Tampereen yliopisto
Tampere University

Dynamic Binding and Virtual Functions

25.9.2018

Virtual functions

- Subclass has an *implementation* of its own for a service given in the base class
- Subclass inherits the *interface*, not the implementation
- Enabled in C++, if the member function is *virtual*
  - keyword: **`virtual`**

Virtual functions

- Choices in a subclass:
  - Accept the implementation given in the base class
  - Write an own implementation (often calls the implementation of the base class)
  - Parameters and the type of the return value **cannot** be changed

Tampereen yliopisto
Tampere University

Dynamic binding

- Virtual functions → it is possible that the interface of a member function has a level, different from that of the implementation
- Concluding the implementation to be called can be impossible at compile time
- Function to be called is bound (selected) at run time (dynamically)

Tampereen yliopisto
Tampere University

Dynamic binding

- Decision, which implementation to call, is made at *run time*
- Pointers/references:
  - Pointer may point either to an object of the base class or that of a subclass
  - Implementation to be called depends on the class of an object

    → same call, different implementation based on the object

Addition to class Book

```
class Book
{
  public:
    virtual void printData(std::ostream& stream) const;
    virtual bool keywordMatches(std::string const& word) const;
  private:
    void printError (std::string const& errorText) const;
};
```

Tampereen yliopisto
Tampere University

Addition to class Book

```
void Book::printError(string const& errorText) const {
    cerr << “Error: " << errorText << endl;
    cerr << “in book: ";
    printData(cerr);
    cerr << endl;
}
void Book::printData(ostream& stream) const {
    stream << author_ << " : \"" << title_ << "\"";
}
bool Book::keywordMatches(string const& word) const {
    return title_.find(word) != string::npos || author_.find(word)
!= string::npos;
}
```

Addition to class LibraryBook

```
class LibraryBook : public Book
{
    // ...
    virtual void printData(std::ostream& stream) const;
};

void LibraryBook::printData(ostream& stream) const
{
    Book::printData(stream);
    stream << ", return " << retDay_;
}
```

Dynamic binding

```
void printBooks(vector<Book*> const& books)
{
    for (unsigned int i = 0; i != books.size(); ++i)
    {
        books[i]->printData(cout);
        cout << endl;
    }
}
```

Dynamic binding

```
int main() {
    vector<Book*> bookShelf;
    bookShelf.push_back( new Book("Axiomatic", "Greg Egan"));
    bookShelf.push_back( new LibraryBook("Matemaattisia olioita",
                                         "Leena Krohn",
                                         Date(31,10,1999)));

    printBooks(bookShelf);
    for (unsigned int i = 0; i != bookShelf.size(); ++i) {
        delete bookShelf[i];
        bookShelf[i] = 0;
    }
}
```

Tampereen yliopisto
Tampere University

Terms

- Virtual function
  - function to be bound dynamically
- Dynamic (=run-time) binding
  - function to be called is chosen on the basis of the object’s current class
  - enables polymorphism
- Polymorphism
  - in O-O: base class instance can be replaced with a subclass instance

Run-time type check of objects

- **RTTI** (*Run-Time Type Identification*) added to ISO C++
- Requires at least one virtual function in a class

Run-time type check of objects

- Subclass object pointed by a base class pointer:
  - Access only to the base class interface
  - (Should be) sufficient in normal cases
- Need to access subclass interface → type cast

Run-time type check of objects

- Type cast:
  - Reasonable only if the object is of type in question → can be failed
  - **dynamic_cast**<Subclass*>(basePointer)
  - If the object is not of the right type → returns 0
- If possible, avoid type casts!

Run-time type check of objects

```
bool lateIsIt(Book* bp, Date const& today)
{
    LibraryBook* lbp = dynamic_cast<LibraryBook*>(bp);
    if (lbp != 0)
    { // If here, then the book is a library book
        return lbp->isLate(today);
    }
    else
    { // If here, then the book is not a library book
        return false; // Therefore the is not late
    }
}
```

Tampereen yliopisto
Tampere University

Finding out the class of an object

- **`dynamic_cast`** tests, if the object belongs to a *certain* class (or to its subclass)

→ It cannot find out, *to which* class the object belongs

- For this purpose C++ has operator **`typeid`** and class **`type_info`**
  - Usage: `#include <typeinfo>`

Finding out the class of an object

- Objects of class **`type_info`**
  - “Represent” a certain class (each of them)
  - Results from expressions:
    `typeid``(object)` and **`typeid`**`(aClass)`
  - Comparison operators == and `!=`
  - The name of a class can be found out with member function **`name`**

Finding out the class of an object

- `typeid` *tests a thing different* from `dynamic_cast`

```
if( typeid(*bp) == typeid(LibraryBook))…
if( dynamic_cast<*LibraryBook>(bp) != 0 ) …
```

Non-virtual function and hiding

- Virtual functions require run-time check (binding) that is not needed in other functions
- Subclass may have a member function with the same name as a *non-virtual* member function of the base class
  - Subclass implementation *hides* the function given in the base class
  - No *dynamic binding*
  - The way of calling determines which function is really called

Non-virtual function and hiding

- To avoid errors, subclass should give new implementations only for *virtual functions*
- Note that virtual property cannot be added in the subclass

→ Remember to declare as virtual *all* such functions of the base class that might be redefined in subclasses

Virtual destructors

- Base class pointer pointing to an object created with **new**
- Problem: how to delete the object without knowing its class (type)?
- Destruction actions are determined at run-time
- This requires destructor to be *virtual in the base class*
- Non-virtual destructor in the base class → functionality undefined

Tampereen yliopisto
Tampere University

Cost of virtual functions

- Run-time check → cost

1. Checks make programs slower:
   - Small effect to the total execution time
   - Not important, if run-time check is unavoidable

Tampereen yliopisto
Tampere University

Cost of virtual functions

2. Type information of objects consumes memory:
   - Typically a pointer (4 bytes) per object
   - Independent on the number of virtual functions
   - In addition some memory is needed for each class

- Compiler has the right to optimize memory consumption and execution time

Virtual functions in constructors and destructors

- The execution order of constructors goes from the base class to the subclasses
- Subclass parts are not yet ready when executing the constructor of the base class

Tampereen yliopisto
Tampere University

Virtual functions in constructors and destructors

→ Object is “not yet an object of the subclass”

→ Object behaves as an object of the base class

→ Dynamic binding cannot use the implementations of subclasses

→ ***Avoid calling virtual functions in constructors!***

- The same holds for destructors

Abstract base classes

- Meant to be used *only* as a base class
- Cannot be instantiated
- Typically includes interface function with no (adequate) implementation

Tampereen yliopisto
Tampere University

Abstract base classes

- Pure virtual function
  - Implementation *must* be given in subclasses
  - Base class usually gives no implementation
  - In class definition, function declaration added with `=0`
- Class is abstract, until all pure virtual functions have an implementation

Tampereen yliopisto
Tampere University

Pure virtual functions

```
class Animal : public Organism {
public:
    virtual ~Animal();
    virtual void move(Location destination) = 0;
};

class Bird : public Animal {
public:
    virtual ~Bird();
    virtual void sing() = 0;
};
```

Pure virtual functions

```
class Hen : public Bird
{
public:
    virtual ~Hen();
    virtual void reproduce(); // Implementation for reproducing
    virtual void move(Location destination); // Implementation
for moving
    virtual void sing(); // Implementation for singing

private:
    // Put here necessary private features
};
```

Pure virtual function with implementation

```
class Animal : public Organism {
public:
    virtual ~Animal();
    virtual void move(Location destination) = 0;
private:
    Location place_;
};
```

Pure virtual function with implementation

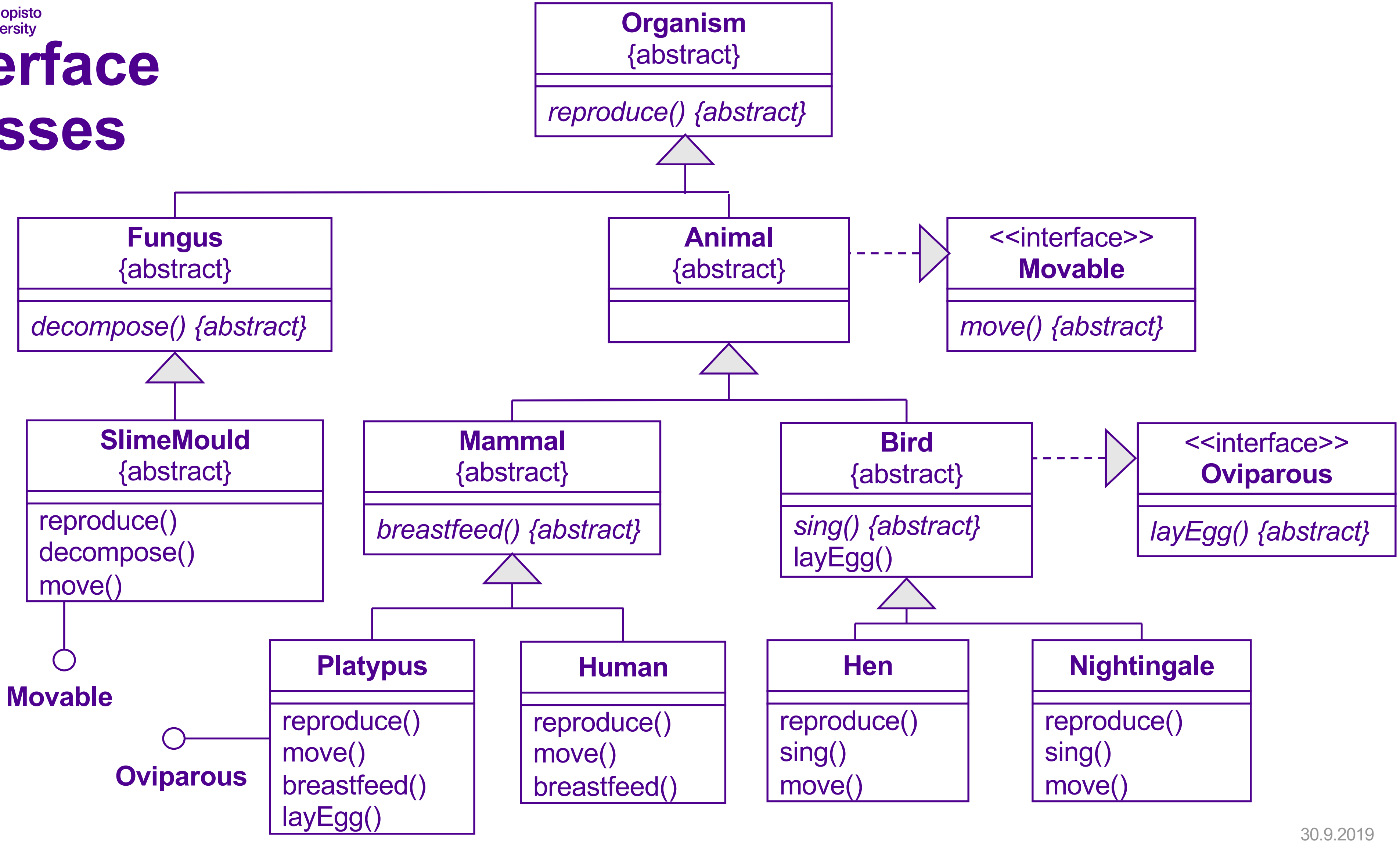
```
void Animal::move(Location destination)
{
    place_ = destination;
}

void Hen::move(Location destination)
{
    // Write here move actions for hen, walking etc.
    Animal::move(destination); // Base class implements common
movement
}
```

Inheritance and interface classes

- Base class including only the definition of an interface → *interface class*
- E.g. Java has interfaces separated from classes (different syntax)
- Problem in class hierarchy: interfaces are independent of each other and concrete classes may have different combinations of the interfaces → concept for *separate interfaces*

Interface classes



C++: abstract base classes and multiple inheritance

```
class Movable {
public:
    virtual ~Movable();
    virtual void move(Location destination) = 0;
};
class Oviparous
{
public:
    virtual ~Oviparous();
    virtual void layEgg() = 0;
};
```

C++: abstract base classes and multiple inheritance

```
class Animal : public Organism,
               public Movable
{
  public:
    virtual ~Animal();
  private:
};
```

```
class Platypus : public Mammal,
public Oviparous
{
  public:
    virtual ~Platypus();
    virtual void reproduce();
    virtual void move (Location
                       destination);
    virtual void breastfeed();
    virtual void layEgg();
};
```