Linguagem C++

Fundamentos de Linguagem de Programação C++

Prof. Ricardo Reis

Universidade Federal do Ceará
Sistemas de Informação

30 de setembro de 2012

Tópicos

Apresentação da Linguagem C++

- Linguagem multi-paradigma, de uso geral e nível médio.
- Desde a década de 1990 possui forte uso comercial e acadêmico. Atualmente 4. posição no índice TIOBE.
- **Bjarne Stroustrup** desenvolveu o C++ (originalmente com nome *C with Classes*) em 1983 no Bell Labs como um adicional à linguagem C.
- Novas características foram adicionadas com o tempo (funções virtuais, sobrecarga de operadores, herança múltipla, templates e tratamento de exceções).
- Padronizações,
  - ISO de 1998
  - revisão em 2003
  - Mais recente especificação ocorreu em setembro de 2011 (C++11 ou C++0x)

Alô Mundo, em C++

alomundo.cpp

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

Instalação do C++

O GCC (GNU Compiler Collection) integra uma versão do C++. Para instalá-lo,

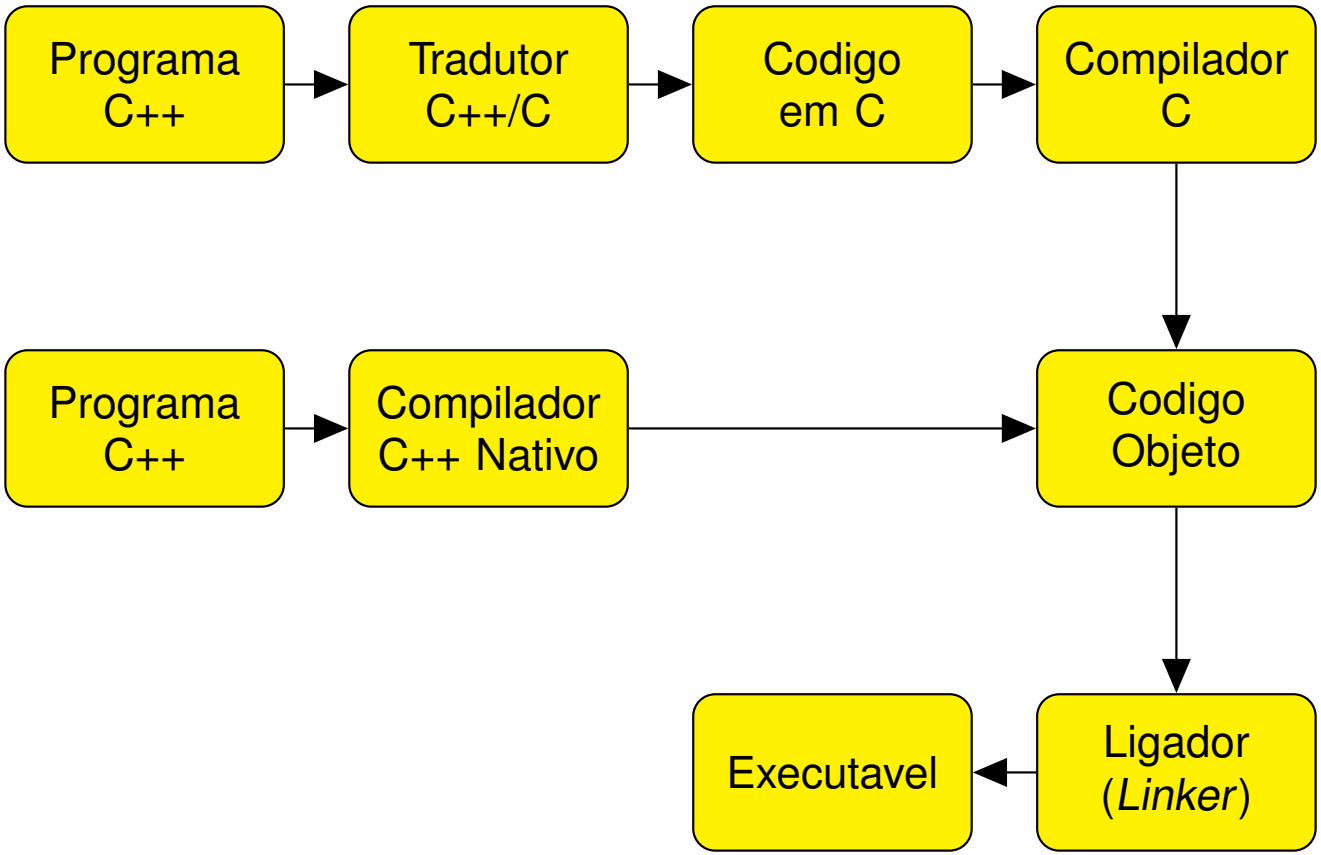
- no Linux,
  - `sudo apt-get install g++`
- no Windows,
  - Cygwin
  - Mingwin (requer configurar PATH) + MSys
  - GCC for Windows

  Podem ser configurados em IDEs como Eclipse e Netbeans.

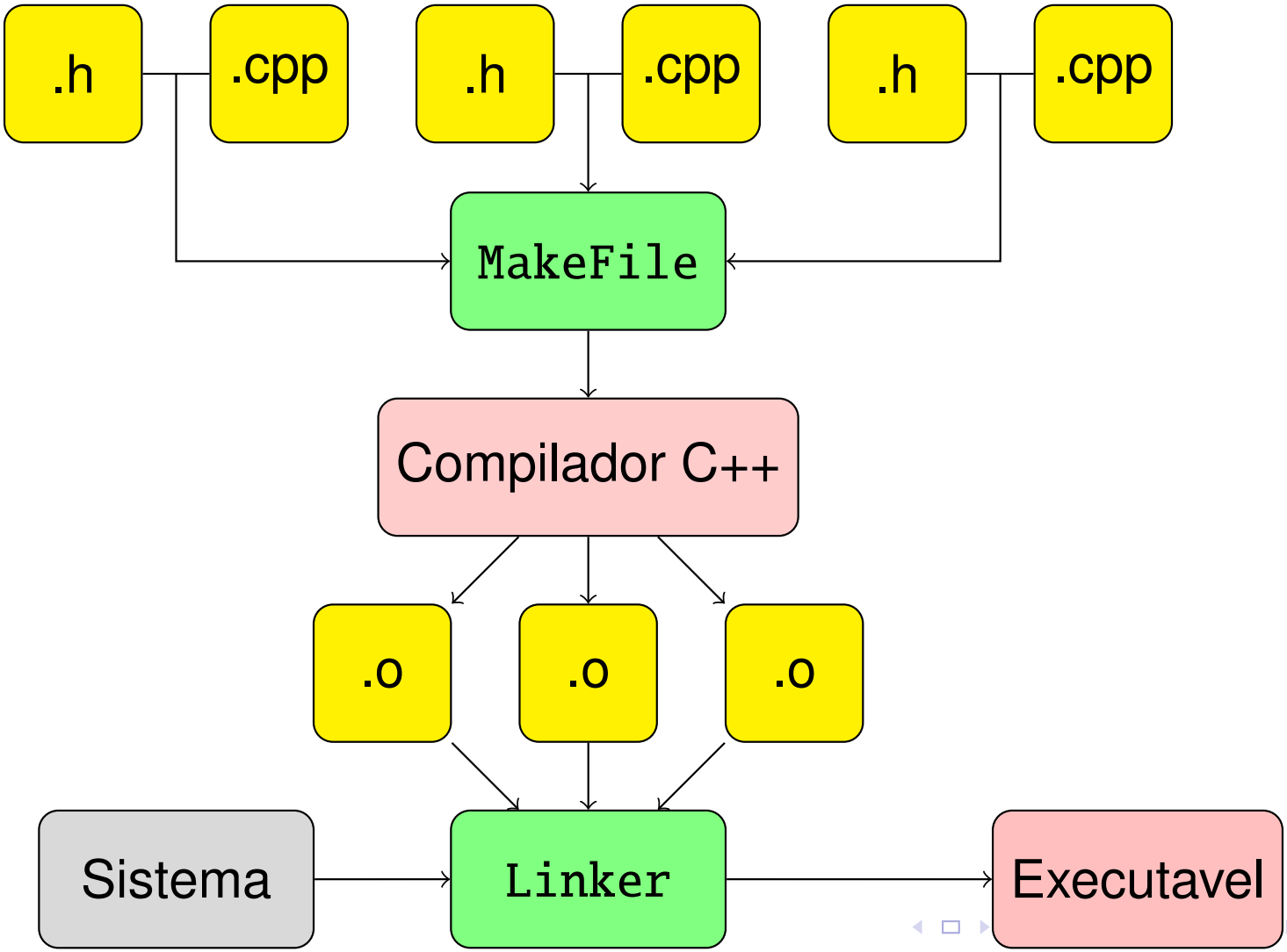
E para testá-lo,

```
$ g++ -v
```

Compilação de Um Programa C++



Compilação Múltipla



Tópicos

Como em Linguagem C ...

- Identificadores
- Representação numérica
- Comentários de código
- Variáveis e Constantes
- Tipos Fundamentais
- Vetores, strings, ponteiros, estruturas e enumerações
- Estruturas de controle de fluxo
- Funções
- entre outras...

Palavras Reservadas do C++

alignas (C++11)	decltype(C++11)	namespace	struct
alignof (C++11)	default(1)	new	switch
and	delete(1)	noexcept(C++11)	template
and_eq	do	not	this
asm	double	not_eq	thread_local(C++11)
auto(1)	dynamic_cast	nullptr (C++11)	throw
bitand	else	operator	true
bitor	enum	or	try
bool	explicit	or_eq	typedef
break	export	private	typeid
case	extern	protected	typename
catch	false	public	union
char	float	register	unsigned
char16_t(C++11)	for	reinterpret_cast	using(1)
char32_t(C++11)	friend	return	virtual
class	goto	short	void
compl	if	signed	volatile
const	inline	sizeof	wchar_t
constexpr(C++11)	int	static	while
const_cast	long	static_assert(C++11)	xor

Tipos Fundamentais e Modificadores

Tipos

- Lógico (`bool`) ⇒ `true, false`
- Caracteres (`char`, `wchar_t`, `char16_t`, `char32_t`)
- Inteiro (`int`)
- Real (`float`, `double`)

Modificadores

- `signed` (`char`, `int`)
- `unsigned` (`char`, `int`)
- `short` (`int`)
- `long` (`int`)
- `long long` (`int`)

Operadores I

Precedência	Operador	Descrição	Asso
1	`::`	Resolução de escopo	Esqu
2	`++ --`	Incremento e decremento pósfixo	
	`()`	Chamada de função	
	`[]`	Subescrito de vetor	
	`.`	Seleção de elemento por referância	
	`->`	Seleção de elemento através de ponteiro	
3	`++ --`	Incremento e decremento pré-fixo	Direi
	`+ −`	Mais e menos unários	
	`! ~`	Negação lógica e bit-a-bit	
	`(type)`	Modelador (Type cast)	
	`*`	Indireção (desreferenciamento)	
	`&`	Endereço de variável	
	`sizeof`	Tamanho em bytes de variável	
	`new, new[]`	Alocador dinâmico de memória	
	`delete, delete[]`	Desalocador dinâmico de memória	
4	`.* ->*`	Ponteiro de membro	Esqu
5	`* / %`	Multiplicação, divisão e resto	
6	`+ −`	Adição e subtração	
7	`<< >>`	Deslocamentos bit-a-bit à esquerda e à direita	
8	`< <=`	Menor e menor-igual	
	`> >=`	Maior e maior-igual	

Operadores II

Precedência	Operador	Descrição
9	`== !=`	Igualdade e desigualdade
10	`&`	Conjunção bit-a-bit
11	`^`	Disjunção exclusiva bit-a-bit
12	`|`	Disjunção bit-a-bit
13	`&&`	Conjunção lógica
14	`||`	Disjunção lógica
15	`?:`	Operador condicional ternário
	`=`	Atribuição
	`+= -=`	Atribuição por soma e com diferença
	`*= /= %=`	Atribuição por produto, divisão e resto
	`<<= >>=`	Atribuição por deslocamento bit-a-bit à esquerda e à direita
	`&= ^= =—`	Atribuição por conjunção, disjunção e disjunção exclusiva
16	`throw`	Disparador de exceções
17	`,`	Vírgula

Controle de Fluxo

- `if` e `switch`
- `while` e `do..while`
- `for`

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

- `break` e `continue`
- `goto`
- `return`

Bibliotecas C++

- Utilitários
- Strings
- Conteiners
- Algoritmos
- Iteradores
- Numérica
- Entrada/Saída
- Localizações
- Expressões Regulares (C++11)
- Operações Atômicas (C++11)
- Suporte a Threads (C++11)

C++ usa Biblioteca C

```
cassert (assert.h)
cctype (ctype.h)
cerrno (errno.h)
cfloat (float.h)
ciso646 (iso646.h)
climits (limits.h)
clocale (locale.h)
cmath (math.h)
csetjmp (setjmp.h)
csignal (signal.h)
cstdarg (stdarg.h)
cstddef (stddef.h)
cstdint (stdint.h)
cstdio (stdio.h)
cstdlib (stdlib.h)
cstring (string.h)
ctime (time.h)
```

Funções

- Protótipo, argumentos, tipo de retorno
- Escopo Global e Local
- Operador de Resolução de Escopo (::)

resolescopo.cpp

```
1 #include <iostream>
2 int x = 11;
3 int main() {
4     int x = 9;
5     std::cout << x << " " << ::x;
6 }
```

Funções e Variáveis Estáticas

- Prefixadas pela palavra chave `static`
- Uma função declarada como estática só é acessível dentro do arquivo onde foi criada.
- Variáveis estáticas são sensíveis ao contexto.

varestat.cpp

```
1 #include <cstdio>
2 void fnc() {
3     static int x = 0;
4     x++;
5     printf("%d ", x);
6 }
7 int main() { fnc(); fnc(); fnc(); }
```

Outros Aspectos em Funções

- Variáveis `Extern` (definição vs. declaração)
- Funções `inline`
- Recursão
- Argumento Default

fncinline.cpp

```
1  inline int soma(int a, int b) {
2      return a+b;
3  }
4
5  int vsoma(int* L = 0, int n = 0) {
6      if (n>0)
7          return n==1 ? L[0] : soma( L[0], vsoma(L+1,n-1) );
8      else return 0;
9  }
```

Argumentos na Linha de Comando

int argc

Comprimento da lista de argumentos

char* argv[]

Lista de argumentos

lncmd.cpp

```
1 #include <iostream>
2 #include <cstdlib>
3 int main(int argc, char *argv[]) {
4     int soma = 0;
5     for (int k = 0; k<argc; k++)
6         soma += std::atoi( argv[k] );
7     std::cout << soma << std::endl;
8 }
```

Vetores e Ponteiros

- Vetores Uni- e Multidimensionais (`[]`)
- Ponteiros (*, &)
- Memória Dinâmica (`new`, `new []`, `delete`, `delete []`)

memdin.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  }
```

- Aritmética de Ponteiros

Funções Passadas Como Argumentos

pntfunc.cpp

```
4  int proc(int L[], int n, int (*fnc)(int,int) ) {
5      if (n < 2) return 0;
6      int res = fnc(L[0], L[1]);
7      for (int k=2; k<n; k++)
8          res = fnc(res, L[k]);
9      return res;
10 }
11
12 int soma(int x, int y) { return x+y; }
13 int mult(int x, int y) { return x*y; }
14
15 int main() {
16     int L[] = {1, 3, 4, 7, -2};
17     cout << "Soma: " << proc(L, 5, soma) << endl;
18     cout << "Produto: " << proc(L, 5, mult) << endl;
19 }
```

Tipo Ponteiro de Função

tipopntfunc.cpp

```
1  #include <iostream>
2  typedef int (*tfnc)(int, int);
3
4  int soma(int a, int b) { return a+b; }
5
6  int main() {
7      tfnc pf = &soma; // ou tfnc pf = soma;
8      std::cout << pf(5, 4) << std::endl;
9      return 0;
10 }
```

Referências

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

retref.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 }
```

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

Entrada e Saída

- 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;
```

- printf()
- scanf()

Tópicos

Classes

fncmembros.cpp

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

Membros de Classes

mbdados.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

construtdestrut.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!\n"; }
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

sconstrutores.cpp

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

Aplicação de Destrutores

appdestrut.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.

escopoclasses.cpp

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

Friends

friends.cpp

```
3  class frac {
4  private:
5      int num, den;
6  public:
7      frac(int a, int b) { num=a; den=b; }
8      void print() { std::cout << num << "/" << den; }
9      friend frac mult(int c, frac x);
10 };
11
12 frac mult(int c, frac x) { return frac(c * x.num, x.den); }
13
14 int main() {
15     frac x(5, 7);
16     mult(3, x).print();
```

Argumentos Default

argsdefault.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`

mbimplicito.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  };
```

Resolução de Escopo em Classes

resolescopoclasse.cpp

```
1  #include <iostream>
2
3  class frac {
4      int num, den;
5  public:
6      frac(int num, int den) { frac::num = num; frac::den = den; }
7      void print() { std::cout << num << "/" << den << std::endl;}
8  };
9
10 int main() {
11     frac x(3, 7);
12     x.frac::print();  // Forma completa
13     x.print();        // Forma abreviada
14     return 0;
15 }
```

Lista de Inicialização de Membros

mbinitlist.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 Constantes

Um membro de classe definido como *const* precisa ser inicializado na lista de inicialização de membros.

mbconst.cpp

```
1 class image {
2     const int width;
3     const int height;
4 public:
5     image(int w, int h): width(w), height(h) {}
6 };
```

Membros Estáticos

mbestat.cpp

```
3  class frac {
4      int num, den;
5      static int cont;
6      static int mdc(int a, int b) {
7          while (b!=0) { int c = a % b;  a = b;  b = c;  }
8          return a;
9      }
10 public:
11     frac(int n, int d) : num(n), den(d) {
12         int m = mdc(n, d); num/=m; den/= m;
13         cont++;
14     }
15     void print() { std::cout << num << "/" << den << std::endl; }
16     static int total() { return cont; }
17     ~frac() { cont--; }
18 };
19
20 int frac::cont = 0;
```

Membros Objetos

mbobjs.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.

vobjs.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;
```

Estruturas

- Estruturas são classes públicas.
- Estruturas foram mantidas para compatibilizar C++ com C

estruturas.cpp

```
4  struct pessoa {
5      char nome[80];
6      int idade;
7      float salario;
8      pessoa(const char* nome, int idade, float salario) {
9          strcpy(pessoa::nome, nome);
10         pessoa::idade = idade;
11         pessoa::salario = salario;
12     }
13     void print() {
14         std::cout << "Nome:\t" << nome <<
15                      "\nIdade:\t" << idade <<
16                      "\nSalario:\t" << salario << std::endl;
17     }
18 };
```

Tópicos

Sobrecarga de Funções Membros

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

sfuncoes.cpp

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

Sobrecarga de Operadores

- Boa parte dos operadores da linguagem podem ser reprogramados por classes de forma a se tornarem especializados em objetos destas classes.
- Podem ser membros da classe ou *friends* globais.
- A palavra chave **`operator`** é prefixada ao operador para formar o nome do membro da classe que o deseja sobrecarregar.

Exemplo de Sobrecarga de Operadores

sopr.cpp

```
13      frac operator+ (const frac& f)
14          { return frac(num*f.den + den*f.num, den*f.den); }
15      frac operator- (const frac& f)
16          { return frac(num*f.den - den*f.num, den*f.den); }
17      frac operator* (const frac& f)
18          { return frac(num*f.num, den*f.den); }
19      frac operator/ (const frac& f)
20          { return frac(num*f.den, den*f.num); }
```

```
25      frac x = frac(1, 3) + frac(5, 4);
26      x.print();
27      (frac(1, 2) + frac(1,5)).print();
28      x = frac(2,5) + 6;
29      x.print();
30      // x = 5 + frac(2,5);     =>     erro!
```

Exemplo de Sobrecarga de Operadores com Friends

soprfriends.cpp

```
1  class frac{
2      /* ... */
3  public:
4      /* ... */
5      friend frac operator+ (const frac& x, const frac& y)
6          { return frac(x.num*y.den + x.den*y.num, x.den*y.den); }
7      friend frac operator- (const frac& x, const frac& y)
8          { return frac(x.num*y.den - x.den*y.num, x.den*y.den); }
9      friend frac operator* (const frac& x, const frac& y)
10         { return frac(x.num*y.num, x.den*y.den); }
11     friend frac operator/ (const frac& x, const frac& y)
12         { return frac(x.num*y.den, x.den*y.num); }
13     /* ... */
14 };
```

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

construtcopia.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 */
```

Sobrecarga do operador ( ) I

soprpar.cpp

```
3  class matrix {
4      float* dat;
5      int ncols, nlins;
6  public:
7      matrix(int nl, int nc) : nlins(nl), ncols(nc){
8          dat = new float[nlins * ncols];
9      }
10     ~matrix() { delete [] dat; }
11     float& operator() (int l, int c) {
12         static float dummy;
13         int k = l*ncols + c;
14         return k>=0 && k<ncols*nlins ? dat[k] : dummy;
15     }
```

Sobrecarga do operador ( ) II

soprpar.cpp

```
21      matrix x(2, 3);
22      x(0,0) = 12;  x(0,1) =  7;  x(0,2) =  9;
23      x(1,0) = -4;  x(1,1) = 14;  x(1,2) = 53;
24      for (int lin=0; lin < x.get_nlins(); lin++) {
25          for (int col=0; col < x.get_ncols(); col++)
26              std::cout << x(lin, col) << "\t";
27          std::cout << std::endl;
28      }
```

Sobrecarga do operador <<

soprmenormenor.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  class frac {
5      int num, den;
6  public:
7      frac(int n, int d) : num(n), den(d) {}
8      friend ostream& operator<< (ostream& os, const frac& r) {
9          os << r.num << "/" << r.den << " ";
10         return os;
11     }
12 };
13
14 int main() {
15     frac x(3, 5);
16     cout << frac(4,3) << frac(7, 9) << x << endl;
17     return 0;
18 }
```

Sobrecarga do operador >>

soprmaiormaior.cpp

```
4  class frac {
5      int num, den;
6  public:
7      frac(int n=0, int d=1) : num(n), den(d) {}
8      friend ostream& operator<< (ostream& os, const frac& r) {
9          os << r.num << "/" << r.den << " "; return os;
10     }
11     friend istream& operator>> (istream& is, frac& r) {
12         static char ch; is >> r.num >> ch >> r.den; return is;
13     }
14 };
15
16 int main() {
17     frac x;    cin >> x;    cout << x << endl;
18     return 0;
19 }
```

Sobrecarga dos operadores ++ e - -

```
4  class frac {
5      int num, den;
6  public:
7      frac(int n, int d) : num(n), den(d) {}
8      friend ostream& operator<< (ostream& os, const frac& r) {
9          os << r.num << "/" << r.den << " "; return os;
10     }
11     frac& operator++ () { num += den; } // e/ou operator--
12     frac& operator++ (int) { num += den; } // e/ou operator--
13 };
14
15 int main() {
16     frac x(3, 4); frac y(1,9);
17     cout << ++x << y++ << endl;
18     return 0;
19 }
```

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

exderiv.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

ctdtderiv.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

exsbderiv.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

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

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

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.

Exemplo de Função Template

fctemplate.cpp

```
1  #include <iostream>
2
3  template <typename T>
4  T max(T vec[], int n) {
5      int imax=0;
6      for (int k=1; k<n; k++)
7          if (vec[k] > vec[imax]) imax = k;
8      return vec[imax];
9  }
10
11 int main() {
12     int L[] = {-4, 6, 11, 9, 1, 7};
13     float M[] = {9.8, 6.7, 7.1, -5.34, -11.05, 5.5};
14     char s[] = "ambocf";
15     std::cout << max(L, 6) << " " << max(M, 6) << " " << max(s, 6) <<
16 }
```

Classes como Argumentos-Template

argstemplate.cpp

```
4  template <class T>
5  T max(T vec[], int n) {
6      int imax=0;
7      for (int k=1; k<n; k++)
8          if (vec[k]>vec[imax]) imax=k; // T suporta ">" ?
9      return vec[imax];
10 }
11
12 template <class T>
13 T soma(T vec[], int n) {
14     frac res = 0;
15     for (int k=0; k<n; k++)
16         res = res + vec[k]; // T suporta "+" e "="?
17     return res;
18 }
```

Utilizando Classes com Argumentos-Template

Operadores utilizados pela implementação precisam estar sobrecarregados.

argstemplate.cpp

```
37 int main() {
38     frac x[] = { frac(7, 3), frac(8, 5), frac(11,25),
39                  frac(1,2),  frac(3,7),  frac(9, 13) };
40     cout << max(x, 6) << soma(x, 2) << endl;
41     return 0;
42 }
```

Classes Template I

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

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,

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(); }
```

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

Manipulação de Exceções

Exceção

É uma reação da aplicação a alguma situação não tratável num contexto pré-definido.

Disparar uma exceção

Significa criar um objeto no escopo onde houve exceção e retorná-lo ao controle do chamador recorrentemente até que seja tratado.

```
throw (/*lista de objetos*/)
```

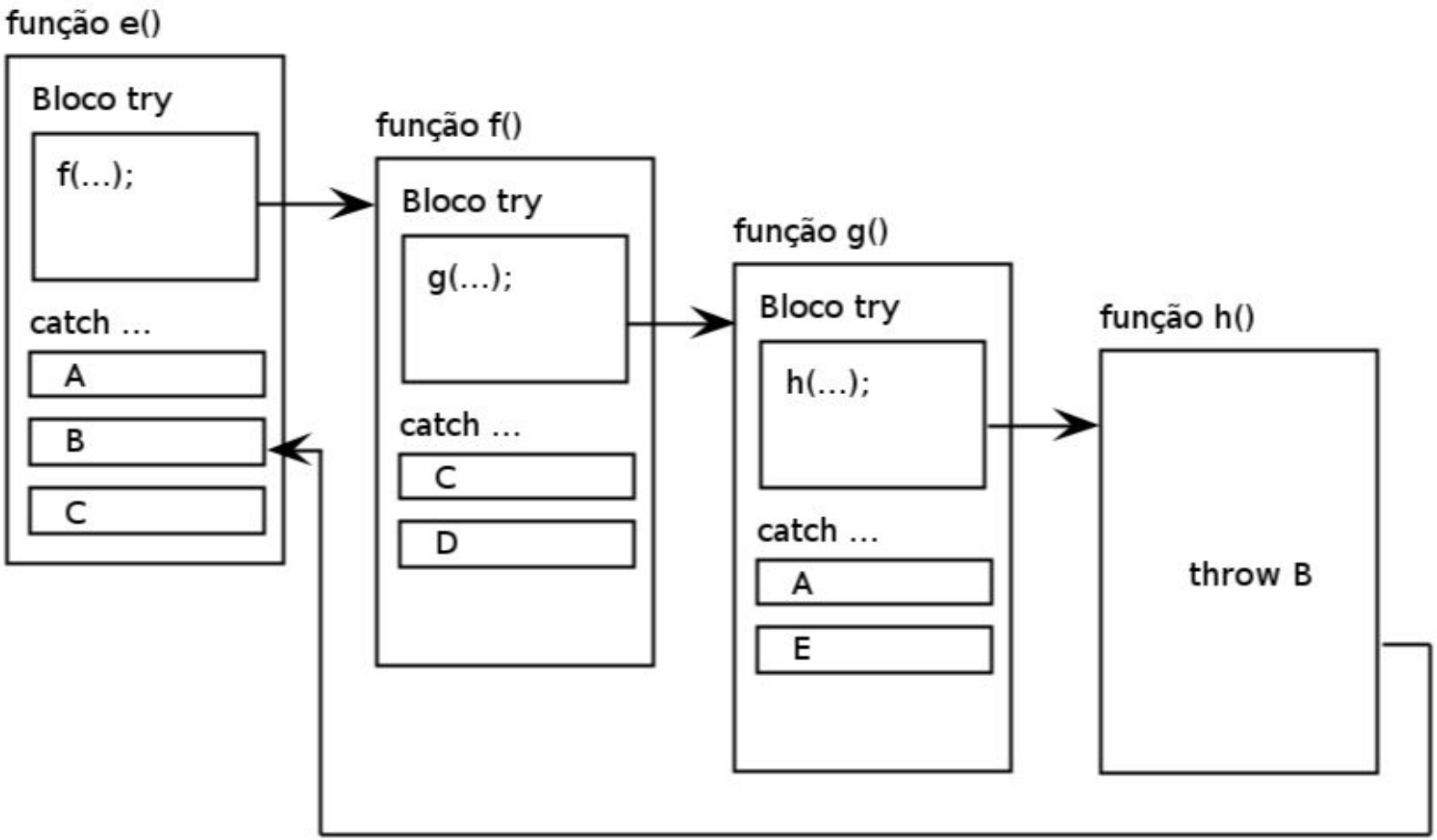
Tratar uma Exceção

Significa manipular objetos-exceção mantendo o fluxo de execução da aplicação.

Sintaxe de Tratamento de Exceções

```
1   try {
2     /* Codigo que pode gerar uma excecao */
3   }
4   catch( /*objeto excecao*/ ) {
5     /* tratamento */
6   }
7   catch( /*objeto excecao*/ ) {
8     /* tratamento */
9   }
10  catch(...) {
11    /* tratamento */
12  }
```

Controle de Fluxo



Exemplo: Uma classe erro

manexcecao.cpp

```
21 class erro {
22     char str[100];
23 public:
24     erro(const char* s) {  strcpy(str, s); };
25     friend ostream& operator<< (ostream& os, const erro& e)
26         { os << e.str << "\n"; }
27 };
```

Exemplo: Disparando Exceções

manexcecao.cpp

```
29 template<typename T>
30 void pilha<T>::push(T x) {
31     if (altura<capacidade)
32         dat[altura++]=x;
33     else throw ( erro("overflow") );
34 }
35
36 template<typename T>
37 T pilha<T>::pop() {
38     if (altura>0)
39         return dat[altura--];
40     else throw ( erro("underflow") );
41 }
```

Exemplo: Tratando Exceções

manexcecao.cpp

```
43 int main() {
44     try {
45         pilha<char> P(5);
46         for (int n = 5000; n>0; n/=2) P.push(n);
47     } catch(erro e) {
48         cout << e;
49     }
50     return 0;
51 }
```

Tópicos

A Classe String

clstring.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

cllist.cpp

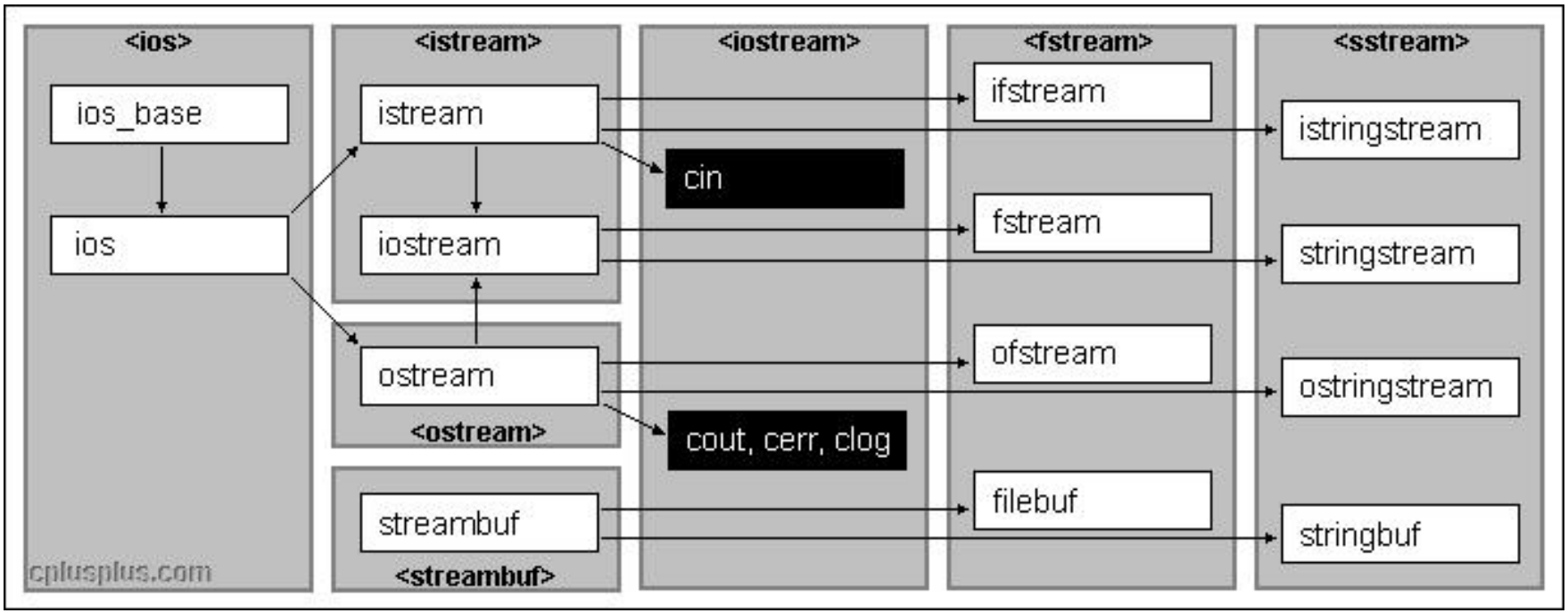
```
7     list<int> a;
8     for (int k = 100; k>0; k/=2) a.push_back(k);  // _front
9     while ( !a.empty() ) {
10        cout << a.back() << " ";
11        a.pop_back(); // _front
12    }
13    list<string> x;
14    string v[] = {"ana", "paulo", "claudio", "ana", "joana", "eva"};
15    for (int k = 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

clvector.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 }
```

Biblioteca de Entrada e Saída



As Classes ofstream e ifstream

cfstream.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

clmap.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 }
```