um inteiro x (não necessariamente positivo), incremente a quantidade de itens com aquele nome registrado no estoque em x unidades.

Exercícios

- Escreva uma função que, dados dois registros do tipo `data` (com campos dia, mês e ano), retorne o número de dias entre as duas datas.
- Escreva uma função que, dado um polinômio do segundo grau $P_2(x)$ (representado através de um registro) e um valor x, retorne o valor do polinômio em x.
- Escreva uma função que, dado um registro `triangulo` (formado por 3 pontos, cada um deles representado por um registro do tipo `Coordenadas`), determine se o triângulo é equilátero, isósceles ou escaleno.