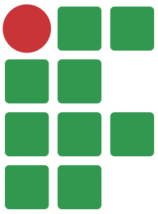


INSTITUTO FEDERAL DA PARAÍBA
CAMPUS CAMPINA GRANDE
BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO
DISCIPLINA DE ESTRUTURAS DE DADOS
PROF. VICTOR ANDRÉ PINHO DE OLIVEIRA

Estruturas de Dados

Aula 3 - Fundamentos de C - Parte 3

Introdução

Olá,

Dando continuidade ao nosso estudo sobre a linguagem C, finalizando, para ser mais preciso, hoje nós veremos dois tópicos importantíssimos. Sem eles, alguém não pode dizer que é um programador C. Me refiro a:

- Ponteiros
- Funções

Aproveitando o estudo de ponteiros, veremos como realizar a alocação dinâmica. No estudo de funções, mostrarei como empregar a recursão, técnica essencial para a implementação de diversos algoritmos e estruturas, sobretudo árvores.

Ponteiros

Já ouvi muita gente temer programar em C por causa dos benditos ponteiros. E as desculpas são as mais variadas. É verdade que o uso dos ponteiros requer um pouco mais de atenção por parte do programador, mas isso é muito longe de dizer que é “difícil”. Como programador mais experiente, posso dizer que ponteiros são a porção mais legal da linguagem. Eles expandem seus horizontes como programador e te permitem realizar algumas tarefas que seria impossível em outras linguagens.

Ponteiros: um caminho para a liberdade!

Penso que algumas coisas se tornam mais simples quando a gente começa por uma definição. Ponteiro é uma delas.

Um **ponteiro** nada mais é que uma variável que aponta para outra. Ponto. Só isso? Só isso. Podemos ser um pouco mais técnicos e dizer que um ponteiro é uma variável que armazena um endereço de uma região de memória. Não é simples? Veja o que quero dizer por meio de um diagrama:

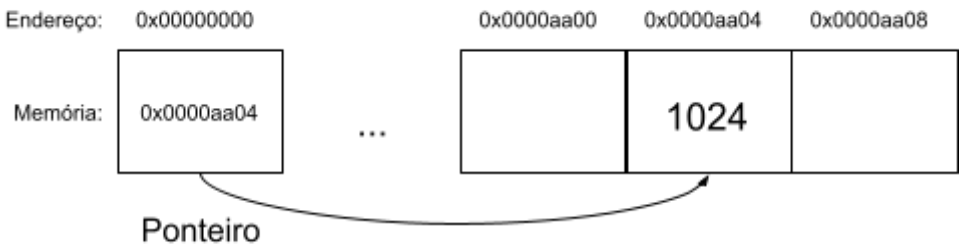


Diagrama 1 - Representação do conceito de ponteiro.

Qualquer tentativa de tornar isso mais claro vai acabar complicando. Então partamos para a prática.

Declarando ponteiros

Podemos dizer que o tipo do ponteiro pode ser determinado pela variável apontada por ele. Assim, ao declarar um ponteiro, é preciso indicar o tipo de dado e usar o símbolo * do lado esquerdo da variável que será o ponteiro.

Veja como é simples:

```
...
char *ch, ch1, *ch2;
int a, *b, c;
float af[100];
...
```

Você declara seus ponteiros como declararia qualquer outra variável. Note, pelo trecho acima, que podemos mesclar a declaração de ponteiros com variáveis comuns. Uma vez que temos que identificar o tipo nos ponteiros, nós os diferenciamos das variáveis comuns pela presença a esquerda do operador *.

No exemplo acima, declaramos 3 variáveis do tipo char, das quais ch e ch2 são ponteiros (indicadas pelo * do lado esquerdo do nome) para char e ch1 é uma variável char comum.

A mesma lógica se aplica às declarações int. Nesta linha, b é um ponteiro para int; a e b são variáveis int comuns.

O que que esse float af[100] está fazendo aqui? Isso não foi tema da aula passada, quando estudamos matrizes? Verdade. Mas quando declaramos uma matriz, estamos, na verdade, criando um ponteiro para o primeiro elemento de uma faixa de memória. Se você conseguir absorver isso, verá como C é demais. Assim, af é um array, que também é um ponteiro. Por essa razão, foi incluído no exemplo acima.

Todo esse raciocínio explorado nesse tópico é válido para declarar variáveis de qualquer tipo, inclusive seus registros (structs). Para não passar em branco, e também

porque tem um detalhe que veremos mais à frente na hora de utilizar ponteiros de registros, vou mostrar que declarar um ponteiro para registro é igualmente elementar:

```
...
typedef struct {
    char nome[100];
    unsigned tempo_servico;
} sPessoa;

sPessoa admin, *pessoas;
...
```

No trecho acima definimos um registro e declaramos duas variáveis, uma admin do tipo sPessoa, e uma pessoas, um ponteiro para sPessoa. Note que declarei o ponteiro como pessoas, no plural, demonstrando que minha intenção ao declarar esse ponteiro é que ele aponte para o primeiro elemento de vários (um array, por exemplo). E por que não declarei como array? Porque podemos realizar alocação dinâmica de um array que ainda desconheço o tamanho no momento da declaração! Ficou curioso? Já, já falaremos sobre isso.

Inicializando ponteiros

Agora veremos como podemos inicializar um ponteiro, isto é, como armazenar nele um endereço de outra variável. Aproveitando o ensejo, é importante dizer o seguinte: nunca use um ponteiro sem antes atribuir-lhe um endereço. Ponteiros são como armas, então tome sempre cuidado para onde estão apontando. Antes de utilizar um ponteiro, esteja certo de que ele foi inicializado corretamente e que está apontando para o local desejado. Isso é tão importante que é melhor fazer assim:

Atenção	Nunca use um ponteiro sem antes atribuir-lhe um endereço! Ponteiros são como armas, então tome sempre cuidado para onde estão apontando. Antes de utilizar um ponteiro, esteja certo de que ele foi inicializado corretamente e que está apontando para o local desejado.

Dados os devidos alertas, vejamos como podemos fazer as inicializações.

Podemos inicializar um ponteiro a qualquer momento após a declaração:

```
...
int a, *p_a, *p;
p_a = &a;
p = NULL;
...
```

No trecho acima, p_a está armazenando o endereço de a. Como p_a é um ponteiro, e sabemos que um ponteiro armazena um endereço, tivemos que utilizar o operador & (operador de endereço) para “extrair” o endereço de a antes de armazenar.

Note que inicializamos o ponteiro p com a constante NULL. Caso declare um ponteiro que ainda não se sabe para onde deve apontar, sempre o inicialize com a constante NULL.

Também podemos inicializar um ponteiro no momento da declaração:

```
...
int a, *p_a = &a;
...
```

Particularmente, sempre que possível, prefiro declarar um ponteiro já realizando sua inicialização. Isso ajuda a evitar esquecimentos e torna o programa mais seguro. Note que a variável a vem antes de p_a. Se fosse depois, não seria possível inicializar p_a com o endereço de a.

Também podemos inicializar um ponteiro aproveitando o endereço presente em outro ponteiro:

```
...
int a, *p_a = &a;
int *p_a2 = p_a;
...
```

Aqui estamos atribuindo p_a a p_a2. Note que não foi preciso usar o & porque p_a já contém um endereço.

E assim também:

```
...
int arr[100], *p_a = arr;
...
```

Aqui declaramos um array de nome arr e armazenamos o endereço do primeiro elemento em p_a. Exatamente, quando removemos os colchetes de um vetor estamos “extraindo” o endereço do primeiro elemento, logo não há necessidade do operador &. Uma vez que p_a é um ponteiro para o array podemos empregar os colchetes para acessar qualquer elemento do array arr.

Usamos o tipo int para as demonstrações acima, mas o mesmo se aplica a qualquer outro tipo de variável.

Uma outra forma de inicializar um ponteiro é ao passar o endereço como argumento para funções. Outra forma ainda, é inicializar um ponteiro ao realizar alocação dinâmica. Ambos os casos serão tratados mais à frente.

Usando ponteiros

Uma vez que temos nossos ponteiros inicializados e apontando para um local seguro, podemos fazer uso deles como se estivéssemos usando qualquer outra variável. Comecemos por um exemplo simples:

```c
#include <stdio.h>

int main()
{
	int a, b;
	int *p_a = &a, *p_b = &b;

	*p_a = 5;
	*p_b = 10;

	printf("%d", a + b);

	return 0;
}
```

No código acima, declaramos duas variáveis int, a e b. Em seguida, dois ponteiros int, p_a e p_b, já inicializando-os com o endereço de a e b, respectivamente. Em seguida, atribuímos 5 a *p_a. Mas o que foi isso? Calma, amigo. Vou explicar! Quando empregamos o operador * num momento posterior à declaração, estamos dizendo que queremos acessar a região para onde o ponteiro aponta. Em outras palavras, no trecho acima, *p_a seria o mesmo que simplesmente a, já que p_a está apontando para a. Logo a está recebendo 5 e b, 10. Prosseguindo, é impresso na tela o valor 15.

Ah, mas eu gostaria de ler os valores do usuário. Que tal:

```c
#include <stdio.h>

int main()
{
	int a, b;
	int *p_a = &a, *p_b = &b;

	scanf("%d %d", &a, &b);

	printf("%d", *p_a + *p_b);

	return 0;
}
```

Atenção às diferenças em relação ao exemplo anterior. Aqui estamos lendo a e b, passando o endereço das variáveis através do operador &. No printf, usamos *p_a e *p_b para imprimir a soma.

Vamos subir mais um degrau nessa escalada e realizar mais uma alteração no código acima:

```c
#include <stdio.h>

int main()
{
      int a, b;
      int *p_a = &a, *p_b = &b;

      scanf("%d %d", p_a, p_b);

      printf("%d", *p_a + *p_b);

      return 0;
}
```

Neste exemplo, ao invés de passar &a e &b para scanf, passamos p_a e p_b e observamos que o programa roda exatamente do mesmo jeito. Mas cadê o operador &? Você consegue me explicar o porquê ele não está presente? Como já foi explicado em uma aula anterior, é preciso passar o endereço da variável para que o scanf possa modificar seu conteúdo. Aqui, não foi preciso usar o operador & porque tanto p_a quanto p_b já armazenam os respectivos endereços das variáveis que queremos ler. Assim, os endereços são passados sem a necessidade do operador &. Usar o operador & seria, nesse caso, errado, pois estaríamos passando o endereço do ponteiro e não o endereço de a e de b (tema que será tratado mais a frente: ponteiro de ponteiro).

E como seria se o meu ponteiro apontasse para um array? Exatamente assim:

```c
#include <stdio.h>

int main()
{
      int arr[] = {1, 2, 4, 8, 16 ,32};
      int *p_arr = arr;

      for (int i = 0 ; i < 6 ; i++)
          printf("%d ", p_arr[i]);

      return 0;
}
```

Declaramos o ponteiro p_arr e o inicializamos com o endereço do array arr. Não foi necessário usar o operador & porque, por ser uma matriz, remover os colchetes implica em “extrair” o endereço do primeiro elemento. Depois, utilizando o ponteiro p_arr, imprimimos o conteúdo do array. Note que o ponteiro foi utilizado como se também fosse um array e, portanto, o uso dos colchetes elimina a necessidade de usar o operador*.

Ponteiros também funcionam para registros, mas devemos atentar para um detalhe adicional. Quando se tem um ponteiro para um registro, deixamos de usar o operador . (ponto) e passamos a usar o operador -> (seta). Sim, o operador seta é um menos (-) e um maior que (>). Vejamos:

```
#include <stdio.h>

typedef struct {
	char nome[100];
	unsigned tempo_servico;
} sPESSOA;

int main()
{
	sPESSOA admin = {"Victor Andre",12}, *p_admin = &admin;

	printf("Admin: %s\n", p_admin->nome);
	printf("Tempo de servico: %u", p_admin->tempo_servico);

	return 0;
}
```

Estamos inicializando p_admin com o endereço de admin. Preste bem atenção em como acessar os membros de um registro por meio de um ponteiro. Como dito, se dá por meio do operador -> (seta).

Ah, mas eu gostava tanto do operador ponto. Então tá:

```
#include <stdio.h>

typedef struct {
	char nome[100];
	unsigned tempo_servico;
} sPESSOA;

int main()
{
```

```
    sPESSOA admin = {"Victor Andre",12}, *p_admin = &admin;

    printf("Admin: %s\n",(*p_admin).nome);
    printf("Tempo de servico: %u",(*p_admin).tempo_servico);

    return 0;
}
```

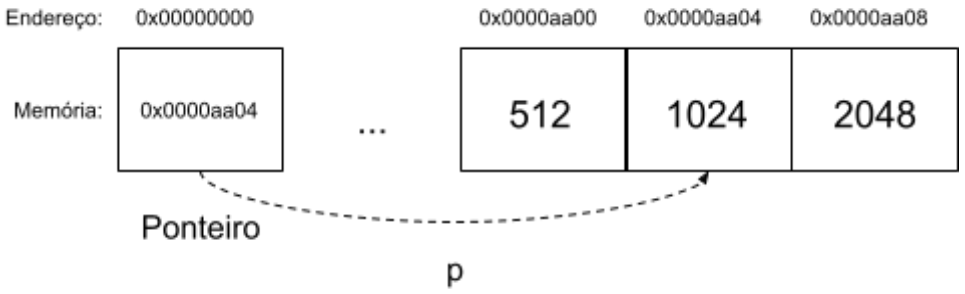
O código acima é praticamente o mesmo do anterior. A mudança se encontra apenas na maneira como acessamos os membros. Como não quisemos usar o operador ->, tivemos que, primeiro, acessar o registro por meio de (*p_admin), para depois usar o operador ponto (já que não temos mais um ponteiro), seguido do nome do membro. Os parênteses são necessários porque o operador . tem precedência maior que o operador *. Se não usássemos os parênteses, o operador * iria para o membro.

Aritmética de ponteiros

Podemos realizar apenas duas **operações aritméticas** simples com ponteiros: soma e subtração. A aritmética de ponteiros é utilizada quando se tem um ponteiro para uma posição de memória - algum elemento de um array, por exemplo - e se quer incrementar ou decrementar o ponteiro para apontar para outra posição relativa de memória - ou do array.

A grande vantagem em utilizar a aritmética de ponteiros se dá em função de o próprio compilador calcular a posição exata de memória considerando o tipo de dado. Lembre-se que diferentes tipos de dados ocupam quantidades diferentes de memória. Assim, se p é um ponteiro para int e eu faço p + 1, isto significa que p + 1 irá apontar para 4 (tamanho padrão int em bytes) endereços depois, ou seja, para o próximo elemento int. Se p é um ponteiro para char e eu faço p - 1, isto significa que p - 1 irá apontar para 1 (tamanho padrão char em bytes) endereço anterior, ou seja, para o elemento char anterior para o qual p aponta.

Observe o que quero dizer, usando o tipo int como exemplo:



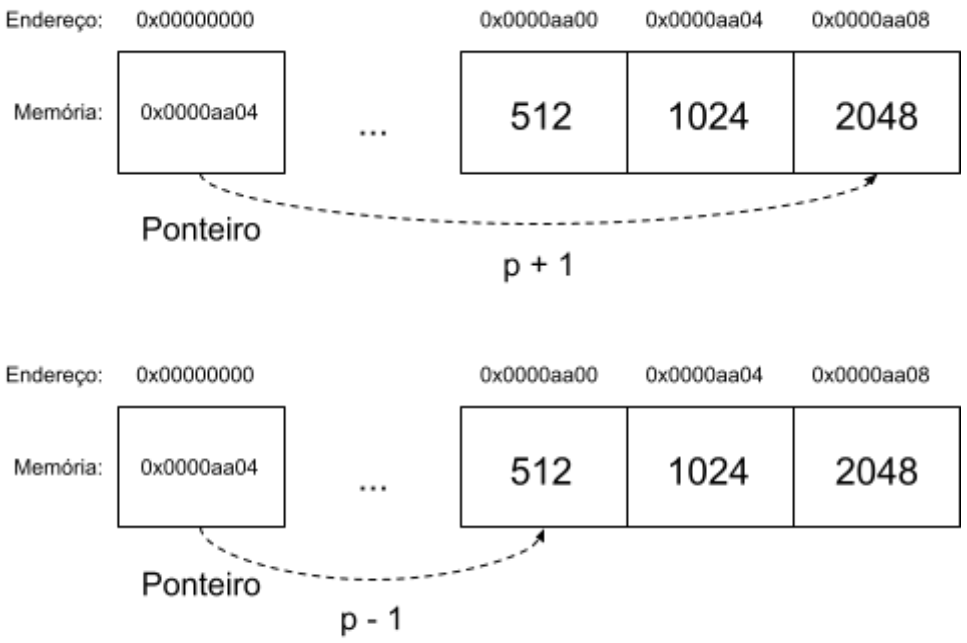


Diagrama 2 - Aritmética de ponteiros.

Note que o valor do ponteiro p não se altera, já que não atribuímos nada a p, mas o resultado da aritmética faz com que o ponteiro resultante caminhe pela memória. No diagrama acima, temos que p aponta para o endereço de 1024, p + 1 para o endereço de 2048 e p - 1 para o endereço de 512.

Vamos transformar isso em código C:

```c
#include <stdio.h>

int main()
{
	int arr[3] = {512, 1024, 2048};
	int *p = arr + 1; // p aponta para o meio do array

	printf("No endereco p %p temos %d\n", p, *p);
	printf("No endereco p + 1 %p temos %d\n", p + 1, *(p +
1));
	printf("No endereco p - 1 %p temos %d\n", p - 1, *(p -
1));

	return 0;
}
```

No código acima, declaramos o array arr e o inicializamos com 3 elementos. Declaramos um ponteiro int e o inicializamos para o elemento do meio usando

aritmética de ponteiros. Em seguida, no primeiro printf mostramos o endereço de p e o conteúdo de *p; depois, o endereço de p + 1 e o conteúdo de *(p + 1); por fim, o endereço de p - 1 e o conteúdo de *(p - 1).

Note que foi necessário fazer *(p + 1), que é diferente de *p + 1. Sabe me dizer a diferença? Sem os parênteses, *p acessa o elemento apontado por p e adiciona 1, ou seja, se p aponta para 1024, fazer *p + 1 resultaria em 1025. Com os parênteses, primeiro o ponteiro é incrementado, fazendo-o apontar para o endereço do elemento 2048, para depois acessar o elemento. Muita atenção ao usar ponteiros!

Analise o código com atenção. Não deixe de rodar na sua IDE de preferência e analisar a saída do programa.

Também é possível alterar valor do elemento apontado com aritmética de ponteiro. Veja um exemplo disso com a tradicional técnica de swap de valores:

```c
#include <stdio.h>

int main()
{
	int arr[3] = {512, 1024, 2048}, t;
	int *p = arr + 1; // p aponta para o meio do array

	t = *(p - 1);
	*(p - 1) = *(p + 1);
	*(p + 1) = t;

	printf("%d %d %d", *(p - 1), *p, *(p + 1));

	return 0;
}
```

O código acima inverte a ordem dos elementos do array simplesmente trocando o primeiro elemento com o último elemento. Mais uma vez, analise o código com atenção e não deixe de rodar na sua IDE de preferência para analisar a saída.

Nos exemplos acima, eu decidir inicializar o ponteiro p apontando para o segundo elemento do array (meio do array) apenas para ser possível demonstrar a soma e a subtração de ponteiros. Em situações normais, o ideal seria deixar p apontando para o início do array mesmo, a menos que saiba o que esteja fazendo.

Também é possível atribuir o incremento ou decremento ao ponteiro, fazendo com que ele armazene o novo endereço:

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

```c
int main()
{
	int arr[3] = {512, 1024, 2048};

	int *p = arr + 1;  // p aponta para o meio do array

	p = p - 1; // p aponta para o início do array
	printf("%d\n",*p);

	p = p + 2; // p aponta para o fim do array
	printf("%d",*p);

	return 0;
}
```

Mas tenha bastante cuidado ao fazer isso, para evitar perder o endereço que serve de ponto de referência inicial.

Aritmética de ponteiros é um recurso extremamente poderoso, assim como é uma bomba atômica. Portanto, tenha sempre muita atenção para evitar de apontar para um endereço de fora dos limites da região a qual se pretende acessar.

Alocação dinâmica

Agora que temos os conhecimentos fundamentais e necessários a respeito de ponteiros, podemos partir para a alocação dinâmica. Como o próprio nome sugere, **alocação dinâmica** consiste em alocar espaço em memória para o programa durante a execução do programa. O espaço alocado é retornado como um ponteiro para o programador. Por outro ângulo, podemos enxergar esse espaço também como um array.

Para realizar a alocação dinâmica, nós fazemos uso de pelo menos duas funções:

```c
void *malloc(unsigned tamanho);
```

e

```c
void free(void * p);
```

Existem outras funções, mas fogem do escopo da disciplina.

Acima, nós temos o que chamamos de assinatura da função. Uma **assinatura** é o cabeçalho da função, a porção que apresenta o retorno, o nome da função e os parâmetros com seus tipos.

A função malloc é usada para alocar memória. Gostaria de chamar atenção para o retorno da função malloc: void *, ou seja, um ponteiro void. Como já mencionado

anteriormente, void é um tipo de dado que significa “sem tipo” ou “sem retorno”, mas, quando usado com o *, significa ponteiro genérico, isto é, significa que é um ponteiro que pode apontar para qualquer tipo de dado. A função malloc aceita apenas um argumento, o tamanho em bytes a ser alocado.

A função free deve ser usada para liberar a memória que foi alocada previamente. Sempre que alocar memória, você deve liberar o espaço, isto é, desalocar. Isso é uma responsabilidade do programador. A função não retorna nada e recebe um ponteiro de uma região alocada previamente.

Vejamos como brincar com alocação dinâmica:

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

int main()
{
      int *arr = NULL, tam;

      printf("Me diz um tamanho de array: ");
      scanf("%d",&tam);

      arr = (int*) malloc(sizeof(int) * tam);

      for (int i = 0 ; i < tam ; i++)
          arr[i] = i;

      for (int i = 0 ; i < tam ; i++)
          printf("%d ", *(arr + i));

      free(arr);
      return 0;
}
```

Não podemos esquecer de incluir a biblioteca <stdlib.h>, pois é ela que “contém” as funções de alocação dinâmica. Declaramos o ponteiro arr e o inicializamos com NULL. O programa começa pedindo ao usuário que diga o tamanho do array a ser alocado.

Note a forma como invocamos a função malloc. Usamos (int*) antes da função para converter o retorno de void* para int*. Chamamos isso de ***cast*** (conversão de tipos). Você deverá sempre fazer o cast para o tipo da variável que está recebendo o ponteiro. A função malloc recebe a quantidade de bytes a ser alocado em memória. A maneira mais adequada de se fazer isso é por meio do operador **sizeof**, um operador de tempo de compilação que retorna o tamanho em bytes de determinado tipo de dados. Usamos sizeof(int) para obter o tamanho em bytes ocupado por um int. Use o mesmo

raciocínio para qualquer outro tipo de dado. Caso deseje alocar mais de um elemento, deve-se multiplicar pela quantidade desejada. Assim, a expressão sizeof(int) * tam retorna a quantidade em bytes de um array de tamanho tam do tipo int.

O programa segue atribuindo o valor do índice a cada elemento. Usamos a forma de array para fazer isso. Depois, foi usado aritmética de ponteiros para exibir o array. A forma de usar ponteiros fica a seu critério.

O exemplo dado acima ilustra como podemos alocar um array, mas nada impede de você alocar espaço para apenas um elemento. Vamos demonstrar isso por meio da alocação de um único registro, situação que será bastante útil para quando formos estudar listas encadeadas e árvores, por exemplo.

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

typedef struct {
      char nome[100];
      unsigned tempo_servico;
} sPessoa;

int main()
{
      sPessoa *admin = NULL;

      admin = (sPessoa*) malloc(sizeof(sPessoa));

      gets(admin->nome);
      scanf("%d",&admin->tempo_servico);

      printf("%s trabalha há %d na empresa", admin->nome,
admin->tempo_servico);

      free(admin);
      return 0;
}
```

Acima, declaramos o ponteiro para sPessoa e o inicializamos com NULL. Mesmo sabendo que logo abaixo seria atribuído um endereço a ele, é uma boa prática de programação inicializar os ponteiros no momento da declaração.

Em seguida alocamos o espaço em memória com malloc. Depois lemos os campos e os apresentamos na tela. Note que ao ler o campo tempo_servico fizemos o seguinte &admin->tempo_servico. O operador -> foi usado porque admin é um ponteiro. E o operador & foi usado para extrair o endereço do membro tempo_servico.

Como nem tudo é um mar de rosas, pode acontecer de malloc não funcionar como esperado. Com isso quero dizer que é possível que malloc não consiga alocar o espaço que estamos requisitando. Nessas situações, a função retorna a constante NULL. Portanto, é correto sempre verificar o ponteiro antes de continuar com o programa. Ajustando o código anterior, teríamos:

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

typedef struct {
      char nome[100];
      unsigned tempo_servico;
} sPessoa;

int main()
{
      sPessoa *admin;

      admin = (sPessoa*) malloc(sizeof(sPessoa));

      if (!admin){
          exit(1);
      }

      gets(admin->nome);
      scanf("%d",&admin->tempo_servico);

      printf("%s trabalha há %d na empresa", admin->nome,
admin->tempo_servico);

      free(admin);

      return 0;
}
```

Fizemos a verificação com if(!admin), isto é, se admin for NULL (0) saia do programa (exit(1)), já que fica inviável continuar.

Ponteiro de ponteiro

A linguagem C também permite que tenhamos um ponteiro para ponteiro, isto é, um ponteiro que armazena um endereço de outro ponteiro. Veja:

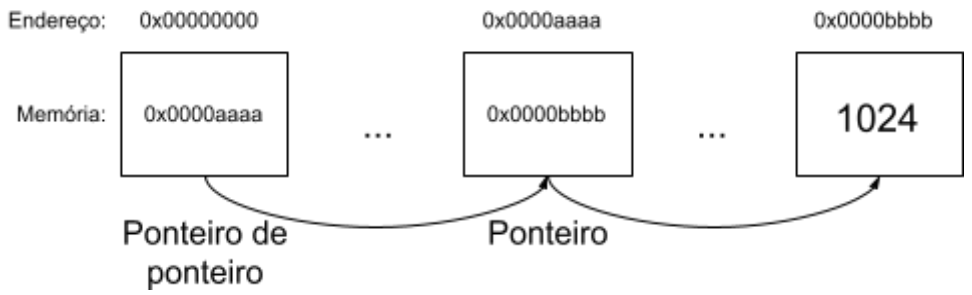


Diagrama 3 - Representação de ponteiro de ponteiro.

Muito embora pareça esquisito e até desnecessário, existem algumas situações em que isto pode ser bastante útil. Veja como podemos fazer isso em C, apenas um trecho de código:

```
...
int a, *p_a = &a, **pp_a = &p_a;
...
```

Aqui, p_a é um ponteiro para inteiro, e pp_a é um ponteiro de ponteiro para inteiro. Veja que, por ser um ponteiro de ponteiro, foi necessário fazer **.

Desse modo, ao usar os ponteiros ao longo do código, temos que: pp_a é um ponteiro para p_a; *pp_a é o mesmo que p_a; e **pp_a é o mesmo que a.

Funções

Uma **função** é um pequeno módulo que contém um trecho de código que pode ser invocado a qualquer momento e em qualquer parte do programa. Em linguagens estruturadas, como é o caso da linguagem C, programar a solução por meio de funções deixa o programa modular, organizado, mais fácil de ser depurado, com menos código repetido e, possivelmente, com menos erros. Sempre que você perceber que um determinado trecho de código pode ser reaproveitável, ou generalizável, transforme ele em uma função.

Minha primeira função

Toda definição de função deve possuir um tipo de retorno, um nome da função e zero ou mais argumentos. No corpo da função deve conter o código da função. Vejamos um exemplo:

```
#include <stdio.h>

void func(){
     printf("Ola, eu sou uma funcao.");
     return;
```

```
}

int main()
{
      func();
      return 0;
}
```

No código acima, definimos uma função func que retorna nada (void) e não recebe argumentos. A única coisa que a função faz é, ao ser invocada, imprimir `"Ola, eu sou uma funcao."`. Observe que usamos a palavra-chave return para retornar da função. Embora em funções void ela não seja necessária, é sempre bom deixar presente no código da função. Dentro da função main realizamos a invocação da função: func();.

Atenção para a localização da função func no arquivo. A função func foi definida acima da função main. Toda função deve ser definida em qualquer posição acima de sua invocação.

Retorno e parâmetros

Vamos tentar criar uma função um pouco mais útil, uma função que recebe dois argumentos int e retorna o menor entre eles.

```
#include <stdio.h>

int min(int a, int b){
      if (a < b)
          return a;
      else
          return b;
}

int main()
{
      printf("%d\n",min(4,8));
      printf("%d\n",min(min(5,9),8));
      printf("%d\n",min(min(5,9),min(4,8)));
      return 0;
}
```

Definimos uma função min que retorna um inteiro e recebe dois inteiros, a e b, como argumento. No cabeçalho da função, a lista de variáveis é chamadas de parâmetros. Os parâmetros são preenchidos com os argumentos passados no momento da invocação.

Continuando, a função min analisa quem é o menor entre a e b e o retorna para quem invocou a função.

Na função main, realizamos várias invocações à função min. Perceba como o código é reaproveitado, além de se tornar mais intuitivo e fácil de ler. No primeiro printf invocamos min passando 4 e 8 como argumentos. Na função, a assume o valor 4 e b assume 8, sempre na mesma ordem da declaração. Como a função vai retornar o tipo int, deixamos "%d" na string de controle do printf.

O segundo printf fica um pouco mais interessante, porque o primeiro argumento passado para min é o retorno de outra invocação a min. Assim, o parâmetro a será o resultado de min(5,9) e b será 8. Isso não é demais?

No terceiro printf fizemos uma chamada a min cujos argumentos são o retorno de outras chamadas a min.

Passando strings C para função

Que tal um exemplo ainda mais interessante, um que envolve parâmetros com ponteiros? Vamos criar uma função que recebe dois ponteiros para char, ou seja, duas strings C, que copia a segunda string na primeira. A função deve retornar o ponteiro para o início da primeira string. Sim a função já existe, como vimos na aula passada, mas vamos criar uma versão dela. Isto é um bom exercício.

```
#include <stdio.h>

char *copia_str(char *str1, char *str2){
     char *str = str1;
     while(*str2){
         *str1 = *str2;
         str1++;
         str2++;
     }
     *str1 = '\0';
     return str1;
}

int main()
{
     char str[50];
     copia_str(str,"C eh uma linguagem tremenda.");

     printf("%s",str);

     return 0;
```

```
}
```

Você consegue entender o funcionamento da função copia_str? Se sim, parabéns. Você entendeu ponteiros direitinho.

A função copia_str retorna um ponteiro para char e recebe dois ponteiros para char. O objetivo da função é copiar a segunda string na primeira. Para realizar este intento, usamos o while para correr pelos caracteres de str2. A expressão *str2 será verdadeira enquanto não chegar no caractere '\0' (nulo). Lembre-se que como str2 está apontando no momento para o primeiro elemento, *str2 acessa o primeiro elemento. Em seguida, copiamos o primeiro elemento de str2 no primeiro elemento de str1. Incrementamos os ponteiros, que agora passam a apontar para o segundo elemento, e assim o loop segue. Quando o loop termina, incluímos manualmente o '\0' na str1. Note que foi preciso armazenar o ponteiro str1 em str logo no início da função para não perdermos o ponteiro, pois dentro do while nós alteramos o ponteiro str1. Por isso, nós retornamos str.

Perceba que declaramos uma variável de nome str dentro da função copia_str e outra dentro da função main. Não tem problemas, pois cada variável está dentro de um escopo local, cuja "visibilidade" fica limitada apenas à função. Logo, são variáveis independentes e diferentes. Variáveis declaradas dentro de uma função são chamadas de variáveis locais. Variáveis declaradas fora do escopo das funções (no início do arquivo) são chamadas de globais.

Na função main, declaramos uma string str de tamanho 50 e invocamos a função copia_str com a ideia de copiar a string `"C eh uma linguagem tremenda"` para str. Note que não armazenamos o retorno da função copia_str. Armazenar o retorno de uma função é opcional. Nesse caso, não foi preciso "pegar" esse retorno porque a string str já vai conter a string copiada.

Não é demais? Quando juntamos ponteiros com funções, pode sair debaixo!!!

Passagem de parâmetro por valor

Antes de encerrar o tema sobre funções propriamente dito é preciso dizer que, em C, por padrão, os argumentos são passados aos parâmetros por valor. Isto quer dizer que é passado uma cópia do argumento para a função, e não o argumento propriamente dito. Isso traz algumas implicações, pois se for necessário alterar o valor de algum parâmetro, o argumento passado não sofrerá alteração. Veja:

```
#include <stdio.h>

void dobra_num(int num){
      num = num * 2;
      return;
}
```

```
int main()
{
	int n = 2;
	dobra_num(n);
	printf("%d",n);

	return 0;
}
```

O que temos aqui? A função dobra_num recebe um número e dobra esse valor. Correto? Corretíssimo. O detalhe é que isso não é refletivo para n, que foi a variável passada como argumento. Mas por quê? Porque em C, a passagem de parâmetros ocorre por valor, então uma cópia é passada.

Passando o endereço como argumento

Existe alguma maneira de "burlar" isso? Isto é, existe alguma forma de passarmos um argumento de modo que a função possa alterar esse argumento? Sim! Como? Ponteiros :). A grande sacada é passar o ponteiro (ou endereço) da variável a ser alterada. Assim, o que fizermos à variável através do ponteiro vai, obviamente, ser refletido no argumento (desde que esse argumento seja uma variável). Veja o código anterior ajustado:

```
#include <stdio.h>

void dobra_num(int *num){
	*num = *num * 2;
	return;
}

int main()
{
	int n = 2;
	dobra_num(&n);
	printf("%d",n);

	return 0;
}
```

Agora, a função dobra_num recebe como parâmetro um ponteiro para int. Na hora de invocar a função, devemos ter o cuidado de passar o endereço da variável, usando o operador &. Rode os dois códigos em sua IDE de preferência e compare os resultados.

Recursão

Agora que já sabemos como criar uma função, definir seu retorno e seus parâmetros, e também como invocá la, já temos toda a bagagem necessária para estudarmos recursividade. **Recursividade**, ou simplesmente **recursão**, é um termo usado para designar uma função que invoca a si mesma.

Observe o código a seguir.

```
#include <stdio.h>

int somatoria(int num){
      if (num == 1)
          return 1;

      return num + somatoria(num - 1);
}

int main()
{
      printf("%d",somatoria(5));
      return 0;
}
```

Encontramos a recursividade na função somatoria. No corpo da função vemos que ela invoca a si mesma. O objetivo da função é calcular num + (num - 1) + (num - 2) + ... + 1. Digamos, por exemplo, que somatoria seja invocada pela primeira vez passando 5 como argumento. Assim, na primeira invocação teremos 5 + somatoria(4). A segunda invocação (o que já caracteriza uma recursão) passa agora o valor 4 para num em uma nova instância de somatoria. Procure entender que a cada invocação é gerada uma nova instância da função. Teremos 4 + somatoria(3). Essa lógica segue até que num seja 1, condição de parada da recursão. Toda recursão deverá, obrigatoriamente, ter uma condição de parada. Assim, na medida que os return vão sendo executados, a soma vai acontecendo de trás pra frente: 1, depois 2 + 1; depois 3 + 3; depois 4 + 6; depois 5 + 10. O printf em main, então, imprimirá 15.

Procure analisar calmamente o exemplo acima.

Sempre que tivermos uma situação em que uma definição usa a si mesma, poderemos empregar recursão. Outro exemplo seria o fatorial. Me acompanhe. Quem é fat(5)? É 5 * fat(4). Quem é fat(4)? É 4 * fat(3)... e por aí vai. Isso em termos matemáticos. Como seria em C? Dá só uma olhada:

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

```c
int fat(int num){
      if (num == 0)
          return 1;

      return num * fat(num - 1);
}

int main()
{
      printf("%d",fat(5));
      return 0;
}
```

A lógica é a mesma da questão anterior. A única diferença é que ao invés da soma usamos multiplicação.

Interessante que tanto a somatória quanto o fatorial podem ser implementados sem recursão, mas você precisaria de um pouco mais de código e de mais variáveis. O único ponto negativo da recursividade é que, devido às invocações, elas rodam de maneira mais lenta se comparado à implementação não recursiva.

Para finalizar, gostaria de te lançar um desafio. Você conseguiria implementar a função copia_str (que mostrei algumas páginas atrás) de maneira recursiva? Pare tudo e tente fazer. Depois volta aqui para ver uma solução.

Já voltou? Eis uma forma de implementar a função copia_str recursiva:

```c
char *copia_str(char *str1, char *str2){
      if (!*str2){
          *str1 = '\0';
          return NULL;
      }

      *str1 = *str2;
      copia_str(str1 + 1,str2 + 1);

      return str1;
}
```

Só C que recursão é bom demais! 🙃

Bom, a aula de hoje ficou um pouco mais extensa do que o esperado. Mas foi necessário em razão da complexidade do assunto.

A partir da aula que vem, nós começaremos a implementar as estruturas de dados. Não vou prometer, mas tenho quase certeza de que as aulas serão bem mais curtas e objetivas, visto que vocês estão estudando a parte teórica na disciplina de Estrutura de Dados. Chegaremos praticamente metendo a mão na massa.

Na próxima aula iremos tratar sobre listas sequenciais.

Vê se não falta, hein!

Bons estudos e até a próxima!