

Estudo de Caso: C++

Historico

No ano de 1979, em sua tese de PhD, Bjarne Stroustup trabalhou muito com a linguagem de programacao *Simula 67*, uma das primeiras a suportar o paradigma orientado a objetos. Bjarne norou que esse paradigma era muito util no desenvolvimento de software, porem a linguagem simula era muito lenta para uso pratico.

Apos isso, Bjarne comecou a trabalhar no o *C com Classes*, um *superset* da linguagem C que tentava adicionar funcionalidades de linguagens orientada a objetos na linguagem C, que e conhecida pela sua performance e utilidade em programacao de baixo nivel.

O primeiro compilador do *C com classes* foi chamado de *Cfront*, que tambem foi escrito em *C com classes*. Porem, o *C com Classes* foi abandonado em 1993 apos se tornar dificilmente extensivel.

Em 1983, o nome da linguagem foi mudado para *C++*. Varias features foram adicionadas junto a essa mudanca, como *funcoes virtuais*, *sobrecarga de funcoes*, referencias a variaveis com o simbolo *&* 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 padrao *C++11* foi finalizado. A Biblioteca *Boost* teve um impacto consideravel nas features adicionadas nesse padrao, com ate alguns modulos sendo adicionados diretamente da boost. Algumas das features adicionadas nesse padrao foram o suporte a expressoes regulares, uma biblioteca de randomizacao, uma nova bibliote para trabalhar com tempo, suporte a operacoes atomicas, uma biblioteca de multi threading, uma nova sintaxe de loops for (igual o foreach), a palavra chave *auto*, novas classes de container, alem de muitas outras.

Atualmente, o padrao vigente e o *C++20*. Algumas das features de padrao sao o operador de comparacao em 3 vias (*<=>*), melhorias na utilizacao de funcoes lambda, coroutines, modulos e muitas outras.

Existem planos para um padrao futuro chamado *C++23*, com muitas mudancas planejadas.

Objetivos, Contextualizacao e Caracteristicas

O *C++* e uma linguagem de alto nivel, com ferramentas para se trabalhar em baixo nivel tambem, compativel com o C e com uma extensivel biblioteca tanto padrao quanto da comunidade.

Ate certo nivel o *C++* e portavel, pois existem diversos compiladores para varias arquiteturas que fazem com que codigo *C++* seja compilado e rodado em varias arquiteturas com pouca ou nenhuma modificacao.

Compilacao

A linguagem *C++* e compilada, ou seja, o compilador traduz codigo escrito em *C++* para o codigo de maquina da arquitetura alvo. Essa compilacao e feita de uma so vez. Alem disso, o compilador pode aplicar otimizacoes de codigo, resultando em um codigo de maquina mais rapido e/ou confiavel.

Nivel de abstracao

A linguagem *C++* e uma linguagem de programacao de alto nivel, ou seja, oferece uma serie de abstracoes para facilitar o entendimento para humanos, como funcoes e objetos. Porem, com o *C++* tambem e possivel realizar manipulacoes de baixo nivel, inclusive sendo uma otima linguagem para isso.

Sistema de tipos

Forte ou Fraca *C++* e fortemente tipada, ou seja, possui bastante restricoes quanto a conversao de tipos entre variaveis. Por exemplo, nao e possivel a conversao entre um *Int* e um objeto *Fruit*.

Inferencia O *C++* suporta tanto a inferencia implicita, que e baseada no contexto que aquela variavel e usada, quanto a inferencia explicita, na qual o programador diz o tipo da variavel.

Checagem Novamente, o *C++* suporta tanto a checagem estatica quanto a checagem dinamica, ou seja, os seus tipos sao checados em tempo de compilacao e tambem em tempo de execucao.

Seguranca de tipos A linguagem *C++* nao e *type unsafe*, ou seja, ela leva em consideracao que o programador sabe o que esta fazendo e permite operacoes de conversao de tipos que podem levar a erros em tempo de execucao.

Paradigmas

A linguagem *C++* e multi paradigma, ou seja, suporta diversos paradigmas e tecnicas no mesmo programa. Alguns dos paradigmas que o *C++* suporta sao: Procedural, generico, orientado a objetos e mais recentemente o funcional.

Tour pela linguagem

Compiladores

Como dito anteriormente, o compilador é o software que traduz código feito em *C++* para código de máquina. 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
```

Podemos usar o *Make* para facilitar nossa vida.

Estrutura de um programa

Um programa simples em *C++* segue a seguinte estrutura:

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

Figure 1: types

A declaração de variáveis é feita da seguinte forma:

```
// variaveis
#include <iostream>

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

int main() {
    // declaração de variáveis com valores padrões
    int a(8), b = 12, c{21};
    // declaração de variáveis não inicializadas
    float result, d;

    // atribuição de valores
    d = -48.0f;
```

```

    // operacoes aritmeticas (com type casting)
    result = float((a - b) * c) / d;

    cout << result;
    return 0;
}

```

A deducao de tipos no *C++* e 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 e a classe `string`, que armazena sequencias 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 e 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 pre-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:

```

#include <iostream>

using namespace std;

int main() {
    // Operadores de atribuicao
    int a = 1, b = 2;

    // 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;
```

```

    return 0;
}

```

As regras de precedencia de operadores sao 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

Figure 2: precedence

Entrada e saida A biblioteca padrao do *C++* define o header `<iostream>` como padrao para operacoes simples de entrada e saida. Alem disso, temos o header `<sstream>` que lida com operacoes 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;
}

```

Controle de Fluxo e Loops A linguagem possui os comandos de selecao padrao: `if`, `else if`, `else` e o `switch case`. Alem disso, possui tambem os loops `for`, `while` e `do while`. Para a utilizacao em loops, temos os comandos de alteracao de fluxo `continue`, `break` e `goto`. Exemplos desses comandos podem ser vistos a seguir:

Referencias

[<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/>]