Faculdade de Ciências Exatas e Tecnológicas
Câmpus Universitário Dep. Est. Renê Barbour
Bacharelado em Ciência da Computação

Ponteiros

Prof. Cleiton Anderson Profilio dos Santos

UNEMAT
Universidade do Estado de Mato Grosso
Carlos Alberto Reyes Maldonado

Ponteiros

- Um ponteiro, como diz o nome, é uma variável que aponta para o endereço de outra variável de certo tipo (ou seja, armazena o endereço de memória da outra variável).
- O compilador aloca um endereço automaticamente a toda variável declarada. Em geral, o programador não precisa se preocupar com a manipulação direta de endereços. Os ponteiros são a principal ferramenta de manipulação desses endereços, durante a execução do programa.

- São declarados pelo uso do *****

```
int * pont; // declaramos aqui um ponteiro para um número inteiro

char * pont; // declaramos aqui um ponteiro para um caractere

float * pont; // declaramos aqui um ponteiro para um número real
```

Ponteiros

- Casas e seus respectivos endereços são uma analogia interessante para a definição de ponteiros. Uma casa de determinado tipo pode ser encontrada pelo seu endereço postal; assim, “Rua da Alegria, 88”, por exemplo, identifica uma casa única, e esse endereço serve de referência para ela.
- Os ponteiros manipulam diretamente os dados contidos em endereços específicos de memória, o que lhes dá grande poder dentro dos programas escritos em Linguagem C.
- Ponteiros utilizam um especificador próprio dentro da função printf( ), que é o **%p**. Esse especificador mostra o endereço de memória do ponteiro, em formato hexadecimal.

Operadores

& = endereço de;

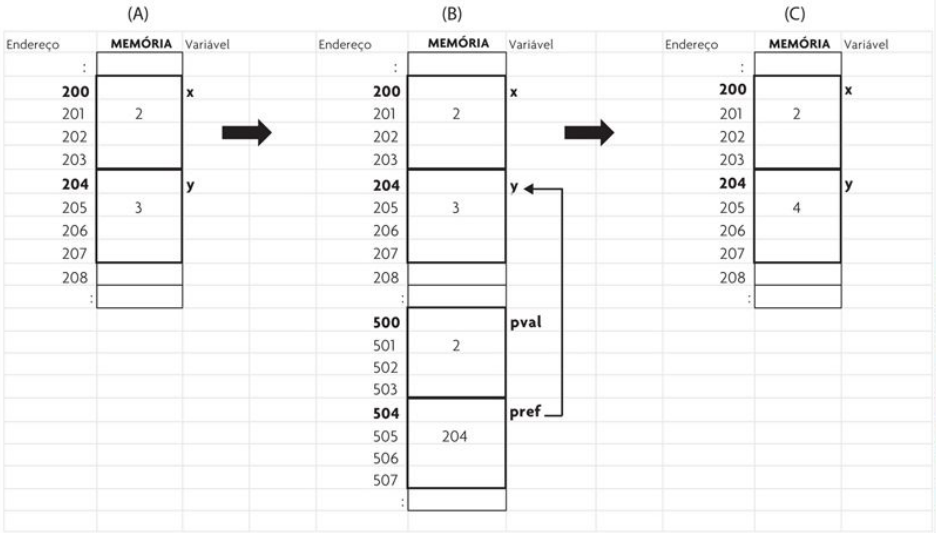
* = conteúdo de.

Exemplo

FIGURA 8.1 Exemplo de ponteiro.

Neste exemplo, o ponteiro *ptr* aponta para a variável *x*, tendo como valor o endereço de memória de *x* (1024). Já o valor da variável à qual o ponteiro faz referência é 8.

Fonte: Soffner (2013, p. 122).




```c
#include <stdio.h>
#include <stdlib.h>
void teste (int pval, int *pref){
    pval++;
    (*pref)++;
}

int main(){
    int x, y;
    int *pref;
    printf("Informe um valor x: \n");
    scanf("%d",&x);
    printf("Informe um valor y: \n");
    scanf("%d",&y);
    pref = &y;
    teste(x, pref); //teste(x, &y);
    printf("x = %d, y = %d\n", x, y);
return 0;
}
```

Fonte: Piva Jr. (2019, p. 462).

Exemplo 01

```c
#include <stdio.h>
#include <stdlib.h>
int main () {
    int x=8;
    int *p = &x;
    printf("Valor de x = %d \n", x);
    printf("Valor de x acessado pelo ponteiro = %d \n", *p);
    printf("Valor de p = %p \n", p);
    system("pause");
}
```

Acesso e alteração de valores

- Se a variável x contém o endereço da variável y, é possível acessar o valor de y a partir de x pelo acréscimo de um asterisco antes de x (***x**). Pode-se, também, alterar diretamente o valor da variável apontada, como ilustra o exemplo a seguir:

```
*ptr = 20; //alteração do valor, supondo que ptr é o endereço de dada variável
```

- Para se exibir o endereço de memória de uma variável, usamos o operador **&** (lembre-se que na função scanf( ) já utilizamos esse recurso). Assim, se considerarmos:

x = valor ou conteúdo de x

&x = endereço de memória de x

Exemplos

int * ptr; // declaramos um ponteiro para o tipo int

ptr = &x; // ptr aponta para o endereço de x

int x = 8; // atribuímos um valor a x

int * ptr = &x; // outra forma de se apontar ptr diretamente para x

int * ptr = NULL; // ptr não apontará inicialmente para nenhuma variável

*ptr // forma de se trabalhar com o valor de x diretamente

int* a, b; //a = ponteiro para int, mas b = int simples

int *a, *b; // a e b são ponteiros para int

Exemplo 02

```c
/* Uso de ponteiros */
#include <stdio.h>
#include <stdlib.h>
int main() {
    int x = 8;
    int * pont = &x;
    printf("O valor de x eh: %d \n", x);
    printf("O endereco de x eh: %p \n", &x); /* pode usar %x para formato hexadecimal */
    printf("O valor de pont eh: %p \n", pont);
    printf("O endereco de pont eh: %p \n", &pont);
    printf("O valor de x acessado por pont eh: %d \n", *pont);
    system("pause");
}
```

```c
#include <stdio.h>
#include <stdlib.h>
void troca();
int main(){
    int a, b;
    a = 5;
    b = 3;
    printf("Valores iniciais: ");
    printf("a = %d, b = %d\n",a,b);
    troca(&a, &b);
    printf("Valores trocados: ");
    printf("a = %d, b = %d\n",a,b);
    return 0;
}

void troca(int *pont_x, int *pont_y){
    int aux = *pont_x;
    *pont_x = *pont_y;
    *pont_y = aux;
}
```

*Tabela 12.4 – int aux = *x;.*

IDENTIFICADOR	ENDEREÇO	VALOR
a	10000	5 (*x)
b	9996	3
x	9992	10000 (aux=*x)
y	9988	9996
aux	9984	5

*Tabela 12.5 – *x = *y;.*

IDENTIFICADOR	ENDEREÇO	VALOR
a	10000	3 (*x=*y)
b	9996	3
x	9992	10000
y	9988	9996
aux	9984	5

*Tabela 12.6 – *y = aux;.*

IDENTIFICADOR	ENDEREÇO	VALOR
a	10000	3
b	9996	5
x	9992	10000
y	9988	9996
aux	9984	5 (*y=*aux)

Fonte: Paes (2016, p. 187-189).

Ponteiros e parâmetros de funções

- Para alterar o valor dos argumentos passados a uma função, é preciso definir os parâmetros da função como ponteiros. Na verdade, vimos isso anteriormente, na forma de **passagem de argumentos por referência**. Ela só é possível, e agora entendemos bem o porquê, devido à manipulação direta de conteúdos de variáveis, pois estamos utilizando endereços de memória, e não valores absolutos que não são visíveis fora das funções com as quais trabalhamos.

Aritmética de ponteiros

- Podemos manipular os ponteiros pelo emprego de aritmética tradicional nos seus valores e recursos; por exemplo, somar um número a um ponteiro é o mesmo que avançar dentro do trecho de memória alocado. Vejamos:

```
ptr++; // próximo endereço de memória

ptr = ptr + 2;

ptr += 4;
```

- Mas atenção: o avanço real é sempre relativo ao tamanho da variável, que pode ser determinado por **sizeof(tipo de dado)**.

Aritmética de ponteiros

Operação	Exemplo	Observação
Atribuição	ptr=&x	recebe endereço
Incremento	ptr=ptr+2	+ 2*sizeof(tipo) de ptr
Decremento	ptr=ptr-10	- 10*sizeof(tipo) de ptr
Apontado por	*ptr	acessa valor apontado
Endereço de	&ptr	manipula endereço
Diferença	ptr1 - ptr2	número de elementos entre os dois
Comparação	ptr1 > ptr2	compara dois elementos em uma estrutura pelo valor de seus endereços

Fonte: Soffner (2013, p. 125).

Ponteiros de ponteiros

- Pode ser de interesse que um ponteiro armazene, como valor, o endereço de um outro ponteiro; isso seria indicado da seguinte forma:

```
int ** ptr_ptr; // um ponteiro para outro ponteiro do tipo int
```

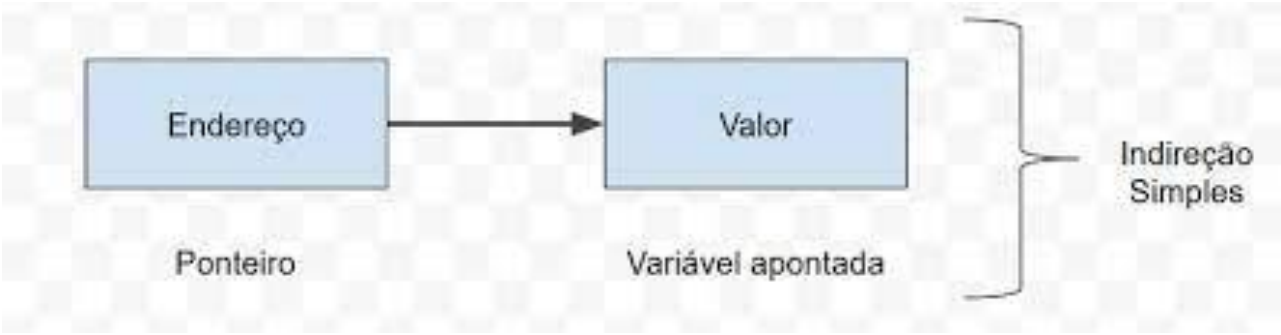
- A atribuição do ponteiro duplo se faria da seguinte maneira:

```
ptr_ptr = &ptr;
```

- Já o acesso ao valor da variável ao qual o ponteiro original faz referência, po- deria ser feito assim:

```
printf("%d \n", **ptr_ptr);
```

Indireção múltipla



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

int main(){
    setlocale(LC_ALL, "");
    int a = 40;
    int *pont1;
    int **pont2;

    pont1 = &a;
    pont2 = &pont1;

    printf("O endereço da variável a é: %p\n",&a);
    printf("O endereço do ponteiro pont1 é: %p\n",&pont1);
    printf("O endereço do ponteiro pont2 é: %p\n",&pont2);
    printf("O endereço apontado por pont1 é: %p\n",pont1);
    printf("O endereço apontado por pont2 é: %p\n",pont2);
    printf("E o valor armazenado em a é: %d\n", **pont2);
    printf("Tamanho de a: %d\n",sizeof(a));
    printf("Tamanho de pont1: %d\n",sizeof(pont1));
    printf("Tamanho de pont2: %d\n",sizeof(pont2));
    return 0;
}
```

Ponteiros e vetores

- O nome de um vetor corresponde ao endereço de seu primeiro elemento, ou seja, se V for um vetor, V == &V[0]. Podemos, portanto, usar vetores como se fossem ponteiros.
- Vejamos:

```
double * pdata; double data[10];

pdata = data; // pois data = &data[0]

pdata = &data[0];
```

- Um outro exemplo:

```
char str[10], *p;

p = str; // o ponteiro recebe o vetor, ou o endereço de seu primeiro elemento
```

Métodos de acesso a um vetor usando ponteiros

- Aritmética de ponteiros (mais rápida),

```
void putstr(char * s) {
        while(*s){
                putchar(*s++);
        }
}
```

Métodos de acesso a um vetor usando ponteiros

- Indexação do vetor (mais clássica)

char s[] = “FATEC”;

char * ptr = s; // ptr aponta para &s[0] – endereço do primeiro elemento de s

- Nesse caso, como acessaríamos o caractere ‘A’ da string?
- s[1];
- Outra forma seria *(ptr + 1);
- Ou *(s + 1) pois s = &s[0];
- Ou ptr[2];

Ponteiros de strings

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    char nome[30] = "FATEC Americana";
    char *ptr_str;
    ptr_str = nome; printf("A string eh referenciada por %c \n",*ptr_str);
    printf("A string exibida via ponteiro: \n");
    while(*ptr_str) {
        putchar(*ptr_str);
        ptr_str++;
    }
    printf("\n");
    return 0;
}
```

Manipulando estruturas com ponteiros

```c
#include <stdio.h>
#include <stdlib.h>
struct Pessoa {
    char nome[30];
    int idade;
};
alteracao(struct Pessoa *acesso){ // adiciona 20 anos à idade
    acesso -> idade += 20;
}
int main() {
    struct Pessoa acesso;
    printf("Entre nome: \n");
    scanf("%s", &acesso.nome);
    printf("Entre idade: \n");
    scanf("%d", &acesso.idade);
    printf("Dados iniciais: \n");
    printf("Nome: %s \n", acesso.nome);
    printf("Idade: %d \n", acesso.idade);
    alteracao(&acesso);
    printf("Dados após mudanças: \n");
    printf("Nome: %s \n", acesso.nome);
    printf("Idade: %d \n", acesso.idade);
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct dados {
    char nome[30];
    int idade;
} Pessoa;

alteracao(Pessoa *acesso){ // adiciona 20 anos à idade
    acesso -> idade += 20;
}
int main() {
    Pessoa p1;
    printf("Entre nome: \n");
    scanf("%s", &p1.nome);
    printf("Entre idade: \n");
    scanf("%d", &p1.idade);
    printf("Dados iniciais: \n");
    printf("Nome: %s \n", p1.nome);
    printf("Idade: %d \n", p1.idade);
    alteracao(&p1);
    printf("Dados após mudanças: \n");
    printf("Nome: %s \n", p1.nome);
    printf("Idade: %d \n", p1.idade);
    return 0;
}
```

Alocação dinâmica de memória

- Ao declararmos um vetor, alocamos memória para o armazenamento dos dados desse vetor. Se precisarmos alterar seu tamanho, teremos que modificar o código do programa, pois a alocação feita é estática.
- Por exemplo, se quisermos copiar uma string em outra, será necessário alocar espaço inicial, que com certeza será maior que o utilizado, o que gera desperdício de recursos (costumamos, por prevenção, alocar mais espaço do que o necessário).
- A saída otimizada para esse problema seria alocar a memória dinamicamente, ou seja, o programa alocaria a memória necessária durante a execução. Para isso, poderemos utilizar as funções calloc( ) e malloc( ).

calloc( )

* calloc(Quantidade_de_Elementos, tamanho)

- Exemplo:

```
int *pont2 = (int *) calloc(n, sizeof(int)); /* aloca espaço para um vetor de 'n' números inteiros */
```

- Tal função aloca quantidade de memória para um vetor com o número de elementos indicado; cada um deles tem seu tamanho expresso em bytes. A função retorna um ponteiro para o início do bloco de memória alocado, ou NULL no caso de erro ou problema.

malloc( )

void * malloc(tamanho)

- Exemplo:

```
int *pont1 = (int *) malloc(sizeof(int)); // aloca espaço para um número inteiro
```

- Essa função aloca uma quantidade de bytes e retorna um ponteiro para o início da memória alocada, ou NULL caso ocorra erro ou problema.

Liberação de memória

- A memória não mais utilizada deve ser liberada por meio da função free():

```
void free(void *ponteiro)
```

- Exemplo

```
free(pont1); // libera o espaço alocado
```

Exemplo

```c
#include <stdio.h>
#include <stdlib.h>
int main () {
    int *p;
    p=(int *) malloc(sizeof(int));
    if (!p) {
        printf ("Memoria Insuficiente! \n");
        exit;
    } else {
        printf ("Memoria alocada com sucesso! \n");
    }
    return 0;
}
```

Alteração do tamanho da memória alocada

- Um bloco de memória alocado pode ter seu tamanho alterado pelo emprego da função realloc( ).
- Sua sintaxe é:

realloc(*ponteiro, bytes)

- Ou seja, a função altera o tamanho do trecho de memória apontado por *ponteiro para a quantidade de bytes adotada.

Exemplo

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    char *str;
    str = (char *) malloc(50);
    if(str)
        printf("50 bytes alocados! \n");
    else
        printf("Erro de alocacao de memoria! \n");
    printf("\n");
    realloc(str,500);
    if(str)
        printf("500 bytes foram agora realocados! \n");
    else
        printf("Erro de alocacao de memoria \n");
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
void carregaVetor(int*);
void exibeVetor(int*);
void alteraVetor(int, int*);
int main() {
    int vetor[10];
    int posicao;
    int *p = vetor; // int *p = &vetor[0];
    carregaVetor(p);
    exibeVetor(p);
    printf("Informe uma posição do vetor entre 0 e 9: \n");
    scanf("%d", &posicao);
    alteraVetor(posicao, p);
    exibeVetor(p);
    return 0;
}
void carregaVetor(int *ponteiro){
    for (int i = 0; i <10; i++){
        *ponteiro = i + 1;
        *ponteiro++;
    }
}
void exibeVetor(int *ponteiro){
    for(int i = 0; i <10; i++){
        printf("Posição %d: %d\n",i,*ponteiro);
        *ponteiro++;
    }
}
void alteraVetor(int posicao_vetor, int *ponteiro){
    int novo_valor;
    printf("Informe um novo valor para a posição %d do vetor: \n",posicao_vetor);
    scanf("%d", &novo_valor);
    *(ponteiro+posicao_vetor) = novo_valor;
}
```

Exercícios

1. Escreva um programa que declare um inteiro, um real e um char, e ponteiros para inteiro, real, e char. Associe as variáveis aos ponteiros (use &). Modifique os valores de cada variável usando os ponteiros. Imprima os valores das variáveis antes e após a modificação, além dos seus respectivos endereços de memória.
2. Elaborar um programa que leia dois valores inteiros (A e B). Em seguida, faça uma função que retorne a soma do dobro dos dois números lidos. A função deverá armazenar o dobro de A na própria variável A e o dobro de B na própria variável B.
3. Crie um programa que carregue um vetor inteiro com 10 posições. Crie um procedimento que modifique o valor de cada posição, atribuindo o dobro a cada uma. Ao final, imprima o vetor atualizado.
4. Crie um programa que tenha uma estrutura para armazenar o nome, idade e telefone de uma pessoa. Por meio do uso de ponteiros, permita que o usuário possa alterar qualquer um dos campus informados. Ao final, exiba os dados atualizados. Sugestão: criar um menu de operações.

Entrega

- Envio dos algoritmos com a extensão **.c** no SIGAA (há uma atividade específica);
- Esta atividade não vale nota, porém vale as 4 presenças do dia 01/06;
- Envio até 05/06.

Referências

- JANDL JUNIOR, Peter. **Curso Básico da Linguagem C**. São Paulo: Novatec, 2019.
- Jr Piva; Dilermando et al. **Algoritmos e Programação de Computadores**. 2. ed. Rio de Janeiro: Elsevier, 2019. 9788595150508. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788595150508/. Acesso em: 20 mar. 2022.
- PAES, Rodrigo de Barros. **Introdução à Programação com a Linguagem C**. São Paulo: Novatec, 2016.
- SOFFNER, Renato. **Algoritmos e Programação em Linguagem C**. São Paulo: Editora Saraiva, 2013. 9788502207530. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788502207530/. Acesso em: 20 mar. 2022.