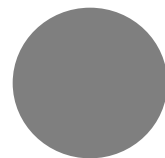
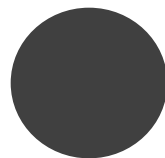
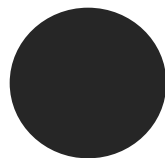


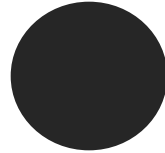
C++

ESTUDO DE CASO

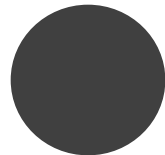


Eduarda Elger, Ellen Bonafin , Gustavo Pauli, Heloisa
Alves e Pablo Huguen
2022

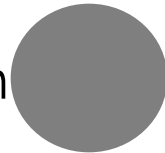
Histórico



Objetivos, Contextualização e Características



Tour pela linguagem



Referências



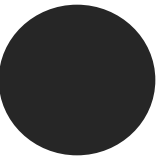
ESTUDO DE CASO

- No ano de 1979, em sua tese de PhD, Bjarne Stroustup trabalhou muito com a linguagem de programação *Simula 67*, uma das primeiras a suportar o paradigma orientado a objetos. Bjarne notou que esse paradigma era muito útil no desenvolvimento de software, porém a linguagem *simula* era muito lenta para uso prático.

Apos isso, Bjarne começou a trabalhar no *C com Classes*, um *superset* da linguagem C que tentava adicionar funcionalidades de linguagens orientada a objetos na linguagem C, que é conhecida pela sua performance e utilidade em programação de baixo nível

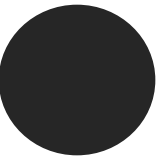
- O primeiro compilador do *C com classes* foi chamado de *Cfront*, que também foi escrito em *C com classes*. Porém, o *C com Classes* foi abandonado em 1993 após se tornar dificilmente extensível.





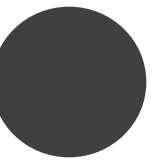
- Em 1983, o nome da linguagem foi mudado para C++. Varias features foram adicionadas junto a essa mudança, como *funções virtuais*, *sobrecarga de funções*, referencias a variáveis com o símbolo & e a palavra reservada const.
- Em 1985, Stroustup publicou o livro referencia para a linguagem c++, chamado de *The C++ Programming Language*.
- Em 1998, o comite de padronizacao do C++ publicou o primeiro padrao inbternacional da linguagem, conhecido como o padrao C++98. Em 2003 esse padrao foi revisado, e foram adiconadas mais features na linguagem, mais notavelmente a Standard Template Library (STL), essa versao foi chamada de C++03.



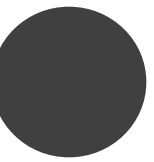


- Na metade de 2011, o padrão C++11 foi finalizado. A Biblioteca *Boost* teve um impacto considerável nas features adicionadas nesse padrão, com até alguns módulos sendo adicionados diretamente da boost. Algumas das features adicionadas nesse padrão foram o suporte a expressões regulares, uma biblioteca de randomização, uma nova biblioteca para trabalhar com tempo, suporte a operações atômicas, uma biblioteca de multi threading, uma nova sintaxe de loops for (igual o foreach), a palavra chave auto, novas classes de container, além de muitas outras.
- Atualmente, o padrão vigente é o C++20. Algumas das features de padrão são o operador de comparação em 3 vias (<=>), melhorias na utilização de funções lambda, coroutines, módulos e muitas outras.

Existem planos para um padrão futuro chamado C++23, com muitas mudanças planejadas.

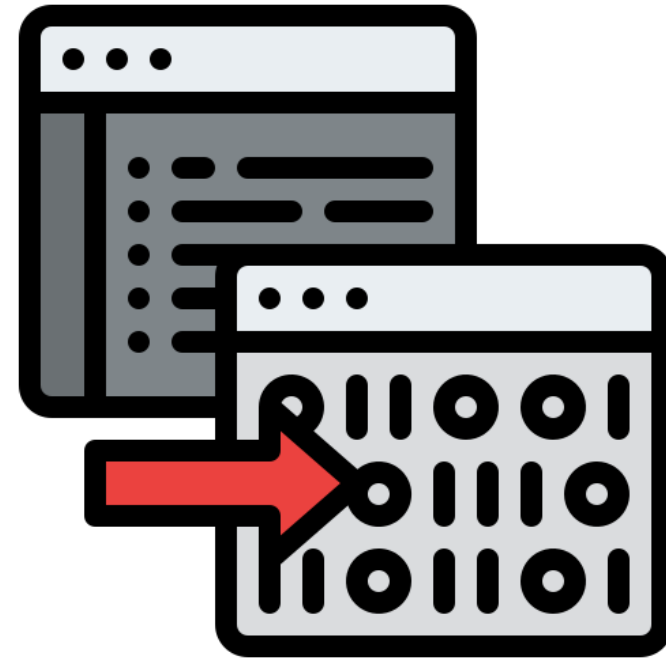


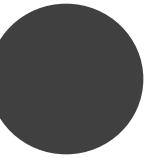
- O C++ é uma linguagem de **alto nível**, com ferramentas para se trabalhar em baixo nível também, compatível com o C e com uma extensível biblioteca tanto padrão quanto da comunidade.
- Até certo nível o C++ é portátil, pois existem diversos compiladores para várias arquiteturas que fazem com que código C++ seja compilado e rodado em várias arquiteturas com pouca ou nenhuma modificação.



Compilação

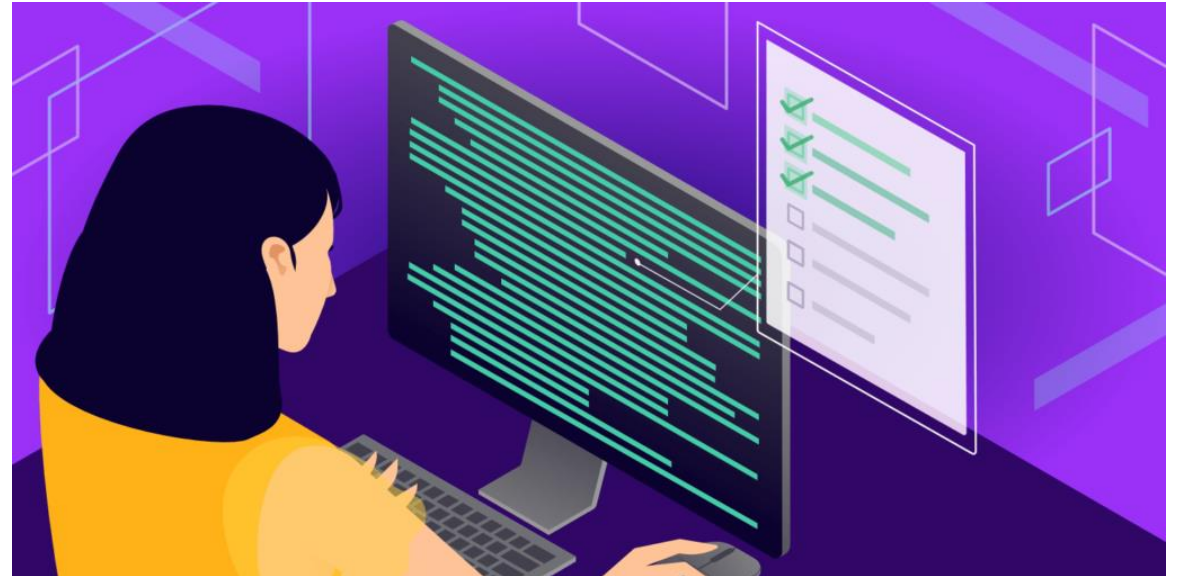
- A linguagem C++ é compilada, ou seja, o compilador traduz código escrito em C++ para o código de máquina da arquitetura alvo. Essa compilação é feita de uma só vez. Além disso, o compilador pode aplicar otimizações de código, resultando em um código de máquina mais rápido e/ou confiável.

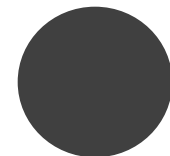




Nível de Abstração

- A linguagem C++ é uma linguagem de programação de alto nível, ou seja, oferece uma série de abstrações para facilitar o entendimento para humanos, como funções e objetos. Porém, com o C++ também é possível realizar manipulações de baixo nível, inclusive sendo uma ótima linguagem para isso..





Sistema de Tipos

- **Forte ou Fraca**

C++ é fortemente tipada, ou seja, possui bastante restrições quanto a conversão de tipos entre variáveis. Por exemplo, não é possível a conversão entre um `int` e um objeto `Fruit`.

- **Inferência**

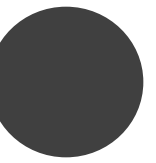
O C++ suporta tanto a inferência implícita, que é baseada no contexto que aquela variável é usada, quanto a inferência explícita, na qual o programador diz o tipo da variável.

- **Checagem**

Novamente, o C++ suporta tanto a checagem estática quanto a checagem dinâmica, ou seja, os seus tipos são checados em tempo de compilação e também em tempo de execução.

- **Segurança de tipos**

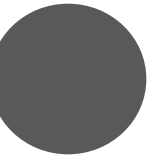
A linguagem C++ não é `type unsafe`, ou seja, ela leva em consideração que o programador sabe o que está fazendo e permite operações de conversão de tipos que podem levar a erros em tempo de execução.



Paradigmas

- A linguagem C++ é multi paradigma, ou seja, suporta diversos paradigmas e técnicas no mesmo programa. Alguns dos paradigmas que o C++ suporta são: Procedural, genérico, orientado a objetos e mais recentemente o funcional

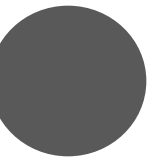




Compiladores

- Atualmente, os dois compiladores mais maduros de C++ são o GCC e o Clang (frontend do llvm).
- Para compilar e rodar um programa em C++ podemos rodar no terminal o comando:

```
clang++ -std=c++11 -stdlib=libc++ hello.cpp -o hello  
./hello
```

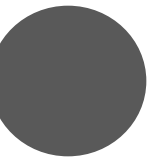


Estrutura de um Programa

- Segue um programa simples em C++ na estrutura abaixo:

```
// Headers e modulos
#include <iostream>

// Funcao principal: Retorno e corpo
int main() {
    // Expressao simples
    std::cout << "Hello, World!" << std::endl;
    // Retorno
    return 0;
}
```

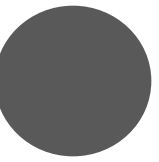


Keywords, Tipos e Variáveis

Keywords

As seguintes expressões são palavras reservadas em C++ e portanto não podem ser usados como nome de variáveis:

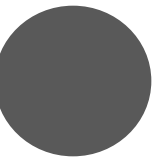
```
alignas, alignof, and, and_eq, asm, auto, bitand, bitor, bool,
break, case, catch, char, char16_t, char32_t, class, compl,
const, constexpr, const_cast, continue, decltype, default,
delete, do, double, dynamic_cast, else, enum, explicit,
export, extern, false, float, for, friend, goto, if, inline,
int, long, mutable, namespace, new, noexcept, not, not_eq,
nullptr, operator, or, or_eq, private, protected, public,
register, reinterpret_cast, 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
```



Tipos de Dados Funcionais

Tipos de dados fundamentais são tipos básicos implementados pela linguagem utilizados para representar unidades de armazenamento atômicas. No C++ temos os seguintes tipos de dados fundamentais::

Group	Type names*	Notes on size / precision
Character types	char	Exactly one byte in size. At least 8 bits.
	char16_t	Not smaller than char . At least 16 bits.
	char32_t	Not smaller than char16_t . At least 32 bits.
	wchar_t	Can represent the largest supported character set.
Integer types (signed)	signed char	Same size as char . At least 8 bits.
	<i>signed short int</i>	Not smaller than char . At least 16 bits.
	<i>signed int</i>	Not smaller than short . At least 16 bits.
	<i>signed long int</i>	Not smaller than int . At least 32 bits.
	<i>signed long long int</i>	Not smaller than long . At least 64 bits.
Integer types (unsigned)	unsigned char	(same size as their signed counterparts)
	unsigned short int	
	unsigned int	
	unsigned long int	
	unsigned long long int	
Floating-point types	float	
	double	Precision not less than float
	long double	Precision not less than double
Boolean type	bool	
Void type	void	no storage
Null pointer	decltype(nullptr)	



- A declaração de variáveis é feita da seguinte forma:

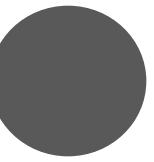
```
// variaveis
#include <iostream>

/** Namespacing
    Dividir declaracao de simbolos em "pacotes",
    para evitar conflitos de nomes.
 */
using namespace std;

int main() {
    // declaracao de variaveis com valores padroes
    int a(8), b = 12, c{21};
    // declaracao de variaveis nao inicializadas
    float result, d;

    // atribuicao de valores
    d = -48.0f;
    // operacoes aritmeticas (com type casting)
    result = float(((a - b) * c)) / d;

    cout << result;
    return 0;
}
```



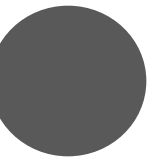
- A dedução de tipos no C++ é feita utilizando as palavras reservadas **auto** e **decltype**:

```
#include <iostream>

/* Funcao anonima (aka lambda) sendo atribuida a uma variavel
   O seu tipo de retorno e inferido pelo compilador(utilizacao do auto)
*/
auto fn = []() {
    return 42;
};

auto main() -> int {
    // inferencia de tipos em c++
    auto result = fn();
    decltype(result) result2 = result + 10;

    std::cout << result << std::endl;
    std::cout << result2 << std::endl;
    return 0;
}
```

Tipos de dados compostos

O C++ possui uma rica biblioteca de tipos de dados compostos. Um exemplo é a classe `string`, que armazena sequências de caracteres:

:

```
#include <iostream>
#include <string>

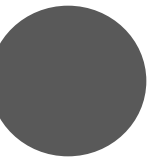
int main() {
    std::string s = "Hello, World!";
    std::cout << s << std::endl;

    // sequencia de caracteres
    for (auto c : s) {
        std::cout << c;
    }
    std::cout << std::endl;

    /* string é uma classe de dados
       Portanto possui muitos metodos
    */
    s.push_back(' ');
    s.replace(0, 5, "Hola");
    s.append("I'm a string");

    std::cout << s << std::endl;

    return 0;
}
```



Constantes

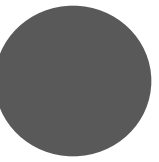
No C++ podemos definir expressões com valores fixos de 4 formas: com o pré processador, com a palavra reservada **const**, com a palavra reservada **constexpr** e com valores literais.

:

```
#include <iostream>

#define PI 3.14159265358979323846
const double E = 2.71828182845904523536;
constexpr double PHI = 1.61803398874989484820;

int main() {
    std::cout << "pi = " << PI << std::endl;
    std::cout << "e = " << E << std::endl;
    std::cout << "phi = " << PHI << std::endl;
    std::cout << "mi = " << 1.84775906502257351225f << std::endl;
    return 0;
}
```



Operadores

O C++ possui uma vasta lista de operadores, alguns deles são::

```
// Operadores aritmeticos
cout << a << " + " << b << " = " << a + b << endl;
cout << a << " - " << b << " = " << a - b << endl;
cout << a << " * " << b << " = " << a * b << endl;
cout << a << " / " << b << " = " << a / b << endl;
cout << a << " % " << b << " = " << a % b << endl;
cout << a << " ^ " << b << " = " << (a ^ b) << endl;
```

```
// Operadores de atribuicao composta
cout << "a += b = " << (a += b) << endl;
cout << "a -= b = " << (a -= b) << endl;
cout << "a *= b = " << (a *= b) << endl;
cout << "a /= b = " << (a /= b) << endl;
cout << "a %= b = " << (a %= b) << endl;
cout << "a ^= b = " << (a ^= b) << endl;
cout << "a &= b = " << (a &= b) << endl;
cout << "a |= b = " << (a |= b) << endl;
cout << "a <= b = " << (a <= b) << endl;
cout << "a >= b = " << (a >= b) << endl;
```

```
// Operadores de incremento e decremento
cout << "a++ = " << (a++) << endl;
cout << "++a = " << (++a) << endl;
```

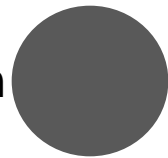
```
// operadores de comparacao
cout << "a == b = " << (a == b) << endl;
cout << "a != b = " << (a != b) << endl;
cout << "a < b = " << (a < b) << endl;
cout << "a > b = " << (a > b) << endl;
cout << "a <= b = " << (a <= b) << endl;
cout << "a >= b = " << (a >= b) << endl;
```

```
// Operadores logicos
cout << "a && b = " << (a && b) << endl;
cout << "a || b = " << (a || b) << endl;
cout << "!a = " << (!a) << endl;
```

```
// Operadores de bitwise
cout << "a & b = " << (a & b) << endl;
cout << "a | b = " << (a | b) << endl;
cout << "a ^ b = " << (a ^ b) << endl;
cout << "~a = " << (~a) << endl;
cout << "a << b = " << (a << b) << endl;
cout << "a >> b = " << (a >> b) << endl;
```

```
// Operador ternario
cout << "a ? b : c = " << (a ? b : 0) << endl;
```

```
// operadores de cast
cout << "(int)a = " << (int)a << endl;
cout << "(double)a = " << (double)a << endl;
cout << "(char)a = " << (char)a << endl;
```



- As regras de precedência de operadores são mostradas a seguir

Level	Precedence group	Operator	Description	Grouping
1	Scope	::	scope qualifier	Left-to-right
2	Postfix (unary)	++ --	postfix increment / decrement	Left-to-right
		()	functional forms	
		[]	subscript	
		. ->	member access	
3	Prefix (unary)	++ --	prefix increment / decrement	Right-to-left
		~ !	bitwise NOT / logical NOT	
		+ -	unary prefix	
		& *	reference / dereference	
		new delete	allocation / deallocation	
		sizeof	parameter pack	
		(type)	C-style type-casting	
4	Pointer-to-member	. * -> *	access pointer	Left-to-right
5	Arithmetic: scaling	* / %	multiply, divide, modulo	Left-to-right
6	Arithmetic: addition	+ -	addition, subtraction	Left-to-right
7	Bitwise shift	<< >>	shift left, shift right	Left-to-right
8	Relational	< > <= >=	comparison operators	Left-to-right
9	Equality	== !=	equality / inequality	Left-to-right
10	And	&	bitwise AND	Left-to-right
11	Exclusive or	^	bitwise XOR	Left-to-right
12	Inclusive or		bitwise OR	Left-to-right
13	Conjunction	&&	logical AND	Left-to-right
14	Disjunction		logical OR	Left-to-right
15	Assignment-level expressions	= *= /= %= += -=	assignment / compound assignment	Right-to-left
		>>= <<= &= ^= =	conditional operator	
16	Sequencing	,	comma separator	Left-to-right



Entrada e saída

A biblioteca padrão do C++ define o header `<iostream>` como padrão para operações simples de entrada e saída. Além disso, temos o header `<sstream>` que lida com operações de **streams** em **strings**:

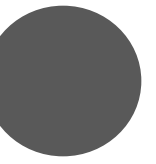
```
#include <iostream>
#include <sstream>
#include <string>

int main() {
    std::string a, b, c, str_int("123");
    int int_a;

    // io simples
    std::cout << "Digite uma palavra: ";
    std::cin >> a;
    std::cout << "Digite outra palavra: ";
    std::cin >> b;
    std::cout << a << " + " << b << " = " << a + b << std::endl;

    // ler uma linha inteira
    std::cout << "Digite uma linha: ";
    std::getline(std::cin, c);
    std::getline(std::cin, c);
    std::cout << c << std::endl;

    // stringstream
    std::cout << "converte string para inteiro: ";
    std::stringstream(str_int) >> int_a;
    std::cout << int_a << std::endl;
    return 0;
}
```



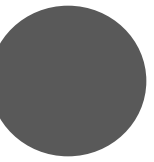
ESTUDO DE CASO

Controle de Fluxo e Loops

A linguagem possui os comandos de seleção padrão: if, elseif, else e o switch case. Além disso, possui também os loops for, while e do while. Para a utilização em loops, temos os comandos de alteração de fluxo continue, break e goto. Exemplos desses comandos podem ser vistos a seguir:

```
// if, else if, else
if (idade < 18) {
    std::cout << "Voce e menor de idade" << std::endl;
} else if (idade >= 18 && idade <= 65) {
    std::cout << "Voce e adulto" << std::endl;
} else {
    std::cout << "Voce e idoso" << std::endl;
}

// switch
switch (sexo) {
case 'M':
    std::cout << "Voce e do sexo masculino" << std::endl;
    break;
case 'F':
    std::cout << "Voce e do sexo feminino" << std::endl;
    break;
default:
    std::cout << "Voce e do sexo desconhecido" << std::endl;
    break;
}
```



```
// while
int i = 0;
while (i < 10) {
    std::cout << i << std::endl;
    i++;
}

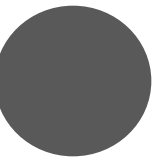
// do while
i = 0;
do {
    std::cout << i << std::endl;
    i++;
} while (i < 10);

// for
for (int j = 0; j < 10; j++) {
    std::cout << j << std::endl;
}

// for (auto)
for (auto c : nome) {
    std::cout << "[" << c << "]" << std::endl;
}
```

```
// for (auto)
for (auto c : nome) {
    std::cout << "[" << c << "]" << std::endl;
}

// jumps
for (int k = 0; k < 10; k += 2) {
    if (k == 5) {
        continue;
    } else if (k == 7) {
        break;
    } else {
        std::cout << k << std::endl;
    }
}
}
```

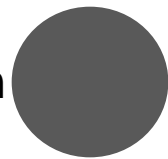


Funções

A sintaxe para definição de funções é a seguir:

```
type name ( parameter1, parameter2, ...) { statements }
```

- Aqui type é o tipo de retorno da função, name é seu nome, (**parameter1, parameter2, ...**) são os parâmetros da função (cada um com o seu tipo) e statements é o corpo da função.
- Tanto nos parâmetros quanto no tipo de retorno, podemos usar modificadores, como o const e o inline que alteram atributos desses valores e permitem ao compilador realizar algumas alterações e otimizações.
- No C++ também temos funções anônimas (aka lambda), que facilitam algumas operações e oferecem de linguagens funcionais ao C++.
- Além disso, podemos passar parâmetros por valor, onde é feita uma copiada variável, ou por referência, onde o endereço da variável é passado no lugar de seu valor.



Funções

:

```
#include <functional>
#include <iostream>

/* Funcao simples com dois parametros
   passados por valor(possui valores padroes) */
int subtraction(int a = 0, int b = 0) { return a - b; }

// Funcao sem parametros e sem retorno, tambem chamada de procedimento
void printmessage() { std::cout << "I'm a function!"; }

/* Passando valores por referencia (em C usariamos ponteiros para essas
   variaveis) */
void duplicate(int &a, int &b) {
    a *= 2;
    b *= 2;
}

/* Modificadores podem ser usados para alterar o comportamento
   * de parametros ou do retorno da funcao
   * inline = o compilador nao fara o stacking da funcao, so chamara ela
   * const = o compilador tera certeza que valor nao sera modificado
   */
inline const std::string concatenate(const std::string &a,
                                     const std::string &b) {
    return a + b;
}
```

```
auto main() -> int {
    int a = 5, b = 10;
    std::string s1 = "Hello", s2 = "World";

    /* Exemplo de recursividade, escopo e funcao anonima */
    const std::function<const int(const int)> factorial =
        [&factorial](const int n) { return n == 0 ? 1 : n * factorial(n - 1); };

    std::cout << "Subtraction of " << a << " and " << b << " is "
        << subtraction(a, b) << std::endl;

    printmessage();
    duplicate(a, b);

    std::cout << "Concatenation of " << s1 << " and " << s2 << " is "
        << concatenate(s1, s2) << std::endl;

    std::cout << "Factorial of " << a << " is " << factorial(a) << std::endl;

    return 0;
}
```

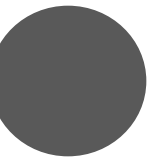


Template e Sobrecarga de Funções

- No C++, diferentes funções podem ter o mesmo nome se o tipo de dados de seus parâmetros são diferentes, ou seja, essas funções estão sobrecarregadas. Podemos usar isso para criar um polimorfismo de parâmetros para uma função.
- Outra maneira de atingir esse polimorfismo é utilizar templates de funções, onde uma função é "gerada" para um tipo específico.

A sintaxe para template functions é a seguinte::

```
template <template-parameters> function-declaration
```



Template e Sobrecarga de Funções

Um exemplo dessa propriedade pode ser visto a seguir:

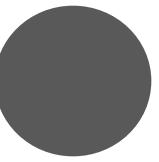
```
#include <iostream>

/* Dependendo do tipo dos parametros
   uma das funcoes sera chamada
*/
const int add(const int a = 0, const int b = 0) {
    std::cout << "int overloading add" << std::endl;
    return a + b;
}

const float add(const float a = 0, const float b = 0) {
    std::cout << "float overloading add" << std::endl;
    return a + b;
}

template <typename T> const T add(const T a, const T b) {
    std::cout << "template overloading add" << std::endl;
    const T result = a+b;
    return result;
}

int main() {
    std::cout << "int add: " << add(1, 2) << std::endl;
    std::cout << "float add: " << add(1.0f, 2.0f) << std::endl;
    std::cout << "template add: " << add<std::string>("a", "b")
               << std::endl;
}
```



Escopo e Namespaces

- No C++ temos o escopo global, escopo de bloco, escopo de funções e escopo por Namespaces.
- Namespaces permitem o agrupamento de símbolos em escopos relacionados para evitar o conflito com escopos maiores. A palavra-chave `using` introduz um símbolo no escopo atual, por exemplo, podemos inserir o nome a que pertence ao escopo `ns1` dentro de outro escopo.
- variáveis locais tem o armazenamento automático, onde a variável é desalocada quando o fluxo sai daquele escopo

```
#include <iostream>

int a = 0;
float b = 0;

namespace ns1 {
    int a = 1;
    float b = 2.0f;
} // namespace ns1

namespace ns2 {
    int a = 3;
    float b = 4.0f;
    std::string c = "4";
} // namespace ns2

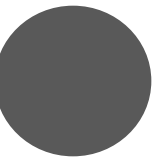
void fn(void) {
    int a = 4;
    float b = 4.0f;

    std::cout << "fn a: " << a << std::endl;
    std::cout << "fn b: " << b << std::endl;
}

void fn2(void) {
    using namespace ns2;

    std::cout << "introduced ns1 c: " << c << std::endl;
}

int main() {
    fn();
    std::cout << "global a: " << a << std::endl;
    std::cout << "global b: " << b << std::endl;
    std::cout << "ns1 a: " << ns1::a << std::endl;
    std::cout << "ns1 b: " << ns1::b << std::endl;
    std::cout << "ns2 a: " << ns2::a << std::endl;
    std::cout << "ns2 b: " << ns2::b << std::endl;
}
```



Arrays

- Variáveis locais tem o armazenamento automático, onde a variável é desalocada quando o fluxo sai daquele escopo

```
type name [elements];
```

```
#include <iostream>

// exemplo de funcao que recebem um array
int sum(int a[], int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += a[i];
    }
    return sum;
}

int main() {
    // Array de inteiros, nao inicializados
    int a[10];

    // Array de inteiros, inicializados
    int b[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // Array de inteiros, multidimensionais
    int c[2][3] = {{1, 2, 3}, {4, 5, 6}};

    for (int i = 0; i < 10; i++) {
        // escrita em uma posicao do array
        a[i] = i;
    }

    for (int i = 0; i < 10; i++) {
        std::cout << a[i] << std::endl;
    }

    std::cout << sum(a, 10) << std::endl;

    for (int i = 0; i < 10; i++) {
        std::cout << b[i] << std::endl;
    }
    return 0;
}
```



Ponteiros

- Ponteiros são referencias para posições de memória que possuem um valor armazenado, que inclusive, podem ser outros ponteiros. Usamos o caractere & para o endereço de um ponteiro e * para o valor dele. Além disso, podemos utilizar os operadores ++ e -- para realizar a aritmética de ponteiros.

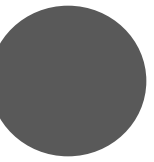
```
// exemplo de ponteiros
#include <iostream>

using namespace std;

void increment_all(int *start, int *stop) {
    int *current = start;
    while (current != stop) {
        ++(*current); // incrementa o valor do ponteiro
        ++current;    // incrementa o endereco do ponteiro
    }
}

void print_all(const int *start, const int *stop) {
    const int *current = start;
    while (current != stop) {
        cout << *current << '\n';
        ++current; // incrementa a posicao de memoria do ponteiro
    }
}

int main() {
    int numbers[] = {10, 20, 30};
    increment_all(numbers, numbers + 3);
    print_all(numbers, numbers + 3);
    return 0;
}
```



Memoria Dinâmica

- Em C, era usado as funções da família *alloc() e free() para a alocação/desalocação dinâmica de memória. No C++ foi introduzido novos operadores para a gerência de memória dinâmica na linguagem:

```
pointer = new type
pointer = new type [number_of_elements]

e

delete pointer;
delete[] pointer;
```

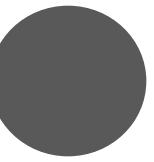
```
#include <iostream>

int main() {
    // criação de um array de inteiros utilizando o new
    int *p = new int[10];

    for (int i = 0; i < 10; i++) {
        p[i] = i;
    }

    for (int i = 0; i < 10; i++) {
        std::cout << p[i] << std::endl;
    }

    // destruição do array criado utilizando o delete
    delete[] p;
}
```



Struct

- O C++ herdou as structs do C, que são um agrupamento de dados relacionados. Porém, como C++ é uma linguagem orientada a objetos, a struct é um sinônimo de classes. A sintaxe para definir uma struct é a seguinte:

```
struct type_name {  
    member_type1 member_name1;  
    member_type2 member_name2;  
    member_type3 member_name3;  
    .  
    .  
} object_names;
```

```
// exemplo de utilizacao de structs em c++  
#include <iostream>  
  
/* structs sao definidas com a palavra reservada struct  
   Note a utilizacao da palavra chave using para a definicao do tipo  
*/  
using Point = struct {  
    int x;  
    int y;  
};  
  
int main() {  
    Point p;  
    p.x = 10;  
    p.y = 20;  
    std::cout << p.x << " " << p.y << std::endl;  
    return 0;  
}
```




Classes

- Classes no C++ são criadas utilizando a palavra-chave `class`. Classes são parecidas com structs, porém podem ter funções como membros:

```
class class_name {  
    access_specifier_1:  
        member1;  
    access_specifier_2:  
        member2;  
    ...  
} object_names;
```

```
#include <iostream>  
using namespace std;  
  
class Pessoa {  
public:  
    // Construtor  
    Pessoa(const string &nome, const int idade);  
    // Metodos  
    string getNome();  
    int getIdade();  
    void setNome(const string &nome);  
    void setIdade(const int idade);  
  
private:  
    // Atributos  
    string nome;  
    int idade;  
};  
  
// Construtor  
Pessoa::Pessoa(const string &nome, const int idade) {  
    this->nome = nome;  
    this->idade = idade;  
}
```

```
// Funcoes membros  
string Pessoa::getNome() { return this->nome; }  
  
int Pessoa::getIdade() { return this->idade; }  
  
void Pessoa::setNome(const string &nome) { this->nome = nome; }  
  
void Pessoa::setIdade(int idade) { this->idade = idade; }  
  
int main () {  
    Pessoa pessoa("Joao", 20);  
  
    cout << "Nome: " << pessoa.getNome() << endl;  
    cout << "Idade: " << pessoa.getIdade() << endl;  
  
    pessoa.setNome("Maria");  
    pessoa.setIdade(30);  
  
    cout << "Nome: " << pessoa.getNome() << endl;  
    cout << "Idade: " << pessoa.getIdade() << endl;  
  
    return 0;  
}
```



Herança e Funções Amigas

- Funções amigas de uma classe são funções especiais que podem acessar os membros privados e protegidos de uma classe:
- Existem funções amigas que seguem o mesmo conceito.

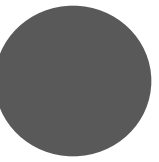
```
// Exemplo do uso de funcoes amigas
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;

public:
    Rectangle() {}
    Rectangle(int x, int y) : width(x), height(y) {}
    int area() { return width * height; }
    friend Rectangle duplicate(const Rectangle &);
};

/* Funcao amiga que retorna um objeto duplicado,
 * possui acesso ao objeto original */
Rectangle duplicate(const Rectangle &param) {
    Rectangle res;
    res.width = param.width * 2;
    res.height = param.height * 2;
    return res;
}

int main() {
    Rectangle foo;
    Rectangle bar(2, 3);
    foo = duplicate(bar);
    cout << foo.area() << '\n';
    return 0;
}
```



Herança e Funções Amigas

- Em C++ classes podem ser estendidas, herdando características e métodos da classe base. Por exemplo, as classes Rectangle e Triangle podem estender a classe Polygon
- No C++ podemos ter herança múltipla, ou seja, uma classe que herda características de duas ou mais classes base.

```
// exemplo de heranca
#include <iostream>
using namespace std;

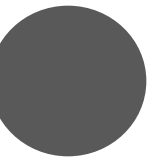
class Polygon {
protected:
    int width, height;

public:
    void set_values(int a, int b) {
        width = a;
        height = b;
    }
};

class Rectangle : public Polygon {
public:
    int area() { return width * height; }
};

class Triangle : public Polygon {
public:
    int area() { return width * height / 2; }
};

int main() {
    Rectangle rect;
    Triangle trgl;
    rect.set_values(4, 5);
    trgl.set_values(4, 5);
    cout << rect.area() << '\n';
    cout << trgl.area() << '\n';
    return 0;
}
```



Polimorfismo

- No C++, ponteiros para classes derivadas são compatíveis com o tipo ponteiro das suas classes base. Assim, Polimorfismo é a técnica que utiliza essa feature para deixar o código mais versátil

```
// exemplo de polimorfismo
#include <iostream>
using namespace std;

class Polygon {
protected:
    int width, height;

public:
    void set_values(int a, int b) {
        width = a;
        height = b;
    }
};

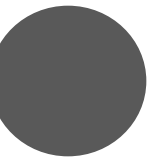
class Rectangle : public Polygon {
public:
    int area() { return width * height; }
};

class Triangle : public Polygon {
public:
    int area() { return width * height / 2; }
};

int main() {
    Rectangle rect;
    Triangle trgl;
    Polygon *ppoly1 = &rect;
    Polygon *ppoly2 = &trgl;

    ppoly1->set_values(4, 5);
    ppoly2->set_values(4, 5);

    cout << rect.area() << '\n';
    cout << trgl.area() << '\n';
    return 0;
}
```



Métodos Virtuais e Classes Abstratas

- Métodos virtuais são métodos de classes base que devem ser implementados nas classes derivadas. Já classes abstratas são classes que servem somente como base para a criação de outras, ou seja, todos os seus métodos são virtuais, e as classes que derivam dela devem implementar esses métodos.

```
// abstract base class example
#include <iostream>
using namespace std;

class Polygon {
protected:
    int width, height;

public:
    void set_values(int a, int b) {
        width = a;
        height = b;
    }
    virtual int area(void) = 0;
};

class Rectangle : public Polygon {
public:
    int area(void) { return (width * height); }
};

class Triangle : public Polygon {
public:
    int area(void) { return (width * height / 2); }
};

int main() {
    Rectangle rect;
    Triangle trgl;
    Polygon *ppoly1 = &rect;
    Polygon *ppoly2 = &trgl;
    ppoly1->set_values(4, 5);
    ppoly2->set_values(4, 5);
    cout << ppoly1->area() << '\n';
    cout << ppoly2->area() << '\n';
    return 0;
}
```



Exceções

- No C++ Exceções são tratadas com as palavras-chave **try catch**. Para lançar uma exceção utilizamos a palavra-chave **throw**. Um exemplo disso:

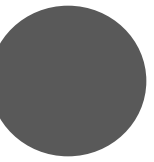
```
// exemplo de exceções
#include <exception>
#include <iostream>
using namespace std;

class myexception : public exception {
    virtual const char *what() const throw() { return "My exception happened"; }
} myex;

int main() {
    // exemplo de exceções: alocação de memória
    try {
        int *myarray = new int[1000];
    } catch (exception &e) {
        cout << "Standard exception: " << e.what() << endl;
    }

    // exemplo de exceções: exceção personalizada
    try {
        throw myex;
    } catch (exception &e) {
        cout << e.what() << '\n';
    }

    return 0;
}
```



Entrada e Saída com arquivos

- No C Trabalhávamos com arquivos utilizando, por exemplo, as funções **fwrite** e **fread**. Já no C++ trabalhamos com arquivos através de **streams** de entrada e saída. Um exemplo:

```
// manipulação de arquivos
#include <iostream>
#include <fstream>
using namespace std;

int main () {
    ofstream myfile;
    myfile.open ("example.txt");
    myfile << "Writing this to a file.\n";
    myfile.close();
    return 0;
}
```



<https://cplusplus.com/info/history/>

<https://en.cppreference.com/w/cpp/20>

<https://www.programmerall.com/article/2405560816/>

<https://m.cplusplus.com/info/description/>

<https://m.cplusplus.com/doc/tutorial/introduction/>