AED1 - Aula 09
Endereços, apontadores e registros

"Os conceitos de endereço e apontador são fundamentais em qualquer linguagem de programação, embora fiquem ocultos em algumas linguagens. Em C, esses conceitos são explícitos. Dominar o conceito de apontador exige algum esforço e uma boa dose de prática" - citação retirada do livro do Paulo Feofiloff.

“Para projetar algoritmos eficientes é importante manipular dados com eficiência. Estruturas de dados são maneiras de organizar dados para atender requisições com rapidez, mas cada estrutura de dados é eficiente em algumas situações e ineficiente em outras. Uma ótima maneira de entender os prós e contras de uma estrutura de dados é implementá-la. Para isso é necessário dominar conceitos como endereços, apontadores e registros, que nos permitem construir estruturas complexas a partir de tipos básicos de dados.”

Endereços

A memória de um computador é uma sequência de bytes

- os quais são numerados sequencialmente
- numa analogia, podemos pensar na memória como um imenso vetor
    - assim, o número (ou índice) de um byte é seu endereço

Cada objeto na memória ocupa um número de bytes consecutivos

- código

```
printf("sizeof(char) = %d\n", sizeof(char));
printf("sizeof(int) = %d\n", sizeof(int));
printf("sizeof(long) = %d\n", sizeof(long));
printf("sizeof(float) = %d\n", sizeof(float));
printf("sizeof(double) = %d\n", sizeof(double));
printf("sizeof(char *) = %d\n", sizeof(char *));
printf("sizeof(int *) = %d\n", sizeof(int *));
printf("sizeof(double *) = %d\n", sizeof(double *));
```

- exemplo
    - sizeof(char) = 1
    - sizeof(int) = 4
    - sizeof(long) = 4
    - sizeof(float) = 4
    - sizeof(double) = 8
    - sizeof(char *) = 4
    - sizeof(int *) = 4
    - sizeof(double *) = 4
- note que usamos 4 bytes (32 bits) para endereçar uma posição da memória

- sabendo que cada byte tem seu próprio endereço,
- qual o total de bytes que podemos endereçar com 32 bits?
  - 2^32 ~= 4GB
  - mas isso é menos do que a memória de muitos computadores modernos. Como fazemos para usar toda a memória?

O endereço de uma variável corresponde ao endereço do seu primeiro byte

- exemplo, após declarar

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

- os endereços poderiam ser algo como:
  - endereço de c = 24515
  - endereço de i = 24516
  - endereço de ponto = 24520
  - endereço de ponto.x = 24520
  - endereço de ponto.y = 24524
  - endereço de v[0] = 24528
  - endereço de v[1] = 24532
  - endereço de v[2] = 24536
  - endereço de v[3] = 24540

Usamos o operador & para obter o endereço de uma variável

- exemplo
  - &i devolve o endereço de i
- código

```
printf("endereco de c = %d\n", &c);
printf("endereco de i = %d\n", &i);
printf("endereco de ponto = %d\n", &ponto);
printf("endereco de ponto.x = %d\n", &ponto.x);
printf("endereco de ponto.y = %d\n", &ponto.y);
printf("endereco de v[0] = %d\n", &v[0]);
printf("endereco de v[1] = %d\n", &v[1]);
printf("endereco de v[2] = %d\n", &v[2]);
printf("endereco de v[3] = %d\n", &v[3]);
```

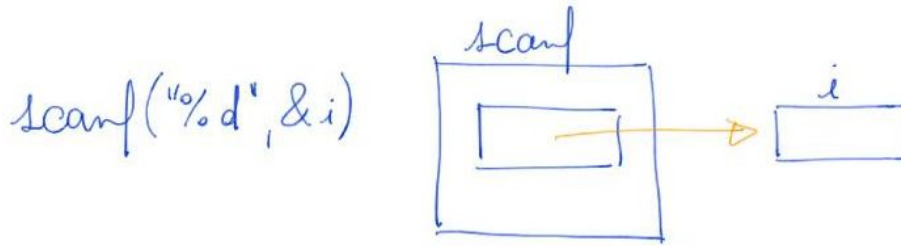
O lugar mais comum de encontrar/usar o operador & é na função scanf

- código

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

```
printf("endereco de i = %p\n", &i);
printf("conteudo de i = %d\n", i);
```

- por que precisamos usar &i?



Apontadores

São variáveis que armazenam endereços.

Declaramos um apontador p de um tipo de variável var colocando

- um * entre o nome do tipo e o nome da variável,
    - i.e., var * p;
- usamos o valor especial NULL para inidicar que
    - um apontador não endereça qualquer variável.
- exemplos

```
char *p1;
int *p2, i;

p1 = NULL;
p2 = &i;
```

Se um apontador p armazena o endereço (&i) de uma variável i, dizemos que

- p aponta para i
- p é o endereço de i

- *p é o objeto apontado por p
    - no exemplo da figura anterior, *p = i
- em geral, se p = &i então *p é igual a i
- exemplo
    - maneira complicada (que brinca com apontadores) de fazer c = a + b

```
int a = 2;
int b = 3;
int c;
int *p;
int *q;
```

```
p = &a;
q = &b;
c = *p + *q;
```

Uso de apontadores como parâmetros na função troca

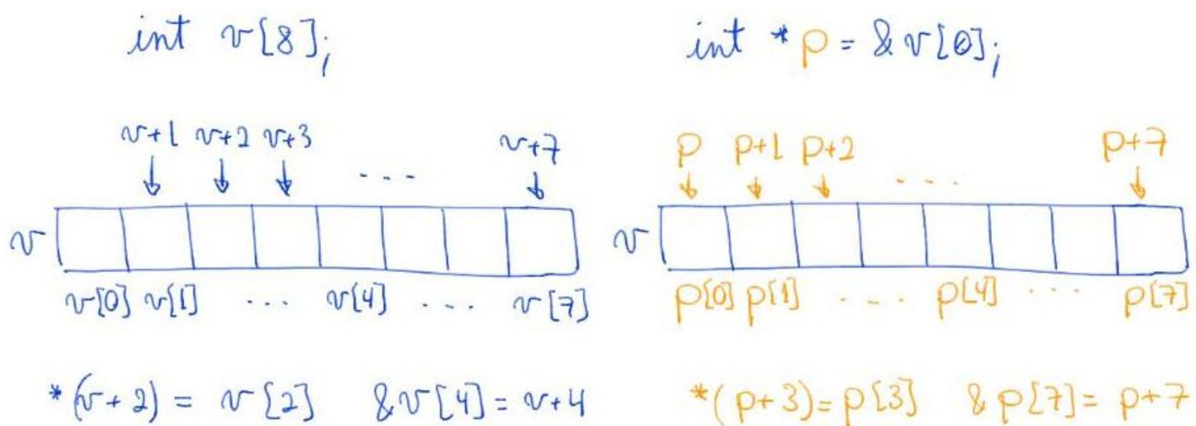
```
void trocaErrada(int i, int j)
{
   int aux;
   aux = i;
   i = j;
   j = aux;
}
void trocaCerta(int *i, int *j)
{
   int aux;
   aux = *i;
   *i = *j;
   *j = aux;
}
```

exemplo de uso

```
a = 1;
b = 2;
printf("a = %d, b = %d\n", a, b);
trocaErrada(a, b);
printf("a = %d, b = %d\n", a, b);
trocaCerta(&a, &b);
printf("a = %d, b = %d\n", a, b);
```

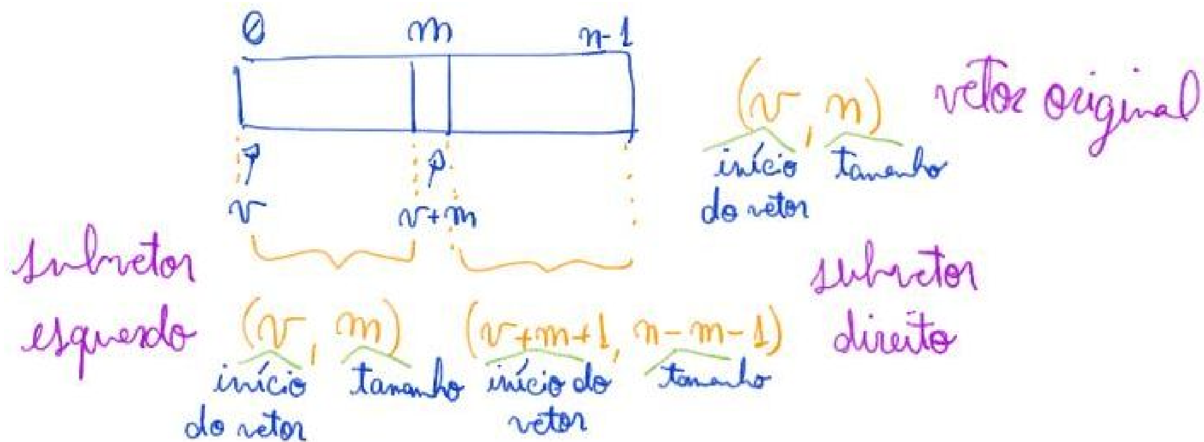
Vetores e aritmética de apontadores:

- em C existe uma relação muito forte entre apontadores e vetores
- o nome de um vetor é sinônimo do endereço da posição inicial do vetor
    - assim, &v[0] é igual a v
    - e, de modo mais geral, &v[i] é igual a v+i
        - ou, v[i] é igual a *(v+i)



    - ponteiros (p) e nomes de vetores (v) são equivalentes em quase tudo

        - mas nomes de vetores alocados estaticamente são imutáveis
            - por isso operações como v++ ou v = v + 3 são ilegais
- na figura a seguir remetemos à busca binária e mostramos como definir subvetores usando aritmética de ponteiros



Como no exemplo das funções trocaCerta e scanf, é comum usarmos apontadores como parâmetros de função para:

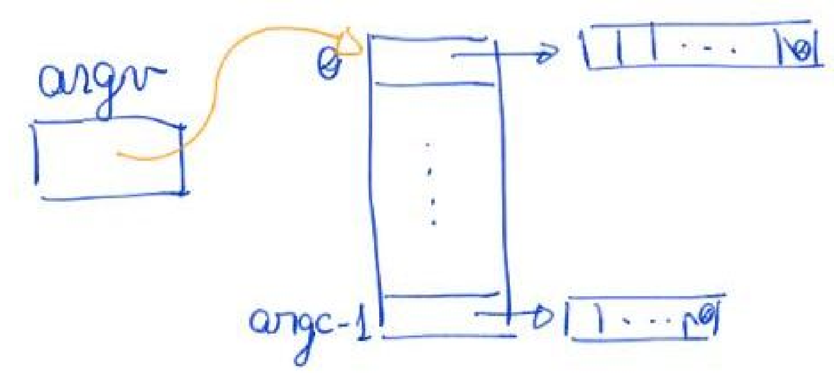
- devolver resultados
    - note que muitas vezes usamos o return para isso
        - mas ele só permite devolver um valor de um tipo simples

Outro uso importante de apontadores como parâmetros de função

- é para passar vetores
    - sem copiar seus conteúdos para a área de memória da função
    - já que isso seria muito ineficiente
- um exemplo de vetores passados como parâmetro são os argumentos da função main
    - argc
        - c vem de count
        - trata-se do número de argumentos na linha de comando
    - argv[ ] é um vetor de strings
        - uma string é um vetor de caracteres terminado em ‘\0’
        - cada string contém um dos argumentos da linha de comando
            - argv[0] é o próprio nome do programa chamado

```
int main(int argc, char *argv[])
{
   int n;
   if (argc != 2)
   {
       printf("Parametros incorretos. Ex.: entrada 15\n");
       return 0;
   }
```

```
    n = atoi(argv[1]);
    printf("%d\n", n);
    return 0;
}
```



Registros

Um registro (struct) é uma coleção de variáveis,

- possivelmente de tipos diferentes.
- exemplo

```
struct
{
    int dia;
    int mes;
    int ano;
} aniversario;
```

Usamos o operador . para acessar um campo de uma variável que é um registro

- exemplo

```
aniversario.dia = 20;
aniversario.mes = 10;
aniversario.ano = 2010;
```

Normalmente damos um nome para os registros que declaramos

- assim fica fácil declarar diversas variáveis daquele tipo
- exemplo

```
struct data
{
    int dia;
    int mes;
    int ano;
};

    struct data aniversario;
    struct data casamento;
```

Note que a declaração “struct nomeEscolhido” define um tipo

- para evitar repetir essa expressão em toda declaração de variável
- usamos typedef para definir uma abreviatura
- exemplo

```
typedef struct data Data;

struct data
{
   int dia;
   int mes;
   int ano;
};
```

ou, numa forma equivalente

```
typedef struct data
{
   int dia;
   int mes;
   int ano;
} Data;
```

Registros e apontadores:

- quando um apontador endereça uma variável que é um registro podemos acessar seus campos de duas formas equivalentes
    - se Data * p = &aniversario temos
        - (*p).mes igual a p->mes igual a aniversario.mes

```
Data *p;
p = &aniversario;
(*p).dia = 10;
p->dia = 10;
```