Herança entre classes em C++

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

Herança entre classes em C++

Tópicos

- Herança - noções base
  - classe base e subclasse
  - herança de funcionalidade de classe base para subclasse
  - visibilidade `protected`
- Redefinição de funções membro
  - redefinição e polimorfismo
  - os modificadores `virtual`, `override` e `final`
  - funções virtuais puras e classes abstractas
- Exemplo de consolidação e aspectos complementares
  - exemplo: hierarquia de classes para formas geométricas

Herança - noções base

Relação de herança entre classes

```
class SubClass : public BaseClass {
    ...
};
```

Em C++ podemos declarar uma classe como sendo **subclasse** (`Subclass` acima) de uma **classe base** (`BaseClass`). A subclasse também é chamada de "classe-filha" e classe base de "classe pai/mãe" ("parent class").

Motivações:

- `SubClass` herda (reutiliza) a funcionalidade de `BaseClass`. e ao mesmo tempo pode definir nova funcionalidade.
- (mais avançado) `SubClass` pode também redefinir / refinar a funcionalidade herdada, e `BaseClass` pode definir funcionalidade abstracta a implementar por subclasses.

Exemplo - classe base

```
class person {
private:
  int pid;
  std::string pname;
public:
  person(int id, const std::string& name);
  int id() const;
  const std::string& name() const;
  ...
};
```

Exemplo - subclasse

```
class teacher : public person {
private:
  std::string tdepartment;
public:
  teacher(int id, const std::string& name,
          const std:string& dept);
  const std::string& department() const;
  ...
};
```

`teacher` herda campos e funções membro de `person` (ex. `id()` e `name()`) e define nova funcionalidade (ex. `department()`).

A hierarquia de classes é extensível ...

Uma classe pode ter várias subclasses, e a hierarquia de classes pode ter vários níveis. Por exemplo poderíaamos ter:

```
// Classe base.
class person { ... };

// Subclasses directas de person
class teacher : public person { ... };
class student : public person { ... };

// Subclasses de student
class erasmus_student : public student { ... };
class working_student : public student { ... };
```

Focamo-nos no entanto apenas na definição simples da classe base `person` e da sua subclasse `teacher`.

Herança de campos e funções membro (cont.)

Sobre um objecto `teacher` podemos invocar a função membro `department()`. Mas como `teacher` é subclasse de `person` podemos também invocar `id()` e `name()`.

```
person p(123, "Joana Doa");
std::cout << p.id() << ' '
          << p.name() << std::endl;

teacher t(124, "John Doe", "Computer Science");
std::cout << t.id() << ' '
          << t.name() << ' '
          << t.department()
          << std::endl;
```

Output:

```
123 Joana Doa
124 John Doe Computer Science
```

Herança de campos e funções membro (cont.)

```
int person::id() const { return pid; }
const std::string&
person::name() const { return pname; }
...
const std::string&
teacher::department() const { return tdepartment; }
```

Implementação para `id()` e `name()` é herdada, não precisa de ser re-implementada. É necessário implementar `department()` em `teacher`.

Construtores de subclasse

```
person::person
(int id, const std::string& name)
  : pid(id), pname(name) {

}
...
teacher::teacher(int id,
 const std::string& name,
 const std::string& dept) :
   person(id, name), // chamada a construtor da classe base
   tdepartment(department) {

}
```

Na lista de inicialisação de membros de `teacher` temos a chamada ao construtor de `person` por forma a inicializar o estado do objecto no que toca à classe base. Se não houvesse tal chamada, o construtor por omissão da classe base seria invocado (neste caso não existe).

Visibilidade `protected`

Além de `public` e `private`, a visibilidade de declarações em C++ pode ser `protected`: nesse caso as declarações são acessíveis por subclasses além da própria classe. Código fora da classe ou subclasses continua a não poder aceder directamente às declarações tal como no caso de `private`.

Na variante de `person` abaixo, o código de `teacher` pode aceder directamente aos campos membro `pid` e `pname` de `person`.

```
class person {
protected:
  // Acessíveis na classe e subclasses.
  int pid;
  std::string pname;
...
};
class teacher : public person { ... };
```

Modificador de visibilidade na herança

Geralmente a herança entre classes é configurada com visibilidade pública, mas é possível (embora muito pouco comum) usar `private` ou `protected`.

```
class teacher : public person { ... };
class teacher : protected person { ... };
class teacher : private person { ... };
```

- `public`: não altera a visibilidade de definições `protected` ou `public` - mantêm-se na subclasse;
- `protected`: definições `public` ou `protected` na classe base passam a ter visibilidade `protected` na subclasse;
- `private`: definições `public` ou `protected` na classe base são `private` na subclasse.

Na cadeira faremos sempre uso de herança com visibilidade `public`.

Redefinição de funções membro, funções virtuais, classes abstractas

Exemplo

No código de `person` e `teacher` (ver código disponibilizado) temos a função membro `print()` definida em **ambas** as classes com a mesma assinatura.

```
class person {
    ...
    void print(std::ostream& out) const;
};
class teacher : public person {
    ...
    void print(std::ostream& out) const;
};
```

`print()` é portanto **redefinida** para objectos de tipo `teacher`.

Exemplo (cont.)

Apesar de ser uma redefinição de uma função membro da classe base `person`, `teacher::print()` pode invocar internamente `person::print()` por forma a reutilizar a sua funcionalidade.

```
void person::print(std::ostream &out) const {
    out << "ID: " << pid << std::endl
        << "Name: " << pname << std::endl;
}
void teacher::print(std::ostream &out) const {
    // Invocação de função na classe base
    person::print(out);
    // Código complementar
    out << "Department: " << tdepartment << std::endl;
}
```

Exemplo (cont.)

Em linha com o código de `teacher.print()` o fragmento definido por

```
teacher t(124, "John Doe", "Computer Science");
t.print(std::cout);
```

produzirá o seguinte output

```
ID: 124
Name: John Doe
Department: Computer Science
```

Exemplo (cont.)

No entanto, para código "algo similar" ...

```
teacher t(124, "John Doe", "Computer Science");
person& rt = t;
person* pt = &t;
rt.print(std::cout);
pt->print(std::cout);
```

obtemos em vez disso ...

```
ID: 124
Name: John Doe
ID: 124
Name: John Doe
```

É invocada a função `person::print()` apesar de o objecto `t` referenciado por `rt` e `pt` ser do tipo `teacher`!

Porquê?

```
teacher t(124, "John Doe", "Computer Science");
person& rt = t;
person* pt = &t;
rt.print(std::cout);
pt->print(std::cout);
```

Estão em causa dois aspectos:

- `teacher` é subclasse de `person` portanto referências ou apontadores do tipo `person` como `rt` e `pt` podem-se referir a um objecto `teacher`.
- A função invocada, tendo em conta a definição de `print()` nas duas classes tal como apresentado até agora, é a definida em `person` e não `teacher`: `rt` e `pt` têm tipo `person`, apesar de o objecto ser do tipo `teacher` ...

Uso de `virtual`

Declarando `print()` como `virtual` – **função virtual** - em `person`, então **a função membro a invocar é determinada dinamicamente em tempo de execução de acordo com o tipo concreto do objecto referenciado**:

```
class person {
   ...
   virtual void print(std::ostream& out) const;
};
class teacher : public person {
   ...
   void print(std::ostream& out) const;
};
```

Uso de `virtual` (cont.)

Revisitando o código anterior, mas com `print()` declarada como `virtual`

```
teacher t(124, "John Doe", "Computer Science");
person& rt = t;
person* pt = &t;
rt.print(std::cout);
pt->print(std::cout);
```

então `rt.print()` e `pt->print()` resultam numa chamada a `teacher::print()`:

```
ID: 124
Name: John Doe
Department: Computer Science
```

Uso de `virtual` (cont.)

Sumário:

- Uma classe pode declarar funções `virtual` para estabelecer interface comum a várias subclasses via referências ou apontadores **polimórficos**: objectos referenciados pelo tipo da classe base podem ter vários tipos concretos distintos em tempo de execução.
- Para funções `virtual` a função membro é determinada dinamicamente em tempo de de acordo com o tipo concreto do objecto - **mecanismo de ligação dinâmica**.
- A classe pode prover uma implementação base, que pode no entanto ser redefinida em subclasses. Em alternativa, podemos ter **funções virtuais puras** definindo uma **classe abstracta** (a cobrir mais à frente nestes slides).

Uso de `override`

Boa prática: se `person::print()` é virtual, `teacher::print()` deve ser anotado com o modificador **override**.

```
class person {
   ...
   virtual void print(std::ostream& out) const;
};
class teacher : public person {
   ...
   void print(std::ostream& out) const override;
};
```

A `override` serve para assinalar que determinada função é a redefinição de uma função `virutal`. O compilador valida nesse caso que de facto de trata de uma redefinição.

Uso de `override` (cont.)

Exemplo de redefinição errada:

```
class person {
    ...
    virtual void print(std::ostream& out) const;
};
class teacher : public person {
    ...
    // int em vez de void para o tipo de retorno!
    int print(std::ostream& out) const override;
};
```

Erro de compilação, que não é reportado na ausência de `override`

```
error: virtual function 'print' has a different return type ('
than the function it overrides (which has return type 'void')
int print(std::ostream& out) const override;`
```

Uso de `final`

Uma função virtual pode ser assinalada como `final` para impedir a sua redefinição.

```
class person {
  ...
  virtual void print(std::ostream& out) const final;
};
class teacher : public person {
  ...
  // Redefinição não permitida!
  void print(std::ostream& out) const override;
};
```

Erro de compilação:

```
error: declaration of 'print' overrides a 'final' function
```

Uso de `final` (cont.)

O modificador `final` pode também ser associado a uma classe para impedir a definição de subclasses desta.

```
class person final { ... };
// Não podemos ter subclasses de person neste caso.
class teacher : public person { ... };
```

Erro de compilação:

```
error: base 'person' is marked 'final'
```

`virtual`, `override` e `final` - sumário

`virtual` → Assinala que função é virtual - chamada à função usará mecanismo de ligação dinâmica tendo em conta o tipo concreto do objecto em tempo de execução.

`override` → modificador (opcional) que indica ao compilador que se trata de uma re-definição de uma função virtual. Compilador verifica que re-definição é válida.

`final` → se aplicado a uma função: inibe a redefinição de uma função; se aplicado a uma classe: inibe a definição de subclasses para a classe.

Funções virtuais puras e classes abstractas

Uma **função virtual pura** consiste numa declaração do tipo `virtual ... func(...) = 0` ex.

```
class person {
  ...
  virtual void print(std::ostream& out) const = 0;
};
```

Neste caso:

- Função em causa **não tem implementação**, funcionando apenas como interface genérico a subclasses.
- A classe diz-se **abstracta** já que não pode ser instanciada directamente, apenas via subclasses.

Funções virtuais puras e classes abstractas (cont.)

Se tivermos:

```
class person {
  ...
  virtual void print(std::ostream& out) const = 0;
};
```

então não podemos instanciar directamente `person`. Para

```
person p(123, "Joana Doa");
```

iremos ter o seguinte erro de compilação:

```
error: variable type 'person' is an abstract class
```

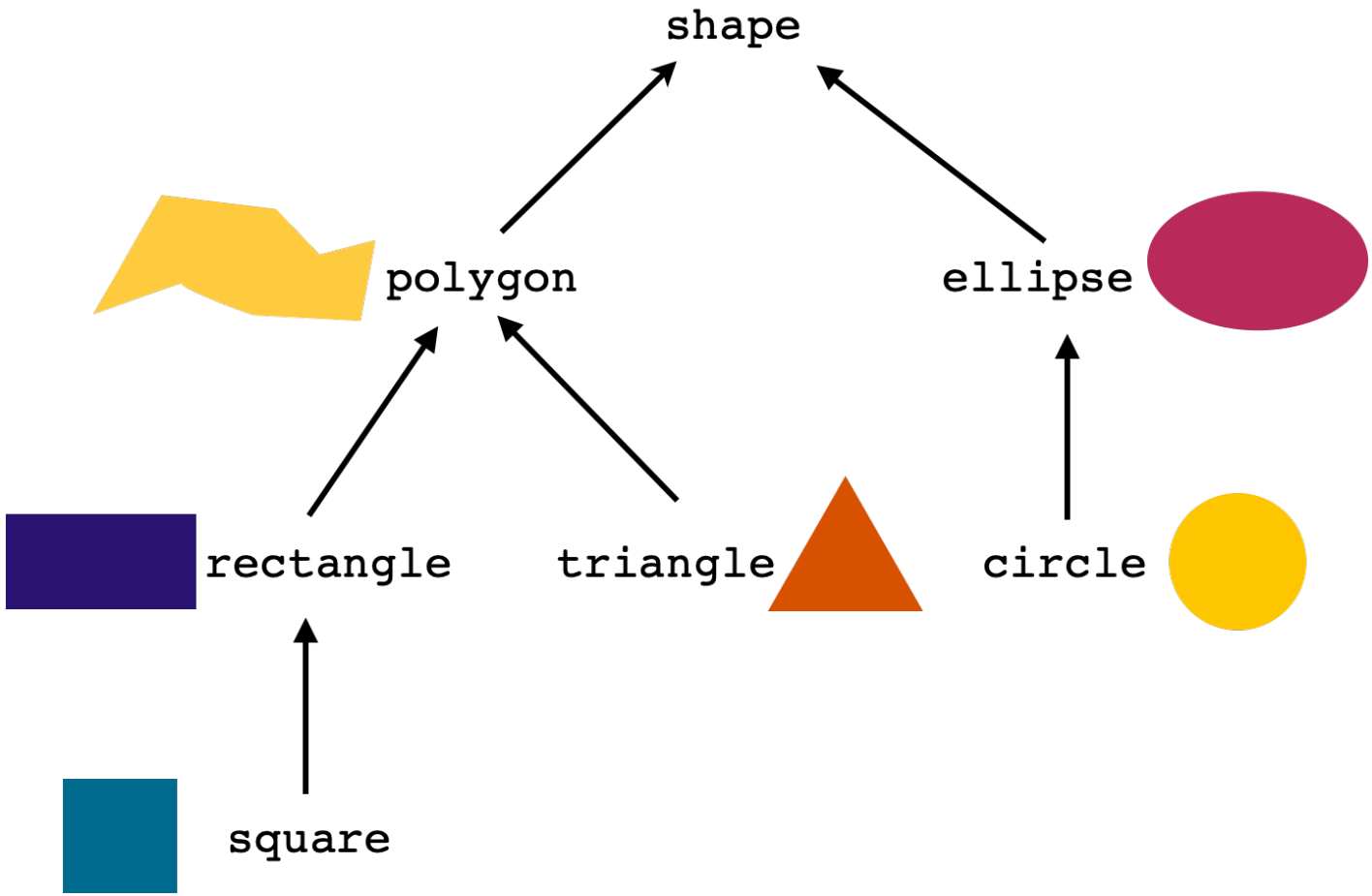
No entanto, como `teacher` implementa `print`, podemos ter como antes:

```
teacher t(124, "John Doe", "Computer Science");
```

Exemplo de consolidação, aspectos complementares

Exemplo

Hierarquia de classes para formas geométricas:



Exemplo: `shape` é uma classe abstracta

```
class shape {
public:
    virtual double area() const = 0;
    virtual coord2d center() const = 0;
    virtual void move(const coord2d& direction) = 0;
    virtual ~shape() { }
};
```

`shape` é uma classe abstracta. Todas as funções membro são virtuais puras (nota: bastaria que apenas uma função membro fosse virtual pura para a classe ser abstracta).

Destrutor virtual

```
class shape {
public:
    ...
    virtual ~shape() { }
};
```

Boa prática: classe base define destrutor virtual. C++ não garante correcta invocação de destrutor de subclasses de outro modo quando usamos memória alocada dinamicamente referenciada com o tipo da classe base, ex:

```
shape* s =  new polygon ( ...);
...
// invoca destrutor de polygon
// se destrutor de shape for virtual
delete s;
```

Uma subclasse de shape

```
class ellipse : public shape {
private:
    coord2d ecenter;
    double erx;
    double ery;
public:
    ellipse(const coord2d& c, double rx, double ry)  :
            ecenter(c), erx(rx), ery(ry) { }
    double radius_x() const { return erx; }
    double radius_y() const { return ery; }
    double area() const override final  {
      { return  M_PI * erx * ery; }
    coord2d center() const override final { return ecenter; }
    void move(const coord2d& movement) override final
      {  ecenter += movement; }
};
```

Uma subclasse de `shape` (cont.)

```
class ellipse : public shape {
...
public:
    ...
    double area() const override final  {
      { return  M_PI * erx * ery; }
    coord2d center() const override final { return ecenter; }
    void move(const coord2d& movement) override final
      {  ecenter += movement; }
};
```

`ellipse` define de forma concreta funções membro virtuais puras `area()`, `center()` e `move()`. Estas funções membro são `final` impedindo a sua redefinição por sua vez em subclasses de `ellipse` (como `circle` a seguir).

Hierarquia de classes - mais um nível ...

```
class circle final : public ellipse {
public:
    circle(const coord2d& c, double r) : ellipse(c, r, r) { }
};
```

Definição de `circle` como subclasse de `ellipse` é trivial.

Classe `circle` é `final` - não podemos ter subclasses de `circle`.

Outras classes na hierarquia ...

Analogamente, é definida uma classe base para polígonos e subclasses correspondentes a alguns polígonos comuns.

`polygon` é a classe base para polígonos e define `area()`, `center()` e `move()` (ver código disponibilizado). Estas funções não precisam de ser redefinidas para subclasses de `polygon`.

```
class polygon : public shape { ... }
class triangle final : public polygon { ... };
class rectangle : public polygon { ... };
class square final : public rectangle { ... };
```

Classe drawing

Um desenho (drawing) agrupa várias formas:

```
class drawing final {
private:
    std::vector<shape*> shapes;
public:
    drawing() { }
    ~drawing() { ... }
    void add_shape(shape* s) { ... }
    void move_all(const coord2d& movement) { ... }
    std::vector<shape*>& get_shapes() { ... }
};
```

Classe `drawing` (cont.)

Por forma a agrupar as formas, `drawing` usa um vector de apontadores para `shape`. Não podemos ter `vector<shape>` porque `shape` é abstracta. E se `shape` não fosse abstracta, com `vector<shape>` não poderíamos guardar objectos que fossem subclasses de `shape`.

```
class drawing final {
private:
    std::vector<shape*> shapes;
public:
    ...
    void add_shape(shape* s) {
        shapes.push_back(s);
    }
    ...
};
```

Classe `drawing` (cont.)

```
class drawing final {
private:
    std::vector<shape*> shapes;
public:
    ...
    void move_all(const coord2d& movement) {
        for (shape* s : shapes) {
            s -> move(movement);
        }
    }
    ...
};
```

A funcionalidade abstracta da classe `shape` permite-nos manipular as formas no desenho.

Classe `drawing` (cont.)

Assume-se que objectos são dinamicamente alocados com `new` externamente à classe e que depois a memória dinâmica é libertada por `drawing` no destrutor via `delete` para cada forma.

Se destrutor de `shape` não fosse virtual (como discutido antes), os destrutores dos objectos (definidos em subclasses de `shape`) poderiam não ser invocados correctamente.

```
class drawing final {
private:
    std::vector<shape*> shapes;
public:
    ...
    ~drawing() {
        for (shape* s : shapes) {
            delete s;
        }
    }
    ...
```

Herança vs. composição

A relação de herança deve reflectir uma relação de especialização / *"is a"* (ex. "um professor é um tipo de pessoa em uma faculdade") Um erro de desenho comum é usar herança de classes para modelar uma relação *"has a"*. por exemplo:

```
//  MAU DESENHO (embora código seja válido)
class drawing : public std::vector<shape*> { ... };
```

Um desenho **não é** um vector de formas. Um desenho **tem** ("has a") / usa um vector de formas, e nesse caso a relação adequada é a de **composição**, i.e., a classe `drawing` tem na sua composição um vector de formas como apresentamos:

```
class drawing {
private:
  std::vector<shape*> shapes;
  ...
};
```