Object Oriented Programming

In C++: Polymorphism

Lilian Aveneau / Hakim Belhaouari
hakim.belhaouari@univ-poitiers.fr

[1]University of Poitiers
Computer Science Department

[2]University Science and Technology of Ha Noi
USTH

2017-2018

Summary

Table of Contents

Table of Contents

Upcasting is dangerous

```cpp
// DEMO?
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat };

class Instrument {
public:
    void play(note) const {
        cout << "Instrument::play" << endl;
    }
};

/// Wind objects are Instruments
/// because they have the same interface:
class Wind : public Instrument {
public:
    /// Redefine interface function:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
};

void tune(Instrument& i) {
    /// ...
    i.play(middleC);
}

int main() {
    Wind flute;
    tune(flute); // Upcasting
}
```

Upcasting

- Correct: `flute` encompasses an `Instrument` instance
- No explicit conversion

Upcasting is dangerous

```cpp
// DEMO?
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat };

class Instrument {
public:
    void play(note) const {
        cout << "Instrument::play" << endl;
    }
};

/// Wind objects are Instruments
/// because they have the same interface:
class Wind : public Instrument {
public:
    /// Redefine interface function:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
};

void tune(Instrument& i) {
    /// ...
    i.play(middleC);
}

int main() {
    Wind flute;
    tune(flute); // Upcasting
}
```

Upcasting

- Correct: `flute` encompasses an `Instrument` instance
- No explicit conversion

Problem

- `play()` is called on `Instrument`
- Function call: *early binding*
- How developer wants the function on `Wind`?

Upcasting is dangerous

```cpp
// DEMO?
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat };

class Instrument {
public:
    void play(note) const {
        cout << "Instrument::play" << endl;
    }
};

/// Wind objects are Instruments
/// because they have the same interface:
class Wind : public Instrument {
public:
    /// Redefine interface function:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
};

void tune(Instrument& i) {
    /// ...
    i.play(middleC);
}

int main() {
    Wind flute;
    tune(flute); // Upcasting
}
```

Upcasting

- Correct: `flute` encompasses an `Instrument` instance
- No explicit conversion

Problem

- `play()` is called on `Instrument`
- Function call: *early binding*
- How developer wants the function on `Wind`?

Solution: the polymorphism, or *late binding* (or *dynamic binding* or *runtime binding*)

Virtual functions

Rule : *late binding occurs only with virtual functions*

Creation

- Keyword `virtual`
- Use inside a root(!) class
- Useless in derived classes (redundant, confusion risk)

Virtual functions

Rule : *late binding occurs only with virtual functions*

Creation

- Keyword `virtual`
- Use inside a root(!) class
- Useless in derived classes (redundant, confusion risk)

Here : add the keyword `virtual` before the function declaration `play()` in the class `Instrument`

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

enum note { middleC, Csharp, Cflat };

class Instrument {
public: // virtual function!
    virtual void play(note) const {
        cout << "Instrument::play" << endl;
    }
};

class Wind : public Instrument {
public:
    /// Override interface function:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
};

void tune(Instrument& i) {
    /// ...
    i.play(middleC);
}

int main() {
    Wind flute;
    tune(flute); // Upcasting
}
```

Virtual functions

Rule : *late binding occurs only with virtual functions*

Creation

- Keyword `virtual`
- Use inside a root(!) class
- Useless in derived classes (redundant, confusion risk)

Here : add the keyword `virtual` before the function declaration `play()` in the class `Instrument`

Result

Display `Wind::play`

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

enum note { middleC, Csharp, Cflat };

class Instrument {
public: // virtual function!
    virtual void play(note) const {
        cout << "Instrument::play" << endl;
    }
};

class Wind : public Instrument {
public:
    /// Override interface function:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
};

void tune(Instrument& i) {
    /// ...
    i.play(middleC);
}

int main() {
    Wind flute;
    tune(flute); // Upcasting
}
```

Extendible: foundation in correct architecture

```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat };

class Instrument {
public:
    virtual void play(note) const {
        cout << "Instrument::play" << endl;
    }
    virtual char* what() const {
        return "Instrument";
    }
    /// Assume this will modify the object:
    virtual void adjust(int) {}
};

class Wind : public Instrument {
public:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
    char*what() const { return "Wind"; }
    void adjust(int) {}
};

class Percussion : public Instrument {
public:
    void play(note) const {
        cout << "Percussion::play" << endl;
    }
    char*what() const { return "Percussion"; }
    void adjust(int) {}
};
```

```
// Derived from Wind ...
class Brass : public Wind {
public:
    void play(note) const {
        cout << "Brass::play" << endl;
    }
    char* what() const { return "Brass"; }
}; // adjust comes from Wind (closest def.)

/// Identical function from before:
void tune(Instrument& i) {
    /// ...
    i.play(middleC);
}

/// New function:
void f(Instrument& i) { i.adjust(1); }

// Upcasting during array initialization:
Instrument* A[] = {
    new Wind, new Percussion, new Brass
};

int main() {
    Wind flute;
    Percussion drum;
    Brass flugelhorn;
    tune( flute );
    tune( drum );
    tune( flugelhorn );
    f( flugelhorn ); // OK, Wind::adjust(1) is
                     // called
}
```

How does it work?

```
#include <iostream>
using namespace std;
// test classes
class NoVirtual {
int a;
public:
   void x() const {}
   int i() const { return 1; }
};
class OneVirtual {
int a;
public:
   virtual void x() const {}
   int i() const { return 1; }
};
class TwoVirtuals {
int a;
public:
   virtual void x() const {}
   virtual int i() const { return 1; }
};
// shows the classes' size
int main() {
   cout << "int: " << sizeof(int) << endl;
   cout << "NoVirtual: "
      << sizeof( NoVirtual ) << endl;
   cout << "void* : " << sizeof(void*)
     << endl;
   cout << "OneVirtual: "
     << sizeof( OneVirtual ) << endl;
   cout << "TwoVirtuals: "
     << sizeof( TwoVirtuals ) << endl;
}
```

How does it work?

```cpp
#include <iostream>
using namespace std;
// test classes
class NoVirtual {
int a;
public:
    void x() const {}
    int i() const { return 1; }
};
class OneVirtual {
int a;
public:
    virtual void x() const {}
    int i() const { return 1; }
};
class TwoVirtuals {
int a;
public:
    virtual void x() const {}
    virtual int i() const { return 1; }
};
// shows the classes' size
int main() {
    cout << "int: " << sizeof(int) << endl;
    cout << "NoVirtual: "
        << sizeof( NoVirtual ) << endl;
    cout << "void* : " << sizeof(void*)
       << endl;
    cout << "OneVirtual: "
       << sizeof( OneVirtual ) << endl;
    cout << "TwoVirtuals: "
       << sizeof( TwoVirtuals ) << endl;
}
```

Results (64 bits)

```
int: 4
NoVirtual: 4
void* : 8
OneVirtual: 16
TwoVirtuals: 16
```

How does it work?

```cpp
#include <iostream>
using namespace std;
// test classes
class NoVirtual {
int a;
public:
    void x() const {}
    int i() const { return 1; }
};
class OneVirtual {
int a;
public:
    virtual void x() const {}
    int i() const { return 1; }
};
class TwoVirtuals {
int a;
public:
    virtual void x() const {}
    virtual int i() const { return 1; }
};
// shows the classes' size
int main() {
    cout << "int: " << sizeof(int) << endl;
    cout << "NoVirtual: "
       << sizeof( NoVirtual ) << endl;
    cout << "void* : " << sizeof(void*)
      << endl;
    cout << "OneVirtual: "
      << sizeof( OneVirtual ) << endl;
    cout << "TwoVirtuals: "
      << sizeof( TwoVirtuals ) << endl;
}
```

Results (64 bits)

```
int: 4
NoVirtual: 4
void* : 8
OneVirtual: 16
TwoVirtuals: 16
```

Functioning

Compiler adds :

- Pointer per class : `VPTR`
- Table per class : `VTABLE`

How does it work?

```
#include <iostream>
using namespace std;
// test classes
class NoVirtual {
int a;
public:
   void x() const {}
   int i() const { return 1; }
};
class OneVirtual {
int a;
public:
   virtual void x() const {}
   int i() const { return 1; }
};
class TwoVirtuals {
int a;
public:
   virtual void x() const {}
   virtual int i() const { return 1; }
};
// shows the classes' size
int main() {
   cout << "int: " << sizeof(int) << endl;
   cout << "NoVirtual: "
      << sizeof( NoVirtual ) << endl;
   cout << "void* : " << sizeof(void*)
     << endl;
   cout << "OneVirtual: "
     << sizeof( OneVirtual ) << endl;
   cout << "TwoVirtuals: "
     << sizeof( TwoVirtuals ) << endl;
}
```

Results (64 bits)

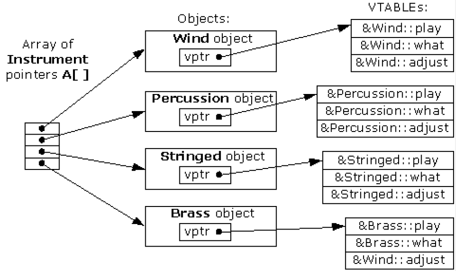
```
int: 4
NoVirtual: 4
void* : 8
OneVirtual: 16
TwoVirtuals: 16
```

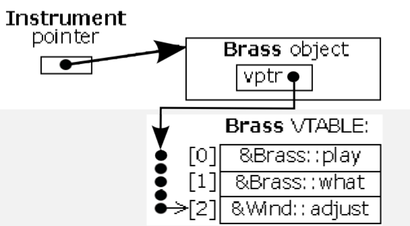
Functioning

Compiler adds :

- Pointer per class : `VPTR`
- Table per class : `VTABLE`
- NB : the size is still not null in spite of absence of members !

Inside virtual functions





Example: ASM of `i.adjust(1)`

```
push   1
push   si
mov    bx, word ptr [si]
call   word ptr [bx+4]
add    sp, 4
```

Explications :

- Push 1
- Push Source Index (this)
- Load VPTR (depuis SI)
- Call function at adr BX+4 (take third position `short pointers`)
- Handle stack (+4)

VPTR and Bindings

VPTR Installation

- Done by compiler ...
- Where exactly ? in constructor !
- → explain why constructor/default are needed

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

class Pet {
public:
    virtual string speak() const {
        return "";
    }
};

class Dog : public Pet {
public:
    string speak() const {
        return "Wouaf!";
    }
};

int main() {
    Dog theo;
    Pet* p1 = &theo;
    Pet& p2 = theo;
    Pet p3;
    // Late binding for both:
    cout << "p1->speak() = " << p1->speak()
        << endl;
    cout << "p2.speak() = " << p2.speak()
        << endl;
    // Early binding (probably):
    cout << "p3.speak() = " << p3.speak()
        << endl;
}
```

VPTR and Bindings

VPTR Installation

- Done by compiler ...
- Where exactly ? in constructor !
- → explain why constructor/default are needed

Not always dynamic

- Useful for pointer or reference
- Not for object ...
- Compiler *may* choose *early binding*

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

class Pet {
public:
    virtual string speak() const {
        return "";
    }
};

class Dog : public Pet {
public:
    string speak() const {
        return "Wouaf!";
    }
};

int main() {
    Dog theo;
    Pet* p1 = &theo;
    Pet& p2 = theo;
    Pet p3;
    // Late binding for both:
    cout << "p1->speak() = " << p1->speak()
        << endl;
    cout << "p2.speak() = " << p2.speak()
        << endl;
    // Early binding (probably):
    cout << "p3.speak() = " << p3.speak()
        << endl;
}
```

VPTR and Bindings

VPTR Installation

- Done by compiler ...
- Where exactly ? in constructor !
- → explain why constructor/default are needed

Not always dynamic

- Useful for pointer or reference
- Not for object ...
- Compiler *may* choose *early binding*

Philosophy? : why all these choices?

- C++ extends C: performance
- Additional cost at each virtual method call /classics calls

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

class Pet {
public:
    virtual string speak() const {
        return "";
    }
};

class Dog : public Pet {
public:
    string speak() const {
        return "Wouaf!";
    }
};

int main() {
    Dog theo;
    Pet* p1 = &theo;
    Pet& p2 = theo;
    Pet p3;
    // Late binding for both:
    cout << "p1->speak() = " << p1->speak()
        << endl;
    cout << "p2.speak() = " << p2.speak()
        << endl;
    // Early binding (probably):
    cout << "p3.speak() = " << p3.speak()
        << endl;
}
```

Table of Contents

Abstract class concept

Usually, root classes define a common interface for derived classes

- No implementation (or partial one)

Abstract class concept

Usually, root classes define a common interface for derived classes

- No implementation (or partial one)
- Example with `Instrument` : just a contract, no wished instances (it is a concept)

```
class Instrument {
public:
  /// Pure virtual functions:
  virtual void play(note) const = 0;
  virtual char* what() const = 0;
  virtual void adjust(int) = 0;
};
// Then Instrument is an abstract class
```

- Virtual pure function: keyword `virtual`, at the end add "=0"

Abstract class concept

Usually, root classes define a common interface for derived classes

- No implementation (or partial one)
- Example with `Instrument` : just a contract, no wished instances (it is a concept)

```cpp
class Instrument {
public:
    /// Pure virtual functions:
    virtual void play(note) const = 0;
    virtual char* what() const = 0;
    virtual void adjust(int) = 0;
};
// Then Instrument is an abstract class
```

- Virtual pure function: keyword `virtual`, at the end add "=0"
- Compiler put 0 in the VTABLE!
- Derived class is abstract, except if it implements virtual pure functions

Abstract class concept

Usually, root classes define a common interface for derived classes

- No implementation (or partial one)
- Example with `Instrument` : just a contract, no wished instances (it is a concept)

```
class Instrument {
public:
    /// Pure virtual functions:
    virtual void play(note) const = 0;
    virtual char* what() const = 0;
    virtual void adjust(int) = 0;
};
// Then Instrument is an abstract class
```

- Virtual pure function: keyword `virtual`, at the end add `"=0"`
- Compiler put 0 in the VTABLE!
- Derived class is abstract, except if it implements virtual pure functions
- Compiler ensures an abstract class is not instantiated

Abstract class concept

Usually, root classes define a common interface for derived classes

- No implementation (or partial one)
- Example with `Instrument` : just a contract, no wished instances (it is a concept)

```
class Instrument {
public:
    /// Pure virtual functions:
    virtual void play(note) const = 0;
    virtual char* what() const = 0;
    virtual void adjust(int) = 0;
};
// Then Instrument is an abstract class
```

- Virtual pure function: keyword `virtual`, at the end add `"=0"`
- Compiler put 0 in the VTABLE!
- Derived class is abstract, except if it implements virtual pure functions
- Compiler ensures an abstract class is not instantiated
- Make an abstract class: structure correctly your code software design

Definition for virtual pures functions

One does not prevent the other!

- Indicates always an abstract class
- Provides a default code (eg. avoid duplication code)

Definition for virtual pures functions

One does not prevent the other!

- Indicates always an abstract class
- Provides a default code (eg. avoid duplication code)

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

class Pet {
public:
    virtual void speak() const = 0;
    virtual void eat() const = 0;
    /// Inline pure virtual definitions
    /// are illegal:
    ///!  virtual void sleep() const = 0 {}
};

// OK, not defined inline
void Pet::eat() const {
    cout << "Pet::eat()" << endl;
}
```

```cpp
void Pet::speak() const {
    cout << "Pet::speak()" << endl;
}

class Dog : public Pet {
public:
    // Use the common Pet code:
    void speak() const { Pet::speak(); }
    void eat() const { Pet::eat(); }
};

int main() {
    Dog simba;
    simba.speak();
    simba.eat();
}
```

Definition for virtual pures functions

One does not prevent the other!

- Indicates always an abstract class
- Provides a default code (eg. avoid duplication code)

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

class Pet {
public:
    virtual void speak() const = 0;
    virtual void eat() const = 0;
    /// Inline pure virtual definitions
    /// are illegal:
    ///!  virtual void sleep() const = 0 {}
};

// OK, not defined inline
void Pet::eat() const {
    cout << "Pet::eat()" << endl;
}
```

```cpp
void Pet::speak() const {
    cout << "Pet::speak()" << endl;
}

class Dog : public Pet {
public:
    // Use the common Pet code:
    void speak() const { Pet::speak(); }
    void eat() const { Pet::eat(); }
};

int main() {
    Dog simba;
    simba.speak();
    simba.eat();
}
```

- Values in VTABLE `Pet` are equal to 0
- But the code exist somewhere and it may be used.

VTABLE and inheritance

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

class Pet {
   string pname;
public:
   Pet(const string& petName) :
      pname(petName) {}
   virtual string name() const {
      return pname;
   }
   virtual string speak() const {
      return "";
   }
};

class Dog : public Pet {
   string name;
public:
   Dog(const string& petName) :
      Pet(petName) {}
   // New virtual function:
   virtual string sit() const {
      return Pet::name() + " sits";
   }
   string speak() const { // Override
      return Pet::name() + " says 'Ouaf!'";
   }
};
```

VTABLE and inheritance

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

class Pet {
    string pname;
public:
    Pet(const string& petName) :
        pname(petName) {}
    virtual string name() const {
        return pname;
    }
    virtual string speak() const {
        return "";
    }
};

class Dog : public Pet {
    string name;
public:
    Dog(const string& petName) :
        Pet(petName) {}
    // New virtual function:
    virtual string sit() const {
        return Pet::name() + " sits";
    }
    string speak() const { // Override
        return Pet::name() + " says Ouaf!'";
    }
};
```

```cpp
int main() {
    Pet* p[] = { new Pet("generic"),
    new Dog("bob")};
    cout << "p[0]->speak() = "
       << p[0]->speak() << endl;
    cout << "p[1]->speak() = "
       << p[1]->speak() << endl;
    ///! cout << "p[1]->sit() = "
    ///!
<< p[1]->sit() << endl; // Illegal
}
```

Functioning

Basically the VTABLE is extended, still exists in `Dog` and derived classes

VTABLE and inheritance

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

class Pet {
    string pname;
public:
    Pet(const string& petName) :
        pname(petName) {}
    virtual string name() const {
        return pname;
    }
    virtual string speak() const {
        return "";
    }
};

class Dog : public Pet {
    string name;
public:
    Dog(const string& petName) :
        Pet(petName) {}
    // New virtual function:
    virtual string sit() const {
        return Pet::name() + " sits";
    }
    string speak() const { // Override
        return Pet::name() + " says 'Ouaf!'";
    }
};
```

```
int main() {
    Pet* p[] = { new Pet("generic"),
    new Dog("bob")};
    cout << "p[0]->speak() = "
       << p[0]->speak() << endl;
    cout << "p[1]->speak() = "
       << p[1]->speak() << endl;
    ///! cout << "p[1]->sit() = "
    ///!
<< p[1]->sit() << endl; // Illegal
}
```

Functioning

Basically the VTABLE is extended, still exists in Dog and derived classes

Attention

New methods are (OBVIOUSLY!) inaccessible by ancestor : compiler checks it

Université de Poitiers USTH

VTABLE and inheritance

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

class Pet {
    string pname;
public:
    Pet(const string& petName) :
        pname(petName) {}
    virtual string name() const {
        return pname;
    }
    virtual string speak() const {
        return "";
    }
};

class Dog : public Pet {
    string name;
public:
    Dog(const string& petName) :
        Pet(petName) {}
    // New virtual function:
    virtual string sit() const {
        return Pet::name() + " sits";
    }
    string speak() const { // Override
        return Pet::name() + " says 'Ouaf!'";
    }
};
```

```
int main() {
    Pet* p[] = { new Pet("generic"),
    new Dog("bob")};
    cout << "p[0]->speak() = "
       << p[0]->speak() << endl;
    cout << "p[1]->speak() = "
       << p[1]->speak() << endl;
    //! cout << "p[1]->sit() = "
    //!
<< p[1]->sit() << endl; // Illegal
}
```

Functioning

Basically the VTABLE is extended, still exists in Dog and derived classes

Attention

New methods are (OBVIOUSLY!) inaccessible by ancestor : compiler checks it

RTTI : Real Time Type Information

Object slicing

Transmission mode: ok when pointer or reference, else problem...

- The object is sliced: become a superclass type
- It loses its own data members.

Object slicing

Transmission mode: ok when pointer or reference, else problem...

- The object is sliced: become a superclass type
- It loses its own data members.

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

class Pet {
    string pname;
public:
    Pet(const string& name) : pname(name) {}
    virtual string name() const {
        return pname;
    }
    virtual string description() const {
        return "This is " + pname;
    }
};

class Dog : public Pet {
string favoriteActivity;
public:
    Dog( const string& name,
        const string& activity)
        : Pet(name),
        favoriteActivity(activity) {}
    string description() const {
        return Pet::name() + " likes to " +
            favoriteActivity;
    }
};

void describe(Pet p) { // Slices the object
    cout << p.description() << endl;
}

int main() {
    Pet p("Alfred");
    Dog d("Fluffy", "sleep");
    describe(p);
    describe(d);
}
```

Object slicing

Transmission mode: ok when pointer or reference, else problem...

- The object is sliced: become a superclass type
- It loses its own data members.

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

class Pet {
   string pname;
public:
   Pet(const string& name) : pname(name) {}
   virtual string name() const {
      return pname;
   }
   virtual string description() const {
      return "This is " + pname;
   }
};

class Dog : public Pet {
string favoriteActivity;
public:
   Dog( const string& name,
      const string& activity)
      : Pet(name),
      favoriteActivity(activity) {}
   string description() const {
      return Pet::name() + " likes to " +
         favoriteActivity;
   }
};

void describe(Pet p) { // Slices the object
   cout << p.description() << endl;
}

int main() {
   Pet p("Alfred");
   Dog d("Fluffy", "sleep");
   describe(p);
   describe(d);
}
```

Result

```
$ ./sources/Slicing
This is Alfred
This is Fluffy
```

Object slicing

Transmission mode: ok when pointer or reference, else problem...

- The object is sliced: become a superclass type
- It loses its own data members.

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

class Pet {
    string pname;
public:
    Pet(const string& name) : pname(name) {}
    virtual string name() const {
        return pname;
    }
    virtual string description() const {
        return "This is " + pname;
    }
};

class Dog : public Pet {
string favoriteActivity;
public:
    Dog( const string& name,
        const string& activity)
        : Pet(name),
        favoriteActivity(activity) {}
    string description() const {
        return Pet::name() + " likes to " +
            favoriteActivity;
    }
};

void describe(Pet p) { // Slices the object
    cout << p.description() << endl;
}

int main() {
    Pet p("Alfred");
    Dog d("Fluffy", "sleep");
    describe(p);
    describe(d);
}
```

Result

```
$ ./sources/Slicing
This is Alfred
This is Fluffy
```

Virtual pure function : enforce security by prohibiting the *object slicing*

Table of Contents

Virtual function and constructors

VPTR Initialisation must be done first.

Performance cost

- "C Macro features" not possible for virtual methods, thus prefer use of *inline*
- Add code inside the constructor
  - Initialize VPTR, test on `this`, call superclass constructor...

Virtual function and constructors

VPTR Initialisation must be done first.

Performance cost

- "C Macro features" not possible for virtual methods, thus prefer use of *inline*
- Add code inside the constructor
  - Initialize VPTR, test on `this`, call superclass constructor...
- Constructors: avoid use *inline*, for reducing code size

Virtual function and constructors

VPTR Initialisation must be done first.

Performance cost

- "C Macro features" not possible for virtual methods, thus prefer use of *inline*
- Add code inside the constructor
  - Initialize VPTR, test on `this`, call superclass constructor...
- Constructors: avoid use *inline*, for reducing code size

Order among constructors calls

Descending, for handling inherited data members

Virtual function and constructors

VPTR Initialisation must be done first.

Performance cost

- "C Macro features" not possible for virtual methods, thus prefer use of *inline*
- Add code inside the constructor
  - Initialize VPTR, test on `this`, call superclass constructor...
- Constructors: avoid use *inline*, for reducing code size

Order among constructors calls

Descending, for handling inherited data members

Call virtual function is prohibited in constructors

1. Risk access to uninitialized data member Risque d'accès membres données
2. VPTR is not correctly initialized, which method must be called?

Virtual functions and Destructors

Rule : destructors may and MUST be virtual

- Otherwise, bad destruction through a superclass pointer...

Virtual functions and Destructors

Rule : destructors may and MUST be virtual

- Otherwise, bad destruction through a superclass pointer...
  - The most specialized destructor must be called
- Example:

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

class Base1 {
public:
   ~Base1() {
      cout << "~Base1()\n";
   }
};

class Derived1 : public Base1 {
public:
   ~Derived1() {
      cout << "~Derived1()\n";
   }
};

class Base2 {
public:
   virtual ~Base2() {
      cout << "~Base2()\n";
   }
};
```

Virtual functions and Destructors

Rule : destructors may and MUST be virtual

- Otherwise, bad destruction through a superclass pointer...
  - The most specialized destructor must be called
- Example:

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

class Base1 {
public:
    ~Base1() {
        cout << "~Base1()\n";
    }
};

class Derived1 : public Base1 {
public:
    ~Derived1() {
        cout << "~Derived1()\n";
    }
};

class Base2 {
public:
    virtual ~Base2() {
        cout << "~Base2()\n";
    }
};
```

```
class Derived2 : public Base2 {
public:
    ~Derived2() {
        cout << "~Derived2()\n";
    }
};

int main() {
    Base1* bp = new Derived1; // Upcast
    delete bp;
    Base2* b2p = new Derived2; // Upcast
    delete b2p;
}
```

Virtual functions and Destructors

Rule : destructors may and MUST be virtual

- Otherwise, bad destruction through a superclass pointer...
  - The most specialized destructor must be called
- Example:

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

class Base1 {
public:
    ~Base1() {
        cout << "~Base1()\n";
    }
};

class Derived1 : public Base1 {
public:
    ~Derived1() {
        cout << "~Derived1()\n";
    }
};

class Base2 {
public:
    virtual ~Base2() {
        cout << "~Base2()\n";
    }
};
```

```
class Derived2 : public Base2 {
public:
    ~Derived2() {
        cout << "~Derived2()\n";
    }
};

int main() {
    Base1* bp = new Derived1; // Upcast
    delete bp;
    Base2* b2p = new Derived2; // Upcast
    delete b2p;
}
```

Displays :

```
~Base1()
~Derived2()
~Base2()
```

Virtual functions and Destructors

Rule : destructors may and MUST be virtual

- Otherwise, bad destruction through a superclass pointer...
  - The most specialized destructor must be called
- Example:

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

class Base1 {
public:
   ~Base1() {
      cout << "~Base1()\n";
   }
};

class Derived1 : public Base1 {
public:
   ~Derived1() {
      cout << "~Derived1()\n";
   }
};

class Base2 {
public:
   virtual ~Base2() {
      cout << "~Base2()\n";
   }
};

class Derived2 : public Base2 {
public:
   ~Derived2() {
      cout << "~Derived2()\n";
   }
};

int main() {
   Base1* bp = new Derived1; // Upcast
   delete bp;
   Base2* b2p = new Derived2; // Upcast
   delete b2p;
}
```

Displays :

```
~Base1()
~Derived2()
~Base2()
```

Risk of insidious bug, when destructor is not virtual...

Use of a pure virtual destructor

Sometimes inevitable (ex: unique method in abstract class), but definition is obligatory...

Must we overload a virtual destructor?

```
class AbstractBase {
public:
  virtual ~AbstractBase() = 0;
};

AbstractBase::~AbstractBase() {}

class Derived : public AbstractBase {};

int main() {
  Derived d;
}
```

Use of a pure virtual destructor

Sometimes inevitable (ex: unique method in abstract class), but definition is obligatory...

Must we overload a virtual destructor?

```
class AbstractBase {
public:
   virtual ~AbstractBase() = 0;
};

AbstractBase::~AbstractBase() {}

class Derived : public AbstractBase {};

int main() {
   Derived d;
}
```

In practice: nope default destructor is enough

Use of a pure virtual destructor

Sometimes inevitable (ex: unique method in abstract class), but definition is obligatory...

Must we overload a virtual destructor?

```
class AbstractBase {
public:
    virtual ~AbstractBase() = 0;
};

AbstractBase::~AbstractBase() {}

class Derived : public AbstractBase {};

int main() {
    Derived d;
}
```

In practice: nope default destructor is enough

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

class Pet {
public:
    virtual ~Pet() = 0;
};
Pet::~Pet() {
    cout << "~Pet()" << endl;
}

class Dog : public Pet {
public:
    ~Dog() {
        cout << "~Dog()" << endl;
    }
};

int main() {
    Pet* p = new Dog; // Upcast
    delete p; // Virtual destructor call
}
```

Use of a pure virtual destructor

Sometimes inevitable (ex: unique method in abstract class), but definition is obligatory...

Must we overload a virtual destructor?

```
class AbstractBase {
public:
    virtual ~AbstractBase() = 0;
};

AbstractBase::~AbstractBase() {}

class Derived : public AbstractBase {};

int main() {
    Derived d;
}
```

In practice: nope default destructor is enough

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

class Pet {
public:
    virtual ~Pet() = 0;
};
Pet::~Pet() {
    cout << "~Pet()" << endl;
}

class Dog : public Pet {
public:
    ~Dog() {
        cout << "~Dog()" << endl;
    }
};

int main() {
    Pet* p = new Dog; // Upcast
    delete p; // Virtual destructor call
}
```

Conclusion

When you add at least one virtual function, you must specify a virtual destructor

Virtual destructor

Late binding works in method excepted in destructor

```
#include <iostream>
using namespace std;
/// Base class
class Base {
public:
    virtual ~Base() {
        cout << "Base1()" << endl;
        f(); /// Which version?
    }
    virtual void f() {
        cout << "Base::f()" << endl;
    }
};
/// Derived class
class Derived : public Base {
public:
    ~Derived() {
        cout << "~Derived()" << endl;
    }
    void f() {
        cout << "Derived::f()" << endl;
    }
};
/// Test: Which version of f is used?
int main() {
    Base* bp = new Derived; // Upcast
    delete bp;
}
```

Virtual destructor

Late binding works in method excepted in destructor

```cpp
#include <iostream>
using namespace std;
/// Base class
class Base {
public:
    virtual ~Base() {
        cout << "Base1()" << endl;
        f(); /// Which version?
    }
    virtual void f() {
        cout << "Base::f()" << endl;
    }
};
/// Derived class
class Derived : public Base {
public:
    ~Derived() {
        cout << "~Derived()" << endl;
    }
    void f() {
        cout << "Derived::f()" << endl;
    }
};
/// Test: Which version of f is used?
int main() {
    Base* bp = new Derived; // Upcast
    delete bp;
}
```

Here, `Base::f()` is called

Virtual destructor

Late binding works in method excepted in destructor

```
#include <iostream>
using namespace std;
/// Base class
class Base {
public:
    virtual ~Base() {
        cout << "Base1()" << endl;
        f(); /// Which version?
    }
    virtual void f() {
        cout << "Base::f()" << endl;
    }
};
/// Derived class
class Derived : public Base {
public:
    ~Derived() {
        cout << "~Derived()" << endl;
    }
    void f() {
        cout << "Derived::f()" << endl;
    }
};
/// Test: Which version of f is used?
int main() {
    Base* bp = new Derived; // Upcast
    delete bp;
}
```

Here, `Base::f()` is called

- Destructors are called by going back up the hierarchy
- If the most specialized function is called in destructor, you may access to already deleted data members
- VPTR still exist, but it is just ignored
- Compiler uses *early binding*, to ensure call of “local” function

Objects hierarchy

Containers problems: who own contents?

```
class Stack {
   struct Link {
      void* data;
      Link* next;
      Link(void* dat, Link* nxt):
      data(dat), next(nxt) {}
   }* head;
public:
   Stack() : head(0) {}
   ~Stack() {
      if ( head != 0 ) {
         cerr<< "Stack not empty" <<endl;
      }
   }
   void push(void* dat) {
      head = new Link(dat, head);
   }
   void* peek() const {
      return head ? head->data : 0;
   }
   void* pop() {
      if(head == 0) return 0;
      void* result = head->data;
      Link* oldHead = head;
      head = head->next;
      delete oldHead;
      return result;
   }
};
```

Objects hierarchy

Containers problems: who own contents?

```
class Stack {
   struct Link {
      void* data;
      Link* next;
      Link(void* dat, Link* nxt):
      data(dat), next(nxt) {}
   }* head;
public:
   Stack() : head(0) {}
   ~Stack() {
      if ( head != 0 ) {
         cerr<< "Stack not empty" <<endl;
      }
   }
   void push(void* dat) {
      head = new Link(dat, head);
   }
   void* peek() const {
      return head ? head->data : 0;
   }
   void* pop() {
      if(head == 0) return 0;
      void* result = head->data;
      Link* oldHead = head;
      head = head->next;
      delete oldHead;
      return result;
   }
};
```

Possible use:

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
/// ...
int main(int argc, char* argv[]) {
   /// File name is argument
   if ( argc != 2 ) return -1;
   ifstream in(argv[1]);
   Stack textlines;
   string line;
   /// Read file and store lines in the stack:
   while( getline(in, line) )
      textlines.push( new string(line) ); // alloc
   /// Pop the lines from the stack and print them:
   string* s;
   while((s = (string*)textlines.pop()) != 0) {
      cout << *s << endl;
      delete s; // User must delete contents
   }
   return 0; // before stack destruction
}
```

Objects hierarchy

Containers problems: who own contents?

```
class Stack {
  struct Link {
    void* data;
    Link* next;
    Link(void* dat, Link* nxt):
    data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack() {
    if ( head != 0 ) {
      cerr<< "Stack not empty" <<endl;
    }
  }
  void push(void* dat) {
    head = new Link(dat, head);
  }
  void* peek() const {
    return head ? head->data : 0;
  }
  void* pop() {
    if(head == 0) return 0;
    void* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
  }
};
```

Possible use:

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
/// ...
int main(int argc, char* argv[]) {
  /// File name is argument
  if ( argc != 2 ) return -1;
  ifstream in(argv[1]);
  Stack textlines;
  string line;
  /// Read file and store lines in the stack:
  while( getline(in, line) )
    textlines.push( new string(line) ); // alloc
  /// Pop the lines from the stack and print them:
  string* s;
  while((s = (string*)textlines.pop()) != 0) {
    cout << *s << endl;
    delete s; // User must delete contents
  }
  return 0; // before stack destruction
}
```

Otherwise *memory leaks*

Object hierarchy implantation

Common solution (in Java): a unique root class

```
#ifndef OSTACK_H
#define OSTACK_H
/// Abstract base objet
class Object {
public:
   virtual ~Object() = 0;
};
inline Object::~Object() {}
/// An objects' stack
class Stack {
   struct Link {
      Object* data; // Store objects
      Link* next;
      Link(Object* dat, Link* nxt) :
      data(dat), next(nxt) {}
   }* head;
public:
   Stack() : head(0) {}
   ~Stack(){ // we can delete objects
      while(head) delete pop();
   }
   void push(Object* dat) {
      head = new Link(dat, head);
   }
   Object* peek() const {
      return head ? head->data : 0;
   }
   Object* pop() {
      if(head == 0) return 0;
      Object* result = head->data;
      Link* oldHead = head;
      head = head->next;
      delete oldHead;
      return result;
   }
};
#endif
```

Object hierarchy implantation

Common solution (in Java): a unique root class

```
#ifndef OSTACK_H
#define OSTACK_H
/// Abstract base objet
class Object {
public:
    virtual ~Object() = 0;
};
inline Object::~Object() {}
/// An objects' stack
class Stack {
    struct Link {
        Object* data; // Store objects
        Link* next;
        Link(Object* dat, Link* nxt) :
        data(dat), next(nxt) {}
    }* head;
public:
    Stack() : head(0) {}
    ~Stack(){ // we can delete objects
        while(head) delete pop();
    }
    void push(Object* dat) {
        head = new Link(dat, head);
    }
    Object* peek() const {
        return head ? head->data : 0;
    }
    Object* pop() {
        if(head == 0) return 0;
        Object* result = head->data;
        Link* oldHead = head;
        head = head->next;
        delete oldHead;
        return result;
    }
};
#endif
```

The container is responsible of destruction!

Object hierarchy implantation

Common solution (in Java): a unique root class

```
#ifndef OSTACK_H
#define OSTACK_H
/// Abstract base objet
class Object {
public:
   virtual ~Object() = 0;
};
inline Object::~Object() {}
/// An objects' stack
class Stack {
   struct Link {
      Object* data; // Store objects
      Link* next;
      Link(Object* dat, Link* nxt) :
      data(dat), next(nxt) {}
   }* head;
public:
   Stack() : head(0) {}
   ~Stack(){ // we can delete objects
      while(head) delete pop();
   }
   void push(Object* dat) {
      head = new Link(dat, head);
   }
   Object* peek() const {
      return head ? head->data : 0;
   }
   Object* pop() {
      if(head == 0) return 0;
      Object* result = head->data;
      Link* oldHead = head;
      head = head->next;
      delete oldHead;
      return result;
   }
};
#endif
```

The container is responsible of destruction!

```
// Multiple inheritance required...
class MyString: public string, public Object {
public:
   ~MyString() { /// For test
      cout << "deleting string: " << *this << endl;
   }
   MyString(string s) : string(s) {}
};
int main(int argc, char* argv[]) {
   /// ... as previous version
   while(getline(in, line)) // Push MyString
   textlines.push(new MyString(line));
   /// Pop some lines from the stack:
   for(int i = 0; i < 10; i++) {
      MyString* s = (MyString*)textlines.pop();
      if( s == 0 ) break;
      cout << *s << endl;
      delete s;
   }
   cout << "Stack's destructor do the rest\n";
}
```

Object hierarchy implantation

Common solution (in Java): a unique root class

```
#ifndef OSTACK_H
#define OSTACK_H
/// Abstract base objet
class Object {
public:
    virtual ~Object() = 0;
};
inline Object::~Object() {}
/// An objects' stack
class Stack {
    struct Link {
        Object* data; // Store objects
        Link* next;
        Link(Object* dat, Link* nxt) :
        data(dat), next(nxt) {}
    }* head;
public:
    Stack() : head(0) {}
    ~Stack(){ // we can delete objects
        while(head) delete pop();
    }
    void push(Object* dat) {
        head = new Link(dat, head);
    }
    Object* peek() const {
        return head ? head->data : 0;
    }
    Object* pop() {
        if(head == 0) return 0;
        Object* result = head->data;
        Link* oldHead = head;
        head = head->next;
        delete oldHead;
        return result;
    }
};
#endif
```

The container is responsible of destruction!

```
// Multiple inheritance required...
class MyString: public string, public Object {
public:
    ~MyString() { /// For test
        cout << "deleting string: " << *this << endl;
    }
    MyString(string s) : string(s) {}
};
int main(int argc, char* argv[]) {
    /// ... as previous version
    while(getline(in, line)) // Push MyString
    textlines.push(new MyString(line));
    /// Pop some lines from the stack:
    for(int i = 0; i < 10; i++) {
        MyString* s = (MyString*)textlines.pop();
        if( s == 0 ) break;
        cout << *s << endl;
        delete s;
    }
    cout << "Stack's destructor do the rest\n";
}
```

Virtual operators(1/2)

Complex case, because requires two operands of unknown types...

Problem

- Operator works with two “upcast” to class `Math`
- Virtual function: solves one case (single dispatch)
- Then we need the *multiple dispatching* ...

Virtual operators(1/2)

Complex case, because requires two operands of unknown types...

Problem

- Operator works with two “upcast” to class `Math`
- Virtual function: solves one case (single dispatch)
- Then we need the *multiple dispatching* ...

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

class Matrix;
class Scalar;
class Vector;

// Abstract class
class Math {
public:
    virtual Math& operator*(Math& rv) = 0;
    virtual Math& multiply(Matrix*) = 0;
    virtual Math& multiply(Scalar*) = 0;
    virtual Math& multiply(Vector*) = 0;
    virtual ~Math() {}
};
```

Virtual operators(1/2)

Complex case, because requires two operands of unknown types...

Problem

- Operator works with two "upcast" to class `Math`
- Virtual function: solves one case (single dispatch)
- Then we need the *multiple dispatching* ...

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

class Matrix;
class Scalar;
class Vector;

// Abstract class
class Math {
public:
    virtual Math& operator*(Math& rv) = 0;
    virtual Math& multiply(Matrix*) = 0;
    virtual Math& multiply(Scalar*) = 0;
    virtual Math& multiply(Vector*) = 0;
    virtual ~Math() {}
};
```

```cpp
class Matrix : public Math {
public:
    Math& operator*(Math& rv) {
        // 2nd dispatch
        return rv.multiply(this);
    }
    Math& multiply(Matrix*) {
        cout << "Matrix * Matrix" << endl;
        return *this;
    }
    Math& multiply(Scalar*) {
        cout << "Scalar * Matrix" << endl;
        return *this;
    }
    Math& multiply(Vector*) {
        cout << "Vector * Matrix" << endl;
        return *this;
    }
};
```

Virtual operators (2/2)

```
class Scalar : public Math {
public:
    Math& operator*(Math& rv) {
        // 2nd dispatch
        return rv.multiply(this);
    }
    Math& multiply(Matrix*) {
        cout << "Matrix * Scalar" << endl;
        return *this;
    }
    Math& multiply(Scalar*) {
        cout << "Scalar * Scalar" << endl;
        return *this;
    }
    Math& multiply(Vector*) {
        cout << "Vector * Scalar" << endl;
        return *this;
    }
};
```

Virtual operators (2/2)

```
class Scalar : public Math {
public:
    Math& operator*(Math& rv) {
        // 2nd dispatch
        return rv.multiply(this);
    }
    Math& multiply(Matrix*) {
        cout << "Matrix * Scalar" << endl;
        return *this;
    }
    Math& multiply(Scalar*) {
        cout << "Scalar * Scalar" << endl;
        return *this;
    }
    Math& multiply(Vector*) {
        cout << "Vector * Scalar" << endl;
        return *this;
    }
};
```

```
class Vector : public Math {
public:
    Math& operator*(Math& rv) {
        // 2nd dispatch
        return rv.multiply(this);
    }
    Math& multiply(Matrix*) {
        cout << "Matrix * Vector" << endl;
        return *this;
    }
    Math& multiply(Scalar*) {
        cout << "Scalar * Vector" << endl;
        return *this;
    }
    Math& multiply(Vector*) {
        cout << "Vector * Vector" << endl;
        return *this;
    }
};

int main() {
    Matrix m; Vector v; Scalar s;
    Math* math[] = { &m, &v, &s };
    for(int i = 0; i < 3; i++)
        for(int j = 0; j < 3; j++) {
            Math& m1 = *math[i];
            Math& m2 = *math[j];
            m1 * m2;
        }
}
```

Virtual operators (2/2)

```
class Scalar : public Math {
public:
    Math& operator*(Math& rv) {
        // 2nd dispatch
        return rv.multiply(this);
    }
    Math& multiply(Matrix*) {
        cout << "Matrix * Scalar" << endl;
        return *this;
    }
    Math& multiply(Scalar*) {
        cout << "Scalar * Scalar" << endl;
        return *this;
    }
    Math& multiply(Vector*) {
        cout << "Vector * Scalar" << endl;
        return *this;
    }
};
```

- Works for others operators
- 9 computations done
- Display (`Matrix * Matrix`, ..., `Scalar * Scalar`)

```
class Vector : public Math {
public:
    Math& operator*(Math& rv) {
        // 2nd dispatch
        return rv.multiply(this);
    }
    Math& multiply(Matrix*) {
        cout << "Matrix * Vector" << endl;
        return *this;
    }
    Math& multiply(Scalar*) {
        cout << "Scalar * Vector" << endl;
        return *this;
    }
    Math& multiply(Vector*) {
        cout << "Vector * Vector" << endl;
        return *this;
    }
};

int main() {
    Matrix m; Vector v; Scalar s;
    Math* math[] = { &m, &v, &s };
    for(int i = 0; i < 3; i++)
        for(int j = 0; j < 3; j++) {
            Math& m1 = *math[i];
            Math& m2 = *math[j];
            m1 * m2;
        }
}
```

Downcasting

Since it's possible to go up hierarchy (upcast), how go down?

- If you must go down then you fail your software architecture ...
- Else, explicit conversion with `dynamic_cast`

```
#include <iostream>
using namespace std;
class Pet { public: virtual ~Pet(){}};
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
   Pet* b = new Cat; // Upcast
   // Try to cast it to Dog*:
   Dog* d1 = dynamic_cast<Dog*>(b);
   // Try to cast it to Cat*:
   Cat* d2 = dynamic_cast<Cat*>(b);
   cout << "d1 = " << (long)d1 << endl;
   cout << "d2 = " << (long)d2 << endl;
}
```

Downcasting

Since it's possible to go up hierarchy (upcast), how go down?

- If you must go down then you fail your software architecture ...
- Else, explicit conversion with `dynamic_cast`

```cpp
#include <iostream>
using namespace std;
class Pet { public: virtual ~Pet(){}};
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
    Pet* b = new Cat; // Upcast
    // Try to cast it to Dog*:
    Dog* d1 = dynamic_cast<Dog*>(b);
    // Try to cast it to Cat*:
    Cat* d2 = dynamic_cast<Cat*>(b);
    cout << "d1 = " << (long)d1 << endl;
    cout << "d2 = " << (long)d2 << endl;
}
```

- Expensive cost ...
- Use `static_cast`?

```cpp
#include <iostream>
#include <typeinfo>
using namespace std;
class Shape {public: virtual ~Shape(){}; };
class Circle : public Shape {};
class Square : public Shape {};
class Other {};
```

```cpp
int main() {
    Circle c;
    Shape* s = &c; /// Upcast: normal and OK
    /// More explicit but unnecessary:
    s = static_cast<Shape*>(&c);
    /// (Since upcasting is such a safe and common
    /// operation, the cast becomes cluttering)
    Circle* cp = 0;
    Square* sp = 0;
    /// Static Navigation of class hierarchies
    /// requires extra type information:
    if(typeid(s) == typeid(cp)) // C++ RTTI
      cp = static_cast<Circle*>(s);
    if(typeid(s) == typeid(sp))
      sp = static_cast<Square*>(s);
    if(cp != 0)
      cout << "It's a circle!" << endl;
    if(sp != 0)
      cout << "It's a square!" << endl;
    /// Static navigation is ONLY an efficiency ha
    /// dynamic_cast is always safer. However:
    /// Other* op = static_cast<Other*>(s);
    /// Conveniently gives an error message, while
    Other* op2 = (Other*)s;
    // does not
} // NB: class type_id has a name() function ...
```