>>> Programação Orientada a Objetos (POO)

... Introdução

Prof: André de Freitas Smaira

[~]$ _

>>> Estrutura de Dados - Exemplo

* Precisamos de uma **lista de inteiros**
* Precisamos guardar os números e sua quantidade
* Como?
    * **Vetor** de números
    * **Variável** com a quantidade

```
int lista[100];
int n;
n = 0;
/* outras operações */
lista[n] = 3;
n++;
```

[~]$ _

>>> Problemas

* lista e n são independentes no código
* Alguma outra forma?
    * struct

```
struct Lista {
    int lista[100];
    int n;
} umaLista;

umaLista.n = 0;
/* outras operações */
umaLista.lista[umaLista.n] = 3;
umaLista.n++;
```

[~]$ _

>>> Vantagens

* lista e n são relacionados pelo código (variável única)
* Temos um tipo que pode ser reutilizado
* Se tivermos mais de uma lista, não temos chance de cofundir as relações

>>> Orientação a Objetos

>>> Vantagens

* Estrutura de Dados associada a operações
* Exemplos:
    * Inserção
    * Busca

>>> Estrutura de Dados - Exemplo

```
struct Lista {
    int lista[100];
    int n;
};

bool busca(Lista *lst, int chave) {
    int i = 0;
    while (i < lst->n && lst->lista[i] != chave)
        i++;
    return !(i == lst->n);
}

void insere(Lista *lst, int valor) {
    lst->lista[lst->n] = valor;
    lst->n++;
}
```

>>> Problema

* No código, há separação entre estrutura e operações
* Isso está relacionado a Tipos Abstratos de Dados (TAD)

>>> Tipos Abstatos de Dados (TAD)

* Ligam **dados** com as **operações**
* Separam **interface** de **implementação**:
    * **Interface**: o que o cliente enxerga
    * **Implementação**: a representação e a implementação interna
* **Encapsulamento** da implementação.

>>> Estrutura de Dados - Exemplo

Tínhamos isso

```
struct Lista {
    int lista[100];
    int n;
};

bool busca(Lista *lst, int chave) {
    int i = 0;
    while (i < lst->n && lst->lista[i] != chave)
        i++;
    return !(i == lst->n);
}

void insere(Lista *lst, int valor) {
    lst->lista[lst->n] = valor;
    lst->n++;
}
```

>>> Estrutura de Dados - Exemplo

Agora isso

```
class Lista
{
    int lista[100];
    int n=0;
public:
    bool busca(int chave);
    void insere(int valor);
};

bool Lista::busca(int chave) {
    int i = 0;
    while (i < n && lista[i] != chave)
        i++;
    return !(i == n);
}

void Lista::insere(int valor)
{
    lista[n] = valor; n++;
}
```

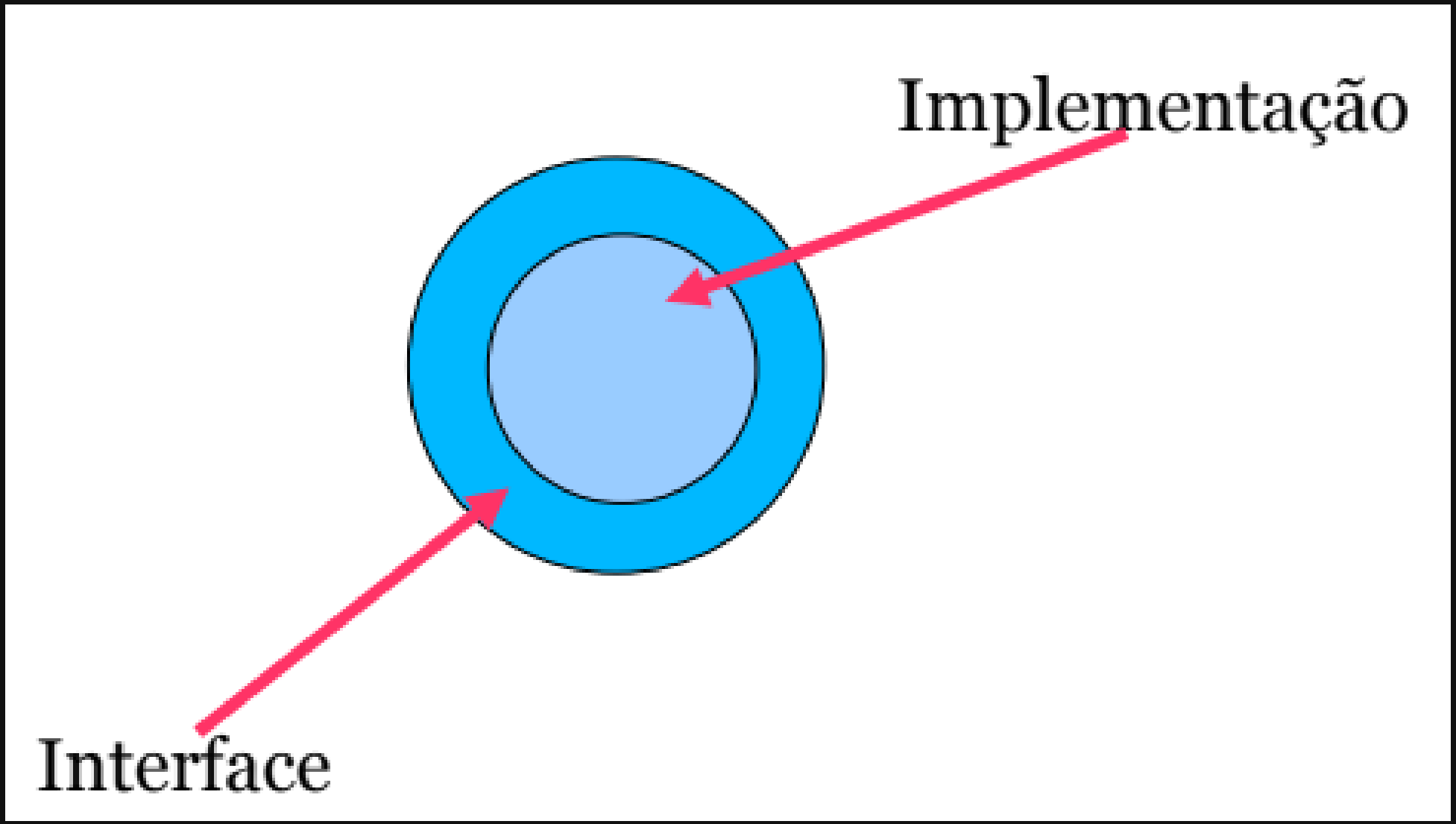
>>> Encapsulamento

* Implementação fica e **inacessível** para o cliente
* **Rotinas** ficam ligadas ao **tipo de dados**
* Se quisermos **mudar a representação** do tipo isso **não afetará os clientes** desde que a **interface permaneça inalterada**.

>>> Exemplo - Lista ligada

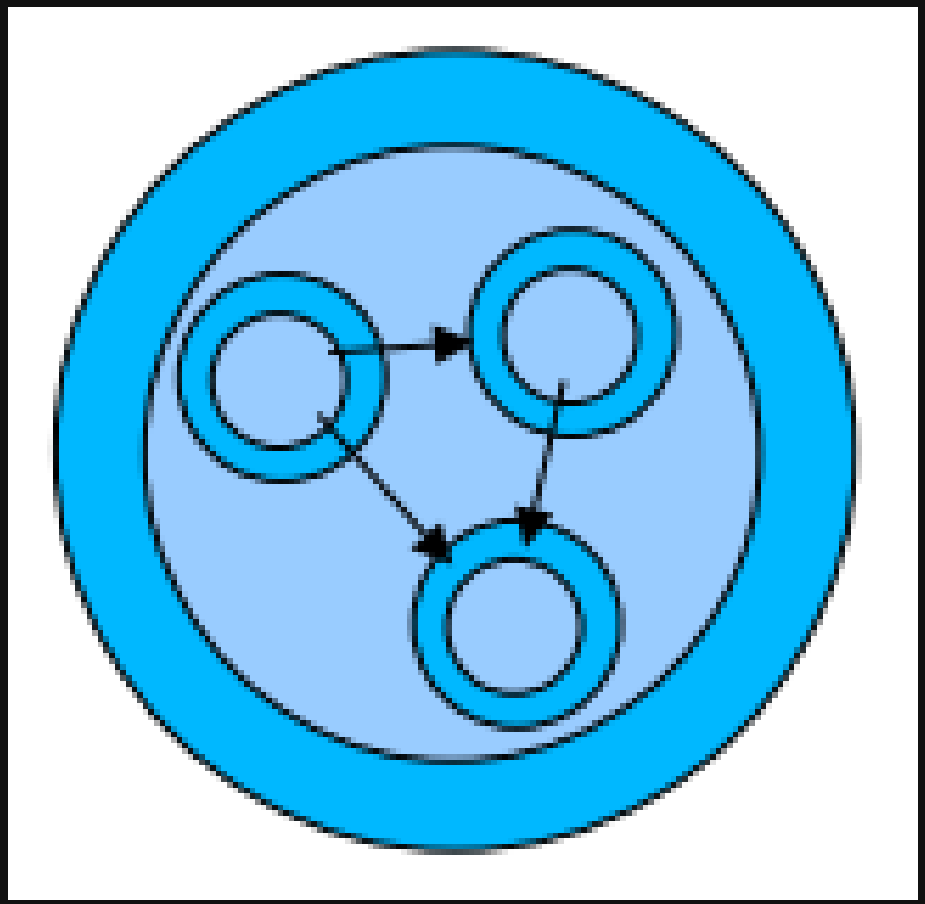
```
class Lista {
    struct No { int val; No *prox; };
    No *lista=0;
public:
    bool busca(int chave);
    void insere(int valor);
};

bool Lista::busca(int chave) {
    No *atual = lista;
    while(atual) {
        if(atual->val == chave) return true;
        atual = atual->prox;
    }
    return false;
}

void Lista::insere(int valor){
    No *novo = new No;
    novo->val = valor;
    novo->prox = lista;
    lista = novo;
}
```

>>> Interface



>>> Composição

* Um objeto pode ser constituído partes
* Cada uma é um tipo de dados distinto (outros objetos)
* Isso é composição



>>> Herança

* Alguns tipos são **subconjuntos** de outros
* **Exemplos**: carro é um veículo, estudante é uma pessoa, portento tem características e ações em comum
* Essa relação é **herança**
* **Reaproveita código** de outro tipo (subtipos)
* **Abstrai características** comuns

>>> Cachorro

```
Cachorro é um animal e ração é uma comida
class Comida { /* ... */ };
class Animal {
public:
    void anda();
    void come(Comida &c);
};
class Cachorro : public Animal {
public:
    void late();
};
class RacaoDeCachorro : public Comida { };
    /* ... */
RacaoDeCachorro mac_cao;
Cachorro rex;
rex.anda();
rex.come(mac_cao);
rex.late();
```

>>> Nomeclatura

* **Classe base** ou **Superclasse**: a mais geral
* **Classe derivada** ou **Subclasse**: a mais específica
* **Hierarquia de classes**: ordem da classes base para suas derivadas

>>> Polimorfismo

* Se objetos são derivados (direta ou indiretamente) de uma **mesma classe base**, então tem características e ações em **comum**
* A **parte comum** é definida pela **classe base**
* As **operações** devem se **adaptar** aos distintos tipos dos objetos
* Se **não sabemos os tipos** dos objetos, então estamos lidando com **polimorfismo**, ou seja, **selecionamos** as funcionalidades utilizadas de **forma dinâmica**

>>> Exemplo

* Num software de desenho temos diversas **figuras**
* **Mesmas ações**: desenhar, mover, apagar, copiar, etc.
* Cada tipo (círculo, triângulo, quadrado, etc.) tem uma **forma diferente** de realizar essas operações
* Operações devem ser realizadas sobre **grupos**
* **Exemplo**: mover várias figuras diferentes **simultaneamente**
* São inseridas numa **coleção**
* Cada figura aplica a **operação definida para si**

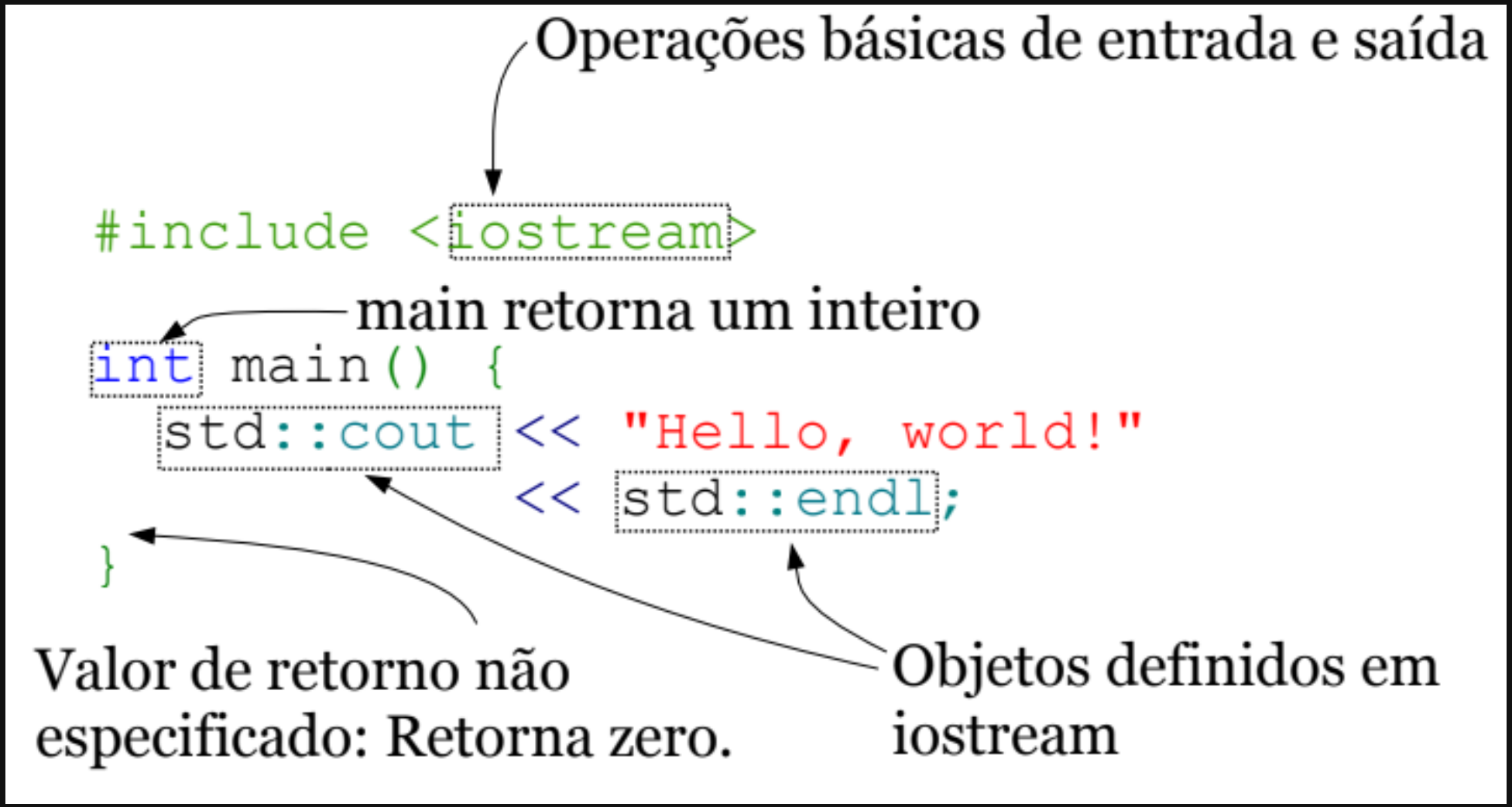
>>> Vantagens

* **Generalidade**: O algoritmo se preocupa apenas com **o que fazer**, mas não **como fazer** para cada objeto separadamente
* O algoritmo **não precisa saber** de antemão **todos os tipos**

>>> Já vimos tudo! e Agora?

* Detalhamento desses conceitos
* Introdução a C++
* Uso dos conceitos em C++ (POO de fato)
* As estruturas de dados em C++
* Alguns (vários?) conceitos adicionais

>>> Primeiro programa



>>> Tipoas de Variáveis

* Definidos pelo usuário
* Pré-definidos
    * bool
    * char, unsigned char, signed char, wchar_t
    * short, unsigned short
    * int, unsigned int
    * long, unsigned long
    * long long, unsigned long long
    * float, double, long double

>>> Variáveis

* Regras para nomes idênticas a C: podem ter letras, números e _, mas não podem começar por números
* C++ tem palavras-reservadas diferentes
    * alignas, alignof, and, and_eq, asm, auto, bitand, bitor, bool, break, case, catch, char, char16_t, char32_t, class, compl, concept, const, constexpr, const_cast, continue, decltype, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, noexcept, not, not_eq, nullptr, operator, or, or_eq, private, protected, public, register, reinterpret_cast, requires, return, short, signed, sizeof, static, static_assert, static_cast, struct, switch, template, this, thread_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while, xor, xor_eq

>>> Operadores

* **Operadores aritméticos**: +, -, *, /, %
* **Operadores binários**: ~(not), & (and), | (or), ^(xor)
* **Operadores lógicos**: ! (not), && (and), || (or)
* **Operadores de incremento/decremento**: ++, --
* **Operadores de comparação**: <, >, <=, >=, ==, !=
* **Operadores de atribuição**: =, +=, -=, etc.

>>> Conversão de Tipoas

Há 3 formas

```
double a;
int b,c,d;
a = 1.255*35;
b = (int)a; // Em C somente esse era válido
c = int(a);
d = static_cast<int>(a);
```

>>> Inicialização

* Valor inicial pode ser especificado na declaração (igual C)

```
int N; // N criada sem valor inicial.
N = 100; // Valor 100 atribuido.

int M = 100; // M criada valendo 100.
```

>>> Inicialização

* Mais uma forma (inválida para C)

```
int N{1000};
float tolerancia{0.00001};
```

>>> Constantes

* Uma variável pode ser declarada **constante** (não pode ser alterada depois)
* O **compilador verifica** se realmente o valor não é alterado
* Precisam de um **valor inicial**

```
int const N = 100;
```

>>> Constantes

* Pode ser usada também a palavra-chave **constexpr**, que indica que a expressão apresentada pode ser calculada em tempo de compilação

```cpp
#include <iostream>

// Função constexpr para calcular o fatorial de um número
constexpr int factorial(int n) {
    return (n <= 1) ? 1 : (n * factorial(n - 1));
}

int main() {
    // Variável constexpr para armazenar o resultado do fatorial de 5
    constexpr int result = factorial(5);

    std::cout << "O fatorial de 5 é: " << result << std::endl;

    return 0;
}
```

>>> Inferência de Tipos

* É possível definir variáveis cujo tipo é inferido pelo compilador a partir do valor de inicialização (novo em C++)

```
auto N = 100; // N é int.
auto x = 2.0; // x é double.
auto total = N*x; // total é double
```

>>> Arrays Unidimensionais

* Arrays de diversos elementos de um **mesmo tipo**
* Indexados **começando em zero**
* Ex: int v[10]: array de 10 elementos int, numerados de 0 a 9
* Elementos acessados com nome do array e índice entre colchetes: v[0], v[1], ..., v[9]
* Arrays podem ser inicializados

```
int v[10] = {1,2,3,4,5,6,7,8,9,10};
int m[2][3] = {{1,2,3},{4,5,6}};
```

* Ou somente para C++

```
int v[10]{1,2,3,4,5,6,7,8,9,10};
int m[][3]{{1,2,3},{4,5,6}};
```

>>> Saídas

* Saída com o uso de:
  * Objeto **std::cout**
  * Operador de inserção **<<** (insere valor em std::cout)
* Todos os **tipos básicos** podem ser utilizados

* C

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

* C++

```cpp
#include <iostream>

int main() {
    std::cout << "Hello World!\n";
    return 0;
}
```

>>> Outros Exemplos

```cpp
std::cout << "Hello, world\n";

std::cout << "Hello, world";
std::cout << std::endl;

std::cout << "Hello, world" << std::endl;

std::cout << "Hello, " << "world"
          << std::endl;

std::cout << "Deu: " << 7*6 << std::endl;
```

>>> Entradas

* Para entradas, utiliza-se:
  * O objeto **std::cin**
  * O operador de extração **>>** (extrai valor de std::cin).
* Aceita todos os **tipos básicos**
* **Não** é necessário **indicar o tipo**

* C

```c
#include <stdio.h>

int main() {
    int i;
    scanf(" %d", &i);
    return 0;
}
```

* C++

```cpp
#include <iostream>

int main() {
    int i;
    std::cin >> i;
    return 0;
}
```

>>> Outros Exemplos

```
double a, b, c;
std::cout << "Valores a multiplicar: ";
std::cin >> b >> c;
std::cout << "Produto: " << b*c
<< std::endl;
```

>>> Referências

* Apostila e Aulas do Gonzalo Travieso (IFSC/USP)