Universidade Federal de Santa Maria
Departamento de Eletrônica e Computação
Prof. Cesar Tadeu Pozzer
pozzer@inf.ufsm.br
08/03/2018

Introdução a Programação Orientada à Objetos na Linguagem C++

1 – Introdução

A programação orientada a objetos surgiu com o principal objetivo de unir os dados e funções em um único elemento: o objeto. Esta metodologia traz uma série de vantagens sobre linguagens de programação procedural, como o C:

- **Reusabilidade**: as classes que compõem um sistema podem ser aproveitadas em outros sistemas, sem qualquer alteração, pois dados e funções estão contidos dentro da classe. Caso haja necessidade, pode-se criar novas classes baseadas em outras já existentes, herdando as características da classe pai. (Modularização)
- **Encapsulamento**: proteção dos dados contra alterações indevidas. O encapsulamento mantém escondidos dados e métodos do objeto. Pode-se explicitamente declarar o grau de visibilidade de atributos e métodos.
- **Produtividade**: A partir do momento que temos a disposição uma coleção de classes devidamente testadas e com um funcionamento a prova de erros, para criar novos sistemas basta usar estas classes, sem nenhuma necessidade de reescrever e adaptar código. Isto dá, sem sombra de dúvida, maior rapidez e, conseqüentemente, produtividade no desenvolvimento de sistemas.

A Linguagem C++ é uma extensão da linguagem C que incorpora recursos de Orientação a Objetos. A linguagem C++ herdou todas as características de C, e adiciona recursos de programação orientada a objetos. Sempre que possível será feita uma analogia entre C e C++. **OBS: todos os arquivos de um projeto C++ devem ter a extensão .cpp.**

2 – Classe

Uma classe é um tipo definido pelo usuário, semelhante a uma estrutura, com o adicional que funções também podem ser inseridas. Estas funções (métodos) vão agir sobre os dados (atributos) da classe. O seguinte exemplo mostra um exemplo hipotético de um fragmento de código em C e seu equivalente em C++.

C	**C++**

`typedef struct{`
`  int x, y:`
`}Ponto;`
 	
`void set(Ponto *p, int x, int y)`
`{`
`  p->x = x;`
`  p->y = y;`
`}`
	`class Ponto {`
	`public:`
	`  int x, y;`
	`  void set(int a, int b)`
	`  {`
	`  x = a;`
	`  y = b;`
	`  }`
	`}; //não esqueca o ";"`

Toda classe deve ter um construtor, que é um método com o mesmo nome da classe, sem valor de retorno. Ele é chamado quanto um objeto é inicializado e nele são definidos, pelo programador, os atributos que devem ser inicializados. Uma classe pode ter vários construtores, com diferentes argumentos (sobrecarga). As classes também podem ter um destrutor, que é um método sem parâmetros e sem retorno chamado quando o objeto é

desalocado. O destrutor não pode ser chamado explicitamente e tem somente como função fornecer ao compilador o código a ser executado quando o objeto é destruído.

```
#include <stdio.h>

class Ponto
{
public:
   int x, y;
   Ponto(void) //método construtor
   {
      x = y = 0;
   }
   ~Ponto(void) //Destrutor. Chamado quando o objeto é liberado
   {
      //não faz nada neste caso
   }
};


int main (void)
{
   Ponto p1;     //chama o construtor default imediatamente
   Ponto p2();   //chama o construtor imediatamente
   Ponto *pp;    //não chama o construtor pois pp é ponteiro
   pp = new Ponto(); //chama o construtor para pp
   printf("%d %d ", p1.x, pp->x);
   return 0;
}
```

2 – Objeto

Um objeto é uma instância da classe, da mesma forma como uma variável de estrutura em C. Por meio dele é que os métodos e variáveis da classe, ou superclasses, podem ser acessados. Os objetos podem ser alocados de forma estática ou dinâmica, com o seguinte exemplo.

C	C++
`Ponto p; //variável`	`Ponto p; //objeto`
`Ponto *p = (Ponto*) malloc(sizeof(Ponto));`	`Ponto *p = new Ponto();`

3 – Método

Métodos são funções associadas a uma classe, e são chamadas pelos objetos pertencentes a classe ou por outros métodos da classe. No seguinte exemplo, para fazer a inicialização do Ponto p, em C deve-se chamar a função inicia passando-se a referência de qual variável será inicializada. Em C++, o objeto p chama seu método próprio inicia(int, int) que faz sua inicialização. Este exemplo também mostra como é feita a inicialização no caso de ponteiros.

C	C++

`Ponto p;`
`inicia(&p, 10, 20);`
	`Ponto p;`
	`p.inicia(10, 20);`
	`Ponto p2(3,3);`

`Ponto *p =(Ponto*)malloc(sizeof(Ponto));` `inicia(p, 19, 20);`	`Ponto *p = new Ponto();` `p->inicia(10, 20);`

4 – Sobrecarga de funções

Ocorre quanto uma classe possui mais de um método com o mesmo nome, porém com diferentes tipos de argumentos ou número de argumentos.

```
class Ponto{
private:
   int x,y;
public:
   Ponto(float x1, float y1)
   {
      x = (int)x1;
      y = (int)y1;
   }
   Ponto(int x1, int y1)
   {
      x = x1;
      y = y1;
   }
   Ponto(Ponto p)
   {
      x = p.x;
      y = p.y;
   }
   Ponto()
   {
      x = 0;
      y = 0;
   }
};
```

5 – Proteção de Dados e Funções

Uma das características da programação orientada a Objetos é o encapsulamento. O C++ oferece 3 níveis (modificadores) de proteção de métodos e atributos:

- **public**: métodos e atributos podem ser acessados de qualquer parte do programa;
- **private**: (valor default) Apenas métodos da classe podem acessar. Tentativas de chamadas de métodos private fora da classe resultam em erro de compilação;
- **protected**: Apenas métodos da classe e derivados (herdados) tem acesso.

No próximo exemplo são ilustrados os vários tipos de permissão de acesso a métodos e atributos de classe. O construtor deve ser sempre public.

```
class Ponto
{
// x e y só podem ser usadas por métodos da classe e derivadas
protected:
   int x, y;
//método que só pode ser chamado por métodos da classe Ponto2D
private:
   bool podeMover(int x1, int y1)
   {
      if( (x+x1) < 100 && (y+y1)<100)
         return 1;
      return 0;
   }
//métodos públicos
public:
   Ponto(int x1, int y1)
   {
      x = x1; y = y1;
   }
   void move(int x1, int y1)
   {
      if( podeMover(x1, y1) )
      {
         x+=x1; y+=y1;
      }
```

```
int main (void)
{
   Ponto p1(0, 0);
   p1.podeMover(1,1);
   p1.imprime();
   p1.move(1,1);
   p1.imprime();
   return 0;
```

não é permitido. Erro de Compilação

```
    }
    void imprime()
    {
        printf("\nx= %d, y= %d", x, y);
    }
};
```

6 – Herança

É o mecanismo que permite uma classe herdar todo o comportamento e atributos de outra classe. A classe que herda é chamada de subclasse e a que fornece é chamada de superclasse. No construtor da subclasse pode-se passar parâmetros para o construtor da classe pai, como no seguinte exemplo. Neste exemplo, se na definição da classe Cavalo não fosse chamado o construtor da classe Animal, geraria um erro de compilação pois não existe um construtor default sem nenhum parâmetro.

```
#include <stdio.h>

class Animal{
    int idade;
public:  //qualquer um pode chamar estes métodos
    Animal(int i)
    {
        idade = i;
    }
    void respira(void)
    {
        printf("\nRespirando...");
    }
};

//Classe Cavalo herda os métodos da classe Animal
class Cavalo : public Animal{
    int velocidade;
public:
    Cavalo(int i, int v): Animal(i) {
        velocidade = v;
    }
    void corre(void)
    {
        printf("\nCorrendo a %d Km/h", velocidade);
    }
};
```

```
int main (void)
{
    Cavalo cavalo(10, 30);
    Animal animal(20);
    cavalo.respira();
    cavalo.corre();
    animal.respira();
    return 0;
}
```

7 – Funções virtuais

Funções de uma classe podem ser redefinidas em classes derivadas. Isto é chamado de overriden de funções. O uso da palavra reservada virtual à frente do método informa ao compilador que podem existir novas definições deste método em classe derivadas. Quando uma função virtual é chamada, é verificada a versão do objeto que a chamou para que seja chamada a função correspondente. No seguinte exemplo, a função Desenha da classe Figura, é definida apenas nas classes derivadas.

```
#include <stdio.h>

class Figura{
```

```
public:
   Figura()
   {
   }
   //este método virtual é definido nas classes derivadas
   virtual void Desenha()
   {
      printf("\nEu sou uma figura ");
   }

   //virtual float Area() = 0; //método virtual puro
};

class FigRetangular: public Figura{
public:
   FigRetangular() : Figura()
   {
   }
   void Desenha()
   {
      printf("\nEu sou uma figura Retangular");
   }
};

class FigOval: public Figura{
public:
   FigOval() : Figura()
   {
   }
   void Desenha();
};

// ***** Implementação do método Desenha de FigOval ******
void FigOval::Desenha()
{
   printf("\nEu sou uma figura oval");
}

int main()
{
   Figura *f1, *f2;
   f1 = new FigOval();       // inicializado com o tipo FigOval */
   f2 = new FigRetangular(); // inicializado com o tipo FigRetangular
   f1->Desenha();  //chama o método da classe FigOval
   f2->Desenha();  //chama o método da classe FigRetangular
   return 0;
}
```

Neste exemplo, o método `Desenha()` de Figura é virtual, ou seja, classes derivadas **podem** reimplementar este método. Por sua vez, o método comentado `virtual Area() = 0;` está definido como sendo virtual puro. Neste caso todas as classes imediatamente derivadas de Figura **devem** obrigatoriamente implementar este método, que pode ser virtual mas não virtual puro.

Funções virtuais é um recurso muito poderoso da linguagem C++. Considere um outro exemplo, como no caso de um jogo onde existem diversos tipos de personagens, como soldados, fuzileiros e tanques. Todos os personagens podem realizar as mesmas funções como correr, atirar, fugir, procurar energia, perseguir o inimigo e morrer.

Inicialmente vamos considerar a implementação deste jogo sem usar funções virtuais e sem os demais recursos da linguagem C++. Supondo que um time controlado pelo computador (adversário) possui várias unidades de vários tipos. Para controlar estas unidades, na implementação pode-se criar listas dinâmicas para cada tipo de

personagem. Assim, vai ter uma lista com todos os soldados, outra com os fuzileiros e outra com os tanques. Quando o jogo está sendo executado, cada unidade deve realizar suas ações, como se deslocar, perseguir o inimigo e atirar. Para isso, o seguinte algoritmo poderia ser utilizado.

```
for(int i=0; i < numSoldados; i++)
{
   processaSoldado(soldado[i]);
}
for(int i=0; i < numFuzileiros; i++)
{
   processaFuzileiro(fuzileiro[i]);
}
for(int i=0; i < numTanques; i++)
{
   processaTanque(tanque[i]);
}
```

Esta mesma estrutura pode ser utilizada em diversos locais do jogo, com por exemplo para a) avaliar o nível de vida de cada uma das unidades para descobrir se existe alguma que necessite de reparo, b) para avaliar mensagens recebidas, ou c) para fazer a renderização gráfica das unidades. Para isso, uma estratégia um pouco melhor seria usar uma lista (vetor) heterogênea, onde cada elemento da lista (vetor) armazena um ponteiro void e um inteiro indicando o tipo de informação armazenada.

```
#define N 100
typedef struct no
{
   int tipoUnidade; //pode ser SOLDADO, FUZILEIRO ou TANQUE.
   void *unidade;
}No;
No unidade[N];

for(int i=0; i<N; i++)
{
   switch(unidade[i].tipoUnidade)
   {
      case SOLDADO:
         processaSoldado((Soldado*)unidade[i].unidade);
         break;
      case FUZILEIRO:
         ....
   }
}
```

Utilizando-se os recursos da linguagem C++, este problema seria resolvido da seguinte forma:

```
for(i=0; i<numUnidades; i++)
{
   unidade[i].processa();
}
```

Além do fato de ser um código muito mais compacto, também apresenta outra vantagem de maior importância ainda. Considere que em um determinado momento, seja necessário incluir mais um tipo de unidade no jogo: o avião. Para os dois primeiros casos, deve-se alterar o código pela adição de mais um for() e mais um case, respectivamente, em todos os locais do programa onde os personagens são processados. Para o último caso, nada precisa ser alterado. Basta criar uma nova classe avião, também derivada da classe UnidadeMovel.

8 – Estruturação de um programa C++

Programas escritos em C++ são compostos por arquivos .cpp e arquivos .h. Nos arquivos .h encontram-se a definição das classes, atributos e métodos, porém é também permitida a colocação de código. Os arquivos .cpp possuem as implementações dos métodos descritos nos arquivos .h. Para o exemplo da Seção 7, poderiam ser gerados 4 arquivos: dois de protótipos e dois de implementação.

Arquivo Ponto.h - Definição da classe Ponto

```
#ifndef __PONTO_H__
#define __PONTO_H__

#include <iostream>

class Ponto
{
   int x, y;
public:
   Ponto();
   Ponto(const Ponto &p); //copy constructor
   Ponto(int x1, int y1);
   void imprime();
};
#endif
```

Arquivo Ponto.cpp - Implementação da classe Ponto

```
#include <stdio.h>
#include "Ponto.h"

Ponto::Ponto()
{
   x = y = 0;
}

Ponto::Ponto(int x1, int y1)
{
   x = x1; y = y1;
```

Define o escopo

```
}

Ponto::Ponto(const Ponto &p)
{
   x = p.x;
   y = p.y;
}

void Ponto::imprime()
{
   printf("\n Dados do ponto: (x=%d, y=%d) ", x, y);
}
```

Arquivo Circ.h - Definição da classe Circ

```
#ifndef __CIRC_H__
#define __CIRC_H__
```

```
#include "Ponto.h"

class Circ : public Ponto
{
   int raio;
public:
   Circ(Ponto p, int r);
   void imprime();
};
#endif
```

Arquivo Circ.cpp - Implementação da classe Circ

```
#include <stdio.h>

#include "Circ.h"

Circ::Circ(Ponto p, int r): Ponto(p)
{
   raio = r;
}
void Circ::imprime()
{
   printf("\nRaio do circulo: %d", raio);
   Ponto::imprime();
}
```

main.cpp – deve incluir os arquivos Ponto.h e Circ.h

```
#include "Ponto.h"
#include "Circ.h"

void main (void)
{
   Ponto p1(10, 40);
   Circ  c1(p1, 40);
   p1.imprime();
   c1.imprime();
}
```

9 – Exemplo Comparativo entre C e C++

```
#include <stdio.h>
typedef struct{
   int x,y;
}Ponto;
void setPonto(Ponto *p, int x, int y)
{
   p->x = x;
   p->y = y;
}
void imprimePonto(Ponto p)
{
   printf("\n\nDados do ponto: ");
   printf("x= %d, y= %d", p.x, p.y);
}
```

```
#include <stdio.h>
class Ponto{
private:
   int x,y;
public:
   Ponto(int x1, int y1)
   {
      x = x1;
      y = y1;
   }
   void imprime()
   {
      printf("\n\nDados do ponto: ");
      printf("x=%d,y=%d", x,y);
```

```
                                        }
                                    };
```

```
typedef struct {
   Ponto p;
   int raio;
}Circ;
void setCirculo(Circ *c, Ponto p, int r)
{
   c->p = p;
   c->raio = r;
}
void imprimeCirculo(Circ c)
{
   printf("\n\nDados do Circulo: ");
   printf("x=%d,y=%d", c.p.x, c.p.y);
   printf(" Raio=%d",  c.raio);
}
```

```
class Circ: public Ponto{
   int raio;
public:
   Circ(int x, int y, int r):Ponto(x,y)
   {
      raio = r;
   }
   void imprime()
   {
      printf("\nDados do circulo:");
      printf("%d", raio);
      Ponto::imprime();
   }
};
```

```
void main (void)
{
   Ponto p1;
   Circ  c1;
   setPonto (&p1, 10, 20);
   setCirculo (&c1, p1, 40);
   imprimePonto(p1);
   imprimeCirculo(c1);
}
```

```
int main (void)
{
   Ponto p1(10, 40);
   Circ  c1(1, 2, 40);
   p1.imprime();
   c1.imprime();
   return 0;
}
```

10 – Namespace (Por Vitor Conrado, Cesar Pozzer)

Namespace é um espaço ou uma região, dentro de um programa, onde um nome (uma variável, uma função, uma classe, etc..) é considerado válido. Bjarne Stroustrup [4] (o criador da linguagem C++) diz em seu livro que namesapce é um mecanismo para expressar agrupamentos lógicos.

Este conceito apareceu porque as primeiras linguagens de programação (como é o caso do BASIC) só tinham as chamadas Global Variables, isto é, uma variável que estaria presente durante todo o programa - mesmo dentro das funções. Isto tornava a manutenção de programas grandes uma tarefa muito difícil. Para resolver o problema as novas linguagens de programação criaram o conceito de namespace. A linguagem C++ permite que o próprio programador crie o seu namespace em qualquer parte do programa. Isso é muito bom para quem cria bibliotecas e quer manter os nomes das suas funções únicas, quando integradas com bibliotecas de outros programadores.

O uso de namespace é aconselhado para programas que tenham um grande número de bibliotecas, pois facilita a organização e a identificação do escopo de cada função/variável/classe. É interessante também, já que permite que existam funções com mesmo nome em namespaces diferentes, que também é uma das vantagens do uso de classes.

```
#include <iostream>
int main(){
   std::cout<<"Ola Mundo !!! "<<std::endl;
   return 0;
}
```

cout (**que faz o papel da função printf do C**) está no escopo `std`, por isso foi necessária a utilização do operador de escopo :: para sua utilização. O mesmo programa poderia ser escrito assim:

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

int main(){
   cout<<"Ola Mundo !!! "<<endl;
   return 0;
}
```

Neste segundo exemplo a linha `using namespace std;` indica ao compilador que está sendo utilizado o escopo std, desta forma não é mais necessário utilizar o operador de escopo. Este mesmo exemplo ainda poderia ser escrito da seguinte forma:

```
#include <iostream>
using std::cout;
using std::endl;

int main(){
   cout<<"Ola Mundo !!! "<<endl;
   return 0;
}
```

Neste terceiro exemplo é indicado que serão usados `cout` e `endl` do escopo `std`, as demais funções/variáveis/classes do escopo std deverão ser utilizadas usando o operador de escopo `std::`.

OBS: não existe diferença quanto ao tamanho do programa final com a utilização de `using namespace std;` ou `using std::cout; using std::endl;` ..., (dúvida encontrada em fóruns de discussão).

Criando Namespace

A sintaxe para a criação de namespace é:

```
namespace Nome{
   [Classes]
   [Funções]
   [Variáveis]
   ...
}
```

Exemplo – Arquivo de Interface Teste.h:

```
namespace Teste2
{
   void func2();
}

namespace Teste
{
   class MinhaClasse
   {
      int valor;
      public:
      MinhaClasse();
      void func();
   };
```

```
   void func();
   int a;
}
```

Exemplo – Arquivo de Implementação Teste.cpp

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

using namespace std;

// aqui poderia usar também
// using namespace Teste; e
// using namespace Teste2;

void Teste2::func2(){
   cout<<"Funcao 2, dentro do namespace Teste2"<<endl;
}

//construtor de MinhaClasse, definida no namespace Teste.
//O escopo da classe ainda deve ser incluido
Teste::MinhaClasse::MinhaClasse()
{
   this->valor = 10;
   cout<<"Construtor da classe Classe"<<endl;
}

void Teste::Classe::func()
{
   cout<<"Valor: "<<valor<<endl;
}
```

Exemplo – Arquivo main.cpp

```
// main.cpp
#include <stdio.h>
#include "teste.h"

using namespace std;
using namespace Teste;
using Teste2::func2;
// ou using namespace Teste2;

int main()
{
   Classe c;
   printf("Ola Mundo");
   func2();
   c.func();
}
```

O uso de namespaces permite que sejam criadas funções com mesmo nome em escopos diferentes, porém ainda existe o problema de existirem namespaces com nomes iguais quando se utiliza bibliotecas de terceiros. Por isso é aconselhado que se utilize nomes pouco comuns ou grandes para namespaces. Para facilitar a utilização desses namespaces pode ser criado um alias, da seguinte forma:

```
namespace Exemplo_de_Namespace{
   ...
}
namespace EN = Exemplo_de_Namespace;

EN::func();
```

11 – Sobrecarga de Operadores

Da mesma forma como métodos, pode-se também fazer a sobrecarga de operadores, o que permite a realização de operações sobre tipos de dados não básicos sem a necessidade de chamadas de funções específicas.
A seguinte tabela ilustra a lista de todos os operadores que podem ser sobrecarregados.

```
+     -     *     /     =     <     >     +=    -=    *=    /=    <<    >>   <<=   >>=   ==    !=
<=    >=    ++    --    %     &     ^     !     |     ~     &=    ^=    |=    &&   ||    %=    []
()    new   delete
```

A definição de sobrecarga é feita pela definição de métodos de uma classe cujo nome é operator seguido pelo operador que se deseja sobrecarregar. O seguinte exemplo ilustra a definição da classe Vector3, para manipulação de vetores.

```
class Vector3
{
public:
   float x, y, z;
   Vector3(float vx, float vy, float vz)
   {
      x = vx;  y = vy;  z = vz;
   }
   Vector3(const Vector3& u)
   {
      x = u.x;    y = u.y;    z = u.z;
   }

   void operator+= (const Vector3& v)
   {
      x += v.x;    y += v.y;    z += v.z;
   }

   //produto vetorial entre dois vetores
   Vector3 operator ^ (Vector3& v)
   {
      Vector3 aux( y * v.z - z * v.y, z * v.x - x * v.z, x * v.y - y * v.x );
      return( aux );
   }

   Vector3 operator + (const Vector3& v)
   {
      Vector3 aux( x + v.x, y + v.y, z + v.z );
      return( aux );
   }

   void imprime( ) const
   {
      printf("%f %f %f", x, y, z);
   }
};
```

Neste exemplo pode-se observar o surgimento de dois operadores: const e &. O operador const é usado para assegurar que o parâmetro passado para a função não poderá ser alterado. O operador & indica que o parâmetro será passado por referência, porém todas as operações sobre ele e em sua chamada são como se fosse uma variável passada por valor. O operador const ao lado da declaração da função indica que a função não poderá alterar nenhum membro da classe.

Com a definição destes operadores, pode-se facilmente somar, incrementar e calcular o produto vetorial de dois vetores, como no seguinte exemplo.

```
  Vector3 v1(1,1,1), v2(2,2,2), v3(1,1,1);
  v1 += v2;
  v1.imprime();
  v3 = v1^v2;
  v3.imprime();
```

A implementação de uma API para fornecer suporte a manipulação de vetores, em C++, é de vital importância para acelerar o desenvolvimento de um jogo ou aplicativo gráfico. Toda API deve operar sobre uma estrutura que define como o vetor está alocado.

O nome da estrutura varia segundo o gosto de cada programador. É comum encontrar definições como Vector, Vector3, Vector3D, dentre outras. Neste documento, é adotado o nome Vector3. O elenco de operações implementadas pode ser muito amplo, podendo contemplar a soma de vetores, multiplicação, rotação, ângulos, produtos escalares, dentre outros. Cabe como exercício a criação desta API.

12 – Métodos Inline

Funções inline são comuns em C++, tanto em funções globais como em métodos de classes. Estas funções têm como objetivo tornar o código mais eficiente. Elas são tratadas pelo compilador quase como uma macro: a chamada da função é substituída pelo corpo da função. Para funções pequenas, isto é extremamente eficiente, já que evita geração de código para a chamada e o retorno da função.

Este tipo de otimização normalmente só é utilizado em funções pequenas, pois funções inline grandes podem aumentar muito o tamanho do código gerado. Do ponto de vista de quem chama a função, não faz nenhuma diferença se a função é inline ou não.

Nem todas as funções declaradas inline são expandidas como tal. Se o compilador julgar que a função é muito grande ou complexa, ela é tratada como uma função normal. No caso de o programador especificar uma função como inline e o compilador decidir que ela será uma função normal, será sinalizada uma warning. O critério de quando expandir ou não funções inline não é definido pela linguagem C++, e pode variar de compilador para compilador. Para a declaração de funções inline foi criada mais uma palavra reservada, inline. Esta deve preceder a declaração de uma função inline:

```
inline double quadrado(double x)
{
   return x * x;
}
```

Alguns detalhes devem ser levados em consideração durante a utilização de funções inline. O que o compilador faz quando se depara com uma chamada de função inline? O corpo da função deve ser expandido no lugar da chamada. Isto significa que o corpo da função deve estar disponível para o compilador antes da chamada. Um módulo C++ é composto por dois arquivos, o arquivo com as declarações exportadas (.h) e outro com as implementações (.c ou .cpp). Tipicamente, uma função exportada por um módulo tem o seu protótipo no .h e sua implementação no .cpp. Isso porque um módulo não precisa conhecer a implementação de uma função para usá-la. Mas isso só é verdade para funções não inline. Por causa disso, funções inline exportadas precisam ser implementadas no .h e não mais no .cpp.

13 – Templates

A sobrecarga de funções permite a declaração de métodos com mesmo nome e diferentes tipos de parâmetros. Porém, para cada tipo de parâmetro, um novo método deve ser definido. Uma solução para este problema é o uso de Templates, que permitem criar funções genéricas que admitem qualquer tipo de parâmetro para argumentos e

retorno. O seguinte exemplo ilustra a sintaxe para definição de uma função template que retorna o maior valor entre dois números de mesmo tipo.

```
template <class GenericType>
   GenericType getMax (GenericType a, GenericType b)
   {
      return ( a>b ? a:b );
   }
```

A função getMax pode ser chamada com qualquer tipo de parâmetro, que será substituído por GenericType em tempo de execução. No seguinte exemplo a função getMax é chamada com parâmetros inteiros e long.

```
template <class T>
T GetMax (T a, T b)
{
   T result;
   result = (a>b)? a : b;
   return result;
}
```

```
int main ()
{
   int i=5, j=6, k;
   long l=10, m=5, n;
   k = GetMax<int>(i, j);
   n = GetMax<long>(l, m);
   k = GetMax (i, j);
   n = GetMax (l, m);
   return 0;
}
```

A definição do tipo de parâmetro "<int>, <long>" é opcional pois o compilador detecta automaticamente o tipo e o converte. Caso os parâmetros forem de tipos diferentes, deve-se definir o Template com dois tipos genéricos

```
template <class T, class U>
   T GetMin (T a, U b)
   {
      return (a<b?a:b);
   }
```

Templates também podem ser definidos para comportar classes, que podem conter métodos com parâmetros genéricos, como no seguinte exemplo:

```
#include <stdio.h>

template <class T>
   class Ponto
   {
      T x, y;
   public:
      Ponto (T xx, T yy)
      {
         x = xx;
         y = yy;
      }
      T getX()
      {
         return x;
      }
      T getY()
      {
         return y;
      }
   };
```

A definição de um objeto do tipo par chamado Ponto, para armazenar dois valores inteiros, tem a seguinte sintaxe:

```
int main()
{
   Ponto<int>   p1(2115, 360);
   Ponto<float> p2(1.66, 0.1);
   int i = p1.getX();
   printf("%d", i);
   return 0;
 }
```

14 – STL - Standard Template Library

STL é um conjunto de tipos abstratos de dados e funções projetados para manipularem diferentes tipos de dados de forma transparente. As operações sobre os dados são realizadas por métodos que também aceitam tipos de dados genéricos. Como o próprio nome diz, STL faz uso de templates, já definidos, para implementação de diversos algoritmos que manipulam dados de forma eficiente, como por exemplo vetores, dinâmicos, listas, pilhas, etc. No STL também foi adicionado um tipo String, que facilita as operações de manipulação de caracteres quando comparado a biblioteca string.h da linguagem C. Além disso, também existem funções de busca, ordenação, dentre outros.

Nesta seção serão abordadas 3 estruturas seqüenciais: vetor, fila dupla (Deque – Double End Queue) e lista. A escolha de uso é dada em função das operações que se deseja realizar sobre dos dados. A seguinte tabela mostra o tempo gasto para diferentes operações sobre cada um dos tipos de representação.

Tabela 1: Tempo de execução de diferentes operações [10]

Operação	Vector	Deque	Lista
Acessar o primeiro elemento	Constante	Constante	Constante
Access último elemento	Constante	Constante	Constante
Accessar elementos randômicos	Constante	Constante	Linear
Add/Delete no início	Linear	Constante	Constante
Add/Delete no final	Constante	Constante	Constante
Add/Delete randomicamente	Linear	Linear	Constante

Ao se usar STL, deve-se ter em mente o conceito de namespace, que permite agrupar classes, objetos e funções e associá-los a um nome, o que facilita a quebra de escopos globais em subescopos menores. Para acessar as variáveis a e b do seguinte exemplo, deve-se indicar o escopo associado usando-se o operador ::.

```
namespace teste
{
   int a, b;
}

int main()
{
    teste::a = 8;
    printf("%d", teste::a);
    return 0;
 }
```

Em STL, todas as classes, objetos e funções são definidos com o namespace std. Pode-se usar o operador de escopo a cada uso da STL ou indicar o namespace corrente, da seguinte forma:

```
using namespace std;
```

Para fazer o percorrimento de todos os elementos da estrutura são usados Iterators. Eles podem ser vistos como ponteiros para posições específicas da estrutura. Nas próximas seções são apresentados vários exemplos do uso de namespace e iterators para 3 tipos de containers STL.

14.1 – Vector

O tipo vector é similar a um array, com a diferença que possui tamanho dinâmico. As principais funções definidas são:

`begin()`	Retorna um iterator para o início do vetor
`end()`	Retorna um iterator para o final do vetor
`push_back()`	Adiciona elemento no final do vetor
`pop_back()`	Destrói elemento no final do vetor
`Size()`	Número de elementos do vetor
`[]`	Operador de acesso randômico

Exemplo sem namespace global

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

int main(void)
{
   int val;
   std::vector<int> v;
   std::vector<int>::iterator iter;

   printf("Digite alguns inteiros: ");
   do{
      scanf("%d", &val);
      v.push_back(val);
   }while(val!=-1);

   for( iter=v.begin(); iter != v.end(); iter++ )
      printf("%d ", *iter);

   //acesso com o operador randomico
   for( int i=0; i<v.size(); i++)
      printf("%d ", v[i] );
   return 0;
}
```

Exemplo com namespace global

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

using namespace std;

int main(void)
{
   int val;
   vector<int> v;
   vector<int>::iterator iter;

   printf("Digite valores: ");
   do{
      scanf("%d", &val);
        v.push_back(val);
   }while(val!=-1);

   return 0;
}
```

14.2 – Deque

Uma Deque é muito similar a um vetor, com a característica de apresentar rápida inserção e remoção de elementos no início e no final. Possui funções de acesso iguais ao Vector, e duas adicionais:

`push_front( )`	Adiciona elemento no início da Deque
`pop_front( )`	Destroi elemento no início da Deque

Para usar este template, deve-se adicionar a diretiva #include <deque>.

14.3 – List

O tipo List, como mostrado na Tabela 1, deve ser usado preferencialmente quando existem muitas inserções e remoções de dados em posições intermediárias. Em todas as demais operações, por não possuir o operador [], tem um alto custo computacional. O tipo List faz uso de uma implementação com listas duplamente encadeadas, o que necessita grande consumo de memória. Os principais métodos são semelhantes ao Deque.

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

using namespace std;
```

```
int main( void )
{
   //lista de ponteiros para Vector
   list<Vector*> lista;
```

```
class Vector
{
   int x, y;
public:

   Vector(){ }

   Vector(const int mx, const int my)
   {
      x = mx;
      y = my;
   }
   void imprime()
   {
      printf("\n\nDados do ponto: ");
      printf("x=%d, y=%d", x, y);
   }
};
```

```
   list<Vector*>::iterator iter;

   for(int i=0; i<3; i++)
   {
      lista.push_front( new Vector(i, i) );
   }

 //insere elemento em posicao intermediaria
   iter = lista.begin();
   iter++;
   lista.insert( iter, new Vector(20,20) );

   for(iter = lista.begin();
       iter!=lista.end(); iter++)
   {
      Vector *v1 = *iter;
      v1->imprime();
   }
   return 0;
}
```

15 – Inclusão Cruzada de Arquivos

Em algumas situações, é necessário que duas classes se auto-referenciem. Como exemplo, suponha que existam as classes Ator e Comportamento. A classe Ator tem uma instância da classe Comportamento e a classe comportamento faz uso de argumentos do tipo Ator. O Seguinte trecho de código não funciona para este caso.

Ator.h

```
#ifndef __ACTOR_H__
#define __ACTOR_H   __

#include "Vector.h"
#include "Comportamento.h"

class Ator
{
   Vector pos, dir;
   Comportamento *c;
public:

   Ator(char *name, int id);
   ...
};
#endif
```

Comportamento.h

```
#ifndef __COMPORT_H__
#define __COMPORT_H __

#include "Vector.h"
#include "Ator.h"

class Comportamento
{
   int tipo;
public:
   Comportamento(Ator *a, int tipo);
   ...
};


#endif
```

Ator.cpp

```
#include "Ator.h"
...
```

Comportamento.cpp

```
#include "Comportamento.h"
...
```

Para solucionar este problema, deve-se na interface da classe comportamento não incluir o arquivo Ator.h, mas sim apenas declarar a existência da classe Ator. No arquivo Comportamento.cpp deve-se adicionar a inclusão do arquivo Ator.h.

Ator.h

```
#ifndef __ACTOR_H__
#define __ACTOR_H   __

#include "Vector.h"
#include "Comportamento.h"

class Ator
{
public:
   Vector pos, dir;
   Comportamento *c;
```

Comportamento.h

```
#ifndef __COMPORT_H__
#define __COMPORT_H __

#include "Vector.h"

class Ator;  //define a classe Ator

class Comportamento
{
   int tipo;
public:
   Comportamento(Ator *a, int tipo);
```

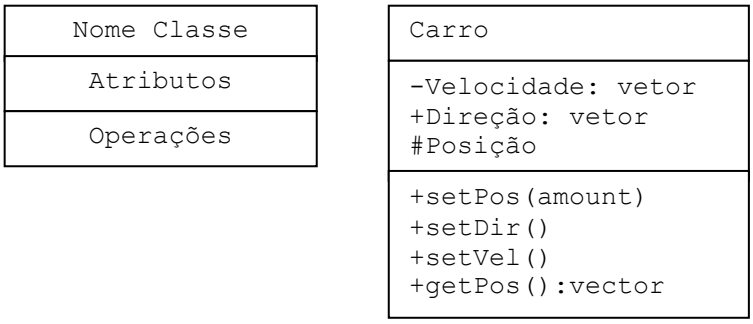
`Ator(char *name, int id);`
`...`
`};`
`#endif`
	`...`
	`};`
	
	`#endif`

Ator.cpp	Comportamento.cpp

`#include "Ator.h"`
`...`
	`#include "Comportamento.h"`
	`#include "Ator.h"` `//inclui a classe Ator`
	`...`

16 – Diagramas de Classe UML

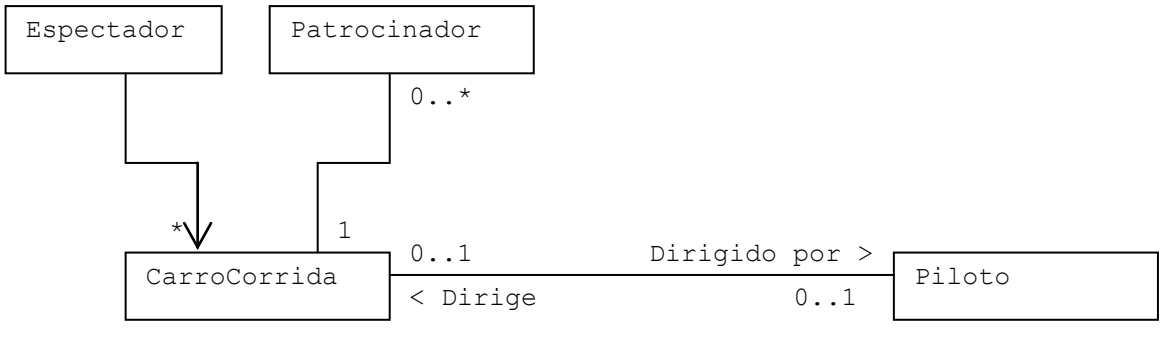
A linguagem UML (Unified Modeling Language) é uma ferramenta muito utilizada na modelagem Orientada a Objetos. O diagrama mais comum desta linguagem é o Diagrama de Classes, utilizado para descrever os relacionamentos entre as classes que compõem a aplicação. Nesta seção é feita uma abordagem introdutória sobre os principais recursos do diagrama de classes. Para definir uma classe, utiliza-se um retângulo que pode ser dividido em 3 regiões: nome da classe, atributos e operações.

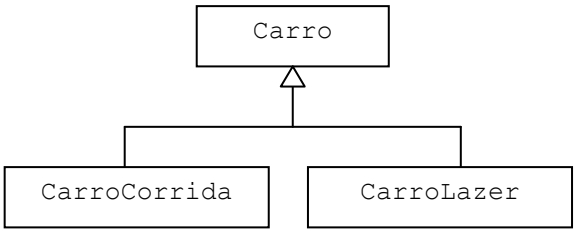


Para este exemplo, a classe é chamada Carro, tem 3 atributos (Velocidade, Direção e Posição) e 4 métodos. O atributo Velocidade é private (-), Direção é public (+) e Posição é protected (#). Todos os métodos são public e o método getPos() retorna um vector.

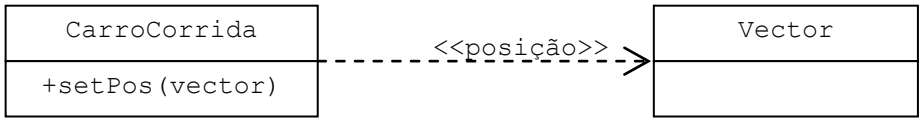
Pode-se definir o relacionamento entre classes por meio de Associação, Herança ou Dependência. No relacionamento de **Associação**, utiliza-se uma linha sólida para descrever o relacionamento. No seguinte exemplo, o carro de corrida é dirigido por até um piloto e o piloto dirige no máximo 1 carro de corrida. Cada carro de corrida pode ter vários patrocinadores, mas cada patrocinador pode ter exatamente 1 carro de corrida (o parâmetro 1 pode ser omitido neste caso). Cada carro de corrida pode ter um número qualquer de espectadores. Neste caso, o carro de corrida não sabe quais são os seus espectadores (devido à linha direcionada).

Para a definição de **herança**, utiliza-se uma seta direcionada apontando na direção do pai da classe, como mostrado no seguinte exemplo.





O último exemplo de relacionamento é por **dependência**. Este ocorre quando, por exemplo, um método de uma classe recebe como parâmetro uma instância de uma outra classe. Utiliza-se uma linha pontilhada direcionada.



17 – Estrutura de um programa Gráfico em C++ com OpenGL

Ao se analisar um código fonte (supostamente para compreender como ele funciona), inicialmente abre-se o arquivo main.cpp, para então começar a analisar classes mais gerais para depois, se necessário, olhar as classes mais específicas. Deste modo, para facilitar a compreensão do código, as classes de mais alto nível devem ser simples, com pouco código e com poucas funções matemáticas. Em um jogo, por exemplo, podem existir classes `Carro`, `Cenario`, `Manager`, `IA`, `Vector`, `Fisica`, `Math`, etc.

Considere um programa em OpenGL, ou implementado por meio da Canvas2D, que implemente um veículo controlado pelo usuário por meio do teclado. Para garantir uma função `main()` compacta, todos os códigos de tratamento de eventos de mouse/teclado devem ser repassados para classes especializadas, como no seguinte exemplo:

```
void keyboardFunc(unsigned char key, int x, int y)
{
   case seta_frente:
      carro->move(1);
   case seta_left:
      ...
}
```

Não se deve na função `keyboardFunc()`, por exemplo, determinar a nova posição do carro. Os cálculos de posição e direção envolvem conceitos matemáticos que devem estar encapsulados dentro da classe `Carro` ou classes herdadas ou associadas.

Como uma aplicação gráfica executa continuamente, pela chamada da função `render()`, caso for pressionada uma tecla para mover o carro para frente, deve-se evitar de incrementar a posição do carro na chamada `move()`. Deve-se indicar na classe `Carro` que a tecla foi pressionada e cada vez que a `render()` do carro for chamada, deve-se então incrementar a posição do veículo. Isso também reduz a utilização de variáveis **globais**.

A função `render()` da `main()` deve mais uma vez ser compacta e deve apenas invocar os métodos `render()` dos objetos que compõem a aplicação.

```
void render()
{
   camera->render();
   carro->render();
   cenario->render();
   text->render()
   ...
}
```

Obviamente que para se obter um código bem estruturado deve-se investir um tempo na especificação das classes e hierarquias.

Apesar da interface do OpenGL ser em linguagem C, pode-se encapsulá-lo em uma classe C++, como no seguinte exemplo.

Arquivo `myGL.h`

```
class myGL
{
public:
   myGL(int argc, char* argv[]);
private:
   //estes metodos devem ser estaticos
   static void callbackDisplay();
   static void callbackKeyboard(unsigned char key, int x, int y);

   void render();
};
```

Arquivo `myGL.cpp`

```
#include <GL/glut.h>
#include <stdio.h>
#include "myGL.h"

myGL *_this; //guarda referencia para o objeto para ser usado em métodos estáticos

myGL::myGL(int argc, char* argv[])
{
   _this = this;
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
   glutCreateWindow ("GLUT");

   glutDisplayFunc(this->callbackDisplay);
   glutKeyboardFunc(myGL::callbackKeyboard);

   glutMainLoop();
}

void myGL::callbackDisplay()
{
   _this->render();
}

void myGL::callbackKeyboard(unsigned char key, int x, int y)
{
   printf("\nTecla pressionada: %d" , key);
}

void myGL::render()
{
  //código de desenho
}
```

Desta forma, a função main() de uma aplicação gráfica em OpenGL pode ser escrita da seguinte forma:

```
int main (int argc, char* argv[])
{
   new myGL();
}
```

18 – Exercícios

1. Definir uma hierarquia de classes para a especificação de:
   a. Um veículo;
   b. Quadro de funcionários de uma empresa;

c. Jogo composto por um cenário, personagens de vários tipos, ações e troca de mensagens entre personagens;

2. Implemente um editor de figuras geométricas utilizando STL e funções virtuais. Implemente o mesmo editor na linguagem C para analisar vantagens da linguagem C++.
3. Implemente a classe Vector2 que disponibiliza operações sobre vetores 2D.
4. Implemente a classe Vector3 que disponibiliza operações sobre vetores 3D.

19 – Referências Bibliográficas

[1] Soulié, J. CPP Tutorial. Disponível em: http://www.cplusplus.com/doc/tutorial/
[2] Ottewell, P. STL Tutorial. Disponível em: http://www.yrl.co.uk/~phil/stl/stl.html
[3] Borges, R., Clinio, A. L. Programação Orientada a Objetos com C++. Tutorial
[4] Stroustrup, B. The C++ Programming Language. Addison-Wesley Professional, 3 edition, 911 p., 2000.
[5] Buckland, M. Programming Game AI by Example. Wordware publishing Inc, 2005.
[6] Nathan Myers. Artigos C++ no Mundo Real. Disponível em: http://www.arnaut.eti.br/op/CPPAR002.htm
[7] Resources Network, Namespaces. Disponível em: http://www.cplusplus.com/doc/tutorial/namespaces.html
[8] BUENO, A. D., Apostila de Programação Orientada a Objeto em C++. Disponível em: http://www.apostilando.com/download.php?cod=283&categoria=C%20e%20C++
[9] The Standard Template Library Tutorial. Disponível em: http://www.infosys.tuwien.ac.at/Research/Component/tutorial/prwmain.htm
[10] Phil Ottewell's STL Tutorial. Disponível em: http://www.pottsoft.com/home/stl/stl.htmlx
[11] Bjarne Stroustrup's homepage. Disponível em http://www.research.att.com/~bs/homepage.html