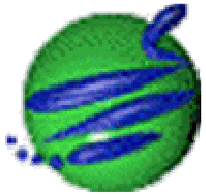


PROGRAMAÇÃO ORIENTADA A OBJETOS EM C++

Adair Santa Catarina
Curso de Ciência da Computação
Unioeste – Campus de Cascavel – PR

Fev/2019

Introdução à linguagem C++

- Criada por Bjarne Stroustrup em 1983:
  - Extensão da linguagem C para suportar classes;
  - Padronizada em 1998 com revisões em:
    - 2003, 2011, 2014, 2017 - ISO/IEC 14882:2017 (C++17).
- O C++ é um superconjunto da linguagem C;
  - Todo programa em C válido também é um programa C++ válido;
- É uma linguagem híbrida pois permite que programas estruturados sejam também desenvolvidos.

Recordando conceitos de POO

- Objetos:
  - Entidades lógicas que contém dados (atributos) e o código (métodos) para manipular estes dados;
  - **Encapsulamento**: Ligação entre atributos e métodos.
- Polimorfismo:
  - Um nome pode ser usado para muitos propósitos relacionados, mas ligeiramente diferentes;
  - Permite que um nome seja usado para uma classe geral de ações.
- Herança:
  - Processo em que um objeto pode adquirir as propriedades de outro objeto → especialização;
  - Definição das característica únicas de um objeto. As outras podem ser herdadas de classes mais gerais.

Entradas e Saídas em C++

- Objetos de E/S:
  - cin: objeto associado ao dispositivo padrão de entrada (teclado);
  - cout: objeto associado ao dispositivo padrão de saída (monitor).
- Operadores de redirecionamento:
  - <<: de saída (da variável para cout);
  - >>: de entrada (de cin para a variável).
- Formatações:
  - Métodos setf() e precision();
  - Nova linha:
    - cout << '\n'; ou
    - cout << endl;

Exemplo 01 – E/S em C++

```
#include <iostream> //biblioteca para operações de E/S em C++
#include <string> //biblioteca para a classe string
using namespace std;

int main ( ){
   int i;
   string str;

   cout << "C++ e facil" << endl;
   cout << "Informe um numero: "; //leitura de um número
   cin >> i;

   cout << "O seu numero e: " << i << endl; //exibição de um número

   cout << "Informe uma string: "; //leitura de uma string
   cin >> str;

   cout << "A sua string e: " << str << endl; //exibição de uma string
   return (0);
}
```

Exemplo 02 – Formatando saídas

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

int main ( ){
  cout.setf(ios_base::hex, ios_base::basefield); //base hexadecimal
  cout.setf(ios_base::showbase); //ativa a exibição da base de saída
  cout << 100 << endl;
  cout.unsetf(ios_base::showbase); //desativa a exibição da base de saída
  cout << 100 << endl;
  return 0;
}

Saídas:
0x64
64
```

Exemplo 03 – Formatando saídas

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

int main () {
  double f = 3.14159;
  cout.setf(ios::left, ios::floatfield); //saída: ponto flutuante não ajustada
  cout.precision(5); //ajusta saída em ponto flutuante com 5 casas
  cout << f <<'\t';
  cout.precision(10); //ajusta saída em ponto flutuante com 10 casas
  cout << f << '\t';
  cout.setf(ios::fixed,ios::floatfield); //saída em ponto flutuante fixa
  cout << f << endl;
  return 0;
}
Saídas:
3.1416   3.14159   3.1415900000
```

Exemplo 04 – Formatando saídas

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

int main () {
  cout << 100 << endl;
  cout.width(10);
  cout << 100 << endl;
  cout.fill('x');
  cout.width(15);
  cout << left << 100 << endl;
  return 0;
}
```

Saídas:

```
100
       100
100xxxxxxxxxxxx
```

Exemplo 05 – Formatando saídas

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

int main () {
   double a, b, c;
   a = 3.1415926534;
   b = 2006.0;
   c = 1.0e-10;
   cout.precision(5);
   cout << a << '\t' << b << '\t' << c << endl;
   cout << fixed << a << '\t' << b << '\t' << c << endl;
   cout << scientific << a << '\t'  << b << '\t'  << c << endl;
   return 0;
}
```

Saídas:

```
3.1416          2006            1e-010
3.14159         2006.00000      0.00000
3.14159e+000    2.00600e+003    1.00000e-010
```

Elementos básicos em C++

- Tipos
  - bool(1), char(1), int(4), float(4), double(8), long(4), short(2), long double(10) → Tamanho e range pode variar conforme a implementação
  - enum
  - void
  - Ponteiros: int*
  - Vetores: char[]
  - Referências: double&
- Declarações e definições
  - char ch;
  - int count = 1;
  - enum hemis {norte, sul};
  - float val = 1.6e-19;

Elementos básicos em C++

- Ponteiros: para um dado tipo `T`, `T*` é do tipo "ponteiro para `T`"
- Indireção: operador `*` retorna "o conteúdo do ponteiro"
- Arrays: para um dado tipo `T`, `T[n]` é um "vetor de *n* elementos do tipo `T`"
- Devem ser inicializados com tamanhos constantes:
  - int v1[10];
  - int v2[ ] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
- Nome de um array pode ser utilizado como ponteiro para o elemento inicial
  - int* p1 = v2;
  - p1++; // p1 aponta para o 2º elemento. (*p1) é igual a 2.

Elementos básicos em C++

- Constantes: `const int tam = 90;`
- Referências: são nomes alternativos para objetos:
  - Uma referência é declarada segundo a estrutura:
    - Tipo& VarRef = VariavelReferenciada;
    - Por exemplo:

    int i = 100;

    int& Ri = i; //ou int &Ri;
  - São diferentes de ponteiros pois uma referência precisa ser inicializada durante sua declaração;

    int &r1, &r2; // → Erro!!!

Exemplo 06 – Entendendo referências

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

int main(){
   char c;
   int i;
   double d;
   char& rc = c;
   int& ri = i;
   double& rd = d;

   cout << "Size of char:    " << sizeof(c) << " bytes" << endl;
   cout << "Size of int:     " << sizeof(i) << " bytes" << endl;
   cout << "Size of double:  " << sizeof(d) << " bytes" <<endl<<endl;
   cout << "Size of char&:   " << sizeof(rc) << " bytes" << endl;
   cout << "Size of int&:    " << sizeof(ri) << " bytes" << endl;
   cout << "Size of double&: " << sizeof(rd) << " bytes" << endl;
   return(0);
}
```

Saídas:

```
Size of char:    1 bytes
Size of int:     4 bytes
Size of double:  8 bytes

Size of char&:   1 bytes
Size of int&:    4 bytes
Size of double&: 8 bytes
```

Exemplo 07 – Entendendo referências

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

int main(){
   int x, y;
   int& rx = x;
   int& ry = y;

   x = 13;
   y = 10;
   rx = 42; //A atribuição é automaticamente aplicada em x.
   ry++; //O incremento é automaticamente aplicado em y.

   cout << "x:  " << x << endl; //Valor de x.
   cout << "y:  " << y << endl; //Valor de y.
   cout << "rx: " << rx << endl; //Valor de rx == x.
   cout << "ry: " << ry << endl; //Valor de ry == y.

   return(0);
}
```

Saídas:

```
x:  42
y:  11
rx: 42
ry: 11
```

Exemplo 08 – Entendendo referências

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

int main(){
   int a = 10, b = 13;
   int &ra = a, &rb = b;
   ra = 42;
   rb = 7;

   cout << "ra  = " << ra << endl; //O valor de ra == 42.
   cout << "rb  = " << rb << endl; //O valor de rb == 7.
   cout << "a   = " << a << endl; //O valor de a == ra == 42.
   cout << "b   = " << b << endl; //O valor de b == rb == 7.

   cout << "&a  = " << &a << endl; //O endereço de a.
   cout << "&b  = " << &b << endl; //O endereço de b.
   cout << "&ra = " << &ra << endl; //O endereço de ra == &a.
   cout << "&rb = " << &rb << endl; //O endereço de rb == &b.
   return(0);
}
```

Saídas:

```
ra  = 42
rb  = 7
a   = 42
b   = 7
&a  = 0x22ff8c
&b  = 0x22ff88
&ra = 0x22ff8c
&rb = 0x22ff88
```

Referências vs Ponteiros

- Referências sempre se referem a um objeto existente, ponteiros podem se referir a um "nulo"

```
void printDouble(const double& rd){
   cout << rd; //não é preciso testar
}
void printDouble(const double *pd){
   if (pd) cout << *pd; //verifica se ponteiro nulo
}
```

- Ponteiros podem mudar o objeto que apontam, referências não.

Alocação dinâmica em C++

- Operadores **new** e **delete**:
  - new é equivalente ao malloc em C;
  - new cria objetos na memória disponível (independe do escopo);
  - um objeto criado por new existe até que seja destruído por delete.

```
//Em C
int main(){
   //aloca ponteiro para double
   double *ptr;
   ptr = (double *)malloc(sizeof(double));
...
}
```

```
//Em C++
int main(){
   //aloca ponteiro para double
   double *ptr;
   ptr = new double;
...
}
```

Alocação dinâmica em C++

- Sintaxes possíveis para o operador new:
  - Como operador:
    - Variável: ptr_tipo = new tipo;

      int *ptr_int = new int;
    - Vetor: ptr_tipo = new tipo[tamanho];

      int *ptr_vet = new int[10];
    - Vetor de ponteiros: ptr_tipo = new tipo*[tamanho];

      int **ptr_vet = new int*[10];
  - Como função:
    - Variável: ptr_tipo = new(tipo);

      int *ptr_int = new(int);

Alocação dinâmica em C++

- Operador delete:
  - Realiza a liberação da memória alocada com o operador new;
- Formas de utilização:
  - Desalocar um único ponteiro:
    - delete ptr;
  - Desalocar um vetor:
    - delete[] ptr;
- Exemplo: programa ExAlocDin.cpp

Elementos básicos: funções

- Passagem de parâmetros:

```
void f(int val, int& ref){
   ++val;
   ++ref;
}
```

```
int main(){
   int i = 1;
   int j = 1;
   f(i, j);
}
```

- Eficiência: passar objetos grandes por referência mas fazê-los constante para evitar modificações

```
void g (const Large& arg){
   //valor de arg não pode ser alterado
}
```

- Argumento default

```
void print(int value, int base = 10);
```

Sobrecarga de funções

- Um tipo de polimorfismo:
- Duas ou mais funções podem compartilhar o mesmo nome, contanto que as suas declarações de parâmetros sejam diferentes.

Por exemplo:

```
//função quadrado é sobrecarregada 3x
int quadrado(const int &i);
double quadrado(const double &d);
long quadrado(const long &l);
```

Exemplo 09 – Sobrecarga de funções

```
#include <iostream>
using namespace std;
int quadrado(const int &i){
   cout << "Dentro da função quadrado() que usa ";
   cout << "um argumento inteiro.\n";
   return (i * i);
}
double quadrado(const double &d){
   cout << "Dentro da função quadrado() que usa ";
   cout << "um argumento double.\n";
   return (d * d);
}
long quadrado(const long &l){
   cout << "Dentro da função quadrado() que usa ";
   cout << "um argumento long.\n";
   return (l * l);
}
int main( ){
   cout << quadrado(10) << endl;
   cout << quadrado(11.0) << endl;
   cout << quadrado(9L) << endl;
   return(0);
}
```

Exemplo 10 – Sobrecarga de funções

```
#include <iostream>
using namespace std;
void solicitacao (const string &str, int &i){
   cout << str;
   cin >> i;
}
void solicitacao (const string &str, double &d){
   cout << str;
   cin >> d;
}
void solicitacao (const string &str, long &l){
   cout << str;
   cin >> l;
}
int main( ){
   int i;
   double d;
   long l;
   solicitacao("Informe um inteiro: ", i);
   solicitacao("Informe um double: ", d);
   solicitacao("Informe um long: ", l);
   return(0);
}
```

Regras para sobrecarga de funções

- int f(int a, int b){...}
- int f(int c, int d){...}
  - Erro!!! → diferenciar os nomes dos parâmetros é insuficiente para diferenciar as funções.

- void f(int x){...}
- int f(int x){...}
  - Erro!!! → as funções não podem apenas diferenciar no tipo de retorno.

Templates

- Forma mais compacta de codificar funções sobrecarregadas que realizam a mesma função sobre conjutos de dados de diferentes tipos;
- A função é declarada com um tipo de dado "em aberto", o qual é substituído em tempo de compilação pelo tipo de dado adequado à cada situação.

```
template <typename X> void swapargs(X &a, X &b){
    X temp;
    ⋮
}
```

Exemplo 11 – Template c/ tipo único

```
#include <iostream>
using namespace std;
template <typename X> void swapargs(X &a, X &b){
      X temp;
      temp = a;
      a = b;
      b = temp;
}
int main( ){
      int i = 10, j = 20;
      double x = 10.1, y = 23.3;
      char a = 'x', b = 'z';
      cout << "Original - i, j: " << i << ' ' << j << endl;
      cout << "Original - x, y: " << x << ' ' << y << endl;
      cout << "Original - a, b: " << a << ' ' << b << endl << endl;
      swapargs(i, j);
      swapargs(x, y);
      swapargs(a, b);
      cout << "Trocados - i, j: " << i << ' ' << j << endl;
      cout << "Trocados - x, y: " << x << ' ' << y << endl;
      cout << "Trocados - a, b: " << a << ' ' << b << endl;
      return(0);
}
```

Exemplo 12 – Template com 2 classes

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

template <class type1, class type2> void myfunc(type1 x, type2 y){
    cout << x << '\t' << y << endl;
}

int main( ){
    myfunc(10, "Hello world!!!");
    myfunc(98.6, 19L);
    myfunc("Hello world!!!", 'A');
    return(0);
}
```

Saídas:

```
10      Hello world!!!
98.6    19
Hello world!!!  A
```

Exemplo 13 – Template especializado

```
#include <iostream>
using namespace std;
template <class X> void swapargs(X &a, X &b){
  X temp;
  temp = a;
  a = b;
  b = temp;
  cout << "Dentro de função - Template" << endl;
}

template<> void swapargs<int>(int &a, int &b){
  int temp = a;
  a = b;
  b = temp;
  cout << "Dentro da função - Template especializada" << endl;
}
```

Exemplo 13 (continuação)

```
int main(){
   int i = 10, j = 20;
   double x = 10.1, y = 23.3;
   char a = 'x', b = 'z';

   cout << "Original - i, j: " << i << ' ' << j << endl;
   cout << "Original - x, y: " << x << ' ' << y << endl;
   cout << "Original - a, b: " << a << ' ' << b << endl;
   cout << endl;

   swapargs(i, j); //ou swapargs<int>(i, j);
   swapargs(x, y);
   swapargs(a, b);

   cout << endl;
   cout << "Trocados - i, j: " << i << ' ' << j << endl;
   cout << "Trocados - x, y: " << x << ' ' << y << endl;
   cout << "Trocados - a, b: " << a << ' ' << b << endl;

   return(0);
}
```

Exemplo 14 – Bubble Sort com Template

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

template <class X> void bubble(X *items, int count){
  register int a, b;
  X t;

  for (a = 1; a < count; a++){
    for (b = count - 1; b >= a; b--){
      if (items[b - 1] > items[b]){
        t = items[b- 1];
        items[b - 1] = items[b];
        items[b] = t;
      }
    }
  }
}
```

Exemplo 14 (continuação)

```
int main(){
   int iarray[7] = {7, 5, 4, 3, 9, 8, 6};
   double darray[5] = {4.3, 2.5, -0.9, 100.2, 3.0};
   int i;

   cout << "Vetor de inteiros não ordenados: ";
   for (i = 0; i < 7; i++) cout << iarray[i] << "  ";
   cout << endl;
   cout << "Vetor de doubles não ordenados : ";
   for (i = 0; i < 5; i++) cout << darray[i] << "  ";
   cout << endl;

   bubble(iarray, 7);
   bubble(darray, 5);

   cout << endl << "Vetor de inteiros ordenados: ";
   for (i = 0; i < 7; i++) cout << iarray[i] << "  ";

   cout << endl << "Vetor de doubles ordenados : ";
   for (i = 0; i < 5; i++) cout << darray[i] << "  ";
   return(0);
}
```

Exemplo 15 – Compactação de Vetores

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

template <class X> void compact (X *items, int count,
                                 int start, int end){
  register int i;

  for (i = end + 1; i < count; i++, start++) items[start] =
    items[i];
  for (; start < count; start++) items[start] = (X)0;
}
```

Exemplo 15 (continuação)

```
int main(){
   int nums[7] = {0, 1, 2, 3, 4, 5, 6};
   char str[18] = "Generic Functions";
   int i;

   cout << "Vetor de inteiros não compactado: ";
   for (i = 0; i < 7; i++) cout << nums[i] << " ";
   cout << endl;
   cout << "Vetor de caracteres não compactado: ";
   for (i = 0; i < 18; i++) cout << str[i] << " ";
   cout << endl;

   compact(nums, 7, 2, 4);
   compact(str, 18, 6, 10);

   cout << endl << "Vetor de inteiros compactado   : ";
   for (i = 0; i < 7; i++) cout << nums[i] << " ";

   cout << endl << "Vetor de caracteres compactado : ";
   for (i = 0; i < 18; i++) cout << str[i] << " ";

   return(0);
}
```

Classes

- Mecanismo de C++ que permite aos usuários a contrução de tipos, que podem ser usados convenientemente como tipos básicos;
- Um tipo é a representação de um conceito. Por exemplo o tipo float, e suas operações +, -, * e /, corresponde à representação do conceito matemático de número real;
- Uma classe é um tipo definido pelo usuário, que não tem similar entre os tipos nativos.

Exemplo de uma classe em C++

```
class Date{
  private:
    int d, m, y; //atributos
    static Date default_date;
  public:
    Date(int dd, int mm, int yy); //métodos
    void addYear(int n);
    void addMonth(int n);
    void addDay(int n);
    static void set_default(int, int, int);
}
```

- Atributos estásticos são parte das classes mas não parte dos objetos

```
void f( ){
   Date::set_default(4, 5,1945);
}
```

Controle de acesso aos membros

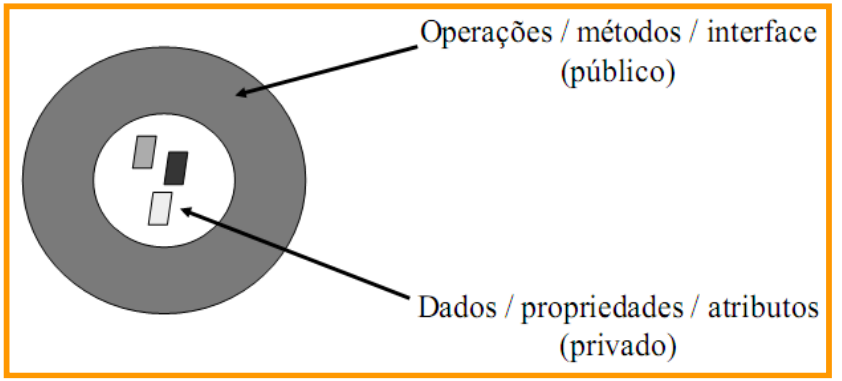
- Atributos e Métodos ***private*** de uma classe são acessíveis somente pelos outros membros da mesma classe ou por classes ***friend***;

- Atributos e Métodos ***protected*** são acessíveis também pelas classes derivadas de uma classe;

- Atributos e Métodos ***public*** são acessíveis a partir de qualquer ponto onde a classe é visível.

Membros de uma classe

- Atributos:
  - Podem ser qualquer tipo de dado, simples ou compostos, ou mesmo outras classes previamente definidas;
- Métodos:
  - São quaisquer funções definidas dentro de uma classe e que podem manipular os atributos de uma classe.



Construtores e Destrutores

- São funções que atuam implicitamente sobre os objetos no momento de sua declaração (construtores) e de sua remoção (destrutores);
- Construtores:
  - Servem para inicializar os atributos de uma classe durante sua declaração;
  - Podem ser sobrecarregados, ou seja, podem ser fornecidas diferentes versões com diferentes tipos de parâmetros.
  - Aceitam valores default:

    Date(int dd, int mm, int yy=2010)

Construtores e Destrutores

- Construtores:
  - Prefira lista de inicialização ao invés de atribuição no corpo do construtor:

    SIM → Date::Date(int dd, int mm, int yy): d(dd), m(mm), y(yy) { }

    NÃO → Date:: Date(int dd, int mm, int yy) {d=dd; m=mm; y=yy;}
    - Defina a lista de inicialização na mesma ordem com que os membros foram declarados;
  - Certifique-se que todos os membros da classe foram inicializados principalmente ponteiros;
  - Sintaxe:
    - Nome_classe([parâmetros]);
    - Não deve apresentar tipo de retorno (nem void);
    - O nome do construtor deve ser o mesmo nome da classe ao qual pertence.

Construtores e Destrutores

- Construtores especiais:
  - Construtor default:
    - É um construtor, sem código e sem parâmetros, criado automaticamente pelo compilador quando não há um codificado pelo programador;
    - Pode também ser escrito pelo programador.
  - Construtor de cópia:
    - Usado quando desejamos declarar um objeto e inicializá-lo com os valores guardados em outro objeto já existente da mesma classe;
    - Sintaxe:

      Nome_classe (Nome_classe &Obj, ...);
      - &Obj: referência ao objeto que deverá ser copiado;
      - Podem existir outros parâmetros, mas devem ter valores default.

Exemplo 16 – Construtor de cópia

```
class Pessoa{
  private:
    int RG, CPF;
    char Nome[31];
    char Endereco[41];
    char Telefone[13];
  public:
    Pessoa( ); // Construtor default
    Pessoa(Pessoa &P); // Construtor de cópia
    int getRG();
    int getCPF();
    char* getNome();
    char* getEndereco();
    char* getTelefone();
    void setRG();
    void setCPF();
    void setNome();
    void setEndereco();
    void setTelefone();
};
```

```
//Construtor de cópia
Pessoa::Pessoa(Pessoa &P){
  RG = P.getRG();
  CPF = P.getCPF();
  strcpy(Nome, P.getNome());
  strcpy(Endereco, P.getEndereco());
  strcpy(Telefone, P.getTelefone());
}
```

Exemplo completo:
Programa ExCls.cpp

Construtores e Destrutores

- Destrutores:
  - São funções complementares aos construtores;
  - São executados automaticamente quando o escopo de duração do objeto se encerra;
  - São responsáveis por "limpar a casa" quando um objeto for removido da memória;
  - Cada classe pode ter um único destrutor.
  - Sintaxe:

  ~Nome_classe();
  - Observações:
    - Não recebe parâmetros e nem possuem valor de retorno.

Arrays de objetos

- Objetos são tipos abstratos de dados;
  - Portanto, como todo tipo de dado, podem também ser usados na construção de arrays.

```
int main( ){
   const int MAX = 4; //define o tamanho da turma
   Pessoa Turma[MAX]; //declaração das variáveis tipo Pessoa

   for (int i = 0; i < MAX; i++) { //Utilizando os objetos
      cout << endl << "Entre com os dados da Pessoa " << i << ": \n ";
      Turma[i].setRG();
      Turma[i].setCPF();
      Turma[i].setNome();
      Turma[i].setEndereco();
      Turma[i].setTelefone();
   }
}
```

Funções Friend

- São funções externas a uma classe que possuem a capacidade de acessar seus membros privados ou protegidos;
- Podem ser de dois tipos:
  - Funções friend não-membros de classe;
  - Funções friend membros de outras classes.
- Funções friend não-membros de classe:
  - Geralmente são funções genéricas de manipulação de dados às quais desejamos enviar atributos de uma classe;
  - Sintaxe:

    Friend Nome_função([parâmetros]);

Exemplo 17 – Função friend

```
class linha;
class box{
   int cor; //cor do box
   int upx, upy; //canto superior esquerdo
   int lowx, lowy; //canto inferior direito
public:
   friend int mesma_cor(linha l, box b);
   void indica_cor(int c);
   void define_box(int x1, int y1, int x2, int y2);
   void exibe_box(void);
};
class linha{
   int cor;
   int comecox, comecoy;
   int tamanho;
public:
   friend int mesma_cor(linha l, box b);
   void indica_cor(int c);
   void define_linha(int x, int y, int l);
   void exibe_linha();
};
```

```
//retorna verdadeiro se linha e box
//têm a mesma cor
int mesma_cor(linha l, box b){
   if (l.cor == b.cor) return (1);
   return (0);
}
```

Funções Friend

- Funções friend membros de outras classes:
  - São utilizadas quando é necessário implementar uma conexão direta entre duas classes;
  - Sintaxe:

```
class Nome_classe{
...
tipo_acesso:
   ...
   friend outra_classe::Nome_funcao_membro;
};
```

  - Programa ExMemFriend.cpp

Exemplo 18 – Função friend

```
class Funcionario; // protótipo da classe
class Caixa{
  private:
    double disponivel;
    double folha;
  public:
    Caixa();
    void fechamento(Funcionario *empregados, unsigned total);
};

class Funcionario{
  private:
    char nome[41];
    double salario;
  public:
    Funcionario();
    friend void Caixa::fechamento(Funcionario *empregados, unsigned total);
};
```

Exemplo 18 – Função friend (cont.)

```
void Caixa::fechamento(Funcionario *empregados, unsigned total){
   for (int i = 0; i < total; i++) folha += empregados[i].salario; //dado privado!!!
   cout << endl << "\nResultados da empresa:\n" << endl
          << "\n\tDisponivel\t: " << disponivel
          << "\n\tFolha\t\t: " << folha
          << "\n\tSaldo\t\t: " << (disponivel - folha) << endl;
}

// Função principal
int main( ){
   Funcionario *empregados = entrada();
   Caixa firma;
   firma.fechamento(empregados, total);
   cout << endl;
   delete[] empregados; //desaloca os funcionarios
   return 0;
}
```

Classes friend

- Classe que pode ter acesso aos atributos internos de outra classe:
- Sintaxe:

```
class A{
   //definição da classe A
};
class B{
   //definição da classe B
tipo_acesso:
   ...
   friend class A;
};
```

- Programa ExClassFriend.cpp

Classes friend

- Comutatividade em classes friend:
  - A relação de amizade não é comutativa;
  - Se a classe A é friend da classe B então A pode acessar os dados privados de B, porém B não pode acessar os dados privados de A.
- "Quem declara a amizade sofre o acesso".
- Transitividade em classes friend:
  - A relação de amizade não é transitiva:
  - Se A é friend de B, e B é friend de C, isto não implica que A é friend de C!

Sobrecarga de operadores

- É o mecanismo da linguagem C++ que permite a redefinição ou sobrecarga dos operadores da linguagem para permitir que certas operações possam ser escritas de forma natural;
- Consiste no acréscimo de significados aos operadores já existentes na linguagem;
- Operadore sobrecarregáveis:
  - Todos os operadores (inclusive new e delete[]), exceto:
    - ".", "::", ".*", "?";

Sobrecarga de operadores

- Restrições na sobrecarga:
  - Não se podem criar novos operadores além dos já existentes na linguagem;
  - O operador sobrecarregado não pode alterar as regras de precedência e associatividade estabelecidas na sua definição original;
  - Ao menos um parâmetro da função operadora deverá ser objeto de classe.
  - Não se pode combinar sobrecargas para gerar uma 3ª função operadora:
    - Sobrecarregar "+" e "=" não sobrecarrega "+=";

Sobrecarga de operadores

- Restrições na sobrecarga (cont.):
  - Os seguintes operadores só podem ser sobrecarregados como método da classe:
    - Atribuição “=“;
    - Apontador-membro “->”;
    - Parênteses “( )”.

- Formas de implementar a sobrecarga:
  - Como método de classe;
  - Como função friend de classe.

Exemplo 19 – Sobrecarga de operadores

```
#include <iostream>
using namespace std;
class tres_d{
   int x, y, z;      //coordenadas 3-D
public:
   tres_d operator+ (tres_d op2);  //op1 está implícito
   tres_d operator= (tres_d op2);  //op1 está implícito
   tres_d operator++ (void);       //op1 também está implícito
   void mostra(void);
   void atribui(int mx, int my, int mz);
};

tres_d tres_d::operator+ (tres_d op2){
   tres_d temp;

   temp.x = x + op2.x;   //estas são adições de inteiros
   temp.y = y + op2.y;   //e o + retém o seu significado
   temp.z = z + op2.z;   //original relativo a eles
   return(temp);
}
```

Exemplo 19 – continuação

```
tres_d tres_d::operator= (tres_d op2){
   x = op2.x;          //estas são atribuições de inteiros
   y = op2.y;          //e o = retém o seu significado
   z = op2.z;          //original relativo a eles
   return(*this);
}

//sobrecarrega um operador unário
tres_d tres_d::operator++ (void){
   x++;
   y++;
   z++;
   return(*this);
}

//mostra as coordenadas x, y, z
void tres_d::mostra(void){
   cout << x << ", ";
   cout << y << ", ";
   cout << z << "\n";
}
```

Exemplo 19 – continuação

```
//atribui coordenadas
void tres_d::atribui(int mx, int my, int mz){
   x = mx;
   y = my;
   z = mz;
}
```

```
Int main( ){
   tres_d a, b, c;
   a.atribui(1, 2, 3);
   b.atribui(10, 10, 10);
   a.mostra();
   b.mostra();
   c = a + b; //agora adiciona a e b
   c.mostra();
   c = a + b + c; //agora adiciona a, b e c
   c.mostra();
   c = b = a; //demonstra atribuição múltipla
   c.mostra();
   b.mostra();
   ++c; //incrementa c
   c.mostra();
   return(0);
}
```

Saídas:

1, 2, 3

10, 10, 10

11, 12, 13

22, 24, 26

1, 2, 3

1, 2, 3

2, 3, 4

Herança

- O C++ implementa:
  - Herança simples: herdar de uma única classe base;
  - Herança múltipla: herdar de várias classes base.
- Herança Simples:
  - Em C++ é implementada a partir de uma declaração de hereditariedade, na qual a classe derivada declara sua classe base, o modificador de acesso da herança e a seguir declara seus membros particulares.
  - Sintaxe:

```
class Nome_classe:[modificador] Nome_classe_base{
  //declaração dos membros da classe derivada
}
```

Herança

- Na sintaxe anterior:
  - O operador ":" especifica a declaração de herança;
  - O modificador especifica a relação de acesso que a classe derivada terá com sua classe base;
  - Os modificadores são dois: public e private.

Tipo de acesso na classe base	Modificador	Tipo de acesso resultante
Public	Public	Public
Private	Public	Inacessível
Protected	Public	Protected
Public	Private	Private
Private	Private	Inacessível
Protected	Private	Private

Exemplo 20 – Herança

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

class veiculos{
   int rodas;
   int passageiros;
public:
   void set_rodas(int num);
   int get_rodas( );
   void set_passageiros(int num);
   int get_passageiros( );
};

class truck:public veiculos{
   int carga;
public:
   void set_carga(int tamanho);
   int get_carga( );
   void show( );
};
```

Exemplo 20 – Herança (continuação)

```
enum tipo {carro, furgao};

class car : public veiculos{
   enum tipo car_type;
public:
   void set_tipo(enum tipo t);
   enum tipo get_tipo( );
   void show( );
};

void veiculos::set_rodas(int num){
   rodas = num;
}

int veiculos::get_rodas( ){
   return rodas;
}

void veiculos::set_passageiros(int num){
   passageiros = num;
}
```

Exemplo 20 – Herança (continuação)

```
int veiculos::get_passageiros( ){
   return passageiros;
}

void truck::set_carga(int num){
   carga = num;
}

int truck::get_carga( ){
   return carga;
}

void truck::show( ){
   cout << "rodas: " << get_rodas() << "\n";
   cout << "passageiros: " << get_passageiros() << "\n";
   cout << "capacidade de carga em metros cubicos: " << get_carga() << "\n";
}

void car::set_tipo(enum tipo t){
   car_type = t;
}
```

Exemplo 20 – Herança (continuação)

```
enum tipo car::get_tipo( ){
   return car_type;
}

void car::show( ){
   cout << "rodas: " << get_rodas() << "\n";
   cout << "passageiros: " << get_passageiros() << "\n";
   cout << "tipo: ";

   switch(get_tipo()){
      case furgao: cout << "furgao\n"; break;
      case carro: cout << "carro\n";
   }
}
```

Exemplo 20 – Herança (continuação)

```
int main(){
   truck t1, t2;
   car c;

   t1.set_rodas(18);
   t1.set_passageiros(2);
   t1.set_carga(3200);

   t2.set_rodas(6);
   t2.set_passageiros(3);
   t2.set_carga(1200);

   t1.show( );
   t2.show( );

   c.set_rodas(4);
   c.set_passageiros(6);
   c.set_tipo(furgao);
   c.show();

   return(0);
}
```

Saídas:

```
rodas: 18
passageiros: 2
capacidade de carga em metros cubicos: 18
rodas: 6
passageiros: 3
capacidade de carga em metros cubicos: 6
rodas: 4
passageiros: 6
tipo: furgao
```

Herança

- Construtores e destrutores:
  - Uma classe derivada pode ter seus próprios construtores e destrutores, independentemente de sua classe base;
  - As classes derivadas não herdam os construtores e destrutores das classes base;
  - Cabe ao programador decidir como os construtores e destrutores das classes derivadas se relacionam com os das classes base;
  - Os objetos são construídos na seguinte ordem: classe base, os membros da classe e a classe derivada.

Herança

- Relacionamentos entre construtores de classes derivadas e classes base. Podem ser implícitos ou explícitos:
  - Relacionamentos implícitos:
    - Caso nada seja especificado pelo programador, o construtor da classe derivada realiza uma chamada implícita ao construtor default da classe base.
  - Relacionamentos explícitos:
    - Deve ser especificada na definição do construtor da classe derivada. Sua sintaxe é:
    - Nome_classe_derivada(parâmetros classe derivada, parâmetros classe base): Nome_classe_base(parâmetros classe base){//código do construtor da classe derivada}

Exemplo 21 – Relacionamentos explícitos

```
#include <iostream>
using namespace std;
class base{
   protected: int i;
   public:
      base(int x) {i = x; cout << "Construindo base\n";}
      ~base() {cout << "Destruindo base\n";}
};
class derivada : public base{
   int j;
   public:
      // derivada usa x; y é passada em lista para a base.
      derivada(int x, int y): base(y) {j = x; cout << "Construindo derivada\n";}
      ~derivada() {cout << "Destruindo derivada\n";}
      void mostrar() {cout << i << " " << j << "\n";}
};
int main( ){
   derivada ob(3, 4);
   ob.mostrar( ); // mostra 4 3
   return 0;
}
```

Destrutores em herança

- Destrutores em herança simples:
  - A ordem de chamada dos destrutores ocorre em ordem inversa à ordem de chamada dos construtores;
  - Os destrutores das classes base são chamados de forma implícita.

Herança – Overriding

- É o mecanismo que permite às classes derivadas modificar métodos herdados das classes base;
- Possibilita introduzir alterações na forma de funcionamento das classes derivadas sem alterar a interface;
- A função sobrescrita ou substituída (Overriding) deve combinar em tipo de retorno, nome e lista de parâmetros com a função antecessora da classe base.

Exemplo 22 – Overriding

```
include <iostream>
using namespace std;
class animal {
  public:
    void comer( );
    void mover( );
    void dormir( ) {cout << "Dormindo..." << endl;}
};
class ave : public animal{
  public:
    void comer( ){cout << "Bicando..." << endl;}
    void mover( ){cout << "Voando..." << endl;}
};
class peixe : public animal {
  public:
    void comer( ){cout << "Mordendo..." << endl;}
    void mover( ){cout << "Nadando..." << endl;}
};
```

```
int main( ){
  ave passarinho;
  peixe sardinha;
  passarinho.mover();
  sardinha.mover();
  return 0;
}
```

Herança múltipla

- Ocorre quando uma classe derivada possui duas ou mais classes base;
- Utilizada quando se precisa combinar características de duas ou mais classes diferentes em uma única classe;
- Sintaxe:
  - class Nome_classe: [modif_1] classe_base1, [modif_2] classe_base2, ..., [modif_n] classe_basen {//Atributos e métodos da classe}

Exemplo 23 – Herança múltipla

```
#include <iostream>
using namespace std;
class base1{
   protected:
      int x;
   public:
      void showx( ) {cout << x << "\n";}
};
class base2{
   protected:
      int y;
   public:
      void showy( ) {cout << y << "\n";}
};
class derived: public base1, public base2{
   public:
      void set(int i, int j) {x = i; y = j;}
};
```

```
int main( ) {
   derived ob;
   ob.set(10, 20); // ← classe derivada
   ob.showx( ); // ← classe base1
   ob.showy( ); // ← classe base2
   return (0);
 }
```

Ambiguidade em herança múltipla

- Quando duas ou mais classes base possuem atributos ou métodos homônimos, e a classe derivada não sobrescreveu estes atributos ou métodos, isto pode causar ambiguidade;
- Para evitá-la pode-se utilizar o operador de resolução de escopo para referir-se ao método, da classe base correta, ao qual se deseja referenciar.

E/S em arquivos em C++

- O C++ permite a extensão do direcionamento de fluxo das classes stream para o armazenamento e recuperação de dados em arquivos;
- Programas que manipulam arquivos em C++ devem incluir o cabeçalho:
  - #include <fstream>
- Esta biblioteca contém as definições das classes que manipulam arquivos.

E/S em arquivos em C++

- Classes da biblioteca fstream para manipulação de arquivos:
  - ifstream: para manipulação de arquivos de entrada;
  - ofstream: para manipulação de arquivos de saída;
  - fstream: para manipulação de arquivos de entrada e saída.

- Exemplos:
  - ExFileMan.cpp;
  - ExFileMan2.cpp;
  - SeqFile.cpp.