ESTRUTURAS DE DADOS

Vetores

Roteiro

- **Noções Básicas**
- **Alocação Dinâmica**
- **Passagem de Parâmetro**

Noções Básicas

- **Vetores são a maneira mais simples de estruturarmos um conjunto de dados.**
- **Os elementos devem ser do mesmo tipo.**
- **O tamanho é fixado na declaração do vetor.**
- **Elementos ocupam regiões consecutivas de memória**

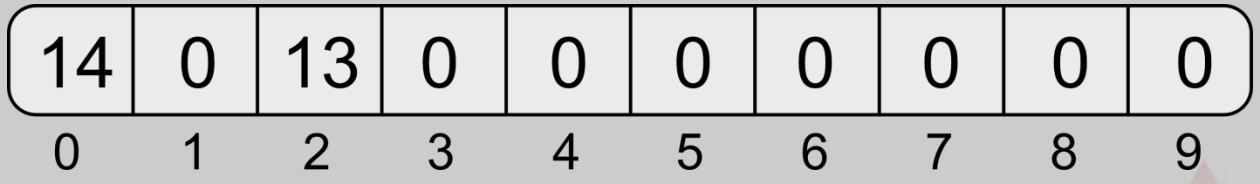
- **Na declaração, informamos o tipo e o número de elementos.**

```
int c[10];
```

- **Declarando e inicializando os elementos**

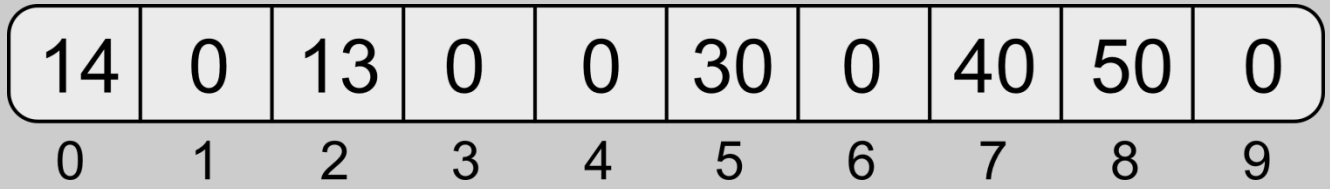
```
int c[10] = {14, 0, 13};
```

  - **Inicializamos os primeiros três elementos.**
  - **Como não fornecemos valores para todos os elementos, o restante iniciará com zero.**



- **Os elementos podem ser acessados com a sintaxe de colchetes.**

```
c[5] = 30;
c[7] = 40;
c[8] = 50;
```



- **É comum iterarmos pelos valores com for.**

```
for (int i = 0; i < 10; i++) {
  std::cout << "c[" << i << "] = " << c[i] << "\n";
}
```

- **É tarefa do programador verificar os limites do vetor antes de fazer o acesso.**

- **A construção a seguir é comum:**

```
int c[] = {14, 0, 13};
```

  - **O tamanho do vetor será o tamanho da lista.**
  - **Não adotaremos esta notação na disciplina por ser difícil saber a quantidade de elementos.**

- **Em alguns casos, inicializaremos os elementos com um laço de repetição:**

```
int c[10];
for (int i = 0; i < 10; i++) {
  c[i] = 2*i;
}
```

- **Especificaremos o tamanho do vetor com uma variável constante:**

```
const int NUM_ELEM = 10;

int main() {
  int c[NUM_ELEM];
  for (int i = 0; i < NUM_ELEM; i++) {
    c[i] = 2*i;
  }
  for (int i = 0; i < NUM_ELEM; i++) {
    std::cout << "c[" << i << "] = " << c[i] << "\n";
  }
  return 0;
}
```

- **Confere mais clareza ao código e torna mais escalonável.**

Alocação Dinâmica

- **Os vetores declarados até aqui eram estáticos, pois o número de elementos era fixado em tempo de compilação.**

- **Podemos declarar vetores em que o número de elementos é conhecido apenas durante a execução com alocação dinâmica.**

- **O tamanho do vetor não poderá mudar após a declaração.**

- **Para fazer a alocação dinâmica, usaremos o operador new que já conhecemos:**

```
int* c = new int[num_elem];
```

  - **O comando alocará uma região de memória de tamanho suficiente para alocar num_elem elementos inteiros contíguos.**
  - **Em outras palavras, o comando cria num_elem elementos inteiros consecutivos (um vetor).**
  - **A variável c recebe o endereço do primeiro elemento do vetor. Feito isso, podemos usar a sintaxe de colchetes.**

```
c[5] = 30;
c[7] = 40;
c[8] = 50;
```

- **O tamanho do vetor pode mudar em diferentes execuções:**

```
int main() {
  int num_elem;
  std::cout << "Digite o tamanho do vetor: ";
  std::cin >> num_elem;

  int* c = new int[num_elem];

  for (int i = 0; i < num_elem; i++) {
    c[i] = 2*i;
  }
  for (int i = 0; i < num_elem; i++) {
    std::cout << "c[" << i << "] = " << c[i] << "\n";
  }
  return 0;
}
```

- **Como a alocação foi feita de forma dinâmica com new, precisamos desalocar a memória com o comando delete.**

```
int* c = new int[num_elem];

delete [] c;
```

Passagem de Parâmetro

- **No comando a seguir:**

```
int c[10];
```

  - **A variável c armazena o endereço base do vetor.**
  - **Nesse caso, c é um ponteiro.**
- **O mesmo ocorre na alocação dinâmica:**

```
int* c = new int[num_elem];
```

- **Quando passamos um vetor para uma função, estamos passando um ponteiro.**

- **Vamos discutir a passagem de ponteiros:**

```
void valor_alocando_memoria(int* p)
{
  p = new int;
  *p = 7;
}
void valor_modificando_memoria(int* p)
{
  *p = 8;
}
void referencia(int*& p)
{
  p = new int;
  *p = 9;
}
```

- **Iremos invocar os métodos:**

```
int a  = 1;
int b  = 2;
int c  = 3;
int* p1 = &a;
int* p2 = &b;
int* p3 = &c;

cout <<"Antes: "<<  p1 <<" "<<  p2 <<" "<<  p3 << endl;
cout <<"Antes: "<< *p1 <<" "<< *p2 <<" "<< *p3 << endl;
cout << endl;

valor_alocando_memoria(p1);
valor_modificando_memoria(p2);
referencia(p3);
```

- **Após a invocação, imprimimos na tela os valores das variáveis novamente.**

```
valor_alocando_memoria(p1);
valor_modificando_memoria(p2);
referencia(p3);

cout << "Depois: "<<  p1 <<" "<<  p2 <<" "<<  p3 << endl;
cout << "Depois: "<< *p1 <<" "<< *p2 <<" "<< *p3 << endl;
cout << "Depois: "<< a <<"  "<< b <<"  "<< c << endl;
```

- **Como resultado, temos:**

```
Antes: 0x7ffee254f958 0x7ffee254f954 0x7ffee254f950
Antes: 1 2 3

Depois: 0x7ffee254f958 0x7ffee254f954 0x7fb82b4058a0
Depois: 1 8 9
Depois: 1 8 3
```

- **Como conclusão do experimento:**

```
void valor_alocando_memoria(int* p)
{
  p = new int;
  *p = 7;
}
```

  - **A função valor_alocando_memoria não altera o valor da variável que estava fora da função.**
  - **A função também não modifica o endereço para o qual o ponteiro de fora da função estava apontando.**
  - **Isso pode ser explicado porque o parâmetro foi passado por valor. Nada do que foi feito internamente afeta as variáveis de fora.**

- **Como conclusão do experimento:**

```
void valor_modificando_memoria(int* p)
{
   *p = 8;
}
```

  - **A função valor_modificando_memoria altera o valor da variável que estava fora da função.**
  - **A passagem de parâmetro foi por valor, mas estamos modificando o endereço de memória para onde o ponteiro aponta.**
  - **A variável b usava essa região de memória e a variável p2 apontava para ela. Nesse caso, ambas foram afetadas.**

- **Como conclusão do experimento:**

```
void referencia(int*& p)
{
  p = new int;
  *p = 9;
}
```

  - **A função referencia altera o endereço para onde aponta a variável p3, pois essa foi passada como referência.**
  - **A variável c não é afetada, pois continua utilizando o endereço antigo de memória.**
  - **Nesse caso, o valor apontado por p3 se torna diferente do valor da variável c.**

- **Como as variáveis usadas nos vetores são ponteiros, a função sabe o endereço onde os elementos estão armazenados.**

- **Nesse caso, as modificações dentro da função naquele endereço de memória surtirão efeito também fora da função.**

- **Em outras palavras, quem invoca a função passando um vetor concede acesso direto aos dados e a permissão de modificá-los.**

- **Não dar essa permissão implicaria em copiar o vetor para outro lugar na memória.**

As duas sintaxes a seguir são válidas:

```
void modifica_vetor_sintaxe_1(int b[], int num_elem)
{
  for (int i = 0; i < num_elem; i++){
    b[i] = b[i] * 2;
  }
}

void modifica_vetor_sintaxe_2(int* b, int num_elem)
{
  for (int i = 0; i < num_elem; i++){
    b[i] = b[i] * 2;
  }
}
```

O efeito é o mesmo com vetores estáticos e vetores alocados dinamicamente.

```
// Alocação Estática
int c[NUM_ELEM] = {1,2,3,4,5,6,7,8,9,10};

// Alocação Dinâmica
int *d = new int[NUM_ELEM];
for (int i = 0; i < NUM_ELEM; i++){
  d[i] = i + 1;
}

modifica_vetor_sintaxe_1(c, NUM_ELEM);
modifica_vetor_sintaxe_2(c, NUM_ELEM);

modifica_vetor_sintaxe_1(d, NUM_ELEM);
modifica_vetor_sintaxe_2(d, NUM_ELEM);
```

O efeito é o mesmo com vetores estáticos e vetores alocados dinamicamente.

```
for (int i = 0; i < NUM_ELEM; i++){
  std::cout << i << " : " << c[i] << " , " << d[i] << std::endl;
}
```

```
0 : 4 , 4
1 : 8 , 8
2 : 12 , 12
3 : 16 , 16
4 : 20 , 20
5 : 24 , 24
6 : 28 , 28
7 : 32 , 32
8 : 36 , 36
9 : 40 , 40
```

O const pode ser usado novamente.

```
void vetor_const_sintaxe_1(const int* b, int num_elem)
{
  // Corpo sem alterar os elementos do vetor
}

void vetor_const_sintaxe_2(const int b[], int num_elem)
{
  // Corpo sem alterar os elementos do vetor
}
```

- Impede que uma função altere o vetor.
- Um erro em tempo de compilação será fornecido se alguma atribuição for feita.

```
error: read-only variable is not assignable
```

ESTRUTURAS DE DADOS

Vetores