Introdução ao C++

Por Leandro Santiago

Sobre quem escreve

- Entusiasta do software livre, é ~~viciado em~~ usuário de C++ desde algum momento não-lembrado durante sua graduação em Informática na Universidade Estadual de Maringá.
- No tempo livre pedala com sua bicicleta antiga e critica “o sistema” em listas de discussões na Internet.

C++: a linguagem com bug até no nome

Mais velha que eu!

- Começou a ser desenvolvida em 1979 (!) por Bjarne Stroustrup e até hoje tem participação ativa deste.
- É uma linguagem de propósito geral: de sistemas operacionais e compiladores até páginas web.
- Seu nome surgiu do operator ++ da linguagem C. Significa “C com incremento”.

Bjarne Stroustrup

Não tem dono

- Em 1998 foi padronizada pela ISO (ISO/IEC 14882:1998), teve sua segunda versão padronizada em 2003 (ISO/IEC 14882:2003) e, por fim, sua terceira versão foi finalizada em 2011 (ISO/IEC 14882:2011).
- Não existe uma implementação padrão ou uma empresa ou instituição que controla o seu desenvolvimento.
- Toda e qualquer modificação na linguagem requer muita discussão, que frequentemente levam anos.

Alta-performance

- Embora seja de propósito geral, é frequentemente escolhida para o desenvolvimento de aplicações de alta performance.
- Não restringe o programador quanto ao uso dos recursos computacionais. Por isso é uma “faca de dois legumes”.
- Quem usa? Google, Facebook, Microsoft, IBM, **eu**, dentre outros menos importantes.

TIOBE maio/2012

Position May 2012	Position May 2011	Delta in Position	Programming Language	Ratings May 2012	Delta May 2011	Status
1	2	↑	C	17.346%	+1.18%	A
2	1	↓	Java	16.599%	-1.56%	A
3	3	=	C++	9.825%	+0.68%	A
4	6	↑↑	Objective-C	8.309%	+3.30%	A
5	4	↓	C#	6.823%	-0.72%	A
6	5	↓	PHP	5.711%	-0.80%	A
7	8	↑	(Visual) Basic	5.457%	+0.96%	A
8	7	↓	Python	3.819%	-0.76%	A
9	9	=	Perl	2.805%	+0.57%	A
10	11	↑	JavaScript	2.135%	+0.74%	A
11	10	↓	Ruby	1.451%	+0.03%	A
12	26	↑↑↑↑↑↑↑↑↑↑↑↑↑↑	Visual Basic .NET	1.274%	+0.79%	A
13	21	↑↑↑↑↑↑↑↑	PL/SQL	1.119%	+0.62%	A
14	13	↓	Delphi/Object Pascal	1.004%	-0.07%	A

Implementações

- Há ao menos 38 implementações do C++, segundo a Wikipedia.
- C++ não é nem compilada, nem interpretada. Tudo depende da implementação.
- As principais implementações são o GCC (open source) e MSVC (proprietário).
- Há versões interpretadas, como o ROOT, do CERN (laboratório do fim-do-mundo)

Porque o C++ não gosta do C?*

- C++ é filha do C e irmã má do Objective-C e Objective-C++
- Runtime pequeno e maior parte das decisões tomadas em tempo de compilação.
- Possui alta compatibilidade com C na maior parte das implementações, tanto em código-fonte quanto em nível binário (geralmente usam o mesmo linker).

*Porque o C não “tem classe”

Críticas

- A especificação da linguagem é enorme. Geralmente usamos somente um subconjunto de todas as características da linguagem.
- (e quando o subconjunto que eu uso for diferente do seu?)
- A linguagem é, por natureza, ambígua.
- Não possui coletor de lixo!
- Não possui suporte nativo à concorrência.
- É uma das linguagens mais criticadas de todos os tempos.

“C++ is a horrible language. It's made more horrible by the fact that a lot of substandard programmers use it”

Linus Torvalds

“The major cause of complaints is C++['s] undoubted success. As someone remarked: There are only two kinds of programming languages: those people always bitch about and those nobody uses.”

Bjarne Stroustrup, comp.lang.c++

“C++ is dead. Nobody uses C++ any more. Why should I bother?”

“When you read this, it is very presumable, that you sit in front of a program, written in C++. Most of the applications we use in our day to day work are written in C++. The problem is, that C++ programs do not tell, that they are written in C++. They do not need to install a C++ run time engine to run C++ applications. C++ programs also look just as programs should look like.”

Tntnet FAQ

A linguagem – Olá Mundo

```
1 #include <iostream>
2
3 int main(int argc, char **argv)
4 {
5    std::cout << "Hello World!" << std::endl;
6    return 0;
7 }
```

Executar?

- Nos unices (Linux, OS X, *BSD, etc.):
- g++ hello.cpp -o hello
- ./hello
- Onde g++ é o compilador da GNU.
- No windows é necessário adicionar a extensão .exe
- Usarei o G++ do GCC nesta apresentação

Estatica- mas não fortemente tipada

- A tipagem no C++, assim como no C, é estática, ou seja, é definida em tempo de compilação. Mas é possível fazer casts e usar inteiros e reais de forma mista.

Instanciando uma variável

- Uma variável pode ser instanciada tanto no frame da função, sendo desalocada ao final do bloco onde foi declarada, ou pode ser instanciada na heap, por meio do operador new, devendo ser manualmente desalocada.
- Também é possível alocarmos variáveis no escopo global.

```
1 int z = 42; // global
2
3 int function()
4 {
5    int b = 2; // pilha
6    int c(4); // pilha
7    int d = c; // pilha
8    int *e = new int(5); // heap
9 }
```

Declarações de funções

- Uma função ou método pode ser declarada infinitas vezes (desde que com a mesma assinatura), mas só pode ser definida (implementada) uma única vez.

```
 1 int myFunction();
 2 int myFunction(int,int);
 3
 4 int myFunction(int a)
 5 {
 6   return a;
 7 }
 8 int myFunction(int a, int b)
 9 {
10   return myFunction(a) + b;
11 }
12 int myFunction(string a, int b = 42) {
13   return a.size() * myFunction(2,b);
14 };
15
16 tuple<string,map<string,int>> myOtherFunction(const vector<double> &v);
~
```

Passagem de parâmetros

- Como em C, toda passagem de parâmetro é por cópia.
- Para passar por referência, podemos passar uma cópia do endereço de memória.
- Ou podemos passar uma referência!

```cpp
int function(int *b)
{
  *b = 3;
  return 2 + (*b);
}

int function(int &b)
{
  b = 3;
  return 2 + b;
}

int function(const string &name)
{
  name = "Leandro"; // ERRO! name é constante!
}

int main()
{
  int c = 42;
  cout << function(&c) << endl; // recebe um endereço de memória ou ponteiro
  cout << function(c) << endl; // recebe uma referência
  cout << function("João") << endl; // recebe um const char * que será usado como string
}

```

A linguagem - OO

- C++ não é orientada a objetos.
- C++ **SUPORTA** orientação a objetos, por meio de classes e estruturas.
- Classes e estruturas são essencialmente a mesma coisa, mas nas estruturas todos os atributos e métodos são, por omissão, públicos, ao contrário das classes, onde estes são privados.

A linguagem – OO - Classes

```
 1 class Foo
 2 {
 3 public:
 4   int foo()
 5   {
 6     return 1;
 7   }
 8
 9 private:
10   int counter;
11
12 protected:
13
14   void doSomething()
15   {
16     // nothing
17   }
18 };
```

Métodos const

- Métodos com a assinatura const só podem ser aplicados em objetos constantes.
- Estes métodos não conseguem modificar o estado do objeto em que são chamados.
- Úteis para getters

```cpp
struct A
{
  int value;

  A(int v): value(v)
  {}

  int getValue() const
  {
    // não posso alterar o estado do objeto
    // e só posso chamar qualquer método const
    return value;
  }

  int getValue()
  {
    // este métoto consegue alterar estado do objeto
    // e posso chamar métodos não const
    value *= 2;
    return value;
  }
};

int main()
{
  A a(10);
  const A a2(10);

  std::cout << a.getValue() << std::endl; // 20
  std::cout << a2.getValue() << std::endl; // 10
}
```

A Linguagem – OO – Herança múltipla

```
22 class Bar: public Foo, public OtherClass
23 {
24 public:
25    const char *bar()
26    {
27       return "Bar";
28    }
29 };
```

A Linguagem – OO – Construtores

- Todo tipo possui um construtor. Inclusive tipos primitivos. No caso destes, existe somente o chamado construtor de cópia.
- Se não declarado, o compilador insere um construtor vazio em toda classe e estrutura, durante a compilação.

A Linguagem – OO – Construtores

- Uma classe ou estrutura pode ter vários construtores. Até a versão 2003, um construtor não poderia chamar (delegar) outro.
- Os construtores são métodos com o mesmo nome da classe, mas sem um tipo de retorno.

A Linguagem – OO – Destrutor

- O compilador cria um destrutor vazio, caso não declarado. Ele deve ser único
- É um método sem retorno e sua assinatura é um til (“~”) seguido do nome da classe ou estrutura. Não pode receber parâmetros.
- É executado implicitamente quando a execução sai do escopo da instância, caso esteja na pilha, ou quando o operador delete é chamado, caso esteja na heap.

Construtores e Destrutor

```
37 class MyClass
38 {
39 public:
40    MyClass(int age) {}
41    MyClass(const MyClass &other) {}
42    MyClass(int age, const std::string name) {}
43    virtual ~MyClass()
44    {
45       // clean the house
46    }
47 };
```

Construtores são mágicos

- Se uma classe A possui um construtor que tem como parâmetro uma variável do tipo B, B pode ser usado em todo lugar onde A pode ser usado.

```
1  struct A
2  {
3    A(int b){}
4  };
5
6  int function(A a)
7  {
8  }
9
10 int main()
11 {
12   A a(4);
13   function(a);
14
15   A a2 = 4; // chama o construtor com inteiro
16
17   // constroi um objeto A com parâmetro 5 e passa como parâmetro
18   function(5);
19 }
20
```

```
string name = "Leandro";
```

Eu mesmo: o ponteiro this

- this é uma variável especial existente (e implícita) em todos os métodos de instância, ou seja, aqueles que não são estáticos. E é um ponteiro :-)

Clamando as classes pais

- Não existe um super(), por não haver só um pai.
- Cada pai deve ser acessado pelo nome.

```
 1 class B: public A, public C
 2 {
 3 public:
 4    B(int c): A(c+2), C(c)
 5    {
 6       // chama um método de A,
 7       // mas usando a instância atual
 8       A::metodoEmA(5);
 9    }
10 };
~
```

Métodos virtuais

- Sem a assinatura “virtual”, um método não pode ser sobrescrito por uma classe filha.

```cpp
#include <iostream>

using namespace std;

struct A
{
  void metodo()
  {
    cout << "A" << endl;
  }
};

struct B: public A
{
  void metodo()
  {
    cout << "B" << endl;
  }
};

int main()
{
  A *a = new B();
  a->metodo(); // imprime A, pq B não sobrescreve metodo()
}
```

```cpp
#include <iostream>

using namespace std;

struct A
{
  virtual void metodo()
  {
    cout << "A" << endl;
  }
};

struct B: public A
{
  void metodo()
  {
    cout << "B" << endl;
  }
};

int main()
{
  A *a = new B();
  a->metodo(); // imprime B, pq B sobrescreve metodo()
}
```

Métodos virtuais puros

```
 1 #include <iostream>
 2
 3 using namespace std;
 4
 5 struct A
 6 {
 7   virtual void metodo() = 0;
 8 };
 9
10 struct B: public A
11 {
12   void metodo()
13   {
14     cout << "B" << endl;
15   }
16 };
17
18 int main()
19 {
20   A *a = new B;
21   a->metodo(); // imprime B, pq B implementa metodo()
22
23   A *a2 = new A; // erro! A não pode ser instanciado!
24 }
~
```

Classes abstratas

- Em C++, uma classe é abstrata caso possua ao menos um método virtual-puro.
- Você pode criar classes somente com métodos virtuais-puros.
- Por isso não existem interfaces em C++!

Métodos estáticos

- São métodos que são aplicados na classe, não podendo ser aplicados na instância. Na prática é um método que não possui o ponteiro implícito this.
- Sua chamada é diferente.

```
 4 struct A
 5 {
 6   int a;
 7   A(int v): a(v) {};
 8
 9   static void metodo(int a)
10   {
11     cout << "print " << a << endl;
12   }
13
14   void metodo2(int a)
15   {
16     cout << "print " << a + this->a << endl;
17   }
18 };
19
20 int main()
21 {
22   A::metodo(2); // print 2
23   A(3).metodo2(2); // print 5
24 }
```

Classes anônimas

- Um dia destes um colega se gabou do Java em relação ao C++ porque o Java tinha classes anônimas e C++ não.
- C possui classes (estruturas) anônimas desde mil novecentos e bolinhas!
- C++, portanto, herdou essa capacidade e a estendeu com herança e métodos.

```
 1 class A
 2 {
 3 public:
 4   void metodo() {}
 5   virtual void metodo2() {}
 6 };
 7
 8 class B: public A
 9 {
10 public:
11   void metodo2() {}
12 } b;
13
14 A *c = new (class: public A
15 {
16 public:
17   virtual void metodo2() {}
18   void metodo3() {}
19 });
20
21 int main()
22 {
23   A a;
24   a.metodo2();
25   b.metodo2();
26   c->metodo2();
27 }
~
```

Sobrecarga de operadores

- Quase todos os 48 operadores do C++ podem ser sobrecarregados (menos o vírgula).
- A sobrecarga de operadores tem como função deixar o código mais legível e permitir a execução de ações complexas com a sintaxe de uma operação simples.

```cpp
#include <iostream>

using namespace std;

class A
{
  int a;
public:
  A(int value = 0): a(value) {}

  int getA()
  {
    return a;
  }

  A operator+(int value)
  {
    A o(a);
    o.a += value;
    return o;
  }
};

int main()
{
  A a(3), b;

  b = a + 4;

  // é o mesmo que
  b = a.operator+(4);

  cout << a.getA() << endl; // 3
  cout << b.getA() << endl; // 7
}
```

Sobrecarga de operadores?

```cpp
#include <stdio.h>
class Evil
{
    class Hack
    {
    public:
        const Hack& operator *() const
        {
            return *this;
        }
    };
public:
    const Evil& operator -() const
    {
        printf( "Hello World!\n" );
        return *this;
    }
    Hack operator *() const
    {
        return Hack();
    }
};
int main( const int, const char* const* const )
{
        *   * ***** *     *       ***         * * *   ***   ***   *      ***
        *   * *     *     *     *    *        * * * *    * *   * *      *   *
        ***** ***   *     *     *    *        * * * *    * ***   *      *   *
        *   * *     *     *     *    *        ***** *    * *   * *      *   *
        *   * ***** **** ****   ***          * * *   ***   *   * **** ***
    -Evil();
    return 0;
}
```

```
 1 int main ()
 2 {
 3    using namespace analog_literals::symbols;
 4    using namespace analog_literals::shapes;
 5
 6    line<3>(I-------I);
 7
 8    rectangle<2, 3>(o-----o
 9                    |     !
10                    !     !
11                    !     !
12                    o-----o);
13
14    cuboid<6, 6, 3>(o-------------o
15                    |L             \
16                    | L             \
17                    |  L             \
18                    |   o-------------o
19                    |   !             !
20                    !   !             !
21                    o   |             !
22                     L  |             !
23                      L |             !
24                       L|             !
25                        o-------------o );
26
27    cuboid<3, 4, 2>(o-------o
28                    |L       \
29                    | L       \
30                    |  o-------o
31                    |  !       !
32                    o  |       !
33                     L |       !
34                      L|       !
35                       o-------o);
36 }
```

Cast operator

- Permite que o objeto de uma classe possa ser utilizado no lugar do de outro tipo (primitivo ou não), podendo ou não imitar seu comportamento.
- O objetivo novamente é tornar o código mais legível substituindo métodos adaptadores por chamadas simples.

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

struct A
{
  operator int()
    {
      return 20;
    }
};

int main()
{
   A a;

   int b = 2 + a;
   cout << b << endl; // 22
}
```

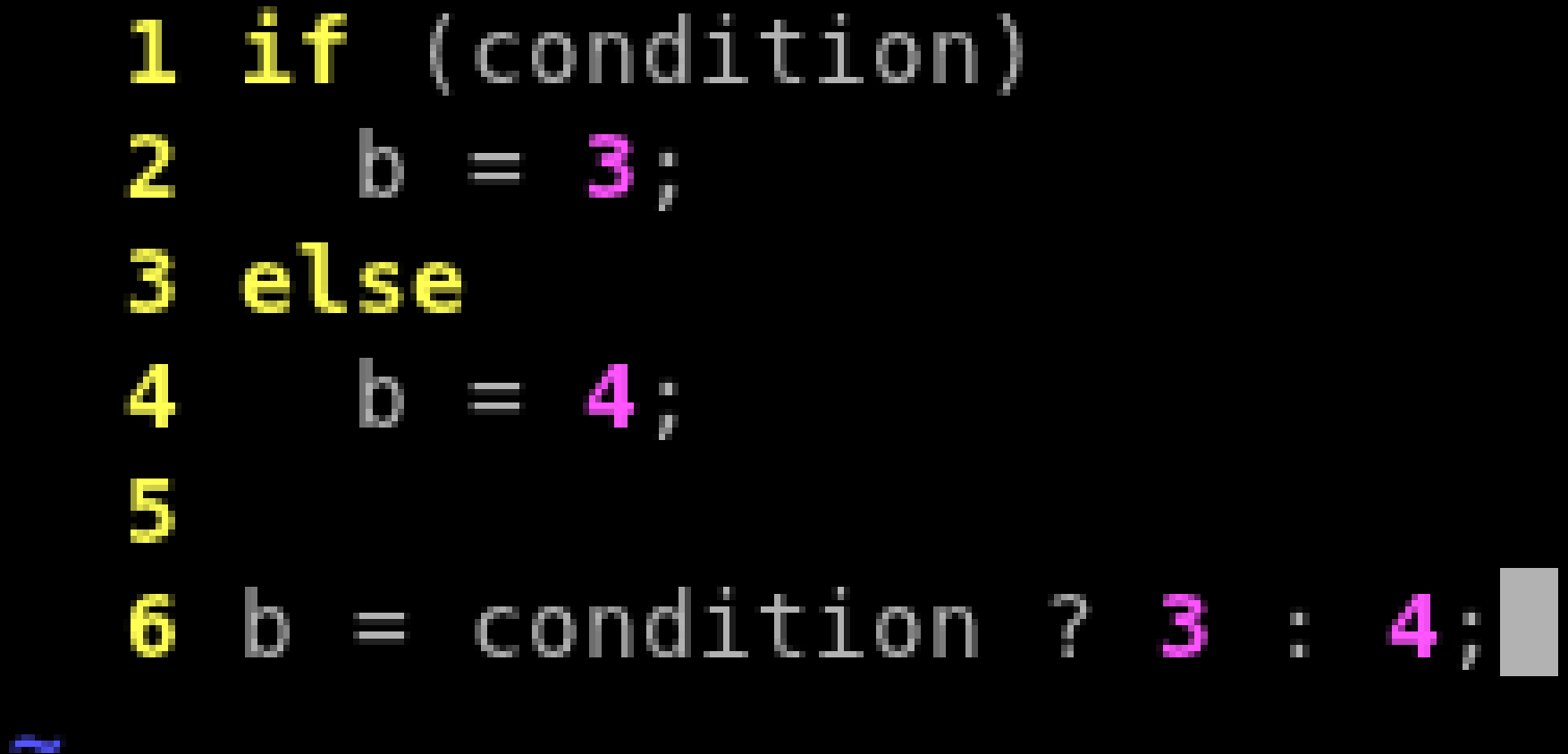
Exceções

- Qualquer variável pode ser lançada como exceção, seja alocada na heap ou na pilha.
- É recomendado usar exceções que herdam de std::exception.

```cpp
#include <string>

using namespace std;

int function(int a, int b, int c)
{
  if (a) throw 2;
  if (b == 42) throw string("lançado b");
  if (c/4 == 67) throw new exception();
  return 5;
}

int main()
{
  try {
    int c = function(98,42,87);
  } catch (int i) {
    // pega exceção de inteiro
  } catch (const exception &e) { // poderia ser
    // pega std::exception
  } catch(string e) {
    // pega string
  } catch (...) {
    // pega qualquer coisa (ou tudo)
  }
}
```

Condicional

```
1 if (condition) {
2     // corpo do if
3 }
4
5 if (condition) {
6     // corpo do if
7 } else doElse();
8
9 if (a) b() else c();
```


1 if (condition)
2 b = 3;
3 else
4 b = 4;
5
6 b = condition ? 3 : 4;

Switch-case

```
11 switch(i) {
12    case 10:
13    case 30:
14       doCode1();
15    case 40:
16       doCode2();
17    break;
18    case 50:
19       doCode3();
20    break;
21    default:
22       doCodeDefault();
23 }
```

Todo for é um while

```
1 for (int i = 0; i < 10; i++) {
2   cout << i << endl;
3 }
4
5 for (;;) {
6   doSomething();
7 }
8
9 while(true) {
10   doSomething();
11 }
12
13 int i = 0;
14 while (i<10) {
15   cout << i << endl;
16   i++;
17 }
```

Namespaces

- Servem para evitar conflito de nomes de classes, variáveis, etc.

```
 1 namespace Que {
 2   namespace Nome {
 3     namespace Grande {
 4       void doSomething()
 5       {
 6         // something
 7       }
 8     }
 9   }
10 }
11
12 int main()
13 {
14   Que::Nome::Grande::doSomething();
15
16   namespace nomao = Que::Nome::Grande;
17   nomao::doSomething();
18
19   {
20     using namespace Que::Nome::Grande;
21     doSomething();
22   }
23 }
24
```

Programação genérica com templates

```
 4 template<typename T>
 5 T sum(const T &t1, const T &t2)
 6 {
 7   return t1 + t2;
 8 }
 9
10 int main()
11 {
12   std::cout << sum<int>(3,4) << std::endl; // 7
13   std::cout << sum<std::string>("Marin","gá") << std::endl; // Maringá
14 }
```

O pré-processador

- O pré-processador é um programa executado sobre o código-fonte antes do compilador em si e tem como função expandir todo o código começado com o caractere “#” para o seu correspondente. Pode ser chamado pelo comando cpp.

```
1 // include.h
2 int myFunc()
3 {
4     return 3;
5 }
6
7 #define DEFINED_VALUE 42
```

```
1 // prepproc.cpp
2 #include "include.h"
3
4 int main()
5 {
6     int a = DEFINED_VALUE;
7 }
~
```

```
1 # 1 "preproc.cpp"
2 # 1 "<embutido>"
3 # 1 "<linha-de-comando>"
4 # 1 "preproc.cpp"
5
6 # 1 "include.h" 1
7
8 int myFunc()
9 {
10    return 3;
11 }
12 # 3 "preproc.cpp" 2
13
14 int main()
15 {
16    int a = 42;
17 }
```

STL

- Vector<>: semelhante ao array do C
- List<>: lista ligada
- Map<>: atua como hash (mas não é hash!)
- Set<>: conjunto como na matemática
- Tuple<> (C++0x): aglomera vários tipos

C++0x ou C++2011

- Templates de quantidade de parâmetros variável
- Templates recursivos
- Lambda
- Delegate constructor
- Inicialização com listas
- For com iteração
- Inferência de tipo com auto

Templates recursivos

- Eles já eram possíveis no C++ 2003, mas foram melhorados com o 2011.

```
 2 template<int num>
 3 struct Fact
 4 {
 5   static const int value = num * Fact<num-1>::value;
 6 };
 7
 8 template<>
 9 struct Fact<1>
10 {
11   static const int value = 1;
12 };
13
14 int main()
15 {
16   std::cout << Fact<6>::value << std::endl;
17 }
```

Funções anônimas
(lambdas não são closures!)

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

int main()
{
  //poderia inferir automaticamente com "auto f ="
  function<int(int,int)> f = [](int a, int b) -> int {return a + b;};

  cout << f(3,4) << endl; // 7
}
```

Testes unitários

- Há vários frameworks de testes unitários no C++ (mais os do C), tais como CppUnit, Unit++, Cxxtools e Boost.Test.

Boost.Test

Testes unitários com Boost.Test

```
 1 #define BOOST_TEST_DYN_LINK
 2 #define BOOST_TEST_MODULE Caixa
 3
 4 #include <boost/test/unit_test.hpp>
 5 #include "caixa.h"
 6
 7 BOOST_AUTO_TEST_CASE(retorna1nota) {
 8   vector<int> expected {10};
 9   BOOST_CHECK_MESSAGE(Caixa::saca(10) == expected,"retorna 1 nova");
10 }
11
12 BOOST_AUTO_TEST_CASE(notasindisponiveis) {
13   BOOST_CHECK_THROW(Caixa::saca(91),std::exception);
14 }
```

Testes unitários com Boost.Test

- Para compilar:

```
$ g++ caixa.cpp caixa_tests.cpp
-lboost_unit_test_framework -std=c++0x -o
caixa_tests

$ ./caixas_tests

Running 2 test cases...

*** No errors detected
```

Projetos

- Como é chato ter que digitar os longos comandos de compilação (que frequentemente ocupam várias linhas), utilizamos algum sistema para automatizar a compilação de um projeto.
- Softwares bastante utilizados com C++ são:
  - CMake
  - Autotools

CMake

```
1 # CMakeLists.txt
2 project(caixa)
3 cmake_minimum_required(VERSION 2.6)
4 add_definitions(-std=c++0x -Wall)
5 add_executable(caixa_test caixa.cpp caixa_test.cpp)
6 target_link_libraries(caixa_test -lboost_unit_test_framework)
```

Perguntas?

Contato

- E-mail/Gtalk: **leandrosansilva@gmail.com**
- Gitorious: **http://gitorious.org/~leandrosansilva/**
- Blog Pessoal (1 post/ano): **http://leandro.setefaces.org/**
- Sem twitter/Sem facebook/Sem Google+ (interneticamente anti-social)