Notas de aula para o curso de C++

© 2002 - Roberto J. Rodrigues
Núcleo de Computação Eletrônica
Universidade Federal do Rio de Janeiro

Bibliografia

- C/C++ e OO em ambiente Multiplataforma – S. Villas-Boas (www.del.ufrj.br/~villas)
- Como programar em C++ - Deitel...
- Projeto de computadores digitais - G. G. Langdon Jr. E. Fregni, Editora Edgard Blucher LTDA.
- Linguagem C, programação e aplicações – Módulo consultoria e informática – Livros técnicos e científicos editora. LTDA.

ÍNDICE

Regras de programação

1. Espaço entre blocos de funcionalidade

Ex:

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

linha em branco

void main()
{
        int c;
        char i;

        linha em branco

        ...
 }
```


2. Identação

Ex:

```
void main()
{
        → int c;

        if (c == 0)
        {
                → c = -1;
        }
}
```

3. Identificadores dentro do contexto

Ex:

```
int   a, b, c, dedinho, minhavo     // ERRO!!!!
int   lucro_liquido, contador       // CERTO
```

4. Comentários relevantes

Ex:

```
int   contador;   // contador → IRRELEVANTE!
float calculo_do_lucro()
//    toma como base os arquivos xxx.dat e yyy.dat
//    a porcentagem inicial foi definida em contrato
```

5. Letras minúsculas ≠ LETRAS MAIÚSCULAS

Ex:

```
Lucro ≠ lucro
abre_arquivo()
abreArquivo()
AbreArquivo()
// as funções acima são consideradas diferentes pelo compilador
```


Comentários

Comentários em C++ podem ser criados com a forma normal (C) ou comentários por linha usando //:

Exemplo:

```
/*
      Comentários
      em C
*/

      // comentário em C++
      int   j;    // contador j
```


Palavras reservadas

As palavras reservadas escritas em negrito no quadro abaixo referem-se às palavras reservadas apenas de C++

Palavras reservadas em C++					
asm	continue	float	**new**	signed	**try**
auto	default	for	**operator**	sizeof	typedef
break	**delete**	**friend**	**private**	static	union
case	do	goto	**protected**	struct	unsigned
catch	double	if	**public**	switch	**virtual**
char	else	**inline**	register	**template**	void
class	enum	int	return	**this**	volatile
const	extern	long	short	**throw**	while

Símbolos

Símbolos para pontuação										
!	%	^	&	*	(	)	-	+	{	}
|	~	[	]	;	‘	“	<	>	?	,
.	/	=								

Símbolos para operadores										
->	++	--	.*	->*	<<	>>	<=	>=	==	!=
&&	||	*=	%=	+=	-=	<<=	>>=	&=	^=	/=
|=	::									

Programa básico

Um programa básico em C++ não requer o uso das bibliotecas padrão convencionais do C. Para a entrada/saída padrão existem objetos e métodos criados para as funções de E/S definidos em bibliotecas referentes. Por exemplo:

ex1:

```
#include <iostream.h>

void main(void)
{
      cout << "Alô mundo!\n";

      for ( int i=0; i<10; i++)
            cout << "i = " << i << endl;
}
```

ex2

```
#include <iostream.h>

void main(void)
{
      int   n;

      cout << "Escreva um número de 1 a 10\n";
      cin >> n;

      for ( int i=0; i<n; i++)
            cout << "i = " << i << endl;
}
```


Chamada de função por referência e por valor

Quando um parâmetro é passado por valor, a variável na função é uma cópia da variável que foi passada, uma vez que a função sempre aloca memória na *stack* para os parâmetros. Qualquer alteração desta variável na *stack* valerá apenas localmente.

Quando o parâmetro é passado por referência, a variável é criada na *stack,* porém esta receberá o endereço da variável que foi passada na função. Desta forma qualquer alteração nesta variável poderá causar alteração na variável que foi passada (referência).

O C++ sobrecarrega o símbolo **&** para tratar referências de modo mais simples. Na verdade, a "*indireção*" será implementada da mesma forma que a construção com ponteiros, porém será "interpretada" como um endereço existente e "*substituído*" pelo valor de referencia (endereço)

Por exemplo:

```
int   a, *p, &c = a;
// a declaração de referência obriga a declaração do referenciado

      a = 0;
      p = &a;
      *p = 10;          // muda o valor de a indiretamente
      c = 34;           // muda o valor de a indiretamente

void f1 ( float z )
{
      z = 1.1;
}
```

```
void f2 ( float *x )
{
      *x = 2.2;
}

void f3 ( float &v )
{
      v = 3.3,
}

void main(void)
{
      float a = 10.1;

      cout << "a=" << a << endl;
      f1 ( a );

      cout << "a=" << a << endl;
      f2 ( &a );

      cout << "a=" << a << endl;
      f3 ( a );
}
```


O resultado deverá ser:

```
a=10.1
a=10.1
a=2.2
a=3.3
```

Função inline

Funções são compiladas em código "a parte" do programa principal. A chamada de função desvia o fluxo para endereço de entrada do código da função que é executada retornando ao programa principal. Uma função com o qualificador *inline* não é compilada "*em separado*" mas "aconselha" o compilador a compilar uma cópia do código da função no próprio local, *em linha* com o código da linha de chamada.

Função:

```
int quad (int x)

{
      return x*x;
}

void main()
{
      int x;
      x = quad ( x );
}
```

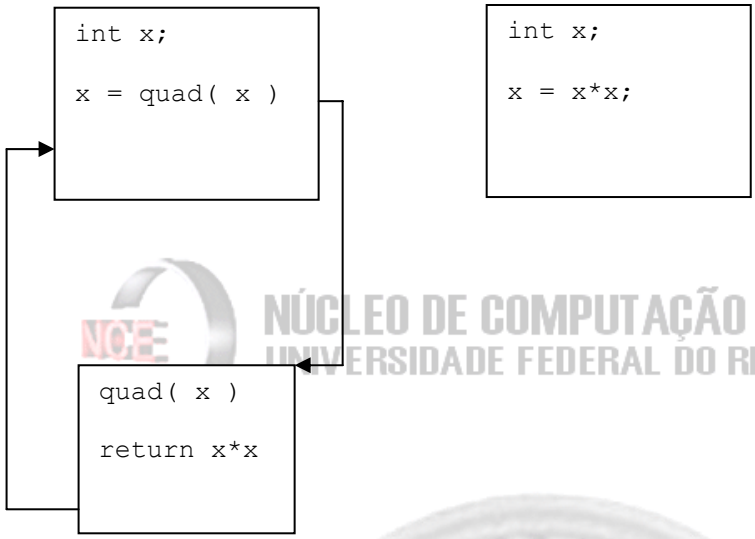
Função *inline*:

```
inline int quad (int x) { return x*x; }


void main()
{
      int x;
      x = quad ( x );
}
```

Compilação:

Função: **Função *inline*:**



Argumentos default

Argumentos *default* (ou implícitos) devem ser os argumentos mais à direita (os argumentos finais) na lista de parâmetros da função. Quando se chama uma função com dois ou mais argumentos default, se um argumento omitido não é o argumento mais à direita na lista de argumentos, todos os argumentos à direita desse argumento também devem ser omitidos. Argumentos *default* deve ser especificados com a primeira ocorrência do nome da função.

Ex:

```
int volume (int comprimento, int altura=1, int largura=1)
{
      return comprimento * altura * largura;
}
...
      int v = volume(10,5); // 10 * 5 * (1)

      cout << v << endl;
```

O resultado será: 50

Operador unário de resolução de escopo

É possível declarar variáveis locais e globais com o mesmo nome dentro de um bloco. Para acessar uma variável global quando uma variável local com o mesmo nome está no escopo usa-se o operador :: (*resolução de escopo)*.
Ex:

```
int ref = 1000;

void main(void)
{
      int j = 0;

      if (j == 0)
      {
            int ref = 2;

            cout << ref * ::ref << endl;
      }
}

O resultado será: 2000
```


Sobrecarga de funções

Sobrecarga de funções é um recurso implementado em C++ que possibilita definições de várias funções com o mesmo nome, desde que tenham parâmetros diferentes. Quando uma função é chamada o código seleciona a função apropriada examinando os argumentos.
Ex:

```
int   quadrado ( int x )
{
      return x * x;
}
double quadrado ( double y )
{
      return y * y;
}

void main ( void )
{
      int         r1;
      double      f1;

      r1 = quadrado ( 7 );
      f1 = quadrado ( 2.2 );

      cout  << "r1=" << r1
            << " f1="<< f1
            << endl;
}
```

Resultado:
r1=49, f1=4.84

Gabaritos de funções

Se a lógica dos programas e suas operações são idênticas para os vários tipos de dados, isto pode ser codificado mais convenientemente usando-se os *gabaritos* (*templates*) de função. Dependendo dos tipos dos parâmetros nas chamadas da função, o C++ gera automaticamente *funções gabarito* para tratar cada tipo de chamada adequadamente.

Ex:

```
template <typename T>
T maximo ( T valor1, T valor2, T valor3 )
{
      T max = valor1;

      if (valor2 > max)
            max = valor2;

      if (valor3 > max)
            max = valor3;

      return max;
}
```


Este gabarito declara um parâmetro formal **T**, como tipo de dados que serão testados pela função. O Compilador substitui T ao longo de toda definição do gabarito na compilação e é gerado uma função completa.

```
int maximo ( int valor1, int valor2, int valor3 )
{
      int max = valor1;

      if ( valor2 > max )
            max = valor2;

      if ( valor3 > max )
            max = valor3;

      return max;
}
```

Exemplo de um programa:

```
template <typename T>
T maximo ( T valor1, T valor2, T valor3 )
{
      T max = valor1;

      if (valor2 > max)
            max = valor2;

      if (valor3 > max)
            max = valor3;

      return max;
}

void main(void)
{
      int i1,i2,i3;

      cout << "Escreva 3 valores inteiros: ";
      cin >> i1 >> i2 >> i3;
      cout << "O maior valor e': " << maximo ( i1, i2, i3 ) << endl;

      double d1,d2,d3;

      cout << "Escreva 3 valores reais: ";
      cin >> d1 >> d2 >> d3;
      cout << "O maior valor e': " << maximo ( d1, d2, d3 ) << endl;
}
```

Técnicas de programação

Deve-se planejar programação pensando na aplicação específica ou genérica. Uma boa programação implica no desenvolvimento de uma boa **biblioteca**. Para o desenvolvimento dos diversos **componentes** de um programa, é comum dividir um determinado problema em componentes específicos e/ou genéricos. Por exemplo: um aplicativo deve traçar gráficos a partir da leitura de dados de um sensor. Os detalhes dos gráficos representam um problema específico, mas o traçado do gráfico é um problema genérico.

Para uma boa reutilização de código citamos:

- **Componente** – é um bloco de código que pode ser usado de várias formas em programas. Por exemplo: Listas, vetores, etc. Devem ser escritos de forma genérica.
- **Biblioteca** – Conjunto de componentes
- ***Framework* (Estrutura)** – Um esqueleto de um programa que é usado para gerar uma determinada aplicação.
- **Aplicação** – Um programa completo.

A programação estruturada representou um aumento na produtividade da programação, mas o desenvolvimento das tecnologias de software produziu conceitos mais evoluídos que a programação estruturada. O uso desses novos conceitos ficou conhecido como "*programação orientada a objeto*".

Programação Orientada a Objetos

A metodologia de orientação objetos representa um novo modelo de abstração de dados que implica em vantagens e desvantagens:

- **Vantagens:**
  - Programas maiores e mais complexos
  - Reaproveitamento dos componentes.
- **Desvantagens:**
  - Tamanho do código aumenta
  - A velocidade de execução fica menor

A orientação a objetos é um modo natural de pensar sobre o mundo e escrever programas.

A programação orientada a objetos modela objetos do mundo real com duplicatas em software. Ela se aproveita das relações de *classe*, nas quais objetos de uma certa classe – tal como uma classe de veículos – têm as mesmas características próprias. Um objeto da classe "conversível" certamente tem as mesmas características da classe mais genérica "automóvel", mas a capota pode ser conversível (sobe e desce).

Para criar as melhores soluções, deve-se seguir um processo detalhado para obter uma *análise* dos requerimentos do projeto. Este projeto envolve a análise e projeto do sistema de um ponto de vista orientado a

objetos, assim, chama-se de *análise e projeto orientados a objetos* (*OOAD, object-oriented analysis and design*).

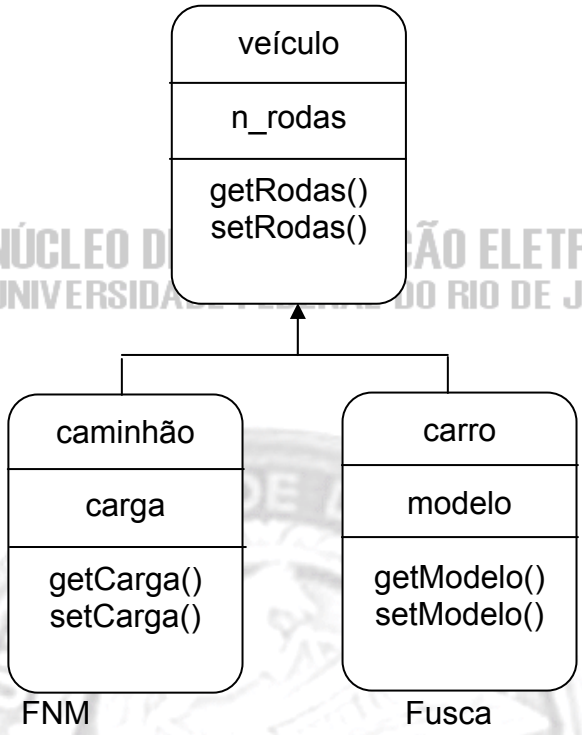
OOAD é um termo genérico para as idéias por trás do processo que se emprega para analisar um problema e desenvolver uma abordagem para resolvê-lo. Existem muitos processos diferentes de OOAD, mas atualmente é mais usada uma linguagem gráfica para a representação dos processos de OOAD. Esta linguagem é conhecida como *Unified Modeling Language (UML)*.

Características da POO

A POO possui diversas características sendo que podemos destacar quatro de maior relevância: Classe, Herança, Polimorfismo e encapsulamento te dados e métodos.

- **Classe**: É um conjunto de características extraídas de um objeto a partir de um mini-mundo específico. As características definidas para representar uma classe são: *atributos* e *métodos*. Os *atributos* definem (em geral) características físicas, enquanto que os *métodos* definem (de um modo geral) o comportamento ou a funcionalidade do objeto. Qualquer classe acima da hierarquia de sua classe é chamada classe ancestral à esta (ou classe *pai*). Uma ***instância*** é a realização em memória de uma classe. As *instâncias* são então, *variáveis objeto*, ou simplesmente ***objetos***.
- **Herança:** A herança é o processo pelo qual um objeto adquire (herda) propriedades de outro. Por exemplo, um *fusca* é um *automóvel* e um *automóvel* é um *veículo*. Um *caminhão* também é um *veículo* e um *FNM* é um *caminhão*. Sem a classificação, cada objeto teria de ser definido por todas as suas propriedades. *Fusca* e *FNM* são instâncias de *automóvel* e *caminhão*, respectivamente.
  A maioria das linguagens OO, implementa herança simples; cada classe tem apenas um *pai*, do qual herda todas as definições deste *pai* ou herda todas as definições por classes mais “altas” na hierarquia. Uma hierarquia de classes então se torna uma espécie de *teia* de classes. Herança múltipla permite que uma classe possa herdar características de várias classes ao mesmo tempo.
- **Polimorfismo**: Significa “muitas formas”. Em programação quer dizer que o mesmo nome pode ser dado à métodos diferentes (implementações diferentes), e que um método pode assumir “diversas formas” dependendo do contexto da utilização do(s) método(s). A implementação básica para a construção de polimorfismo é a sobrecarga de funções (métodos). Por exemplo: Para se desenhar objetos gráficos (quadrados, retângulos, linhas, círculos etc) usa-se apenas um método *desenhar()*, embora existam várias implementações (uma para cada classe)
- **Encapsulamento de dados**: A OOP encapsula dados (atributos) e métodos (comportamento) em pacotes chamados de *objetos*; os dados e funções de um objeto estão intimamente amarrados. Os objetos têm a propriedade de *ocultação* de informações. Isto ‘significa que, embora os objetos possam saber como se comunicar uns com os outros através de *interfaces* bem definidas, normalmente não é permitido aos objetos saber como outros objetos são implementados –

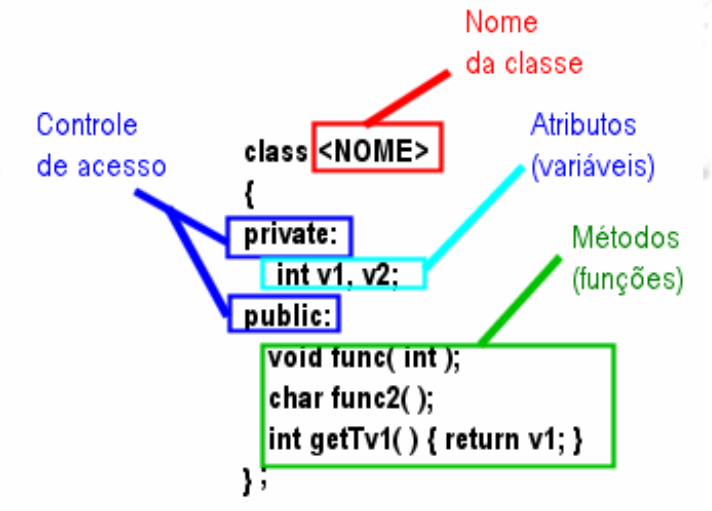
os detalhes de implementação ficam escondidos dentro dos objetos. O encapsulamento define três modos básicos de acesso à infomação: *public* – acesso público; *protected* – acesso limitado e *private* – acesso restrito.



Visão gráfica das entidades (classes)

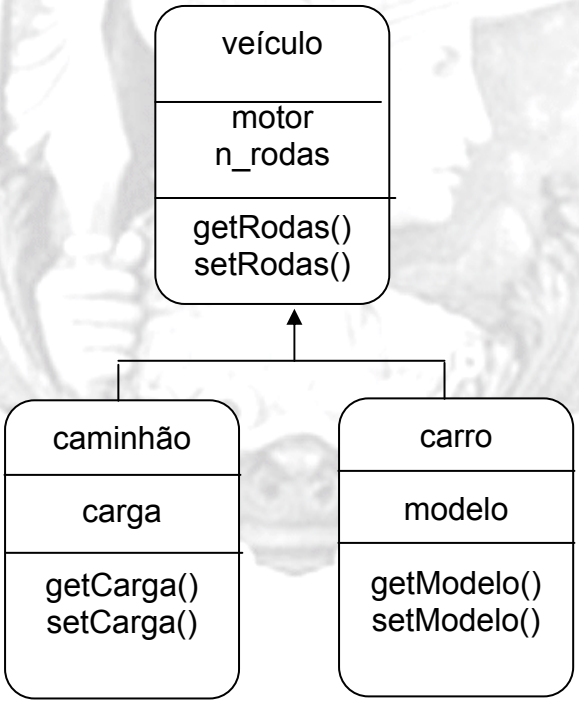
Classes

Classes em C++ são implementadas a partir da construção das *structs*.



Exemplo: O Cliente da empresa Veiculos S. A. Deseja construir um Fusca de 4 rodas do tipo SEDAN, e um FNM de 12 rodas com capacidade para 3000Kg, com base na descrição acima.
Para a implementação do programa, podemos estabelecer 5 passos básicos:

a) A análise dos requerimentos do problema: O cliente deseja construir (instanciar) dois objetos. O programa deverá então instanciar os dois objetos, com os atributos fornecidos e mostrar para o cliente.
b) Classificação: O fusca é um automóvel e um FNM é um caminhão.
    Fusca: automóvel
        Motor, 4 rodas, tipo(SEDAN).
    FNM: Caminhão
        Motor, 12 rodas, carga(3000)
c) Verificação de redundâncias: Esta fase detecta classes derivadas a partir dos membros comuns entre as diversas classes. O objetivo é rescrever as classes reavaliando os relacionamentos.
    Fusca: automóvel
        Motor, 4 rodas, tipo(SEDAN).
    FNM: Caminhão
        Motor, 12 rodas, carga(3000)
    Existem membros comuns: **motor** e **rodas**. Estes atributos pertencem a uma classe ancestral que chamaremos de ***Veículo***.
    Então as classes rescritas ficarão da seguinte forma:
    Veículo:
        Motor, 4 rodas
    Caminhão : Veículo
        carga
    Automóvel : Veículo
        Tipo.
d) Diagramas UML: Escrever os diagramas. A princípio, apenas o diagrama de classes.



e) Implementação:

```
#include <iostream.h>

// Classe basica
```

```
class Veiculo
{
      int   motor,
            n_rodas;
public:
      void setNRodas(int n);
      int getNRodas();
};

void Veiculo::setNRodas(int n)
{
      n_rodas = n;
}

int Veiculo::getNRodas()
{
      return n_rodas;
}

//Primeira classe secundaria

class Caminhao : public Veiculo
{
      int n_carga;
public:
      void setCarga(int c);
      int getCarga();
      void mostra();
};

void Caminhao::setCarga(int c)
{
      n_carga = c;
}

int Caminhao::getCarga()
{
      return n_carga;
}

void Caminhao::mostra()
{
      cout << "Caminhao: n rodas = "  << getNRodas()
            << " carga = " << getCarga() << endl;
}

//Segunda classe secundaria

enum TIPO { SEDAN, CAMPO, LUXO, UTILITARIO };

class Automovel : public Veiculo
{
      TIPO n_tipo;
public:
      void setTipo(TIPO t);
      TIPO getTipo();
      void mostra();
};
```

```
void Automovel::setTipo(TIPO t)
{
      n_tipo = t;
}

TIPO Automovel::getTipo()
{
      return n_tipo;
}

void Automovel::mostra()
{
      cout << "Automovel: n rodas = "  << getNRodas()
            << " tipo = " << getTipo() << endl;
}

void main()
{
      Caminhao FNM;
      Automovel Fusca;

      FNM.setNRodas(12);
      FNM.setCarga(300);
      FNM.mostra();

      Fusca.setNRodas(4);
      Fusca.setTipo(SEDAN);
      Fusca.mostra();
}
```


Separando as classes da implementação

As classes como as estruturas (*struct*) definem novos tipos de dados. Em um projeto mais complexo onde se desenvolve módulos para um programa, é necessário separar as definições (classes) de suas implementações.

A única implementação que permanece dentro da classe é a implementação de métodos do tipo *inline*. Tais modelos não implementam código como já foi explicado.

Desta maneira, podemos refazer o programa do Fusca/FNM dividindo em três arquivos separados. Um arquivo ***veiculo.h*** contendo apenas os tipos, ou seja, as *classes*, *structs*, *enums* e *unions,* um arquivo ***veiculo.cpp*** contendo as implementações da classe e por último um arquivo chamado de ***programa.cpp*** que é o programa final deste exemplo.

Para todos os compiladores:

- Arquivo ***veiculo.h***

```
enum TIPO { SEDAN, CAMPO, LUXO, UTILITARIO };

// Classe basica

class Veiculo
{
   int n_rodas;
public:
```


```
   void setNRodas(int n);
   int getNRodas() { return n_rodas; } // metodo inline
};

//Primeira classe secundaria

class Caminhao : public Veiculo
{
   int n_carga;
public:
   void setCarga(int c);
   int getCarga() { return n_carga; }  // metodo inline
   void mostra();
};

//Segunda classe secundaria

class Automovel : public Veiculo
{
   TIPO n_tipo;
public:
   void setTipo(TIPO t);
   TIPO getTipo() { return n_tipo; }  // metodo inline
   void mostra();
};
```


- Arquivo ***veiculo.cpp***

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

void Automovel::setTipo(TIPO t)
{
   n_tipo = t;
}

void Veiculo::setNRodas(int n)
{
   n_rodas = n;
}

void Caminhao::setCarga(int c)
{
   n_carga = c;
}

void Caminhao::mostra()
{
   cout << "Caminhao: n rodas = "  << getNRodas()
          << " carga = " << getCarga() << endl;
}

void Automovel::mostra()
{
   cout << "Automovel: n rodas = "  << getNRodas()
          << " tipo = " << getTipo() << endl;
}
```

- Arquivo ***programa.cpp***

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

void main()
{
   Caminhao FNM;
   Automovel Fusca;

   FNM.setNRodas(12);
   FNM.setCarga(300);
   FNM.mostra();

   Fusca.setNRodas(4);
   Fusca.setTipo(SEDAN);
   Fusca.mostra();
}
```


Para o VC++:

a) Inicie um novo Workspace com o nome VEICULOSSA.
b) Crie os arquivos **.h** com a opção ***File/new/C++ header.***
c) Crie os arquivos **.cpp** com a ***opção File/new/C++ Source***
d) Compile e execute.

Para o TurboC++:

a) Crie todos os arquivos **.h** e **. cpp** com a opção ***File/new***
b) Crie um novo projeto VEICULOSSA.PRJ com a opção ***Project/Open***
c) Inclua todos os arquivos **.cpp** no projeto
d) Compile e execute.

NOTA: *É possível a declaração da implementação dentro da definição das classes. Contudo este tipo de programação é aplicada apenas à pequenos programas, visto que não seria possível a separação em módulos. Se o programa dos VeículosSA fosse rescrito desta forma seria:*

```
#include <iostream.h>

enum TIPO { SEDAN, CAMPO, LUXO, UTILITARIO };

class Veiculo
{
  int n_rodas;
public:
  void setNRodas(int n) { n_rodas = n; }
  int getNRodas() { return n_rodas; } // metodo inline
};

class Caminhao : public Veiculo
{
  int n_carga;
public:
  void setCarga(int c) { n_carga = c; }
  int getCarga() { return n_carga; }  // metodo inline
  void mostra() { cout << "Caminhao: n rodas = "  << getNRodas()
          << " carga = " << getCarga() << endl; }
};

class Automovel : public Veiculo
{
  TIPO n_tipo;
```

```
public:
  void setTipo(TIPO t) { n_tipo = t; }
  TIPO getTipo() { return n_tipo; }  // metodo inline
  void mostra() { cout << "Automovel: n rodas = "  << getNRodas()
          << " tipo = " << getTipo() << endl; }
};

void main()
{
  Caminhao FNM;
  Automovel Fusca;

  FNM.setNRodas(12);
  FNM.setCarga(300);
  FNM.mostra();

  Fusca.setNRodas(4);
  Fusca.setTipo(SEDAN);
  Fusca.mostra();
}
```


Um exemplo simples que podemos analisar e implementar uma classe inicial é a modelagem de um relógio. Um relógio é um objeto que marca horas. Em termos de classificação, não é necessário atribuir marcas nem modelos, assim como não é necessário saber se o relógio é eletrônico ou mecânico.

Os únicos atributos necessários para se classificar um relógio (mínimos) são: hora, minuto e segundo.
Assim teremos a seguinte classificação:

```
#include <iostream.h>

class CRelogio
{
      int   horas,
            minutos,
            segundos;
public:
      void  setHoras(int h);
      void  setMinutos(int m);
      void  setSegundos(int s);
      int   getHoras() { return horas; }
      int   getMinutos() { return minutos; }
      int   getSegundos() { return segundos; }
      void  mostra();
};

void CRelogio::setHoras(int h)
{
      horas = (h>=0 && h<=24) ? h : 0;
}

void CRelogio::setMinutos(int m)
{
      minutos = (m>=0 && m<=59) ? m : 0;
}

void CRelogio::setSegundos(int s)
{
      segundos = (s>=0 && s<=59) ? s : 0;
}
```

```
void CRelogio::mostra()
{
      cout << "Relogio: " << getHoras()
             << ":" << getMinutos()
             << ":" << getSegundos() << endl;
}
void main()
{
      CRelogio R1;

      R1.setHoras(12);
      R1.setMinutos(20);
      R1.setSegundos(0);

      R1.mostra();
}
```


Neste exemplo é fácil perceber que o cálculo das horas, minutos e segundos tem a mesma estrutura funcional. Desta forma pode-se criar um método transparente que calcule em um único bloco.

```
#include <iostream.h>

class CRelogio
{
      int   horas,
            minutos,
            segundos;
      void  setAtributo(int &atributo, int valor, int limite);
public:
      void  setHoras(int h);
      void  setMinutos(int m);
      void  setSegundos(int s);
      int   getHoras() { return horas; }
      int   getMinutos() { return minutos; }
      int   getSegundos() { return segundos; }
      void  mostra();
};

void CRelogio::setAtributo(int &atributo, int valor, int limite)
{
      atributo = (valor>=0 && valor<=limite) ? valor : 0;
}

void CRelogio::setHoras(int h) { setAtributo(horas,h,24); }
void CRelogio::setMinutos(int m) { setAtributo(minutos,m,59); }
void CRelogio::setSegundos(int s) { setAtributo(segundos,s,59); }

void CRelogio::mostra()
{
      cout << "Relogio: " << getHoras()
             << ":" << getMinutos()
             << ":" << getSegundos() << endl;
}
```

Para inicializar o relógio com todos os valores de hora, minuto e segundo, podemos criar um método *ajustar()* que atribui estes valores:

```
void CRelogio::ajustar(int h, int m, int s)
{
```


```
        setHoras(h);
        setMinutos(m);
        setSegundos(s);
}
...
void main()
{
        CRelogio R1;

        R1.ajustar(12,20,0);

        R1.mostra();
}
```

Exercício: Um diretor de uma escola deseja cadastrar (armazenar em um vetor) 10 alunos e 10 professores. Implemente um programa que armazene os 10 alunos e professores em um vetor, mostrando-os em seguida. Note que neste exemplo teremos 3 classes. Para os alunos, usar as características: nome, telefone, id, matrícula e média. Para os professores, usar as seguintes características: nome, telefone, id, registro e salário.

Alocação dinâmica de memória

O procedimento para alocação dinâmica usado em C padrão faz uso das funções `alloc()/malloc()/calloc()` e `free()` para alocação e liberação de recursos de memória. O uso de C++ (POO) vem facilitar o processo de alocação dinâmica com a implementação dos operadores ***new*** e ***delete***.

Os operadores ***new*** e ***delete*** fornecem um meio mais agradável de executar alocação dinâmica de memória (ara qualquer tipo primitivo ou definido pelo usuário) que as chamadas de funções *malloc* e *free* convencionais do C padrão.

Considere o seguinte programa em C:

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

struct Registro
{
      int val;
      int quad;
};

void main()
{
  A   struct Registro *R;
      int i, tam = 10;

      R = (struct Registro *) malloc (tam * sizeof(struct Registro));

      if (R == NULL)
  B   {
            perror("Erro de alocacao\n");
            exit(-1);
      }

      for (i=0; i<tam; i++)
      {
  C         R[i].val = i;
            R[i].quad = i*i;
      }

      for (i=0; i<tam; i++)
  D         printf ("Val=%d, quad=%d\n",R[i].val,R[i].quad);

E     free(R);
}
```

Onde temos que:
A – Declaração de variáveis
B – Alocação de memória
C – Atribuição de valores (entrada de dados)
D – Impressão de valores (saída de dados)
E – Liberação de memória

Passando o mesmo código para C++ teremos:

```
#include <iostream.h>

struct Registro
{
      int val;
      int quad;
};

void main()
{
      Registro *R;
      int i, tam = 10;

      R = new Registro[tam];

      if (R == NULL)
            cerr << "Erro de alocacao\n";
      else
      {
            for (i=0; i<tam; i++)
            {
```

```
                R[i].val = i;
                R[i].quad = i*i;
        }

        for (i=0; i<tam; i++)
                cout  << "Val=" << R[i].val
                      << " quad=" << R[i].quad << endl;

        delete R;
    }
}
```

Construtores e Destrutores

Um relógio sempre que é construído marcará uma hora qualquer aleatoriamente. Após o objeto ser criado, o usuário deve ajustar a hora (atributos) e pode, posteriormente reajustar os atributos.

Um fusca (um automóvel no exemplo descrito anteriormente), entretanto deverá ser criado com 4 rodas no instante da criação. Não faz sentido criar-se um fusca "sem rodas" e ajustar após a criação e nem mesmo após a criação, reajustar o atributo *rodas* para outro valor.

O aluno do exemplo da escola, deverá ser criado com uma identificação (por exemplo, nome, endereço e telefone), pois não seria possível (do ponto de vista de análise) criar o aluno "sem nome", porque para escola este aluno não "existiria". No entanto, é possível reajustar os atributos na vida útil do aluno na escola.

Os objetos podem ser automaticamente inicializados por funções especiais chamadas de construtores. Os construtores são descritos **sempre** com o mesmo nome de sua classe e não tem nenhum tipo de retorno. Desta forma é possível, por sobrecarga, criar-se vários modelos de construtores, que vai depender do tipo de objeto que se está modelando.

Um construtor especial que existe num objeto é o construtor padrão (*default* ou implícito). Este tipo de construtor não tem parâmetros e é chamado implicitamente se a classe não define nenhum construtor explicitamente.

Por exemplo; no exemplo do relógio, é possível criar vários construtores, iremos usar apenas construtores que instanciem um relógio zerando os atributos, outro construtor que dê valor apenas para hora (zero para os outros) e um outro construtor que dá valor para hora, minuto e segundo. Exemplo:

```
void main()
{
      CRelogio R1, R2(13), R3(0,13,4);

      R1.mostra();
      R2.mostra();
      R3.mostra();
}
```

A definição e implementação da *class CRelogio* seria:

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

```
class CRelogio
{
      int   horas,
            minutos,
            segundos;
      void setAtributo(int &atributo, int valor, int limite);
public:
      CRelogio();
      CRelogio(int h, int m=0, int s=0);
      void ajustar(int h, int m, int s);
      void setHoras(int h);
      void setMinutos(int m);
      void setSegundos(int s);
      int   getHoras() { return horas; }
      int   getMinutos() { return minutos; }
      int   getSegundos() { return segundos; }
      void mostra();
};

CRelogio::CRelogio() { ajustar(0,0,0); }
CRelogio::CRelogio(int h, int m, int s) { ajustar(h,m,s); }

void CRelogio::ajustar(int h, int m, int s)
{
      setHoras(h);
      setMinutos(m);
      setSegundos(s);
}

void CRelogio::setAtributo(int &atributo, int valor, int limite)
{
      atributo = (valor>=0 && valor<=limite) ? valor : 0;
}

void CRelogio::setHoras(int h) { setAtributo(horas,h,23); }
void CRelogio::setMinutos(int m) { setAtributo(minutos,m,59); }
void CRelogio::setSegundos(int s) { setAtributo(segundos,s,59); }

void CRelogio::mostra()
{
      cout << "Relogio: " << getHoras()
           << ":" << getMinutos()
           << ":" << getSegundos()<< endl;
}
```

Assim como os “construtores” são métodos especiais para a criação dos objetos, os “destrutores” são métodos especiais para a “destruição” dos mesmos. Um destrutor é chamado quando um objeto sai do escopo ou é apagado (no caso de alocação dinâmica) diretamente. Todo objeto possui um destrutor *default* que apaga o objeto. No caso dos objetos realizarem operações dinâmicas como por exemplo alocações de memória ou abertura (e uso) de arquivos, o destrutor default, quando chamado, não desfaz as alocações de recursos destas operações. Neste caso é necessário sobrecarregar o destrutor padrão para a liberação dos recursos. Assim como os construtores, os destrutores **sempre** usam os mesmos nomes das classes associadas precedidos por um **“~”**.

No exemplo do relógio, como não há alocações dinâmicas, não há necessidade de ser sobrecarregar o destrutor. Vamos ver um pequeno exemplo do uso de destrutores.

Exemplo: Definir uma classe *CString* que crie uma cadeia de caracteres em memória e tenha alguma funcionalidade como por exemplo retornar o tamanho das *string* e concatenar *strings*.

O programa exemplo apresenta: Os construtores: um construtor *default*, um construtor inicializando uma string com 20 bytes e um construtor que inicialize o objeto contendo uma string "STRING". Os métodos: *atribui()*, cujo argumento é uma string convencional e, por sobrecarga com atributo Cstring, e um método *concatena()* cujos atributos são objetos do tipo ***CString***. E por fim um método que escreva a string.


```
void main()
{
      CString s1(20), s2("STRING"), s3;

      s1.atribui("NOVA ");
      s1.concatena(s2);
      s3.atribui(s1);
      s3.mostra();
}
```

A classe *CString* com seus atributos e métodos para este "*pedido*" seria:

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

class CString
{
      char *s;
      int tam;
      bool aloca(int t);
public:
      CString();
      CString(int t);
      CString(char *p);
      ~CString();
      void setTam(int t);
      int   getTam() { return tam; }
      char *getS() { return s; }
      void atribui(char *p); // setS()
      void atribui(CString &s2);
      void concatena(CString &s2);
      void mostra();
};

bool CString::aloca(int t)
{
      tam = 0;
      s = new char[t+1];
      if (s != NULL)
            tam = t;
      return tam != 0;
}

CString::CString()
```

```
{
	s = 0;
	tam = 0;
}

CString::CString(int t)
{
	s = 0;
	aloca(t);
}

CString::CString(char *p)
{
	atribui(p);
}

CString::~CString()
{
	delete s;
}

void CString::atribui(char *p)
{
	int t = strlen(p);

	if (aloca(t))
		strcpy(s,p);
}

void CString::atribui(CString &s2)
{
	int t = s2.getTam();

	if (aloca(t))
	{
		tam = s2.getTam();
		strcpy(s,s2.getS());
	}
}

void CString::concatena(CString &s2)
{
	CString aux(s);
	int t = aux.getTam()+s2.getTam();

	if (s != NULL)
		delete s;

	if (aloca(t))
	{
		strcpy(s,aux.getS());
		strcat(s,s2.getS());
	}
}

void CString::mostra() { cout << getS() << endl; }

void main()
{
	CString s1(20), s2("STRING"), s3;
```

```
        s1.atribui("NOVA ");
        s1.concatena(s2);
        s3.atribui(s1);
        s3.mostra();
}
```

Quando a classe não especifica nenhum construtor padrão, é obrigatório usar um dos construtores da lista para instanciar o objeto.

Se uma classe herda os membros de uma classe que não implemente um construtor padrão, ela deverá "invocar" um dos construtores na linha de implementação do(s) construtor(es), antes da atribuição dos próprios atributos.

Exemplo:

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

class CA
{
        int ca;
public:
        CA(int m);
};

CA::CA(int m) { ca = m; }

class CB : public CA
{
        int cb;
public:
        CB(int m, int n);
};

CB::CB(int m, int n):CA(m)
{
        cb = n;
}
```

Herança múltipla

Herança múltipla significa que uma classe derivada herda os membros de várias classes base. Este recurso deve ser usado quando existe um relacionamento "*é um*" entre um novo tipo e dos ou mais tipos existentes.
Exemplo:

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

class Base1
{
        int base;
public:
        Base1(int);
        int getBase() { return base; }
        void mostra();
```

```
};

Base1::Base1(int m) { base = m; }
void Base1::mostra() { cout << "Base1: " << getBase() << endl; }

class Base2
{
      int base;
public:
      Base2(int);
      int getBase() { return base; }
      void mostra();
};

Base2::Base2(int m) { base = m; }
void Base2::mostra() { cout << "Base2: " << getBase() << endl; }

class Final : public Base1, Base2
{
      int final;
public:
      Final(int,int,int);
      int getFinal() { return final; }
      void mostra();
};

Final::Final(int a, int b, int c):Base1(a),Base2(b)
{
      final = c;
}

void Final::mostra()
{
      Base1::mostra();
      Base2::mostra();
      cout << "Final: " << getFinal() << endl;
}

void main()
{
      Final f1(10,20,30);

      f1.mostra();
}
```

Unified Modeling Language (UML)

A UML é atualmente o esquema de representação gráfica mais amplamente usado para modelagem de sistemas orientados a objetos. Ela certamente unificou os diversos esquemas de notação que existiam no finao da década de 80. A UML é uma linguagem gráfica complexa e repleta de recursos. A UML é usada para modelar sistemas, que podem possuir uma diversidade muito grande. Pode ser usada também em diferentes estágios de desenvolvimento de um sistema, desde a especificação dos requerimentos até os teste de um sistema finalizado.

Cenário de uma aplicação:

Um sistema de software para uma locadora de veículos.

http://www.microsoft.com/brasil/msdn/tecnologias/visualstudio_modelagem.asp

Este cenário detalha como os oito tipos de diagramas UML poderão ser utilizados na modelagem de um sistema de software para uma locadora de veículos. A partir de três casos práticos muito simples, estes exemplos capturam os principais processos inerentes a um sistema desta natureza.

- Diagrama de caso de uso
- Diagrama de classes
- Diagrama de seqüência
- Diagrama de colaboração
- Diagrama de estado
- Diagrama de atividade
- Diagrama de Componentes
- Diagrama de implementação

Diagrama de Caso de Uso.

Um Caso de Uso especifica uma interação entre um usuário e o sistema, no qual o usuário tem um objetivo muito claro a atingir. Um sistema típico de software pode incluir centenas de Casos de Uso simples. Alguns casos de uso que podem ser aplicados a uma locadora de veículos são:

- **O cliente faz a reserva de um carro**. Antes de receber seu carro, o usuário precisa fazer uma reserva. O cliente entra em contato com a locadora e faz um pedido. A locadora aceita ou rejeita a solicitação do cliente, com base em uma série de critérios, entre eles a disponibilidade do carro ou o histórico do cliente perante a locadora. Se a reserva for aceita, a agência preenche um formulário contendo os dados do cliente. O pagamento de um depósito fecha esta etapa da reserva.
- **O cliente retira o carro**. Quando o cliente chega à loja para retirar seu carro, a locadora destaca o modelo solicitado pelo cliente, de acordo com a sua disponibilidade no estoque. Após efetuar o pagamento, o cliente recebe seu carro.
- **O cliente devolve o veículo**. O cliente devolve o carro à locadora no dia especificado pelo contrato de locação.

A Figura abaixo ilustra o diagrama de caso de uso para os três casos de uso assinalados anteriormente.

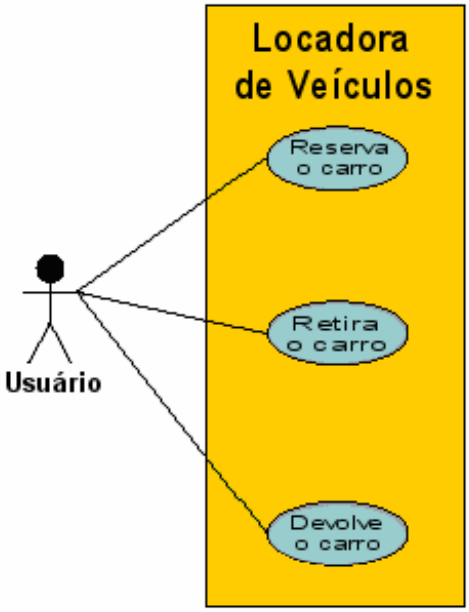


Diagrama para um caso de uso simples.

Diagrama Estrutura Estática (ou Classe)

A próxima tarefa é a classificação dos objetos envolvidos neste processo e a relação de uns com os outros. O exame dos casos de uso ajuda a identificar as classes. As Classes de objetos são modeladas através do uso de diagramas de estrutura estática, ou classe, que mostram a estrutura geral do sistema e também as suas propriedades relacionais e de comportamento.

Num diagrama de Classe, os objetos envolvidos em um sistema de locação de automóveis são agrupados em classes. Cada classe contém um nome de seção e uma atribuição dessa seção. Algumas classes incluem também uma seção operacional que especifica de que forma os objetos devem se comportar dentro daquela classe em particular.

Na classe de clientes, os atributos incluem o nome, telefone, número da carteira de motorista e endereço. A data de nascimento é exigida para garantir que o cliente esteja na faixa etária que o autorize a alugar e dirigir um automóvel. A classe de clientes também armazena algumas outras operações, como, por exemplo, a reserva de veículos.

Os Diagramas de Classe suportam herança de outros sistemas. Na figura abaixo, por exemplo, as classes de Mecânica e Locação herdam alguns atributos, tais como nome e endereço, da classe de Funcionários.

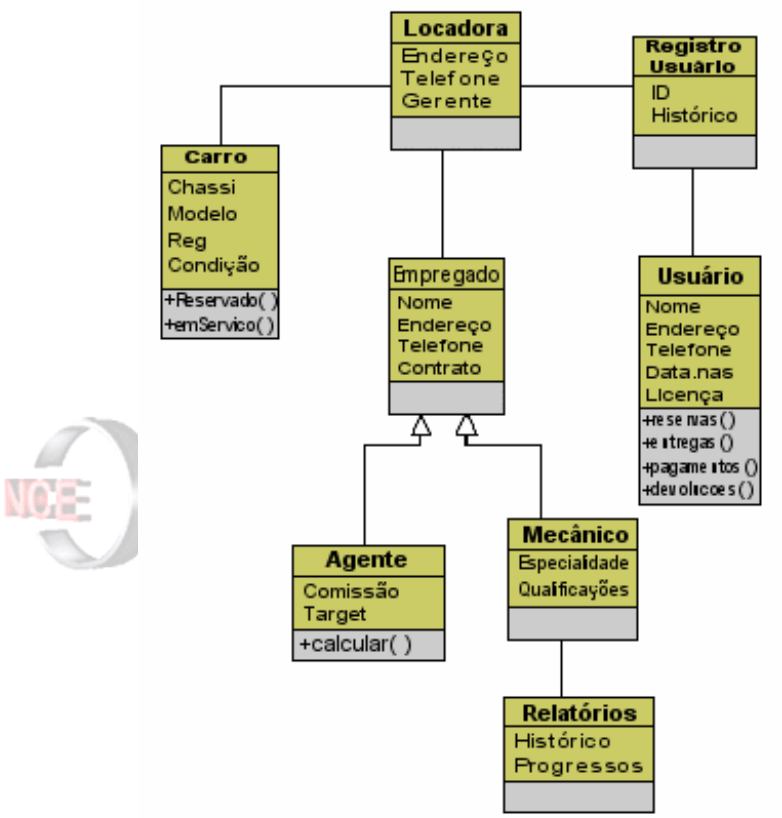


Diagrama de uma estrutura estática ou de classe.

Diagrama de Seqüência

Um Diagrama de Seqüência oferece uma visão detalhada de um caso de uso. Ele mostra uma interação organizada em forma de uma seqüência, dentro de um determinado período de tempo, e contribui para a que se processe a documentação do fluxo de lógica dentro da aplicação. Os participantes são apresentados dentro do contexto das mensagens que se transitam entre eles. Num sistema de software abrangente, um Diagrama de Seqüência pode ser bastante detalhado, e pode incluir milhares de mensagens.

Imagine um cliente que queira fazer uma reserva de um carro. A locadora precisa, antes de tudo, verificar o histórico do cliente, para certificar-se que ele concretizará sua intenção de alugar um carro. Se o cliente já alugou um carro desta empresa antes, seu registro já estará gravado, e a locadora só terá que verificar se todas as operações anteriores foram realizadas sem problemas. Por exemplo, a locadora poderá confirmar se a devolução do veículo se realizou efetivou dentro do prazo estipulado, se o pagamento foi correto, e outros detalhes. Após a aprovação do histórico do cliente, a locadora poderá proceder à aprovação da locação para aquele cliente. Este processo poderá ser representado por um diagrama de seqüência, como o apresentado na Figura abaixo.

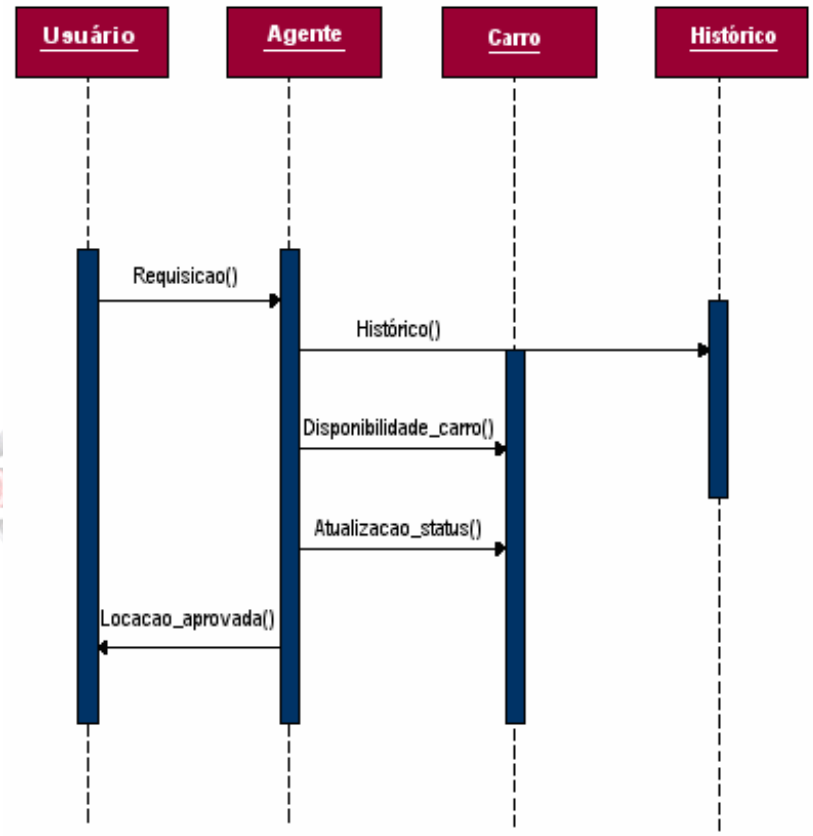


Diagrama de Seqüência.

Diagrama de Colaboração

Um Diagrama de Colaboração é outro tipo de diagrama de interação. Assim como no Diagrama de Seqüência, o Diagrama de Colaboração mostra como um grupo de objetos num caso de uso interage com os demais. Cada mensagem é numerada para documentar a ordem na qual ela ocorre.

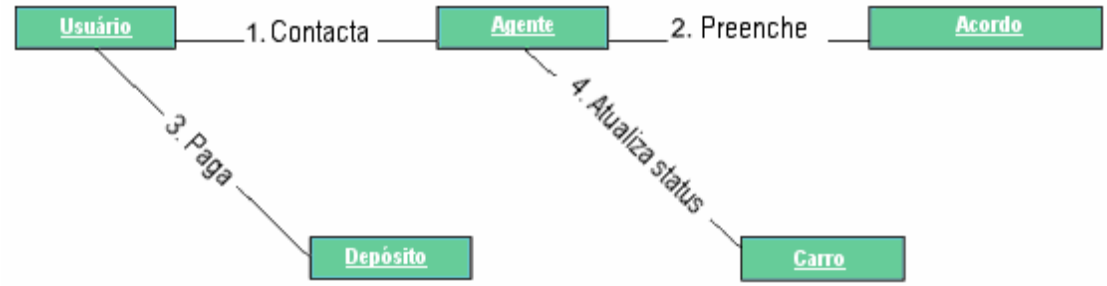


Diagrama de Colaboração

Diagrama de Estado

O estado de um objeto é definido pelos seus atributos em um determinado momento. Os objetos se movem através de diferentes estados, por serem influenciados por estímulos externos. O Diagrama de Estado mapeia estes diferentes estados em que se encontram os objetos, e desencadeia eventos que levam os objetos a se encontrarem em determinado estado em um dado momento. Por exemplo, no sistema de

locação que estamos analisando, o objeto é um veículo. À medida que o veículo se desloca através do sistema de locação, seus vários estados produzem um diagrama complexo, mas altamente ilustrativos. Em primeiro lugar, por exemplo, o veículo é incorporado à frota. Ele permanece no estado de “veículo em estoque” até que ele seja alugado. Após a locação o veículo retorna à frota e ao seu estado de “veículo em estoque. Durante vários momentos de sua vida útil, o carro poderá requerer reparos. Neste caso seu estado passa a ser o de um “veículo em manutenção”. Quando o automóvel finalmente chega ao fim de sua vida útil, ele é vendido ou sucateado, para dar lugar a um veículo novo.

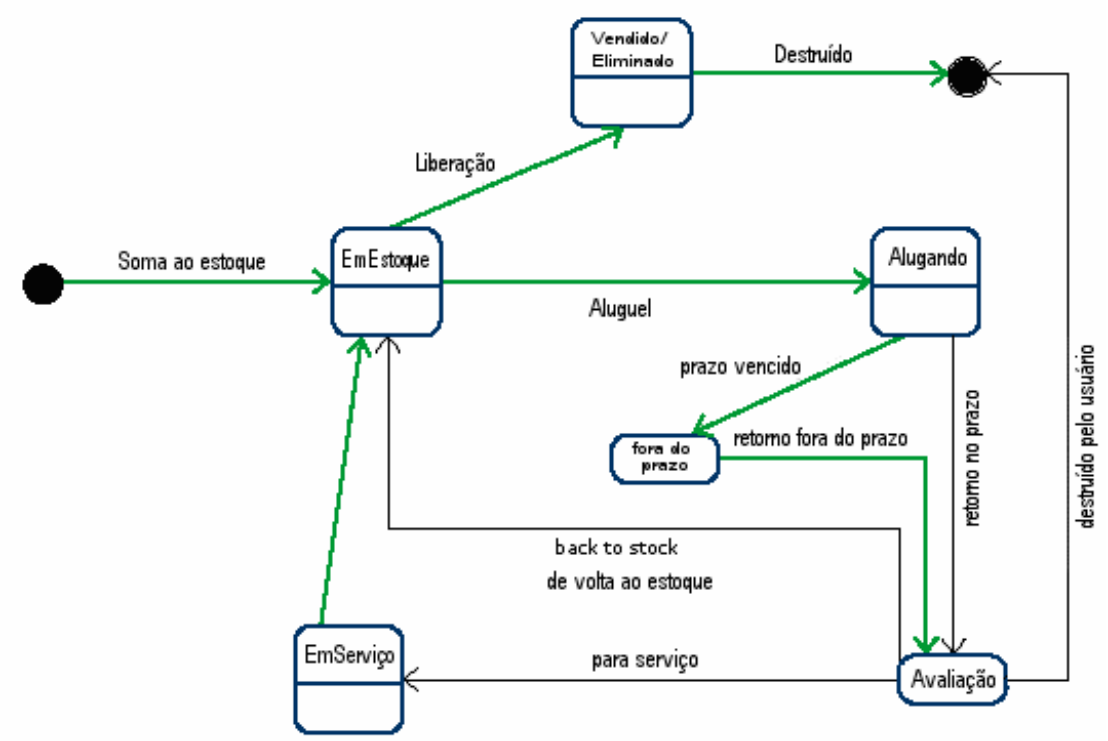


Diagrama de Estado.

Diagrama de Atividade

O Diagrama de Atividade apresenta a lógica que ocorre em resposta a ações desencadeadas internamente. Um Diagrama de Atividade se reporta a uma determinada classe ou caso de uso, mostrando os passos necessários para o desencadeamento de determinada operação.

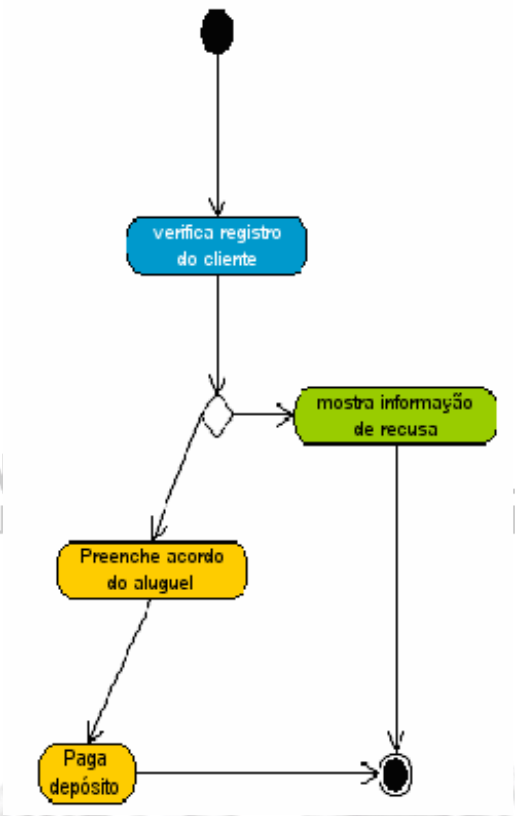


Diagrama de Atividade

Diagrama de Componentes

Um Diagrama de Componentes mostra como os diferentes subsistemas de software formam a estrutura total de um sistema, sistema este que está construído sobre um banco de dados centralizado que contém dados históricos de locações, detalhes dos veículos, registros de manutenção, dados de clientes e funcionários. É muito importante que estas informações estejam centralizadas em um banco de dados, pois os níveis de estoque poderão sofrer alterações a cada hora, e todos os participantes do processo comercial precisam ter acesso às informações mais recentes. Manter os dados atualizados é uma tarefa que requer atualizações em tempo real por parte de todos os integrantes do processo. Os subsistemas de software, por exemplo, têm que estar de posse dos dados sobre os veículos, os serviços, as vendas, os clientes e os funcionários.

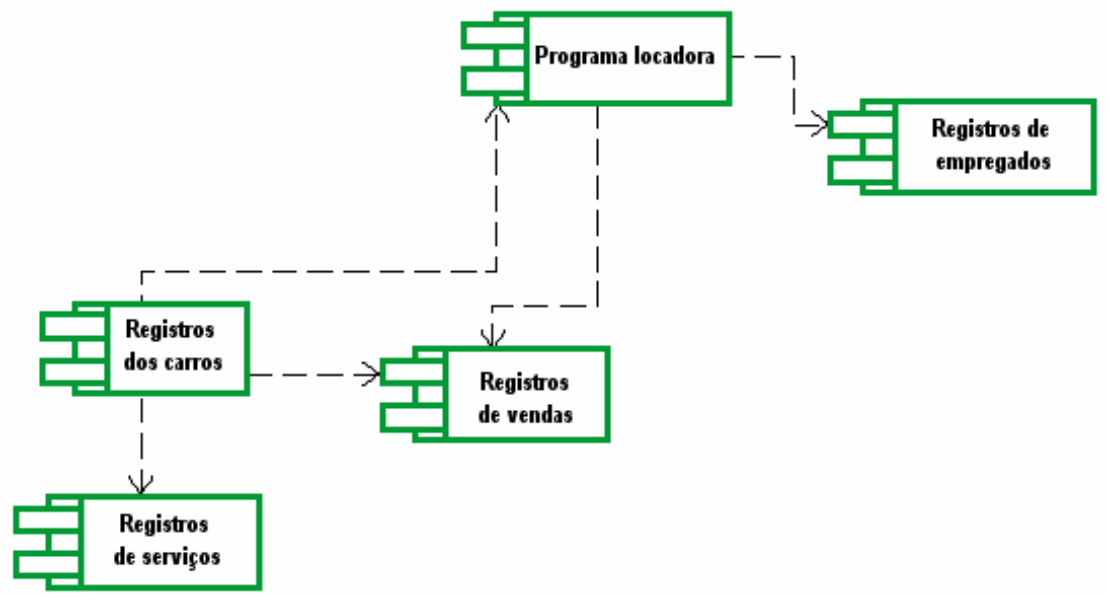


Diagrama de Componentes

Diagrama de Implementação

O Diagrama de Implementação mostra como estão configurados o hardware e o software dentro de um determinado sistema. A empresa locadora tem a necessidade de processar seus dados num sistema cliente/servidor, com um banco de dados centralizado que contenha todos os registros que os profissionais da empresa terão que acessar. Os representantes de locação de veículos precisam ter acesso imediato aos dados sobre a disponibilidade de veículos. Por outro lado, os mecânicos precisam ter meios para destacar que determinado carro está passando por um processo de manutenção.

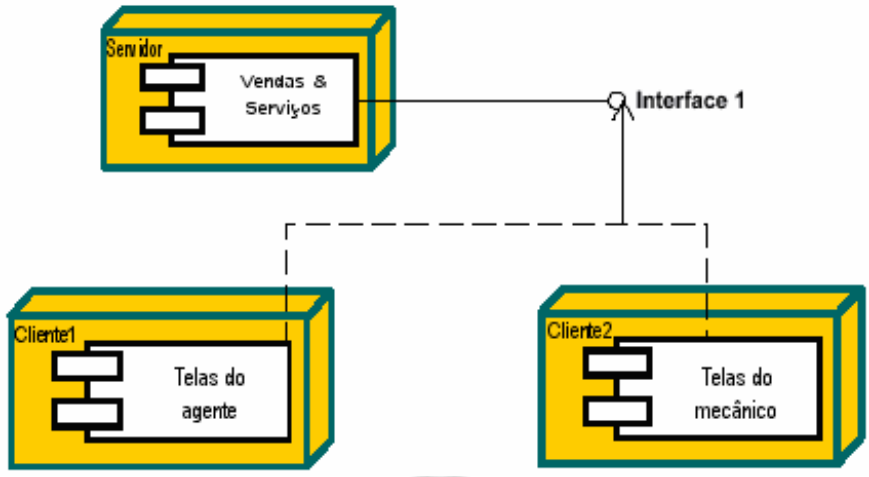


Diagrama de Implementação.

Exercício 1: implementar as classes Empregado, Agente e Mecânico.

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

class CEmpregado
{
```

```
      char  nome[20];
      char  endereco[20];
      int         contrato;
public:
      CEmpregado(char *n, int cntrato=0);
      void setNome(char *n);
      void setEndereco(char *e);
      void setContrato(int c);
      int  getContrato() { return contrato; }
      char *getNome() { return nome; }
      char *getEndereco() { return endereco;}
      void mostra();
};

CEmpregado::CEmpregado(char *n, int cntrato)
{
      setNome(n);
      setContrato(cntrato);
}

void CEmpregado::setNome(char *n)
{
      if (*n != 0)
            strcpy(nome, n);
}

void CEmpregado::setContrato(int c)
{
      contrato = (c > 0) ? c : 0;
}

//-----------------------------------------

class CAgente:public CEmpregado
{
      double      comissao;
public:
      CAgente(char *n, int cntrato=0, double com=0.0);
      void setComissao(double c);
      double getComissao() { return comissao; }
      double calcular() {};
      void mostra();
};

CAgente::CAgente(char       *n,       int       cntrato,       double
com):CEmpregado(n,cntrato)
{
      setComissao(com);
}

void CAgente::setComissao(double c)
{
      comissao = (c > 0) ? c : 0;
}

//-----------------------------------------

enum TESPECIALIDADE {PINTURA, MECANICA, LANTERNAGEM};

class CMecanico:public CEmpregado
```

```
{
      TESPECIALIDADE especialidade;
public:
      CMecanico(char *n,int c=0, TESPECIALIDADE e=MECANICA);
      void setEspecialidade(TESPECIALIDADE e);
      TESPECIALIDADE getEspecialidade() { return especialidade; }
      void mostra();
};

CMecanico::CMecanico(char *n,int c, TESPECIALIDADE e):CEmpregado(n,c)
{
      setEspecialidade(e);
}

void CMecanico::setEspecialidade(TESPECIALIDADE e)
{
      especialidade = e;
}

//-----------------------------------------------
```

Objetos const

Usa-se *const* para especificar um objeto que não possa ser modificável e, que qualquer tentativa de modificar o objeto deve ser considerada como erro de sintaxe. Declarar um objeto como *const* ajuda a seguir o *princípio do mínimo privilégio*. As tentativas de modificar o objeto são detectadas durante a compilação, em vez de causar erros durante a execução.

Basicamente, o *princípio do mínimo privilégio* significa que qualquer objeto (usuário, administrador, programa, sistema, etc.) deveria ter somente os privilégios que o objeto precisa para realizar as suas tarefas. O mínimo privilégio é um princípio importante para limitar a exposição aos ataques e para limitar os danos causados por ataques.

Por exemplo:


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

class Hora
{
      int hora;
public:
      Hora(int h=0);
      void setHora(int);
      int getHora() const { return hora; }
      void mostra();
};

Hora::Hora(int h) { setHora(h); }
void Hora::setHora(int h) { hora = h; }
void Hora::mostra() {   cout << getHora() << endl; }

void main()
{
      Hora h1(6);
      const Hora h2(3);

      h1.setHora(12);
      h2.setHora(21);

      h1.mostra();
      h2.mostra();
}
```

Funções e classes friend

Uma função ***friend*** de uma classe é definida fora do escopo desta, mas tem acesso aos membros ***private*** e ***protected*** da classe. Uma das utilidades para esta função é a generalização do uso de interfaces. Algumas pessoas na comunidade de POO consideram que o uso inadequado de ***friend*** corrompe a ocultação de informações e enfraquece a abordagem de projeto orientado a objetos.

Exemplo:

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

```
#include <string.h>

class Hora
{
      int hora;
      friend void mostra(Hora &h);
public:
      Hora(int h=0);
      void setHora(int);
      int getHora() const { return hora; }
      void mostra();
};

Hora::Hora(int h) { setHora(h); }
void Hora::setHora(int h) { hora = h; }

void mostra(Hora &h)
{
      cout << h.hora << endl;
}

void main()
{
      Hora c1(12);
      mostra(c1);
}
```


Ponteiro this

Todo objeto tem acesso ao seu próprio endereço através de um ponteiro chamado ***this***. Este ponteiro não faz parte do objeto mas é passado para o objeto como primeiro parâmetro implícito em toda a chamada de algum método não-***static*** para o objeto.

Este ponteiro é usado como referência tanto para os atributos e métodos de um objeto.

Nota: Por motivos de economia de memória só existe uma cópia de cada método por classe e este método é invocado para todos os objetos daquela classe. Cada objeto, por outro lado, tem a sua própria cópia dos atributos.

Exemplo:

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

class TesteThis
{
      int x;
public:
      TesteThis(int = 0);
      void print();
};

TesteThis::TesteThis(int a) { x = a; }
void TesteThis::print()
{
      cout  << "\n         x = " << x
            << "\n   this->x = " << this->x
```

```
            << "\n (*this).x = " << (*this).x << endl;
}

void main()
{
      TesteThis objeto( 12 );

      objeto.print();
}
```

Membros static

Cada objeto de uma classe tem sua própria cópia de todos os membros da classe. Em certos casos, todos os objetos de uma classe podem compartilhar apenas uma cópia de uma membro. Um membro da classe ***static*** é a representação das informações de âmbito de toda a classe.

Embora os atributos ***static*** possam se parecer com variáveis globais, eles possuem escopo de classe e devem ser inicializados uma vez apenas. Qualquer modificação feita a partir de qualquer objeto de classe onde implementa um membro ***static*** ficará disponível para todos os objetos.

Um método pode ser declarado como ***static*** se este não faz acesso a nenhum outro membro da classe. Um método ***static*** não possui ponteiro ***this*** porque membros de dados e métodos ***static*** existem independentemente de quaisquer objetos de uma classe.

Exemplo:

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

class Bolsa
{
      double alta, baixa;
      static double dolar;
public:
      Bolsa(double a=0, double b=0);
      void setDolar(double d);
      void mostra();
};

double Bolsa::dolar = 3.20;

Bolsa::Bolsa(double a, double b)
{
      alta = a;
      baixa = b;
}

void Bolsa::setDolar(double d)
{
      dolar = d;
}

void Bolsa::mostra()
{
      cout  << "\nAlta="  << alta
            << "\nBaixa=" << baixa
            << "\nDolar=" << dolar << endl;
}
```

```
void main()
{
      Bolsa BVSP(10,20), BVRJ(12,22);

      BVSP.mostra();
      BVSP.setDolar(4.0);

      BVRJ.mostra();
}
```

Classes containers e iteradores

As classes do tipo *container* (coleção) são classes projetadas para guardar coleções de objetos. Geralmente oferecem funções de inserção, exclusão, busca, classificação, verificação e etc. Exemplos: vetores, pilhas, filas, árvores e listas encadeadas. Um *iterador* é um objeto que retorna o próximo item de uma coleção. Geralmente é associado às classes *containers*.

Alocação dinâmica com os operadores new e delete

O operador *new* cria automaticamente um objeto do tamanho apropriado, chama o construtor do objeto e retorna um ponteiro do tipo correto (ou 0 caso não alocar espaço suficiente).

Para liberar a memória destruído o objeto usa-se delete. Para liberar área alocada para vetores usa-se o operador delete [ ]. Usar delete em vez de delete[ ] para vetores pode levar a erros de lógica durante a execução. Para evitar problemas, o espaço criado como um vetor deve ser apagado como delete [ ] e o espaço criado como um objeto individual deve ser apagado com o operador delete. Veja o exemplo:

```
#include <iostream.h>

class Ponto
{
      int x, y;
public:
      Ponto() { x=y=0; }
      Ponto(int,int);
      void mostra();
};

Ponto::Ponto(int a, int b) { x=a; y=b; }
void Ponto::mostra() { cout << x << "," << y << endl; }

void main()
{
      Ponto *var, *vetor;

      var = new Ponto(10,10);
      vetor = new Ponto[3];

      var->mostra();
      vetor[0].mostra();

      delete var;
      delete[] vetor;
}
```

Sobrecarga de operadores

C++ possibilita a sobrecarga da maioria dos operadores para serem sensíveis ao contexto em que são usados. O compilador gera o código apropriado com base na maneira como o operador é usado. Alguns operadores são freqüentemente sobrecarregados, como operadores de atribuição e aritméticos, tais como + e –.

A sobrecarga de operadores contribui para a extensibilidade da linguagem. Use a sobrecarga de operadores quando for possível tornar o programa mais claro que utilizar chamadas de funções explícitas.

Qualquer operador pode ser sobrecarregado com duas exceções:

a) O operador de atribuição (=) pode ser usado com todas as classes sem sobrecarga explícita (O comportamento padrão é a atribuição membro a membro). Este tipo de atribuição não é recomentado para atributos dinâmicos (tipo ponteiro).
b) O operador de endereço (&) também pode ser usado com objetos de qualquer classe sem ser sobrecarregado.

Operadores que não podem ser sobrecarregados:

. .* :: ?: sizeof

Como um operador tem a funcionalidade de um étodo, a sintaxe é a mesma da declaração de métodos com adição da palavra **operator**.

Ex: *void **operator >>**(OBJETO &v);*

Sobrecarrega de operadores unários

Um operador unário de uma classe pode ser sobrecarregado como um método não *static* sem argumentos.

Exemplo:

```
#include <iostream.h>

class Cor
{
      short c;
public:
      Cor(short v=0);
      Cor &operator !();
      void mostra();
};

Cor::Cor(short v) { c = v; }
Cor &Cor::operator !()
{
      c = 255-c;
      return *this;
}

void Cor::mostra() {    cout << c << endl;}

void main()
{
      Cor c(34), v;
```

```
      v = !c;

      c.mostra();
}
```

Sobrecarrega de operadores binários

Um operador binário pode ser sobrecarregado como uma função membro não *static* com um argumento. Por exemplo, ao sobrecarregar o operador **+=** como um método (não *static*) de uma classe, com um argumento, se **y** e **z** são objetos desta classe, então **y += z** é tratado como se fosse **y.operator +=(z).**
Veja o exemplo:


```
#include <iostream.h>

class Cor
{
      short c;
public:
      Cor(short v=0);
      Cor &operator !();
      Cor &operator +=(Cor &c2);
      void mostra();
};

Cor::Cor(short v) { c = v; }
Cor &Cor::operator !()
{
      c = 255-c;
      return *this;
}

Cor &Cor::operator +=(Cor &c2)
{
      c += c2.c;
      return *this;
}

void Cor::mostra() {    cout << c << endl;}

void main()
{
      Cor c(34), v(6);

      c += (v);

      c.mostra();
}
```

Funções virtuais

Suponha um conjunto de classes de formas geométricas como **círculo**, **triângulo**, **retângulo**, **quadrado** e outras. e sejam todas derivadas da classe **Forma**. Na programação OO, cada uma dessas classes poderia ser dotada da habilidade de desenhar a si própria. Embora cada classe tenha a sua própria função *desenha()*, a função *desenha()* para cada forma é bastante diferente. Portanto, quando se desejar desenhar uma forma qualquer, deve-se usar o método *desenha()* e deixar o programa determinar em tempo de execução qual o método de uma classe derivada que deve ser usado.

Para permitir este tipo de comportamento, declara-se o método *desenha()* na classe básica como uma função virtual e sobrescreve-se (*override*) o método *desenha()* em cada uma das classes derivadas.

Uma vez que o método tenha sido declarado como virtual, este permanece como virtual por toda a hierarquia da herança, mesmo que na hierarquia não se declare como virtual explicitamente.

Veja o exemplo:

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

class Forma
{
      int x,y;
      short cor;
public:
      Forma(int,int,int=0);
      void setX(int);
      void setY(int);
      void setCor(int);
      int getX() { return x; }
      int getY() { return y; }
      int getCor() { return cor; }
      void desenha();
};

Forma::Forma(int a, int b, int c)
{
      setX(a);
      setY(b);
      setCor(c);
}

void Forma::setX(int a) { x = a; }
void Forma::setY(int b) { y = b; }
void Forma::setCor(int c) { cor = c; }
void Forma::desenha()
{
      cout << "Forma x:" << getX() << ", y:"
            << getY() << endl;
}

class Circulo: public Forma
{
      int raio;
```

```
public:
      Circulo(int,int,int,int=0);
      void setRaio(int);
      int getRaio() { return raio; }
      void desenha();
};

Circulo::Circulo(int a, int b, int r, int c):Forma(a,b,c)
{
      setRaio(r);
}

void Circulo::setRaio(int r) { raio = r; }
void Circulo::desenha()
{
      cout << "Circulo x:" << getX() << ", y:" << getY() << ", raio="
           << getRaio() << endl;
}

void main()
{
      Circulo *X;
      Forma *Y;

      //X = new Forma(10,10,10);
      Y = new Circulo(5,5,5);

      //X->desenha();
      Y->desenha();

      //delete X;
      delete Y;

}
R:
Forma x:5, y:5
```


Note que neste exemplo, foi tentado criar um círculo com base em uma forma e uma forma com base em um circulo. Criar um círculo com base em uma forma não é possível pois um círculo não “cabe” em uma forma, mas criar uma forma em um círculo é possível. Entretanto, o resultado não é o esperado pois o método utilizado não é do circulo mas sim da forma.

Para solucionar este problema, basta declarar os métodos a serem utilizado nas demais classes como ***virtual***. Veja a modificação do exemplo:

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

class Forma
{
      int x,y;
      short cor;
public:
      Forma(int,int,int=0);
      void setX(int);
      void setY(int);
      void setCor(int);
      int getX() { return x; }
```

```
      int getY() { return y; }
      int getCor() { return cor; }
      virtual void desenha();
};

Forma::Forma(int a, int b, int c)
{
      setX(a);
      setY(b);
      setCor(c);
}

void Forma::setX(int a) { x = a; }
void Forma::setY(int b) { y = b; }
void Forma::setCor(int c) { cor = c; }
void Forma::desenha()
{
      cout << "Forma x:" << getX() << ", y:"
           << getY() << endl;
}

class Circulo: public Forma
{
      int raio;
public:
      Circulo(int,int,int,int=0);
      void setRaio(int);
      int getRaio() { return raio; }
      void virtual desenha();
};

Circulo::Circulo(int a, int b, int r, int c):Forma(a,b,c)
{
      setRaio(r);
}

void Circulo::setRaio(int r) { raio = r; }
void Circulo::desenha()
{
      cout << "Circulo x:" << getX() << ", y:" << getY() << ", raio="
           << getRaio() << endl;
}

void main()
{
      Forma *Y;

      Y = new Circulo(5,5,5);

      Y->desenha();

      delete Y;
}
R:
Circulo x:5, y:5, raio=5
```

Classes abstratas

Uma classe abstrata é aquela em que não se deve instanciar nenhum objeto, mas que serve de base para uma série de outras classes desejadas.

Por exemplo, ao se pensar na classe Mamíferos, não faz sentido instanciar (criar) nenhum mamífero, pois não existe animal "mamífero". O método "*locomover()",* por exemplo não poderia ter nenhuma implementação pois não se sabe qual mamífero se refere.

Por exemplo. A classe Forma, não prediz nenhuma "forma" praticamente. assim sendo, pode ser transformada em uma classe abstrata pois não seria possível desenhar uma forma da qual não se sabe a "forma". A maneira de transformar uma classe em abstrata e declarar um método como "puro":

```
virtual void desenha() = 0;
```

Veja o exemplo:


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

class Forma
{
      int x,y;
      short cor;
public:
      Forma(int,int,int=0);
      void setX(int);
      void setY(int);
      void setCor(int);
      int getX() { return x; }
      int getY() { return y; }
      int getCor() { return cor; }
      virtual void desenha() = 0; // virtual pura
};

Forma::Forma(int a, int b, int c)
{
      setX(a);
      setY(b);
      setCor(c);
}

void Forma::setX(int a) { x = a; }
void Forma::setY(int b) { y = b; }
void Forma::setCor(int c) { cor = c; }

class Circulo: public Forma
{
      int raio;
public:
      Circulo(int,int,int,int=0);
      void setRaio(int);
      int getRaio() { return raio; }
      void virtual desenha();
};

Circulo::Circulo(int a, int b, int r, int c):Forma(a,b,c)
{
      setRaio(r);
}

void Circulo::setRaio(int r) { raio = r; }
void Circulo::desenha()
```

```
{
      cout << "Circulo x:" << getX() << ", y:" << getY() << ", raio="
            << getRaio() << endl;
}

void main()
{
      Forma a; // vai causar erro....
      Forma *Y;

      Y = new Circulo(5,5,5);

      Y->desenha();

      delete Y;
}
```


O polimorfismo é implementado por meio de funções virtuais. Quando é feito um pedido através de um ponteiro (ou referência) de uma classe básica para usar um método virtual, o C++ escolhe o método (sobrecarregado) correto na classe derivada associada com o objeto.

Embora não se pode instanciar objetos de classe abstratas, pode-se declarar ponteiros e referências para estas classes. Estes ponteiros (ou referências) podem ser usados para possibilitar manipulações polimórficas de objetos de classes derivadas quando tais objetos forem instanciados a partir de classes concretas.

Entrada e saída com streams

O sistema de entrada e saída em C++ ocorre em *streams* de bytes. Um *stream* é simplesmente uma seqüência de bytes. Nas operações de entrada, os bytes fluem de um dispositivo (o teclado por exemplo) para a memória principal. Nas operações de saída, os bytes fluem da memória principal para um dispositivo (vídeo por exemplo).

O C++ fornece recursos de E/S de baixo e de alto nível. Os recursos de baixo nível se referem a E/S não formatada. Especificam que um número de bytes deve ser transferido diretamente de um dispositivo para a memória e vice-versa. Estes recursos fornecem alta velocidade de transferência. Na E/S de alto nível, os bytes são agrupados em unidade significativas, como inteiros números de ponto flutuante, *strings* e tipos definidos pelo usuário. Estes recursos (orientados a tipos) são satisfatórios para a maioria das operações de E/S, exceto para o processamento de grandes arquivos.

A biblioteca iostream oferece centenas de recursos de E/S. Define os objetos **cin**, **cout**, **cerr e clog** (*iostream.h*) que correspondem ao *stream* padrão de entrada. O arquivo *iomanip.h* declara os serviços úteis para executar operações de processamento de arquivos, com os chamados *manipuladores* de *streams parametrizados*.

A classe *istream* e *ostream* são derivadas (por herança simples) da classe base *ios*. A classe *istream* é derivada através de herança múltipla.

- **cin** é uma instância da classe *istream* e é vinculado ao dispositivo de entrada padrão.
- **cout** é uma instância da classe *ostream* e é vinculado ao dispositivo de saída padrão.
- **cerr** é uma instância da classe *ostream.* E é vinculado ao dispositivo de erro padrão e
- **clog**, instância da classe *ostream,* é vinculado ao dispositivo de erro padrão.

As saídas para **cerr** não são colocadas em um *buffer.* Cada inserção do *stream* **cerr** faz com que a saíra apareça imediatamente. Já com **clog**, as saídas são colocadas em um *buffer*. Cada inserção em **clog** pode fazer com que sua saída seja mantida em um *buffer* até estar cheio ou ser esvaziado.

Manipuladores de streams

C++ oferece vários *manipuladores de streams* que executam tarefas de formatação. Os recursos podem ser definição de larguras de campo, definição de precisão, definição e redefinição de indicadores de formato etc.

- **setbase**: define a base. Pode ser 10, 8 ou 16
- **hex**: define a base como hexadecimal
- **oct**: define a base como octal
- **dec**: define a base como decimal

```
int n = 212;
cout  << n << endl << hex << n << endl
      << setbase(10) << n << endl;
```

- **setprecision**: define a precisão para números de ponto flutuante

```
double x = 3.1415;
cout << setprecision(2) << x << endl;
```

- **setw**: define a largura do campo

```
int w = 4;
cout << setw(5) << w << endl;
```

Processamento de arquivos

Os arquivos representam a armazenagem de dados em mídias de conservação permanente (teoricamente...). O processamento de arquivos padrão pode ser classificado segundo o tipo de acesso: *seqüencial* ou *aleatório* e, segundo ao modo de acesso: acesso de alto (*formatado*) e de baixo nível (*não formatado*). Em C++, pode-se classificar o tipo de tratamento do arquivo pelo formato. Isto é: arquivos texto e arquivos binários.

Em qualquer linguagem de programação e para todo tipo de arquivo, sempre serão realizadas as operações básicas de entrada/saída:

1. Definição dos ponteiros dos arquivos
2. Abertura (criação) dos arquivos
3. Processamento dos arquivos (transferência, gravação...)
4. Fechamento dos arquivos.

Arquivos texto

Utilizam *ifstream* e *ofstream* para abrir (criar) arquivos. A transferência pode ser realizada usando os operadores **>>** e **<<** sobrecarregados ou os métodos *get()* e *getline()*.
Veja os exemplos:

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

void main()
{
	char c;

	ofstream Saida("c:\\temp\\saida.txt");
	ifstream Entrada("c:\\temp\\entrada.txt");

	if (!Saida || !Entrada)
	{
		cerr << "Nao foi possivel abrir o(s) arquivo(s)\n";
		exit(-1);
	}

	while ( Entrada >> c )
		Saida << c;
}
```

Neste exemplo, a saída será gravada ignorando espaços em branco, pois o operador **>>** foi implementado para ignorar espaços em branco. Para este tipo de caso, pode-se usar o método *get()* e *put()*:

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

void main()
{
```

```
      char c;

      ofstream Saida("c:\\temp\\saida.txt");
      ifstream Entrada("c:\\temp\\entrada.txt");

      if (!Saida || !Entrada)
      {
            cerr << "Nao foi possivel abrir o(s) arquivo(s)\n";
            exit(-1);
      }

      while ( (c = Entrada.get()) != EOF)
            Saida.put(c);

      // Entrada.close();
      // Saida.close();
}
```


Para usar um arquivo que permita gravação e leitura basta alterar o segundo parâmetro para *ios::in* ou *ios::out*. Para a gravação e leitura é bastante útil usar os métodos *ifstream::seekg()* e *ofstream::seekp()*. (também pode-se usar os métodos *ifstream::tellg()* e *ofstream::tellp()* para retornarem a posição corrente do ponteiro do arquivo).

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

void main()
{
      char c;

      ofstream Saida("c:\\temp\\saida.txt", ios::in || ios::out);
      ifstream Entrada("c:\\temp\\entrada.txt");

      if (!Saida || !Entrada)
      {
            cerr << "Nao foi possivel abrir o(s) arquivo(s)\n";
            exit(-1);
      }

      Saida.seekp(0,ios::end);

      while ( (c = Entrada.get()) != EOF)
            Saida.put(c);

      // Entrada.close();
      // Saida.close();
}
```

Onde o parâmetro ios::end significa fim do arquivo. Esta opção pode ser *ios::beg*, *ios::cur* e *ios::end*.

Arquivos binários

Utilizam os métodos *read()* e *write()* para a leitura e gravação dos dados respectivamente. O método *write()* espera um dado do tipo *const char* * como primeiro arqumento, por isso é necessário compatibilizar o tipo dos

dados. Pode-se usar o operador *reinterpret_cast* para converter o tipo. Veja o exemplo:

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

void grava()
{
	double f;
	int i;

	ofstream saida;

	saida.open("c:\\temp\\saida2.dat",ios::binary);
	if (!saida)
	{
		cerr << "Erro na criacao do arquivo\n";
		exit(-1);
	}

	for (i=0; i<10; i++)
	{
		f = sqrt(i);
		saida.write((unsigned char *)&f,sizeof(f));
	}
	//saida.close();
}

void le()
{
	double f;

	ifstream entrada;

	entrada.open("c:\\temp\\saida2.dat",ios::binary);

	if (!entrada)
	{
		cerr << "Erro na criacao do arquivo\n";
		exit(-1);
	}

	while(entrada.read(reinterpret_cast<unsigned
char*>(&f),sizeof(f)))

		cout << f << endl;
	//entrada.close();
}

void main()
{
	grava();
	le();
}
```

Gabaritos de classes

Uma classe pode ser definida contendo um ou mais tipos genéricos (classes genéricas) definidas em *template*. Assim como a simples definição de uma classe não gera código, a definição de uma classe com *template* também não. Com *templates*, pode-se desenvolver dois tipos de elementos: funções globais genéricas (já visto) e classes genéricas.
Exemplo:

```
#include <iostream.h>

template <class T>

class matrix33
{
      T v[3][3];
public:
      T &operator() (int i, int j) { return v[i][j]; }
};

typedef matrix33 <int>              i_matrix;
typedef matrix33 <float>            f_matrix;
typedef matrix33 <double>           d_matrix;
typedef matrix33 <long double>      l_matrix;

void main()
{
      i_matrix im;
      f_matrix fm;
      l_matrix lm;
      d_matrix dm;

      im(0,0) = 3;       cout << im(0,0) << endl;
      fm(0,0) = 1.43;    cout << fm(0,0) << endl;
      lm(0,0) = 4.31;    cout << lm(0,0) << endl;
      dm(0,0) = 0.14;    cout << dm(0,0) << endl;
}
```

Tratamento de exceções

O código de tratamento de erros varia em natureza e entre diversos sistemas de software. Produtos comerciais possuem muito mais códigos de tratamento de erros que programas chamados de “informais”. Alguns exemplos comuns de erros, ou exceções, são falhas na operação de alocação de memória, usando new, um subscrito de vetor fora dos limites, estouro em operações matemáticas; divisão por zero e parâmetros de funções inválidos.

O tratamento de exceções é especialmente apropriado para situações que o programa não é capaz de se recuperar, mas precisa fazer uma “limpeza final” organizada e então terminar “elegantemente”.

Evite usar tratamento de exceções para fins diferentes do tratamento de erros, pois pode reduzir a clareza dos programas.

O tratamento de exceções deve ser usado somente para processar situações excepcionais.

O tratamento de exceções em C++ se destina a situações em que a função que possibilita um erro fica impossibilitada de tratá-lo. Esta função dispara (*throw*) uma exceção. O código que pode gerar o erro deve estar dentro de um bloco ***try*** seguido por um ou mais blocos ***catch***. Cada bloco ***catch*** contém um tratador de exceção que especifica o tipo da exceção que pode capturar e tratar. Se nenhum tratador dor encontrado, é chamado a função ***terminate***, que, por default, chama a função ***abort()***.

Exemplo: divisão por zero.

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

class ExcecaoDeDivisaoPorZero
{
      const char *mensagem;
public:
      ExcecaoDeDivisaoPorZero():mensagem("Divisao por zero") {}
      const char *OQue() const { return mensagem; }
};

double quociente(int num, int denum)
{
      if (denum == 0)
            throw ExcecaoDeDivisaoPorZero();

      return (double) num/denum;
}

void main()
{
      int num1 = 0, num2 = 0;
      double result;

      cout << "Digite dois inteiros: ";
      cin >> num1 >> num2;

      try
      {
            result = quociente(num1,num2);
            cout << "O quociente e':" <<  result << endl;
      }
      catch ( ExcecaoDeDivisaoPorZero ex )
      {
            cout << "Ocorreu um erro:" << ex.OQue() << endl;
      }
}
```

Outro exemplo:

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

void throwException()
{
      try
      {
            cout << "Funcao throwException()\n";
            throw exception();
```


```
	}
	catch( exception ex )
	{
		cout << "Excecao tratada na funcao throwException()\n";
		throw;
	}
	cout << "Este codigo nao deve ser impresso\n";
}

void main()
{
	try
	{
		throwException();
		cout << "Isto nao deve ser impresso\n";
	}
	catch ( exception e )
	{
		cout << "Excecao tratada em main()\n";
	}
	cout << "Controle do programa continua apos a captura em
main()\n";
}
```