UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
(DAELN)

Curso: Engenharia Eletrônica, **Disciplina:** Fundamentos de Programação II (IF62C), **Turma:** S__
Professores: Hermes Del Monego ( ), Jean Simão ( ), Robinson Vida ( ) **Data:** _________.

Aluno: **Código:**_____, **Início** __:__ **Fim** __:__

2ª Prova

1) $_{[2{,}5\,pt]}$ Antônio possui os seguintes materiais de construção: tijolos, madeira, telhas, portas (portas com tamanhos diferentes: altura, largura e quantidade de folhas), janelas (janelas com tamanhos diferentes: altura, largura e quantidade de folhas), escadas. Neste caso, o Antônio precisa de ajuda para que os produtos comprados por ele sejam agrupados de acordo com as suas características físicas. Usando os conceitos de orientação a objetos lecionados durante o semestre, monte o diagrama de classes completo de maneira que possa ajudá-lo a construir uma casa com escada e cobertura, composta de no mínimo 5 peças as quais contenham pelo menos: paredes, portas e janelas.

2) $_{[1{,}0\,pt]}$ Em C++, uma função virtual pura se caracteriza por:

a) Ter uma implementação *default*, podendo ser redefinida nas classes derivadas

b) Ter uma implementação *default*, devendo ser redefinida nas classes derivadas

c) Não ter implementação definida, permitindo que a classe não seja abstrata

d) Não permitir polimorfismo

e) Não ter implementação definida, permitindo que a classe seja abstrata

3) $_{[2{,}0\,pt]}$ Dado as versões de código abaixo Programa 1, Programa 2, Programa 3, assinale com V ou F as alternativas, considerando que o *main()* apresentado a seguir é comum aos três programas:

```
int main()
{
    Personagem* pP = NULL;
    Jogador j;
    j.setNome("Romulo");
    Inimigo i;
    i.setNome("Remulo");
    pP = &j;
    pP-> mover();
    pP = &i;
    pP->mover();

    return 0;
}
```

Programa 1

```
#include <string.h>
#include <iostream>
using namespace std;
class Entidade
{
protected:
    char nome[200];

public:
    Entidade() { strcpy(nome, ""); }
    ~Entidade() { }

    void setNome(const char n[]) { strcpy(nome, n); }
    virtual void mover()=0;
};

class Personagem : public Entidade
{
public:

    virtual void mover()
    {
        cout << nome << " é um personagem!" << endl;
    }
};
class Jogador : public Personagem
{
public:
    void mover()
    {
        cout << nome << " é um jogador!" << endl;
    }
};
class Inimigo : public Personagem
{
public:
    void mover()
    {
        cout << nome << " é um inimigo!" << endl;
    }
};
```

Programa 2

```
#include <string.h>
#include <iostream>
using namespace std;
class Entidade
{
protected:
    char nome[200];
public:
    Entidade() { strcpy(nome, ""); }
    ~Entidade() { }
    void setNome(const char n[]) { strcpy(nome, n); }
    virtual void mover()=0;
};
class Personagem : public Entidade
{
public:

    virtual void mover() = 0;
};
class Jogador : public Personagem
{
public:
    void mover()
    {
        cout << nome << " é um jogador!" << endl;
    }
};
class Inimigo : public Personagem
{
public:
    void mover()
    {
        cout << nome << " é um inimigo!" << endl;
    }

};
```

Programa 3

```
#include <string.h>
#include <iostream>
using namespace std;
class Entidade
{
protected:
    char nome[200];
public:
    Entidade() { strcpy(nome, ""); }
    ~Entidade() { }
    void setNome(const char n[]) { strcpy(nome, n); }
    virtual void mover()=0;
};

class Personagem : public Entidade
{
public:
    virtual void mover() = 0;
};
class Jogador : public Personagem
{
public:
    void mover()
    {
        cout << nome << " é um jogador!" << endl;
    }
};

class Inimigo : public Personagem
{
public:

    virtual void mover()
    {
        cout << nome << " é um inimigo!" << endl;
    }

};
```

a) ( ) Os Programas 1, 2 e 3, permitem executar o código da main()!

b) ( ) Apenas os Programas 1 e 2 permitem executar o código da main()!

c) ( ) Nenhum dos Programas apresentados permitem polimorfismo!

d) ( ) Todos os Programas permitem polimorfismo!

e) ( ) Não há classe abstrata e portanto há polimorfismo!

4) $_{[1,0\,pt]}$ Uma classe em orientação a objetos se caracteriza por:

a) Servir como modelo para um conjunto de instâncias

b) Servir como modelo para um conjunto unitário de instâncias

c) Servir unicamente como um repositório de funções

d) Servir unicamente como um repositório de atributos

e) Servir principalmente como base para derivação de outras classes

5) $_{[1,5\,pt]}$Um aluno digitou o código C++ a seguir:

Quadro 1	`------------------------------`
	`Sou instancia de Pai`
	`Segunda Impressao:Sou instancia de Pai`
	`------------------------------`
	`Sou instancia de Filha`
	`Segunda Impressao:Sou instancia de Filha`

Quadro 2	`------------------------------`
	`Sou instancia de Pai`
	`Segunda Impressao:Sou instancia de Pai`
	`------------------------------`
	`Sou instancia de Pai`
	`Segunda Impressao:Sou instancia de Filha`

Quadro 3	`------------------------------`
	`Sou instancia de Pai`
	`Segunda Impressao:Sou instancia de Pai`
	`------------------------------`
	`Sou instancia de Filha`
	`Segunda Impressao:Sou instancia de Pai`

Quadro 4	`char *c = NULL;` `q->toString(&c);`

Quadro 5	`Filha *f = new Filha();`
	`char *c = NULL;`
	`f->toString(&c);`
	`printf("\nvalor de c >> %s", c);`

Quadro 6	`Pai *z = new Filha();` `Filha *w = z;`

```
#include <string.h>
#include <stdio.h>
#include <iostream>
#include <string>
using namespace std;
////////////////////////////
/*
 * main.cpp
 *
 *  Created on: Dec 7, 2015
 *      Author: robinsonvidanoronha
 * Ajustes menores Simao 11/12/2015.
 */
#include <string.h>
#include <stdio.h>
#include <iostream>
#include <string>
using namespace std;
class Pai
{
  public :
          Pai(){}
   virtual ~Pai(){}
          virtual string* toString()
          {
             string* saida = NULL;
             saida = new string("Sou instancia de
Pai");
      return saida;
          }

          void toString(string *v)
          {
                  const string* t = toString();
                  v->append( *t );
          }
};
class Filha : public Pai
{
public :
          Filha(){}
   virtual ~Filha(){}
          virtual string* toString()
          {
     string* saida = NULL;
     saida = new string("Sou instancia de Filha");
     return saida;
          }
          void toString(string* v)
          {

                  const string* t = toString();
                  v->append( *t );
          }
          void toString(char **v)
          {
                  string *t =  toString();
                  int tam = (int)t->length();
                  char aux;
                  char* listaAux = NULL;
                  //listaAux = (char*) malloc(tam
* sizeof(char));
                  listaAux = new char[tam];

                  for (int i = 0; i < tam; i++)
                  {
                          aux = t->at(i);
                          listaAux[i] = aux;
                  }
                  *v = listaAux;
          }
};
void imprime(Pai *q)
{
          string t = *(q->toString());
          cout << "\n" << t << std::endl;
}
int main()
{
  Pai *p = new Pai();
  Pai *q = new Filha();
  cout << "\n";
  for (int i = 0; i < 30; i++)
  {
     cout << "-" ;
  }
  imprime(p);
  string t2;
  p->toString(&t2);
  cout << "Segunda Impressao:" << t2 << endl;
////////////////////////////////////////////////
  cout << "\n";
  for (int i = 0; i < 30; i++)
  {
     cout << "-" ;
  }

  imprime(q);

  string t3;
  q->toString(&t3);
  cout << "Segunda Impressao:" << t3 << endl;


  delete p;
  delete q;
  //free (c);
  return 0;
}
```

Analise esse código digitado pelo aluno e em seguida, responda V ou F para as seguintes afirmações.

a) ( ) A resultado da execução do programa imprimirá na tela o Quadro 1!

b) ( ) A resultado da execução do programa imprimirá na tela o Quadro 2!

c) ( ) A resultado da execução do programa imprimirá na tela o Quadro 3!

d) ( ) O termo overloading ou sobrecarga ou ad hoc polymorphism é utilizado para descrever situações onde um nome de função tem várias alternativas de implementação. Esse tipo de polimorfismo não ocorre no código digitado pelo aluno para a definição da classe Pai!

e) ( ) O termo *overloading* ou sobrecarga ou *ad hoc polymorphism* é utilizado para descrever situações onde um nome de função tem várias alternativas de implementação. Os métodos da classe Filha *virtual std::string *toString()*, *void toString(std::string *v)* e *void toString(char **v)* são exemplos da ocorrência desse tipo de polimorfismo!

f) ( ) O termo *overriding* ou sobrescrita ou *inclusion polymorphism* ocorre na relação de herança entre duas classes quando essas possuem funções com a mesma assinatura. Esse tipo de polimorfismo não ocorre no código digitado pelo aluno!

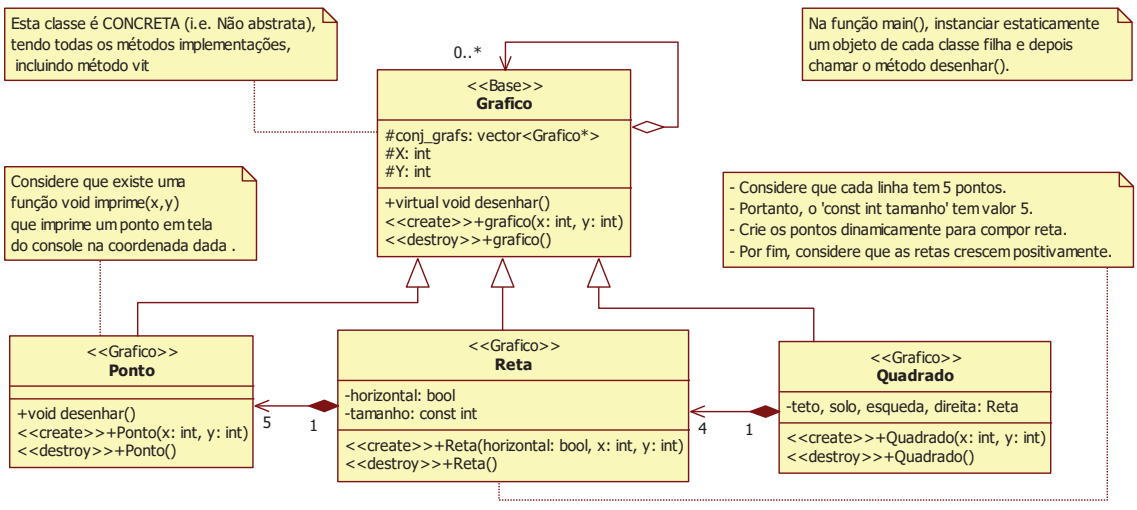
g) ( ) O termo *polymorphic variable* ou variável polimórfica ou *assignment polymorphism* é uma variável que é declara de um tipo mas que de fato recebe valores de tipos diferentes. Esse tipo de polimorfismo não ocorre no programa digitado pelo aluno!

h) ( ) Se o aluno inserir antes da instrução *delete p* do programa principal (*main*) as instruções definidas no Quadro 4, o compilador acusará erro de compilação!

i) ( ) Se o aluno inserir antes da instrução *delete p* do programa principal (*main*) as instruções do Quadro 5, o compilador não acusará erro algum!

j) ( ) As instruções contidas no Quadro 6, se inseridas no programa principal, irão ocasionar erros de compilação:

6) $_{[2{,}0\,pt]}$ De acordo com o diagrama apresentado, escreva um único código em C++ que polimorficamente permita que:



- Cada objeto da classe reta se componha de objetos da classe ponto.
- O código deve permitir que objetos da classe quadrado se componham de objetos da classe ponto.