

Tipos Compostos de Dados

VETORES

Introdução

- Programas são compostos por:
 - **Instruções**
 - Entrada, saída, atribuição, etc.
 - Ex.: `cin >> val; cout << total; peso = 0;`
 - **Expressões**
 - Aritméticas, binárias, etc.
 - Ex.: `10 * a + b; mascara & estado;`
 - **Dados**
 - Variáveis e constantes
 - Ex.: `total, num, "Digite valor:", 30, 4.52`, etc.

Introdução

- As **variáveis** e **constantes** armazenam informações
 - Elas ocupam espaço na memória
 - Possuem um tipo
- Os **tipos básicos** armazenam valores:

Inteiros	{	char	ch	=	'W';
		short	sol	=	25;
		int	num	=	45820;
Pontos flutuantes	{	float	taxa	=	0.25f;
		double	peso	=	1.729156E5;

Introdução

- Porém, com os tipos básicos não é possível armazenar um **conjunto de informações**
 - Como armazenar as notas de 30 alunos?

```
float n1 = 8.0;  
float n2 = 7.0;  
float n3 = 4.5;  
...  
float n29 = 5.0;  
float n30 = 2.0;
```

Criar 30 variáveis diferentes não é a melhor solução.

- Como armazenar um cadastro completo de 30 alunos? (nome, identidade, CPF, endereço, etc.)

Introdução

- É preciso utilizar tipos de dados compostos, tipos que armazenam múltiplos valores:
 - Vetores
 - Strings
 - Registros
 - Uniões
 - Enumerações
- Os tipos compostos são coleções formadas a partir dos tipos de dados básicos

Vetores

- Um **vetor** armazena múltiplos valores, todos do mesmo tipo:

- As notas de 30 alunos
30 valores tipo float

9.5	8.0	5.0	3.2	7.3	4.0	2.1	...	0.4	8.0
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

- As 100 primeiras teclas pressionadas no teclado
100 valores tipo char
- O número diário de visitas de um site web por um período de um ano
365 valores tipo unsigned int

Declaração de Vetores

- Para criar um vetor utiliza-se uma **instrução de declaração**
 - A **declaração de um vetor** deve conter :

O tipo de cada elemento

A quantidade de elementos

```
int visitas[365];
```

O nome do vetor

Declaração de Vetores

- O tamanho do vetor deve ser um **valor inteiro constante**:

- Uma constante inteira

```
float notas[30];
```

```
const int Max = 30;  
float notas[Max];
```

- Uma expressão com valor inteiro constante[†]

```
int num[5 * sizeof(int)];
```

```
const int Tam = 5 * sizeof(int);  
int num[Tam];
```

[†] Calculado durante o processo de compilação

Declaração de Vetores

- O **constexpr** ajuda a eliminar dúvidas:
 - Declara uma constante que pode ser inicializada apenas para valores **conhecidos em tempo de compilação**

```
// Qtd é um valor constante  
const int Qtd = 30;
```

```
// e conhecido durante a compilação  
constexpr int Qtd = 30; ✓
```

```
// Tam é um valor constante  
const int Tam = 5 * sizeof(int);
```

```
// e conhecido durante a compilação  
constexpr int Tam = 5 * sizeof(int); ✓
```

```
// Max é um valor constante  
const int Max = rand();
```

```
// mas só é conhecido na execução  
constexpr int Max = rand(); ✗
```

Declaração de Vetores

- A quantidade de elementos deve ser conhecida no momento da **compilação** e portanto **não pode ser uma variável**

```
int tam = 30;  
int notas[tam]; ❌ // inválido, tam não é constante
```

```
int quant;  
cin >> quant;  
int notas[quant]; ❌ // inválido, quant não é constante
```

Declaração de Vetores

```
#include <iostream>
using namespace std;

int main()
{
    const int Tam = 5 * sizeof(int);           // valor constexpr
    const int Max = rand();                    // valor const mas não constexpr

    cout << "Tam:" << Tam << endl;            // 20
    cout << "Max:" << Max << endl;            // número aleatório

    int val[Tam];                               // Tam é constexpr ✓
    int vet[Max];                               // Max não é conhecido durante a compilação ✗

    return 0;
}
```

Vetores

- Um **vetor** é um **tipo de dado derivado**
 - Ele é formado por múltiplos valores
 - Ele depende de **um tipo base**
- Um **vetor não é um tipo**
 - Ele é um conjunto de dados do mesmo tipo
 - Não existe um tipo chamado "vetor"
 - Todo vetor tem um tipo
Ex.: vetor de **int**, vetor de **char**, vetor de **double**, etc.

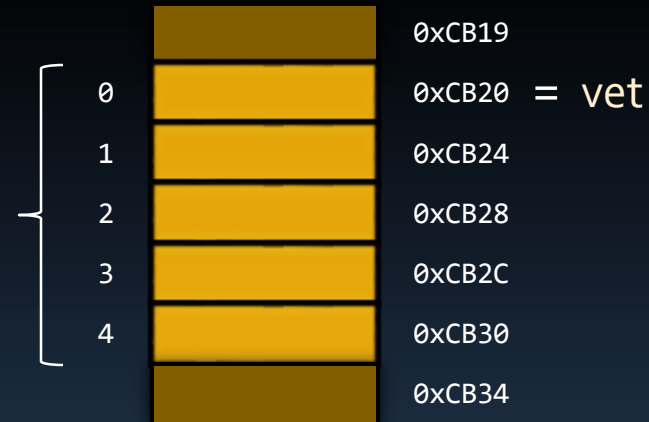
Vetores

- Os elementos são armazenados em **posições consecutivas**
- Cada elemento é representado por um **índice**
- Em C++ o **índice começa em 0** e não em 1

```
int main()
{
    int vet[5];

    ...
}
```

5 valores
inteiros



Vetores

- Os elementos são **acessados individualmente pelo seu índice**

```
int visitas[365];
```

```
visitas[0] = 38;
```

```
visitas[1] = 54;
```

```
visitas[2] = 88;
```

```
cout << visitas[1];           // 54
```

```
visitas[3] = visitas[0] + 2;  // 40
```

0	1	2	3	4	5	7	8	9	10	11		363	364
38	54	88	40								...		

Vetores

```
#include <iostream>
using namespace std;
int main()
{
    int batatas[3]; // cria vetor de 3 elementos
    batatas[0] = 7; // atribui valor ao 1º elemento
    batatas[1] = 8; // atribui valor ao 2º elemento
    batatas[2] = 6; // atribui valor ao 3º elemento

    int custo[3] = {20, 30, 5}; // cria e inicializa vetor

    cout << "Quantidade de batatas = ";
    cout << batatas[0] + batatas[1] + batatas[2] << endl;
    cout << "0 pacote com " << batatas[1] << " batatas custa ";
    cout << custo[1] << " centavos por batata.\n";

    int total = batatas[1] * custo[1];
    cout << "0 segundo pacote custa " << total << " centavos.\n";
}
```

Vetores

- A saída do programa é:

```
Quantidade de batatas = 21  
O pacote com 8 batatas custa 30 centavos por batata.  
O segundo pacote custa 240 centavos.
```

- Um vetor **não inicializado** contém valores indefinidos

```
int batatas[3]; // os valores armazenados não são iguais a 0  
                // eles são valores indefinidos até que  
                // seja feita uma atribuição de valor
```


Vetores

```
#include <iostream>
using namespace std;
int main()
{
    int vet[3]; // cria vetor de 3 elementos

    cout << "Conteúdo da posição 0: " << vet[0] << endl;
    cout << "Conteúdo da posição 1: " << vet[1] << endl;
    cout << "Conteúdo da posição 2: " << vet[2] << endl << endl;

    vet[0] = 0; vet[1] = 0; vet[2] = 0;

    cout << "Conteúdo da posição 0: " << vet[0] << endl;
    cout << "Conteúdo da posição 1: " << vet[1] << endl;
    cout << "Conteúdo da posição 2: " << vet[2] << endl;

    cout << "\nO vetor tem " << sizeof vet << " bytes.\n";
    cout << "Um elemento tem " << sizeof vet[0] << " bytes.\n";
}
```

Vetores

- A saída do programa:

Conteúdo da posição 0 = -858993460

Conteúdo da posição 1 = -858993460

Conteúdo da posição 2 = -858993460

Conteúdo da posição 0 = 0

Conteúdo da posição 1 = 0

Conteúdo da posição 2 = 0

O vetor tem 12 bytes.

Um elemento tem 4 bytes.

Inicialização de Vetores

- A **inicialização com o uso das chaves** só funciona na declaração do vetor

```
int cartas[4] = {3, 6, 8, 10};  ✓  
int mao[4];                    ✓  
mao = {5, 6, 7, 8};            ✗
```

- Após a declaração do vetor seus valores só podem ser alterados com **atribuição individual a cada elemento**

```
int mao[4];    // ok  
mao[0] = 5;    // atribuição de valor  
mao[1] = 6;    // atribuição de valor  
mao[2] = 7;    // atribuição de valor  
mao[3] = 8;    // atribuição de valor
```

Inicialização de Vetores

- Um vetor **não pode** ser atribuído a outro

```
int cartas[4] = {30, 60, 80, 100}; // ok
int mao[4]; // ok
mao = cartas; // inválido, alterando endereço
```

0	30	0xCB20 = cartas
1	60	0xCB24
2	80	0xCB28
3	100	0xCB2C
0		0xCB30 = mao
1		0xCB34
2		0xCB38
3		0xCB3C

O nome de um vetor representa o endereço inicial do conjunto de dados.
Esse endereço é fixado na criação do vetor e **não pode ser alterado**

Inicialização de Vetores

- Ao inicializar um vetor é permitido **fornecer menos valores** que o tamanho do vetor

```
// inicializa apenas os dois primeiros elementos  
float juros[5] = {5.0, 2.0};
```

- Ao **inicializar parcialmente um vetor**, os demais elementos recebem o valor zero

```
// primeiro elemento é 1 e os demais são 0  
long totais[500] = {1};
```

```
// todos os 500 elementos são iguais a zero  
long totais[500] = {0};
```

Inicialização de Vetores

- Deixando os **colchetes vazios na inicialização** o compilador conta os elementos para você

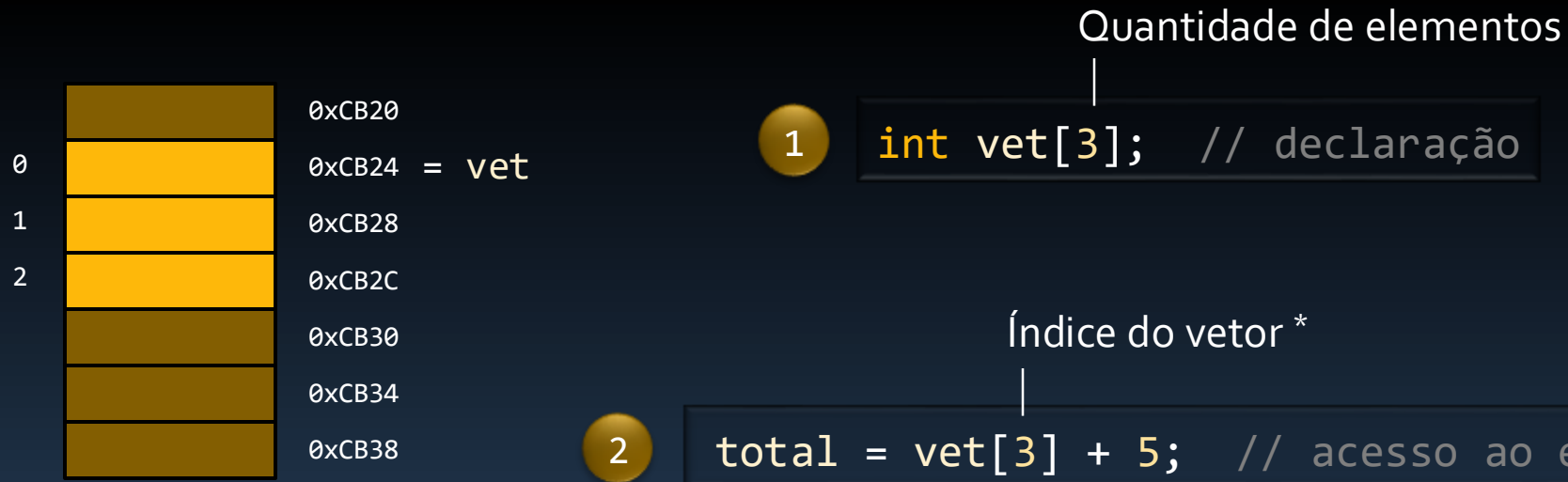
```
// cria um vetor com 4 elementos  
short coisas[] = {1, 5, 3, 8};
```

- **Omitir o número de elementos** sem inicializar o vetor constitui um erro

```
// compilador não sabe o tamanho do vetor  
short coisas[]; ❌
```

Acessando Elementos

- Os **colchetes** são usados:
 - 1 Para **declarar o vetor**
 - 2 Para **acessar seus elementos**



Acessando Elementos

- Acessar **posições inválidas de um vetor** é um erro grave que pode ter resultados inesperados

```
int vet[5] = {10, 20, 30, 40, 50};
```

```
vet[5] = 60;    // erro muito grave
```

```
vet[5000] = 60; // erro grave
```

- Uma posição inválida é uma **localização na memória** que:
 - Não pertence ao **vetor**
 - Não pertence ao **programa**

Vetores e Funções

- Vetores podem ser passados como **argumentos de funções**
 - Deve-se usar colchetes no protótipo

```
// função recebe um vetor de inteiros
```

```
int somaVetor(int []);
```

```
// função recebe um vetor de caracteres
```

```
int ultimoChar(char []);
```

- A definição deve dar um nome para o vetor

```
int somaVetor(int vet[])
```

```
{
```

```
...
```

Vetores e Funções

```
#include <iostream>
using namespace std;

int somaVetor(int []);

int main()
{
    int batatas[3] = {7, 8, 6};
    cout << "Total de batatas = ";
    cout << somaVetor(batatas) << endl;
    return 0;
}

int somaVetor(int vet[])
{
    return vet[0] + vet[1] + vet[2];
}
```

Vetores e Funções

- A saída do programa:

Total de batatas = 21

- O programa considera um **número fixo de elementos**
 - Para usar um número variável de elementos seria preciso passar também o **tamanho do vetor**

```
int somaVetor(int vet[], int tam)
{
    for (int i = 0; i < tam; ++i)
    {
        ...
    }
}
```

Alternativas para um Vetor

- A classe template **vector** é uma alternativa ao vetor tradicional da linguagem C++
 - O tamanho do vetor cresce automaticamente

```
#include <vector>
using namespace std;           // using std::vector;

int main()
{
    vector<int> vet;             // cria vetor vazio de inteiros
    vet.push_back(10);
    vet.push_back(20);

    size_t tam = vet.size();    // vetor tem 2 elementos
}
```

Alternativas para um Vetor

- A classe template **array** é outra alternativa
 - Tem tamanho fixo
 - Fornece maior segurança no acesso com `at()`

```
#include <array>
using namespace std; // using std::array;

int main()
{
    array<int, 5> vet;
    vet[0] = 10;
    vet[5] = 60;      // mesmo problema do vetor
    vet.at(5) = 60;   // verifica índice antes do acesso
}
```

Resumo

- **Vetores** podem armazenar **múltiplos valores**
 - Todos do mesmo tipo
 - Usando um único identificador

```
long totais[500]; // 500 valores tipo long
```

- Os elementos de um vetor são acessados através de **índices**

```
cout << totais[0]; // mostra primeiro elemento
```

- Um vetor pode ser **inicializado parcialmente**

```
// todos os 500 elementos são iguais a zero  
long totais[500] = {0};
```