Notas de aula – AED 1 – ponteiros
Prof. Jefferson T. Oliva

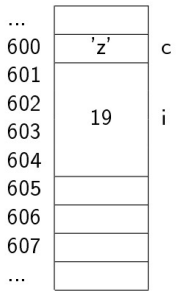
Resumo Structs

- Definição
- Vetor
- Operações
- Argumento e saída de função

Aula de hoje: Ponteiros

Variáveis armazenam dados na memória: int, float, double, char, struct

```
char c = 'z'; // 1 byte
int i = 19; // 4 bytes
```



Por exemplo, ao declararmos uma variável do tipo int, um espaço de 4 bytes será alocado na memória em tempo de compilação. Para uma struct, caso uma variável seja declarada, o espaço alocado será a soma do tamanho de todos os seus respectivos campos.

Para obtermos o tamanho de uma variável, podemos utilizar a função **sizeof**:

```
typedef struct retangulo{
    int x, y;
}Retangulo;

int main(void){
    printf("%d\n", sizeof(double));
    printf("%d\n", sizeof(int));
    printf("%d\n", sizeof(char));
    printf("%d\n", sizeof(Retangulo));
}
```

Como cada variável, ao ser declarada, é alocada na memória, então podemos obter o seu respectivo endereço. Para isso, deve ser utilizado o operador &.

```
int main(void){
	int i = 19;
	char c = 'z';
	printf("Valor de i: %d\n", i);
	printf("Endereco de i: %i\n", &i);
	printf("End. de i em hexadecimal: %x\n", &i);
	printf("Valor de c: %c\n", c);
	printf("Endereco de c: %i\n", &c);
	printf("End. de c em hexadecimal: %x\n", &c);
}
```

Na disciplina Fundamentos de Programação vocês utilizavam uma função na qual são utilizadas endereços como parâmetro: scanf, onde os argumentos básicos são o tipo e o endereço do variável.

Os endereços de memória não devem ser interpretados como números inteiros. O sinal do número depende do maior conjunto de bits (assumindo a representação de complemento de dois, que é usada pela grande maioria dos sistemas atualmente em uso), portanto um endereço de memória acima de 0x80000000 em um sistema de 32 bits será negativo e uma memória endereço abaixo 0x80000000 será positivo. Não há significado real para isso.

Hoje veremos ponteiros, que é o assunto principal dessa aula. Ponteiro é uma variável que contém um endereço de memória. Esse endereço pode ser a posição de uma outra variável na memória.

Cada ponteiro pode armazenar o endereço de seu respectivo tipo de dado. Por exemplo: um ponteiro de int pode “apontar” (atribuir o endereço da variável ao ponteiro) para a uma variável do tipo int.

Assim, se uma variável contém o endereço de outra, então a primeira aponta para a segunda.

Os ponteiros são similares aos tipos de dados que vocês viram até agora. A principal diferença é que o ponteiro possui uma declaração especial: O nome da variável deve ser precedido pelo operador *, para indicar que um endereço será armazenado:

tipo *nome;
ou
tipo* nome;
ou
tipo * nome;

Exemplos
int *pi;
char *pc;

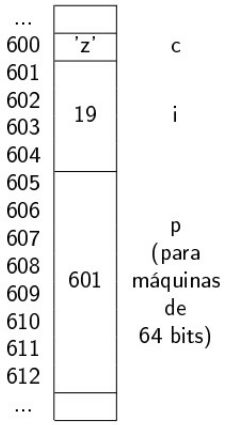
Operadores relacionados aos ponteiros:

- & ("o endereço de"): Retorna o endereço de memória do operando. Exemplo: a = &b
– * (“o valor de”): Declarar variável do tipo ponteiro e é usado para acessar o valor armazenado no endereço apontado.

Exemplos:

```
int i;
int *p = &i;
printf("%i\n", i);
printf("%x\n", &i);
printf("%x", p); // Endereço apontado
printf("%i\n", *p); // valor armazenado no endereço apontado
printf("%x\n", &p); // endereço da variável ponteiro
printf("%i\n", *i); // erro
```

```
char c = 'z';
int i = 19;
int *p = &i;
```




```
int main(void){
    int i;
    int *p_i;

    i = 30;
    p_i = &i;
    printf("%d\n", i);
    printf("%d\n", *p_i); // imprime valor no endereço
    printf("%d\n", &i); // imprime valor no endereço de i
    printf("%d\n", p_i);// imprime o endereço armazenado

    return 0;
}

int main(void){
    int *p_i
    *p_i = 30; // Não pode. Por quê? Não há endereço alocado ao p_i
     return 0;
}
```

Passagem de parâmetros

O código do slide 15 (troca1) irá compilar, mas a troca de valores não será realizada, pois as variáveis foram passadas por cópia.

```
void troca1(int a, int b){
      int aux;
      aux = a;
     a = b;
     a = aux;
}
void troca2(int *a, int *b){
     int aux;
     aux = *a;
     *a = *b;
     *a = aux;
}
int main(void){
     int x, y;

     x = 5;
     y = 10;

     troca1(x, y);

     printf("%d\n", x);
     printf("%d\n", y);
     troca2(&x, &y);
     printf("%d\n", x);
     printf("%d\n", y);

     return 0;
}
```

Explicação troca2: abrir a apresentação da aula 2 e ver do slide 15 a 26.

Aritmética de ponteiros

A adição e a subtração em ponteiros devem ser feitos cuidadosamente, pois dependem do tamanho do tipo de dados

```
char c = ‘0’;
int i = 0;
char *pc;
int *pi;
pc = &c;
pi = &i;
pc++; // desloca 1 byte
pi++; // desloca 4 bytes
```

A comparação entre ponteiros também é válida, desde que os mesmos princípios dos dados numéricos sejam seguidos

```
int main(void){
     char a, b;
    char *pa, *pb;
    pa = &a;
    pb = &b;
    if (pa == pb)
        printf("mesmo endereco.\n");
    else
        printf("Enderecos diferentes.\n");
    return 0;
}
```

Vetores: são referenciados por um nome e pelos índices, sendo o nome um ponteiro para o primeiro índice. Os índices informam a quantidade de deslocamentos a partir da posição zero. Assim, em passagem de parâmetros, o vetor não é passado por cópia. Por exemplo, imagine cinco funções que recebem um vetor como entrada. Caso um vetor ou uma matriz fosse passado por cópia, acarretaria em desperdício de recursos computacionais.

```
int main(void){
     int i;
    int v[5];
    *v = 0;
    *(v + 1) = 10;
    *(v + 2) = 20;
    *(v + 3) = 30;
    *(v + 4) = 40;
    for (i = 0; i < 5; i++)
        printf("v[%d] = %d\n", i, *(v + i));

    return 0;
}
```

Operações aritméticas do ponteiro respeitam o tamanho do tipo de dado.

Ponteiros genéricos: void*

Podem apontar para todos os tipos de dados existentes.

Se eu criar um ponteiro do tipo void, posso apontá-lo para variáveis de qualquer outro tipo.

Atenção: não confundir void* (ponteiro do tipo indefinido) com void (vazio)

```
int main(void){
     char c = ‘Q’;
    char *pc;
    void *p;
    p = &c;
    printf("Char: %c\n", c);
    printf("ponteiro: %p\n", p); // %p Ponteiro: exibe o endereço de memória do ponteiro em notação hexadecimal.
    p = &pc; // endereço do ponteiro
    printf("Char: %p\n", &pc);
    printf("ponteiro: %p\n", p);
    return 0;
}
```

Para acessar ao valor do endereço genérico, deve ser utilizado um modificador de tipo (cast)

```
int main(void){
     char c = ‘Q’;
    char *pc;
    void *p;
    p = &c;
    printf("Char: %c\n", c);
    printf("ponteiro: %c\n", *(char*) p);
    return 0;
}
```

A definição do tipo do ponteiro genérico necessita de cuidados: para acessar valores sempre deve se converter para o tipo de ponteiro utilizado (cast) e as operações aritméticas sempre utilizam 1 byte.

Vantagens do uso de ponteiros: maior liberdade na manipulação de variáveis e no uso de funções e uso de alocação dinâmica de memória.

Principal desvantagem para o uso de ponteiros: possibilidade de acessar posições não alocadas ou indevidas.

No código a seguir, o que será impresso em tela?

```
int main(void){
     int x[4];
    int *a, *b;
    *x = 118;
    *(x + 2) = 4;
    *(x + x[2] - 1) = 51;
    x[1] = 25;
    a = &x;
    b = &x[2];
    printf(" %d\n", x[0]);
    printf(" %d\n", x[1]);
    printf(" %d\n", x[2]);
    printf(" %d\n", x[3]);
    printf(" %d\n", *a);
    printf(" %d\n", *b);
    return 0;
}
```

Resposta:
118
25
4
51
118
4

Referências

Arakaki, R.; Arakaki, J.; Angerami, P. M.; Aoki, O. L.; Salles, D. S. Fundamentos de programação C: técnicas e aplicações. LTC, 1990.

Deitel, H. M.; Deitel, P. J. Como programar em C. LTC, 1999.

Pereira, S. L. Estrutura de Dados e em C: uma abordagem didática. Saraiva, 2016.

Tenenbaum, A.; Langsam, Y. Estruturas de Dados usando C. Pearson, 1995.