Breve introdução à Linguagem C++

Estrutura de Dados — QXD0010

Prof. Roberto Cabral

rbcabral@ufc.br

Universidade Federal do Ceará

2º semestre/2022

Referência

Este material foi baseado nos slides e nas notas de aula do professor Atílio.

Ementa

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Noções de análise de algoritmos;
- Recursividade;
- Tipos Abstratos de Dados;
- Algoritmos de Ordenação;
- Listas Sequenciais e Encadeadas;
- Pilhas;
- Filas;
- Árvores.

Objetivos

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Ensinar os alunos os conceitos fundamentais das estruturas de dados mais empregadas.
  - Compreender a importância de algoritmos eficientes na construção de estruturas de dados.
  - Aprender a analisar algoritmos e decidir sobre qual a melhor escolha a ser feita durante a implementação de uma dada estrutura de dados.
  - Aprender a implementar estruturas de dados essenciais como listas, pilhas, filas e árvores binárias tendo em vista sempre a eficiência e a reusabilidade de código.

Avaliações

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Avaliações:
  - Duas Avaliações Parciais, p_1 e p_2;
  - Dois Trabalhos, t_1 e t_2;
  - x Labs, l_i para $0 < i \leq x$.
- Composição da Nota:
  - Seja L a média dos labs, temos:
    - $\texttt{Média} = (p_1 + p_2 + t_1 + t_2 + L)/5$.

Considerações das Avaliações

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Não é permitido comunicação de qualquer espécie durante as avaliações, a menos que seja explicitamente liberado pelo professor;
- Não é permitido o uso de equipamentos eletrônicos de qualquer espécie, durante as avaliações, a menos que seja explicitamente definido pelo professor;
- Em caso de constatação de fraude, todos os alunos envolvidos ficarão com nota Zero;
- O conteúdo das avaliações é acumulativo.

Critérios de Aprovação

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- **Condição de Aprovação:** `Média` igual ou superior a 7 **E** frequência acima de 75%, ficando com conceito A;
- **Prova Final:**
  - Os alunos com `Média` maior que ou igual a 4 e menor que 7 tem o direito à Avaliação Final.
  - $\texttt{Nota Final} = 0,5 * \texttt{médiaParcial} + 0,5 * \texttt{avaliaçãoFinal}$.
  - **Condição de Aprovação:** O aluno é considerado aprovado caso obtenha pelo menos 4 na `Avaliação Final` **E** `Nota Final` igual ou superior a 5 **E** frequência acima de 75%, ficando com conceito B.

Tópicos desta Aula

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Compilação e execução
- Elementos básicos da linguagem C++
- Estruturas de seleção e repetição
- Vetores
- Structs e enumerações
- Ponteiros
- Alocação dinâmica de memória
- Ponteiros para ponteiros
- Arrays de caracteres

Tutoriais online para estudo rápido e consulta

- Site LearnCpp.com: https://www.learncpp.com
- Site cppreference.com: https://en.cppreference.com/w/
- Site cplusplus.com: http://www.cplusplus.com/reference

Apresentação da Linguagem C++

- Linguagem multi-paradigma, de uso geral e nível médio.
- Bjarne Stroustrup desenvolveu o C++ (originalmente com nome *C with Classes*) em 1983 no Bell Labs como um adicional à linguagem C.

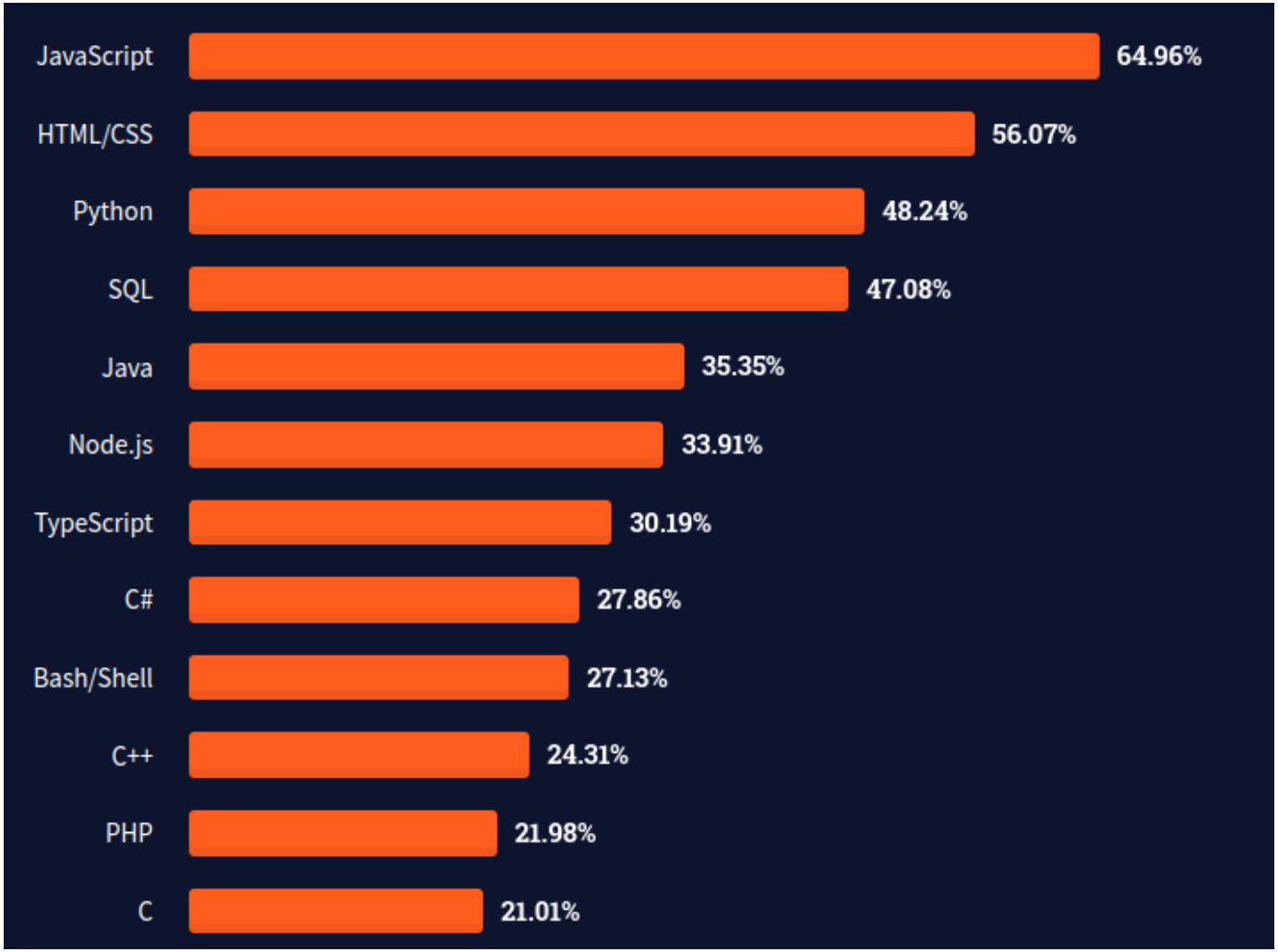
- Desde a década de 1990 possui forte uso comercial e acadêmico.
- Novas características foram adicionadas com o tempo (funções virtuais, sobrecarga de operadores, herança múltipla, templates e tratamento de exceções).

Apresentação da Linguagem C++

Padronizações da linguagem:

- C++98: ISO de 1998
- C++03: revisão em 2003
- Em 2011, o padrão C++11 foi lançado, adicionando vários recursos novos, ampliando ainda mais a biblioteca padrão e fornecendo mais facilidades aos programadores C++.
- C++14 foi lançada em dezembro de 2014.
- C++17 foi lançada em dezembro de 2017.
- Versão mais recente: C++20, lançada em 2020, mas nem todas as funcionalidades estão disponíveis ainda no compilador g++.

Linguagens mais usadas em 2021



Fonte: https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies

Primeiro Programa — Hello World

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```cpp
#include <iostream>

int main() {
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

Supondo que o programa acima esteja no arquivo `programa.cpp`, para compilar no terminal do Linux:

- `$ g++ -Wall -Wextra programa.cpp -o main`

Para executar no terminal:

- `$ ./main`

Elementos básicos da linguagem C++

Elementos básicos da linguagem

Como em outras linguagens:

- Comentários de código
- Variáveis e Constantes
- Identificadores
- Tipos Fundamentais
- Representação numérica
- Vetores, strings, ponteiros, estruturas e enumerações
- Estruturas de controle de fluxo
- Funções
- entre outras...

Comentários

- Um comentário é uma descrição inserida diretamente no código-fonte do programa e que é ignorada pelo compilador. Serve apenas para uso do programador.
- C++ permite fazer comentários de duas maneiras diferentes: por linha ou por bloco.

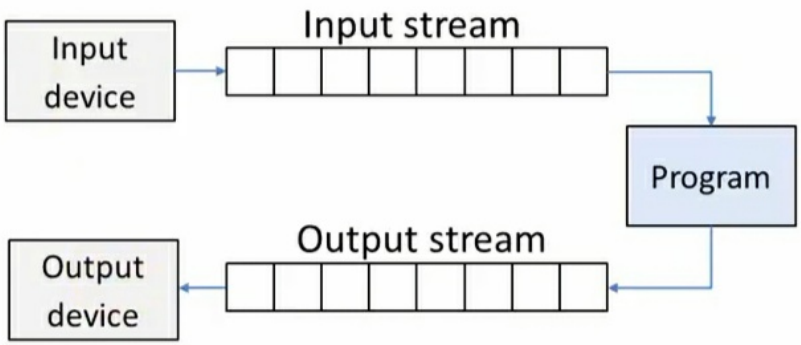
```
1  #include <iostream>
2
3  int main() {
4    /* --- Exemplo de comentario em bloco ---
5    A funcao std::cout
6    serve para
7    escrever na tela
8    */
9    std::cout << "Hello World"; // Um comentario em linha
10
11   return 0;
12 }
```

Entrada/Saída em C++

Streams

- Um stream (fluxo) é uma sequência de bytes que podem ser acessados sequencialmente.
  - Em C++, Entrada e Saída são implementados com streams.

- Existem dois tipos de streams:
  - **Input Stream**: usadas para gerenciar dados de entrada vindos de um teclado, arquivo, rede, etc.
    - **Dispositivo de entrada padrão:** teclado
  - **Output Stream**: usadas para gerenciar dados de saída enviados para um terminal, arquivo, impressora, etc.
    - **Dispositivo de saída padrão:** terminal



Streams

- Cabeçalho responsável por I/O: `iostream`.
  - Esse cabeçalho define os tipos: `istream` (input stream) e `ostream` (output stream), assim como duas variáveis, chamadas `std::cin` (common input) e `std::cout` (common output).

- As variáveis `std::cin` e `std::cout` são usadas em conjunto com os operadores $>>$ (operador de extração) e $<<$ (operador de inserção), respectivamente.

- O formato geral para ler dados do teclado é:

  std::cin >> variavel1 >> variavel2 >> ... >> variavelN

- O formato geral para mostrar dados na tela é:

  std::cout << item1 << item2 << ... << itemN

  - Os dados na saída podem ser variáveis ou constantes literais de qualquer tipo nativo.

Formatação da Saída

- O cabeçalho `iomanip` fornece um conjunto de funções chamadas manipuladores, que podem ser usadas para formatar a saída.

- Os manipuladores podem ser usados como uma cadeia em uma declaração:

```
std::cout << manip1 << manip2 << manip3 << item;
```

Alguns manipuladores e o seu significado

Números inteiros e manipulação de base

Manipulador	Significado
`std::dec`	usa a base decimal
`std::oct`	usa a base octal
`std::hex`	usa a base hexadecimal
`std::setbase(int base)`	seta a base para 8, 10 ou 16
`std::showbase`	Faz com que a base do número seja escrita antes do número: 0 para octal, 0x para hexadecimal. Resetado com `noshowbase`.

Alguns manipuladores e o seu significado

Números de ponto-flutuante

Manipulador	Significado
`std::scientific`	usa notação científica
`std::fixed`	usa notação de ponto fixo
`std::setprecision(int d)`	seta o número de casas decimais para d
`std::showpoint`	Força números de ponto flutuante serem impressos com ponto. Geralmente usado com `fixed` para garantir um certo número de casas decimais, mesmo que sejam zeros. Resetado com `noshowpoint`.
`std::showpos`	Faz com que números positivos sejam precedidos pelo sinal +. Resetado com `noshowpos`.

Exemplo 1

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // Controlling precision of floating-pointing values
2  #include <iostream> // manipulators1.cpp
3  #include <iomanip>
4  #include <cmath>
5  using namespace std;
6
7  int main() {
8      double root2{sqrt(2.0)}; // calculate square root of 2
9
10     cout << scientific << "sqrt(2) = " << root2 << endl;
11
12     cout << "Square root of 2 with precisions 0-9.\n";
13     cout << fixed; // use fixed point notation
14
15     // set precision for each digit, then display square root
16     for(int places{0}; places <= 9; ++places) {
17         cout << setprecision(places) << root2 << "\n";
18     }
19 }
```

Exemplo 2

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // Controlling precision of floating-pointing values
2  #include <iostream> // manipulators2.cpp
3  #include <iomanip>
4  using namespace std;
5
6  int main() {
7      double number1{4.0 / 2.0};
8      double number2{4.0 / 3.0};
9
10     cout << showpos; // show plus sign on positive numbers
11
12     cout << number1 << "\n"
13          << number2 << "\n\n" << flush;
14
15     cout << fixed << showpoint << setprecision(2);
16
17     cout << number1 << "\n"
18          << number2 << endl;
19 }
```

Outros manipuladores

Manipulador	Significado
`std::setw(int n)`	seta a largura do campo para n
`std::setfill(char c)`	faz o caractere de preenchimento ser o c
`std::left`	Ajusta a saída para a esquerda
`std::right`	Ajusta a saída para a direita
`std::boolalpha`	Faz com que valores booleanos sejam impressos como as palavras `true` ou `false`. Resetado pelo manipulador `noboolalpha`.
`std::endl`	Insere uma nova linha e libera o stream

Exemplo 3

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #include <iostream> // manipulators3.cpp
2  #include <iomanip>
3  using namespace std;
4
5  int main() {
6      int number {12345};
7
8      cout << setfill('*'); // sticky manipulator
9
10     cout << "\"" << left
11          << setw(10)
12          << number << "\"" << endl;
13
14     cout << "\"" << number << "\"" << endl;
15
16     cout << "\"" << right
17          << setw(10)
18          << number << "\"" << endl;
19 }
```

Formatação da saída com manipuladores

Exemplos

- `std::cout << std::setw(10) << 12345;`
  - Imprime o valor 12345 alinhado à direita em um campo com largura 10.
- `std::cout << std::setw(10) << std::setfil('*') << 1.23;`
  - Imprime na tela ******1.23
- `std::cout << std::fixed << std::setw(10) << setprecision(4) << sqrt(2);`
  - Imprime o valor `sqrt(2)` com 4 casas decimais em um campo de largura 10.
- `std::cout << std::endl;`
  - Insere uma nova linha, liberando a stream `std::cout`.

Estados de erro de uma Stream

- Todo objeto stream contém um conjunto de variáveis de erro que representam o estado de uma stream.
- Por exemplo, a stream `std::cin` contém as seguintes variáveis:
  - `failbit`: possui valor `1` se e somente se o tipo de dado da entrada estiver errado.
  - `eofbit`: possui valor `1` se e somente se estiver chegado ao final da leitura.
  - `badbit`: possui valor `1` se e somente se a operação falhar de modo irrecuperável. Exemplo: falha de disco no momento da leitura.
- `std::cin` possui as funções `fail()`, `eof()` e `bad()`, que devolvem o valor dessas variáveis.

Estados de erro de uma Stream

- `std::cin` possui a função `good()` que retorna `1` quando todas as três funções `fail()`, `eof()` e `bad()` retornam `0`.

- `std::cin` possui a função `clear()`, que limpa o estado de erro da stream `std::cin`, setando o valor de goodbit para 1 e os valores dos demais bits para 0.
  - Deste modo, usando a função clear(), é possível continuar usando a stream, mesmo depois de ocorrer um erro.

Exemplo


```cpp
#include <iostream> //errorTreatment.cpp
#include <limits>
using namespace std;

void ignore_line() {
  cin.ignore(numeric_limits<streamsize>::max(), '\n');
}

int readInt() {
  while(true) {
    int value{};
    cin >> value;
    if(cin.fail()) {
      cerr << "fail: enter a valid integer\n";
      cin.clear();
      ignore_line();
    }
    else {
      ignore_line();
      return value;
    }
  }
}
```

Exemplo (cont.)


```
24 int main()
25 {
26   int x = readInt();
27   cout << "x = " << x << endl;
28 }
```

Materiais de estudo

Onde encontrar materiais sobre I/O em C++?

- Capítulo 15 (Entrada/saída de fluxo) do livro C++ Como Programar, Quinta Edição.
- `https://cplusplus.com/reference/library/manipulators/`
- `https://en.cppreference.com/w/cpp/io/manip`
- `https://www.learncpp.com/cpp-tutorial/stdcin-and-handling-invalid-input/`

Variáveis

Variáveis

- Em computação, uma variável é uma posição de memória onde poderemos guardar determinado dado ou valor e modificá-lo ao longo da execução do programa.
- Quando criamos uma variável e armazenamos um valor dentro dela, o computador reserva um espaço associado a um endereço de memória onde podemos guardar o valor dessa variável.
- O nome da variável é chamado identificador.

```
1 #include <iostream>
2
3 int main() {
4   int x;  //declara a variavel mas nao define o valor
5   std::cout << "x = " << x << "\n";
6   x = 5;  //define o valor de x como sendo 5
7   std::cout << "x = " << x << std::endl;
8   return 0;
9 }
```

Inicialização de variáveis


```
1  #include <iostream> // prog07.cpp
2  using namespace std;
3
4  int main() {
5    int z = 23; // por atribuicao -- assignment
6    std::cout << "Valor de z: " << z << std::endl;
7    int k = 45.89; // ok
8    std::cout << "Valor de k: " << k << std::endl;
9
10   int w( 23 ); // direct initialization
11   std::cout << "Valor de w: " << w << std::endl;
12   int s( 32.75 ); // ok
13   std::cout << "Valor de s: " << s << std::endl;
14
15   int y{ 23 }; // uniform initialization (C++11)
16   std::cout << "Valor de y: " << y << std::endl;
17   int x{ 5.6 }; /** ERRO DE COMPILACAO **/
18   std::cout << "Valor de x: " << x << std::endl;
19
20   return 0;
21 }
```

Identificadores

A linguagem C++ estipula algumas regras para a escolha dos identificadores:

- Um identificador é um conjunto de caracteres que podem ser letras, números ou *underscores* (_).
- O identificador deve sempre iniciar com uma letra ou o *underscore* (_).
- A linguagem C é case-sensitive, ou seja, uma palavra escrita utilizando caracteres maiúsculos é diferente da mesma palavra escrita com caracteres minúsculos.
- Palavras reservadas não podem ser usadas como nome de variáveis.
- As palavras reservadas são um conjunto de 84 palavras reservadas da linguagem C++. Elas formam a sintaxe da linguagem e possuem funções específicas.

Palavras-chave da linguagem C++

A partir do padrão C++17:

alignas (C++11)	decltype (C++11)	namespace	struct
alignof (C++11)	default	new	switch
and	delete	noexcept (C++11)	template
and_eq	do	not	this
asm	double	not_eq	thread_local (C++11)
auto	dynamic_cast	nullptr (C++11)	throw
bitand	else	operator	true
bitor	enum	or	try
bool	explicit	or_eq	typedef
break	export	private	typeid
case	extern	protected	typename
catch	false	public	union
char	float	register	unsigned
char16_t (C++11)	for	reinterpret_cast	using
char32_t (C++11)	friend	return	virtual
class	goto	short	void
compl	if	signed	volatile
const	inline	sizeof	wchar_t
constexpr (C++11)	int	static	while
const_cast	long	static_assert (C++11)	xor
continue	mutable	static_cast	xor_eq

Tipos de dados fundamentais

- `char`: representa um caractere. Um char requer exatamante 1 byte de espaço de memória e varia de -128 a 127.
- `int`: tipo de dado que armazena um inteiro. Inteiros normalmente requerem 4 bytes de espaço de memória e variam de -2.147.483.648 a 2.147.483.647.
- `long`: armazena um inteiro e requer pelo menos 8 bytes de espaço de memória. Varia de -9.223.372.036.854.775.808 a 9,223.372.036.854.775.807.
- `bool`: representa os valores booleanos `true` e `false`. Ocupa 1 byte de memória.
- `float`: representa valores de ponto flutuante. Requer pelo menos 4 bytes de espaço de memória.
- `double`: representa valores de ponto flutuante de precisão dupla. Requer pelo menos 8 bytes de espaço de memória.

Tipos de dados fundamentais

Modificadores de Tipo de Dados: são usados para modificar o intervalo de valores que um tipo de dado fundamental pode suportar. Os modificadores de tipo de dados disponíveis em C++ são:

- `unsigned`
- `short` (pode ser combinado com `int`)
- `long` (pode ser combinado com `int` e `double`)

Operador `sizeof`

Podemos exibir o tamanho de todos os tipos de dados usando o operador **`sizeof()`**:

```
1  #include <iostream> // prog05.cpp
2  using namespace std;
3
4  int main() {
5      cout << "char: " << sizeof(char) << endl;
6      cout << "bool: " << sizeof(bool) << endl;
7      cout << "short: " << sizeof(short) << endl;
8      cout << "short int: " << sizeof(short int) << endl;
9      cout << "int: " << sizeof(int) << endl;
10     cout << "long int: " << sizeof(long int) << endl;
11     cout << "long: " << sizeof(long) << endl;
12     cout << "long long: " << sizeof(long long) << endl;
13     cout << "float: " << sizeof(float) << endl;
14     cout << "double: " << sizeof(double) << endl;
15     cout << "long double: " << sizeof(long double) << endl;
16 }
```

Constantes

Uma constante é uma variável especial que permite guardar determinado dado na memória do computador, com a certeza de que ele não se alterará durante a execução do programa.

A fim de declarar uma constante, basta colocar a palavra-chave **const** antes do tipo de variável:

```
1  #include <iostream> // prog04.cpp
2  using namespace std;
3
4  int main() {
5    const double gravidade = 9.8; // declarando constante
6    gravidade = 9.9; // vai dar erro de compilacao
7
8    std::cout << "Digite sua idade: ";
9    int idade;
10   std::cin >> idade;
11
12   const int idadeUsuario = idade; // declarando constante
13   std::cout << idadeUsuario;
14   return 0;
15 }
```

Constantes

O C++ suporta dois tipos de constantes:

- Constantes em tempo de compilação: são aquelas cujos valores de inicialização podem ser resolvidos em tempo de compilação. Exemplo: `gravidade`, do exemplo anterior.
  - Constantes deste tipo permitem que o compilador realize otimizações.

- Constantes em tempo de execução: são aquelas cujos valores de inicialização só podem ser resolvidos em tempo de execução. Exemplo: `idadeUsuario`, do exemplo anterior.

Constantes — palavra-chave `constexpr`

- O especificador constexpr declara que é possível avaliar o valor da variável em tempo de compilação.
  - recurso adicionado na versão C++11.

- Exemplo:

```
1 // o valor da constante 'gravidade' sera determinado
2 // em tempo de compilacao
3 constexpr double gravidade { 9.8 };
4
5 // o valor da constante 'soma' sera determinado
6 // em tempo de compilacao
7 constexpr int soma { 4 + 5 };
```

- **Dica:** Se a variável que não deve ser modificada após a inicialização e o inicializador é conhecido em tempo de compilação, então declare-a como constexpr.

Namespaces

Namespaces

- Um namespace é uma região declarativa que fornece um escopo para os identificadores (os nomes de tipos, funções, variáveis, etc) dentro dele.
- São usados para organizar o código em grupos lógicos a fim de evitar colisões de nomes que podem ocorrer especialmente quando sua base de código inclui várias bibliotecas.
- Exemplo: namespace `std` (standard)

```
1 #include <iostream> // prog60.c
2
3 int main() {
4   std::cout << "Hello world" << std::endl;
5   return 0;
6 }
```

- O símbolo **::** é chamado operador de resolução de escopo.

Namespaces

- Os identificadores fora do namespace podem acessar os membros das seguintes formas:
  - usando o nome totalmente qualificado para cada identificador.
    Exemplo: `std::vector<std::string> vec;`
  - por meio de uma declaração `using` para um único identificador.
    Exemplo: `using std::cout;`
  - por meio de uma diretiva `using` para todos os identificadores no namespace.
    Exemplo: `using namespace std;`

- Boa prática: Use os prefixos de namespace explicitamente a fim de acessar os identificadores definidos no namespace.

Definindo seu próprio namespace

- A palavra-chave `namespace` é usada para declarar um escopo que contém um conjunto de objetos relacionados.

```
1 /**
2  * Arquivo mymath.h
3  */
4 namespace math {
5   int sum(int x, int y) { return x+y; }
6   int sub(int x, int y) { return x-y; }
7   int mul(int x, int y) { return x*y; }
8   int div(int x, int y) { return x/y; }
9 }
```

Definindo seu próprio namespace (Cont.)


```
1  #include <iostream> //prog61.cpp
2  #include "mymath.h"
3
4  int main() {
5    int a{ 5 }, b = 10;
6    std::cout << math::sum(a,b) << std::endl;
7    std::cout << math::sub(a,b) << std::endl;
8    std::cout << math::mul(a,b) << std::endl;
9    std::cout << math::div(a,b) << std::endl;
10   return 0;
11 }
```

Declarações using – Exemplo 1

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #include <iostream> //prog62.cpp
2  using std::cout;
3  using std::endl;
4
5  namespace math {
6    int sum(int x, int y) { return x+y; }
7    int sub(int x, int y) { return x-y; }
8    int mul(int x, int y) { return x*y; }
9    int div(int x, int y) { return x/y; }
10 }
11
12
13
14 int main() {
15   using namespace math;
16   int a{ 5 }, b{ 4 };
17   cout << sum(a,b) << endl;
18   cout << sub(a,b) << endl;
19   cout << mul(a,b) << endl;
20   cout << div(a,b) << endl;
21   return 0;
22 }
```

Declarações using – Exemplo 2

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #include <iostream> //prog63.cpp
2
3  namespace math {
4    int sum(int x, int y) { return x+y; }
5    int sub(int x, int y) { return x-y; }
6    int mul(int x, int y) { return x*y; }
7    int div(int x, int y) { return x/y; }
8  }
9
10 int main() {
11   using std::cout; // using declaration
12   using std::endl; // using declaration
13
14   int a{ 5 }, b{ 4 };
15   cout << math::sum(a,b) << endl;
16   cout << math::sub(a,b) << endl;
17   cout << math::mul(a,b) << endl;
18   cout << math::div(a,b) << endl;
19   return 0;
20 }
```

Diretiva using

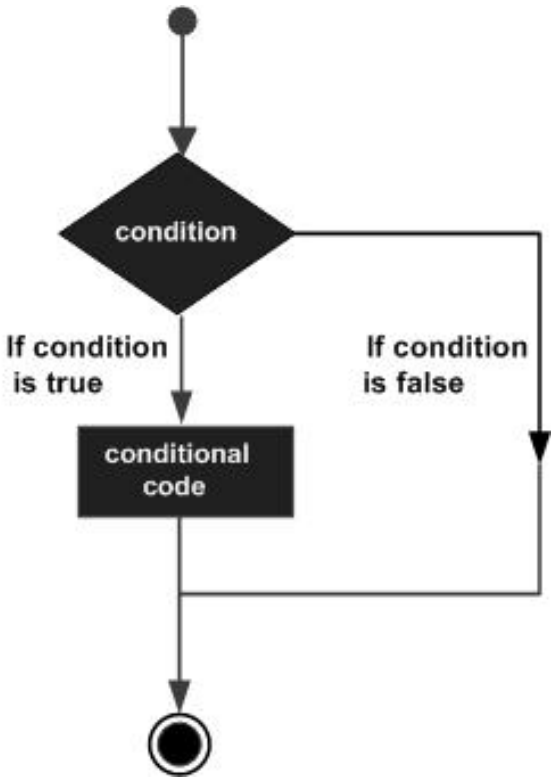

```
1  #include <iostream> //prog64.cpp
2
3  namespace math {
4    int sum(int x, int y) { return x+y; }
5    int sub(int x, int y) { return x-y; }
6    int mul(int x, int y) { return x*y; }
7    int div(int x, int y) { return x/y; }
8  }
9
10 using namespace math;
11
12 int main() {
13   using namespace std;
14   int a{ 5 }, b{ 4 };
15   cout << sum(a,b) << endl;
16   cout << sub(a,b) << endl;
17   cout << mul(a,b) << endl;
18   cout << div(a,b) << endl; // Erro de compilacao
19   return 0;
20 }
```

- `div_t div(int numer, int denom);` retorna um struct e está definido sob o namespace `std`

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Estruturas de Seleção

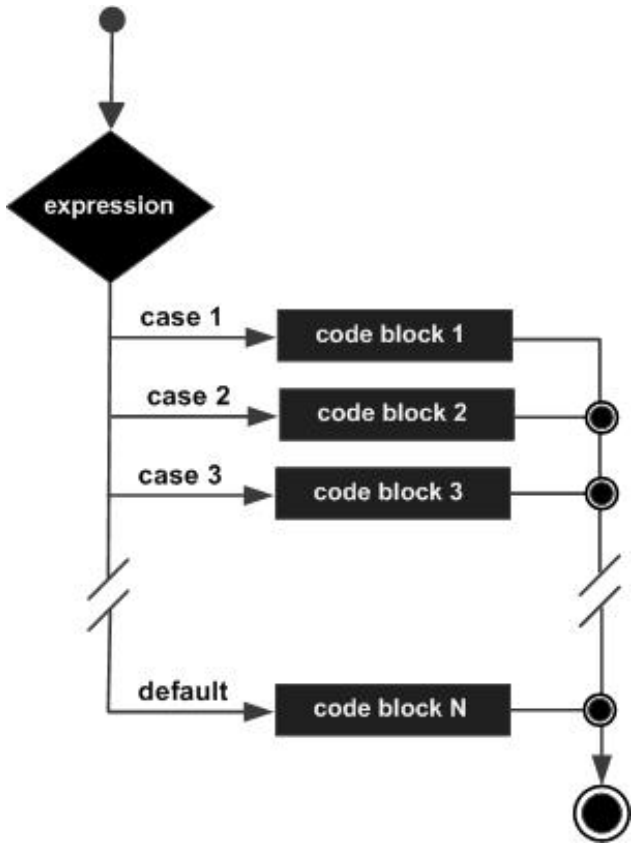
Estruturas de Seleção — If.. else




```
1  int c;
2  std::cin >> c;
3
4  if (c == 1) { // If
5    std::cout << "igual a 1";
6  }
7
8  if (c == 2) { // If .. else
9    std::cout << "igual a 2";
10 } else {
11   std::cout << "nao eh 1 nem 2";
12 }
13
14 if (c == 3) { // If else encadeado
15   std::cout << "igual a 3";
16 } else if (c == 4) {
17   std::cout << "igual a 4";
18 } else {
19   std::cout << "nao eh 1,2,3,4";
20 }
```

Estruturas de Seleção — Switch

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ



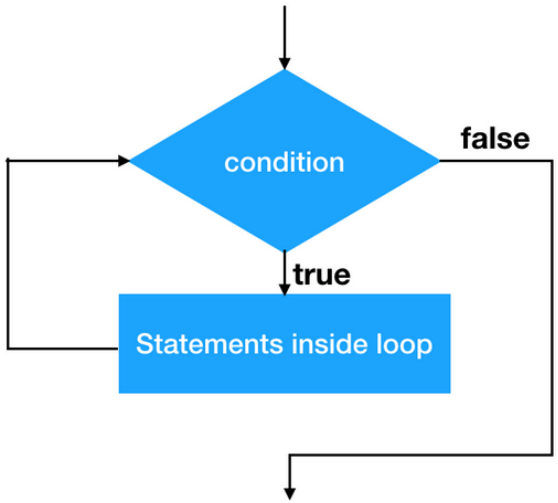

```
1  int c;
2  std::cin >> c;
3
4  switch (c) // int,char,short,long
5  {
6      case 1:
7          std::cout << 1 << '\n';
8          break;
9      case 2:
10         std::cout << 2 << '\n';
11         break;
12     case 3:
13         std::cout << 3 << '\n';
14         break;
15     case 4:
16         std::cout << 4 << '\n';
17         break;
18     default:
19         std::cout << 5 << '\n';
20         break;
21 }
```

Estruturas de Repetição

Estruturas de repetição (loops)

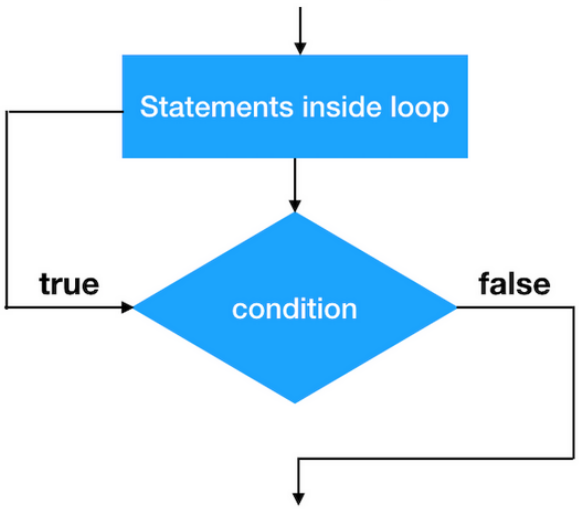
UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

while loop




```
1 int contador = 0;
2
3 while (contador < 10) {
4   std::cout << count << " ";
5   count++;
6 }
```

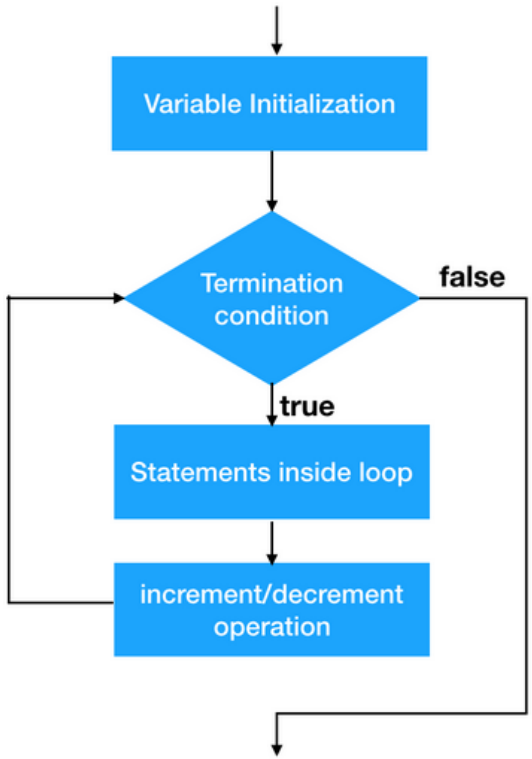
do..while loop




```
1 do {
2   std::cout << contador << " ";
3   contador++;
4 } while (contador < 10);
```

Estruturas de repetição (loops)

for loop




```
1 for (int i = 0; i < 10; i++)
2     std::cout << i << " ";
```

Vetores

Vetores (Arrays)

- Um vetor (array) é uma estrutura de dados que nos permite acessar muitas variáveis do mesmo tipo por meio de um único identificador.
- Em quais casos na programação pode ser necessário o uso de um vetor?
- Exemplo:

```
1  #include <iostream> // prog15.cpp
2
3  int main() {
4      double array[3]; // aloca 3 doubles
5      array[0] = 2.0;
6      array[1] = 3.0;
7      array[2] = 4.3;
8
9      std::cout << "A media eh " <<
10         (array[0] + array[1] + array[2]) / 3 << "\n";
11
12     return 0;
13 }
```

Declarando e inicializando vetores

`int numbers[10];`	Um array de 10 inteiros.
`constexpr int SIZE = 10;` `int numbers[SIZE];`	É uma boa ideia usar uma variável constante para o tamanho.
`int size = 10;` `int numbers[size];`	**Atenção:** Em C++ padrão, o tamanho do vetor deve ser uma constante. Esta definição de vetor não funcionará em todos os compiladores.
`int vec[5] {0,1,4,9,16};`	Um vetor de cinco inteiros, inicializado com "brace inicialization"
`int vec[] {0,1,4,9,16};`	O tamanho do vetor pode ser omitido neste caso, pois ele é definido pelo número de valores iniciais.
`int squares[5] = {0,1,4};`	Se você fornecer menos valores iniciais que o tamanho, os valores restantes serão definidos como 0. Esse array contém 0, 1, 4, 0, 0.

Tamanho de um vetor

Se estivermos no mesmo escopo em que um array foi definido, podemos determinar seu tamanho usando o operador `sizeof`.

Exemplo:

```
1  #include <iostream> // prog16.cpp
2
3  int main() {
4      int array[] = { 1, 1, 2, 3, 5, 8, 13, 21 };
5
6      std::cout << "Tamanho do vetor: ";
7      std::cout << sizeof(array) / sizeof(array[0]) << "\n";
8
9      return 0;
10 }
```

- **Obs.:** Só funcionará se `sizeof` estiver na mesma função na qual o vetor estiver declarado.

Vetores e laços `for-each`

- O C++11 introduziu o **loop for-each**, que fornece um método mais simples para iterar sobre os elementos de um vetor.

```
1  #include <iostream> // prog31.cpp
2
3  int main() {
4      double fibonacci[] = {0,1,1,2,3,5,8,13};
5
6      for (int n : fibonacci)
7          std::cout << n << endl;
8
9      return 0;
10 }
```

- Atenção: o vetor e o loop devem estar no mesmo escopo.

Palavra-chave auto

- A partir do C++11, quando uma variável é declarada seguida de inicialização, podemos usar a palavra-chave `auto` no lugar do tipo da variável.
  - Com isso, o tipo da variável será inferido da expressão de inicialização.
- Exemplo (equivalente ao do slide anterior):

```
1  #include <iostream> // prog32.cpp
2
3  int main() {
4      int fibonacci[] = {0,1,1,2,3,5,8,13,21,34,55,89};
5
6      for (auto n : fibonacci)
7          std::cout << n << ' ';
8
9      return 0;
10 }
```

- Qual vantagem de se usar a palavra-chave auto?

Arrays de caracteres – Strings

Arrays de caracteres — Strings

- Uma string é uma sequência de caracteres adjacentes na memória do computador, ou seja, um array em que cada elemento é do tipo `char`.
- C++ suporta dois tipos diferentes de string:

(1) strings como array de caracteres (C-style strings). Exemplo:

```
1 char palavra[20] = "abracadabra";
2 char minhaString[] = "string";
```

(2) `std::string` (como parte da biblioteca padrão). Exemplo:

```
1 std::string palavra;
2 palavra = "abracadabra"; // Atribuicao funciona aqui
```

- Na verdade, `std::string` é implementada usando C-style strings.

C-style Strings — Características

- O último caractere de toda string é o caractere '\0'.
- Inicializando uma string:

```
1 char str [10] = { 'J', 'o', 'a', 'o', '\0' };
2
3 // A forma de inicializacao abaixo possui a vantagem
4 // de ja inserir o caractere '\0' no final da string
5 char str [10] = "Joao" ;
```

- Acessando um elemento da string: por se tratar de um array, cada caractere pode ser acessado individualmente por indexação como em qualquer outro vetor.

```
char str[6] = "Teste";
```

T	e	s	t	e	\0

```
str[0] = 'L';
```

L	e	s	t	e	\0

C-style Strings — Características

- Inicialização de strings: C-style strings seguem as mesmas regras que os arrays: você pode inicializar a string no momento da criação, mas você não pode atribuir valores a ela usando o operador de atribuição depois disso.

```
1  #include <iostream> // prog18.cpp
2
3  int main() {
4    char str1[20] = "Oi gente"; // Ok!
5    char str2[20];
6
7    str2 = str1; // ERRADO!
8    str2 = "SOL"; // ERRADO!
9    str2[0] = 'S';
10   str2[1] = 'O';
11   str2[2] = 'L';
12   str2[3] = '\0';
13
14   return 0;
15 }
```

C-style Strings — Lendo e Imprimindo

- Imprimindo com std::cout: `std::cout` imprime caracteres até encontrar o terminador nulo '$\backslash 0$'.
- Lendo strings: uma forma segura de ler strings do teclado é usar o comando
  `std::cin.get(char* s, streamsize n)`,
  onde `s` é um array de caracteres e `n-1` é o número máximo de caracteres que será lido e armazenado em `s`.
- Exemplo:

```
1  #include <iostream> // prog19.cpp
2
3  int main() {
4      char name[10]; // declara um array de tamanho 10
5      std::cout << "Digite seu nome: ";
6      std::cin.get(name, 10);
7      std::cout << "Voce digitou: " << name << "\n";
8
9      return 0;
10 }
```

C-style Strings — Lendo e Imprimindo


```
1  #include <iostream> // prog65.cpp
2  #include <cstring>
3  #include <limits>
4  using namespace std;
5
6  const int MAX{ 10 };
7
8  int main() {
9    char vec[MAX];
10
11   do {
12     cout << "Digite uma string: " << endl;
13     cin.get(vec, MAX);
14     cin.ignore(numeric_limits<streamsize>::max(), '\n');
15     cout << "String digitada foi: " << vec << endl;
16   } while(strcmp(vec, "0") != 0);
17
18   return 0;
19 }
```

Manipulando C-style Strings (Biblioteca `cstring`)

Duas funções da biblioteca `cstring`:

- `char* strcpy(char *destino, const char *origem);`
  Copia a string `origem` para a string `destino`, incluindo o caractere nulo de terminação (e parando nesse ponto).
  Para evitar overflows, o tamanho do vetor `destino` deve ser longo o suficiente para conter a string `origem`.

- `size_t strlen(const char *str);`
  Retorna o número de caracteres da string `str`, excluindo o caractere nulo de terminação. O valor `size_t` é qualquer inteiro sem sinal (`unsigned int`).

- Para outras funções, consultar:
  `http://www.cplusplus.com/reference/cstring/`

Exemplo com strlen e strcpy


```
1  #include <iostream> // prog20.cpp
2  #include <cstring>
3
4  int main () {
5    char str1[] = "oi gente";
6    char str2[40];
7    char str3[40];
8
9    strcpy(str2,str1);
10
11   strcpy(str3, "Copia bem sucedida");
12
13   std::cout << "str1: " << str1 << "\nstr2: " << str2;
14   std::cout << "\nstr3: " << str3 << "\n";
15   std::cout << "Tamanho de str1: " << strlen(str1);
16
17   return 0;
18 }
```

- Para outras funções da biblioteca cstring, consultar: http://www.cplusplus.com/reference/cstring/

Exercícios

Exercícios

(5) Implemente a seguinte função usando ponteiros:

`char *strcpy(char *destino, char *origem)`

Essa função copia a string `origem` em `destino`. Ela também supõe que o tamanho de `destino` é maior ou igual ao de `origem`. O valor retornado é `destino`.

(6) Implemente a seguinte função usando ponteiros:

`int strcmp(char *str1, char *str2)`

Essa função retorna 0 se `str1 == str2`, retorna -1 se `str1 < str2` e retorna 1 de `str1 > str2`.

Tipo de dado std::string

O tipo de dado `std::string`

- Como strings são comumente usadas em programas, o C++ oferece um tipo de dados de string como parte da biblioteca padrão, o `std::string`.
- Para usar esse tipo, primeiro precisamos incluir o cabeçalho `<string>`. Feito isso, podemos definir variáveis do tipo `std::string`.
- Exemplo:

```
1 #include <string> // prog21.cpp
2 #include <iostream>
3
4 int main() {
5     std::string myName {"Alex"};
6     std::cout << "Meu nome eh " << myName;
7
8     return 0;
9 }
```

Lendo um std::string do teclado

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #include <string> // prog22.cpp
2  #include <iostream>
3
4  int main() {
5      std::cout << "Digite seu nome completo: ";
6      std::string nome;
7      std::cin >> nome; /* isto nao funcionara como esperado
8                           pois std::cin quebra as palavras
9                           por espacos em branco */
10
11     std::cout << "Digite sua idade: ";
12     std::string idade;
13     std::cin >> idade;
14
15     std::cout << "Seu nome eh " << nome << " e sua idade eh "
       << idade;
16
17     return 0;
18 }
```

- `std::cin` não é recomendado para ler palavras com espaços em branco.

Lendo um `std::string` do teclado (2)

- A fim de ler uma string completa (incluindo os espaços em branco), é melhor usar a função `std::getline()`, que recebe dois parâmetros: o primeiro é `std::cin` e o segundo é uma variável do tipo `std::string`.

```
1  #include <string> // prog23.cpp
2  #include <iostream>
3
4  int main() {
5      std::cout << "Digite seu nome completo: ";
6      std::string nome;
7      std::getline(std::cin, nome); /* le caracteres e os
8                                       armazena na variavel
9                                       nome ate que seja
10                                      digitado o caractere '\n' */
11
12     std::cout << "Seu nome eh " << nome;
13
14     return 0;
15 }
```

Misturando std::cin e std::getline


```
1  #include <string> // prog24.cpp
2  #include <iostream>
3
4  int main() {
5      std::cout << "Escolha 1 ou 2: ";
6      int escolha { 0 };
7      std::cin >> escolha;
8
9      std::cout << "Digite seu nome: ";
10     std::string nome;
11     std::getline(std::cin, nome); // Problema: nao sera lido
12
13     std::cout << "Ola, " << nome;
14     std::cout << ", voce escolheu " << escolha << '\n';
15
16     return 0;
17 }
```

- Após ler um valor com `std::cin`, o caractere de nova linha `'\n'` permanece no fluxo de entrada.
- Assim, ao chamar a função `std::getline` na sequência, ela vê que `'\n'` está no fluxo, e deduz que inserimos uma string vazia!

Misturando std::cin e std::getline

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```cpp
#include <string> // prog25.cpp
#include <limits>
#include <iostream>

int main() {
  std::cout << "Escolha 1 ou 2: ";
    int escolha { 0 };
    std::cin >> escolha;

    // Ignora o primeiro caractere armazenado no buffer.
    std::cin.ignore();

    std::cout << "Digite seu nome: ";
    std::string nome;
    std::getline(std::cin, nome); // OK agora!

    std::cout << "Ola, " << nome;
    std::cout << ", voce escolheu " << escolha << '\n';

  return 0;
}
```

Concatenando std::string — Exemplo

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```cpp
#include <string> // prog26.cpp
#include <iostream>

int main() {
    std::string a("45");
    std::string b("11");

    std::cout << a + b << "\n"; // a, b serao concatenadas
    a += " volts";
    std::cout << a << "\n";

    // todo objeto do tipo std::string tem um metodo length()
    // que retorna o numero de caracteres da string
    std::cout << "Tamanho da string \'" << a << "\' = ";
    std::cout << a.length();

    return 0;
}
```

Accesando elementos em um `std::string`


```
1  #include <iostream> // prog27.cpp
2  #include <string>
3  using namespace std;
4
5  int main () {
6    string str("abcdefg");
7    cout << str[5] << endl; // Usando o operador [ ]
8    str[5] = 'X';
9    cout << str << endl;
10
11   string str2("mnopqrstuv");
12   cout << str2.at(5) << endl; // usando o metodo at()
13   str2.at(5) = 'X';
14   cout << str2 << endl;
15
16   return 0;
17 }
```

- O método `at()` é mais lento que o operador `[]`, pois usa exceções para verificar se o índice passado como parâmetro é válido.

Estruturas (structs)

Tipos de dados definidos pelo programador

A linguagem C++ permite criar novos tipos de dados a partir dos tipos básicos. Para criar um novo tipo de dado, um dos seguintes comandos pode ser utilizado:

- Estruturas: comando `struct`
- Enumerações: comando `enum`
- Renomear um tipo existente: comando `typedef`

Estruturas: `struct`

- Uma estrutura pode ser vista como um conjunto de variáveis sob o mesmo nome, e cada uma delas pode ter qualquer tipo válido.

- Definindo uma estrutura:

```
1 struct Empregado {
2     short id;
3     int idade;
4     double salario;
5 };
```

Exemplo — Instanciando e inicializando um struct

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #include <iostream> // prog08.cpp
2
3  struct Empregado {
4      short id;
5      int idade;
6      double salario;
7  };
8
9  void print(Empregado e) {
10   std::cout << "Id: " << e.id << ", idade: " <<
11   e.idade << ", Salario: " << e.salario << "\n";
12 }
13
14 int main () {
15   Empregado carlos;
16   carlos.id = 10;
17   carlos.idade = 23;
18   carlos.salario = 985.98;
19   print(carlos);
20
21   Empregado maria {11, 34, 2340.98}; // a partir do C++11
22   print(maria);
23   return 0;
24 }
```

Exemplo — Instanciando e inicializando um `struct`


```
1  #include <iostream> // prog09.cpp
2
3  struct Point3d {
4      double x;
5      double y;
6      double z;
7  };
8
9  Point3d getZeroPoint() {
10     return Point3d { 0.0, 0.0, 0.0 };
11 }
12
13 int main() {
14     Point3d zero = getZeroPoint();
15
16     if (zero.x == 0.0 && zero.y == 0.0 && zero.z == 0.0)
17         std::cout << "O ponto eh zero\n";
18     else
19         std::cout << "O ponto nao eh zero\n";
20
21     return 0;
22 }
```

Exercícios

- Crie uma estrutura para representar as coordenadas de um ponto no plano (posições X e Y). Em seguida, declare e leia do teclado um ponto e exiba a distância dele até a origem das coordenadas, isto é, a posição (0,0).

  Dica: A distância entre dois pontos $P(x_p, y_p)$ e $Q(x_q, y_q)$, é dada pela fórmula $dist(P, Q) = \sqrt{(x_q - x_p)^2 + (y_q - y_p)^2}$.

- Crie uma estrutura chamada `Retangulo`. Essa estrutura deverá conter o ponto superior esquerdo e o ponto inferior direito do retângulo. Cada ponto é definido por uma estrutura `Ponto`, a qual contém as posições X e Y. Faça um programa que declare e leia uma estrutura Retangulo e exiba a área, o comprimento da diagonal e o perímetro desse retângulo.

- Usando a estrutura Retângulo do exercício anterior, faça um programa que declare e leia uma estrutura Retângulo e um Ponto, e informe se esse ponto está ou não dentro do retângulo.

Exercícios — Estruturas

- Crie uma estrutura representando um aluno de uma disciplina. Essa estrutura deve conter o número de matrícula do aluno, seu nome e as notas de três provas. Agora, escreva um programa que leia os dados de cinco alunos e os armazena nessa estrutura. Em seguida, exiba o nome e as notas do aluno que possui a maior média geral dentre os cinco.

Enumerações

- Uma enumeração é um tipo definido pelo usuário que consiste em um conjunto de constantes integrais nomeadas, denominadas enumeradores.
- Enumerações são definidas por meio da palavra-chave **enum**.
- Exemplo:

```
1  // Define uma nova enumeracao chamada Cor
2  enum Cor {
3      /* Abaixo seguem as constantes.
4       * Elas definem todos os valores que esse tipo pode
5       * armazenar. Cada constante eh separada por virgula */
6      BLACK, RED, BLUE, GREEN, WHITE, CYAN, YELLOW, MAGENTA
7  }; // declaracao termina com ponto-e-virgula
8
9  // Define algumas variaveis do tipo enumeracao Cor
10 Cor papel = WHITE;
11 Cor casa (BLUE);
12 Cor bandeira { RED };
```

Exemplo — Enumerações

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```cpp
#include <iostream> // prog12.cpp
#include <string>
using namespace std;

enum semana {Domingo, Segunda, Terca,
             Quarta, Quinta, Sexta, Sabado};


int main() {
  string array[7] {"Domingo", "Segunda", "Terca", "Quarta",
                   "Quinta", "Sexta", "Sabado"};
  enum semana s1, s2, s3;
  s1 = Segunda;
  s2 = Terca;
  s3 = (semana) (s1+s2);

  std::cout << "Domingo = " << array[Domingo] << "\n";
  std::cout << "s1 = " << array[s1] << "\n";
  std::cout << "s2 = " << array[s2] << "\n";
  std::cout << "s3 = " << array[s3] << "\n";

  return 0;
}
```

Enumerações — Resultados indesejados

O uso clássico de enumerações pode gerar resultados indesejados:

```
1  #include <iostream> // prog13.cpp
2
3  int main() {
4    // RED e Color estao no mesmo escopo
5    enum Color { RED, BLUE };
6    // BANANA e Fruit estao no mesmo escopo
7    enum Fruit { BANANA, APPLE };
8
9    // Nenhum prefixo eh necessario para acessar constante 'RED'
10   Color cor = RED;
11   // Nenhum prefixo eh necessario para acessar 'BANANA'
12   Fruit fruta = BANANA;
13
14   // O compilador compara a e b como inteiros e
15   // descobre que sao iguais
16   if (cor == fruta)
17     std::cout << "cor e fruta sao iguais\n";
18   else
19     std::cout << "cor e fruta sao diferentes\n";
20
21   return 0;
22 }
```

Outro conceito de enumeração: `enum class`

- Conversões implícitas de enumeradores em inteiros podem levar a efeitos colaterais não intencionais.

- Para ajudar a eliminar erros de programação associados aos enums sem escopo, o C++11 fornece enum class (enumeração com escopo), que torna os enumeradores fortemente tipados.

- Os enumeradores com escopo devem ser qualificados pelo nome do tipo enum (identificador) e não podem ser convertidos implicitamente. Exemplo:

```
1 enum class Fruta {MELANCIA, JACA, ACEROLA};
2
3 Fruta f1 = Fruta::ACEROLA;
4 Fruta f2 = Fruta::JACA;
```

Exemplo — enum class

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #include <iostream> // prog14.cpp
2
3  int main() {
4      enum class Color { RED, BLUE };
5      enum class Fruit { BANANA, APPLE };
6
7      // RED e BANANA nao sao mais acessiveis diretamente.
8      // Temos que usar Color::RED e Fruit::BANANA.
9      Color cor = Color::RED;
10     Fruit fruta = Fruit::BANANA;
11
12     // Erro de compilacao. O compilador nao sabe como comparar
13     // os tipos diferentes Color e Fruit
14     if (cor == fruta)
15         std::cout << "cor e fruta sao iguais\n";
16     else
17         std::cout << "cor e fruta sao diferentes\n";
18
19     return 0;
20 }
```

Comando `typedef`

- A linguagem C++ permite que o programador renomeie um tipo de dado existente e passe a usar esse novo nome como sinônimo.
- Para isso, utiliza-se o comando `typedef`.
- Exemplo:

```
typedef  int  Numero;
```

  O comando `typedef` cria um sinônimo (`Numero`) para o tipo `int`. Esse novo nome se torna equivalente ao tipo já existente.

Exemplo — usando o comando typedef

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #include <iostream> // prog10.cpp
2
3  typedef int Numero;
4
5  Numero dobro(Numero x) {
6    return x*x;
7  }
8
9  int main() {
10   Numero y;
11   std::cout << "Digite um numero: ";
12   std::cin >> y;
13
14   std::cout << "Dobro = " << dobro(y) << std::endl;
15
16   return 0;
17 }
```

Ponteiros

Ponteiros

- Variável: é um espaço reservado de memória usado para guardar um valor que pode ser modificado pelo programa.
  - Toda variável tem um endereço na memória

- Ponteiro: é uma variável usada para guardar um endereço de memória.

Declarando ponteiros

- Em C++, a declaração de um ponteiro pelo programador segue esta forma:

```
tipo_de_dado * nome_do_ponteiro;
```

- É o operador asterisco (*) que informa ao compilador que a variável `nome_do_ponteiro` não vai guardar um valor, mas um endereço de memória para o tipo especificado.

- Exemplo:

```
1 int *p_int; // p_int eh um ponteiro para int
2 double *p_d; // p_d eh um ponteiro para double
```

- Quando declaramos um ponteiro, informamos ao compilador para que tipo de variável poderemos apontá-lo.

Acessando endereços de memória

- Para saber o endereço onde uma variável está guardada na memória, usa-se o operador de endereçamento, &, na frente do nome da variável.

```
1  #include <iostream>
2
3  int main() {
4      int x = 5;
5      std::cout << x << '\n'; // imprime 5
6
7      // imprime o endereco de memoria da variavel x
8      std::cout << &x << '\n';
9
10     return 0;
11 }
```

- Ao se trabalhar com ponteiros, duas tarefas básicas serão sempre executadas:
  - Acessar o endereço de memória de uma variável, usando o operador &
  - Acessar o conteúdo de um endereço de memória, usando o operador *

Manipulando ponteiros — Exemplo


```cpp
#include <iostream> // prog35.cpp
using namespace std;

int main() {
  // Declara uma variavel int contendo o valor 10
  int count = 10;

  // Declara um ponteiro para int e atribui ao ponteiro
  // o endereco da variavel int
  int *p;
  p = &count;
  cout << "Conteudo de p: " << p << "\n";
  //Imprime 10
  cout << "Conteudo apontado por p: " << *p << "\n";

  // Atribui um novo valor a posicao de memoria apontada por p
  *p = 12;

  // As duas linhas abaixo imprimem o numero 12 na tela
  cout << "Conteudo apontado por p: " << *p << "\n";
  cout << "Conteudo de count: " << count << "\n";

  return 0;
}
```

Atribuição entre ponteiros

- Em geral, no C++, um ponteiro só pode receber o endereço de memória de uma variável do mesmo tipo do ponteiro.

```
1  #include <iostream> // prog37.cpp
2
3  int main() {
4    float f = 45.78;
5    int *ptr = &f; // Erro de compilacao
6    int i = 54;
7    float *f2 = &i; // Erro de compilacao
8
9    float *fptr = &f; // OK
10   int *iptr = &i; // OK
11   fptr = iptr; // Erro de compilacao
12   float *f3 = fptr; // OK
13
14   // imprime 45.78 e 54
15   std::cout << *fptr << ", " << *iptr << std::endl;
16
17   return 0;
18 }
```

Qual o tamanho de um ponteiro?

- O tamanho de um ponteiro depende da arquitetura para a qual o programa é compilado.
  - Arquitetura de 32 bits — ponteiro ocupa 32 bits (4 bytes).
  - Arquitetura de 64 bits — ponteiro ocupa 64 bits (8 bytes). Independentemente do que está sendo apontado.

Ponteiro nulo

- Além de endereços de memória, todo ponteiro pode armazenar o valor nulo.
- Valor nulo é um valor especial que significa que o ponteiro não está apontando para nada. Um ponteiro que contém um valor nulo é chamado de ponteiro nulo.
- Em C++, duas formas de tornar um ponteiro nulo consiste em atribuir-lhe a macro `NULL` ou o literal `0`.
- Uma outra forma, introduzida pelo C++11, consiste em atribuir a palavra-chave `nullptr`.

```
1 float *ptr2; // ptr2 nao foi inicializado
2 ptr2 = NULL; // ptr2 agora eh um ponteiro nulo
3 ptr2 = 0; /** OK, mas nao usaremos isso! **/
4 ptr2 = nullptr; // nullptr tambem representa valor nulo
```

Ponteiro nulo

- Atenção: Tentar acessar dados através de um ponteiro nulo é ilegal e fará com que seu programa seja encerrado.

```
1 int *ptr = nullptr;
2 std::cout << *ptr << endl; // erro: falha de segmentacao
```

- Boa prática de programação 1: Antes de usar um ponteiro, certifique-se de que ele não é um ponteiro nulo. Exemplo:

```
1 if (ptr != nullptr) {
2   // ptr NAO eh um ponteiro nulo
3 } else {
4   // ptr EH um ponteiro nulo
5 }
```

- Boa prática de programação 2: Se, no momento da declaração de um ponteiro, nenhum endereço de memória válido for atribuído ao ponteiro, então inicialize o ponteiro com um valor nulo.

Ponteiros e arrays

Ponteiros e arrays

- Ponteiros e arrays estão intrinsecamente relacionados.
- Quando um array é usado em uma expressão, ele decai (é implicitamente convertido) em um ponteiro que aponta para o primeiro elemento do vetor. Exemplo:

Ponteiros e arrays — Exemplo


```
1  #include <iostream> // prog38.cpp
2  using namespace std;
3
4  int main() {
5      int array[5] = { 9,7,5,3,1 };
6
7      // imprime o endereco do primeiro elemento de 'array'
8      cout << "Endereco de array[0]: " << &array[0] << '\n';
9
10     // imprime o valor do ponteiro para o qual 'array' decai
11     cout << "array decai para um ponteiro com endereco: ";
12     cout << array << '\n';
13
14     for(int i = 0; i < 5; ++i)
15         cout << i << " :" << array[i] << endl;
16
17     return 0;
18 }
```

Ponteiros e arrays

- Como um array, usado em uma expressão, decai para um ponteiro que aponta para o primeiro elemento do array, podemos derreferenciar o array a fim de obter o valor do primeiro elemento:

```
1  #include <iostream> // prog39.cpp
2
3  int main() {
4    int array[5] = { 9,7,5,3,1 };
5
6    // dereferenciar um array retorna o primeiro elemento
7    std::cout << *array << "\n"; // imprime 9!
8
9    // Dada essa propriedade dos arrays, podemos declarar
10   // um ponteiro do tipo int e fazer ele apontar para array
11   int *ptr = array;
12   std::cout << *ptr << "\n"; // imprime 9
13
14   return 0;
15 }
```

Passando arrays como argumentos para funções

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Em C++, arrays são sempre passados por referência.
- Ao passar um array como um argumento para uma função, ele decai em um ponteiro e o ponteiro é passado para a função:

Passando arrays como argumentos para funções

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #include <iostream> // prog42.cpp
2  using namespace std;
3
4  void printSize(int array[]) {
5      // array eh tratado como ponteiro aqui
6      // o tamanho do ponteiro sera impresso
7      std::cout << sizeof(array) << '\n';
8  }
9
10 int main() {
11     int array[] = { 1,1,2,3,5,8,13,21 };
12
13     // imprime sizeof(int) * array size que eh igual a 32
14     std::cout << sizeof(array) << '\n';
15
16     printSize(array); // array decai para um ponteiro aqui
17
18     return 0;
19 }
```

Passando arrays como argumentos para funções


```cpp
#include <iostream> // prog43.cpp

// ptr contem uma copia do endereco passado como parametro
void modifica_array(int ptr[]) {
    *ptr = 5; // o que faz essa linha?
}

int main() {
    int vec[] = { 1,4,2,3,7,8,13,21 };
    modifica_array(vec);
    int size = sizeof(vec) / sizeof(vec[0]);
    for(int i = 0; i < size; i++)
        std::cout << "vec[" << i << "] = " << vec[i] << '\n';
    return 0;
}
```

- Quando `modifica_array()` é chamado, `vec` decai em um ponteiro e o valor desse ponteiro é copiado no parâmetro `ptr`.
- Embora o valor em `ptr` seja uma cópia do endereço de `vec`, `ptr` ainda aponta para o vetor real. Assim, quando `ptr` é desreferenciado, o vetor é desreferenciado.

Aritmética de ponteiros

- Apenas duas operações aritméticas podem ser utilizadas nos endereços armazenados pelos ponteiros: adição e subtração.
- Se `ptr` apontar para um inteiro, `ptr+1` é o endereço do próximo inteiro na memória após o `ptr`.
  E `ptr-1` é o endereço do inteiro antes de `ptr`.

```
1  #include <iostream> // prog44.cpp
2
3  int main() {
4      int array[5] = { 0,1,2,3,4 };
5      int *ptr = array; // ptr aponta para o primeiro
6                        // elemento do array
7
8      std::cout << ptr << ": " << *ptr << '\n';
9      std::cout << ptr+1 << ": "<< *(ptr+1) << '\n';
10     std::cout << ptr+2 << ": " << *(ptr+2) << '\n';
11     std::cout << ptr+3-2 << ": " << *(ptr+3-2) << '\n';
12
13     return 0;
14 }
```

Ponteiros e indexação de arrays

- Quando o compilador vê o operador de indexação [ ], ele o traduz em uma adição de ponteiro seguida de derreferência.
  - ou seja, `array[n]` é o mesmo que `*(array+n)`, onde `n` é um inteiro.

```
1  #include <iostream> // prog47.cpp
2
3  int main() {
4       int array [5] = { 9,7,5,3,1 };
5
6       // imprime o endereco do elemento array[1]
7       std::cout << &array[1] << '\n';
8       // imprime o endereco do ponteiro (array+1)
9       std::cout << array+1 << '\n';
10
11      std::cout << array[1] << '\n';    // imprime 7
12      std::cout << *(array+1) << '\n'; // imprime 7
13
14     return 0;
15 }
```

Ponteiros e structs

Ponteiros e estruturas

- Ao criarmos uma variável de um tipo `struct`, esta é armazenada na memória como qualquer outra variável, e portanto possui um endereço.

- É possível então criar um ponteiro para uma variável de um tipo `struct`.

Ponteiros e estruturas

- Para acessarmos os campos de uma variável `struct` via um ponteiro, podemos utilizar o operador **(*)** juntamente com o operador **(.)** como de costume:

```
1 Ponto p1, *p3;
2 p3 = &p1;
3 (*p3).x = 1.5;
4 (*p3).y = 1.5;
```

- Em C também podemos usar o operador (->) para acessar campos de uma estrutura via um ponteiro.

```
1 Ponto p1, *p3;
2 p3 = &p1;
3 p3->x = 1.5;
4 p3->y = 1.5;
```

- Para acessar campos de estruturas via ponteiros use um dos dois:
  - `ponteiroEstrutura->campo`
  - `(*ponteiroEstrutura).campo`

O que será impresso pelo programa abaixo?


```
1  #include <iostream> // prog98.c
2
3  struct Ponto {
4    double x;
5    double y;
6  };
7
8  int main() {
9    Ponto p1, p2, *p3, *p4;
10   p3 = &p1;
11   p4 = &p2;
12
13   p1.x = 1; p1.y = 2;
14   p2.x = 3; p2.y = 4;
15
16   (*p3).x = 2.5;
17   (*p3).y = 2.5;
18
19   p4->x = 4.5;
20   p4->y = 4.5;
21
22   std::cout << "p1 = (" << p1.x << "," << p1.y << ")\n";
23   std::cout << "p1 = (" << p2.x << "," << p2.y << ")\n";
24 }
```

Exercício

Um ponto no plano catersiano é definido pela sua coordenada `x` e sua coordenada `y`. Seja `Ponto` um struct com dois campos `x` e `y`, do tipo `float`.

- Implemente uma função que recebe dois valores do tipo Ponto como argumento e troca os valores dos pontos. Sua função deve obedecer o protótipo:
  ```
  void troca(Ponto *p1, Ponto *p2);
  ```
- Implemente uma função que recebe um valor do tipo Ponto como argumento e dobra os valores das suas coordenadas.

Exemplo 1 — Ponteiros e Estruturas


```
1  #include <iostream> // prog99.c
2
3  struct Ponto {
4    float x;
5    float y;
6  };
7
8  void troca(Ponto *p1, Ponto *p2) {
9    Ponto aux;
10   aux = *p1;
11   *p1 = *p2;
12   *p2 = aux;
13 }
14
15 int main() {
16   Ponto a = {2, 3}, b = {4.5, 4.5};
17   troca(&a, &b);
18   // O que sera impresso?
19   std::cout << "a = (" << a.x << "," << a.y << ")\n";
20   // O que sera impresso?
21   std::cout << "b = (" << b.x << "," << b.y << ")\n";
22 }
```

Exemplo 2 — Ponteiros e Estruturas

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #include <iostream> // prog101.c
2
3  struct Ponto {
4    float x;
5    float y;
6  };
7
8  void dobraCoordenada(Ponto *p) {
9    (*p).x = 2 * (*p).x;
10   (*p).y = 2 * (*p).y;
11 }
12
13 int main() {
14   Ponto a = {2, 3};
15   dobraCoordenada(&a);
16   std::cout << a.x << ", " <<  a.y; // O que sera impresso?
17 }
```

Exemplo 3 — Ponteiros e Estruturas

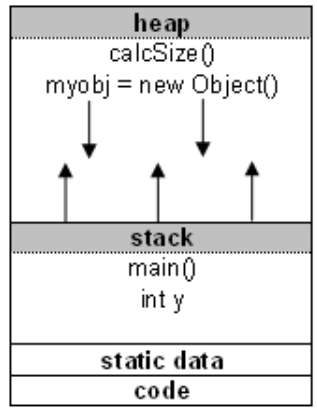
Equivalente ao exemplo anterior:

```
1  #include <iostream> // prog100.c
2
3  struct Ponto {
4    float x;
5    float y;
6  };
7
8  void dobraCoordenada(Ponto *p) {
9    p->x = 2 * p->x;
10   p->y = 2 * p->y;
11 }
12
13 int main() {
14   Ponto a = {2, 3};
15   dobraCoordenada(&a);
16
17   // O que sera impresso?
18   std::cout << "a = " << "(" << a.x << "," << a.y << ")";
19 }
```

Alocação Dinâmica de Memória

Organização da memória: Static, Stack e Heap

Os programas escritos em C++, enxergam e dividem a memória em três diferentes regiões:



Static: persiste durante toda a vida do programa e é geralmente usada para armazenar o código fonte, variáveis globais e variáveis `static`.

Stack (Pilha): usada para armazenar variáveis locais. É gerenciada automaticamente pela CPU. Quando uma função termina a execução, todas as variáveis associadas a essa função na pilha são excluídas e a memória que elas usam é liberada.

Heap: memória livre disponível que não é gerenciada automaticamente pela CPU — o programador deve alocar e desalocar explicitamente, usando funções como `new` e `delete`.

Alocação dinâmica de variáveis

- Para alocar dinamicamente uma variável, usamos o operador `new` e para desalocar, usamos o operador `delete`.
  - Como `new` retorna o endereço de memória da região alocada, devemos atribuir esse endereço a um ponteiro.

- Sintaxe (alocação de memória):
  `<tipo_de_dado> *variavel = new <tipo_de_dado>;`

- Sintaxe (liberação de memória):
  `delete variavel;`

- Exemplo:

```
int *ptr = new int;
*ptr = 5;
delete ptr;
ptr = nullptr; // evita 'dangling pointer'
```

Alocação dinâmica de arrays

- Para alocar dinamicamente um array, usamos o operador `new[]` e, para desalocar, usamos o operador `delete[]`.
- Sintaxe (alocação de memória):
  `<tipo_de_dado> *ptr = new <tipo_de_dado> [];`
- Sintaxe (liberação de memória):
  `delete[] ptr;`
- Exemplo:

```
int *ptr = new int[15];
ptr[0] = 5;
delete[] ptr;
ptr = nullptr; // evita 'dangling pointer'
```

Atividade

Escreva um programa que aloca dinâmicamente um array de inteiros de tamanho n. O valor n é entrado pelo usuário no início do programa.

(a) Escreva uma função que recebe como parâmetro o array alocado dinâmicamente e preenche esse array com valores inteiros digitados pelo usuário. Sua função deve obedecer o protótipo:
`void preencheArray(int *A, int n);`

(b) Escreva uma função que recebe como parâmetro o array alocado dinâmicamente e imprime na tela os seus elementos. Sua função deve obedecer o protótipo:
`void imprimeArray(int *A, int n);`

(c) Antes do programa acabar, libere a memória alocada dinâmicamente.

Verificar se alocação foi bem-sucedida

- Pode ser que `new` não retorne a memória solicitada. Neste caso, uma exceção será lançada e, se não for tratada, o programa terminará com um erro.
- Uma forma alternativa ao tratamento da exceção consiste em informar `new` para retornar um ponteiro nulo se a memória não puder ser alocada.
  - Isso é feito adicionando a constante `std::nothrow` entre o operador `new` e o tipo de dado:

```
1 double *ptr = new (std::nothrow) double[100000000000];
2 if (!ptr) {
3   std::cout << "Memory overflow" << std::endl;
4   exit(1);
5 }
```

- Boa prática: certifique-se de que a solicitação de memória foi bem-sucedida ante de usá-la.

Exemplo — Estouro de memória

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // prog52.cpp
2  #include <iostream>
3  using namespace std;
4
5  int main () {
6    double *ptr = new (std::nothrow) double[100000000000];
7
8    if (ptr == nullptr) {
9      cout << "Error: Memory overflow" << endl;
10   }
11   else {
12     delete[] ptr;
13     ptr = nullptr;
14   }
15
16   return 0;
17 }
```

Vazamento de memória

- Vazamentos de memória acontecem quando o programa perde o endereço de uma região de memória alocada dinamicamente antes de devolvê-lo ao sistema operacional.
  - Quando isso acontece, seu programa não pode excluir a memória alocada dinamicamente porque não sabe mais onde ela está.
  - O sistema operacional também não pode usar essa memória, pois considera que ela ainda está sendo usada pelo seu programa.
- Exemplo: que problema pode haver com esse trecho de código?

```
1 void funcao() {
2     int *ptr = new int[10];
3 }
```

Vazamento de memória

- Há outras formas de vazamento de memória. Por exemplo, um vazamento de memória pode ocorrer se um ponteiro que contém o endereço da memória alocada dinamicamente receber outro valor:

```
1 int v = 5;
2 int *ptr = new int; // aloca memoria
3 ptr = &v; // endereco antigo perdido: vazamento de memoria
```

- Também é possível obter um vazamento de memória via alocação dupla:

```
1 int *ptr = new int;
2 ptr = new int;
```

- Uma forma de evitar esse tipo de vazamento consiste em liberar a memória antes de atribuir novo valor ao ponteiro:

```
1 int *ptr = new int;
2 delete ptr;
3 ptr = new int;
```

Observações adicionais sobre arrays dinâmicos

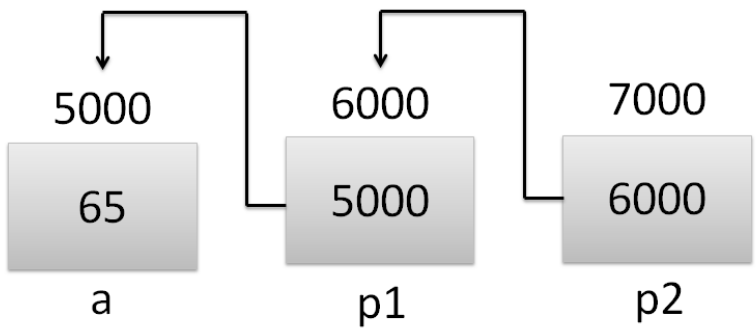
- Na expressão `int *ptr = new int[10]`, o ponteiro `ptr` aponta para o primeiro elemento do array.
  - Deste modo, `ptr` desconhece o tamanho do array e o operador `sizeof` não retorna o tamanho total da memória alocada para `ptr`.
  - Pelo mesmo motivo, não é possível usar um loop `for-each` para processar elementos de um array dinâmico.

Ponteiros para ponteiros

Ponteiros para ponteiros

- Um ponteiro para ponteiro é um ponteiro que guarda o endereço de outro ponteiro.
- Em C++, a declaração de um ponteiro para ponteiro criado pelo programador segue esta forma geral:
  `tipo_do_ponteiro **nome_do_ponteiro;`

```
1 int a = 65;
2 int *p1 = &a;
3 int **p2 = &p1;
```



Ponteiros para ponteiros

- Um ponteiro para um ponteiro pode ser desreferenciado a fim de recuperar o valor apontado. Como esse valor é, ele próprio, um ponteiro, é possível desreferenciá-lo novamente para chegar ao valor subjacente:

```
1 int value = 5;
2
3 int *ptr = &value;
4 std::cout << *ptr; // imprime 5
5
6 int **ptrptr = &ptr;
7 std::cout << **ptrptr; // imprime 5
```

Arrays de ponteiros

- Um uso comum de ponteiros para ponteiros consiste em alocar dinamicamente um array de ponteiros:

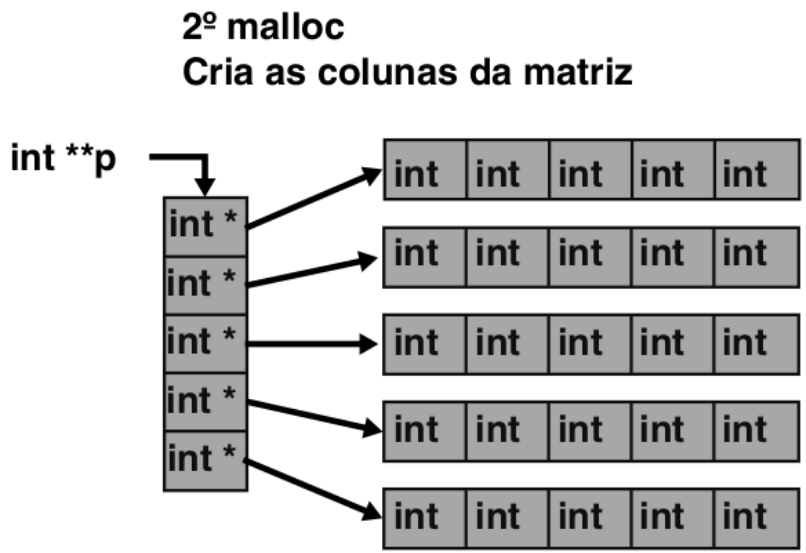
```
1 // aloca um array de ponteiros para inteiros de tamanho 10
2 int **array = new int*[10];
```

- Ponteiros para ponteiros também são usados para facilitar a alocação dinâmica de arrays multidimensionais.
  - Primeiro, alocamos um array de ponteiros (como acima).
  - Depois, percorremos o array de ponteiros e alocamos um array dinâmico para cada elemento.

```
1 // aloca um array de ponteiros para inteiros de tamanho 3
2 // essas sao as linhas (3 linhas)
3 int **array = new int*[3];
4
5 // para cada elemento de array, aloca um array de tamanho 4
6 // estas sao as colunas
7 for (int count = 0; count < 3; ++count)
8     array[count] = new int[4];
```

Alocação dinâmica de arrays bidimensionais

- Exercício: Escreva um programa para criar uma matriz de inteiros dinamicamente usando ponteiro para ponteiro.
- Basicamente, para alocar uma matriz utiliza-se um ponteiro com dois níveis.



- Em um ponteiro para ponteiro, cada nível do ponteiro permite criar uma nova dimensão no array.

Arrays bidimensionais – Exemplo 1


```
1  #include <iostream> //prog53.cpp
2
3  int main() {
4    int **array = new int*[3];
5    for (int count = 0; count < 3; ++count) {
6      // aloca colunas e inicializa com zeros
7      array[count] = new int[4]{};
8    }
9
10   for (int i = 0; i < 3; i++) { // imprime matriz
11     for (int j = 0; j < 4; j++)
12       std::cout << array[i][j] << " ";
13     std::cout << '\n';
14   }
15
16   for (int i = 0; i < 3; i++) // liberando a matriz
17     delete[] array[i];
18   delete[] array;
19
20   return 0;
21 }
```

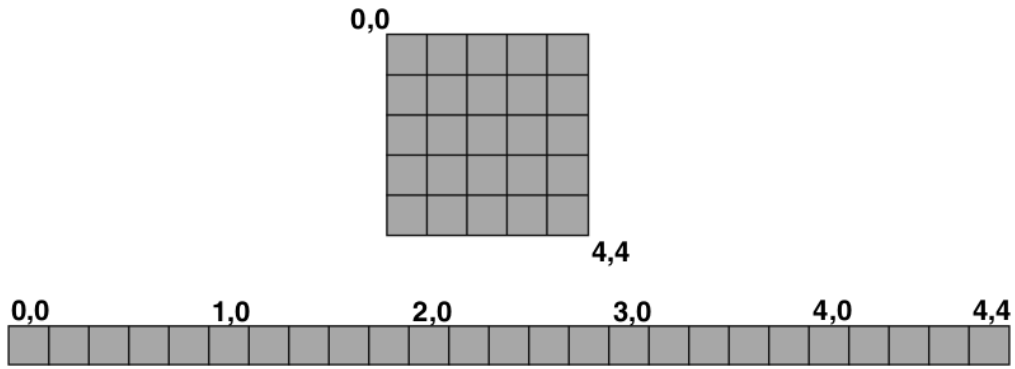
Arrays bidimensionais – Exemplo 2


```
1  #include <iostream> //prog54.cpp
2
3  void imprime_matriz(int **M, int lin, int col) {
4    for (int i = 0; i < lin; i++) { // imprime matriz
5      for (int j = 0; j < col; j++)
6        std::cout << M[i][j] << " ";
7      std::cout << '\n';
8    }
9  }
10
11 int main() {
12   int **array = new int*[3];
13   for (int count = 0; count < 3; ++count) {
14     // aloca colunas e inicializa com zeros
15     array[count] = new int[4]();
16   }
17
18   imprime_matriz(array, 3, 4);
19
20   for (int i = 0; i < 3; i++) // liberando a matriz
21     delete[] array[i];
22   delete[] array;
23
24     return 0;
25 }
```

Ponteiros e Matrizes

Ponteiros e arrays multidimensionais

- Arrays multidimensionais são armazenados linearmente na memória.
- Por exemplo, a matriz `int mat[5][5];` apesar de ser bidimensional, é armazenada como um simples array na memória:



- Podemos acessar os elementos de um array multidimensional usando a notação tradicional de colchetes `mat[i][j]` ou a notação por ponteiros: `*(mat+(i*COL)+j)`, onde `COL` é o número de colunas da matriz `mat`.

Exemplo — Ponteiros e arrays multidimensionais

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```cpp
#include <iostream> // prog48.cpp

void imprime_matriz(int *M, int linha, int coluna) {
  for (int i = 0; i < linha; i++) {
    for (int j = 0; j < coluna; j++) {
      std::cout << M[(i*coluna)+j] << " ";
    }
    std::cout << '\n';
  }
}

int main() {
  constexpr int LIN = 3, COL = 4;

  int matriz[LIN][COL] = {{1,2,3,4},{5,6,7,8},{9,0,1,2}};

  imprime_matriz( &matriz[0][0], LIN, COL );

  return 0;
}
```

Aplicações de ponteiros

Aplicações de ponteiros

- Passar argumentos por referência. A passagem por referência serve a dois propósitos:
  (i) Modificar o valor de uma variável externa dentro de uma função. Exemplo: trocar os valores de duas variáveis.

  (ii) Eficiência: podemos passar um dado de tamanho grande (por exemplo, um array) para uma função de forma que não envolva a cópia dos dados.

- Acessar elementos de um array. O compilador usa internamente ponteiros para acessar os elementos de um array.

- Permitir que uma função "retorne" vários valores.

Aplicações de ponteiros

- Programar no nível do sistema, onde os endereços de memória são úteis.
- Alocação dinâmica de memória: podemos usar ponteiros para alocar memória dinamicamente. A vantagem da memória alocada dinamicamente é que ela não é excluída até que seja excluída explicitamente.
- Implementar diversas estruturas de dados. Eles serão usados na implementação eficiente de diversas estruturas de dados que veremos durante o curso.

Exercícios

Exercícios

(1) Escreva uma função `troca` que receba como entrada duas variáveis inteiras e troque os seus valores. Escreva também uma função `main` que use a função `troca`.

(2) Escreva uma função `mm` que receba um vetor inteiro A com n elementos e os endereços de duas variáveis inteiras, digamos `min` e `max`, e deposite nestas variáveis o valor de um elemento mínimo e o valor de um elemento máximo do vetor. Escreva também uma função `main` que use a função `mm`.

(3) Escreva um programa que leia um inteiro n seguido de n números inteiros e imprima esses n números em ordem invertida (primeiro o último, depois o penúltimo, etc.) O seu programa não deve impor quaisquer restrições ao valor de n.

Exercícios

(4) Faça uma função `MAX` que recebe como entrada um inteiro n, uma matriz inteira $A_{n \times n}$ e devolve três inteiros: k, l e c, tal que

- k é o maior elemento de A e é igual a $A[l][c]$.

Se o elemento máximo ocorrer mais de uma vez, indique em l e c qualquer uma das possíveis posições. Use ponteiros para os argumentos.

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM