PUCRS – Escola Politécnica – Bacharelado em Engenharia de Computação
Disciplina: Programação Orientada a Objetos - EC – Prof. Roland Teodorowitsch

GUIA DE REFERÊNCIA – LINGUAGEM C++

Compilação

```
g++ -std=c++11 codigo_fonte.cpp -o codigo_executavel
```

`make` e Makefile

O comando `make` é usado para automatizar a compilação de programas.
O arquivo `Makefile`, que é lido pelo comando `make`, contém uma série de alvos, dependências e comandos que indicam como a construção de uma aplicação deve ser feita. Basicamente o `Makefile` contém entradas no formato:

```
alvo:     dependências
          comando ou @comando
```

Estrutura Básica de um Programa em C++

```
#include <...> // 1) inclusoes
using namespace std; // 2) definicao do espaco de nomes
// 3) macros
// 4) implementacao de classes
public class NomeDaClasse {
  // implementacao da classe
};
// 5) programa principal
int main(int argc, char *argv[]) {
  // implementacao do programa principal
  return 0;
}
```

Comandos e Estruturas da Linguagem

Comando e estruturas	Descrição	Exemplo

Declaração de variáveis	Cria e inicializa novas variáveis	`double polegadas;`
		`int casas = 100;`
		`string nome = "Joao";`

Declaração de constantes	Cria e define novas constantes	`#define MAXIMO 1000` `const int TAMANHO = 100;`

Blocos	Define um conjunto de comandos e declarações	`{`
		`  // Comandos e declaracoes`
		`  // entre chaves`
		`}`

Atribuição	Atribuir o resultado de expressões para variáveis ou elementos de *arrays*	`delta = b*b - 4*a*c;`
		`b = pow(c,5);`
		`strDia = hoje.substr(0,2)`
		`dia = stoi(strDia);`

Atribuição condicional	Atribuir expressões de forma condicional	`pos = a>=0 ? true : false;` `q = (x>0 && y>0)?(x+y):(x-y);`

`if`	Comando condicional simples	`if ( a > 10 ) {`
		`  b = a - 10;`
		`}`

`if/else`	Comando condicional com cláusula para senão	`if ( a > 10 ) {`
		`  b = a - 10;`
		`}`
		`else {`
		`  b = a + 10;`
		`}`

`if/else/if`	Comandos condicionais encadeados	`if ( a < 1 ) {`
		`  b = 0;`
		`}`
		`else if ( a < 10 ) {`
		`  b = a + 10;`
		`}`
		`else if ( a < 20 ) {`
		`  b = a + 20;`
		`}`
		`else {`
		`  b = a;`
		`}`

`switch`	Comando de seleção	`switch (valor) {`
		`  case 0:`
		`  case 1:`
		`  cout << "Zero";`
		`  cout << " ou Um" << endl;`
		`  break;`
		`  case 2:`
		`  cout << "Dois" << endl;`
		`  break;`
		`  case 3:`
		`  cout << "Tres" << endl;`
		`  break;`
		`  default:`
		`  cout << "Outro" << endl;`
		`}`

Comando e estruturas	Descrição	Exemplo

`while`	Laço com pré-teste	`int i = 0;`
		`while (i < 10) {`
		`  cout << i << endl;`
		`  i++;`
		`}`

`for`	Laço com pré-teste	`int v[10];`
		`for (i=0; i<10; i++) {`
		`  v[i] = i+1;`
		`}`

`do/while`	Laço com pós-teste	`i = 0;`
		`do {`
		`  cout << i << endl;`
		`  i++;`
		`} while (i < 10);`

`break`	Saída de um laço ou bloco	`for (i=0; i<10; i++) {`
		`  if (i==5)`
		`  break;`
		`  cout << i << endl;`
		`}`

`continue`	Vai para o próximo passo do laço	`for (i=0; i<10; i++) {`
		`  if (i==5)`
		`  continue;`
		`  cout << i << endl;`
		`}`

`return`	Retorno de método	`return;` `return a+b;`

Entrada e Saída

Inclusões:

```
#include <iostream> // para cout, cin
#include <iomanip>  // para manipuladores de fluxos
```

Entrada:

```
cin >> x;
```

Saída:

```
cout << x << endl;
```

Manipuladores de fluxos:
- Para formatos de valores inteiros: `hex`, `oct`, `setbase`
- Para ponto flutuante: `fixed`, `setprecision(int p)`
- Definição de tamanho: `setw(int w)`
- Preenchimento: `setfill(char c)`
- Alinhamento: `left`, `right`

Ponteiros

Declarações:

```
int variavel;  // armazena um valor/conteúdo
int *ponteiro; // armazena o endereço de um valor/conteúdo
Classe objeto;
Classe *pObjeto; // pode receber o endereço de um objeto
                 //   (um único ou um vetor de objetos)
```

Uso:

```
ponteiro =  &variavel;  // ponteiro aponta para variável ou
                        //  ponteiro recebe endereço da variável
pObjeto =  &objeto;     // ponteiro aponta para  o objeto ou
                        //  ponteiro recebe endereço do objeto
```

Acesso:

```
Classe o1;   // o1 é um objeto
o1.metodo(); // chamada de um método de o1
Classe *o2 = new Classe(); // o2 aponta para novo objeto
o2->metodo();              // chamada de um método de o2
delete o2;                 // destrói o2
Classe *o3 = &o1;          // o3 aponta para o1
o3->metodo();              // chamada de um método de o1
                           //   através de o3
Classe *o4 = nullptr;      // o4  NÃO aponta para nada
int v = 10;         // v tem o valor 10
int *p = &v;        // p aponta para v
cout << p << endl;  // imprime p (ou seja, um endereço)
cout << *p << endl; // imprime o conteúdo de p (ou seja, 10)
int vet[10];          // vetor de 10 inteiros
int *pVet = &vet[0];  // pVet aponta para o v[0]
++pVet;               // pVet aponta agora para v[1]
```

Classes e Objetos

Classe:

```
class NomeDaClasse {
  private:
    // declaracao de variaveis de instancia e metodos
  protected:
    // declaracao de variaveis de instancia e metodos
  public:
    // declaracao ou implementacao de metodos
};
```

Para método implementados fora da classe, usar `NomeDaClasse::` (ou seja, operador de escopo):

```
string NomeDaClasse::obtemNome() { return nome; }
```

Construtores e destrutores são métodos especiais usados, respectivamente, para inicializar variáveis de instância (quando o objeto é criado) e para desalocar estruturas de dados (quando o objeto é destruído). Ambos trazem o nome da classe e não possuem tipo de retorno.

```
NomeDaClasse::NomeDaClasse() { /* ... */ }  // Construtor
NomeDaClasse::~NomeDaClasse() { /* ... */ }  // Destrutor
```

Objetos:

```
// Alocacao no escopo atual
NomeDaClasse nomeObjeto1;
NomeDaClasse nomeObjeto2(a);
// Alocacao dinamica (no HEAP)
NomeDaClasse nomeObjeto3 = new meu();
NomeDaClasse nomeObjeto4 = new meu(a);
// Cuidado, pois eh um prototipo de funcao:
NomeDaClasse nomeFuncao();
```

Arquivos de Texto

Inclusões:

```
#include <iostream>
#include <fstream>
```

Escrita:

```
ofstream arqsaida;
arqsaida.open("teste.txt", ios::out);
if (!arqsaida.is_open()) { cerr << “ERRO” << endl; exit(1); }
arqsaida << “CONTEUDO” << end;
if (arqsaida.fail()) { cerr << “ERRO” << endl; exit(1); }
arqsaida.close();
```

Leitura:

```
ifstream arqsaida;
arqent.open("teste.txt", ios::in);
if (!arqent.is_open()) { cerr << “ERRO” << endl; exit(1); }
string palavra;
arqent >> palavra;
// Leitura de linha
string linha;
getline(arqent, linha);
if (arqent.fail()) { cerr << “ERRO” << endl; exit(1); }
if (arqent.bad()) { cerr << “ERRO” << endl; exit(1); }
if (arqent.eof()) { cout << “FIM DE ARQUIVO” << endl; }
if (arqent.good()) { cout << “TUDO CERTO” << endl; }
arqend.close();
```

Sobrecarga de Operadores

Sobrecarga de operador unário com função (não definida na classe):

```
void operator!(Ponto &p) {
  p.setX(0);
  p.setY(0);
}
```

Sobrecarga de operador unário com método (definido na própria classe):

```
void operator!() {
  this->x = 0;
  this->y = 0;
}
```

Sobrecarga de operador binário com função (não definida na classe):

```
Ponto operator+(Ponto &p1, Ponto &p2) {
  Ponto temp;
  temp.setX(p1.getX()+p2.getX());
  temp.setY(p1.getY()+p2.getY());
  return temp;
}
```

Sobrecarga de operador binário com método (definido na própria classe):

```
Ponto operator+(Ponto &p) {
  Ponto temp;
  temp.setX(p.getX()+this->x);
  temp.setY(p.getY()+this->y);
  return temp;
}
```

Herança

Classe Base:

```
class Base {
private:
        // variáveis de instância e métodos
protected:
        // variáveis de instância e métodos herdados
public:
        // construtor(es)
        // métodos da interface pública
}
```

Classe Derivada:

```
class Derivada : /* private, protected, private */ Base {
        // outros membros...
}
```

Sintaxe para construtor/métod da Derivada chamar o construtor/método da Base:

```
class Base {
public:
        Base(int b) {}
        void funcao() {}
}
class Derivada : public Base {
public:
        Derivada(int d):Base(d) {}
        void funcao() {
                Base::funcao();
        }
}
```