Linguagem C++

Linguagem C++ para Desenvolvimento em Qt

Prof. Ricardo Reis

Universidade Federal do Ceará
Campus de Quixadá

12 de março de 2013

Tópicos

A Linguagem C++

- **Bjarne Stroustrup** desenvolveu o C++
- Multi-paradigma (Modulação + Orientação a Objetos)
- Compilada
- Mais recente especificação: setembro de 2011 (C++11 ou C++0x)

Framework Qt

- Trolltech ⇒ Nokia ⇒ Digia
- Uso Comercial e Livre (Qt Project)
- Multiplataforma
- Muitos Módulos (QtCore, QtGUI, QtMultimedia, QtNetwork e etc)
- Qt Creator IDE

Alô Mundo, em C++ puro

ALO-MUNDO-PURO.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      cout << "Ola Mundo" << endl;
7      return 0;
8  }
```

Alô Mundo, com UI Qt

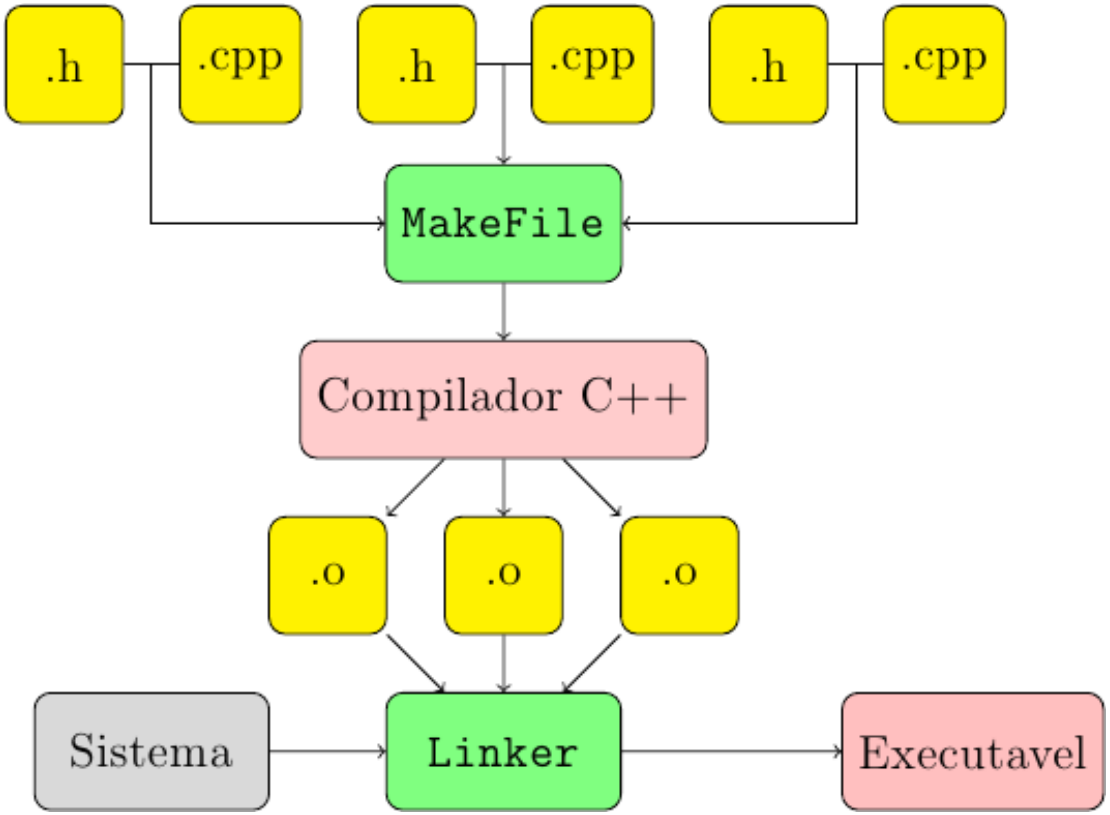
ALO-MUNDO-QT.cpp

```
1  #include <QtGui>
2
3  int main(int argc, char *argv[]) {
4      QApplication app(argc, argv);
5      QLabel label("Alo, Mundo!");
6      label.show();
7      return app.exec();
8  }
```

Instalação

- C++ puro
  - No Linux, g++, via repositório ou instalador
  - No Windows, Projeto `MinGW` + `MSYS`
  - IDEs: eclipse, netbeans, code::blocks, Editores *PlanText* (notepad, gedit e etc).
- Framework Qt: Instalador inclui Biblioteca + Qt Creator IDE (Linux, Windows e MacOS)

Compilação C++ pura



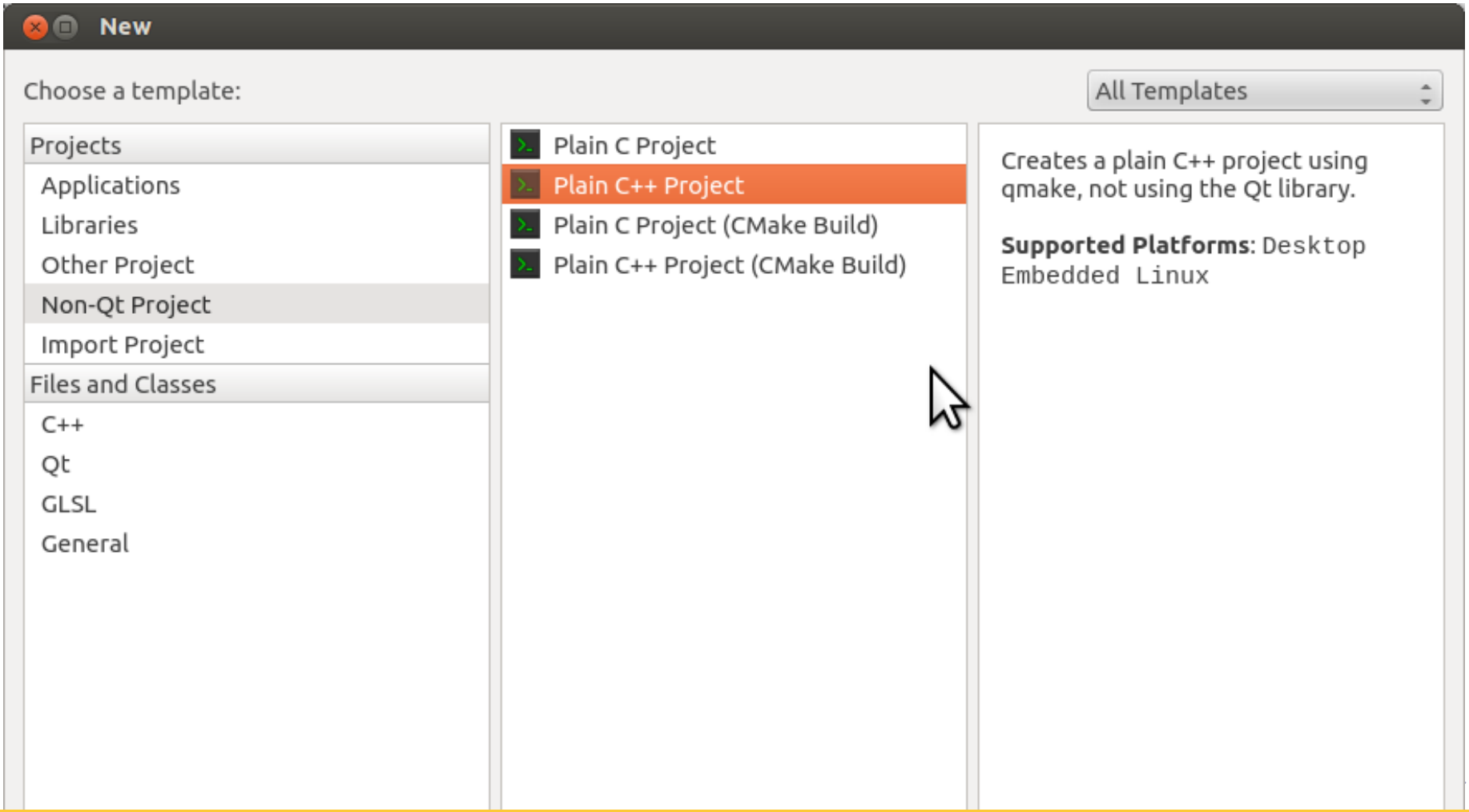

```
g++ arq1.cpp arq2.cpp main.cpp
```

Compilação Qt, linha de comando

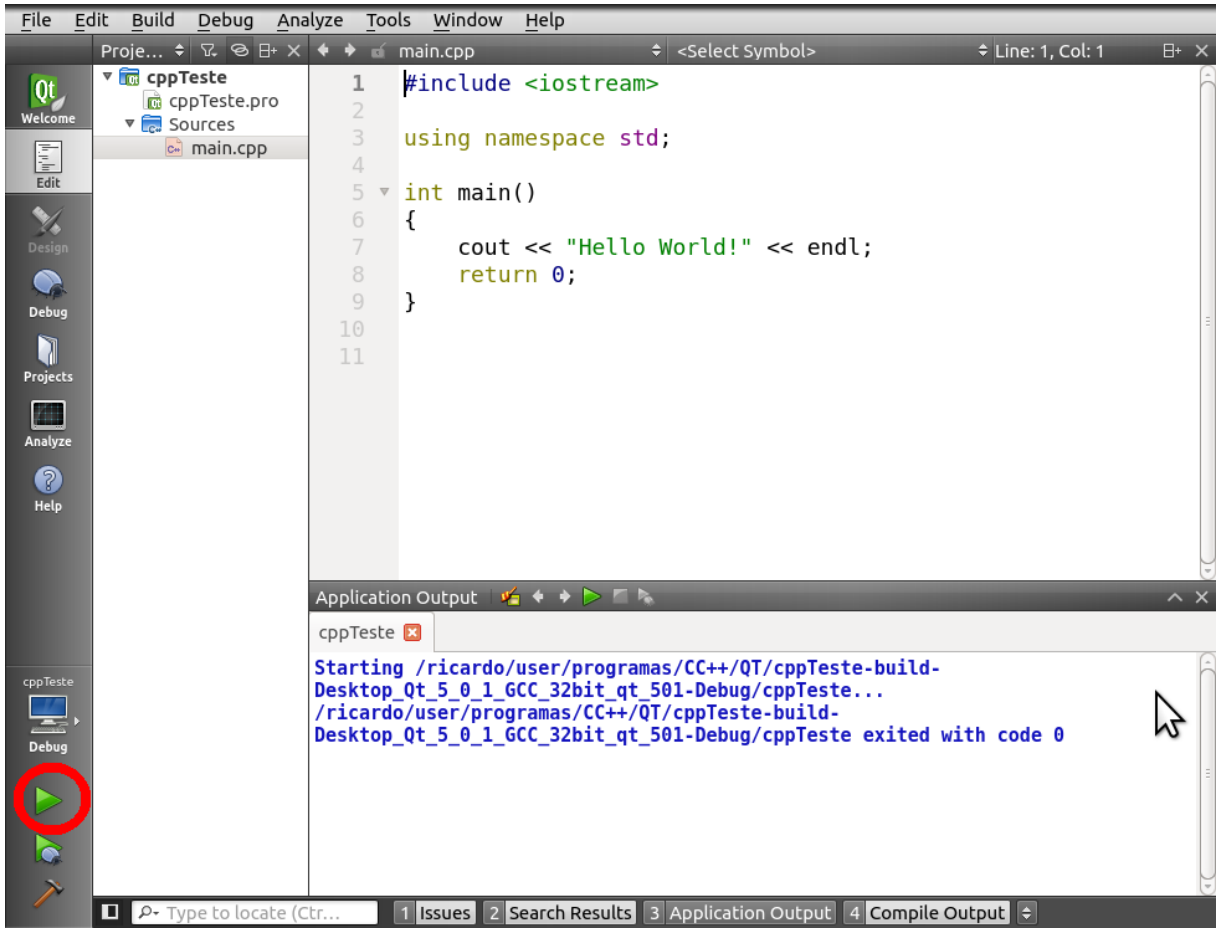
Criar pasta vazia e inserir os arquivos `.cpp` e `.h` do projeto. Na linha de comando, nesta pasta, deve-se,

1. `$ qmake -project` % *cria projeto*
2. `$ qmake` % *cria makefiles*
3. `$ make` % *compila*

Qt Creator, Projeto C++



Qt Creator, Compilação



Tópicos

Como em Linguagem C ...

Todos os elementos da linguagem C são reconhecidos em C++. Algumas famílias de recursos foram extendidas (como operadores) e outras foram criadas (relacionadas à orientação a objetos).

Algumas Novidades do C++

- Possui um tipo lógico `bool` e as palavras-chave *true* e *false* (ainda equivalentes a números).
- Suporta representação de inteiros de 64-bit, o *long long* (novas versões do C já suportam).
- O laço *for* permite definição de contadores. Exemplo,

```
1       for (int k = 0; k<n; k++)
2           \\...
```

- As bibliotecas do C podem ser utilizadas. Usar um `c` prefixo ao antigo nome sem o *.h*,

```
1   #include <cstdio>
2   #include <cstdlib>
3   ...
```

Namespaces

- Um namespace é uma definição de escopo utilizada para resolver problemas de nomes.
- Recursos de mesmo nome podem ser alocados em namespaces de nomes distintos permitindo que sejam utilizados num mesmo contexto.
- Recursos de nomes distintos podem ser espalhados em namespaces de mesmo nome definidos em lugares diferentes.

Exemplo de Namespaces

NAMESPACES.cpp

```
1  namespace abc {
2      void print(int x, int y) { cout << x+ y << endl; }
3  };
4
5  namespace xyz {
6      void print(int x, int y) { cout << x*y << endl; }
7  };
8
9  using xyz::print;
10
11 int main() {
12     abc::print(12, 8);
13     print(12, 8);
14     return 0;
15 }
```

Entrada e Saída

- O que é necessário?

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

- cout

```
1  cout << 23;
2  cout << x + y << endl;
3  cout << "Total a pagar:" << total << '\n';
```

- cin

```
1  cin >> x;
2  cin >> a >> b >> c;
```

Alocação Dinâmica

Além dos tradicionais * e & o C++ possui novos operadores especializados em alocação (`new`) e desalocação (`delete`/`delete []`) de memória.

MEM-DIN.cpp

```
1  int main() {
2      int *k = new int;
3      int *u = new int[20];
4      for (int i = 0; i<20; i++)
5          u[i] = i*i+1;
6      delete k;
7      delete [] u;
8  }
```

Referências

Uma referência é um ponteiro implícito.

REFERENCIAS.cpp

```
1  #include <iostream>
2
3  void swap(int &a, int &b) { int t = a; a = b; b = t; }
4
5  int main() {
6      int i = 25, j = 7;
7      int &r = i;  //Nova referencia
8      r *= 2;
9      swap(i, j);
10     std::cout << i << " " << j << std::endl;
11     return 0;
12 }
```

Retornando Referências

RET-REF.cpp

```
1  #include <iostream>
2
3  int v[5];
4
5  int& get(int k) {
6      static int dummy = -1;
7      return k>=0 && k<5 ? v[k] : dummy;
8  }
9
10 int main() {
11     for (int k=0; k<10; k++) get(k) = 2*k + 1;
12     for (int k=0; k<10; k++) std::cout << get(k) << " ";
13     return 0;
14 }
```

Tópicos

Classes

FNC-MEMBROS.cpp

```
1  class X {
2      // ...
3      int oi() { /* ... */ }; // inline
4      void ola();
5      // ...
6  };
7
8  void X::ola() {
9      // ...
10 }
```

Membros de Classes

MB-DADOS.cpp

```
1  #include <iostream>
2
3  class frac {
4      int num;
5      int den;
6  public:
7      void load(int a, int b) { num = a; den = b; }
8      void print() { std::cout << num << '/' << den << '\n'; }
9  };
10
11 int main() {
12     frac x;
13     x.load(3, 2);
14     x.print();
15     return 0;
16 }
```

Construtores e Destrutores

CONSTRUT-DESTRUT.cpp

```
3  class frac {
4      int num;
5      int den;
6  public:
7      frac(int a, int b) { num = a; den = b; }
8      void print() { std::cout << num << '/' << den << '\n'; }
9      ~frac() { std::cout << "morri!" << num << std::endl; }
10 };
```

```
13     frac x(3, 2); // frac x = frac(3, 2);
14     x.print();
15     frac *pt = new frac(5, 9);
16     pt->print();
17     delete pt;
```

Sobrecarga de Construtores

S-CONSTRUTORES.cpp

```
1  #include <iostream>
2  #include <cmath>
3
4  using namespace std;
5
6  class Point {
7      int xVal, yVal;
8  public:
9      Point (float, float); // coordenadas polares
10     Point (int x, int y) { xVal = x; yVal = y; cout << "1\n"; }
11     Point (void) { xVal = yVal = 0; } // origem
12 };
13
14 Point::Point (float len, float angle) {
15     xVal = (int) (len * cos(angle));
16     yVal = (int) (len * sin(angle));
17     cout << "2\n";
18 }
```

Aplicação de Destrutores

APP-DESTRUT.cpp

```
1  class pilha {
2      int *dat;
3      int m, n;
4  public:
5      pilha(int _m) {
6          m = _m;
7          dat = new int[m];
8      }
9      // ...
10     ~pilha() { delete [] dat; }
11 };
12
13 int main() {
14     pilha p(20);
15     return 0;
16 }
```

Escopo de Classes

- `public`: Membros acessíveis aos objetos.
- `private`: Membros inacessíveis aos objetos.
- `protected`: Membros inacessíveis aos objetos, mas acessíveis a classes herdeiras.

ESCOPO-CLASSES.cpp

```
1 class teste {
2 private:
3   /* ... */
4 public:
5   /* ... */
6 protected:
7   /* ... */
8 };
```

Argumentos Default

ARGS-DEFAULT.cpp

```
1  class Point {
2      int xVal, yVal;
3  public:
4      Point (int x = 0, int y = 0) { xVal=x; yVal=y; };
5      //...
6  };
7
8  int main() {
9      Point p1;          // mesmo que: p1(0, 0)
10     Point p2(10);      // mesmo que: p2(10, 0)
11     Point p3(10, 20);
12     return 0;
13 }
```

Argumento Membro Implícito

Uso de `this`

MB-IMPLICITO.cpp

```
1  class Point {
2      int x, y;
3  public:
4      Point(int x, int y) {
5          this->x = x; this->y = y;
6      }
7  };
```

Lista de Inicialização de Membros

MB-INIT-LIST.cpp

```
1
2  class frac {
3      int num, den;
4  public:
5      frac(int _n, int _d = 1) : num(_n), den(_d) {}
6  };
7
8  class image {
9      int width, height;
10 public:
11     image(int w, int h);
12 };
13
14 image::image(int w, int h) : width(w), height(h) {}
```

Membros Objetos

MB-OBJS.cpp

```
3  class point {
4      int x, y;
5  public:
6      point(int x, int y)
7          { this->x = x; this->y = y; }
8      void print()
9          { std::cout << "(" << x << ", " << y << ")"; }
10 };
11
12 class rect {
13     point u;
14     point v;
15 public:
16     rect (int x, int y, int w, int h): u(x,y), v(x + w, y + h) {}
17     void print()
18         { std::cout << "["; u.print(); v.print();
19           std::cout << "]\n"; }
20 };
```

Vetores de Objetos

- A inicialização implícita de objetos num vetor só ocorre quando a classe correspondente possui uma versão de construtor que não precisa de argumentos.
- Do contrário uma lista explícita é necessária.

VEC-OBJS.cpp

```
24  // point w[3]   =>   erro
25  point w[3] = { point(1,2), point(5,5), point(9,11) };
26  // point *p = new point[6];   =>   erro
27  frac vec[5] = {1, 6, 8, 3, 21};
28  complex teste[3] = {4, -1.9, 0.1};
29  complex d[5];
30  complex *q = new complex[13];
31  delete [] q;
```

Tópicos

Sobrecarga de Funções Membros

- Métodos com mesmo nome, mas com protótipos diferentes.
- Argumentos `Default` podem causar ambiguidades

S-FUNCOES.cpp

```
1 class Time {
2    //...
3 public:
4     long GetTime (void) { /* ... */ }
5     void GetTime (int &hours, int &minutes, int &seconds) { /* ... */
6 };
```

Construtor de Cópia, sobrecarga da Atribuição e sobrecarga do operador []

CONSTRUT-COPIA.cpp

```
4       int *dat, n;
5       void copy_from(const vec& r) {
6           delete [] dat; dat = new int[r.n]; n = r.n;
7           for (int k=0; k<n; k++) dat[k] = r.dat[k];
8       }
9   public:
10      vec(int n) { vec::n = n; dat = new int[n]; }
11      vec(const vec& ref) : dat(0) { copy_from(ref); }
12      vec& operator= (const vec& ref) { copy_from(ref); return *this; }
13      int& operator[] (int k) {
14          static int dummy;
15          return k>=0 && k<n ? dat[k] : dummy;
```

```
23      vec x(5); for (int k=0; k<x.len(); k++) x[k] = 2*k + 1;
24      vec y(x); /* vec y = x */
```

Tópicos

Classes Derivadas

- C++ permite herança, ou seja, novas classes (**derivadas**) podem ser implementadas a partir de outras classes (**primitivas** ou **de base**).
- C++ permite herança simples e múltipla.
- C++ permite **hierarquia de classes** (tanto no âmbito de herança simples como múltipla).

Exemplo de Derivação

EXEMPLO-DERIVACAO.cpp

```
4  class point {
5  protected:
6      float x, y;
7  public:
8      point(float x = 0, float y = 0) { point::x = x; point::y = y; }
9      friend ostream& operator<< (ostream& os, const point& p) {
10         os << "(" << p.x << ", " << p.y << ") \n"; return os;
11     }
12 };
13 class vect : public point {
14 public:
15     vect(float mx, float my) : point(mx, my) {}
16     float len() { return sqrt(x*x + y*y); }
17 };
```

Tipos de Herança

Se *Y* é uma classe derivada de *X* então,

- Na **herança pública** o que é público em *X* se mantém público em *Y*, o que é privado se mantém privado e o que é protegido se mantém protegido.
- Na **herança privada** todos os recursos de *X* são privados em *Y*.
- Na **herança protegida** o que é público e protegido em *X* se tornam protegidos em *Y* e o que é privado em *X* se mantém privado em *Y*.

Escopo Protegido

- Um objeto de uma dada classe só acessa recursos públicos, ou seja, nem o que for privado ou protegido é acessível.
- O que é público ou protegido numa classe base é acessível a classes derivadas, pelo menos em caso de herança pública ou protegida.

Construtores e Destrutores em Classes Derivadas

- A criação de um objeto de uma dada classe pertencente a uma hierarquia de classes provoca a criação de *subobjetos*, um para cada classe primitiva da hierarquia.
- A construção dos objetos na hierarquia mencionada ocorre desde a classe *mais* primitiva para a *menos* primitiva.
- A destrução dos objetos na hierarquia mencionada ocorre desde a classe *menos* primitiva para a *mais* primitiva.

Exemplo de Construtores e Destrutores em Classes Derivadas

CONSTRUT-DESTRUT-DERIV.cpp

```
1  #include <iostream>
2  using namespace std;
3  struct A {
4    A() { cout << "nasceu A\n"; }
5    ~A() { cout << "morreu A\n"; }
6  };
7  struct B: public A {
8    B() { cout << "nasceu B\n"; }
9    ~B() { cout << "morreu B\n"; }
10 };
11 struct C: public B {
12   C() { cout << "nasceu C\n"; }
13   ~C() { cout << "morreu C\n"; }
14 };
15 int main() { C* p = new C(); delete p; }
```

Sobreposição em Classes Derivadas

- Sobreposição (*overwriting*), numa hierarquia de classes, é o mecanismo de redefinição de um método da classe primitiva na classe derivada.
- Métodos sobrepostos não precisam ter mesmo protótipo.
- Versões mais primitivas de métodos sobrepostos ainda podem ser acessados via resolução de escopo.

Exemplo de Sobreposição em Classes Derivadas

EXEMPLO-SOBRECARGA-DERIV.cpp

```
1  #include <iostream>
2  struct X {
3      void A() { std::cout << "A em X\n"; }
4      void B() { std::cout << "B em X\n"; }
5  };
6  struct Y: public X {
7      void A() { std::cout << "A em Y\n"; }
8      //void B() { X::B(); }
9      void B(int) { std::cout << "B em Y\n"; }
10 };
11 int main() {
12     X x; Y y;
13     x.A(); x.B();
14     y.A(); y.B(1); y.X::B();
15 }
```

Ligação Precoce Versus Ligação Tardia

Ligação ou **Binding** refere-se a associação de chamadas de funções ou métodos a seus respectivos endereços de memória. Podem ser,

- **Ligação Precoce** ou **Ligação Estática**: A ligação é definida em tempo de carregamento e não muda em tempo de execução.
- **Ligação Tardia** ou **Ligação Dinâmica**: A ligação é definida em tempo de execução. Uso de ponteiros.

Métodos Virtuais

- Métodos sobrepostos numa hierarquia de classes precisam ser definidos como virtuais para habilitar a ligação tardia.
- Basta inserir `virtual` antes do protótipo da versão mais primitiva do método sobreposto na hierarquia para permitir a ligação tardia em quaisquer objetos da hierarquia (desde que alocados dinamicamente).

Hierarquia de Métodos Virtuais

METODOS-VIRTUAIS.cpp

```
3  class animal {
4  public:
5     virtual void print() { std::cout << "grrrrr\n"; };
6  };
7  class gato: public animal {
8  public:
9     void print() { std::cout << "miauuuu\n"; };
10 };
11 class cachorro: public animal {
12 public:
13    void print() { std::cout << "auauauauau\n"; };
14 };
15 class gato_rajado: public gato {
16 public:
17    void print() { std::cout << "mrrrrrrriiiaauu\n"; }
18 };
```

Efeito da Virtualização

METODOS-VIRTUAIS.cpp

```
21      animal* a[4] = { new animal(),
22                       new gato(),
23                       new cachorro(),
24                       new gato_rajado() };
25      for (int k=0; k<4; k++) {
26           a[k]->print();
27           delete a[k];
28      }
```

Classes Abstratas

- Uma **classe abstrata** é aquela que possui um ou mais métodos *sem* código.
- Mantidas para servirem de base a classes derivadas e para declarar ponteiros que manipulam objetos dessas classes (ligação tardia).
- A inexistência de código num método não impede a instanciação de objetos.

Classes Abstratas Puras

- Um **Membro Abstrato Puro** substitui o corpo da função por =0.
- Uma **Classe Abstrata Pura** possui pelo menos um método abstrato puto.
- Classes abstratas puras não podem ser instanciadas.
- Classes derivadas de classes abstratas puras precisam reimplementar os métodos abstratos puros se não quiserem ser abstratas puras também.

Exemplo de Classe Abstrata Pura

CLASSES-ABSTRATAS.cpp

```
3  struct animal {
4      virtual void som() = 0;
5  };
6
7  struct gato: animal {
8      void som() { std::cout << "miiaauu\n"; }
9  };
10
11 int main() {
12     //animal x; x.som();  => erro!
13     gato y;   animal*p = &y; animal* q = new gato;
14     y.som();
15     p->som();
16     q->som();
17     delete q;
18 }
```

Tópicos

Templates

- **Templates** (gabaritos) são funções ou classes de uso genérico.
- Templates recebem tipos e dados como argumentos (argumentos-template).
- Um template em si não gera código de máquina na compilação.
- O código de máquina de uma função template só será gerado se houver pelo menos uma chamada a ela. chamadas com argumentos-template distintos geram códigos de máquina de funções distintos.
- O código de máquina de uma classe template só será gerado se houver pelo menos uma definição de objeto desta classe. Objetos com argumentos-template distintos geram códigos de máquina de classes distintos.

Classes Template I

CLASSES-TEMPLATES.cpp

```
4  template <typename T>
5  class pilha {
6      T *dat;
7      int capacidade, altura;
8  public:
9      pilha(int c) : capacidade(c), altura(0)
10         { dat = new T[c]; }
11     ~pilha() { delete [] dat; }
12     void push(T x);
13     T pop();
14     T top() { return dat[altura-1]; }
15     bool empty() { return altura==0; }
16     bool full() { return altura==capacidade; }
17 };
```

Classes Template II

CLASSES-TEMPLATES.cpp

```
19 template<typename T>
20 void pilha<T>::push(T x) {
21     if (altura<capacidade)
22         dat[altura++]=x;
23 }
24
25 template<typename T>
26 T pilha<T>::pop() {
27     if (altura>0)
28         return dat[--altura];
29 }
```

Utilizando Classes Template

Ao contrário das funções, definições de objetos de classes templates precisam declarar explicitamente seus argumentos-template,

CLASSES-TEMPLATES.cpp

```
32      pilha<char> P(40);
33      char s[] = "abcdefghijklmn";
34      for (int k=0; s[k]; k++) P.push( s[k] );
35      while ( !P.empty() ) { cout << P.pop(); }
```

CLASSES-TEMPLATES.cpp

```
37      pilha<int> Q(100);
38      for(int x = 123; x>0; x/=2) Q.push(x % 2);
39      while ( !Q.empty() ) { cout << Q.pop(); }
40      cout << endl;
```

Tópicos

A Classe String

CLASSE-STRING.cpp

```
7     string x = "ola mundo!";
8     for (int k = x.length()-1; k>=0; k--)
9         cout << x[k];
10    cout << endl;
11    x[9] = ' ';
12    x += "grande !";
13    x.replace(4, 5, "ceara");
14    cout << x << endl;
15    cout << "\nEscreva seu nome: ";
16    cin >> x;
17    for (string::iterator it = x.begin(); it != x.end(); it++)
18        cout << *it << " ";
```

A Classe List

CLASSE-LIST.cpp

```
7     list<int> a;
8     for (int k = 100; k>0; k/=2) a.push_back(k);  // _front
9     while ( !a.empty() ) {
10        cout << a.back() << " ";
11        a.pop_back(); // _front
12    }
13    list<string> x;
14    string v[] = {"ana", "paulo", "claudio", "ana", "joana", "eva"};
15    for (int k = 0; k<6; k++) x.push_front( v[k] );
16    x.unique();
17    x.sort();
18    for (list<string>::iterator it = x.begin();
19          it != x.end(); it++)
20        cout << *it << endl;
```

A Classe Vector

CLASSE-VECTOR.cpp

```
5  ostream& operator<< (ostream& os, vector<int>& r) {
6      for (vector<int>::iterator it = r.begin();
7           it != r.end(); it++) os << *it << " ";
8      return os;
9  }
10
11 int main() {
12     vector<int> x(5); //{7, 9, 11, 6};
13     x[0] = 12; x[1] = 9; x[2] = -7; x[3] = 2; x[4] = 78;
14     x[5] = 67; // sem efeito
15     for (int k=0; k<x.size(); k++)
16        cout << x[k] << " ";
17     cout << endl << x << endl;
18     while ( !x.empty() )
19        { cout << x.back() << " "; x.pop_back(); }
20 }
```

As Classes ofstream e ifstream

CLASSE-FSTREAM.cpp

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  using namespace std;
5
6  int main() {
7      string nomes[] = {"maria", "joao", "daniel", "amanda",
8                        "pedro", "marta", "carlos", "ana"};
9      ofstream f("nomes");
10     for (int k=0; k<8; k++) f << nomes[k] << " ";
11     f.close();
12     ifstream g("nomes");
13     while ( !g.eof() )
14         { string x; g >> x; cout << x << endl; }
15     g.close();
16 }
```

A Classe Map

CLASSE-MAP.cpp

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  #include <map>
5  using namespace std;
6  int main() {
7      map<string, int> tab;
8      ifstream f("nomes.txt");
9      for (string x; !f.eof(); ) {
10         f >> x;
11         tab[x]++;
12     }
13     f.close();
14     for (map<string, int>::iterator it = tab.begin();
15           it != tab.end(); it++)
16           cout << it->first << '\t'
17                << it->second << endl;
18 }
```