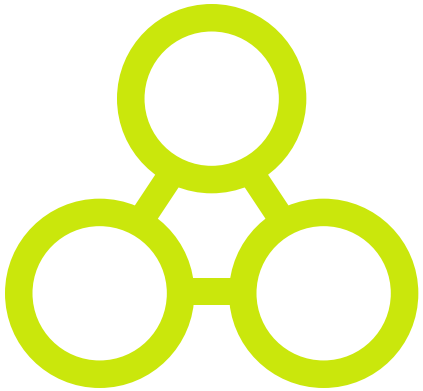


nemo

ontology & conceptual modeling research group

# [ Curso Rápido de C++ ]

## Desenvolvimento OO em C++ para quem já sabe C e Java


Vítor E. Silva Souza

(vitor.souza@ufes.br)

http://www.inf.ufes.br/~vitorsouza

Departamento de Informática

Centro Tecnológico

Universidade Federal do Espírito Santo

Este obra foi licenciada sob uma Licença Creative Commons Atribuição 3.0 Não Adaptada.

# Sobre o Curso Rápido de C++

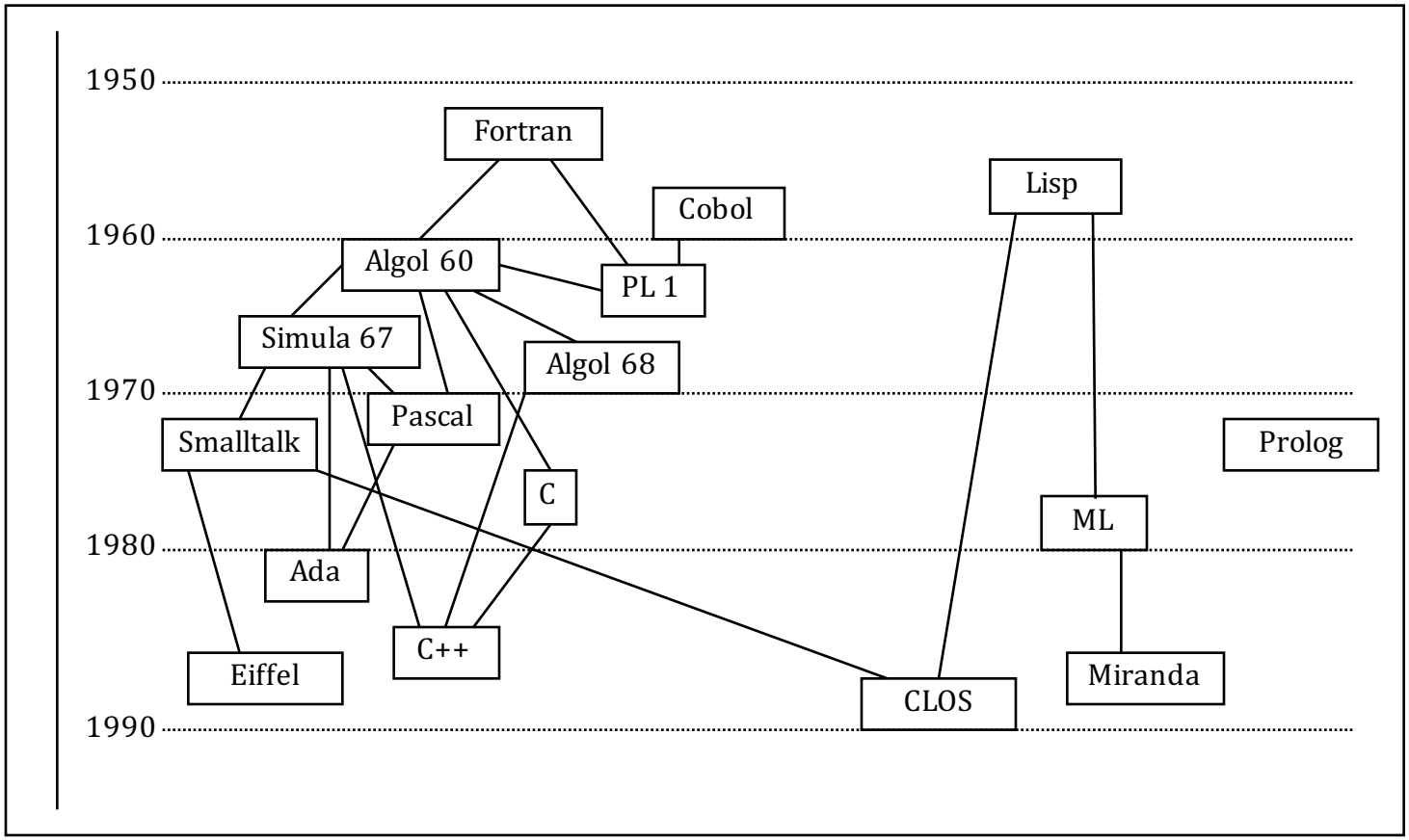
- Pré-requisito:
  - Desenvolver bem em C;
  - Saber outra linguagem OO (de preferência Java);
  - Conceitos OO não serão apresentados, apenas como aplicar tais conceitos em C++.
- Bibliografia básica:
  - Apostila Programação Orientada a Objetos em C++ (Berilhes B. Garcia, Flávio M. Varejão);
  - P. Deitel, H. Deitel. C++ Como Programar, 5ª Edição. Pearson Education, 2010.

# A linguagem C++

- Criada por Bjarne Stroustrup no AT&T Bell Laboratories (hoje Bell Labs) em 1979 – 1983;
- Adiciona características OO à linguagem C ("C with Classes" -> C++):
  - Classes, funções virtuais, sobrescrita, herança múltipla, *templates*, exceções, ...
- Tornou-se padrão ISO/IEC em 1998, versão mais atual: ISO/IEC 14882:2011 (informalmente C++11);
- Vários compiladores disponíveis:
  - C++ Builder, gcc, MinGW, Turbo C++, Visual C++, ...

# A linguagem C++

- Total compatibilidade com C (qualquer programa em C é um programa C++);
- Buscou ideias em Simula 67 e Algol 68:



# C -> C++, Java vs. C++

- C possui um único mecanismo de passagem de parâmetros (por valor):
  - Passagem de ponteiros é subterfúgio complexo;
  - C++ resolve o problema de forma eficiente.
- C++ traz ainda:
  - Classes e variáveis/funções membros (atributos/métodos);
  - Sobrecarga (mais completa do que Java);
  - Herança (permite herança múltipla);
  - Funções virtuais (sobrescrita de métodos);
  - Etc.

# Começando com C

- Programa simples, com operações aritméticas em valores numéricos;
- Problema: e pra fazer o mesmo com valores muito grandes, que não cabem em `int`s e `long`s?

```
#include <stdio.h>

main () {
  int a = 193;
  int b = 456;
  int c;
  c = a + b + 47;
  printf("%d\n",c);
}
```

# Incrementando C: C++

- C++ permite:
  - Definir uma classe para números grandes;
  - Manter o código bem parecido com o programa C.
- O mesmo programa em C (e até mesmo em Java) não são tão convenientes e elegantes.

```
#include "BigInt.h"

main() {
  BigInt a = "25123654789456";
  BigInt b = "456023398798362";
  BigInt c;
  c = a + b + 47;
  c.print();
  printf("\n");
}
```

# E na especificação de BigInt...

- A especificação de BigInt deve dizer como:
  1. Criar objetos da classe BigInt (de *strings*!);
  2. Somar BigInt x BigInt e BigInt x int/long;
  3. Imprimir um BigInt.

```
#include "BigInt.h"

main() {
  BigInt a = "25123654789456";
  BigInt b = "456023398798362";
  BigInt c;
  c = a + b + 47;
  c.print();
  printf("\n");
}
```

1

2

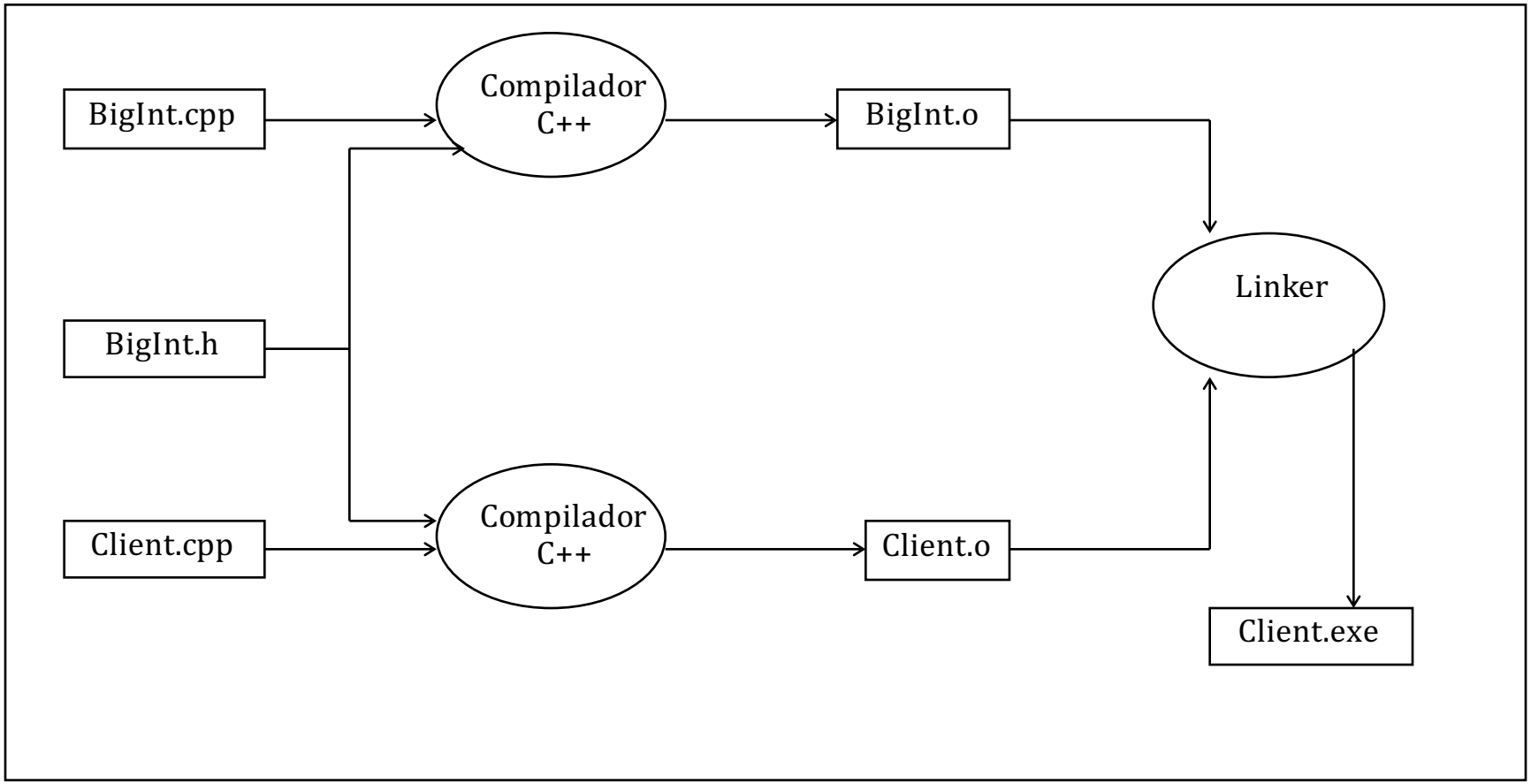
3

# Destruição de objetos

- Ainda falta ao exemplo a destruição de objetos `BigInt` desnecessários;
- C++ não possui coletor de lixo como Java;
- Porém as instruções de criação/destruição são mais simples do que `malloc/free`.

# Arquitetura da solução exemplo

- Similares aos TADs;
- Porém com abstrações mais adequadas.



[ Curso Rápido de C++ ]

# DECLARANDO CLASSES (DEFINIÇÃO DA INTERFACE)

# O arquivo de declarações

```
#include <stdio.h>

class BigInt {
  char* digitos;
  unsigned ndigitos;

public:
  BigInt(const char*);
  BigInt(unsigned n = 0);
  BigInt(const BigInt&);

  void operator=(const BigInt&);
  BigInt operator+(const BigInt&) const;

  void print(FILE* f = stdout) const;

  ~BigInt() { delete digitos; }
};
```

Atributos

Construtores

Sobrescrita = e +

Impressão

Destrutor

# Declaração de classes

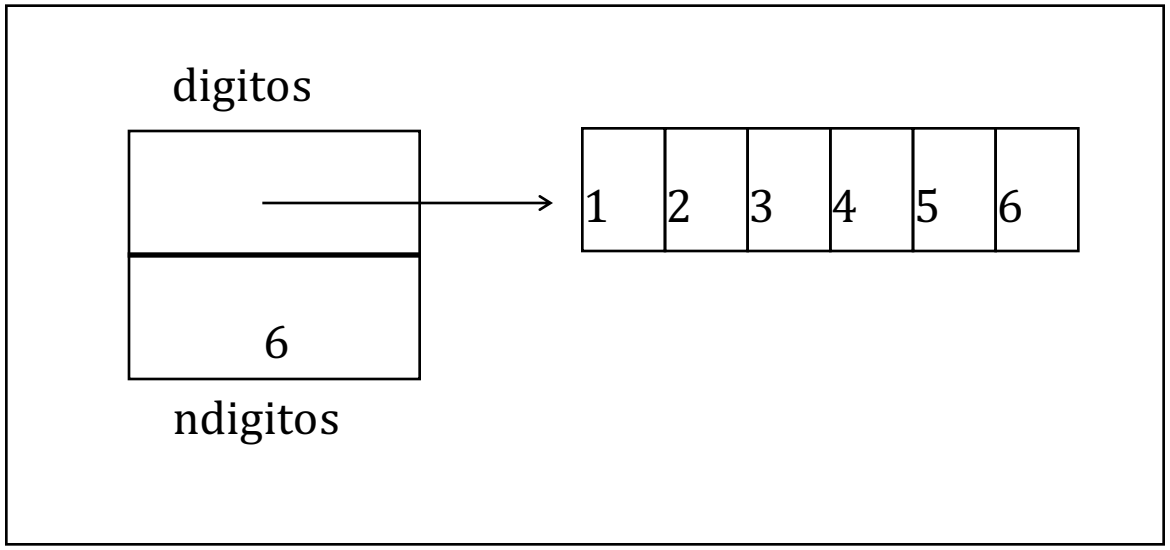
- Similar ao conceito de `structs`/TADs em C, porém:
  - Construtores, sobrescrita, destrutores, especificadores de acesso, etc.
- Similar à declaração de classes em Java, porém:
  - Construtores especiais;
  - Sobrescrita de operadores primitivos;
  - Destrutores (`finalize()` ?);
  - Especificadores de acesso/visibilidade em bloco;
  - Sem especificador: o *default* é `private`;
  - Implementação em arquivo separado.

# Sobrecarga

- C++, assim como Java, permite sobrecarga (várias funções com mesmo nome);
- Em BigInt, o exemplo é o construtor;
- Como em Java:
  - As funções devem ter assinaturas diferentes;
  - Para assinatura, conta o nome da função e os tipos dos parâmetros. O tipo de retorno não conta;
  - Há exceções para funções que usem *templates* (assunto avançado, veremos mais adiante).

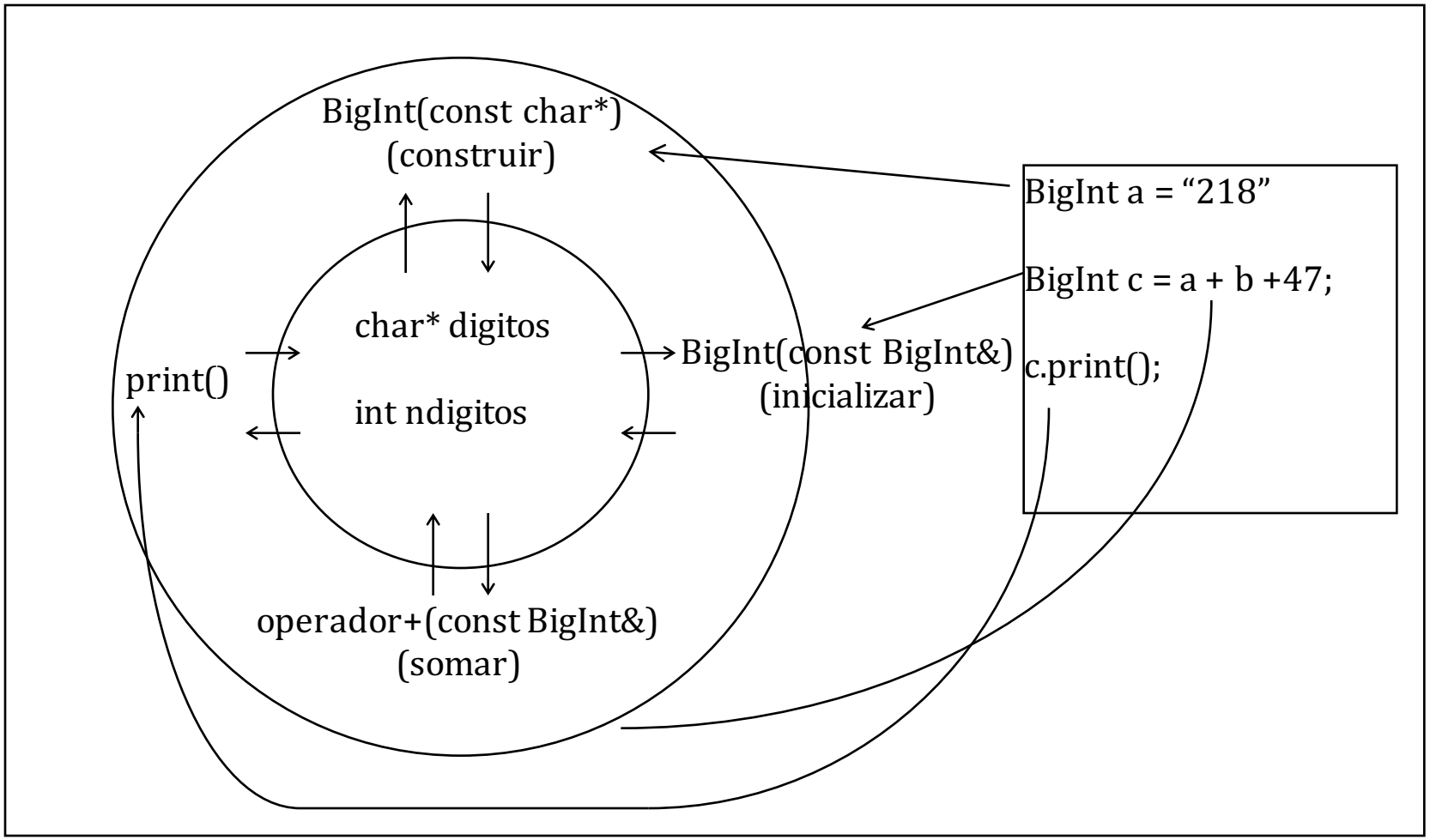
# Explorando a classe `BigInt`

- Classes são similares a `structs`;
- `BigInt` possui um vetor de chars (ponteiro) e um contador do número de dígitos;
- O número é armazenado de trás para frente;
  - Ex.: o `BigInt` “654321” seria:



# Explorando a classe `BigInt`

- Atributos (variáveis membro) privadas, métodos (funções membro) públicas:



# Explorando a classe `BigInt`

- Sobrecarga como em Java: o mesmo nome é permitido, desde que os parâmetros sejam diferentes:

```
/* ... */

  BigInt(const char*);
  BigInt(unsigned n = 0);
  BigInt(const BigInt&);

/* ... */
```

# Construtor default

- Mesma regra de Java:
  - Se nenhum construtor é dado, um construtor default (sem argumentos) é adicionado pelo C++;
  - Se algum construtor for dado, o construtor sem argumentos não é adicionado automaticamente;
- Em BigInt, um dos construtores serve de construtor default (sem argumentos), pois seu argumento tem valor default:

```
/* ... */

  BigInt(unsigned n = 0);

/* ... */
```

Só pode ter um, senão:
`error: call to constructor of 'BigInt' is ambiguous`

# Explorando a classe `BigInt`

- Funções podem ter argumentos *default*:

```
/* ... */

  BigInt(unsigned n = 0);

/* ... */

  void print(FILE* f = stdout) const;

/* ... */
```

- Se forem chamadas sem argumento, o valor *default* é utilizado;
- Argumentos com *default* devem estar à direita.

# Instâncias diretas e ponteiros

- Em Java, instâncias eram sempre referências (ponteiros). C++ possui os dois tipos;
- Os já conhecidos `.` e `->` de C são usados para acessar membros da classe:

```
BigInt a = "25123654789456";
a.print();

BigInt* p;
p = &a;
p->print();
```

# Construtores (instâncias diretas)

- Mais versáteis que Java, atenção às instâncias diretas!

```
// Chama o construtor default: BigInt (unsigned n = 0):
BigInt a, vetor[10];

// Chama o mesmo construtor, só que especificando valor:
BigInt b = 123;

// O mesmo construtor também é chamado para conversão:
BigInt c = a + b + 47;

/* Acima, o construtor BigInt (const BigInt&) também é
acionado para criar um BigInt a partir de outro. */

// Chama o construtor BigInt (const char*):
BigInt d = "456";
BigInt e("789");
```

# Sobrescrita

- C++ permite sobrescrever qualquer operador, exceto o operador de seleção (`.`) e a expressão condicional (`?:`);

```
// Na classe BigInt:
void operator=(const BigInt&);
BigInt operator+(const BigInt&) const;

// No cliente:
BigInt a = "25123654789456";
BigInt b = "456023398798362";
BigInt c;
c = a + b + 47;
```

- Obviamente, o desenvolvedor deve implementar os métodos `operator=` e `operator+`.

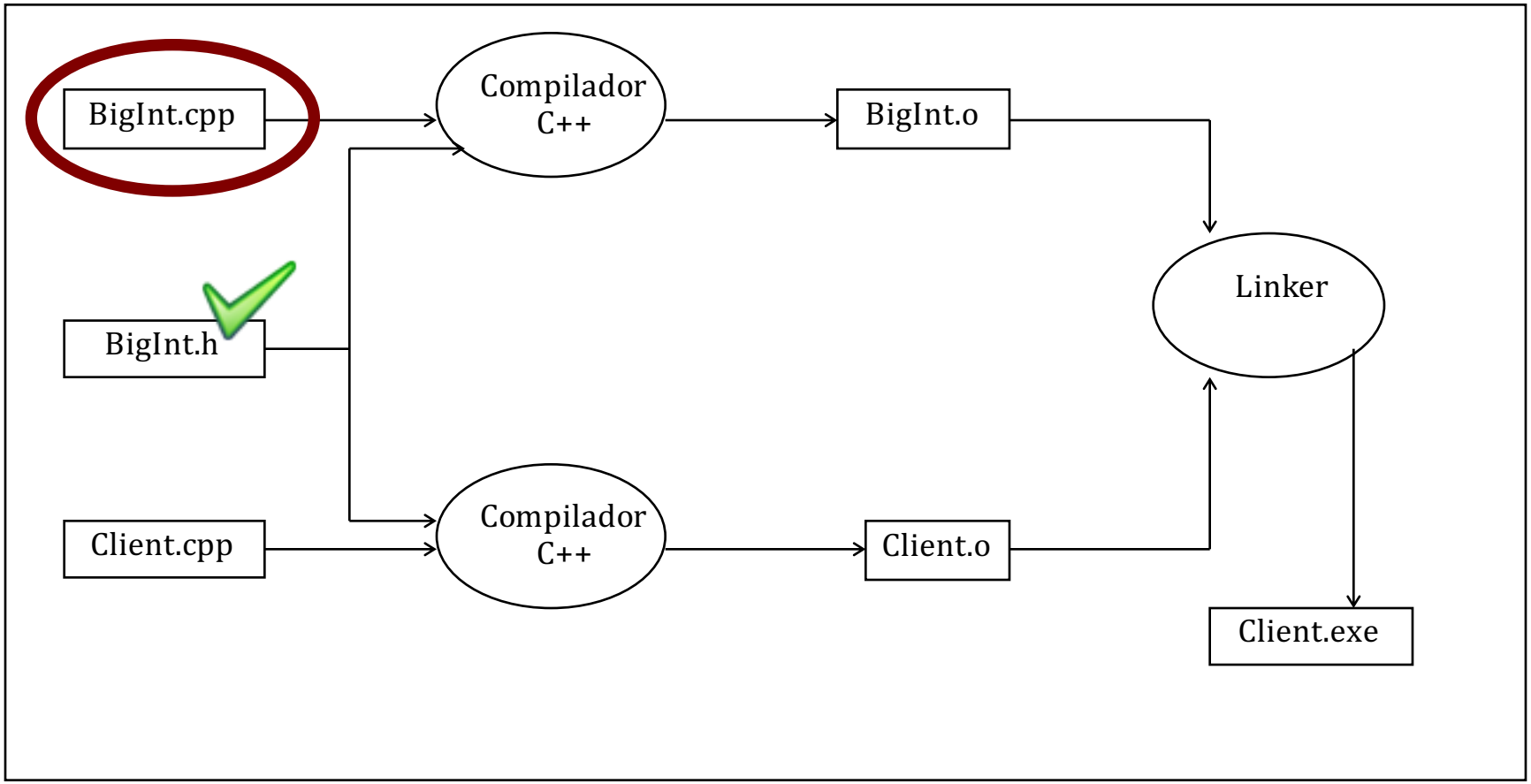
# Destrutores

- Em C++, o desenvolvedor deve destruir as instâncias que criou, não há coletor de lixo;
  - O C++ irá, no entanto, destruir instâncias diretas criadas na pilha.
- Portanto, destrutores são sempre chamados (diferente do Java, que não garante o `finalize()`);
- Destrutores devem liberar memória dinâmica alocada pela classe durante a construção da instância.

```
// No exemplo, o destrutor foi declarado e definido:
~BigInt() { delete digitos; }

// free() no C -> delete no C++.
```

# Arquitetura da solução exemplo

- O arquivo *header* está OK!
- Falta agora a implementação.



[ Curso Rápido de C++ ]

# DEFININDO CLASSES (IMPLEMENTAÇÃO)

# Implementação/definição

- Similar a um TAD, o arquivo de definição deve:
  - Importar o arquivo de declaração (*header*);

```
#include "BigInt.h";
```

  - Prover uma implementação para todas as funções (métodos) declaradas no *header*.
- A declaração de uma classe diz **o que** as instâncias da classe podem fazer (contrato);
- A definição/implementação de uma classe diz **como** as instâncias da classe farão o que encontra-se no contrato.

# Explorando a classe BigInt

- Vamos ver algumas características de C++ presentes na implementação da classe BigInt:

```
BigInt:: BigInt (const char* digitString) {
  unsigned n = strlen(digitString);
  if (n != 0) {
     digitos = new char[ndigitos = n];
     char *p = digitos;
     const char* q = &digitString[n];
     // Converte o digito ASCII para binário.
     while (n--) *p++ = *--q - '0';
  }
  else {
     // String vazia.
     digitos = new char[ndigitos = 1];
     digitos[0] = 0;
  }
}
```

# Resolução de escopo

- A implementação de um método é prefixada com o operador de resolução de escopo (`::`):

```
BigInt:: BigInt (const char* digitString) {
```

- Motivo: diferente de Java, o método não encontra-se obrigatoriamente dentro do escopo da classe;
- Pode-se usá-lo também para variáveis:

```
int var = 0;
int main(void) {
  int var = 0;
  ::var = 1;    // Altera a variável global.
  var = 2;      // Altera a variável local.
}
```

# Constantes

- A palavra-chave `const` define constantes em C++:

```
BigInt:: BigInt (const char* digitString) {
  // ...
  const char* q = &digitString[n];
```

- Sugere-se usá-la ao invés de #define, usado em C;
  - Verificação de tipo, constantes com escopo;
- Há dois tipos. (1) Ponteiro para constante:

```
const char* pc = "Nome"; // Ponteiro para uma constante.
pc[3] = 'a';             // Erro.
pc = "teste";            // OK.
```

# Constantes

- (2) Ponteiro constante:

```
char *const cp = "Nome"; // Ponteiro constante.
cp[3] = 'a';             // OK.
cp = "Teste";            // Erro.
```

- Podemos também misturar os dois, criando um ponteiro constante para constante:

```
// Ponteiro constante para objeto constante.
const char *const cp = "Nome";
cp[3] = 'a';                    // Erro.
cp = "Teste";                   // Erro.
```

- Como esperado, um ponteiro para constante não pode ser atribuído para um ponteiro para não-constante.

# Métodos constantes

- `const` define também métodos constantes:

```
BigInt operator+(const BigInt&) const;

void print(FILE* f = stdout) const;
```

- Não tem nada a ver com métodos finais em Java;
- Um método constante é um método que não modifica o objeto sobre o qual é aplicado, podendo ser chamado em objetos constantes:

```
const BigInt c = "29979250000";
c.print();
const BigInt* cp = &c;
cp->print();
```

# O operador new

- Substitui o `malloc()` de C:

```
// C:
struct BigInt* p;
p = (struct BigInt*) malloc(sizeof(struct BigInt));

// C++:
BigInt* p;
p = new BigInt;    // Sem cast ou sizeof!
```

- Assim como em Java, new chama o construtor da classe, inicializando o objeto;
- Diferente de Java, () não é obrigatório;
- Diferente dos construtores de instâncias diretas vistos anteriormente, o new retorna um ponteiro.

# Explorando a classe BigInt

- Veja a implementação do construtor que recebe um inteiro. Note que não é preciso repetir o valor *default*:

```
BigInt:: BigInt(unsigned n) {
  char d[3 * sizeof(unsigned) + 1];
  char *dp = d;
  ndigitos = 0;
  do {
     *dp++ = n % 10;
     n /= 10;
     ndigitos++;
  } while (n > 0);

  digitos = new char[ndigitos];
  for (int i = 0; i < ndigitos; i++)
     digitos[i] = d[i];
}
```

# Explorando a classe BigInt

- E o construtor cópia:

```
BigInt::BigInt(const BigInt& n) {
  unsigned i = n.ndigitos;
  digitos = new char[ndigitos = i];
  char *p = digitos;
  char *q = n.digitos;
  while (i--) *p++ = *q++;
}
```

- Como em Java, o acesso ao atributo privado `n.digitos` é permitido por estarmos no escopo de `BigInt`.

# Passagem por referência

- O construtor cópia utiliza passagem por referência:

```
BigInt::BigInt(const BigInt& n) {
```

- Em C, a passagem é sempre feita por valor:

```
void inc(int x) { x++; }
int y = 1;
inc(y);
printf("%d\n", y);   // 1
```

- Pode-se contornar passando o valor de um ponteiro:

```
void inc(int* x) { *x++; }
int y = 1;
inc(&y);
printf("%d\n", y);   // 2
```

# Passagem por referência

- Em C++ existe a passagem por referência:

```
void inc(int& x) { x++; }

int y = 1;
inc(y);
printf("%d\n", y);   // 2
```

- Note que:
  - A variável x não precisou ser derreferenciada em `inc()`;
  - A variável y foi passada como argumento normalmente, e não seu endereço &y.

# Referências

- Portanto, além de ponteiros, C++ possui referências:

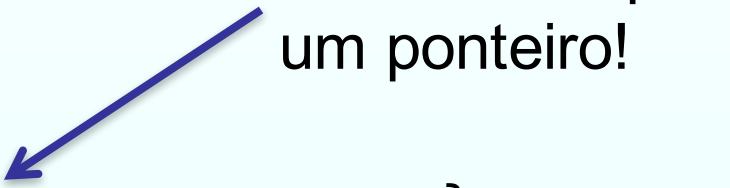
```
int i;
int* p = &i;     // Um ponteiro para i.
int& r = i;      // Uma referência para i.
r++;             // Incrementa o valor de i.
```

- Você pode usar `r` em lugar de `*p` (ou seja, `i`) e `&r` em lugar de `p` (como se fosse `&i`) Porém, referências:
  - Devem ser inicializadas quando declaradas;
  - Não podem apontar para um objeto diferente do qual foram inicializadas.
- Referências são como apelidos, *aliases* para objetos (sintaxe mais segura do que ponteiros).

# Uso do this

- Assim como em Java, `this` pode ser usado para diferenciar entre parâmetro e atributo:

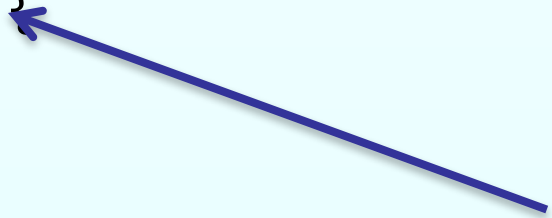
```
class Objeto {
  long attr;

public:
  Objeto(long attr = 0) { this->attr = attr; }
  void setAttr(long attr) { this->attr = attr; }
  void print() { printf("%ld\n", attr); }
};

int main () {
  Objeto c(10);
  c.print();          // 10
  c.setAttr(20);
  c.print();          // 20
}
```



# A classe SeqDigitos

- Para implementar a soma de BigInts, precisaremos de uma classe auxiliar: sequência de dígitos:

```
class SeqDigitos {
  char* dp;      // Ponteiro para o digito corrente.
  unsigned nd;   // Número de dígitos restantes.
public:
  SeqDigitos (const BigInt& n) {
     dp = n.digitos;
     nd = n.ndigitos;
  }
  unsigned operator++(int) {
     if (nd == 0) return 0;
     else {
        nd--;
        return *dp++;
     }
  }
};
```



Sobrescreve apenas o x++, não o ++x!

# Somando dois objetos BigInt

```
BigInt BigInt::operator+(const BigInt& n) const {
  unsigned maxdigitos = (ndigitos > n.ndigitos ?
     ndigitos : n.ndigitos) + 1;
  char* sumPtr = new char[maxdigitos];
  BigInt sum(sumPtr, maxdigitos);
  SeqDigitos a(*this);
  SeqDigitos b(n);
  unsigned i = maxdigitos - 1;
  unsigned carry = 0;
  while (i--) {
     *sumPtr = (a++) + (b++) + carry;
     if (*sumPtr >=10) {
        carry = 1;
        *sumPtr -= 10;
     }
     else carry = 0;
     sumPtr++;
  }                          // Continua...
```

this, como em Java (BigInt *)

# Somando dois objetos BigInt

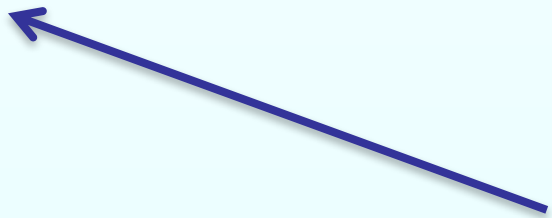
```
  if (carry) {
     *sumPtr = 1;
     return sum;
  }
  else {
     i = maxdigitos - 1;
     char* p = new char[i];
     BigInt s(p, maxdigitos - 1);
     char* q = sum.digitos;
     while (i--) *p++ = *q++;
     return s;
  }
}
```

Novo construtor, deve ser definido como privativo (uso interno). Simplesmente atribui os parâmetros para digitos e ndigitos respectivamente.

# Há algo errado na classe SeqDigitos

- Você reparou que tem algo errado no código abaixo?

```
class SeqDigitos {
  char* dp;      // Ponteiro para o digito corrente.
  unsigned nd;   // Número de dígitos restantes.
public:
  SeqDigitos (const BigInt& n) {
     dp = n.digitos;
     nd = n.ndigitos;
  }
  unsigned operator++( ) {
     if (nd == 0) return 0;
     else {
        nd--;
        return *dp++;
     }
  }
};
```



Os atributos digitos e ndigitos são privativos de BigInt. SeqDigitos não consegue acessá-los!

# Classes/métodos amigos

- Para permitir o acesso, `BigInt` pode declarar `SeqDigitos` (ou seu construtor) como amiga:

```
class BigInt {
  // ...

  // Declarando a classe toda como amiga:
  friend class SeqDigitos;

  // Declarando somente o construtor como amigo:
  friend SeqDigitos::SeqDigitos(const BigInt&);

  // ...
};
```

# Construtor cópia vs. sobrescrita do =

```
BigInt::BigInt(const BigInt& n) {
  unsigned i = n.ndigitos;
  digitos = new char[ndigitos = i];
  char* p = digitos;
  char* q = n.digitos;
  while (i--) *p++ = *q++;
}

void BigInt::operator=(const BigInt& n) {
  if (this == &n) return;      // Para o caso x = x;
  delete digitos;              // Porque this já existe!
  unsigned i = n.ndigitos;
  digitos = new char[ndigitos = i];
  char* p = digitos;
  char* q = n.digitos;
  while (i--) *p++ = *q++;
}
```

# Atribuições múltiplas de BigInt

- Da forma como foi declarado/definido, não é possível fazer atribuições múltiplas de BigInt:

```
void BigInt::operator=(const BigInt& n) { /* ... */ }

// No cliente:
BigInt c = 243;
BigInt a, b;
a = b = c;      // Erro! (b = c) retorna void!
```

- Para isso, é necessário retornar um valor:

```
BigInt& BigInt::operator=(const BigInt& n) {
  /* ... */
  return *this;
}
```

# Retorno de objetos/ponteiros/referências

- É perfeitamente possível;
- Ao retornar objetos, o construtor de cópia da referida classe é chamado. Ex.: `BigInt f() { return 0; };`
- Ao retornar ponteiros e referências, deve-se ter o cuidado de garantir que o referido objeto sobreviva:

```
Ponto* f() {
  Ponto a;
  // ...
  return &a;  // Objeto apontado será desalocado!
}

Ponto& f() {
  Ponto a;
  // ...
  return a;   // Objeto referido será desalocado!
}
```

# O Operador delete

- Como já visto, `delete` é o contrário de `new` e libera a memória alocada para um objeto (como `free` em C);
- `delete` aciona o destrutor da classe:

```
BigInt* a = new BigInt("9834567843");
// [...] Chama a->~BigInt();
delete a;
```

- Ao liberar vetores, é preciso usar colchetes:

```
BigInt* b = new BigInt[10];
delete b;        // Erro!
delete[] b;      // OK!
```

# Funções *inline*

- Assim como os `#defines` de C, funções *inline* são copiadas para o local de suas chamadas na compilação;
- Aumento de desempenho por evitar chamada de função;
- Podem ser definidas de duas maneiras:
  - Definindo sua implementação no arquivo de definição (*header*), como feito com `~BigInt();`
  - Usando a palavra reservada `inline`.
- Deve-se ter cuidado ao definir funções *inline*. Recomenda-se usar apenas para funções bem simples.

[ Curso Rápido de C++ ]

# MELHORIAS EM RELAÇÃO A C

# Strings

- A classe string esconde a complexidade de lidar com vetores de caracteres:

```
#include <string>
using namespace std;

int main () {
  string s1 = "Um", s2("Dois");
  if (s1 != s2) {
     int tam = s1.length();
     s2.clear();

     if (s2.empty()) {
        s2 = s1 + " mais Um";
        char c = s2.at(0);
        const char *cStr = s1.c_str();
     }
  }
}
```

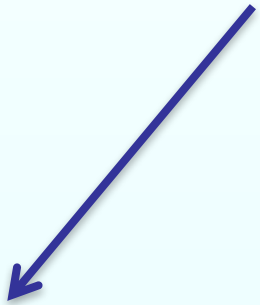
Veja mais em:
**http://www.cplusplus.com/reference/string/**

# Booleanos

- C++ adiciona o tipo booleano: `bool`.

```
int main () {
  string resposta;
  bool resultado = true;

  while (resultado) {
     cout << "Deseja sair (S/N)? ";
     getline(cin, resposta);
     int cmp = resposta.compare("S");
     resultado = cmp;

     // Altenativamente:
     // resultado = (resposta == "N");
  }
}
```

Conversão de `int` para `bool` automática



# Fluxos

- C++ facilita a entrada e saída de dados por meio dos fluxos (*streams*) cout e cin:

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

int main () {
  cout << "Hello, world!\n";
  cout << "Um inteiro: ";
  cout << 10;
  cout << " e um float: " << 10.5 << '\n';

  int valor;
  cout << "Digite um valor: ";
  cin >> valor;
  cout << "O dobro é: " << valor * 2 << '\n';
}
```

# Fluxos

- A constante endl representa quebra de linha ('\n'):

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

int main () {
  cout << "Hello, world!\n";
  cout << "Um inteiro: ";
  cout << 10;
  cout << " e um float: " << 10.5 << endl;

  int valor;
  cout << "Digite um valor: ";
  cin >> valor;
  cout << "O dobro é: " << valor * 2 << endl;
}
```

# Fluxos e *strings*

- Fluxos leem dados até o próximo espaço em branco (como um *scanner* padrão em Java):

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

int main () {
  char nome[50];
  cout << "Qual é o seu nome?\n";
  cin >> nome;
  cout << "Olá, " << nome << "!\n";
}

// Problema:
// Vítor          -> "Olá, Vítor!"
// Vítor Souza    -> "Olá, Vítor!"
```

# Fluxos e *strings*

- Melhor usar `getline()` da biblioteca de *strings*:

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

int main () {
  string nome;
  cout << "Qual é o seu nome?\n";
  getline(cin, nome);
  cout << "Olá, " << nome << "!\n";
}

// OK!
// Vítor          -> "Olá, Vítor!"
// Vítor Souza    -> "Olá, Vítor Souza!"
```

# Uso de espaços de nome

- A declaração `using namespace` é útil quando se usa bibliotecas de função do C++;
- Tais bibliotecas são declaradas dentro do espaço de nomes `std`. O comando evita a repetição de `std::`.

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

int main () {
  std::string nome;
  std::cout << "Qual é o seu nome?\n";
  std::getline(std::cin, nome);
  std::cout << "Olá, " << nome << "!\n";
}
```

[ Curso Rápido de C++ ]

# CONCEITOS OO

# Herança em C++: classes derivadas

```
class Pessoa {
  string nome;
  int idade;

public:
  Pessoa (string n, int a) {nome = n; idade = a;}
  string& getNome() const { return nome; };
  void mudarIdade(int a) { idade = a; }
  virtual void print( );
};

// Um empregado é uma pessoa.
class Empregado: public Pessoa {};
```

# Herança em C++: classes derivadas

- Note a sintaxe do construtor de Empregado, chamando o construtor de Pessoa (em Java usaríamos super):

```
// Um empregado é mais que uma pessoa!
class Empregado: public Pessoa {
  float salario;

public:
  Empregado(string n, int a, float s) :
     Pessoa (n, a) { salario = s; }

  void mudarSalario(float r)     { salario = r; }
  void print();

  // Implementação de mudarIdade() é herdada.
};
```

# Construtores e composição

- Esta mesma sintaxe pode ser usada no caso de composição, para inicialização:

```
class Ponto{
  int xc, yc;
public:
  Ponto (int x, int y) { xc = x; yc = y; }
  int getXC() const { return xc; }
  int getYC() const { return yc; }
};

class Circulo {
  Ponto org; int raio;
public:
  Circulo(const Ponto& c, int r): org(c.getXC(),
       c.getYC()) { raio = r; }
};
```

Neste caso obrigatório, pois Ponto não possui construtor sem parâmetros.

# Funções virtuais / sobrescrita

- Ao contrário de Java, em C++ a ligação tardia (*late binding*) não é feita por padrão:

```
class Pessoa {
  // ...
  void print() { cout << "Uma pessoa\n"; }
};
class Empregado: public Pessoa {
  // ...
  void print() { cout << "Um empregado\n"; }
};
main() {
  Pessoa* p = new Pessoa();
  Empregado* e = new Empregado();
  p->print();                          // Uma pessoa
  e->print();                          // Um empregado
  delete p; p = e;
  p->print();                          // Uma pessoa!
}
```

# Funções virtuais / sobrescrita

- Para “ativar” a ligação tardia, é preciso marcar o método como virtual (na primeira declaração):

```
class Pessoa {
  // ...
  virtual void print() { cout << "Uma pessoa\n"; }
};
class Empregado: public Pessoa {
  // ...
  void print() { cout << "Um empregado\n"; }
};
main() {
  Pessoa* p = new Pessoa();
  Empregado* e = new Empregado();
  p->print();                          // Uma pessoa
  e->print();                          // Um empregado
  delete p; p = e;
  p->print();                          // Um empregado!
}
```

# Funções virtuais / sobrescrita

- Funcionam somente com ponteiros e referências;
- Independente de ser virtual, na sobrescrita o método da superclasse pode ser acessado:

```
class Pessoa {
  // ...
  virtual void print() {
     cout << nome << ", " << idade;
  }
};

class Empregado: public Pessoa {
  // ...
  void print() {
     Pessoa::print(); cout << ", " << salario;
  }
};
```

# Herança pública e privativa

- Na herança pública, os membros públicos da superclasse são também públicos na subclasse;
- Na herança privativa os membros públicos da superclasse são privativos na subclasse;
  - E, portanto, não disponível para os clientes da subclasse.

```
// Herança pública:
class EmpregadoA: public Pessoa { /* ... */ };

// Herança privativa:
class EmpregadoB: Pessoa { /* ... */ };
```

# Herança privada e polimorfismo

- Como se pode concluir, herança privada impede o polimorfismo:

```
void imprimir(Pessoa *p) {
  cout << p->getNome();
}

int main () {
  Pessoa* p = new Pessoa("Fulano", 32);
  EmpregadoA* a = new EmpregadoA("Ana", 32, 2000.00);
  EmpregadoB* b = new EmpregadoB("Bruno", 32, 2000.00);
  imprimir(p);  // OK!
  imprimir(a);  // OK!
  imprimir(b);  // 'Pessoa' is an inaccessible base
}
```

# Bom uso da herança privada

- Quando queremos reutilizar membros, porém a subclasse não É UMA derivação da superclasse:

```
class Lista {
  /* ... */

public:
  void insereInicio(Info valor);
  Info removeInicio();
  int estaVazia();
};

class Pilha: Lista {
public:
    void empilha(Lista::Info val) { insereInicio(val); }
    Lista::Info desempilha() { return removeInicio(); }
};
```

# Herança seletiva

- Se quiser herdar apenas alguns métodos como públicos, é preciso repeti-los manualmente:

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

class Pai {
public:
  void um() { cout << "1" << endl; };
  void dois() { cout << "2" << endl; };
  void tres() { cout << "3" << endl; };
};

class Filho: Pai {
public:
  void dois() { Pai::dois(); }
};
```

# Classes abstratas (incompletas)

- São criadas a partir do que C++ chama de “funções virtuais puras”, que não possuem implementação:

```
class Forma {
public:
  virtual void mover(const Ponto&) = 0;
  virtual void desenhar() const = 0;
};
```

- Assim como Java, o compilador não permite criação de instâncias da classe incompleta;
- Diferente do Java, caso a subclasse não mencione o método, ele é herdado como virtual puro (e a classe é automaticamente abstrata).

# Ordem de construção / destruição

- C++ inicializa instâncias da seguinte maneira:
  - Se existe classe base, inicializa suas instâncias primeiro;
  - Em seguida, inicializa os atributos da classe na ordem em que são declaradas;
- Observe que:
  - Atributos constantes devem ser inicializados obrigatoriamente no construtor;
  - Não se pode inicializar um atributo não-*static* diretamente na sua declaração.
- Ao destruir uma instância, C++ destrói seus atributos na ordem inversa que eles foram construídos.

# Construtores/operadores pré-definidos

- Para classes cujos atributos não utilizam alocação dinâmica de memória (new), não é necessário definir:
  - Construtor de cópia;
  - Operador de atribuição;
  - Construtor *default*.
- Os mesmos são pré-definidos em C++, existem para todas as classes;
- Construtor de cópia e operador de atribuição realizam a cópia membro a membro de uma instância à outra;
- Assim como em Java, o construtor *default* é provido somente quando não há nenhum construtor declarado.

# O acesso protegido

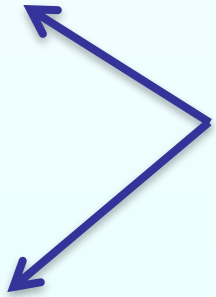
- Já vimos 2 especificadores de acesso:
  - `private`: somente a própria classe e seus amigos podem acessar;
  - `public`: qualquer um pode acessar.
- Assim como em Java, existe em C++ o acesso protegido:
  - `protected`: somente a própria classe, seus amigos, as classes derivadas e seus amigos podem acessar.

```
class Pessoa {
protected:
  string nome;
  int idade;

public:
  /* ... */
};
```

# Herança múltipla

- C++, ao contrário de Java, permite a herança múltipla:

```
class Surf {
  time_t data;
  float notas[10];
public:
  void registrarNota(float nota);
};

class Vela {
  time_t data;
  float tempos[10];
public:
  void registrarTempo(float tempo);
};

class WindSurf : public Surf, public Vela {
  char competicao;
};
```



Subclasse herda ambos os métodos

# Ambiguidades da herança múltipla

- Colisão de nomes:

```
class Surf {
  time_t data;
  float notas[10];
public:
  virtual void ordena();
};

class Vela {
  time_t data;
  float tempos[10];
public:
  virtual void ordena();
};

class WindSurf : public Surf, public Vela {
  char competicao;
};
```

Não há colisão de atributos se os mesmos forem privativos

Colisão de nomes gerando ambiguidade.

# Ambiguidades da herança múltipla

- Programador deve resolver ambiguidade “na mão”:

```
int main () {
  WindSurf ws;
  ws.Surf::ordena();
}
```

- Ou:

```
void WindSurf::ordena() {
  // N = competição por nota; T = por tempo.
  if (competicao == 'N' )
     Surf::ordena();
  else
     Vela::ordena();
}
```

# Ambiguidades da herança múltipla

- Herança repetida:

```
class Competicao {
  float premio;
public:
  virtual float getPremio() { return premio; };
};

class Surf: public Competicao { /* ... */ };
class Vela: public Competicao { /* ... */ };

// Herda getPremio() duas vezes:
class WindSurf : public Surf, public Vela { /* ... */ };

int main() {
  WindSurf ws;
  cout << ws.getPremio(); // request is ambiguous
}
```

# Ambiguidades da herança múltipla

- Resolve-se usando a palavra-chave `virtual`:

```
class Competicao {
  float premio;
public:
  virtual float getPremio() { return premio; };
};

class Surf: virtual public Competicao { /* ... */ };
class Vela: virtual public Competicao { /* ... */ };

class WindSurf : public Surf, public Vela { /* ... */ };

int main() {
  WindSurf ws;
  cout << ws.getPremio(); // OK!
}
```

Obs.: herança virtual NÃO resolve colisão de nomes!

[ Curso Rápido de C++ ]

# CONCEITOS AVANÇADOS

# A palavra-chave `static`

- Relembrando: `static` em C:
  - Quando em funções, mantém seu valor mesmo quando o escopo da função termina;
  - Quando globais, marca a variável como escopo de arquivo ao invés de escopo global.
- Em C++:
  - Atributos estáticos funcionam como em Java (atributos da classe, não das instâncias);
  - Métodos estáticos idem.

# A palavra-chave static

- Inicialização é um pouco mais complexa que Java:

```
#include <iostream>

class GeradorId {
  // Declara a variável na interface:
  static int proxId;

public:
  static int getId() { return proxId++; }
};

// Define a variável na implementação:
int GeradorId::proxId = 1;

int main () {
  for (int i = 0; i < 10; i++)
    std::cout << GeradorId::getId() << endl;
}
```

# Inicialização

- Atenção, pois funciona de forma diferente de Java:

```
class Classe {
  // ISO C++ forbids initialization of member ‘atributo’
  int atributo = 100;
};
```

- Atributos devem ser inicializados sempre nos construtores da classe;
- Atributos `static` são inicializados como visto no slide anterior.

# Classes internas

- Assim como Java, C++ permite a definição de classes internas, especificando seu nível de acesso:

```
class Lista {
public:
  class Info {
     int info;
  public:
     Info(int info = 0) { this->info = info; }
  };

private:
  class No {
     No* prox;
     Info info;
  };
  No* prim;

  // Continua...
```

# Classes internas

```
public:
  Lista() { prim = nullptr; }
  void insereInicio(Info valor);
  void insereFim(Info valor);
  Info removeInicio();
  int estaVazia();
};

// Assumindo que Lista é implementada em algum lugar...
int main () {
  Lista::Info info(10);
  Lista lista;
  lista.insereInicio(info);

  // class Lista::No' is private:
  // Lista::No no;
}
```

# *Templates*: classes genéricas

- Assim como Java (> 5), C++ possui um mecanismo para definição de funções e classes genéricas:

```
template <class T>
class Lista {
  class No {
     No* prox;   T info;
  };
  No* prim;
public:
  Lista() { prim = NULL; }
  void insereInicio(T valor);
  void insereFim(T valor);
  T removeInicio();
  int estaVazia();
};

template <class T>
void Lista<T>::insereInicio(T valor) { /* ... */ }
```

# Outro exemplo: Casulo.h

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

#ifndef CASULO_H_
#define CASULO_H_

template <class T>
class Casulo {
  T conteudo;
public:
  Casulo(T conteudo) { this->conteudo = conteudo; }
  void imprimir() const;
};

template <class T>
void Casulo<T>::imprimir() const {
  cout << conteudo << endl;
}
#endif
```

# Outro exemplo: main.cpp

```
#include <iostream>
#include <string>
#include "Casulo.h"
using namespace std;

void test() {
  Casulo<string> cs("Hello, world!");
  Casulo<int> ci(10);
  Casulo<double> cd(3.14);
  Casulo<Casulo<int>*> cp(&ci);

  cs.imprimir();   ci.imprimir();
  cd.imprimir();   cp.imprimir();
}
```

A classe Casulo não foi dividida entre .h e .cpp: isso é uma limitação do mecanismo de *templates* (em alguns compiladores). O código-fonte é reutilizado, e não o código objeto!

# Espaços de nome

- À medida que os códigos crescem em tamanho, aumenta a chance de colisão de nomes;
- Para evitá-lo, C++ oferece o mecanismo de *namespaces*;
  - Dois nomes iguais em espaços de nomes diferentes não colidem;
  - A API padrão do C++ é declarada no espaço de nome `std`, já visto anteriormente.
- Define-se um *namespace* como se define classes:

```
namespace mylib {
  /* ... */
}
```

# Espaços de nome

- Ao contrário de classes, a redefinição de um *namespace* (ex.: em outro arquivo) continua a definição anterior;
- É possível criar apelidos para *namespaces*:

```
namespace VitorSouzaLibrary {
  void hello() { cout << "Hello, world!\n"; }
}
namespace vsl = VitorSouzaLibrary;

int main () {
  VitorSouzaLibrary::hello();
  vsl::hello();
}
```

- Como já visto, podemos usar também a diretiva:
  `using namespace <nome-do-namespace>;`

# Modularização com *namespaces*

- Dividindo em “pacotes” estilo Java:

```
#include <string>
using namespace std;
#ifndef DEPARTAMENTO_H_
#define DEPARTAMENTO_H_
#include "Funcionario.h"

namespace exercicio02 {
  class Departamento {
     string nome;
     Funcionario* funcionarios[100];
     int numFuncs;
  public:
     /* ... */
  };
}

#endif
```

Como em C, #define é usado para evitar declaração repetida.

“Pacote” é definido usando um espaço de nome.

# Modularização com *namespaces*

```
#include <string>
using namespace std;
#ifndef FUNCIONARIO_H_
#define FUNCIONARIO_H_
namespace exercicio02 {
  class Departamento;

  class Funcionario {
     string nome;
     double salario;
     string dataAdmissao;
     Departamento* departamento;
     friend class Departamento;

  public:
     /* ... */
  };
}
#endif
```

Não podemos aqui incluir "Departamento.h" pois gera inclusão circular. Portanto, fazemos uma declaração antecipada (*forward declaration*).

# Modularização com *namespaces*

```
#include <iostream>
#include "Departamento.h”
namespace exercicio02 {
  Departamento::Departamento(const char* nome) {
     /* ... */
  }
  /* ... */
}
```

```
#include <iostream>
#include "Departamento.h”

namespace exercicio02 {
  Funcionario::Funcionario(const char* nome, double
                salario, const char* dataAdmissao) {
     /* ... */
  }
  /* ... */
}
```

Departamento.h inclui Funcionario.h.

Artigo interessante sobre o assunto:
http://www.cplusplus.com/forum/articles/10627/

# I/O em arquivos

- Segue o mesmo esquema de fluxos de cin/cout:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

const int MAX = 10;

int main () {
  ofstream out("arq.txt");
  for (int i = 0; i < MAX; i++) out << i << endl;
  out.close();

  ifstream in("arq.txt");
  string linha;
  while (getline(in, linha)) cout << linha << endl;
  in.close();
}
```

[ Curso Rápido de C++ ]

# TRATAMENTO DE EXCEÇÕES

# Exceções em C++

- Muito similares às exceções em Java:

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

class Excecao {
  string motivo;
public:
  Excecao(const string& motivo) { this->motivo=motivo; }

  friend ostream& operator<< (ostream &out, const
Excecao& excecao);
};

ostream& operator<< (ostream &out, const Excecao&
excecao) {
    return out << excecao.motivo;
}
```

# Exceções em C++

```
class ExcecaoA: public Excecao {
public:
  ExcecaoA(const string& motivo): Excecao(motivo) {};
};

class ExcecaoB: public Excecao {
public:
  ExcecaoB(const string& motivo): Excecao(motivo) {};
};

void lancarA() {
  throw ExcecaoA("lancarA() foi chamado");
}

void lancarB() {
  throw ExcecaoB("lancarB() foi chamado");
}
```

# Exceções em C++

```
int main () {
  try {
     lancarA();
     lancarB();
  }
  catch (Excecao e) {
     cout << e << endl;
  }
}
```

- Note que:
  - Qualquer objeto pode ser lançado (inclusive de classes que já existam);
  - São lançadas instâncias diretas. O programa automaticamente desaloca a memória delas.

# A biblioteca <exception>

- A biblioteca <exception> define tipos (incluindo a classe base exception) e utilitários para exceções.

```
#include <iostream>
#include <typeinfo>
#include <exception>
using namespace std;

class Teste { virtual void metodo(){} };
int main () {
  try {
    Teste *t = 0;
    typeid(*t);  // Exceção (bad_typeid)!
  }
  catch (std::exception& e) {
    std::cerr << "Capturada: " << e.what() << endl;
  }
}
```

# Omitindo a variável

- No `catch()`, não é necessário especificar uma variável que representa a exceção lançada se a mesma não for utilizada:

```
int main () {
  try {
     lancarA();
     lancarB();
  }
  catch (Excecao) {
     cout << "Uma exceção foi lançada...\n";
  }
}
```

# Captura genérica

- Ao contrário de Java, C++ não restringe o lançamento de instâncias de Throwable;
- Para capturar qualquer tipo de exceção, portanto, usamos o símbolo `...`:

```
int main () {
  try {
     lancarA();
     lancarB();
  }
  catch (ExcecaoB) {
     cout << "Uma exceção B foi lançada...\n";
  }
  catch (...) {
     cout << "Uma outra exceção foi lançada...\n";
  }
}
```

# Relançamento

- Funciona como em Java, porém não é necessário especificar a instância que está sendo relançada:

```
int main () {
  try {
     try {
        lancarA();
     }
     catch (Excecao e) {
        cout << "1o nivel: " << e << endl;
        throw;    // throw e; também funciona!
     }
  }
  catch (Excecao e) {
     cout << "2o nivel: " << e << endl;
  }
}
```

# Tratamento da exceção

- Assim como em Java, quando uma exceção é lançada, tenta-se localizar seu tratamento no contexto atual;
- Caso não seja encontrado, a exceção é lançada para o contexto imediatamente externo e assim por diante;
- Diferente de Java, não é necessário declarar que métodos podem lançar exceção;
- O programa como um todo será abortado em 2 casos:
  - Se a exceção chegar até ao método `main()` e não for tratada;
  - Se uma exceção for lançada e, antes que seja tratada, outra exceção for lançada (por exemplo, no construtor da primeira exceção).

[ Curso Rápido de C++ ]

# UTILITÁRIOS

# C++ Standard Library

- Duas partes principais:
  - Standard Function Library: funções independentes, não fazem parte de classes;
  - Object Oriented Class Library: classes utilitárias.
- A Function Library contém funções para I/O, manipulação de strings, matemática, data/hora, etc.
- A OO Library contém classes para I/O, manipulação de *strings* e números, coleções, tratamento de exceções, etc.

# C++ STL

- A Standard Template Library (STL) reúne classes que utilizam o mecanismo de *templates* (tipos genéricos);
- Em seu núcleo, há três elementos principais:
  - Coleções (*Containers*): estruturas de dados para armazenamento de outros elementos;
  - Algoritmos: inicialização, busca, ordenação e transformação de coleções (`<algorithm>`);
  - Iteradores: navegação em containers (`<iterator>`).

# Containers

- `<array>`: vetor de tamanho fixo;
- `<deque>`: fila duplamente encadeada;
- `<list>`: lista encadeada;
- `<map>`: mapeamento (chave x valor);
- `<queue>`: fila e fila com prioridade;
- `<set>`: conjunto;
- `<stack>`: pilha;
- `<vector>`: vetor de tamanho variável.

# Exemplo: a classe Vector

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

int main() {
  vector<int> vec;
  int i;

  for (i = 0; i < 5; i++) vec.push_back(i);
  cout << "Tamanho = " << vec.size() << endl;
  for (i = 0; i < 5; i++)
     cout << "vec[" << i << "] = " << vec[i] << endl;

  vector<int>::iterator v = vec.begin();
  while (v != vec.end()) {
     cout << "v = " << *v << endl;
     v++;
  }
}
```

# Outras bibliotecas interessantes

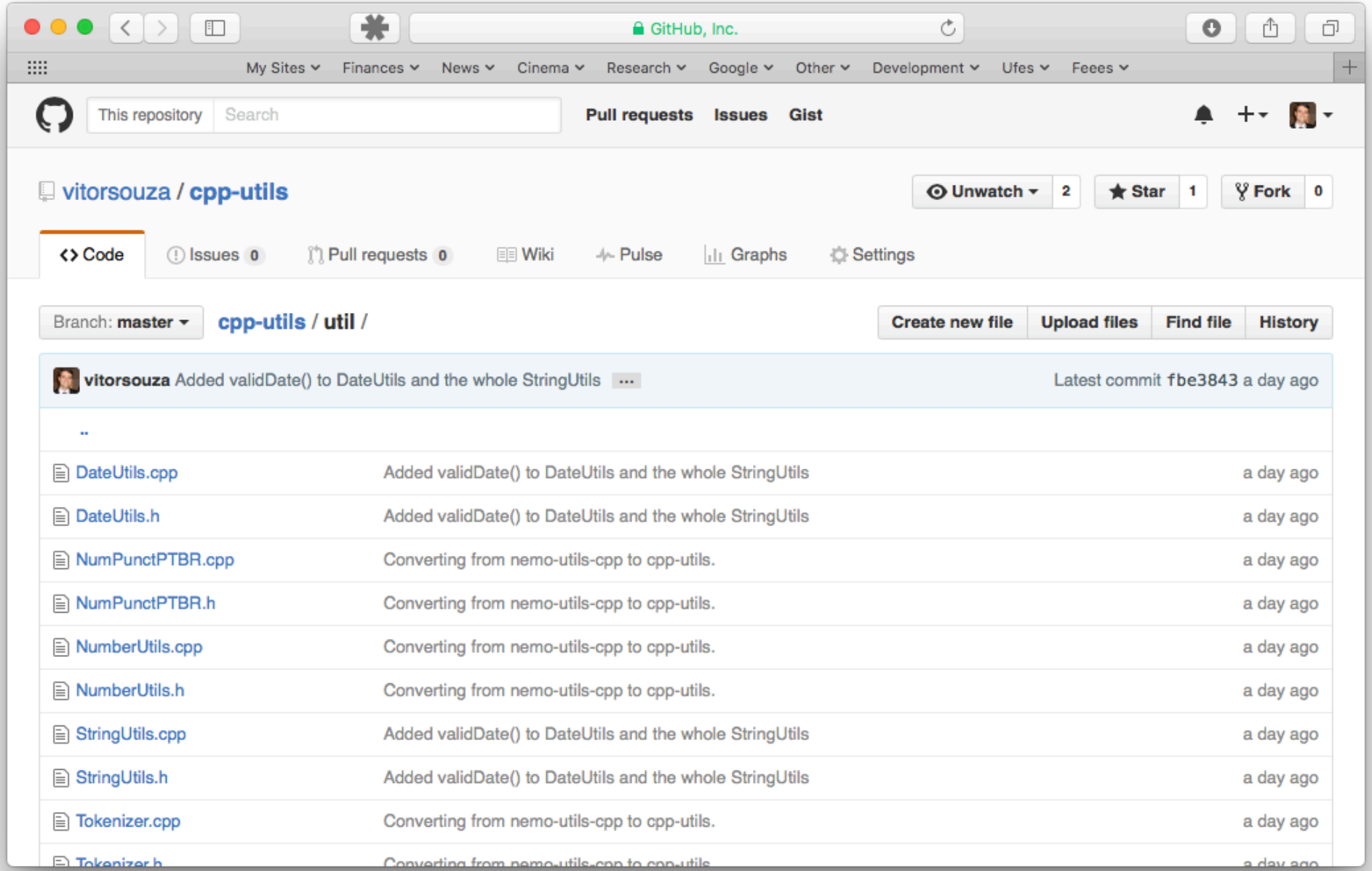
- <chronos>: manipulação de valores temporais (a partir de C++11, gcc >= 4.5);
- <complex>: para lidar com números complexos;
- <locale>: localização (internacionalização);
- <numeric>: operações em sequências de valores numéricos;
- <random>: geração de números pseudoaleatórios;
- <regex>: expressões regulares;
- Dentre outras...

Novamente,
**http://www.cplusplus.com/reference/**
é seu amigo...

[ Curso Rápido de C++ ]

# FAQ PARA PROGRAMADORES JAVA

# cpp-utils


GitHub, Inc.
My Sites
Finances
News
Cinema
Research
Google
Other
Development
Ufes
Feees
This repository
Search
Pull requests
Issues
Gist
vitorsouza / cpp-utils
Unwatch 2
Star 1
Fork 0
Code
Issues 0
Pull requests 0
Wiki
Pulse
Graphs
Settings
Branch: master
cpp-utils / util /
Create new file
Upload files
Find file
History
vitorsouza Added validDate() to DateUtils and the whole StringUtils
Latest commit fbe3843 a day ago
..
DateUtils.cpp Added validDate() to DateUtils and the whole StringUtils a day ago
DateUtils.h Added validDate() to DateUtils and the whole StringUtils a day ago
NumPunctPTBR.cpp Converting from nemo-utils-cpp to cpp-utils. a day ago
NumPunctPTBR.h Converting from nemo-utils-cpp to cpp-utils. a day ago
NumberUtils.cpp Converting from nemo-utils-cpp to cpp-utils. a day ago
NumberUtils.h Converting from nemo-utils-cpp to cpp-utils. a day ago
StringUtils.cpp Added validDate() to DateUtils and the whole StringUtils a day ago
StringUtils.h Added validDate() to DateUtils and the whole StringUtils a day ago
Tokenizer.cpp Converting from nemo-utils-cpp to cpp-utils. a day ago

# ParseException (datas)

- Como faço pra verificar que uma string contém uma data válida (ParseException em Java)?

```
// Já implementado no DateUtils:
bool validDate(const string& str, const string& formato)
{
    struct tm tm;
    return strptime(str.c_str(), formato.c_str(), &tm);
}
```

# ParseException (números)

- Como faço pra verificar que uma string contém um número inteiro válido (ParseException em Java)?

```
// Já implementado no StringUtils:
bool isNumber(string& s) {
    if (s.empty()) return false;
    for (int i = 0; i < s.size(); i++)
        if (! isdigit(s.at(i))) return false;
    return true;
}
```

A função poderia ser adaptada para aceitar também números reais, aceitando também o separador (vírgula ou ponto).

# trim()

- Como fazer java.lang.String.trim() em C++?

```
// Já implementado no StringUtils (from Stack Overflow):
string& ltrim(string &s) {
    s.erase(s.begin(), find_if(s.begin(), s.end(),
not1(ptr_fun<int, int>(isspace))));
    return s;
}

string& rtrim(string &s) {
    s.erase(find_if(s.rbegin(), s.rend(),
not1(ptr_fun<int, int>(isspace))).base(), s.end());
    return s;
}

string& trim(string& s) {
    return ltrim(rtrim(s));
}
```

# Collator

- Como faço comparação de strings com Collator (para considerar letras com acentos)?

```
// Já implementado no StringUtils:
bool stringCompare(string s1, string s2) {
    const collate<char>& col =
        use_facet<collate<char> >(locale());
    transform(s1.begin(), s1.end(), s1.begin(),
        ::tolower);
    transform(s2.begin(), s2.end(), s2.begin(),
        ::tolower);
    const char* pb1 = s1.data();
    const char* pb2 = s2.data();
    return (col.compare(pb1, pb1 + s1.size(), pb2, pb2 +
        s2.size()) < 0);
}
```

# Collections.sort()

- Como faço para ordenar um vetor de objetos, tipo java.util.Collections.sort()?
  - Função sort() da biblioteca <algorithm>;
  - Parâmetros: iterador início, iterador fim, função de comparação;
  - Para ordernar um vector v<T> inteiro: v.begin(), v.end();
  - Função comparação: bool f(const T, const T).

# Collections.sort()

- Exemplo: classe Emprestimo (tomador, data). Para ordenar um vector<Emprestimo*>:

```
// Dentro de uma função qualquer:
sort(emprestimos.begin(), emprestimos.end(),
comparaEmprestimos);

// Em algum outro ponto, defina a função:
bool comparaEmprestimos(const Emprestimo* esq, const
Emprestimo* dir) {
    int diff = difftime(esq->getDataEmprestimo(),
        dir->getDataEmprestimo());
    if (diff != 0) return (esq != dir) && (diff >= 0);
    return stringCompare(esq->getTomador(),
        dir->getTomador());
}
```

# TreeSet & TreeMap

- Como faço coleções ordenadas, tipo java.util.TreeSet e java.util.TreeMap?
  - As classes-template set e map já são ordenadas;
  - Tipos genéricos a instanciar: classe do elemento que será armazenado, classe comparadora;
  - Diferentemente do sort(), é necessária uma classe comparadora, não uma função comparadora.
- Exemplo: um mapa que associa pessoas (atores, diretores) às mídias em que participam:

```
map<Pessoa*, set<Midia*, MidiaComparator>,
PessoaComparator> producoes;
```

# TreeSet & TreeMap

```
// Declaração:
class MidiaComparator {
public:
    bool operator()(const Midia*, const Midia*) const;
};

class PessoaComparator {
public:
    bool operator()(const Pessoa*, const Pessoa*) const;
};

// Definição (PessoaComparator é equivalente):
bool MidiaComparator::operator()(const Midia* esq,
                                 const Midia* dir) const {
    return (esq != dir) && stringCompare(esq->nome,
          dir->nome);
}
```

MidiaComparator deve ser amiga de Midia!

# FileNotFoundException

- Como faço pra lançar uma exceção tipo java.io.FileNot-FoundException quando um arquivo não existe?

```
ifstream in(nomeArquivo);
if (! in.good()) {
  in.close();
  throw FileNotFoundException();   // Precisa defini-la!
}

// Restante do código de leitura, normal...
```

# Map: contains() e put()

- Como faço para verificar se um elemento existe num mapa? Ou para substitui-lo?
  - map.get(chave) em C++ lança exceção se o elemento não existe (diferente de Java, que retorna nulo);

```
if (mapa.count(chave) == 0) { /* Faça alguma coisa. */ }
```

  - map.add(chave, valor), diferente do put() em Java, adiciona um outro valor à chave, caso já tenha um;
  - map.count(chave) passa a ser 2...
  - Para substituir, use find() e iterator. Exemplo:

```
map<Genero*, int>::iterator itGen = qtdGen.find(genero);
if (itGen != qtdGen.end()) itGen->second += 1;
```

# Outras molezas...

```
#include <ctime>
#include <iostream>
#include "util/DateUtils.h"
#include "util/NumberUtils.h"
#include "util/Tokenizer.h"

using namespace std;
using namespace cpp_util;

int main() {
  // Parses a date (string -> time_t) using
  // the pt-BR locale.
  string data = "15/07/2014";
  cout << validDate(data, DATE_FORMAT_PT_BR_SHORT) <<
endl;
  time_t hoje = parseDate(data, DATE_FORMAT_PT_BR_SHORT);
  cout << hoje << endl << endl;
```

# Outras molezas...

```
  // Adds 30 hours to the date and formats it
  // back to string.
  hoje += 30 * 60 * 60;
  cout << formatDate(hoje, DATE_FORMAT_PT_BR_SHORT) <<
endl << endl;

  // Parses a decimal number using the pt-BR locale.
  string numero = "935,79";
  double d = parseDouble(numero, LOCALE_PT_BR);
  cout << d << endl << endl;

  // Multiplies the number by 2 and formats it back to
  // string, normal and currency styles.
  d *= 2;
  cout << formatDouble(d, LOCALE_PT_BR) << endl;
  cout << formatDoubleCurrency(d, LOCALE_PT_BR) << endl
<< endl;
```

# Outras molezas...

```
  // Using the Tokenizer to read tokens from a string
  // separated by ';'.
  string linha = "Um;Dois;Três;Quatro;Cinco";
  Tokenizer tok(linha, ';');
  while (tok.hasNext()) cout << tok.next() << endl;
  cout << endl;

  // Again the Tokenizer, but this time obtaining a
  // vector of strings.
  Tokenizer tok2(linha, ';');
  vector<string> vec = tok2.remaining();
  for (string s : vec) cout << s << endl;
}
```

**http://nemo.inf.ufes.br/**