Técnicas Avançadas de Programação

- Apontamentos -

Francisco Morgado

Programa

I- Introdução às Técnicas de P. O. O.

- ☆O que é a programação orientada por objectos
- Perspectiva tradicional versus orientada aos objectos
- Conceitos básicos
- Conceito de generecidade e polimorfismo

II- Classes

- ☆Construtor e *destructor*
- Relação entre classes estruturas e uniões
- Funções *in-line*
- Atribuição, passagem e devolução de objectos
- Funções *friend*

III- Vectores, Ponteiros e Referências

- ☆Vectores de objectos
- Ponteiros para objectos e *this*
- Alocação dinâmica - *new* e *delete*
- Atribuição, passagem e devolução de referências para objectos

Programa

IV- *Overloading* de Funções e Operadores

- ☆*Overloading* de construtores
- *Overloading* de operadores
- Utilização de funções *friend*

V- Herança

- ☆Controle do acesso à classe base
- Membros protegidos - *protected*
- Construtores, *destructores* e herança
- Herança múltipla
- Classes bases virtuais

VI- Funções Virtuais

- ☆Ponteiros para classes derivadas
- Utilização de funções virtuais
- Funções virtuais puras - classes abstractas
- Polimorfismo

Programa

VII- Sistema de Entrada/Saída

- Entrada/saída formatada
- Leitura e escrita de objectos
- Entrada/saída binária
- Acesso aleatório

VIII-Generecidade e Excepções

- Funções e classes genéricas - *templates*
- Excepções - *exception handling*
- Métodos estáticos
- Funções de conversão entre tipos

IX- Regras Práticas de P. O. O.

- Criação de uma interface de objectos
- Interface de uma aplicação MDI
- Criação de um modelo de construção e manipulação de objectos

Bibliografia

Teach Yourself - C++ Second Edition

Herbert Schildt

Osborne/Mc Graw-Hill - 1994

C++ *The Complete Reference Second Edition - A Compreensive Desktop Resource*

Herbert Schildt

Osborne/Mc Graw-Hill - 1994

Grafics Programming in Turbo C++

Ben Ezzel

Addison/Wesley - 1990

Programação Orientada para Objectos

Brad J. Cox

Addison/Wesley - 1986

C ++

Bruce Eckel

Osborne/McGraw-Hill

Bibliografia

- *C Avançado*
  Ruben Romano Borges Barbosa
  Graficria - 1994

- *The C Programming Language*
  Kernighan & Ritchie
  Prentice-Hall - 1988 Cota: 03-05-03

- *Top Speed C*
  L. John Ribar
  McGraw-Hill

- *A Book on C*
  Al Kelley & Ira Pohl
  Benjamin-Cummings Cota: 03-05-15

O que é a Programação Orientada por Objectos

Todas as linguagens de programação orientadas por objectos partilham os seguintes três conceitos:

ENCAPSULAMENTO

Privado
Protegido
Público

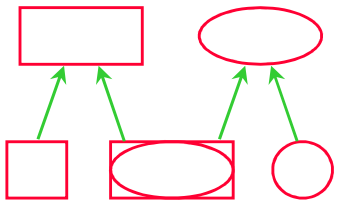
- Embalagem de estado (propriedades) e comportamento (métodos) dos objectos;
- Estado apenas acessível através de operações invocadas por mensagens;
- Interface externa do objecto separada da implementação.

POLIMORFISMO

Uma Interface
Múltiplas Implementações

- Definição de uma interface para actividades relacionadas;
- Criação de uma classe com comportamento genérico;
- Classes derivadas com comportamento específico.

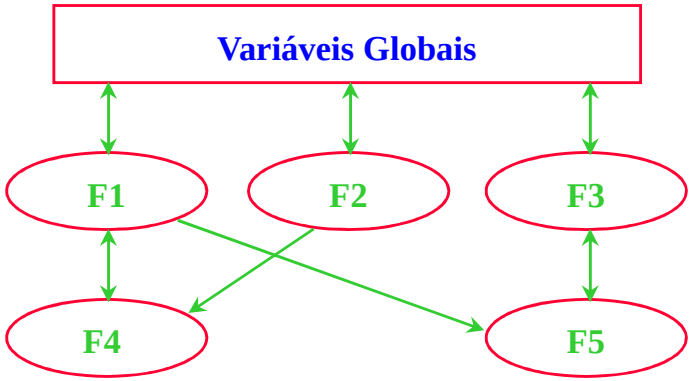
HERANÇA



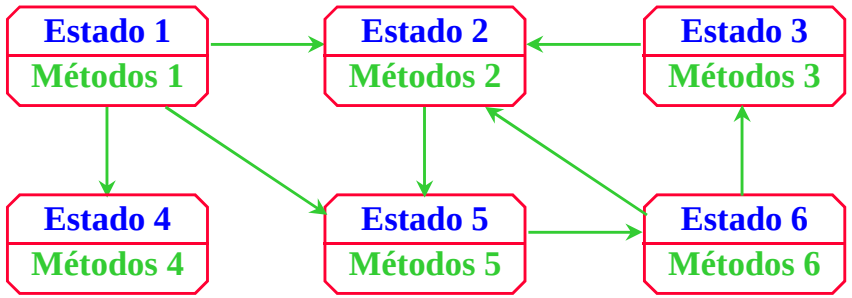
- Herança das propriedades e comportamento protegidos e públicos das classes derivadas;
- Todos herdamos características e comportamentos dos nossos pais.

Perspectiva Tradicional / Orientada aos Objectos

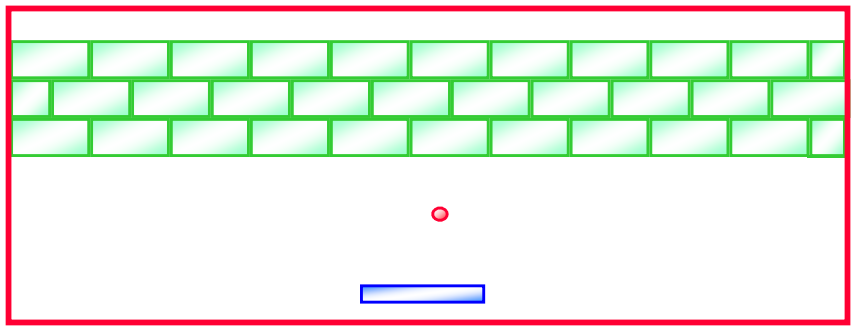
- Perspectiva tradicional : programação orientada por funções

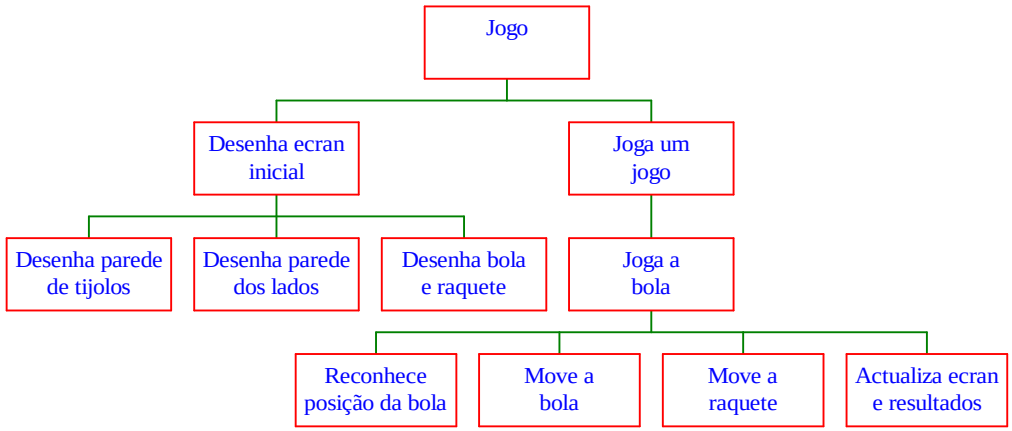


- Perspectiva orientada por objectos



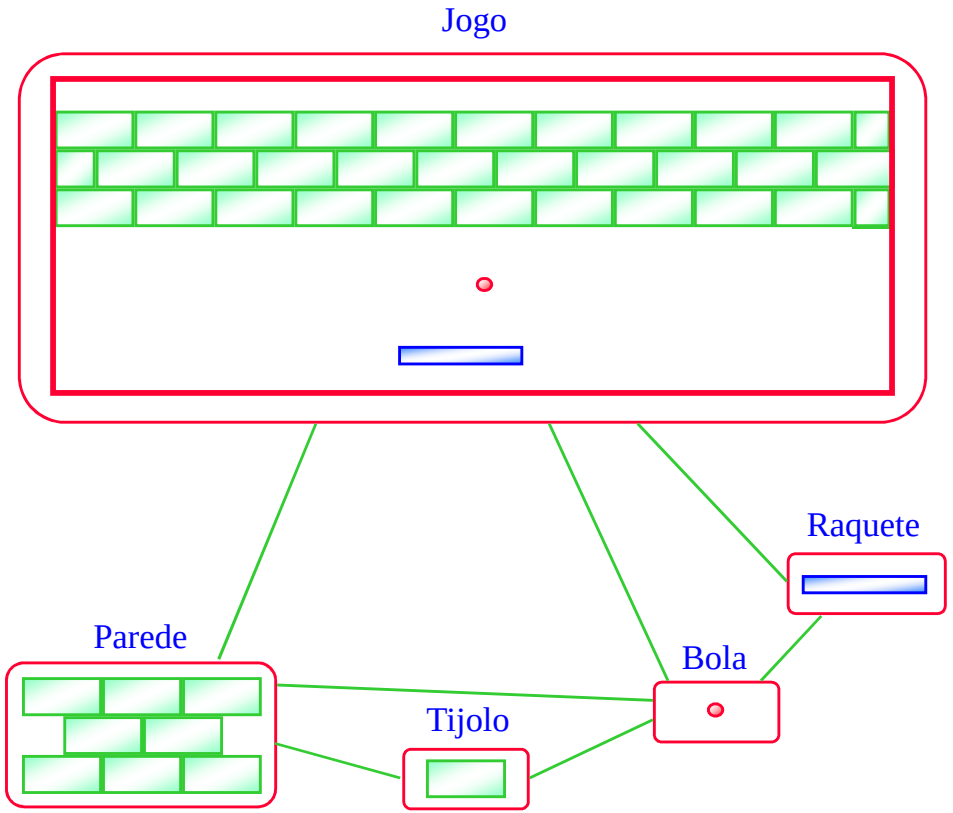
Exemplo: Jogo - Solução Tradicional





- Identificar as funções
- Identificar as relações entre as funções
- Identificar as características das funções

Exemplo: Jogo - Solução Orientada por Objectos



- Identificar os objectos
- Identificar as relações entre os objectos
- Identificar as características dos objectos

Estruturação do Problema

- A prespectiva tradicional decide primeiro quais os métodos (funções) necessários, sendo os dados encaixados depois.
- A prespectiva orientada por objectos começa por decidir quais as entidades (objectos) a manipular e o estado (propriedades) que as caracteriza.

Jogo	Bola	Raquete	Parede	Tijolo
bola raquete lados parede	posição diâmetro direcção velocidade	posição largura altura	tijolos	posição largura altura

- De seguida completam-se os objectos com os métodos necessários. Exemplo: bola.

Estado
- Posição
- Diâmetro
- Direcção
- Velocidade

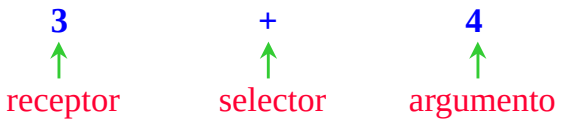
Comportamento
- Posição?
- Mover
- Fazer ricochete
- Bateu num obstáculo?

Conceitos Básicos

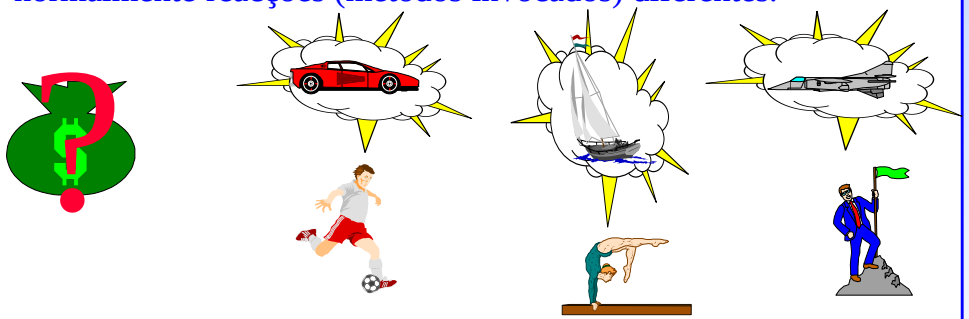
- **Objectos** = dados (atributos) + operações(métodos)
- **Classe** - Descrição de objectos com características comuns
- **Instância** - Objecto descrito por uma classe. As instâncias duma classe têm todas operações e estrutura de dados comuns, mas valores de dados próprios
- **Abstração** - Descrição formal do comportamento de uma dada entidade. Uma classe concretiza uma dada abstração com uma implementação específica das operações
- **Encapsulamento** - Os dados de um objecto só podem ser acedidos pelos seus métodos
- **Mensagem** = selector + argumentos
- **Selector** - Especifica o método a invocar no objecto receptor
- **Receptor** - Objecto que recebe uma mensagem

Conceitos Básicos

- A recepção de uma mensagem invoca um dado método.
- Os objectos interagem exclusivamente através de mensagens.
- Exemplo:



- A diferença relativamente à perspectiva tradicional é que quem controla a resposta à mensagem é o receptor (**3**) e não o "+" (que é selector e não operador).
- A mesma mensagem enviada para objectos diferentes origina normalmente reacções (métodos invocados) diferentes.



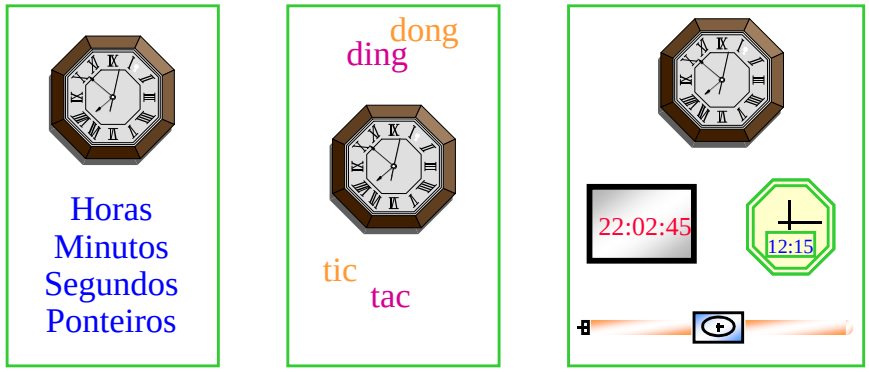
- Exemplo:

Definicão de Objecto (segundo vários autores)

- Um objecto pode ser definido como uma abstracção de software que modela todos os aspectos relevantes de uma única entidade conceptual ou tangível, que pertença ao domínio da solução [Donald Firesmith];
- Um objecto é uma coisa, criada como uma instância de um tipo de objectos. Cada objecto tem uma identidade única distinta e independente de quaisquer das características. Cada objecto tem uma ou mais operações (http://www.omg.org);
- Algo ao qual se pode fazer qualquer coisa; tem estado, comportamento e identidade [Grady Booch];
- Objectos são entidades reais ou conceptuais que podem ser encontradas no mundo que nos rodeia [E. V. Berard];

- Uma abstracção de qualquer coisa no domínio de um problema, reflectindo a capacidade do sistema de manter informação sobre ele e de interagir com ele; é um encapsulamente de valores de atributos e dos seus serviços exclusivos[Peter Coad and Edward Yourdon].
- Um objecto é um conceito, abstracção ou coisa com fronteiras bem definidas e com significado para o problema em questão; promove a reutilização e funciona como uma base concreta para a respectiva implementação em software [J. Rumbaugh].
- Um objecto é uma abstracção de um conjunto de coisas do mundo real de tal forma que todos os elementos do conjunto (instâncias) têm as mesmas características e respeitam as mesmas regras e políticas[S. Shlaer and Neil Lang].

Objectos



- Um **objecto** é composto por:

  - **estado** (variáveis)

  - **comportamento** (métodos)

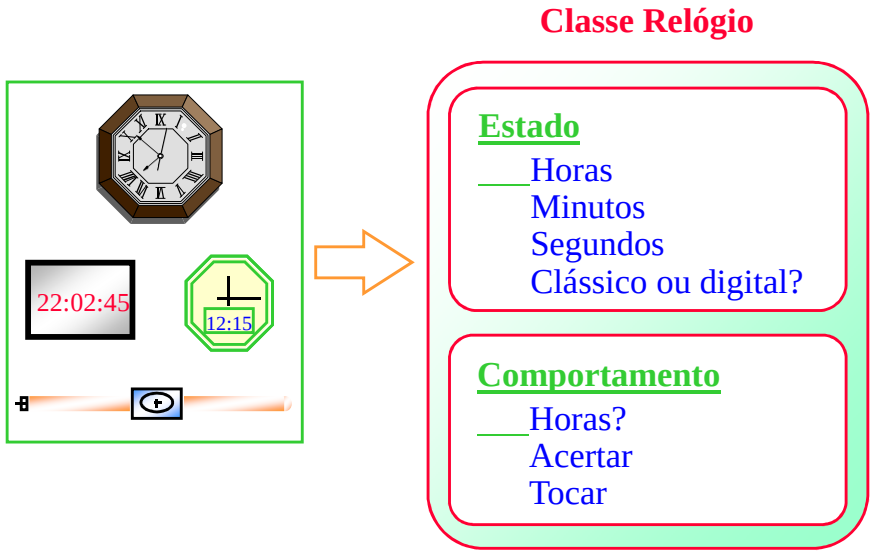
  - **identidade única** (características próprias)

- Os objectos conhecem a identidade de outros e o seu comportamento

- Os objectos interagem por mensagens

- Um objecto corresponde, muitas vezes, a uma entidade do mundo real, podendo, no entanto representar entidades abstractas (exemplos: o número 3, letra A, etc)

Classes



- A classe descreve a estrutura comum a vários objectos, onde cada instância da classe tem o seu próprio estado

- A classe é especificada pelo programador, definindo a interface e a implementação dos objectos

- A classe é a unidade de modularidade na programação orientada por objectos

- Um objecto não é uma classe (mas em algumas linguagens uma classe é um objecto)

Benefícios da Programação Orientada por Objectos

- **Produtividade** - Construir *software* é mais fácil e barato
- **Robustez** - O *software* é mais fiável (por reutilização e melhores regras)
- **Manutenção** - Muito mais fácil melhorias e achar os “bugs”
- **Extensibilidade** - Muito mais fácil acomodar novos requisitos
- **Modularidade** - Abstração e encobrimento da informação
- **Naturalidade** - Programação orientada por abstrações do mundo real

Dificuldades na Utilização da Programação Orientada por Objectos

- Linguagens diferentes usam conceitos e terminologias não totalmente coincidentes
- Não há um modelo semântico de objectos padrão e universal
- Não existem metodologias de análise e projecto universalmente aceites (por exemplo: método ***Fusion***)
  - Técnicas convencionais de análise estruturada não são aplicáveis
  - Ainda é uma àrea de investigação activa
- Formação
  - Linguagens de P. O. O. são conceptualmente diferentes das tradicionais
  - É preciso desaprender “algumas” coisas, pois estamos tão habituados a formas iterativas que esquecemos que o mundo real é constituído por objectos
  - Ler sobre o assunto não é suficiente, é sobretudo essencial programar
  - O tempo de aprendizagem é bastante elevado, mesmo em linguagens de 4ª geração (*Visual Basic*, *Delphi*, etc).

Generecidade

- Consiste em definir classes genéricas (parametrizadas por outras classes)
- Tal como as funções convencionais, há parâmetros formais e actuais (parâmetros genéricos)
- Exemplos:

Classe Associação<C, V>

Estado
- C Chave
- V Valor

Comportamento
- V Valor?
- C Chave?
- AlterarValor(V val)

Classe Fila<T>

Estado
- T Vector[Max]
- int NumElem

Comportamento
- T Frente?
- Colocar(T elem)
- Retirar()

```
Associação<char, int>  A('k', 10);
A.AlterarValor(5);
Fila<int>  FI;
Fila<Associação<char, int>>  FA;
FI.Colocar(3);
FA.Colocar(A);
FI.Colocar(A)   // erro, pois FI apenas pode receber inteiros
```


Polimorfismo

- O polimorfismo permite em parte resolver o problema da limitação da generecidade. Com efeito, supondo que existem a classe Polígono e as classes derivadas Rectângulo e Quadrado, podemos ter uma Fila de Polígonos, com rectângulos e quadrados.
- Polígono **P**; Rectângulo **R**; Quadrado **Q**;
  P = **R**; **P** = **Q**; **R** = **Q**;
  R = **P**; // Erro, pois um polígono não é um rectângulo
- **P** é ponteiro para um polígono e como **R** e **Q** também são, podem ser tratados como tal, o que permite que um ponteiro para um Polígono possa apontar para qualquer objecto das suas subclasses (polimorfismo de inclusão) - relação "is-a"
- No entanto, a partir de um polígono não podemos aceder aos métodos privados de um rectângulo ou quadrado. Resolução:

  Polígono ***P**; Rectângulo **R**, ***PR**;

  P = &**R**; // A partir de **P** apenas se acede à parte do polígonos

  PR = *TYPESAFE_DOWNCAST*(**P**, Rectângulo); // **PR** é um ponteiro para o rectângulo **R** podendo aceder aos seus métodos

Entradas/Saídas em C++

- Em C:

```
#include <stdio.h>
void main()
{
     int   i;     /* Comentário em C */
     float   f;
     char   c,  s[10];
     printf (“\nI = “);
     scanf (“%d”, &i);
     printf (“\n(C, F, S) = “);
     scanf (“%c%f%s”, &c, &f, s);
     printf (“\nI= %d  (C, F, S)= (%c, %f, %s) ”, i, c, f, s);
}
```

- Em C++:

```
#include <iostream.h>
void main()
{                        /* Comentário em C e C++ */
     int   i;            // Comentário em C++ até ao fim da linha
     float   f;                 // Em C++ int, float, char, etc são classes
     char   c,  s[10];          // e não apenas tipos como em C
     cout  <<  "\nI= ";         // cout é o ficheiro de saída padrão
     cin  >>  i;                // cin é o ficheiro de entrada padrão
     cout  <<  "\n(C, F, S) = ";
     cin  >>  c  >>  f  >>  s;
     cout  <<  "\nI= "  <<  i;
     cout  <<  " (C, F, S)= ("  <<  c  <<  ", "  <<  f  <<  ", "
                                <<  s  <<  ")";
}
```

- << e >> são os operadores de escrita e leitura, respectivamente.

Overloading de Funções

- Overloading da função ***divisao***:

```
#include <iostream.h>

int   Divisao (int   i, int   j);
float   Divisao (float   i, int   j);

void main()
{
     int   a = 5, b = 2;
     float   c=5.0;
     cout << "\nDivisãoII(" << a << ", " << b << ")= " <<
          Divisao(a, b);
     cout << "\nDivisãoFI(" << c << ", " << b << ")= " <<
          Divisao(c, b);
}

int Divisao (int   i, int   j)
{
     return (i / j);
}

float Divisao (float   i, int   j)
{
     return (i / j);
}
```

- ***Divisao***(5.2, 2.3) não existe pois não existe Divisao(float, float)
- Não se pode efectuar o overloading de funções com iguais argumentos devolvendo tipos diferentes.

```
float   Divisao (float   i, int   j);     int   Divisao (float   i, int   j);
```

Membros de uma Classe

- Categorias de funções membro:
  - **Funções de gestão** (iniciação, atribuição, gestão de memória, conversão de tipos). Exemplo: construtores.
  - **Funções de implementação** (métodos de instância, utilizadas pelos objectos da classe).
  - **Funções auxiliares** (funções privadas, usadas apenas pelas funções membro da classe).
  - **Funções de acesso** (permitem o acesso às variáveis membro da classe).
- O acesso aos membros pode ser efectuado como nas estruturas em C, com os operadores **.** (*dot*) e **->**

  Rectângulo **R**, ***PR** = **&R**;
  R.PontoInicial; // ou **PR**->PontoInicial;
  R.Diagonal(); // ou **PR**->Diagonal();

- **class** <nome>
  {
  // membros privados
  protected: // membros protegidos
  public: // membros públicos
  }<lista de variáveis>;

Construtores e *Destrutores*

```
#include <iostream.h>

class TPonto
{
    int   X, Y;
public:
    TPonto();   // Construtor
    ~TPonto();   // Destrutor
    void Mostrar();
};

TPonto::TPonto()
{
    cout << "\nConstruir Ponto\n";
    cin >> X  >> Y;
}

TPonto::~TPonto()
{
    cout << "\nDestruir Ponto";
}

void TPonto::Mostrar()
{
    cout << "\nPonto=(" << X << ", " << Y << ")";
}

void main()
{
    TPonto P;   // Utiliza o construtor ao alocar variável P
    P.Mostrar();
}       // Utiliza o destrutor ao desalocar a variável P
```

Construtor com Argumentos

```
#include <iostream.h>

class TPonto
{
      int   X, Y;
public:
      TPonto(int x, int y);   // Construtor com argumentos
      ~TPonto();   // Destrutor
      void Mostrar();
};

TPonto::TPonto(int x, int y)
{
      cout << "\nConstruir Ponto\n";
      X = x;
      Y = y;
}

TPonto::~TPonto()
{
      cout << "\nDestruir Ponto";
}

void TPonto::Mostrar()
{
      cout << "\nPonto=(" << X << ", " << Y << ")";
}

void main()
{
      int K;
      cin >> K;
      TPonto P(K, 3), Q = TPonto(5, -2);   // Dois modos de declaração
      P.Mostrar();
      Q.Mostrar();
}
```


Herança - Introdução

```
#include <iostream.h>

class Base {
      char C;
public:
      char Chave();
      void DefinirChave(char c);
};

class Derivada : public Base {
      int V;
public:
      void DefinirValor(int v);
      void Mostrar();
};

char Base::Chave() { return C; }

void Base::DefinirChave(char c) { C = c; }

void Derivada::Mostrar()
{
      cout << "\nChave= " << Chave() << "  Valor= " << V;
                  // Acesso à classe base
}

void Derivada::DefinirValor(int v) { V = v; }

void main()
{
      Derivada D;
      D.DefinirChave('A');          // Acesso à classe base
      cout << "\nChave= " << D.Chave();
      D.DefinirValor(10);
      D.Mostrar();
}
```

Relação entre Classes, Estruturas e Uniões

- Uma estrutura pode definir uma classe onde por defeito os membros são públicos

```
struct <nome>
{
                // membros públicos
protected:      // membros protegidos
private:        // membros privados
}<lista de variáveis>;
```

- Por seu lado uma união permite definir uma classe (não podendo ser derivada nem derivar de outra) onde os seus atributos sejam partilhados

```
union <nome>
{
                // membros públicos
protected:      // membros protegidos
private:        // membros privados
}<lista de variáveis>;
```

- Deve apenas utilizar estruturas para classes sem membros.
- A utilização de uma união deve ser efectuada apenas para dados partilhados, utilizando uma classe com métodos que use essa união.
- Uma união não pode ter objectos com construtor ou *destrutor*.

POO

Relação entre Classes, Estruturas e Uniões

```
#include <iostream.h>

union Bits
{
    double D;
    unsigned char V[sizeof(double)];
    Bits(double d);
    void Mostrar();
};

Bits::Bits(double d)
{
    D = d;
}

void Bits::Mostrar()
{
    int i, j;
    for (j = sizeof(double)-1; j>=0; j--)
    {
        cout << "Bits do byte " << j << ": ";
        for (i = 128; i; i >>= 1)
            cout << (i & V[j] ? "1" : "0");
        cout << "\n";
    }
}

void main()
{
    Bits BD(12.34);
    BD.Mostrar();
}
```

- Implemente o programa anterior utilizando uma união UB e uma classe CB utilizando a união UB.

Funções *In-Line*

- Uma função *in-line* é semelhante a um macro, pelo que não necessita do mecanismo de chamada e devolução de uma função em *run-time*, o que permite efectuar a sua execução de um modo mais rápido que as funções normais.

```
#include <iostream.h>

inline float Triplo(float x)        // Função in-line explícita
{
    return(x * x * x);
}

class Simples
{
    int I;
public:
    Simples(int i) { I = i; }                       // Função in-line automática
    void Mostrar() { cout << "\nI= " << I; } // Função in-line automática
};

void main()
{
    float F = Triplo(3.0);  // Equivalente a      float F = 3.0 * 3.0 * 3.0;
    cout << "\nF= " << F;
    Simples(3).Mostrar();   // Equivalente a     Simples S(3); S.Mostrar();
}
```

- Uma função *in-line* não pode ter variáveis estáticas, um ciclo, ou um *switch*, nem ser recursiva.

Atribuição de Objectos

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include “ponto.h”

class Ident
{
    char *Nome;
public:
    Ident(char *nome);
    ~Ident() {cout << "\nDestruir Ident"; free (Nome);};
    void Mostrar() { cout << "\nNome: " << Nome; }
    void Alterar(char *nome) { strcpy(Nome, nome); }
};

Ident::Ident(char *nome)
{
    cout << "\nConstruir Ident";
    Nome = (char *) malloc(strlen(nome)+1);
    strcpy(Nome, nome);
}

void main()
{
    TPonto P1(4, -1), P2 = TPonto(2, 3);
    P2 = P1;            // Efectua cópia das coordenadas do ponto P1
    P2.Mostrar();
    Ident I1("abc"), I2("xpto");
    I1.Mostrar();
    I2 = I1;        // Apenas faz o nome de I1 apontar para nome de I2
    I2.Mostrar(); // Deveria efectuar strcpy
                // Produz um erro ao desalocar o Nome de I1 duas vezes
    I1.Alterar("Novo"); // O Nome de I1 e I2 estão na mesma zona de
    I2.Mostrar();           // memória pelo que o nome de I2 é alterado
}
```

Passagem e Devolução de Objectos

```
#include "ponto.h"

class TLinha {
      TPonto P1, P2;
public:
      TLinha(TPonto p1, TPonto p2);
      ~TLinha() { cout << "\nDestruir Linha"; }
      TPonto PontoMedio();
      void Mostrar();
};

TLinha::TLinha(TPonto p1, TPonto p2)  // Necessário definir construtor
{                                     // por defeito do TPonto
      cout << "\nConstruir Linha";
      P1 = p1;
      P2 = p2;
}

TPonto TLinha::PontoMedio()   // Necessário colocar X e Y públicos
{                             // em TPonto
      return TPonto((P2.X - P1.X) / 2, (P2.Y - P1.Y) / 2);
}

void TLinha::Mostrar()
{
      cout << "\nLinha";
      P1.Mostrar();
      P2.Mostrar();
}

void main()
{
      TPonto P(3, 5);
      TLinha L(TPonto(-2, 2), P);
      L.Mostrar();
      L.PontoMedio().Mostrar();
}
```


Funções *friend* - Introdução

- Uma função *friend* acede aos membros não públicos sem no entanto pertencer à classe.

```
#include <iostream.h>

class TLinha;

class TPonto
{
      int   X, Y;
public:
      TPonto() { cout << "\nConstruir Ponto Defeito\n"; X = 0; Y = 0; }
      TPonto(int x, int y) { cout << "\nConstruir Ponto\n"; X = x; Y = y; }
      ~TPonto() { cout << "\nDestruir Ponto"; }
      void Mostrar() { cout << "\nPonto=(" << X << ", " << Y << ")"; }
      TPonto Juntar(TLinha lin);
      friend int Iguais(TPonto p);
      friend TLinha Exemplo(TLinha lin, TPonto p);
      friend TLinha;    // Necessário para TLinha::PontoMedio()
};

class TLinha
{
      TPonto P1, P2;
public:
      TLinha() { cout << "\nConstruir Linha Defeito\n";
                 P1 = TPonto(); P2 = TPonto(); }
      TLinha(TPonto p1, TPonto p2);
      ~TLinha() { cout << "\nDestruir Linha"; }
      void Mostrar() { cout << "\nLinha"; P1.Mostrar(); P2.Mostrar(); }
      TPonto PontoMedio() { return TPonto((P2.X - P1.X) / 2,
                                          (P2.Y - P1.Y) / 2); }
      friend TPonto TPonto::Juntar(TLinha lin);
      friend TLinha Exemplo(TLinha lin, TPonto p);
};
```

Funções *friend* - Introdução

```
TPonto TPonto::Juntar(TLinha lin)
{
    return TPonto (lin.P1.X, Y);  // Acede a P1 pois é friend de TLinha
}

TLinha::TLinha(TPonto p1, TPonto p2)
{
    cout << "\nConstruir Linha";
    P1 = p1;
    P2 = p2;
}

int Iguais(TPonto p)
{
    return (p.X == p.Y);  // Acede a  X e Y pois é friend de TPonto
}

TLinha Exemplo(TLinha lin, TPonto p)
{       // Acede a P1 pois é friend de TLinha
        // Acede a X e Y pois é friend de TPonto
    return TLinha(lin.P1, TPonto(p.Y, p.X));
}

void main()
{
    int X, Y;
    cin >> X >> Y;
    TPonto P(X, Y);
    TLinha L(TPonto(2, 6), P);
    L.PontoMedio().Mostrar();
    Exemplo(L, P).Mostrar();
    cout << (Iguais(P) ? "\nDiferentes" : "\nIguais");
}
```


Funções *friend* - Introdução

- A função **Iguais** é *friend* da classe TPonto para poder aceder às coordenadas (X e Y) não públicas de p.
- A função **TPonto::Juntar** é *friend* da classe TLinha para poder aceder às coordenadas não públicas de lin.
- A função **Exemplo** é *friend* das classes TPonto e TLinha para poder aceder às coordenadas não públicas de lin e p.
- Podemos igualmente ter uma classe *friend* de outra (todos os membros não públicos da classe podem ser acedidos pela outra classe), por exemplo: a classe TLinha *friend* da classe TPonto.
- Neste momento, já deve ter verificado o número exagerado de vezes que são construídos e destruídos os objectos. Tal, deve-se à passagem por valor dos objectos. Este problema será resolvido através da utilização da passagem por referência, a introduzir no capítulo seguinte.

Vectores e Ponteiros de Objectos

```
#include "ponto.h"

class Simples {
      int I;
public:
      Simples() { I = 0; cout << "\nConstruir Defeito Simples"; I = 0; }
      Simples(int i) { cout << "\nConstruir Simples"; I = i; }
      void Mostrar() { cout << "\nI= " << I; }
};

void main() {
      // Iniciação de vectores de classes com construtor com um argumento
      Simples VS1[2], VS2[2]={1,-2}, VS3[2] = {Simples(1), Simples(2)},
                  VS4[2][3] = { {1, 0, -3},
                                {4, 8, -7}  };
      VS1[0] = 1; VS1[1] = -2;
      for (int i = 0; i < 2; i ++) {
            VS1[i].Mostrar();
            VS2[i].Mostrar();
            VS3[i].Mostrar();
            for (int j = 0; j < 3; j ++)
                  VS4[i][j].Mostrar();
      }
      // Iniciação de vectores de classes com construtor com +1 argumento
      TPonto VP1[3], VP3[3] = {TPonto(1,2), TPonto(-5,3), TPonto(4,-2)},
                  VP4[3][2] = { { TPonto(-1, 2), TPonto(-5, 1) },
                                { TPonto(4, -2), TPonto(1, -3) },
                                { TPonto(-2, 6), TPonto(3, -3) } }, *PP3=VP3;
      VP1[0]= TPonto(1,2); VP1[1]= TPonto(-5,3); VP1[2]= TPonto(4,-2);
      for (i = 0; i < 3; i ++) {
            VP1[i].Mostrar();
            PP3->Mostrar();   PP3 ++;
            for (int j = 0; j < 2; j ++)
                  VP4[i][j].Mostrar();
      }
}
```


Ponteiro *this*

- ***this*** é um ponteiro que é automaticamente passado para qualquer função membro quando é chamada, onde ***this*** é o ponteiro para o objecto que gera a chamada.

```
#include <iostream.h>

class TPonto
{
public:
      int   X, Y;
      TPonto() { cout << "\nConstruir Ponto Defeito\n"; X = 0; Y = 0; }
      TPonto(int x, int y) { cout << "\nConstruir Ponto\n"; X = x; Y = y; }
      ~TPonto() { cout << "\nDestruir Ponto"; }
      void Mostrar() { cout << "\nPonto=(" << X << ", " << Y << ")"; }
      void MoverEMostrar(int dx, int dy);
      TPonto *Ponteiro() { return this; }
};

void TPonto::MoverEMostrar(int dx, int dy)
{                       // Equivalente a
      X += dx;          // this->X += dx;
      Y += dy;          // this->Y += dy;
      Mostrar();        // this->Mostrar();
}

void main()
{
      TPonto P(3, 2);
      P.MoverEMostrar(2, 4);
      P.Ponteiro()->Mostrar();      // Note-se que não podemos aceder a
}                       // P->this, pois this é um membro privado
```

- Em C++ a chamada *P.MoverEMostrar(2, 4)* é transformado internamente em *MoverMostrar(&P, 2, 4)*.

Alocação Dinâmica - *new* e *delete*

- Em C:

```
pont = (tipo *) malloc (sizeof (tipo));
free (pont);
```

- Em C++:

```
pont = new tipo;
delete pont;
```

- *new* e *delete* são operadores que efectuam a alocação e desalocação dinâmica de memória contígua.

```
#include "ponto.h"

void main()
{
    int *PI1 = new int, *PI2 = new int(-6);  // Alocação e iniciação de PI2
    if (!PI1)
    {
        cout << "\nErro de Alocacao";
        return;
    }
    *PI1 = 15;
    cout << "\nValor1: " << *PI1 << " no Endereço: " << PI1;
    cout << "\nValor2: " << *PI2 << " no Endereço: " << PI2;
    delete PI1;
    delete PI2;
    TPonto *PP1 = new TPonto,  *PP2 = new TPonto (3, 2);
    cout << "\nIntroduza X e Y: ";          // Alocação e iniciaçãode PP2
    cin >> PP1->X >> PP1->Y;
    PP1->Mostrar();
    PP2->Mostrar();
    delete PP1;
    delete PP2;
}
```


Alocação Dinâmica de Vectores - *new[ ]* e *delete[ ]*

- A alocação e desalocação dinâmica de vectores é efectuada do seguinte modo:

```
pont = new tipo [num_elementos];
delete[ ] pont;
```

```
#include "ponto.h"

void main()
{
    int *VI = new int[5];
    cout << "\nIntroduza 5 inteiros\n";
    for (int i = 0; i < 5; i ++)
        cin >> VI[i];
    for (i = 0; i < 5; i ++)
        cout << " VI[" << i << "]= " << *(VI + i);
    delete[ ] VI;
    TPonto *VP = new TPonto[3];
    cout << "\nIntroduza 3 pontos\n";
    for (i = 0; i < 3; i ++)
    {
        cout << "Introduza Ponto[" << i << "]= ";
        cin >> VP[i].X >> VP[i].Y;
    }
    for (i = 0; i < 3; i ++)
    {
        cout << "\nVP[" << i << "]= ";
        VP[i].Mostrar();
    }
    delete[ ] VP;
}
```

Atribuição, Passagem e Devolução de Referências para Objectos

- Uma referência representa um "ponteiro" implícito (não é um ponteiro), que para todos os efeitos actua como outro nome para uma variável.

```
#include "ponto.h"
    // Passagem e devolução de pontos por valor
TPonto PontoMedioValor(TPonto P1, TPonto P2)
{
    return TPonto((P2.X - P1.X) / 2, (P2.Y - P1.Y) / 2);
}
    // Passagem e devolução de ponteiros para pontos
TPonto *PontoMedioPont(TPonto *P1, TPonto *P2)
{
    return new TPonto((P2->X - P1->X) / 2, (P2->Y - P1->Y) / 2);
}
    // Passagem e devolução de referências para pontos
TPonto &PontoMedioRef(TPonto &P1, TPonto &P2)
{
    return * (new TPonto((P2.X - P1.X) / 2, (P2.Y - P1.Y) / 2));
}

void main()
{
    TPonto P1(2, 3), P2(8, 5);
    PontoMedioValor(P1, P2).Mostrar();
    PontoMedioPont(&P1, &P2)->Mostrar();
    PontoMedioRef(P1, P2).Mostrar();
}
```

- A passagem de variáveis por referência evita a cópia efectuada na passagem por valor, iniciação e devolução de variáveis.

Atribuição, Passagem e Devolução de Referências para Objectos

- A devolução de uma referência para uma variável permite a consulta, alteração do valor e acesso ao endereço desta.

```
#include <iostream.h>

class TPonto {
public:
    int   X, Y;
    TPonto() { cout << "\nConstruir Ponto Defeito\n"; X = 0; Y = 0; }
    TPonto(int x, int y) { cout << "\nConstruir Ponto\n"; X = x; Y = y; }
    ~TPonto() { cout << "\nDestruir Ponto"; }
    void Mostrar() { cout << "\nPonto=(" << X << ", " << Y << ")"; }
};

class TLinha {
    TPonto P1, P2;
public:
    TLinha() { cout << "\nConstruir LD\n"; P1=TPonto(); P2=TPonto();}
    TLinha(TPonto &p1, TPonto &p2);
    ~TLinha() { cout << "\nDestruir Linha"; }
    void Mostrar() { cout << "\nLinha"; P1.Mostrar(); P2.Mostrar(); }
    TPonto &Ponto1() { return P1; }  // Devolve uma referência para P1
    TPonto &Ponto2() { return P2; }  // Devolve uma referência para P2
};

TLinha::TLinha(TPonto &p1, TPonto &p2) {
    cout << "\nConstruir Linha";
    P1 = p1; P2 = p2;
}

void main() {
    TLinha L(TPonto(2, 6), TPonto(7, 4));
    L.Ponto1().X = 5;              // Equivalente a  L.P1.X = 5;
    L.Ponto2() = TPonto(1, 1);     // Equivalente a  L.P2 = TPonto(1, 1)
    L.Mostrar();                   // Se P1 e P2 fossem públicos
}
```

Atribuição, Passagem e Devolução de Referências para Objectos

```
#include "ponto.h"

class TVectorPontos
{
      int NumPontos;
      TPonto *VP;
public:
      TVectorPontos(int np);
      ~TVectorPontos() { delete[ ] VP; }
      TPonto &Colocar(int i);
      TPonto Consultar(int i);
};

TVectorPontos::TVectorPontos(int np)
{
      VP = new TPonto[NumPontos = np];
      if (!VP)
      {
            cout << "\nErro de Alocação";
            exit(1);
      }
}

      // Permite consultar e alterar o ponto na posição i
TPonto &TVectorPontos::Colocar(int i)
{
      if (i < 0  ||  i >= NumPontos)
      {
            cout << "\nPosição incorrecta";
            exit(1);
      }
      return VP[i];
}
```


Atribuição, Passagem e Devolução de Referências para Objectos

```
// Permite consultar o ponto na posição i
TPonto TVectorPontos::Consultar(int i)
{
      if (i < 0  ||  i >= NumPontos)
      {
            cout << "\nPosição incorrecta";
            exit(1);
      }
      return VP[i];
}

void main()
{
      TVectorPontos V(4);  // não confundir com V[4]
      for (int i = 0; i < 4; i ++)
            cin >> V.Colocar(i).X >> V.Colocar(i).Y;
      V.Colocar(2) = TPonto(8, 8); // Equivalente a V.VP[2]= TPonto(8, 8);
      V.Colocar(0).Mostrar();      // Equivalente a V.VP[0].Mostrar();
      for (i = 1; i < 4; i ++)     // Se VP fosse público
            V.Consultar(i).Mostrar();  // O método Consultar é dispensável
}
```

- Deve ter em atenção o seguinte erro:

```
int &f()
{
      int x;
      return x;      // Devolve a referência para uma variável local
}               // Solução: alocação dinâmica

void main()
{
      f() = 10;    // Atribuição de um valor a uma variável desalocada
}
```

Overloading dos Construtores

- Construtor por defeito - utilizado para declarar objectos ou vectores de objectos sem definir o seu estado inicial

```
TPonto P();   // equivalente a TPonto P;
TPonto P[8];
```

- Construtor com argumentos - utilizado para declarar objectos definindo o seu estado inicial

```
TLinha(P1, P2);       // Executa TLinha(TPonto, TPonto)
TLinha(2, 6, -4, 8);  // Executa TLinha(int, int, int, int)
```

- Construtor por cópia - utilizado na iniciação de objectos que pode ocorrer dos seguintes três modos:
  - ↓ Quando um objecto é usado para iniciar outro

```
TPonto P2 = P1;       // Invoca TPonto(P1)
```

  - ↓ Quando um objecto é passado por valor numa função

```
P1.Alterar(P2);       // Invoca TPonto(P2)
```

  - ↓ Quando um objecto é usado para iniciar outro

```
P=L.PontoMedio();  // Invoca TPonto(L.PontoMedio())
```

A forma geral do construtor por cópia é:

```
nomeclasse::nomeclasse(const nomeclasse   &obj)
{
    // implementação
}
```

ATENÇÃO: O construtor por cópia não afecta o operador de atribuição.Exemplo: P2=P1; // Invoca operador de atribuição

Overloading dos Construtores

```
#include <iostream.h>

class TPonto {
public:
    int   X, Y;
    TPonto() { cout << "\nConstruir Ponto Defeito"; X = 0; Y = 0; }
    TPonto(int x, int y) { cout << "\nConstruir Ponto"; X = x; Y = y; }
    TPonto(const TPonto &p) { cout << "\nConstruir Ponto Cópia";
                              X = p.X;   Y = p.Y; }
    ~TPonto() { cout << "\nDestruir Ponto"; }
    void Mostrar() { cout << "\nPonto=(" << X << ", " << Y << ")"; }
};

class TLinha {
    TPonto P1, P2;
public:
    TLinha() { cout << "\nConstruir Linha Defeito";
               P1 = TPonto(); P2 = TPonto(); }
    TLinha(const TPonto &p1, const TPonto &p2)
           {cout << "\nConstruir Linha"; P1 = p1; P2 = p2; }
    TLinha(int xi, int yi, int xf, int yf) {cout << "\nConstruir Linha 2";
               P1 = TPonto(xi, yi); P2 = TPonto(xf, yf); }
    TLinha(const TLinha &lin) { cout << "\nConstruir Linha Cópia";
                                P1 = lin.P1; P2 = lin.P2; }
    ~TLinha() { cout << "\nDestruir Linha"; }
    void Mostrar() { cout << "\nLinha"; P1.Mostrar(); P2.Mostrar(); }
    TPonto Dif(TPonto p) { return TPonto(p.X - P1.X, p.Y - P2.Y); }
};

void main()
{
    TPonto P1 = TPonto(2, 6), P2 = P1;    // Invoca o construtor por cópia
    TLinha L(P1, TPonto(3, -4));
    P2 = L.Dif(P1);
    P1 = P2;            // Não invoca o construtor por cópia
}
```

Overloading dos Construtores

- Na definição do protótipo de uma função podemos definir um valor por defeito para os últimos parâmetros desta.

  Ident(char *nome = “”, int i = 0; char c = ‘A’);

  Permite invocar:

  Ident(“Abc”, 1, ´B´);

  Ident(“Abc”, 1);

  Ident(“Abc”);

  Ident( );

  Erros:

  Ident(1); // Chamada incorrecta

  Ident(‘B’); // Chamada incorrecta

  Ident(int a = 0; char c); // Declaração incorrecta

- Podemos substituir o construtor por defeito da classe TPonto por: ***TPonto(int x = 0; int y = 0);***

  o que tem como vantagem poder invocar:

  TPonto(), TPonto(X) e TPonto(X, Y)

- Podemos substituir o construtor por defeito da classe TLinha por: ***TLinha(const TPonto &p1 = TPonto(), const TPonto &p2 = TPonto());***

  o que tem como vantagem poder invdocar:

  TLinha(), TLinha(P1) e TLinha(P1, P2)

Overloading e Ambiguidades

- Conversão automática

```
#include <iostream.h>
float f(float a) { return (2 * a); }
float f(double b) { return (3 * b); }

void main() {
    float x = 1.5;
    double y = 1.5;
    cout << f(x);        // Invoca f(float);
    cout << f(y);        // Invoca f(double);
    f(10);               // Ambiguidade - converte 10 para float ou double?
}
```

- Passagem por valor e por referência

```
#include <iostream.h>
char f(char a, int b) { return (a + b); }
char f(char &a, int b) { return (a += b); }

void main() {
    int c = 'G';
    cout << f('K', 2);   // Invoca f(char, int)
    cout << f(c, 3);     // Ambiguidade
}
```

- Argumentos por defeito

```
#include <iostream.h>
char f(char a) { return a; }
char f(char a, int b = 1) { return (a + b); }

void main() {
    cout << f('A', 5);   // Invoca f(char, int)
    cout << f('C');      // Ambiguidade
}
```

Endereço de uma Função *Overloading*

```
#include <iostream.h>

char Fun(char a) { return a; }

char Fun(char a, int b) { return (a + b); }

void main()
{
      char (* PFun1)(char) = Fun;
      char (* PFun2)(char, int) = Fun;
      cout << "\nEndereco de Fun(char): " << PFun1;
      cout << "\nEndereco de Fun(char, int): " << PFun2;
      char C;
      int I;
      cout << "\nIntroduza C: ";
      cin >> C;
      cout << "\nIntroduza I: ";
      cin >> I;
      cout << "\nFun(C): " << (*PFun1)(C);
      cout << "\nFun(C, I): " << (*PFun2)(C, I);
}
```


Overloading de Operadores

- O *overloading* de operadores tem a forma geral:

```
tipo nomeclasse::operator nomeoperador(argumentos)
{
    // implementação
}
```

- Não pode efectuar o overloading dos operadores: . :: .* ?

```
#include <iostream.h>

class Simples
{
      int I;
public:
      Simples(int i) { I = i; }
      void Mostrar() { cout << "\nI= " << I; }
      Simples  operator+(const Simples &s);
};

Simples  Simples::operator+(const Simples &s)
{
      return Simples(I + s.I);
}

void main()
{
      Simples S1(2), S2 = 3;
      Simples S3 = S1 + S2;
      cout << "\nS1 + S2= ";
      S3.Mostrar();
      cout << "\nS3 + S1= ";
      (S3 + S1).Mostrar();
}
```

Overloading de Operadores Binários

```
#include <iostream.h>

class TPonto {
      int   X, Y;
public:
      TPonto(int x=0, int y=0) { cout << "\nConstruir Ponto"; X=x; Y=y; }
      ~TPonto() { cout << "\nDestruir Ponto"; }
      void Mostrar() { cout << "\nPonto=(" << X << ", " << Y << ")"; }
      TPonto operator+(const TPonto &p);
      TPonto operator+(int i);
      TPonto operator=(const TPonto &p);
};

TPonto TPonto::operator+(const TPonto &p)
{  return TPonto (X + p.X, Y + p.Y); }

TPonto TPonto::operator+(int i)
{ return TPonto (X + i, Y + i); }

TPonto TPonto::operator=(const TPonto &p) {
      X = p.X;
      Y = p.Y;
      return *this;
}

void main() {
      TPonto P1(2, 3), P2(4, 5), P3;
      P3 = P1 + P2;                 // Invoca operator+(const TPonto &p)
      P3.Mostrar();
      P2 + 10;                      // Invoca operator+(int)
                                    // 10 + P2 ???
      P2.Mostrar();
      P1 = P3;
      P1.Mostrar();
}
```


Overloading de Operadores Unários

```
#include <iostream.h>

class TPonto
{    // Ver página anterior
      TPonto operator-();
      TPonto operator++();                          // Pré-incremento:   ++p
      TPonto operator++(int /* nao usado */);  // Pós-incremento:   p++
};

TPonto TPonto::operator-() {
      return TPonto (-X, -Y);       // Cuidado se os argumentos forem
                                    // dependentes - TPonto(++X, X)
}

TPonto TPonto::operator++() {
      ++X;
      ++Y;
      return *this;
}

TPonto TPonto::operator++(int) {
      return TPonto (X++, Y++);
}

void main() {
      TPonto P1(1, 2), P2(3, 4), P3;
      P3 = -P1;                     // Invoca operator-()
      P3.Mostrar();
      P3 = ++P2;                    // Invoca operator++()
      P3.Mostrar();
      P3 = P1++;                    // Invoca operator++(int)
      P3.Mostrar();
      P1.Mostrar();
}
```

Overloading de Operadores Relacionais e Lógicos

```
#include <iostream.h>

class TPonto
{    // Ver página nº 42
     int operator==(const TPonto &p);
     int operator<(const TPonto &p);
     int operator&&(const TPonto &p);
};

int TPonto::operator==(const TPonto &p)
{
     return (X == p.X  &&  Y == p.Y);
}

int TPonto::operator<(const TPonto &p)
{
     return (X < p.X  ||  (X == p.X  &&  Y < p.Y));
}

int TPonto::operator&&(const TPonto &p)
{
     return ((X && p.X)  &&  (Y && p.Y));
}

void main()
{
     TPonto P1, P2;
     cout << "\nIntroduza 1º ponto:";
     cin >> P1.X >> P1.Y;
     cout << "\nIntroduza 2º ponto:";
     cin >> P2.X >> P2.Y;
     cout << (P1 == P2 ? "\nIguais" : "\nDiferentes");
     cout << (P1 < P2 ? "\nMenor" : "\nMaior ou igual");
     cout << (P1 && P2 ? "\nP1 && P2" : "\n~(P1 && P2)");
}
```


Overloading de Operadores *friend* e de Conversão de Tipos

```
#include <iostream.h>
#include <stdio.h>

class TPonto {
     int   X, Y;
public:
     TPonto(int x=0, int y=0) { cout << "\nConstruir Ponto"; X=x; Y=y; }
     ~TPonto() { cout << "\nDestruir Ponto"; }
     friend TPonto operator+(int i, const TPonto &p);          // i + p;
     friend ostream &operator<<(ostream& os, const TPonto &p);//os<<p;
     friend istream &operator>>(istream& is, TPonto &p);    // is >> p;
     operator int() { return (X * Y); }     // Conversão para inteiro -   (int) p
};

TPonto operator+(int i, const TPonto &p) {
     return TPonto(p.X + i, p.Y + i);
}

ostream &operator<<(ostream& os, const TPonto &p) {
     os << "Ponto=(" << p.X << ", " << p.Y << ") ";
     return os;
}

istream &operator>>(istream& is, TPonto &p) {
     is >> p.X >> p.Y;
     return is;
}

void main() {
     TPonto P1, P2;
     cout << "\nIntroduza 2 pontos:\n";
     cin >> P1 >> P2;
     cout << (3 + P1); // Invoca operator(int, const TPonto&)
     int I = P1, J;  // Conversão automática (int) P1
     J = (int) P2 + 4;  // Conversão explícita
     cout << "\nI= " << I << " J= " << J;
}
```

Tipo de Acesso aos Membros de uma Classe

- Os membros privados de uma classe não podem ser acedidos pelos objectos instanciados, nem pelas classes derivadas.
- Os membros protegidos de uma classe não podem ser acedidos pelos objectos instanciados, podendo ser acedidos pelas classes derivadas.
- Os membros públicos de uma classe podem ser acedidos pelos objectos instanciados e pelas classes derivadas.

```
#include <iostream.h>

class Base {
      char Codigo;
protected:
      int Numero;
public:
      void Atribuir(char codigo, int numero);
};

void Base::Atribuir(char codigo, int numero)
{
      Codigo = codigo;
      Numero = numero;
}

void main()
{
      Base B;
      B.Atribuir('K', 12);
      cout << B.Codigo;    //  Erro, pois Codigo é um membro privado
      cout << B.Numero;    //  Erro, pois Numero é um membro protegido
}
```


Tipo de Acesso à Classe Base

- A forma geral de derivação de uma classe é:

```
class nomeclassederivada : acesso nomeclassebase
{
    // declaração
};
```

onde o *acesso* pode ser:

- ↓ *public* (público) - A classe derivada tem acesso protegido e público, respectivamente, aos membros protegidos e públicos da classe base.
- ↓ *protected* (protegido) - A classe derivada tem acesso protegido aos membros protegidos e públicos da classe base.
- ↓ *private* (privado) - A classe derivada tem acesso privado aos membros protegidos e públicos da classe base.

Acesso Público à Classe Base

```
#include <iostream.h>
#include <stdio.h>

class Base {
      char Codigo;
protected:
      int Numero;
public:
      void Atribuir(char codigo, int numero);
};

void Base::Atribuir(char codigo, int numero)
{ Codigo = codigo;   Numero = numero; }

class DerivadaPublico : public Base {
      char Ident[20];
protected:    // Acesso protegido aos membros protegidos da classe Base
      int Chave;
public:       // Acesso público aos membros públicos da classe Base
      void Atribuir(char codigo, int numero, int chave);
};

void DerivadaPublico::Atribuir(char codigo, int numero, int chave) {
      Base::Atribuir(codigo, numero);
      Chave = chave;
      sprintf(Ident, "%i%c%i", Chave, Codigo, Numero);
}           // Erro, pois Codigo não é acessível na classe DerivadaPublico

void main() {
      DerivadaPublico D;
      D.Base::Atribuir('K', 12);   // Acesso à função Atribuir da classe Base
      cout << D.Numero;  // Erro, pois Numero é protegido na classe Base
      D.Atribuir('G', 5, 8);
      cout << D.Chave;     //  Erro, pois Chave é protegido
}
```


Acesso Protegido à Classe Base

```
#include <iostream.h>
#include <stdio.h>

class Base {
      char Codigo;
protected:
      int Numero;
public:
      void Atribuir(char codigo, int numero);
};

void Base::Atribuir(char codigo, int numero)
{ Codigo = codigo;   Numero = numero; }

class DerivadaProtegido : protected Base {
      char Ident[20];
protected:    // Acesso protegido aos membros protegidos
              // e públicos da classe Base
      int Chave;
public:
      void Atribuir(char codigo, int numero, int chave);
};

void DerivadaProtegido::Atribuir(char codigo, int numero, int chave) {
      Base::Atribuir(codigo, numero);
      Chave = chave;
      sprintf(Ident, "%i%c%i", Chave, Codigo, Numero);
}           // Erro, pois Codigo não é acessível na classe DerivadaProtegido

void main() {
      DerivadaProtegido D;
      D.Base::Atribuir('K', 12);    // Erro, pois Base::Atribuir() é protegido
      cout << D.Numero;   // Erro, pois Numero é protegido
      D.Atribuir('G', 5, 8);
      cout << D.Chave;      //  Erro, pois Chave é protegido
}
```

Acesso Privado à Classe Base

```
#include <iostream.h>
#include <stdio.h>

class Base {
      char Codigo;
protected:
      int Numero;
public:
      void Atribuir(char codigo, int numero);
};

void Base::Atribuir(char codigo, int numero)
{ Codigo = codigo;   Numero = numero; }

class DerivadaPrivado : private Base
{       // Acesso privado aos membros protegidos e públicos da classe Base
      char Ident[20];
protected:
      int Chave;
public:
      void Atribuir(char codigo, int numero, int chave);
};

void DerivadaPrivado::Atribuir(char codigo, int numero, int chave) {
      Base::Atribuir(codigo, numero);
      Chave = chave;
      sprintf(Ident, "%i%c%i", Chave, Codigo, Numero);
}           // Erro, pois Codigo não é acessível na classe DerivadaPrivado

void main() {
      DerivadaPrivado D;
      D.Base::Atribuir('K', 12); // Erro, pois Base::Atribuir() é privado
      cout << D.Numero;  // Erro, pois Numero é privado
      D.Atribuir('G', 5, 8);
      cout << D.Chave;     //  Erro, pois Chave é protegido
}
```


Lista de Iniciações do Construtor

- Um construtor da classe **A** pode ter uma lista de iniciações, que permite invocar os construtores das classes bases e das variáveis membro da classe **A**, antes da execução da primeira instrução do construtor **A**. A forma geral de um construtor é:

```
nomeclassederivada(args)
  : nomeclassebase1(vals), ..., nomeclassebaseN(vals),
    variávelmembro1(vals), ..., variávelmembroM(vals)
{    // Implementação
};
```

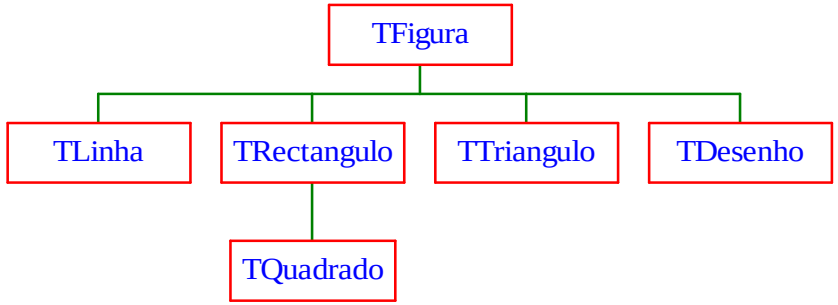
```
#include <iostream.h>

class TPonto  {
public:
      int   X, Y;
      TPonto(int x = 0, int y = 0) : X(x), Y(y)   // Iniciação de X e Y
            { cout << "\nConstruir Ponto\n";}
      ~TPonto() { cout << "\nDestruir Ponto"; }
};

class TLinha  {
      TPonto P1, P2;
public:
      TLinha(const TPonto &p1= TPonto(), const TPonto &p2= TPonto());
      ~TLinha() { cout << "\nDestruir Linha"; }
};
      // Iniciação de P1 e P2 através da lista de iniciações de TLinha
TLinha::TLinha(const TPonto &p1, const TPonto &p2) : P1(p1), P2(p2)
{ cout << "\nConstruir Linha"; }

void main() {
      TLinha L(TPonto(2, 6), TPonto(7, 4));
}
```

Hierarquia de Figuras




```
#include "ponto.h"
#include <stdlib.h>

class TFigura
{
    TPonto Posicao;
protected:
    TFigura(const TPonto &p) : Posicao(p) { cout << "\nC. Figura"; }
public:
    ~TFigura() { cout << "\nD. Figura"; }
    TPonto Localizacao() { return Posicao; }
    void Mover(const TPonto &p) { Posicao += p; }
};
```

- 1ª versão da classe TLinha

```
class TLinha
{
    TPonto P1, P2;
public:
    TLinha(const TPonto &p1 = TPonto(), const TPonto &p2 = TPonto())
        : P1(p1), P2(p2) { cout << "\nC. Linha"; }
    ~TLinha() { cout << "\nD. Linha"; }
    void Mover(const TPonto &p) { P1 += p; P2 += p; }
};
```


Hierarquia de Figuras

- Versão da classe TLinha derivando da classe TFigura

```
class TLinha : public TFigura
{
    TPonto P2;
public:
    TLinha(const TPonto &p1 = TPonto(), const TPonto &p2 = TPonto())
        : TFigura(p1), P2(p2) { cout << "\nC. Linha"; }
    ~TLinha() { cout << "\nD. Linha"; }
    void Mover(const TPonto &p) { TFigura::Mover(p); P2 += p; }
};
```

- 1ª versão da classe TTriangulo

```
class TTriangulo {
    TPonto P1, P2, P3;
public:
    TTriangulo(const TPonto &p1 = TPonto(),
            const TPonto &p2 = TPonto(), const TPonto &p3 = TPonto())
        : P1(p1), P2(p2), P3(p3) { cout << "\nC. Triângulo"; }
    ~TTriangulo() { cout << "\nD. Triângulo"; }
    void Mover(const TPonto &p) { P1 += p; P2 += p; P3 += p; }
};
```

- Versão da classe TTriangulo derivando da classe TFigura

```
class TTriangulo : public TFigura {
    TPonto P2, P3;
public:
    TTriangulo(const TPonto &p1=TPonto(),
            const TPonto &p2=TPonto(), const TPonto &p3 = TPonto())
        : TFigura(p1), P2(p2), P3(p3) { cout << "\nC. Triângulo"; }
    ~TTriangulo() { cout << "\nD. Triângulo"; }
    void Mover(const TPonto &p) {TFigura::Mover(p); P2+=p; P3+=p; }
};
```

Hierarquia de Figuras

- Versão da classe TRectangulo derivando da classe TFigura

```
class TRectangulo : public TFigura {
    TPonto P2;
public:
    TRectangulo(const TPonto &p1 = TPonto(),
                const TPonto &p2 = TPonto())
        : TFigura(p1), P2(p2) { cout << "\nC. Rectângulo"; }
    ~TRectangulo() { cout << "\nD. Rectângulo"; }
    void Mover(const TPonto &p) { TFigura::Mover(p); P2 += p; }
};
```

- Versão da classe TQuadrado derivando da classe TFigura

```
class TQuadrado : public TRectangulo
{
public:
    TQuadrado(const TPonto &p1 = TPonto(),
              const TPonto &p2 = TPonto())
        : TRectangulo(p1, p2) { cout << "\nC. Quadrado"; }
    ~TQuadrado() { cout << "\nD. Quadrado"; }
};
```

- Versão da classe TDesenho derivando da classe TFigura

```
class TDesenho : public TFigura
{
    TFigura **VF;     // Vector de ponteiros para figuras
    int MaxFig, NumFig;
public:
    TDesenho(int mf, const TPonto &p);
    ~TDesenho();
    void Inserir(TFigura &f);
    void Mover(const TPonto &p);
};
```


Hierarquia de Figuras

```
TDesenho::TDesenho(int mf, const TPonto &p = TPonto())
    : TFigura(p), MaxFig(mf), NumFig(0)
{
    cout << "\nC. Desenho";
    VF = new (TFigura **)[MaxFig];
}

TDesenho::~TDesenho()
{
    cout << "\nD. Desenho";
    delete[] VF;
}

void TDesenho::Inserir(TFigura &f)
{
    if (NumFig == MaxFig)
    {
        cerr << "\nCapacidade do desenho excedida";
        exit(1);
    }
    VF[NumFig++] = &f;
}

void TDesenho::Mover(const TPonto &p)
{
    TFigura::Mover(p);
    for (int i = 0; i < NumFig; i++)
        VF[i]->TFigura::Mover(p);
}
```

Hierarquia de Figuras

```
void main()
{
      TDesenho Des(4);
      TPonto P1, P2, P3;
      cout << "\nIntroduza 1º ponto: ";
      cin >> P1;
      cout << "\nIntroduza 2º ponto: ";
      cin >> P2;
      cout << "\nIntroduza 3º ponto: ";
      cin >> P3;
      TLinha L(P1, P2);
      TRectangulo R(P1, P3);
      TTriangulo T(P1, P2, P3);
      TQuadrado Q(P2, P3);
      Des.Inserir(L);
      Des.Inserir(R);
      Des.Inserir(T);
      Des.Inserir(Q);
      Des.Mover(P2);
}
```


Herança Múltipla

```
#include <iostream.h>

class Chave {
      char C;
public:
      Chave(char c) : C(c) { cout << "\nC. Chave"; }
      ~Chave() { cout << "\nD. Chave"; }
      void Mostrar() { cout << "\nChave: " << C; }
};

class Valor {
      int I;
public:
      Valor(int i) : I(i) { cout << "\nC. Valor"; }
      ~Valor() { cout << "\nD. Valor"; }
      void Mostrar() { cout << "\nValor: " << I; }
};

class ChaveValor : public Chave, public Valor
{
public:
      ChaveValor(char c, int i);
      ~ChaveValor() { cout << "\nD. ChaveValor"; }
      void Mostrar() { Chave::Mostrar(); Valor::Mostrar(); }
};          // Invoca métodos Mostrar das classes bases

ChaveValor::ChaveValor(char c, int i) : Chave(c), Valor(i)
{
      cout << "\nC. ChaveValor";
}

void main()
{
      ChaveValor CV('D', 4);
      CV.Mostrar();
}
```

Herança Múltipla

- Ao pretender criar um triângulo rectângulo a partir de um triângulo e de um rectângulo surge a duplicação da informação da classe TFigura, pois é a classe base de ambos.

```
class TTrianguloRectangulo : public TTriangulo, public TRectangulo
{
public:
    TTrianguloRectangulo(const TPonto &p1 = TPonto(),
            const TPonto &p2 = TPonto(), const TPonto &p3 = TPonto());
    ~TTrianguloRectangulo() { cout << "\nD. Triângulo Rectângulo"; }
    void Mover(const TPonto &p) { TTriangulo::Mover(p); }
};

    // Ao construir um triângulo rectângulo são construídas duas figuras
    // uma para o triângulo e outra para o rectângulo
TTrianguloRectangulo::TTrianguloRectangulo(const TPonto &p1,
                                    const TPonto &p2, const TPonto &p3)
        : TTriangulo(p1, p2, p3), TRectangulo(p1, p2)
{
    cout << "\nC. Triângulo Rectângulo";
}

void main()
{
    TPonto P1, P2, P3;
    cout << "\nIntroduza 1º ponto: ";
    cin >> P1;
    cout << "\nIntroduza 2º ponto: ";
    cin >> P2;
    cout << "\nIntroduza 3º ponto: ";
    cin >> P3;
    TTrianguloRectangulo TR(P1, P2, P3);
    TR.Mover(P2);
}
```


Classes de Base Virtual

- Para evitar a duplicação de informação é necessário que as classes TTriangulo e TRectangulo derivem virtualmente da classe TFigura.

```
class TTriangulo : virtual public TFigura { ... };

class TRectangulo : virtual public TFigura { ... };

class TTrianguloRectangulo : public TTriangulo, public TRectangulo {
public:
    TTrianguloRectangulo(const TPonto &p1 = TPonto(),
            const TPonto &p2 = TPonto(), const TPonto &p3 = TPonto());
    ~TTrianguloRectangulo() { cout << "\nD. Triângulo Rectângulo"; }
    void Mover(const TPonto &p) { TTriangulo::Mover(p); }
};

   // Ao construir um triângulo rectângulo é apenas construída uma figura
TTrianguloRectangulo::TTrianguloRectangulo(const TPonto &p1,
                                    const TPonto &p2, const TPonto &p3)
        : TFigura (p1), TTriangulo(p1, p2, p3), TRectangulo(p1, p2)
{           // Figura tem que ser construída pela classe mais derivada
    cout << "\nC. Triângulo Rectângulo";
}

void main() {
    TPonto P1, P2, P3;
    cout << "\nIntroduza 1º ponto: ";
    cin >> P1;
    cout << "\nIntroduza 2º ponto: ";
    cin >> P2;
    cout << "\nIntroduza 3º ponto: ";
    cin >> P3;
    TTrianguloRectangulo TR(P1, P2, P3);
    TR.Mover(P2);
}
```

Funções Virtuais - Introdução

- Uma **função virtual** (ou função de interface) é um membro de uma classe que é declarado na classe base podendo ser redefinida nas classes derivadas.
- As funções virtuais permitem implementar o conceito de ***"uma interface, múltiplos métodos"***, isto é, facultar as potencialidades do polimorfismo de inclusão (*run-time*).
- Uma **função virtual** é definida colocando a palavra reservada ***virtual*** antes da sua declaração na classe.

```
class Exemplo
{
    ...
public:   // ou protected:
    virtual int func (char c);
};
```

- Nas classes derivadas não é necessário colocar a palavra chave ***virtual***, no entanto a sua colocação permite distinguir as funções de interface das restantes.
- Uma classe com uma função virtual é designada por classe virtual, classe de interface ou classe polimórfica.
- Não se deve confundir a redefinição de uma função virtual com o *overloading* de uma função, pois este último pode diferir no tipo e número dos parâmetros, o que não pode acontecer com uma função virtual, pois esta só pode ser reimplementada para as classes derivadas mantendo a mesma interface.

Exemplo Prático

```
#include <iostream.h>

class Base
{
    char C;
public:
    Base(char c) : C(c) { cout << "\nC. Base"; }
    char Valor();
    virtual int Resultado();
    virtual void Mostrar();
};

class DerivadaInt : public Base
{
    int I;
public:
    DerivadaInt(char c, int i) : Base(c), I(i)
        { cout << "\nC. DerivadaInt"; }
    char Valor();
    virtual int Resultado();
    virtual void Mostrar();
};

class DerivadaCar : public Base
{
    char K;
public:
    DerivadaCar(char c, char k) : Base(c), K(k)
        { cout << "\nC. DerivadaCar"; }
    virtual int Resultado();
};
```

Exemplo Prático

```
char Base::Valor()
{
      cout << "\nBase::Valor";
      return C;
}

int Base::Resultado()
{
      cout << "\nBase::Resultado";
      return C;
}

void Base::Mostrar()
{
      cout << "\nBase::Mostrar";
      cout << "\nValor= " << Valor();              // Invoca Base::Valor
      cout << "\nResultado= " << Resultado();      // Invoca Resultado da
                                                   // classe da instância
}

char DerivadaInt::Valor()
{
      cout << "\nDerivadaInt::Valor";
      return (Base::Valor() + I);
}

int DerivadaInt::Resultado()
{
      cout << "\nDerivadaInt::Resultado";
      return I;
}

void DerivadaInt::Mostrar()
{
      cout << "\nDerivadaInt::Mostrar";
      cout << "  Valor= " << Valor();              // Invoca Base::Valor
      cout << "  Resultado= " << Resultado();      // Invoca Resultado da
                                                   // classe da instância
}
```


Exemplo Prático

```
int DerivadaCar::Resultado()
{
      cout << "\nDerivadaCar::Resultado";
      return K;
}

void main()
{
      Base B('A');
      cout << "\nValor, Resultado e Mostrar B";
      cout << ": " << B.Valor();
      cout << ": " << B.Resultado();
      B.Mostrar();
      DerivadaInt DI('D', 3);
      cout << "\nValor, Resultado e Mostrar DI";
      cout << ": " << DI.Valor();
      cout << ": " << DI.Resultado();
      DI.Mostrar();
      DerivadaCar DC('A', 'B');
      cout << "\nValor, Resultado e Mostrar DC";
      cout << ": " << DC.Valor();
      cout << ": " << DC.Resultado();
      DC.Mostrar();
      Base *PB;
      PB = &DI;
      cout << "\nValor, Resultado e Mostrar PB=&DI";
      cout << ": " << PB->Valor();          // Invoca Base::Valor
      cout << ": " << PB->Resultado();      // Invoca DerivadaInt::Resultado
      PB->Mostrar();                        // Invoca DerivadaInt::Mostrar
      PB = &DC;
      cout << "\nValor, Resultado e Mostrar PB=&DC";
      cout << ": " << PB->Valor();          // Invoca Base::Valor
      cout << ": " << PB->Resultado();      // Invoca DerivadaCar::Resultado
      PB->Mostrar();                        // Invoca Base::Mostrar
}
```

Funções Virtuais Puras

- Uma **função virtual pura** é uma função virtual sem implementação de código na classe base, devendo ser implementada nas classes derivadas.
- Uma **função virtual pura** é definida do modo seguinte:

```
class Exemplo
{
    ...
public:   // ou protected:
    virtual char funcvp (int i) = 0;
};
```

- Uma classe com uma função virtual pura é designada por classe abstracta, pelo facto de ter pelos menos um método sem código concreto.
- Uma classe abstracta não pode ser instânciada, podendo, no entanto, ser declarados ponteiros para essa classe, apontando para instâncias das classes derivadas (desde que não sejam abstractas).

```
void main()
{
    BaseAbstracta   B;   // Erro, pois não se pode intânciar a
                         // classe BaseAbstracta
    DerivadaBaseAbstracta   D;
    BaseAbstracta   *PB = &D;
        // Correcto, pois PB é um ponteiro para a classe
        // BaseAbstracta, podendo apontar para instâncias
        // das classes derivadas
};
```


Polimorfismo

- O **polimorfismo** é o processo, pelo qual uma interface comum é aplicada em duas ou mais situações semelhantes, mas técnicamente diferentes, implementando a filosofia de ***"uma interface, múltiplos métodos"***.
- O **polimorfismo de inclusão** (relação "*is-a*") é processo, pelo qual um ponteiro para uma classe de interface pode apontar para instâncias das suas classes derivadas, permitindo através deste elaborar os métodos de interface das instâncias apontadas e não da classe de interface.

```
void main() {
    BaseInterface   *P;  // Desenhar() é método de interface
    DerivadaABaseInterface   A;
    DerivadaBBaseInterface   B;
    P = &A;   P->Desenhar();   // Invoca A.Desenhar()
    P = &B;   P->Desenhar();   // Invoca B.Desenhar()
}
```

- O **polimorfismo global** é processo, pelo qual se tem acesso aos métodos próprios das instâncias das classes derivadas apontadas por um ponteiro para uma classe de interface.

```
void main() {
    DerivadaABaseInterface   A, *PA;
    BaseInterface   *P = &A;
    PA=TYPESAFE_DOWNCAST(P, DerivadaABaseInterface);
    PA->MetodoProprio();   // Invoca A. MetodoProprio()
}
```

Binding

- ***Binding*** - Determinação do método a executar correspondente a um dado selector:
  - *Binding* estático (mais eficiente) - O compilador sabe qual é a classe do receptor e rigorosamente, qual o método a invocar (exemplo: TQuadrado Q(P1, P2); Q.Desenhar())
  - *Binding* dinâmico - O compilador sabe apenas que o receptor pertence a uma dada classe ou às suas classes derivadas. Verifica apenas se o método existe na classe (em *compile-time*), determinando em *run-time*, qual o selector apropriado. É utilizado em C++ sempre que se aplica o polimorfismo de inclusão (exemplo: TFigura *PF; PF = LerTipoObjecto(); PF->Desenhar();)
  - *Binding* tardio (menos eficiente) - O compilador não sabe qual a classe do receptor, pelo que tem que efectuar a sua escolha em *run-time* (utilizado em *SmallTalk*)

Hierarquia de Figuras Utilizando o Polimorfismo

- Na página 55, como o método Mover() não é virtual é invocado o método TFigura::Mover(), o que apenas move a localização do desenho e o primeiro ponto das figuras deste, e não as figuras como pretendido.

```
void TDesenho::Mover(const TPonto &p) {
    TFigura::Mover(p);
    for (int i = 0; i < NumFig; i++)
        VF[i]->TFigura::Mover(p);
}
```

- Na página 56, no programa principal ao movermos o desenho **Des**, apenas é movido o primeiro ponto de cada figura nele inserida.

```
void main() {
    TDesenho Des(4);
    TPonto P1, P2, P3;
    cout << "\nIntroduza 1º ponto: ";
    cin >> P1;
    cout << "\nIntroduza 2º ponto: ";
    cin >> P2;
    cout << "\nIntroduza 3º ponto: ";
    cin >> P3;
    TLinha L(P1, P2);
    TRectangulo R(P1, P3);
    TTriangulo T(P1, P2, P3);
    TQuadrado Q(P2, P3);
    Des.Inserir(L);
    Des.Inserir(R);
    Des.Inserir(T);
    Des.Inserir(Q);
    Des.Mover(P2);
}
```

Hierarquia de Figuras Utilizando o Polimorfismo

```
#include "ponto.h"
#include <stdlib.h>
#include <typeinfo.h>

#define TYPESAFE_DOWNCAST(P,T) (dynamic_cast<T *> (P))

class TFigura
{
protected:
    TPonto Posicao;
    TFigura(const TPonto &p = TPonto()) : Posicao(p) { }
public:
    enum { Linha, Rectangulo, Triangulo, Quadrado, Desenho };
    ~TFigura() { }
    TPonto Localizacao() { return Posicao; }
    virtual void Mover(const TPonto &p) { Posicao += p; }
    virtual void Desenhar(/* TDC *dc */) = 0;
    virtual int Tipo() = 0;
};

class TLinha : public TFigura
{
protected:
    TPonto P2;
public:
    TLinha(const TPonto &p1 = TPonto(), const TPonto &p2 = TPonto())
        : TFigura(p1), P2(p2) { }
    ~TLinha() { }
    virtual void Mover(const TPonto &p)
        { cout << "\nMover Linha"; TFigura::Mover(p); P2 += p; }
    virtual void Desenhar(/* TDC *dc */) { cout << "\nDesenhar Linha";}
    virtual int Tipo() { return TFigura::Linha; }
    int DistanciaX() { return (P2.X - Localizacao().X); }
};
```


Hierarquia de Figuras Utilizando o Polimorfismo

```
class TTriangulo : public TFigura
{
protected:
    TPonto P2, P3;
public:
    TTriangulo(const TPonto &p1 = TPonto(),
            const TPonto &p2 = TPonto(), const TPonto &p3 = TPonto())
        : TFigura(p1), P2(p2), P3(p3) { }
    ~TTriangulo() { }
    virtual void Mover(const TPonto &p) { cout << "\nMover Triangulo";
        TFigura::Mover(p); P2 += p; P3 += p; }
    virtual void Desenhar(/* TDC *dc */)
        { cout << "\nDesenhar Triangulo"; }
    virtual int Tipo() { return Triangulo; }
    TPonto Ponto3() { return P3; }
};

class TRectangulo : public TFigura
{
protected:
    TPonto P2;
public:
    TRectangulo(const TPonto &p1 = TPonto(),
                const TPonto &p2 = TPonto())
        : TFigura(p1), P2(p2) { }
    ~TRectangulo() { }
    virtual void Mover(const TPonto &p)
        { cout << "\nMover Rectangulo"; TFigura::Mover(p); P2 += p; }
    virtual void Desenhar(/* TDC *dc */)
        { cout << "\nDesenhar Rectangulo"; }
    virtual int Tipo() { return Rectangulo; }
    TLinha Diagonal() { return TLinha(Localizacao(), P2); }
};
```

Hierarquia de Figuras Utilizando o Polimorfismo

```
class TQuadrado : public TRectangulo
{
public:
      TQuadrado(const TPonto &p1 = TPonto(),
                  const TPonto &p2 = TPonto())
          : TRectangulo(p1, p2) { }
      ~TQuadrado() { }
      virtual int Tipo() { return Quadrado; }
      TLinha Lado() { return TLinha(TPonto(P2.X, Localizacao().Y), P2);}
};


class TDesenho : public TFigura
{
protected:
      TFigura **VF;  // Vector de ponteiros
      int MaxFig, NumFig;
public:
      TDesenho(int mf, const TPonto &p = TPonto());
      ~TDesenho();
      virtual void Mover(const TPonto &p);
      virtual void Desenhar(/* TDC *dc */);
      virtual int Tipo() { return Desenho; }
      void Inserir(TFigura *f);
      TFigura *Figura(int i) { return VF[i]; }
};

TDesenho::TDesenho(int mf, const TPonto &p)
      : TFigura(p), MaxFig(mf), NumFig(0)
{
      VF = new (TFigura **)[MaxFig];
}
```


POO

Hierarquia de Figuras Utilizando o Polimorfismo

```
TDesenho::~TDesenho()
{
      delete[] VF;
}

void TDesenho::Inserir(TFigura *f)
{
      if (NumFig == MaxFig)
      {
            cerr << "\nCapacidade do desenho excedida";
            exit(1);
      }
      VF[NumFig++] = f;
}

void TDesenho::Mover(const TPonto &p)
{
      cout << "\nMover Desenho";
      TFigura::Mover(p);            // Move a localizacao do desenho
      for (int i = 0; i < NumFig; i++)
            VF[i]->Mover(p);        // Move a figura i, invocando o
                                    // respectivo método virtual Mover()
}                                   // Polimorfismo de inclusão

void TDesenho::Desenhar(/* TDC *dc */)
{
      cout << "\nDesenhar Desenho";
      for (int i = 0; i < NumFig; i++)
            VF[i]->Desenhar();      // Desenha a figura i, invocando o
                                    // respectivo metodo virtual Desenhar()
}                                   // Polimorfismo de inclusão
```

Hierarquia de Figuras Utilizando o Polimorfismo

```
void main()
{
    TDesenho Des(5);
    TPonto P1, P2, P3;
    cout << "\nIntroduza 1º ponto: ";
    cin >> P1;
    cout << "\nIntroduza 2º ponto: ";
    cin >> P2;
    cout << "\nIntroduza 3º ponto: ";
    cin >> P3;
    TLinha L(P1, P2);
    TRectangulo R(P1, P3);
    TTriangulo T(P1, P2, P3);
    TQuadrado Q(P2, P3);
    Des.Inserir(&L);
    Des.Inserir(&R);
    Des.Inserir(&T);
    Des.Inserir(&Q);
    cout << "\nEscolha tipo de objecto [0..3]: ";
    int Tp;
    cin >> Tp;
    TFigura *PF;
    switch (Tp)
    {
        case TFigura::Linha :
            PF = new TLinha(P1, P3);
        break;
        case TFigura::Rectangulo :
            PF = new TRectangulo(P1, P3);
        break;
        case TFigura::Triangulo :
            PF = new TTriangulo(P1, P2, P3);
        break;
        case TFigura::Quadrado :
            PF = new TQuadrado(P1, P3);
    }
```


Hierarquia de Figuras Utilizando o Polimorfismo

```
    Des.Inserir(PF);
    Des.Desenhar();
    Des.Mover(P2);
    for (int I = 0; I < 5; I++)
        switch (Des.Figura(I)->Tipo())
        {
            // Acesso aos métodos próprios das figuras
            // Polimorfismo Global
            case TFigura::Linha :
                cout << "\nDX= " <<
                    TYPESAFE_DOWNCAST(Des.Figura(I), TLinha)
                        ->DistanciaX();
            // Note-se que Des.Figura(3)->DistanciaX() não é acessival,
            // pois Des.Figura(3) é um ponteiro para uma TFigura
            break;
            case TFigura::Rectangulo :
                TYPESAFE_DOWNCAST(Des.Figura(I), TRectangulo)
                    ->Diagonal().Mover(P3);
            break;
            case TFigura::Triangulo :
                cout << "\nPonto3= " <<

    TYPESAFE_DOWNCAST(Des.Figura(I),TTriangulo)
                        ->Ponto3();
            break;
            case TFigura::Quadrado :
                TQuadrado   *PQ =
                  TYPESAFE_DOWNCAST(Des.Figura(I),TQuadrado);
                PQ->Diagonal().Mover(P3);
                PQ->Lado().Mover(P3);
            break;
        }
}
```

Entrada e Saída Formatada

- O sistema de entradas/saídas em *C++* é definido pela hierarquia de classes derivadas da classe ***ios*** definida no ficheiro *iostream.h*

- ***Flags* de formato de entrada/saída em *C++***
  - *skipws* - define o desprezo dos caracteres espaço na leitura
  - *left* - define escrita justificada à esquerda
  - *right* - define escrita justificada à direita
  - *internal* - define escrita com preenchimento com espaços brancos se necessário
  - *dec* - define escrita em decimal
  - *oct* - define escrita em octal
  - *hex* - define escrita em hexadecimal
  - *showbase* - define a escrita da base numérica
  - *showpoint* - define a escrita obrigatória do ponto decimal
  - *uppercase* - define a escrita da base em maiúsculas
  - *scientific* - define a escrita em notação científica
  - *fixed* - define a escrita em notação normal
  - *unitbuf* - define a actualização da informação em disco (*flush*) após cada instrução de saída
  - *stdio* - define a actualização da informação em disco dos ficheiros stdout e stderr

Entrada e Saída Formatada

- As flags de formato de leitura e escrita em C++ são enumeradas na classe ***ios***.

- **Ficheiros padrão em *C++***
  - *cin* - entrada padrão (teclado por defeito)
  - *cout* - saída padrão (ecran por defeito)
  - *cerr* - saída de erro padrão (ecran por defeito)
  - *clog* - saída de erro padrão *bufferizada* (ecran por defeito)

- **Funções de manipulação do formato de E/S (classe *ios*)**
  - long ***setf*** (long *flags*);
    Activa as *flags* e devolve o formato anterior.
  - long ***unsetf*** (long *flags*);
    Desactiva as *flags* e devolve o formato anterior.
  - long ***flags*** ();
    Devolve o formato actual.
  - long ***flags*** (long *flags*);
    Estipula o formato através das *flags* e devolve o formato anterior. A utilização deste método implica a desactivação de todas as flags anteriormente activadas.

Entrada e Saída Formatada

```
#include <iostream.h>

void main()
{
    cout << "\nReais= " << -32.83 << " " << 32.83;
    cout << "\nInteiros" << -30 << " " << 30;
    cout.setf(ios::oct | ios::showpos | ios::scientific);
    cout << "\nReais: " << -32.83 << " " << 32.83;
    cout << "\nInteiros: " << -30 << " " << 30;
    cout.unsetf(ios::showpos | ios::oct);
    cout << "\nReais: " << -32.83 << " " << 32.83;
    cout << "\nInteiros: " << -30 << " " << 30;
    cout.flags(ios::fixed);
    cout << "\nReais: " << -32.83 << " " << 32.83;
    cout << "\nInteiros: " << -30 << " " << 30;
    cout << "\nios::scientific= " << (cout.flags() & ios::scientific);
}
```

- Produz a seguinte saída:

```
Reais=  -32.83  32.83
Inteiros=  -30  30
Reais=  -3.283000e+01  +3.283000e+01
Inteiros=  37777777742  36
Reais=  -3.283000e+01  3.283000e+01
Inteiros=  -30  30
Reais=  -32.830000  32.830000
Inteiros=  -30  30
ios::scientific=  0
```


Formatação do Tamanho, Comprimento e Preenchimento

- int ***width*** (int *w*);

  Fixa o tamanho do campo para escrita e devolve o tamanho anterior.

- int ***precision*** (int *p*);

  Fixa a precisão de um campo (por defeito é 6) e devolve a precisão anterior.

- char ***fill*** (char *ch*);

  Define o caracter de preenchimento quando necessário, devolvendo o anterior.

- Exemplo:

```
#include <iostream.h>
void main() {
    cout.width(10);   cout.precision(5);   cout.fill('*');
    cout << "POO" << "\n";
    cout.width(10);  // Necessário repetir
    cout << 123.4567 << "\n";
    cout.setf(ios::left);
    cout.width(8);  // Necessário repetir
    cout << "POO" << "\n";
    cout.width(8);  // Necessário repetir
    cout << 123.4567 << "\n";
}
```

- Produz a seguinte saída:

```
*******POO
****123.46
POO*****
123.46**
```

Manipuladores de Entrada/Saída

- Os manipuladores de entrada/saída são funções especiais, que podem ser executadas numa instrução de leitura ou escrita.
- ***Manipuladores* de entrada/saída**
  - *dec* - define escrita de números em decimal
  - *endl* - escreve '\n' e actualiza informação em disco
  - *ends* - escreve *null*(0)
  - *flush* - actualiza informação em disco
  - *hex* - define escrita em hexadecimal
  - *oct* - define escrita em octal
  - *resetiosflags*(long *f*) - desactiva as *flags* em *f*
  - *setbase*(int *b*) - estipula a base numérica *b*
  - *setfill*(int *ch*) - define o caracter de preenchimento
  - *setiosflags*(long *f*) - activa as *flags* em *f*
  - *setprecision*(int *p*) - define a precisão numérica
  - *setw*(int *w*) - define o tamanho do campo
  - *ws* - despreza espaços brancos

```
#include <iostream.h>
#include <iomanip.h>
void main()
{
    cout << setw(10) << setprecision(5) << setfill('*') << "POO" << endl;
    cout << setw(10) << 123.4567 << endl;
}
```


Leitura e Escrita de Objectos

- O *overloading* dos operadores << e >> tem a forma geral:

```
ostream &operator<<(ostream &os, const classe &obj) {
    // implementação
    return os;
}
istream &operator>>(istream &is, classe &obj) {
    // implementação
    return is;
}
```

```
#include <iostream.h>

class TPonto {
    int   X, Y;
public:
    TPonto(int x=0, int y=0) : X(x), Y(y) {}
    friend ostream &operator<<(ostream &os, const TPonto &p);//os<<p;
    friend istream &operator>>(istream &is, TPonto &p);    // is >> p;
};

ostream &operator<<(ostream &os, const TPonto &p) {
    os << "Ponto=(" << p.X << ", " << p.Y << ") ";
    return os;
}

istream &operator>>(istream &is, TPonto &p) {
    is >> p.X >> p.Y;
    return is;
}

void main() {
    TPonto P1, P2;
    cout << "\nIntroduza 2 pontos:\n";
    cin >> P1 >> P2;
    cout << P1 << P2;
}
```

Implementação de Manipuladores de Entrada/Saída

- Os manipuladores de entrada/saída têm a forma geral:

```
ostream &nomemanipulador(ostream &os) {
    // implementação
    return os;
}

istream & nomemanipulador(istream &is) {
    // implementação
    return is;
}
```

```
#include "ponto.h"

ostream &setup(ostream &os) {
    os.width(5);
    os.precision(3);
    os.fill('-');
    return os;
}

ostream &atencao(ostream &os) {
    os << " Atenção ";
    return os;
}

istream &introducaoponto(istream& is) {
    cout << "\nIntroduza ponto: ";
    return is;
}

void main() {
    TPonto P;
    cin >> introducaoponto >> P;
    cout << setup << atencao << P;
}
```


Ficheiros de Entrada/Saída

- As classes ***ifstream***, ***ofstream*** e ***fstream*** definem respectivamente ficheiros de entrada, saída e entrada/saída formatada, e estão definidas no ficheiro *fstream.h*
- void ***open*** (const char **fic*, int *modo*, int *acesso*);

  Abre um ficheiro, onde o *modo* é pelo menos um dos valores:
  - *ios::app* - abre um ficheiro para acréscimo de informação
  - *ios::ate* - abre um ficheiro, colocando-se no seu fim
  - *ios:: binary* - abre um ficheiro em modo binário
  - *ios:: in* - abre um ficheiro com potencialidades de leitura
  - *ios:: nocreate* - abre um ficheiro, apenas caso já exista
  - *ios:: noreplace* - abre um ficheiro, apenas caso não exista
  - *ios:: out* - abre um ficheiro com potencialidades de escrita
  - *ios:: trunc* - abre um ficheiro, destruindo a informação anterior

  e onde o *acesso* é pelo menos de um dos tipos:
  - *0* - ficheiro com acesso normal
  - *1* - ficheiro com acesso apenas de leitura
  - *2* - ficheiro escondido
  - *4* - ficheiro de sistema
  - *8* - arquivo de conjunto de bits
- void ***close*** ();

  Fecha o ficheiro, actualizando o seu conteúdo em disco.
- int ***eof*** ();

  Define se o ponteiro para o ficheiro chegou ao fim deste.

Ficheiros de Entrada/Saída

```
#include <iostream.h>
#include <fstream.h>

// Este programa efectua a cópia do ficheiro entradas
// para o ficheiro saidas
int main()
{
    ifstream FEnt("entradas");
    if (!FEnt)
    {
        cout << "Erro ao abrir ficheiro entradas";
        return 1;
    }
    ofstream FSai;
    FSai.open("saidas", ios::out | ios::noreplace, 0 | 2);
    if (!FSai)
    {
        cout << "Erro ao abrir ficheiro saidas";
        return 1;
    }
    char C;
    while(!FEnt.eof())
    {
        FEnt >> C;
        FSai << C;
    }
    FEnt.close();
    FSai.close();
}
```


Entrada/Saída Binária

- As funções binárias de manipulação de ficheiros podem ser executadas em qualquer tipo de ficheiro, não sendo necessário abrir o ficheiro como binário (***ios::binary***).
- istream &***get*** (char &*ch*);

  Lê um caracter a partir do ficheiro associado e coloca o valor em *ch*. Devolve uma referência para o ficheiro, ou *null* em caso de erro.
- ostream &***put*** (char *ch*);

  Escreve o caracter *ch* no ficheiro associado e devolve o ficheiro.
- istream &***read*** (unsigned char **buf*, int *num*);

  Lê *num bytes* a partir do ficheiro associado, colocando-os no *buffer* apontado por *buf*. Devolve o ficheiro, ou *null* em caso de erro.
- ostream &***write*** (unsigned char **buf*, int *num*);

  Escreve *num bytes* no ficheiro associado a partir do *buffer* apontado por *buf* e devolve o ficheiro.
- long ***gcount*** ();

  Devolve o nº de caracteres lidos pela última operação binária de entrada.

Entrada/Saída Binária

- istream &***get*** (unsigned char **buf*, int *num*, char *delim* = '\n');

  Lê caracteres a partir do ficheiro associado até ter lido *num* caracteres ou ter lido o caracter *delim*, colocando-os no *buffer* apontado por *buf*. Devolve o ficheiro, ou *null* em caso de erro.

- int ***get*** ();

  Lê um caracter a partir do ficheiro associado, devolvendo esse caracter ou *EOF* caso o fim do ficheiro seja encontrado.

- istream &***getline*** (unsigned char **buf*, int *num*, char *delim*='\n');

  Identica à versão de *get*(*buf*, *num*, *delim*).

- int ***peek*** ();

  Lê um caracter a partir do ficheiro associado sem o retirar do ficheiro associado, devolvendo esse caracter ou *EOF* caso o fim do ficheiro seja encontrado.

- istream &***putback*** (char **ch*);

  Devolve o caracter *ch* ao ficheiro associado, onde *ch* é o último caracter lido.

- ostream &***flush***();

  Actualiza a informação do ficheiro em disco, delvovendo uma referência para o ficheiro.

Entrada/Saída Binária

```
#include "ponto.h"
#include <iostream.h>
#include <fstream.h>

void main()
{
    TPonto VP[5];
    cout << "\nIntroduza 5 pontos:\n";
    for (int i = 0; i < 5; i++)
        cin >> VP[i];
    ofstream fpo("pontos");
    int N = 5;
    fpo.write((char *) &N, sizeof(int));
    fpo.write((char *) VP, N * sizeof(TPonto));
    fpo.close();
    ifstream fpi("pontos");
    fpi.read((char *) &N, sizeof(int));
    fpi.read((char *) VP, N * sizeof(TPonto));
    fpi.close();
    for (i = 0; i < 5; i++)
        cout << VP[i] << "\n";
}
```

- Note-se que é necessário efectuar a conversão *(char *)* na chamada das funções *read()* e *write()* quando se pretende ler ou gravar tipos de informação, que não vectores de caracteres. O *C++* não efectua nestes casos a conversão automática a menos que esteja definido o conversor de tipos (exemplo: na classe TPonto, char *() { return ((char *) this);}).

Acesso Aleatório

- istream &***seekg*** (streamoff *offset*, seek_dir *origin*);

  Move o ponteiro de leitura (*get pointer*) do ficheiro associado, *offset* número de bytes a partir da origem (*origin*) especificada.

- ostream &***seekp*** (streamoff *offset*, seek_dir *origin*);

  Move o ponteiro de escrita (*put pointer*) do ficheiro associado, *offset* número de bytes a partir da origem (*origin*) especificada.

- **streamoff** é definido no ficheiro ***iostream.h***, sendo um tipo enumerado com os valores seguintes:

  - *ios::beg* - move o ponteiro de leitura ou escrita a partir início do ficheiro.
  - *ios::cur* - move o ponteiro de leitura ou escrita a partir da posição corrente do ponteiro do ficheiro.
  - *ios::end* - move o ponteiro de leitura ou escrita a partir do fim do ficheiro.

- streampos ***tellg*** ();

  Devolve a posição do ponteiro de leitura do ficheiro associado.

- streampos ***tellp*** ();

  Devolve a posição do ponteiro de escrita do ficheiro associado.

Acesso Aleatório

```
#include "ponto.h"
#include <fstream.h>
#include <stdlib.h>

class TVectorFicheiro {
     int NumElem;       fstream *Fic;
public:
     TVectorFicheiro(int numElem, char *nomeFic);
     ~TVectorFicheiro() { Fic->close(); delete Fic; }
     void Colocar(int i, const TPonto &p);
     TPonto Ponto(int i);
};

TVectorFicheiro::TVectorFicheiro(int numElem, char *nomeFic) {
     NumElem = numElem;
     if (! (Fic = new fstream(nomeFic, ios::in | ios::out))    exit (1);
}

void TVectorFicheiro::Colocar(int i, const TPonto &p) {
     if (i < 0  ||  i >= NumElem)   exit(1);
     Fic->seekp(i * sizeof(TPonto), ios::beg);
     Fic->write((char *) &p, sizeof(TPonto));
}

TPonto TVectorFicheiro::Ponto(int i) {
     if (i < 0  ||  i >= NumElem)   exit(1);
     Fic->seekg(i * sizeof(TPonto), ios::beg);
     TPonto P;
     Fic->read((char *) &P, sizeof(TPonto));
     return P;
}

int main() {
     TVectorFicheiro VPF(5, "VecFic");
     VPF.Colocar(3, TPonto(-3, 3));
     cout << "Ponto[3]= " << VPF.Ponto(3);
}
```

Funções Genéricas

- Uma **função genérica** (*template function*) define um conjunto de operações gerais que podem ser aplicadas a vários tipos de dados. Uma **função genérica** é igualmente designada por **função parametrizada**, por necessitar da passagem dos tipos de dados com os quais vai operar.
- A definição de uma **função genérica** tem a forma geral:

```
template<class cp1, class cp2, ..., class cpN>
tipo nomefuncao(args)  {
    // implementação
}
```

```
#include "ponto.h"
#include <iostream.h>

template <class T>
void Trocar(T &x, T &y)  {
    T Temp = x;
    x = y;
    y = Temp;
}

void main()  {
    int I = 1, J = 2;
    TPonto P1(1, 2), P2(3, 4);
    Trocar(I, J);
    Trocar(P1, P2);
    cout << "\nI= " << I << " J= " << J;
    cout << "\nP1= " << P1 << " P2= " << P2;
}
```


Classes Genéricas

- Uma **classe genérica** (*template class*) é uma classe que é definida utilizando outras classes que recebe como parâmetros, pelo que, é igualmente designada por **classe parametrizada**.
- As classes genéricas são bastantes úteis nas estruturas de dados que contêm uma lógica genérica. Por exemplo, listas, pilhas, filas, etc.
- A definição de uma **classe genérica** tem a forma geral:

```
template<class cp1, class cp2, ..., class cpN>
class nomeclasse : ... {
    // implementação
}
```

```
#include <iostream.h>

template <class C, class V>
class TAssociacao  {
    C Cha;
    V Val;
public:
    TAssociacao() {}
    TAssociacao(const C &cha, const V &val);
    const C &Chave() {return Cha; }
    V &Valor() { return Val; }
    int operator==(const TAssociacao<C, V> &a);
    V &operator()() { return Val; }
    V &operator[](const C &cha);
    friend ostream &operator<<(ostream &os,
                               const TAssociacao<C,V> &a);
    friend istream &operator>>(istream &is, TAssociacao<C, V> &a);
};
```

Classe Genérica TAssociacao

```
template <class C, class V>
TAssociacao<C, V>::TAssociacao(const C &cha, const V &val)
{
      Cha = cha;
      Val = val;
}

template <class C, class V>
int TAssociacao<C, V>::operator==(const TAssociacao<C, V> &a)
{     // As classes C e V necessitam de definir o operador==
      return (Cha == a.Cha  &&  Val == a.Val);
}

template <class C, class V>
ostream &operator<<(ostream &os, const TAssociacao<C, V> &a)
{     // As classes C e V necessitam de definir o operador<<
      os << "\nAssociação(" << a.Cha << ", " << a.Val << ")";
      return os;
}

template <class C, class V>
istream &operator>>(istream &is, TAssociacao<C, V> &a)
{     // As classes C e V necessitam de definir o operador>>
      is >> a.Cha >> a.Val;
      return is;
}

template <class C, class V>
V &TAssociacao<C, V>::operator[](const C &cha)
{  // Altera chave e valor
      Cha = cha;
      return Val;
}
```


Classe Genérica TAssociacao

```
#include "associac.h"
#include "ponto.h"

void main()
{
      TAssociacao<int, char> AIC1(3,'A'), AIC2(3, 'K');
      cout << "Iguais= " << (AIC1 == AIC2);
      AIC1() = 'C';
      cout << AIC1 << AIC2;
      TAssociacao<char, TPonto> ACP('a', TPonto(1, 2));
      ACP['K'] = TPonto(10, 20);
      cout << ACP;
      TAssociacao<TPonto, TPonto> ACPL;
      cout << "\nIntroduza associacao de pontos:\n";
      cin >> ACPL;
      cout << ACPL;
      ACPL[ACPL.Chave()] = TPonto(ACPL().Y, ACPL().X);
      cout << ACPL;
      ACP == AIC1;   // Erro, pois ACP e AIC1 são de tipos diferentes
      TAssociacao<TPonto, int> A("abc", 3);
                              // Erro, pois "abc" não é do tipo TPonto
}
```

Criação de uma Lista Genérica Ordenada

```
template <class T>
class TNoListaDupla
{
public:
     T   Elem;
     TNoListaDupla<T>   *Ant, *Seg;
     TNoListaDupla(const T &elem)
          : Elem(elem) { Seg = Ant = this; }
     TNoListaDupla(const T &elem, TNoListaDupla<T> *seg)
          : Elem(elem) { Seg = seg; Ant = seg->Ant;
                         Seg->Ant = Ant->Seg = this; }
};

template <class T>
class TIteradorListaDupla;   // Declaração forward

template <class T>
class TListaDupla
{
     TNoListaDupla<T> *Inicio;
     TNoListaDupla<T> *Pesquisar(const T &elem);
                                   // Devolve ponteiro para seguinte
public:
     typedef void (*FuncOper)(T &, void *);
     typedef int  (*FuncCond)(const T &, void *);
     friend TIteradorListaDupla<T>; // Para poder aceder ao Inicio da lista
     TListaDupla(const T &lim) { Inicio =new TNoListaDupla<T>(lim); }
     void Inserir(const T &elem);
     int Remover(const T &elem);
     void EfectuarOperacao( FuncOper oper, void *args);
     T &PrimeiroQue(FuncCond cond, void *args );
};
```


Criação de uma Lista Genérica Ordenada

```
template <class T>
TNoListaDupla<T> *TListaDupla<T>::Pesquisar(const T &elem)
{
     TNoListaDupla<T>   *S = Inicio->Seg;
     while (S->Elem < elem)
          S = S->Seg;
     return S;
}

template <class T>
void TListaDupla<T>::Inserir(const T &elem)
{
     TNoListaDupla<T>   *S = Pesquisar(elem);
     new TNoListaDupla<T>(elem, S);
}

template <class T>
int TListaDupla<T>::Remover(const T &elem)
{
     TNoListaDupla<T>   *S = Pesquisar(elem);
     if (S->Elem == elem)
     {
          S->Ant->Seg = S->Seg;
          S->Seg->Ant = S->Ant;
          delete S;
          return 1;
     }
     return 0;
}
```

Criação de uma Lista Genérica Ordenada

```
template <class T>
void TListaDupla<T>::EfectuarOperacao
                              (TListaDupla<T>::FuncOper oper, void *args)
{
     TNoListaDupla<T>   *S = Inicio->Seg;
     while (S != Inicio)
     {
          oper(S->Elem, args);
          S = S->Seg;
     }
}


template <class T>
T &TListaDupla<T>::PrimeiroQue
                              (TListaDupla<T>::FuncCond cond, void *args)
{
     TNoListaDupla<T>   *S = Inicio->Seg;
     while (S != Inicio)
     {
          if (cond(S->Elem, args))
               return S->Elem;
          S = S->Seg;
     }
     return Inicio->Elem;
}
```


Criação de uma Lista Genérica Ordenada

```
template <class T>
class TIteradorListaDupla
{
     TListaDupla<T> *Lista;
     TNoListaDupla<T> *Cor;
     int  Tipo;
public:
     enum { Inicio, Fim };
     TIteradorListaDupla(TListaDupla<T> &lista, int tipo = Inicio)
     { Lista = &lista; Tipo = tipo;
       Cor = (Tipo == Inicio) ? Lista->Inicio->Seg : Lista->Inicio->Ant; }
     operator int () { return Cor != Lista->Inicio; }
     T &Corrente()
     {
          if (! int(*this))
               exit(1);
          return Cor->Elem;
     }
     T &operator++()
     {
          if (Cor->Seg == Lista->Inicio)
               exit(1);
          Cor = Cor->Seg;
          return Cor->Elem;
     }
```

Criação de uma Lista Genérica Ordenada

```
    T &operator++(int)
    {
        if (Cor == Lista->Inicio)
            exit(1);
        TNoListaDupla<T> *Aux = Cor;
        Cor = Cor->Seg;
        return Aux->Elem;
    }
    T &operator--()
    {
        if (Cor->Ant == Lista->Inicio)
            exit(1);
        Cor = Cor->Ant;
        return Cor->Elem;
    }
    T &operator--(int)
    {
        if (Cor == Lista->Inicio)
            exit(1);
        TNoListaDupla<T> *Aux = Cor;
        Cor = Cor->Ant;
        return Aux->Elem;
    }
    void Reiniciar()
    {
        if (Tipo == Inicio)
            Cor = Lista->Inicio->Seg;
        else
            Cor = Lista->Inicio->Ant;
    }
};
```


Criação de uma Lista Genérica Ordenada

```
#include <iostream.h>
#include "listadpl.h"

void Escrever(int &i, void *)
{
    cout << " " << i;
}

void main()
{
    TListaDupla<int> LDI(100);
    LDI.Inserir(1);
    LDI.Inserir(5);
    LDI.Inserir(2);
    LDI.Inserir(4);
    cout << "\n";
    LDI.EfectuarOperacao(Escrever, 0);
    LDI.Remover(4);
    cout << "\n";
    LDI.EfectuarOperacao(Escrever, 0);
    TIteradorListaDupla<int> ILDI(LDI);
    cout << "\n";
    while (ILDI)
        cout << " " << ILDI++;
    TIteradorListaDupla<int>FLDI(LDI,TIteradorListaDupla<int>::Fim);
    cout << "\n";
    while (FLDI)
        cout << " " << FLDI--;
}
```

Definição e Encaminhamento de Excepções

- A definição e encaminamento de uma **excepção** tem a forma geral:

```
try
{   // Bloco de definição de excepções
    ... throw   excepção1
    ... throw   excepção2
    ... throw   excepçãoK
}
catch (tipo1 arg)
{   // Encaminhamento da excepção de tipo1
}
catch (tipo2 arg)
{   // Encaminhamento da excepção de tipo2
}
...
catch (tipoN arg)
{   // Encaminhamento da excepção de tipoN
}
```

- ***try*** (sondar) - define um bloco onde se efectua a sondagem de excepções definidas por ***throw***.
- ***throw*** (atirar) - define uma excepção a ser encaminhada para o respectivo ***catch***.
- ***catch*** (apanhar) - define a resposta a uma dada excepção encaminhada por um ***throw***.

Exemplo Inicial

```
#include <iostream.h>

void main()
{
    cout << "\nExemplo de excepção";
    int K;
    try
    {
        cout << "\nBloco try";
        cout << "\nIntroduza um inteiro: ";
        cin >> K;
        throw K;
        cout << "\nMensagem não executada";
    }
    catch(int i)
    {
        cout << "\nCatch inteiro: " << i;
    }
    cout << "\nFim";
}
```

- Resultado:

```
Exemplo de excepção
Bloco try
Introduza um inteiro: 3
Catch inteiro: 3
Fim
```

Definição de uma Função de Excepções

- O bloco **catch** pode responder a qualquer tipo de excepções:

```
catch (...)
{    // Encaminhamento de qualquer tipo de excepção
}
```

```
#include <iostream.h>

void Excepcoes(int i) {
    try {
        if (i < 0)
            throw i;
        if (i == 0)
            throw "Valor Nulo";
        throw 15.24;
    }
    catch(char *s) {
        cout << "\nCatch string: " << s;
    }
    catch(int i) {
        cout << "\nCatch inteiro: " << i;
    }
    catch(...) {
        cout << "\nCatch por defeito";
    }
}

void main() {
    cout << "\nExcepções";
    Excepcoes(-3);
    Excepcoes(0);
    Excepcoes(8);
    cout << "\nFim";
}
```


Definição de uma Precondição

- Resultado:

```
Excepções
Catch inteiro: -3
Catch string: Valor Nulo
Catch por defeito
Fim
```

```
#include <iostream.h>
#include <stdlib.h>

#define PRECONDICAOX(cond, mens) \
    try { if (! cond) throw cond; } \
    catch (...) { cout << mens; exit(1); }

#define PRECONDICAO(cond) PRECONDICAOX(cond, #cond)

float Divisao(int i, int j)
{
    PRECONDICAOX(j, "Divisão por zero");
    return (i / (float) j);
}

void main()
{
    int A, B;
    cout << "\nIntroduza dois inteiros: ";
    cin >> A >> B;
    cout << A << " / " << B << " = " << Divisao(A, B);
}
```

Redefinição da Classe TIteradorListaDupla com Excepções

```
template <class T>
class TIteradorListaDupla
{
      TListaDupla<T> *Lista;
      TNoListaDupla<T> *Cor;
      int  Tipo;
public:
      enum { Inicio, Fim };
      TIteradorListaDupla(TListaDupla<T> &lista, int tipo = Inicio)
      { Lista = &lista; Tipo = tipo;
        Cor = (Tipo == Inicio) ? Lista->Inicio->Seg : Lista->Inicio->Ant; }
      operator int () { return Cor != Lista->Inicio; }
      T &Corrente()
      {
            PRECONDITION( int(*this));
            return Cor->Elem;
      }
      T &operator++()
      {
            PRECONDITION(Cor->Seg != Lista->Inicio);
            Cor = Cor->Seg;
            return Cor->Elem;
      }
      T &operator++(int)
      {
            PRECONDITION(Cor != Lista->Inicio);
            TNoListaDupla<T> *Aux = Cor;
            Cor = Cor->Seg;
            return Aux->Elem;
      }
```


Redefinição da Classe TIteradorListaDupla com Excepções

```
      T &operator--()
      {
            PRECONDITION(Cor->Ant != Lista->Inicio);
            Cor = Cor->Ant;
            return Cor->Elem;
      }
      T &operator--(int)
      {
            PRECONDITION(Cor != Lista->Inicio);
            TNoListaDupla<T> *Aux = Cor;
            Cor = Cor->Ant;
            return Aux->Elem;
      }
      void Reiniciar()
      {
            if (Tipo == Inicio)
                  Cor = Lista->Inicio->Seg;
            else
                  Cor = Lista->Inicio->Ant;
      }
};
```

- Em *Borland C++* no ficheiro ***checks.h*** estão definidos os macros:

  PRECONDITION(cond) - se cond = 0 define a excepção e escreve a mensagem #cond

  PRECONDITIONX(cond,mens) - se cond = 0 define a excepção e escreve a mensagem mens

Membros Estáticos

```
#include <iostream.h>

class A
{
      char Cod;
      static int NumInst;   // Variável estática (ou de classe)
public:
      A(char cod) { Cod = cod; NumInst++;}
      ~A() { NumInst--; }
      // Método estático (ou de classe)
      static void Iniciar(int numInst) { NumInst = numInst; }
      static int NumeroInstancias() { return NumInst; }
      void Mostrar() { cout << "\nCódigo= " << Cod; }
};

// Definição da variável NumInst da classe A
int A::NumInst;

class B : public A
{
      int K;
public:
      B(char cod, int k) : A(cod) { K = k; }
      void Mostrar() { A::Mostrar(); cout << " K= " << K; }
};
```


Membros Estáticos

```
void main()
{
      char C;
      int V;
      cout << "\nIntroduza código: ";
      cin >> C;
      cout << "\nIntroduza valor: ";
      cin >> V;
      A::Iniciar(0);    // Invocação de método da classe A
      A VA(C);
      VA.Mostrar();
      B VB(C, V);
      VB.Mostrar();
      cout << "\nEstão alocadas ";
      cout << A::NumeroInstancias() << " instâncias de A ou B";
      A *PA = new A('K');
      B *PB = new B('C', 3);
      cout << "\nEstão alocadas ";
      cout << A::NumeroInstancias() << " instâncias de A ou B";
                  // Invocação de método de classe A
      delete PB;
      cout << "\nEstão alocadas ";
      cout << A::NumeroInstancias() << " instâncias de A ou B";
      delete PA;
      cout << "\nEstão alocadas ";
      cout << A::NumeroInstancias() << " instâncias de A ou B";
}
```

Membros Estáticos

```
#include <iostream.h>

class A
{
      char Cod;
      static int NumInst;   // Variável estática (ou de classe)
public:
      A(char cod) { Cod = cod; NumInst++;}
      ~A() { NumInst--; }
      // Método estático (ou de classe)
      static void Iniciar(int numInst) { NumInst = numInst; }
      static int NumeroInstancias() { return NumInst; }
      void Mostrar() { cout << "\nCódigo= " << Cod; }
};

// Definição da variável NumInst da classe A
int A::NumInst;

class B : public A
{
      int K;
public:
      B(char cod, int k) : A(cod) { K = k; }
      void Mostrar() { A::Mostrar(); cout << " K= " << K; }
};
```


Membros Estáticos

```
void main()
{
      char C;
      int V;
      cout << "\nIntroduza código: ";
      cin >> C;
      cout << "\nIntroduza valor: ";
      cin >> V;
      A::Iniciar(0);    // Invocação de método da classe A
      A VA(C);
      VA.Mostrar();
      B VB(C, V);
      VB.Mostrar();
      cout << "\nEstão alocadas ";
      cout << A::NumeroInstancias() << " instâncias de A ou B";
      A *PA = new A('K');
      B *PB = new B('C', 3);
      cout << "\nEstão alocadas ";
      cout << A::NumeroInstancias() << " instâncias de A ou B";
                  // Invocação de método de classe A
      delete PB;
      cout << "\nEstão alocadas ";
      cout << A::NumeroInstancias() << " instâncias de A ou B";
      delete PA;
      cout << "\nEstão alocadas ";
      cout << A::NumeroInstancias() << " instâncias de A ou B";
}
```