A introdução e prática de estrutura para a vida de um programador

```
struct rational {
        int num;
        int den;
};
```

Criando estrutura com conjunto de dois inteiros declarados - uma quantidade de dados sendo aglomerado = ‘struct’ ;
terminar com o ponto e vírgula é uma regra para a criação das estruturas;
Lembre-se que número racional nunca terá denominador ZERO.
Saiba que;
-Você não está construindo uma estrutura, você está declarando para o compilador e isso não compõe exatamente uma parte do seu programa em si; ou seja, não é um objeto existente na memória, é uma informação pro compilador
Logo, vc n está pedindo para alocar espaço, por isso vc não vai inicializar de forma alguma.
exemplo do erro:

```
struct point {
  int x = 0;
  int y = 0;
}
```

Como não alocamos espaço e apenas declaramos os inteiros, essa forma está completamente insensata. Então isso não existe...

Para usá-lo podemos apenas chamar a estrutura, tal qual fazemos quando vamos chamar um int, porém chamando a estrutura, veja:
struct point p; - errado: point p;
Pois assim vc diz pro compilador: preciso alocar espaço para eu colocar algo, mas que seja do tamanho da struct 'point', e vamos chamá-la de 'p'.
Agora sim podemos escrever os inteiros e coisas do tipo dentro dessa estrutura.

Vejamos um exemplo simples:

```
#include <stdio.h>

int main(void) {

        struct point{

                short int x; /*para 'x' vou precisar de 2 bytes*/
                short int y; /*para 'y' vou precisar de 2 bytes*/

        }; /*declaramos pro compilador uma estrutura de dois inteiros por 2 bytes cada para utilização*/
```

```
	/*declarando os pontos da estrutua e alocar espaço*/
	struct point p1;
	struct point p2;


	/*a sintaxe para escrever um número em 'x' de struct point:
```

Nome da região de memória onde está alocando a estrutura p1 + '.x' e o valor '= 0' => valor de 'x' sendo indicado.
Perceba que seria impossível fazer isso sem alocarmos um espaço na memória (p1). Para o 'y' funcionará da mesma forma;
Veja:

```
	p1.x = 0; p1.y = 0 /*Ponto Origem (0,0)*/
	p2.x = 1; p2.y = 1 /*Ponto (1,1*)/

	/*Impressões dos números*/
	printf("A origem do plano: (%d, %d).\n", p1.x, p1.y);
	printf("Um ponto no plano: (%d, %d).\n", p2.x, p2.y);
	printf("O tamanho de /struct point/: %ld.\n", sizeof p1);
	printf("O tamanho de /struct point/: %ld.\n", sizeof p2);
	printf("O tamanho de /struct point/: %ld.\n", sizeof (struct point));
	printf("O tamanho de /struct point */: %ld.\n", sizeof (struct point *));
	printf("O tamanho de /x/: %ld.\n", sizeof p1.x); /*teste para ver o tamanho de cada ponto (2bytes) na tela*/
	printf("O tamanho de /y/: %ld.\n", sizeof p1.y); /*teste para ver o tamanho de cada ponto (2bytes) na tela*/

	return 0;
}
```

Nele podemos observar em prática o que apresentamos anteriormente, com os comentários adicionais nos códigos podemos entender também como funcionam cada parte do código.

Em resumo, a gente apresenta uma estrutura de dados, mostrando pro compilador como ele funcionará, para depois alocarmos um espaço na memória com base na estrutura que criamos, assim
conseguindo declarar um valor para cada inteiro e usá-los em nosso código. Essa forma nos permite usar a mesma estrutura em diversas situações diferentes no código sem precisar escrever
códigos que fazem a mesma função mais de uma vez. Uma estrutura pode conter arrays/listas, o que nos deixa claro que ela pode ser uma estrutura grande.
Assim, é legal deixar bem claro para sabermos que as structs e os procedimentos não são como arrays, elas são mais como inteiros e caracteres, ou seja, se usar uma struct como argumento, ela será copiada para o procedimento.
Já com o array, ele acaba não sendo passado, e sim o seu endereço de memória, fica como um ponteiro para esse array. Isso acaba deixando esse modo de estrutura extremamente útil.

(Em outras palavras: estrutura não são como arrays, que decaem para ponteiros quando passados para procedimentos, elas são copiadas para dentro do procedimento em sí)
Para facilitar ainda mais, podemos usar o typedef para não precisarmos escrever a estrutura em todo lugar, ou seja, estamos criando um sinônimo para nossa estrutura.
Para ficar mais fácil de entender, veja o código abaixo:

```
#include <stdio.h>

struct rational {
        int num;
        int den;
};

struct rational mul(struct rational r1, struct rational r2) {
        struct rational ret;
        ret.num = r1.num * r2.num;
        ret.den = r1.den * r2.den;
        return ret;
}

int main(void) {
        struct rational a;
        struct rational b;}
        a.num = 4; a.den = 3;
        b.num = 3; b.ben = 4;

        struct rational c; c = mul(a, b);

  printf("%d/%d * %d/%d = %d/%d\n", a.num, a.den, b.num, b.den, c.num, c.den);
  return 0;

}
```

Veja que existem várias vezes a chamada da struct 'rational', ou seja, repete-se muito 'struct rational' para usar a estrutura criada. Podemos colocar um nome sinônimo de atalho para facilitar nossa vida, apenas colocando esta linha:
'typedef struct rational TESTE' -> typedef + estrutura declarada + nome sinônimo para ela
-Pode ser colocado o atalho antes de declarar a estrutura, o compilador não se importa, pois seria como eu falasse pro compilador: Quando eu falar em TESTE eu vou estar querendo dizer da estrutura 'rational' que já vai ter sido declarada.

Agora o código poderá ficar assim:

```
#include <stdio.h>

 typedef struct rational teste;

 struct rational {
```

```
        int num;
        int den;
};

teste mul(teste r1, teste r2) {
        teste ret;
        ret.num = r1.num * r2.num;
        ret.den = r1.den * r2.den;
        return ret;
}

int main(void) {
        teste a;
        teste b;}
        a.num = 4; a.den = 3;
        b.num = 3; b.ben = 4;

        teste c; c = mul(a, b);

  printf("%d/%d * %d/%d = %d/%d\n", a.num, a.den, b.num, b.den, c.num, c.den);
  return 0;

}
```

Perceba que ao invés de repetir 'struct rational' estou usando 'teste' e funciona da mesma forma e deixa o código bem mais clean.
Sem falar que, como o compilador passa a conhecer o apelido 'teste', em caso de erros pode ser de grande ajuda e ainda podemos também fazer uma organização,

Agora falaremos sobre aritmética de frações e sua aplicação na linguagem.

Para fazermos contas de fração de números reais e racionais positivos de denominadores diferentes temos duas técnicas usadas que aprendemos em época de colégio:
Com MMC (mínimo múltiplo comum) e cruzando os números, que é a forma mais usada para concurseiros pois perde-se menos tempo.

A 'cruzamento', como já falado, basicamente se resume em multiplicar um denominador com o numerador do número do lado, fazendo um cruzamento, e o novo denominador para esses novos numeradores seria apenas a multiplicação deles.
Em outras palavras, cria um novo denominador para a nova fração, pegando os dois denominadores e multiplicando. Cria os novos numeradores pegando os antigos denominadores e multiplica pelos numeradores da diagonal, ou seja, da fração vizinha.
Por exemplo:

(4*1)= (10*1) =

1 1 (4) + (10) 14 7

__ + ____ = _________________ = ______ = _____

4 10 40 40 20

Já com o MMC seria achá-lo, dividir pelos denominadores e do resultado multiplicar pelo seu numerador e somar tudo. Depois só resta pegar a forma irredutível:

(20/4=5*1) / (20/10=2*1)

1 1 5 + 2 7

__ + ____ = _________________ = ______

4 10 mmc(4,10) é: 20 20

Mais à frente explicaremos melhor o funcionamento do MMC, como aprendemos no ensino fundamental, mostraremos também uma forma de pegar esse resultado de forma simplificada em programação.

Caso queria apenas multiplicar as frações, não tem mistério, apenas multiplicar numerador com numerador e denominador com denominador:

a / b * c / d == a * c / b * d

Já a divisão, será invertendo a múltipla fração e multiplicando:

a / b / c / d == a / b * d / c

ex: 3/5 : 1/2 = 3/5*2/1 = 6/5

Após relembrar essas nossas contas, podemos considerar então que:

a/b + c/d = ad+cb/bd

Então,
seja m = mmc(b,d)

a/b+c/d = a*(m/b) + c*(m/d)/m

Podemos usar essas fórmulas para criar funções que fazem essas contas para colocarmos no programa, mas iremos utilizar apenas a multiplicação.

Mais sobre o MMC:
mínimo múltiplo comum = mmc
. vai achar o menor valor que seja múltiplo de ambos os números e positivo diferente de zero;

Uma forma de vermos isso com clareza é se verificarmos o 4 e o 10, precisamos de um jeito que seja fácil de colocar em programação e que sempre funcione dentro da regra.

Primeiro vamos olhar como é feito da forma tradicional como nos foi ensinado no colegial: Separamos os dois números e dividimos sempre dando preferência ao menor número, até sobrar 1 de cada para depois multiplicarmos os números que usamos na divisão.

_____

10 - 4 | 2
 5 - 2 | 2
 5 - 1 | 5
 1 - 1 | 1
 - - | --
       | 20

_____

2*2 = 4
4*5 = 20
20*1 = 20/

_____

Buscando forma para usar de forma mais simplificada em programação:

4 #### 4
10 ########## 10

4 + 4 ######## 8
10 ########## 10

8 + 4 ############ 12
10 ########## 10

12 ############ 12
10 + 10 ##################### 20

12 + 4 ################# 16
20 ##################### 20

16 + 4 ##################### 20
20 ##################### 20

=

20 ##################### 20

Resumo:
Ir adicionando sempre o valor original de um dos números até ultrapassar o outro, caso ocorra continue a adição até que cheguem no mesmo valor.
Com isso em mente, podemos fazer uma condicional e assim o cálculo de MMC será resolvido facilmente, veja:

```
        if (m == n) return m;
        if (m < n) return mmc0(m + mcopy,n);
        if (n < m) return mmc0(m, n + ncopy);
```

Porém, este código ficará errado, pois sempre que for rodar dnv o m e o n serão alterados, e portanto ele não somará com o valor original e sim o novo valor.
Ou seja, ao invés de ficar '20 + 10' ficará '20 + 20', e por aí vai. Portanto precisamos salvar o original, para isso podemos usar os argumentos;

```
int mmc0(int m, int n, int mcopy, int ncopy) {
```

O 'mcopy' e o 'ncopy' serão, então, os valores originais pra conta.

ficando assim o código:

```
int mmc0(int m, int n, int mcopy, int ncopy) {
        if (m == n) return m;
        if (m < n) return mmc0(m + mcopy,n, mcopy, ncopy);
        if (n < m) return mmc0(m, n + ncopy, mcopy, ncopy);
        return -1 /*colocaremos esse return para evitar o aviso do compilador, pois os if's já irão cobrir todos os casos, ou simplesmente desligar o aviso do compilador*/
/*-- return -1 nunca ocorrerá*/
}
```

Como terá que chamar por 4 argumentos e o usuário acabará colocando apenas dois, podemos criar uma nova função para essa entrada e até fazer com que ela verifique os erros futuramente.

```
int mmc(int m, int n) {
        return mmc0(m, n, m, n);
}
```

Pronto, assim o cálculo de MMC está perfeitamente concluído.

Agora, veremos a parte do código responsável pela multiplicação:

```
struct racional mul(struct racional r1, struct racional r2) {
  struct racional ret;
  ret.num = r1.num * r2.num;
  ret.den = r1.den * r2.den;
  return ret;
}
```

Podemos ver que ele puxou a estrutura racional na criação dessa função, que se baseia pelo código mostrado no início da redação:

```
struct rational {
  int num;
  int den;
};
```

Ou seja, sempre terá um inteiro numerador e um denominador (diferente e maior que zero). Em outras palavras, dois inteiros sendo eles cada parte que compõe uma fração.
Cada argumento também deverá ter a mesma estrutura por serem frações.
A própria estrutura de dentro dessa função deverá ser 'struct racional' por ser uma função para multiplicação dos numeradores e denominadores. Ou seja, será a nova fração feita pelo resultado da multiplicação das duas frações dos argumentos.

Para não ficar sempre escrevendo struct racional, como vimos no início dessa redação, é mais prático usar o typedef e escolher um nome adequado:

'typedef struct rational racional'

agora podemos só usar 'racional' no lugar de 'struct rational':

```
typedef struct rational racional

struct rational {
  int num;
  int den;
};

racional mul(racional r1, racional r2) {
  racional ret;
  ret.num = r1.num * r2.num;
  ret.den = r1.den * r2.den;
  return ret;
}
```

Prontinho... e isso funcionará por todo o código.

Agora que organizamos a estrutura dando um apelido a ele e vimos a função de multiplicação de numerador e denominador, além de termos visto como funcionará nossas funções de MMC,
podemos passar agora para a função 'main' do programa.

Nela poderemos utilizar o apelido 'racional' e colocaremos essas funções que criamos, como 'mmc' e 'mul', e teremos nossa fração para o cálculo de multiplicação e o controle de impressão de tela.

O código em si>

```
int main(void) {
  racional a;
  racional b;
  a.num = 4; a.den = 3; /* 4/3 */
  b.num = 3; b.den = 4; /* 3/4 */

  racional c; c = mul(a, b);

  printf("%d/%d * %d/%d = %d/%d\n", a.num, a.den, b.num, b.den, c.num, c.den);

  printf("mmc(%d, %d) == %d\n", 2, 5, mmc(2,5));

  printf("mmc(%d, %d) == %d == %d\n", 2, 5, mmc(2,5), 10); /*verificações*/
  printf("mmc(%d, %d) == %d == %d\n", 3, 6, mmc(3,6), 6); /*verificações*/
  printf("mmc(%d, %d) == %d == %d\n", 1, 1, mmc(1,1), 1); /*verificações*/
  printf("mmc(%d, %d) == %d == %d\n", 1, 11001, mmc(1,11001), 11001); /*verificações*/

  return 0;
}
```

Criamos uma 'fração' chamada 'a' e uma chamada 'b', colocamos que o 'num' da 'a' será = 4 e o 'den' = 3; já o da 'b' será 'num' = 3 e 'den' = 4.

Nós podemos criar outras funções para a soma ou a divisão de frações, por exemplo, mas como só criamos a multiplicação de frações na função mul(), faremos 4/3 * ¾.

Para isso temos que criar a fração 'c' e usar a multiplicação de 'a' e 'b', assim armazenando o resultado dessa multiplicação, assim poderemos printar esse resultado depois.

Agora podemos ver na tela o que quisermos, os numeradores e denominadores de cada função, o MMC, e por aí vai, até onde a imaginação deixar.

Por exemplo, quero ver a multiplicação das frações:
printf("%d/%d * %d/%d = %d/%d\n", a.num, a.den, b.num, b.den, c.num, c.den);

Quero saber o MMC de 2 e 5: printf("mmc(%d, %d) == %d\n", 2, 5, mmc(2,5));

ETC…

Podemos criar/implementar uma função para a soma de frações utilizando o MMC e usando da fórmula que eu especifiquei lá atrás, indo incrementando esses conteúdos a mais para deixar o programa com mais funcionalidade.

Veja, abaixo, um dos testes que estava fazendo para a soma de frações:

```
racional soma(racional k, racional l) {
 /*
 fração -> k.num = a k.den = b // l.num = c l.den = d
 */

   racional conta;
   conta.num = (l.den * k.num) + (k.den * l.num);
   conta.den = (k.den * l.den);
/*  a/b + c/d = ad+cb/bd = (l.den*k.num) + (k.den*l.num) / (k.den * l.den)  */
   return conta;
}
```

Porém, para isso funcionar legal necessitaria de uma função para fazer o resultado ficar em uma fração irredutível.
Além de podermos adicionar essas funções a mais, podemos também colocar o MDC e entre outras coisas. Depois basta finalizarmos no main e colocar os prints adequados com nossas novas funcionalidades.

Então aprendemos como organizar o código com o “typedef”, como funciona a estrutura, sua utilidade e como trabalhamos com números racionais na linguagem C.