Programando com registros
Professor Wladimir A. Tavares

Organizando os dados

Considere a seguinte situação:

Queremos fazer um programa que manipula datas comemorativas, podemos representar uma data por três inteiros: `int dia`, `int mes`, `int ano` e `char evento[50]`.

```
int dia;
int mes;
int ano;
char evento[50];
```

Se quisermos trabalhar com 10 datas diferentes podemos fazer da seguinte maneira:

```
int dia[10];
int mes[10];
int ano[10];
char evento[10][50];
```

A maneira acima possui várias desvantagens e precisamos sempre lembrar que `dia[i]`, `mes[i]`,`ano[i]` estão associadas a mesma data.

A linguagem C permite que tratemos dados relacionados como uma única unidade lógica. Essa organização facilita o entendimento e a manipulação dessas informações que pertencem a um mesmo contexto.

Declarando registros

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

struct Data{
    int dia;
    int mes;
    int ano;
    char evento[50];
};


int main(){
    struct Data d1 = {7,9,1822,"Dia da Independencia"};
    printf("%02d/%02d/%04d : %s", d1.dia, d1.mes, d1.ano, d1.evento);
}
```

Criando Apelidos

```c
#include <stdio.h>
#include <string.h>
struct Data{
    int dia;
    int mes;
    int ano;
    char evento[50];
};
typedef struct Data Data;
```

Manipulando registros com funções

Seria interessante fazer uma função que mostrasse uma data. Podemos fazer da seguinte maneira:

```c
#include <stdio.h>
#include <string.h>
struct Data cria_data(int dia, int mes,
    int ano,char evento[]){
    struct Data d;
    d.dia = dia;
    d.mes = mes;
    d.ano = ano;
    //error: assignment to expression with array type
    d.evento = evento;
    return d;
}
```

Consertando a função

```c
#include <stdio.h>
#include <string.h>
struct Data cria_data(int dia, int mes,
    int ano,char evento[]){
    struct Data d;
    d.dia = dia;
    d.mes = mes;
    d.ano = ano;
    strcpy(d.evento, evento); //string.h
    return d;
}
```

Comparando Datas

```c
int compara_data(struct Data d1, struct Data d2){
    if( d1.ano < d2.ano) return -1;
    else if( d1.ano > d2.ano) return 1;
    else if( d1.mes < d2.mes ) return -1;
    else if( d1.mes > d2.mes ) return 1;
    else if( d1.dia > d2.dia) return -1;
    else if( d1.dia < d2.dia) return 1;
    else return 0;
}
```

Registro Ponto

Um ponto 2D pode ser representado por dois inteiros:

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

struct ponto {
    int x;
    int y;
};

typedef struct ponto ponto;

ponto cria_ponto(int x, int y){
    struct Ponto p;
    p.x = x;
    p.y = y;
    return p;
}

float distancia(ponto p1, ponto p2){
    int dx = p1.x - p2.x;
    int dy = p1.y - p2.y;
    return sqrt( dx*dx + dy*dy );
}

void imprime_ponto(ponto p){
    printf("(%d, %d)", p.x, p.y);
}

int main(){
    struct Ponto p1 = cria_ponto(1,1);
    struct Ponto p2 = cria_ponto(2,2);
    imprime_ponto(p1); printf("\n");
    imprime_ponto(p2); printf("\n");
    printf("distancia %f\n", distancia(p1, p2) );
}
```

Processo de compilação

```
$ gcc main.c -o main -lm
```

Registro Retangulo

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

typedef struct retangulo retangulo;

retangulo cria_retangulo(ponto inf_esq, ponto sup_dir){
    retangulo r;
    r.inf_esq = inf_esq;
    r.sup_dir = sup_dir;
    return r;
}

int area_retangulo(retangulo r){
    int base = r.p_sup_dir.x - r.p_inf_esq.x;
    int altura = r.p_sup_dir.y - r.p_inf_esq.y;
    return base*altura;
}

int pertence_retangulo(retangulo r, ponto p){
    return    p.x >= r.p_inf_esq.x &&
            p.x <= r.p_sup_dir.x &&
            p.y >= r.p_inf_esq.y &&
            p.y <= r.p_sup_dir.y;
}
int main(){
    struct Ponto p1 = cria_ponto(1,1);
    struct Ponto p2 = cria_ponto(2,2);
    printf("distancia %f\n", distancia(p1, p2) );
    struct Retangulo r = cria_retangulo(p1, p2);
    printf("area %d\n", area(r) );
    struct Ponto p3 = cria_ponto(1,2);
    printf("%d\n", pertence(r, p3));
}
```

Exercício

1. Faça o registro para `Circulo`
2. Faça a função `cria_circulo(struct Ponto centro, float raio)`
3. Faça a função `imprime_circulo(struct Circulo c)`
4. Faça a função `pertence_circulo(struct Circulo c, struct Ponto p)`

Restringindo valores

Seria interessante se fosse possível restringir os valores do mês para um conjunto de valores válidos. Essa restrição traria várias vantagens como legibilidade, clareza, facilitaria a manutenção.

Eu poderia tentar criar uma lista de constantes para os meses da seguinte maneira:

```c
#include <stdio.h>

#define JANEIRO 1
#define FEVEREIRO 2
#define MARCO 3
#define ABRIL 4
#define MAIO 5
#define JUNHO 6
#define JULHO 7
#define AGOSTO 8
#define SETEMBRO 9
#define OUTUBRO 10
#define NOVEMBRO 11
#define DEZEMBRO 12

struct Data{
    int dia;
    int mes;
    int ano;
};
void imprime_data(struct Data d){
    printf("%02d/%02d/%4d\n", d.dia, d.mes, d.ano);
}
int main(){
    struct Data data;
    data.dia = 1;
    data.mes = JANEIRO;
    data.ano = 1971;
    imprime_data(data);
}
```

Trabalhando com enumerações

A linguagem C permite a criação de uma lista de constantes que podem ser tratadas com um tipo. No caso, podemos criar uma lista de constantes relacionadas com os meses do ano.

```c
#include <stdio.h>


enum Mes{
JANEIRO = 1,
FEVEREIRO = 2,
MARCO = 3,
ABRIL = 4,
MAIO = 5,
JUNHO = 6,
JULHO = 7,
AGOSTO = 8,
SETEMBRO = 9,
OUTUBRO = 10,
NOVEMBRO = 11,
DEZEMBRO = 12
};

typedef enum Mes Mes;


struct Data{
    int dia;
    Mes mes;
    int ano;
};

struct Data cria_data(int dia, Mes mes, int ano){
    Data d;
    d.dia = dia;
    d.mes = mes;
    d.ano = ano;
    return d;
}

void imprime_data(Data d){
    printf("%02d/%02d/%4d\n", d.dia, d.mes, d.ano);
}

int main(){
    struct Data data1 = cria_data(1,JANEIRO,1971);
    imprime_data(data1);
    struct Data data2 = cria_data(1,JANEIRO,1971);
    imprime_data(data2);
}
```

Infelizmente, nem a linguagem C/C++ fornece uma checagem do tipo para enums.

Melhorando a organização do código

Podemos colocar a parte de declaração do tipo Data juntamente com as funções que manipulam em um arquivo separado com o nome `data.h`.

data.h

```c
#ifndef DATA_H  // Previne múltiplas inclusões do arquivo
#define DATA_H
#include <stdio.h>
enum Mes{
JANEIRO = 1,
FEVEREIRO = 2,
MARCO = 3,
ABRIL = 4,
MAIO = 5,
JUNHO = 6,
JULHO = 7,
AGOSTO = 8,
SETEMBRO = 9,
OUTUBRO = 10,
NOVEMBRO = 11,
DEZEMBRO = 12
};
typedef enum Mes Mes;
struct Data{
    int dia;
    Mes mes;
    int ano;
};
typedef struct Data Data;
Data cria_data(int dia, Mes mes, int ano){
    Data d;
    d.dia = dia;
    d.mes = mes;
    d.ano = ano;
    return d;
}
void imprime_data(Data d){
    printf("%02d/%02d/%4d\n", d.dia, d.mes, d.ano);
}
#endif
```

data.c

```c
//main.c
#include <stdio.h>
#include "data.h"
int main(){
    Data data1 = cria_data(1,JANEIRO,1971);
    imprime_data(data1);
}
```

Não é uma boa prática colocar a implementação de funções na arquivo .h. Essa prática pode levar a vários problemas como:

- Aumento no tempo de compilação uma vez que qualquer mudança no .h forçara a compilação de todos os arquivos .c que incluem esse arquivo .h.

Melhorando ainda mais a organização

data.h

```c
//data.h
#ifndef DATA_H  // Previne múltiplas inclusões do arquivo
#define DATA_H

enum Mes{
JANEIRO = 1,
FEVEREIRO = 2,
MARCO = 3,
ABRIL = 4,
MAIO = 5,
JUNHO = 6,
JULHO = 7,
AGOSTO = 8,
SETEMBRO = 9,
OUTUBRO = 10,
NOVEMBRO = 11,
DEZEMBRO = 12
};
typedef enum Mes Mes;

struct Data{
    int dia;
    Mes mes;
    int ano;
};
typedef struct Data Data;
Data cria_data(int dia, Mes mes, int ano);
void imprime_data(Data d);
#endif
```

data.c

```c
//data.c
#include <stdio.h>
#include "data.h"

Data cria_data(int dia, Mes mes, int ano){
    Data d;
    d.dia = dia;
    d.mes = mes;
    d.ano = ano;
    return d;
}
void imprime_data(Data d){
    printf("%02d/%02d/%4d\n", d.dia, d.mes, d.ano);
}
```

main.c

```c
//main.c
#include <stdio.h>
#include "data.h"
int main(){
    Data data1 = cria_data(1,JANEIRO,1971);
    imprime_data(data1);
}
```

Compilação

```
$ make
gcc -Wall -c data.c -o data.o
gcc -Wall -c main.c -o main.o
gcc data.o main.o -o main
$ ./main
01/01/1971
```

Makefile

```
# Compilador e flags
CC = gcc
CFLAGS = -Wall

data.o : data.c
    $(CC) $(CFLAGS) -c data.c -o data.o

main.o : main.c
    $(CC) $(CFLAGS) -c main.c -o main.o

all: data.c main.c
    $(CC) data.o main.o -o main
```

Compilação

```
$ make all
gcc data.o main.o -o main
```