Object Oriented C++

Ettore Speziale

Politecnico di Milano

Contents

Contents

Object Orientation
You Should Known ...

Today's software is complex:

- code base becomes bigger and bigger

Object Oriented programming:

- has been proven to be effective [1] on handling big things

Common knowledge:

- cool features – e.g. polymorphism
- features have a cost – e.g. virtual calls

C++ statement:

- if you known what you are doing, you can have both features and performance

[1] essential

Object Orientation
A Running Example

For this/these classes we will use a small example:

- a processor simulator

It is very simple:

- just print instructions instead of executing them

Contents

Classes
Evolving C **struct**

Abstract Data Types force programmer focusing on:

- data
- operations on data

Classes are the preferred way to implement ADT in C++:

- just take C **struct**
- allow to declare member functions as **struct** fields
- add scoping rules to implement information hiding
- introduce the **class** keyword as syntactic sugar

C++ classes are ready!

Classes
Hardware is Good

Considering classes an extension of **struct** allows to focus on data, without loosing the hardware perspective:

Big

```
class Big {
  char a;
  int b;
  char c;
};
```

Compact

```
class Compact {
  char a;
  char c;
  int b;
};
```

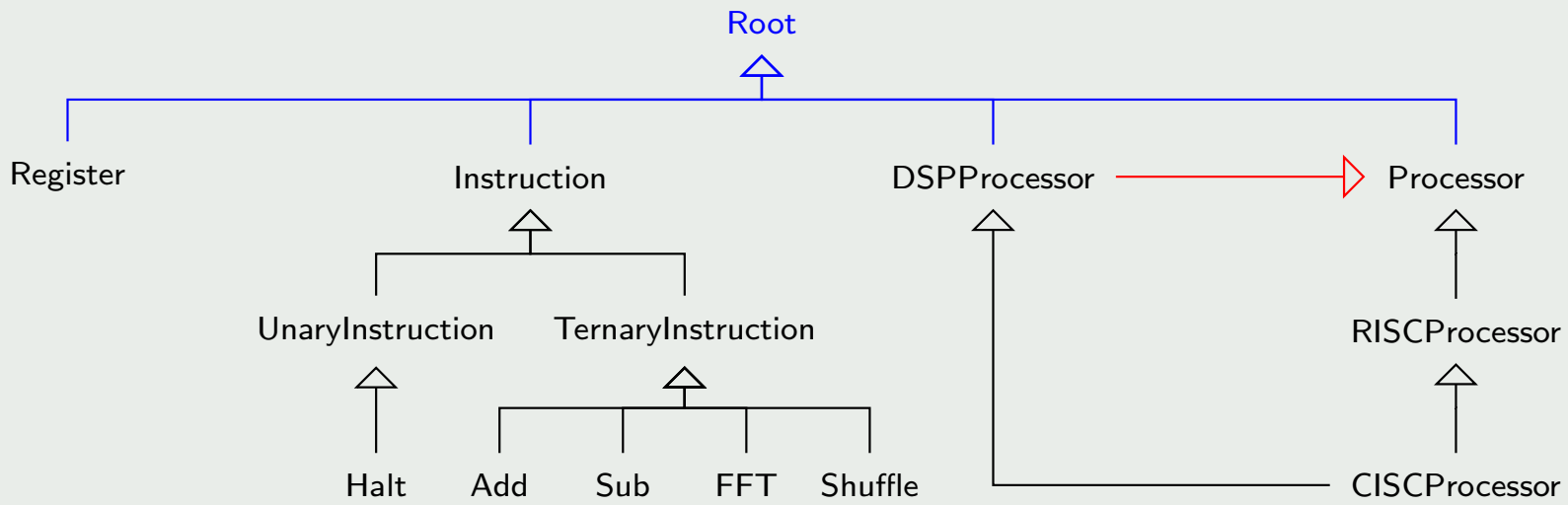
How much memory is consumed by Big and Compact?

- simple test in `simple-mem-layout.cpp`

The class hierarchy is used to:

- organizing classes
- establishing a contract with class users
- promoting code re-using

Processor Class Hierarchy



Class Hierarchy
Files

Showing code on slides is both boring and error-prone, so I will use as much as possible `vi` *and the shell. All sources are available on the course site. They are heavily commented. On slides there are only some tips.*

"Talk is cheap, show me the code" [5]

A Java-to-C++ translation table is available at slides end.

The reported class diagram refers to examples found in files `{java,cpp,dsp}-processor.cpp`

Class Hierarchy
Comment

C++ inheritance tree is not single rooted:

- it is not a tree, it is a DAG
- blue lines do not exist

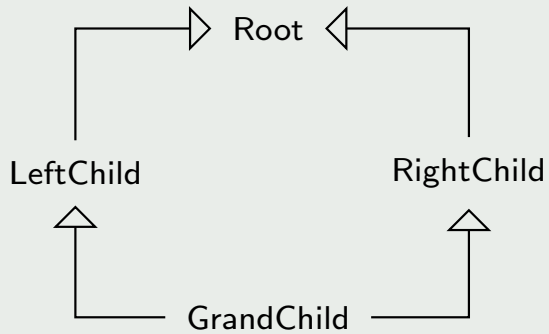
It is up to programmer avoiding dangerous inheritance shapes:

- diamond problem
- red extension should not be used

Let us analyze the diamond problem

It is an ambiguity in the class derivation process:

Basic Diamond



Both LeftChild and RightChild inherits members from Root. How are they inherited by GrandChild?

- a precise semantic is needed
- problem addressed in different ways
- we will see two basic solutions

Diamond Problem
Avoid by Construction

The problem arises only in presence of multiple inheritance:

- force using single inheritance – e.g. Java
- adopt a programming discipline – avoid using it

Problems:

- new constructs could be added to the language – e.g. Java interfaces
- limit code reuse

Diamond Problem
Construct Hierarchy Carefully

C++ does not prevent you using multiple inheritance:

- useful in many applications

It is up to the programmer avoiding dangerous derivations:

- it makes no sense deriving DSPProcessor from Processor
- it cannot be used alone
- its purpose is to implement code for DSP-specific instructions

Diamond Problem
Code

Showing code on slides is both boring and error-prone, so I will use as much as possible `vi` and the shell. All sources are available on the course site. They are heavily commented. On slides there are only some tips.

"Talk is cheap, show me the code" [5]

A Java-to-C++ translation table is available at slides end.

In the file `diamond-mem-layout.cpp` is shown how C++ allows building diamond in the derivation process, but only under the assumption that ambiguities can be solved at compile-time.

Polymorphism
Same Interface, Different Implementations?

Polymorphism is a key concept on modern programming languages:

- it enable handling different data types using the same interface
- it allows defining the behavior of a family of data types, decoupling specification from implementation

C++ supports two kind of polymorphism:

- subtype polymorphism
- parametric polymorphism

Now, focus on the first kind

Subtype Polymorphism
Recalling Java . . .

You should known it:

- a class define some functionalities
- subclasses refine functionalities
- programmers uses root class interface
- call to the actual implementation resolved at runtime – dynamic binding

Dynamic binding performed using a virtual table

Subtype Polymorphism
Java Example

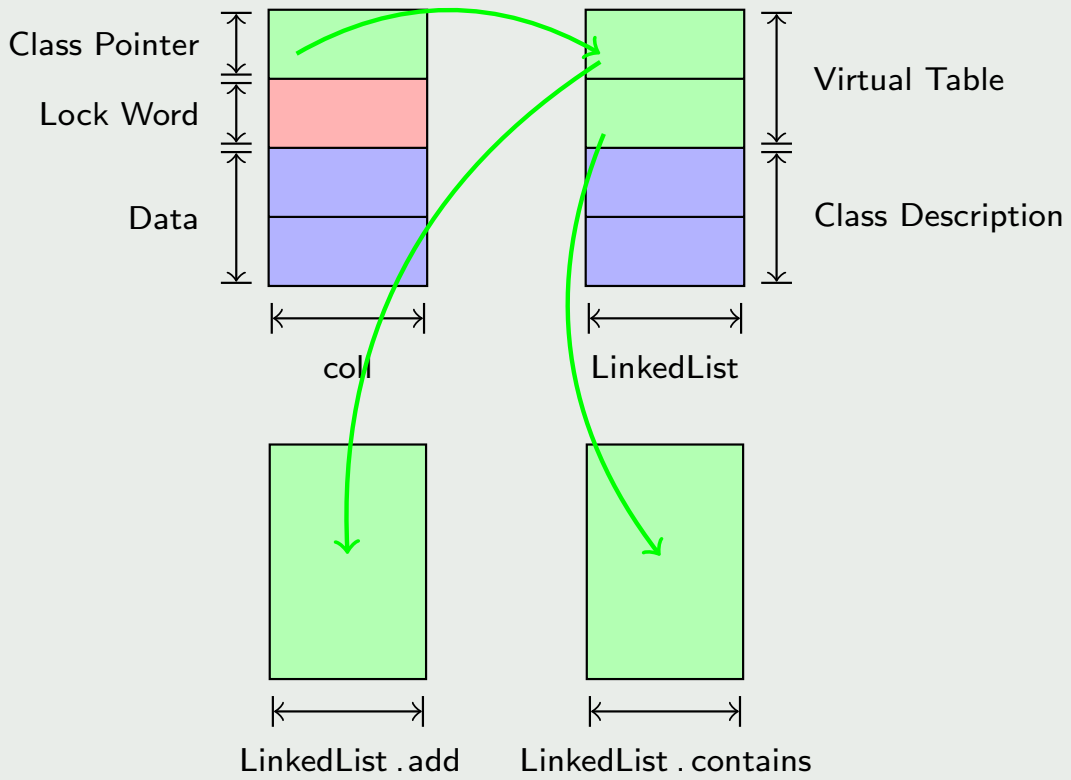
Dynamic Collections

```
interface Collection {
  public boolean add(Object o);
  public boolean contains();
}

class LinkedList implements Collection {
  public boolean add(Object o) { ... }
  public boolean contains() { ... }
}

Collection coll = new LinkedList();
coll.add(new Object());
```

LinkedList in HotSpot Memory



C++ class layout is similar [2]

[2] See `cpp-processor.cpp`

Subtype Polymorphism
Code

Showing code on slides is both boring and error-prone, so I will use as much as possible `vi` *and the shell. All sources are available on the course site. They are heavily commented. On slides there are only some tips.*

"Talk is cheap, show me the code" [5]

A Java-to-C++ translation table is available at slides end.

Following slides will refer to `{java,cpp}-processor.cpp` files

Subtype Polymorphism
Comment

Try running `java-processor.cpp`:

- you do not get the expected result!

In Java all non-static methods are virtual:

- dynamic dispatching by default
- JVM tries to optimize calls

In C++ virtual tables is under programmer control:

- member functions calls resolved at compile-time by default
- dynamic dispatching enabled using the **virtual** keyword

Try running `cpp-processor.cpp`:

- size of *Processor increases due to VT emission!

Subtype Polymorphism
Overhead

Virtual Call

```
movl (%eax), %eax
movl (%eax), %edx
call *%edx

movl (%eax), %eax
addl $4, %eax
movl (%eax), %edx
call *%edx
```

Call overhead:

- VT fetching
- VT lookup
- indirect call

On average [1]:

- 5.2%/13.7% of execution time

Normal Call

```
call _ZN10LinkedList4sizeE
```

Call overhead:

- zero

Name Decoration

Some C++ features are not hardware-oriented:

- e.g. function overloading

These features allows an user to refer to different things using the same name:

- impossible for hardware distinguish between them
- C++ front-end decorates names in order to uniquely identify functions, classes, variables, ...

The `c++filt` tool allows to un-decorate names:

Decorated	Un-decorated
_ZN10LinkedList4sizeE	LinkedList :: size

Pure Virtual Functions
Code

Showing code on slides is both boring and error-prone, so I will use as much as possible `vi` *and the shell. All sources are available on the course site. They are heavily commented. On slides there are only some tips.*

"Talk is cheap, show me the code" [5]

A Java-to-C++ translation table is available at slides end.

In `program.cpp` there is an example of pure virtual function

Multiple Inheritance
Code

Showing code on slides is both boring and error-prone, so I will use as much as possible vi and the shell. All sources are available on the course site. They are heavily commented. On slides there are only some tips.

"Talk is cheap, show me the code" [5]

A Java-to-C++ translation table is available at slides end.

The `dsp-processor.cpp` file contains an example of multiple inheritance. Please keep attention at the inheritance graph:

- DSPProcessor is a processor, but it cannot be used alone
- it does not inherit from Processor, avoiding the diamond problem

Contents

Iterators
Augmenting Pointers

Recalling streams:

- an extension of files

Iterators built starting from pointers:

Pointers:

- abstract from memory holding data
- used to pass data by address

Iterators:

- mainly associated with containers
- abstract from the specific container holding pointed element

Iterators act like pointers:

- **operator**$*$, **operator**$->$, **operator**$++$, ...

Iterators
Code

Showing code on slides is both boring and error-prone, so I will use as much as possible vi and the shell. All sources are available on the course site. They are heavily commented. On slides there are only some tips.

"Talk is cheap, show me the code" [5]

A Java-to-C++ translation table is available at slides end.

Following slides refer to
`program{,-pointers,-custom-iterator,-iterator}.cpp`

Iterators
Comment

Starting from `program.cpp`:

- explicit access to instruction through **operator**[]

With pointers – `program-pointers.cpp`:

- C-style interface

With iterators – `program-custom-iterator.cpp`:

- same interface used with pointers

After refactoring, you get `program-iterator.cpp`

Iterators
Interaction with C++ Library

Showing code on slides is both boring and error-prone, so I will use as much as possible `vi` *and the shell. All sources are available on the course site. They are heavily commented. On slides there are only some tips.*

"Talk is cheap, show me the code" [5]

A Java-to-C++ translation table is available at slides end.

In `program-functors.cpp` iterators are used together with std :: for_each . Many algorithms of C++ library are iterator-based

Contents

Java Concepts

Java shares many concepts with C++:

- both are object-oriented
- both are C-based
- ...

Here is an incomplete translation table from Java to C++:

- Java abstract methods are called pure virtual functions in C++
- A C++ class is said to be abstract if it has at least one pure virtual function. If the only members of a C++ class are pure virtual functions, the class is said to be pure abstract
- Java interfaces can be realized in C++ using pure abstract classes

Contents

Bibliography I

Karel Driesen and Urs Hölzle.
The Direct Cost of Virtual Function Calls in C++.
In *OOPSLA*, 1996.

Bruce Eckel.
Thinking in C++ – Volume One: Introduction to Standard C++.
http://mindview.net/Books/TICPP/ThinkingInCPP2e.html.

Bruce Eckel and Chuck Allison.
Thinking in C++ – Volume Two: Practical Programming.
http://mindview.net/Books/TICPP/ThinkingInCPP2e.html.

Bibliography II

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design Patterns: Elements of Reusable Object-Oriented Software.
1994.

Linus Torvalds.
Re: SCO: "thread creation is about a thousand times faster than onnative.
https://lkml.org/lkml/2000/8/25/132.