Fundamentos de Programação

structs

Structs, também conhecidas como **Registros**, definem tipos de dados que agrupam variáveis sob um mesmo tipo de dado.

A ideia de usar uma **struct** é permitir que, ao armazenar os dados de uma mesma entidade, isto possa ser feito com uma única variável. Por exemplo, se for preciso armazenar a *altura*, o *peso* e a idade de uma pessoa, pode-se criar uma **struct** chamada **Pessoa** e agrupar os dados em um único tipo de dado.

Em C, structs (ou estruturas) são usadas para agrupar diferentes tipos de dados sob um único nome, permitindo a criação de tipos de dados complexos. Elas são mais comparáveis a classes simples ou registros e dicionários em outras linguagens de programação.

Definição de uma Struct

O comando typedef em C é usado para criar um alias para tipos de dados existentes. Isso pode simplificar a sintaxe e tornar o código mais legível, especialmente quando se trabalha com tipos complexos, como structs

```c
typedef struct // Cria uma STRUCT para armazenar os dados de uma pessoa
{
    float Peso;   // define o campo Peso
    int Idade;    // define o campo Idade
    float Altura; // define o campo Altura
} Pessoa; // Define o nome do novo tipo criado
```

Declaração e manipulação de uma struct

```
Pessoa Joao, P1, P2;
Pessoa Povo[10];  // cria um vetor de 10 pessoas.

Joao.Idade = 15;
Joao.Peso = 60.5;
Joao.Altura = 1.75;

Povo[4].Idade = 23;
Povo[4].Peso = 75.3;
Povo[4].Altura = 1.89;
```

Exercício: Cadastro de Alunos

Crie um programa em C que utilize structs para armazenar informações sobre alunos. O programa deve:

1. Definir uma struct chamada Aluno com os seguintes campos:
    1. Nome (uma string de até 50 caracteres)
    2. Idade (um inteiro)
    3. Nota (um float)

2. Declarar um array de structs para armazenar informações de até 3 alunos.
3. Permitir ao usuário inserir dados para cada aluno.
4. Exibir as informações de todos os alunos cadastrados.

Passagem de Structs por Parâmetro

Para passar uma struct **por valor** basta declará-la como um dos parâmetros, como no exemplo a seguir

```c
void ImprimePessoa(Pessoa P) // declara o parâmetro como uma struct
{
  printf("Idade: %d  Peso: %f Altura: %f\n", P.Idade, P.Peso, P.Altura);
}

int main()
{
    Pessoa Joao;

    Joao.Idade = 15;
    Joao.Peso = 60.5;
    Joao.Altura = 1.75;

    // chama a função que recebe a struct como parâmetro
    ImprimePessoa(Joao);
```

Retorno de Structs em Funções

Como uma struct define um tipo de dado, este tipo pode ser retornado em uma função, da mesma forma que ocorre com qualquer outro tipo de dado.

```
Pessoa SetPessoa(int idade, float peso, float altura)
{
    Pessoa P;
    P.Idade = idade;
    P.Peso = peso;
    P.Altura = altura;
    return P; // retorna a struct contendo os dados passados por parâmetro
}
void ImprimePessoa(Pessoa P) // declara o parâmetro como uma struct
{
  printf("Idade: %d  Peso: %f Altura: %f\n", P.Idade, P.Peso, P.Altura);
}

int main()
{
    Pessoa Joao;

    Joao = SetPessoa(15,60.5,1.75); // atribui o retorno da função a uma variável struct
    ImprimePessoa(Joao);
   return 0;
}
```

Passagem de Structs por Referência

Para passar uma struct por referência, deve-se passar um ponteiro para a struct, como no exemplo a seguir.

```
void SetPessoa(Pessoa *P, int idade, float peso, float altura)
{ // Nesta função o parâmetro P é um ponteiro para uma struct
    (*P).Idade = idade;  // o campo pode ser acessado desta forma
    P->Peso = peso;      // ou desta
    P->Altura = altura;
}

int main()
{
    Pessoa Joao;
    SetPessoa(&Joao, 15, 70.5, 1.75);
    ImprimePessoa(Joao);

    return 0;
}
```

Exercício: Função que Retorna uma Struct

Crie um programa em C que utilize uma função para criar e retornar uma struct Aluno. O programa deve:

1.Definir uma struct chamada Aluno com os seguintes campos:

1. Nome (uma string de até 50 caracteres)
2. Idade (um inteiro)
3. Nota (um float)

2.Implementar uma função chamada criarAluno que recebe nome, idade e nota como parâmetros e retorna uma struct Aluno preenchida com esses valores.

3.Na main, chame a função criarAluno para criar um aluno e exiba as informações do aluno criado.

Fundamentos de Programação

Endereços e ponteiros

Endereços

A memória RAM (random access memory) de qualquer computador é uma sequência de bytes. A posição (0, 1, 2, 3, etc.) que um byte ocupa na sequência é o endereço (= address) do byte. (É como o endereço de uma casa em uma longa rua que tem casas de um lado só.) Se e é o endereço de um byte então e + 1 é o endereço do byte seguinte.

Cada variável de um programa ocupa um certo número de bytes consecutivos na memória do computador. Uma variável do tipo char ocupa 1 byte. Uma variável do tipo int ocupa 4 bytes e um double ocupa 8 bytes em muitos computadores. O número exato de bytes de uma variável é dado pelo operador sizeof. A expressão sizeof (char), por exemplo, vale 1 em todos os computadores e a expressão sizeof (int) vale 4 em muitos computadores.

Cada variável (em particular, cada registro e cada vetor) na memória tem um endereço

```
char c;
int i;
struct {
    int x, y;
} ponto;
int v[4];
```

c	*89421*
i	*89422*
ponto	*89426*
v[0]	*89434*
v[1]	*89438*
v[2]	*89442*

Fonte: ICMC/USP

O endereço de uma variável é dado pelo operador & . Assim, se i é uma variável então &i é o seu endereço. (Não confunda esse uso de & com o operador lógico and, representado por && em C.)

Exemplo: o segundo argumento da função de biblioteca scanf é o endereço da variável que deve receber o valor lido do teclado:

```
int i;
scanf ("%d", &i);
```

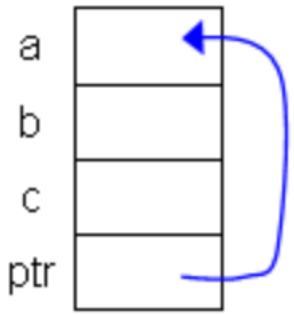
E se eu quisesse saber o tamanho total de uma Struct???

Um ponteiro (apontador = pointer) é um tipo especial de variável que armazena um endereço. Um ponteiro pode ter o valor NULL que é um endereço inválido.

A macro NULL está definida na interface stdlib.h e seu valor é 0 (zero) na maioria dos computadores.

Se um ponteiro p armazena o endereço de uma variável i, podemos dizer p aponta para i ou p é o endereço de i

```c
int a;
int *ptr;  // declara um ponteiro para um inteiro
           // um ponteiro para uma variável do tipo inteiro
a = 90;
ptr = &a;
printf("Valor de ptr: %p, Conteúdo de ptr: %d\n", ptr, *ptr);
```



Acessando o Conteúdo de uma Posição de Memória através de um Ponteiro

Para acessar o conteúdo de uma posição de memória, cujo endereço está armazenado em um ponteiro, usa-se o operador de **derreferência** (*)

```c
#include <stdio.h>
void main()
{
  int x;
  int *ptr;
  x = 5;
  ptr = &x;
  printf("O valor da variável X é: %d\n", *ptr);  // derreferenciando um ponteiro
  *ptr = 10;  // usando derreferencia no "lado esquerdo" de uma atribuição
  printf("Agora, X vale: %d\n", *ptr);
}
```

Tipos de ponteiros

Há vários tipos de ponteiros: ponteiros para bytes, ponteiros para inteiros, ponteiros para ponteiros para inteiros,etc . O computador precisa saber de que tipo de ponteiro você está falando. Para declarar um ponteiro p para um inteiro, escreva

```
int *p;
```

Um ponteiro r para um ponteiro que apontará um inteiro (como no caso de uma matriz de inteiros) é declarado assim:

```
int **r;
```

Exemplo prático

```c
int *p;
int **r; // ponteiro para ponteiro para inteiro
p = &a;  // p aponta para a
r = &p;  // r aponta para p e *r aponta para a
c = **r + b;
```

Trocando valores de variáveis

```
void troca (int *p, int *q)
{
    int temp;
    temp = *p; *p = *q; *q = temp;
}
```

Para aplicar essa função às variáveis i e j basta dizer troca (&i, &j) ou então

```
int *p, *q;
p = &i;
q = &j;
 troca (p, q);
```

Alocação Dinâmica de Memória

Durante a execução de um programa, pode-se alocar dinamicamente memória para usar como variáveis do programa. Em C, a alocação é feita com a função **`malloc`** (e, para liberar memória, usa-se a função **`free`**).

```c
int *ptr_a;

ptr_a = malloc(sizeof(int));
// cria a área necessária para 01 inteiro e
// coloca em 'ptr_a' o endereço desta área.

 if (ptr_a == NULL)
 {
   printf("Memória insuficiente!\n");
   exit(1);
 }

 printf("Endereço de ptr_a: %p\n",  ptr_a);
 *ptr_a = 90;
 printf("Conteúdo de ptr_a: %d\n", *ptr_a);  // imprime 90
 free(ptr_a);  // Libera a área alocada
```

Alocação dinâmica de vetores

```c
int i;
int *v;
v = (int*)malloc(sizeof(int)*10);  // 'v' é um ponteiro para uma área que
                                   // tem 10 inteiros.
                                   // 'v' funciona exatamente como um vetor
v[0] = 10;
v[1] = 11;
v[2] = 12;
// continua...
v[9] = 19;

for(i = 0; i < 10; i++)
  printf("v[%d]: %d\n", i, v[i]);

printf("Endereço de 'v': %p", v);  // imprime o endereço da área alocada para 'v'
free(v);
```

Faça um programa que lê um vetor de n elementos e um número inteiro Em seguida o programa deve fazer a multiplicação do vetor pelo número. Use alocação dinâmica.

Fundamentos de Programação

Cabeçalhos

Arquivos-cabeçalhos são aqueles que temos mandado o compilador incluir no início de nossos exemplos e que sempre terminam em .h. A extensão .h vem de header (cabeçalho em inglês). Já vimos exemplos como stdio.h, stdlib.h, math.h. Estes arquivos, na verdade, não possuem os códigos completos das funções. Eles só contêm protótipos de funções. É o que basta. O compilador lê estes protótipos e, baseado nas informações lá contidas, gera o código correto. O corpo das funções cujos protótipos estão no arquivo-cabeçalho, no caso das funções do próprio C, já estão compiladas e normalmente são incluídas no programa no instante da "linkagem". Este é o instante em que todas as referências a funções cujos códigos não estão nos nossos arquivos fontes são resolvidas, buscando este código nos arquivos de bibliotecas.

Criando nossa própria “biblioteca” com arquivos .h

Criar-se um arquivo .h contendo apenas:

```
#ifndef FOOBAR_H   // guardas de cabeçalho, impedem inclusões cíclicas
#define FOOBAR_H


int foo(int arg); // declaração de uma função
int bar(void);     // outra função

#endif
```

Depois precisamos implementar as funções em um arquivo .C

```c
#include "foobar.h"

int foo(int arg) {
    arg += 5;
    return arg;
}

int bar(void) {
    return 4;
}
```

Exercícios

Crie um programa que vai solicitar ao usuário que informe a quantidade de números que deseja armazenar e, em seguida, vai alocar dinamicamente um array para armazenar esses números. Após a alocação, o programa deve pedir que o usuário insira os números e, por fim, vai exibir os números na ordem inversa.

Crie um programa o qual solicita ao usuário 3 números. Em seguido imprima o endereço de memória onde cada número foi armazenado. Use ponteiros

Exercícios

Sistema de Gerenciamento de Biblioteca
Implementar um sistema de gerenciamento de biblioteca em C utilizando structs. O sistema deve permitir:

1. Cadastrar livros, com título, autor e quantidade disponível.
2. Cadastrar usuários, com nome, ID e lista de livros emprestados.
3. Emprestar um livro para um usuário, diminuindo a quantidade disponível do livro e adicionando o livro à lista de livros emprestados do usuário.
4. Devolver um livro, aumentando a quantidade disponível do livro e removendo o livro da lista de livros emprestados do usuário.
5. Exibir informações sobre os livros cadastrados, os usuários cadastrados e os livros emprestados por cada usuário.

Desafio!

Faça um programa que lê um vetor de n elementos e uma matriz de m x n elementos. Em seguida o programa deve fazer a multiplicação do vetor pelas colunas da matriz. Use alocação dinâmica.