Notas de aula - 20/03

Ponteiros e estruturas

Assim como qualquer outro tipo de dados, é possível criar funções que recebem ponteiros para estruturas ou que retornam ponteiros para estruturas. Porém no caso de estruturas precisamos tomar um cuidado a mais. Considere a estrutura Cliente, que guarda nome, rua e número da casa de um cliente:

```
typedef struct cliente Cliente;
struct cliente {
	char nome[30];
	char rua[30];
	int numero;
};
```

O código abaixo cadastra e imprime um cliente:

```
main() {
	Cliente cliente;
	strcpy(cliente.nome, "Maria");
	strcpy(cliente.rua, "Rua Guarani");
	cliente.numero = 10;

	printf("\nNome: %s\n", cliente.nome);
	printf("Rua: %s\n", cliente.rua);
	printf("Numero: %d\n", cliente.numero);
}
```

Imagine agora que queremos escrever uma função que altera a rua e o número da casa de um cliente. Se escrevermos a função da seguinte forma:

```
void altera_cadastro(Cliente cliente, char rua[], int num) {
	strcpy(cliente.rua, rua);
	cliente.numero = num;
}
```

Ao chamar a função altera_cadastro na função main, nada irá acontecer, pois os parâmetros da função são passados por cópia. Qualquer alteração feita dentro dessa função será feita em cima de uma cópia do cliente e só terá efeito dentro do escopo dessa função.

Podemos contornar esse problema de duas formas.

1) Retornando o cadastro atualizado (usando uma função que retorna uma estrutura - ver notas de aula do dia 14/03):

```
Cliente altera_cadastro(Cliente cliente, char rua[], int num) {
	strcpy(cliente.rua, rua);
	cliente.numero = num;
```

```
      return cliente;
}
```

2) Usando uma função que recebe ponteiros (passagem de parâmetros por referência e não por cópia):

```
void altera_cadastro(Cliente *cliente, char *rua, int num) {
      strcpy((*cliente).rua, rua);
      (*cliente).numero = num;
}
```

Como nessa função recebe ponteiros, ela consegue alterar valores diretamente dentro do endereço de memória, fazendo com que as alterações reflitam também no escopo da função main.

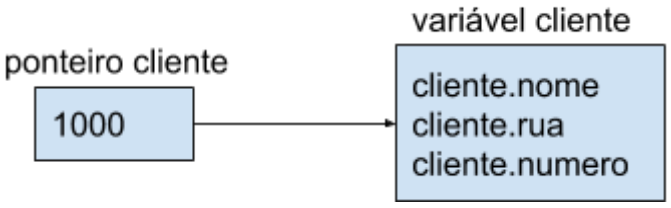
Lembre-se que para alterar o conteúdo apontado por um ponteiro é preciso usar o operador de deferência (*). Se fizéssemos simplesmente:

```
*cliente.numero = num;
```

iria acontecer um erro. Isto é devido a ordem de precedência dos operadores. O operador . tem precedência sobre o operador * (relembre aqui: https://www.ime.usp.br/~pf/algoritmos/apend/precedence.html). Assim, a linha acima seria interpretada como:

```
*(cliente.numero) = num;
```

O ponteiro cliente não possui um campo chamado “numero”. Ele possui apenas um endereço. O campo “numero” faz parte do conteúdo guardado no endereço apontado pelo ponteiro cliente:



Para acessar esse conteúdo corretamente é preciso usar parênteses para alterar a precedência e realizar a operação de deferência primeiro:

```
(*cliente).numero = num;
```

A linha acima pode ser reescrita da seguinte forma:

```
cliente->numero = num;
```

As duas formas são equivalentes, porém, a segunda forma elimina o uso de parênteses, deixando o código mais claro. Este formato também é bem mais comum de ser encontrado.

Assim, a função pode ser reescrita da seguinte maneira:

```
void altera_cadastro(Cliente *cliente, char *rua, int num) {
	strcpy(cliente->rua, rua);
	cliente->numero = num;
}
```

Teste o código abaixo e veja como a função acima consegue alterar o endereço de um cliente corretamente:

```
#include <stdio.h>

typedef struct cliente Cliente;
struct cliente {
	char nome[30];
	char rua[30];
	int numero;
};

void altera_cadastro(Cliente *cliente, char *rua, int num) {
	strcpy(cliente->rua, rua);
	cliente->numero = num;
}

main() {
	Cliente cliente;
	strcpy(cliente.nome, "Maria");
	strcpy(cliente.rua, "Rua Guarani");
	cliente.numero = 10;
	printf("\nNome: %s\n", cliente.nome);
	printf("Rua: %s\n", cliente.rua);
	printf("Numero: %d\n", cliente.numero);

	altera_cadastro(&cliente, "Avenida Tupi", 15);
	printf("\nNome: %s\n", cliente.nome);
	printf("Rua: %s\n", cliente.rua);
	printf("Numero: %d\n", cliente.numero);
}
```

Apesar de funcionar corretamente, o código pode ser melhorado escrevendo a parte de imprimir os dados do cliente em uma função, evitando repetição de código. Abaixo, uma sugestão de função para imprimir os dados do cliente. Veja que como a função de imprimir tem apenas o objetivo de mostrar dados e não de manipular dados, os parâmetros podem ser passados por cópia e ela não precisa de retorno:

```c
#include <stdio.h>

typedef struct cliente Cliente;
struct cliente {
	char nome[30];
	char rua[30];
	int numero;
};

void altera_cadastro(Cliente *cliente, char *rua, int num) {
	strcpy(cliente->rua, rua);
	cliente->numero = num;
}

void imprime_cadastro(Cliente cliente) {
	printf("\nNome: %s\n", cliente.nome);
	printf("Rua: %s\n", cliente.rua);
	printf("Numero: %d\n", cliente.numero);
}

main() {
	Cliente cliente;
	strcpy(cliente.nome, "Maria");
	strcpy(cliente.rua, "Rua Guarani");
	cliente.numero = 10;
	imprime_cadastro(cliente);

	altera_cadastro(&cliente, "Avenida Tupi", 15);
	imprime_cadastro(cliente);
}
```

Quando usar ponteiros

Se precisamos de uma função que altera apenas um valor, podemos simplesmente criar uma função que retorne esse valor, não se faz necessário usar ponteiros.

Quando desejamos criar uma função que altera mais de um valor, usamos ponteiros, para que as alterações feitas dentro da função reflitam no escopo global (uma vez que não conseguimos retornar mais de um valor em uma função em C).

Exemplos:
a) Preciso criar uma função que troque os valores das variáveis “a” e “b”. - Como não vou conseguir retornar os dois valores atualizados, uso ponteiros:

```c
#include <stdio.h>

void troca(int *a, int *b) {
	int t;
	t = *a; // t recebe o conteúdo do end. apontado por a
	*a = *b; // o conteúdo do end. apontado por a recebe o conteúdo do end. apontado por b
	*b = t; // o conteúdo do end. apontado por b recebe o valor de t
}

main() {
	int a = 10;
	int b = 15;
	troca(&a, &b);
	printf("A: %d - B: %d", a, b);
}
```

b) Preciso criar uma função que receba uma string e retorne a string invertida. - Uma string é formada por vários char, portanto não é possível retornar uma string inteira. É possível retornar um ponteiro que aponte para a string.

O que acontece no código abaixo? Ele está imprimindo a string invertida?

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

char inverte_string(char s[]) {
	int i, j, tamanho;
	tamanho = strlen(s);
	char s_invertida[tamanho];

	j = 0;
	for(i = tamanho - 1; i >= 0; i--){
	s_invertida[j] = s[i];
	j++;
	};
	s_invertida[j] = '\0';

	return s_invertida;
}

main() {
	char s[] = "ponteiros";
	printf("%s", inverte_string(s));
}
```

Solução: criar um ponteiro que aponta para a variável s_invertida e retornar esse ponteiro.

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

char * inverte_string(char s[]) {
	int i, j, tamanho;
	tamanho = strlen(s);
	char s_invertida[tamanho];
	char *ps = &s_invertida;

	j = 0;
	for(i = tamanho - 1; i >= 0; i--){
	    s_invertida[j] = s[i];
	    j++;
	}
	s_invertida[j] = '\0';

	return ps;
}

main() {
	char s[] = "ponteiros";
	printf("%s", inverte_string(s));
}
```

c) Preciso criar uma função que retorne um vetor de inteiros, composto por 10 números aleatórios. - Um vetor de inteiros é formado por vários int, portanto não será possível retornar o vetor inteiro. Será necessário retornar um ponteiro para o início do vetor:

```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int * gera_vetor_rand(int n) {
	int i, vetor[n], *pi;
	srand(time(NULL));

	for(i = 0; i < n; i++)
	    vetor[i] = rand() % 100;
	pi = &vetor;

	return pi;
}
```

```
main() {
	int *v, i;
	v = gera_vetor_rand(10);
	for(i = 0; i < 10; i++) {
		printf("%d - %ld\n", *(v + i), v + i);
		// ou:
		printf("%d - %ld\n\n", v[i], &v[i]);
	}
}
```

Perceba que é possível imprimir os elementos fazendo o ponteiro andar pelo vetor ou é possível imprimir o vetor normalmente, visto que a linguagem C já trata vetores como ponteiros. A segunda forma é equivalente a primeira.

Links interessantes:
Vídeo aulas prof. André Backes sobre ponteiros: https://youtu.be/SJzd9x2S2yg