Ficha 7

Programação (L.EIC009)

Objectivos

- Exercícios sobre herança entre classes em C++.

Recursos

Slides das aulas teóricas: HTML PDF

1

A classe `person` abaixo (parecida com a que vimos nas aulas teóricas) representa a informação de uma pessoa em uma faculdade (de forma muito simplificada!): existem atributos para o número de identificação e o nome de uma pessoa.

```
#include <iostream>
#include <string>

class person  {
private:
    int pid;
    std::string pname;
public:
    person(int id, const std::string& name)
      : pid(id), pname(name) { }
    int id() const {
      return pid;
    }
    const std::string& name() const {
      return pname;
    }
    virtual  void print(std::ostream& out) const {
      out << "ID: " << pid << std::endl
          << "Name: " << pname << std::endl;
    }
};
```

Defina classes `student` e `erasmus_student` atendendo a que:

- `student` seja subclasse de `person`, e represente informação para o nome do curso do aluno (um objecto `std::string`) acessível via função membro pública `course()`;
- `erasmus_student` seja subclasse de `student`, e represente informação para o país de origem do aluno (também um objecto `std::string`) acessível via função membro `country()`;
- `student` e `erasmus_student` devem redefinir convenientemente a função virtual `print()` para imprimir todos os atributos das respectivas classes, e re-aproveitando a funcionalidade de `print()` nas respectivas classe base.
- como boa prática, deve empregar a anotação `override` convenientemente.

Exemplo de uso das classes:

```
int main(void) {
    person p(123, "Matias Oliveira");
    student s(124, "Maria Oliveira", "LEIC");
    erasmus_student es(125, "John Zorn", "LEIC", "United States");
    p.print(std::cout);
    s.print(std::cout);
    es.print(std::cout);
    return 0;
}
```

Output:

```
ID: 123
Name: Matias Oliveira
ID: 124
Name: Maria Oliveira
Course: LEIC
ID: 125
Name: John Zorn
Course: LEIC
Course: United States
```

2

Explique o que está errado em cada um dos casos envolvendo a definição de uma classe base `A` e de uma subclasse `B` de `A`.

2.1

```
class A final {
public:
   A() { }
};
class B : public A {
public:
   B() { }
};
```

2.2

```
class A {
private:
  int x;
public:
   A() { x = 0; }
};
class B : public A {
public:
   B() {  }
   int f() { return x; }
};
```

2.3

```
class A {
private:
  int x;
public:
   A() {  x = 0; }
};
class B : public A {
public:
   B(int x) : A(x) { }
};
```

2.4

```
class A {
protected:
  int x;
public:
   A(int v) {  x = v; }
```

```
};
class B : public A {
public:
   B()  { x = 1; }
};
```

2.5

```
class A {
protected:
  int x;
public:
   A(int v) {  x = v; }
   virtual int f() { return x; }
};
class B : public A {
public:
   B(int v) : A(v) { }
   int f(int v) override { return x + v; }
};
```

2.6

```
class A {
protected:
  int x;
public:
   A() {  x = 0; }
   virtual int f() final { return x; }
};
class B : public A {
public:
   B() { x = 1; }
   int f() override { return x + 1; }
};
```

2.7

```
class A {
protected:
  int x;
public:
   A(int v) {  x = v; }
   virtual int f() = 0;
```

```
};
class B : public A {
public:
   B(int v) : A(v) { }
   int f() override { return x + v; }
   int g(int v) {
      A a(v + x);
      B b(v + x);
      return a.f() + b.f();
   }
};
```

3

3.1

Defina uma hierarquia de classes para sólidos 3D nos seguintes passos:

1. Defina uma classe base abstracta `solid`. Esta deverá definir um construtor sem argumentos e definir funções virtuais puras `double volume() const` e `double area() const`, cujo propósito (em abstracto!) será respectivamente devolver o volume e área da superfície de um sólido.

2. Defina uma classe `sphere` como subclasse de `solid` para representar uma esfera. A classe deve ter um construtor que tome como argumento o raio da esfera. (Relembra-se que uma esfera com raio `r` tem volume $\frac{4}{3}\pi r^3$ e área $4\pi r^2$; use a constante `M_PI` definida no header `cmath` para o valor de π.)

3. Defina uma classe `cuboid` para representar cubóides, também subclasse de `solid`. A classe deve ter um construtor que tome como argumentos os comprimentos de arestas `lx`, `ly` e `lz` do cubóide em cada eixo (e portanto o valor do volume é dado por `lx * ly * lz` e o valor da área é dado por `2 * (lx * ly + lx * lz + ly * lz)`).

4. Define uma classe `cube` como subclasse de `cuboid`. A classe deve ter um construtor que tome como o argumento o comprimento de aresta do cubo. Assinale a classe como `final` para que `cube` não admita subclasses.

3.2

Suponha que queremos obter para objectos `solid` o valor do rácio área-volume (SA:V). Acrescente a funcionalidade necessária à hierarquia de classes. Podemos modificar apenas uma classe para o efeito? Faz sentido de alguma forma aplicarmos o

modificar `final`?

3.3

Suponha que queremos obter o número de faces em um sólido quando o valor estiver definido (por exemplo para um cubóide ou cubo mas não para uma esfera). Acrescente a funcionalidade necessária à hierarquia de classes por forma a que por omissão seja retornado o valor `0` pela implementação base em `solid`.

3.4

Acrescente a `solid` uma função membro `void print() (std::ostream& out)` para imprimir os valores obtidos para as várias funções membro de `solid` para um conjunto de sólidos das alíneas anteriores.

Exemplo de uso:

```
sphere s(1.0);
cuboid c1(1.0, 2.0, 3.0);
cube c2(2.5);

s.print();
c1.print();
c2.print();
```

Exemplo possível de output para o exemplo:

```
area: 12.5664, volume: 4.18879, a/v: 3, faces: 0
area: 22, volume: 6, a/v: 3.66667, faces: 6
area: 37.5, volume: 15.625, a/v: 2.4, faces: 6
```

4

Tenha em conta as seguintes classes `A`, `B` e `C`.

```
class A {
protected:
   int v;
public:
   A(int x) : v(x) { }
   int f(int a) { return a + f(); }
```

```
    virtual int f() { return v; }
};
class B : public A {
public:
    B(int x) : A(x) { }
    int f(int a) { return a - v; }
    int f() override { return - v; }
    virtual int g(int a) { return v + a; }
};
class C : public B {
public:
    C(int x) : B(x) { }
    int g(int a) override { return v - a; }
};
```

Explique quais são as funções membro invocadas para classes `A`, `B` e `C` em cada um dos seguintes fragmentos, e o resultado obtido no final na variável `v`. Para validar se a sua análise está correcta, execute os fragmentos de código de seguida.

4.1

```
A a(1);
int v = a.f() + a.f(1);
```

4.2

```
C c(2);
int v = c.f() + c.f(1) + c.g(1);
```

4.3

```
A* a = new B(3);
int v = a -> f() + a -> f(1);
```

4.4

```
C c(2);
B& b = c;
int v = b.f() + b.f(1) + b.g(1);
```

4.5

```
B* b = new B(5);
int v = b -> f() + b -> f(1);
```

4.6

```
B* b = new B(6);
int v = b -> f() + b -> f(1) + b -> g(1);
```

4.7

```
B* b = new C(7);
int v = b -> f() + b -> f(1) + b -> g(1);
```