Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-15/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 24

- Data abstraction
- Obiect Oriented programming

Towards Data Abstraction

- Control abstraction is a very old idea (subroutines!), though few languages provide it in a truly general form
- Data abstraction is somewhat newer, though its roots can be found in Simula67
  - An **Abstract Data Type** is one that is defined in terms of the operations that it supports (i.e., that can be performed upon it) rather than in terms of its structure or implementation

On abstractions

- Why abstractions?
  - easier to think about - hide what doesn't matter
  - protection - prevent access to things you shouldn't see
  - plug compatibility
    - replacement of pieces, often without recompilation, definitely without rewriting libraries
    - division of labor in software projects

Basic Data Abstractions

- We met some kinds of data abstraction when speaking about naming and scoping
- Recall that we traced the historical development of abstraction mechanisms
  - Static set of var — Basic
  - Locals — Fortran
  - Statics — Fortran, Algol 60, C
  - Modules — Modula-2, Ada 83
  - Module types — Euclid
  - Objects — Smalltalk, C++, Eiffel, Java Oberon, Modula-3, Ada 95

Basic Data Abstractions

- *Statics* allow a subroutine to retain values from one invocation to the next, while hiding the name in-between
- *Modules* allow a collection of subroutines to share some statics, still with hiding
  - If you want to build an abstract data type, though, you have to make the module a manager
- *Module types* allow the module to *be* the abstract data type - you can declare a bunch of them

Object-Oriented Programming

- Objects add **inheritance** and **dynamic method binding**
- Simula 67 introduced these, but didn't have data hiding
- The 3 key factors in OO programming
  - **Encapsulation**
  - **Inheritance**
  - **Dynamic method binding**

Encapsulation and Inheritance

- Visibility rules
  - Public and Private parts of an object declaration/definition
  - Two reasons to put things in the declaration
    - so programmers can get at them
    - so the compiler can understand them
  - At the very least the compiler needs to know the size of an object, even though the programmer isn't allowed to get at many or most of the fields (members) that contribute to that size
    - That's why private fields have to be in declaration

Encapsulation and Inheritance Classes (C++)

- C++ distinguishes among
  - public class members
    - accessible to anybody
  - protected class members
    - accessible to members of this or derived classes
  - private
    - accessible just to members of this class
- A C++ structure (*struct*) is simply a class whose members are public by default
- Ulike Java, C++ base classes can also be public, private, or protected

Encapsulation and Inheritance Classes (C++)

- Example:
- `class circle : public shape { ...`
  anybody can convert (assign) a circle* into a shape*
- `class circle : protected shape { ...`
  only members and friends of circle or its derived classes can convert (assign) a circle* into a shape*
- `class circle : private shape { ...`
  only members and friends of circle can convert (assign) a circle* into a shape*
- Visibility of base class affects visibility of inherited members

Initialization and Finalization

- We defined the **lifetime** of an object to be the interval during which it occupies space and can hold data
- Most object-oriented languages provide some sort of special mechanism to ***initialize*** an object automatically at the beginning of its lifetime
  - When written in the form of a subroutine, this mechanism is known as a ***constructor***
  - Note: A constructor does not allocate space
- A few languages provide a similar ***destructor*** mechanism to ***finalize*** an object automatically at the end of its lifetime
  - Mainly for efficient storage management

Initialization and Finalization

Issues

- choosing a constructor
  - Overloading, default constructors…
- references and values
  - If variables are references, then every object must be created explicitly - appropriate constructor is called
  - If variables are values, then object creation can happen implicitly as a result of elaboration
- execution order
  - When an object of a derived class is created in C++ or Java, the constructors for any base classes will be executed before the constructor for the derived class
- garbage collection vs. destructors

Invoking constructor of base class or members in C++

- `foo::foo(`*foo-params*`): bar(`*bar-args*`) { ...`
  - Constructor of foo invokes constructor of bar
- `foo::foo(`*args0*`):member(`*args1*`){ …`
  - Constructor of *member1* is executed before the body of foo's constructor, so *member* is not initialized with default constructor

Initialization vs Assignment in C++

`foo::operator=(&foo)` *versus*

`foo::foo(&foo)` //constructor

```
foo b;
   // calls no-argument constructor
foo f = b;
   // calls constructor with b as argument
foo b, f;
   // calls no-argument constructor
f = b;
// calls operator=
```

Dynamic binding

Key question: if *child* is derived from *parent* and we have a *parent* p* that points to an object *c* that's actually a *child*, what member function do we get when we call **`p->func()`** ?

- C++ default rule: static binding
  - The *func()* defined in *parent* is executed
- Dynamic binding for *virtual functions*
  - If *func()* is defined *virtual,* then the *func()* defined in *child* is called

More on C++ virtual functions

- If a virtual function has a "0" body in the parent class, then the function is said to be a *pure virtual* function and the parent class is said to be *abstract*
- You can't create objects of an abstract class; you have to declare them to be of derived classes
- Moreover any derived class ***must*** provide a body for the pure virtual function(s)
- Standard C++ supports multiple inheritance

Dynamic Method Binding

- Virtual functions in C++ are an example of ***dynamic method binding***
  - you don't know at compile time what type the object referred to by a variable will be at run time
- Simula also had virtual functions (all of which are abstract)
- In Smalltalk, Eiffel, Modula-3, and Java ***all*** member functions are virtual
- In Java a method or a class can be declared *final*

Dynamic Method Binding

- Note that inheritance does not obviate the need for generics
  - You won’t be able to talk about the elements of a container because you won't know their types
  - Generics allow to abstract over types
- Java has generics since 1.5, before that (checked) dynamic casts needed to be used

Dynamic Method Binding

- Data members of classes are implemented just like structures (records)
  - With (single) inheritance, derived classes have extra fields at the end
  - A pointer to the parent and a pointer to the child contain the same address - the child just knows that the struct goes farther than the parent does

Dynamic Method Binding

- Non-virtual functions require no space at run time; the compiler just calls the appropriate version, based on type of variable
  - Member functions are passed an extra, hidden, initial parameter: *this* (called *current* in Eiffel and *self* in Smalltalk)
- C++ philosophy is to avoid run-time overhead whenever possible (Sort of the legacy from C)
  - Languages like Smalltalk have (much) more run-time support

Virtual functions

- Virtual functions are the only thing that requires any trickiness
  - They are implemented by creating a dispatch table (*vtable*) for the class and putting a pointer to that table in the data of the object
  - Objects of a derived class have a different dispatch table
    - In the dispatch table, functions defined in the parent come first, though some of the pointers point to overridden versions

Dynamic Method Binding

```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k( ...
    virtual int l( ...
    virtual void m();
    virtual double n( ...
    ...
} F;
```

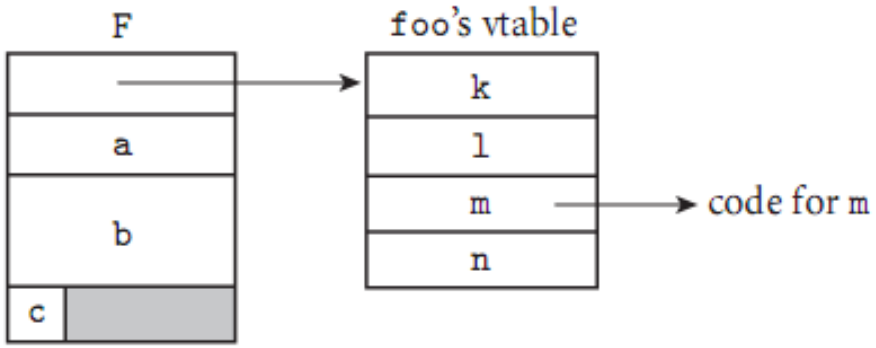

Figure 9.3 Implementation of virtual methods. The representation of object F begins with the address of the vtable for class foo. (All objects of this class will point to the same vtable.) The vtable itself consists of an array of addresses, one for the code of each virtual method of the class. The remainder of F consists of the representations of its fields.

Dynamic Method Binding

```
class bar : public foo {
    int w;
public:
    void m();  //override
    virtual double s( ...
    virtual char *t( ...
    ...
} B;
```

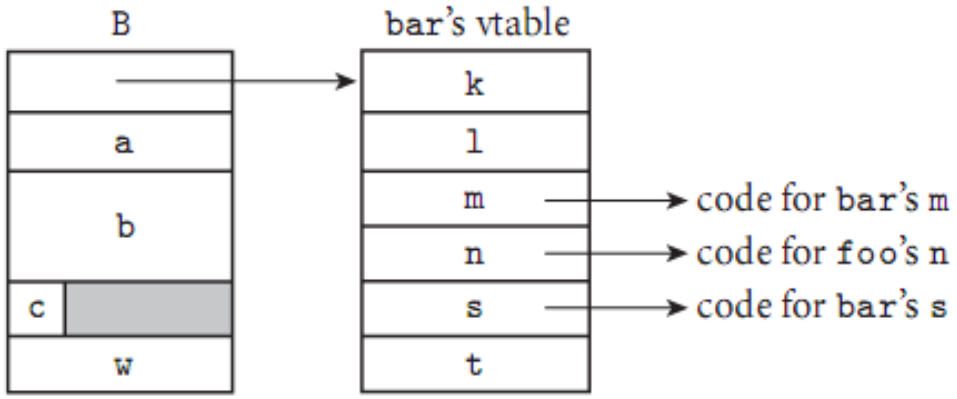

Figure 9.4 Implementationof single inheritance. As in Figure 9.3, the representation of object B begins with the address of its class' s vtable. The first four entries in the table represent the same members as they do for foo, except that one —m— has been overridden and now contains the address of the code for a different subroutine. Additional fields of bar follow the ones inherited from foo in the representation of B; additional virtual methods follow the ones inherited from foo in the vtable of class.

Dynamic Method Binding

- Note that if you can query the type of an object, then you need to be able to get from the object to run-time type info
  - The standard implementation technique is to put a pointer to the type info at the beginning of the vtable
  - You only have a vtable in C++ if your class has virtual functions
    - That's why you can't do a dynamic_cast on a pointer whose static type doesn't have virtual functions

Multiple Inheritance

- In C++, you can say

```
class professor : public
teacher, public researcher {
        ...
    }
```

  Here you get all the members of teacher ***and*** all the members of researcher
  - If there's anything that's in both (same name and argument types), then calls to the member are ambiguous; the compiler disallows them

Multiple Inheritance

- You can of course create your own member in the merged class

```
professor::print () {
   teacher::print ();
   researcher::print (); ...
}
```

- Or you could get both:

```
professor::tprint () {
   teacher::print ();
}
professor::rprint () {
   researcher::print ();
}
```

Multiple Inheritance

- Virtual base classes: In the usual case if you inherit from two classes that are both derived from some other class B, your implementation includes two copies of B's data members
- That's often fine, but other times you want a ***single*** copy of B
  - For that you make B a **virtual** base class

Object-Oriented Programming

- Anthropomorphism is central to the OO paradigm - you think in terms of *real-world* objects that interact to get things done
- Many OO languages are strictly sequential, but the model adapts well to parallelism as well
- Strict interpretation of the term
  - uniform data abstraction - everything is an object
  - inheritance
  - dynamic method binding

Object-Oriented Programming

- Lots of conflicting uses of the term out there object-oriented ***style*** available in many languages
  - data abstraction crucial
  - inheritance required by most users of the term O-O
  - centrality of dynamic method binding a matter of dispute

Object-Oriented Programming

- SMALLTALK is the canonical object-oriented language
  - It has all three of the characteristics listed above
  - It's based on the thesis work of Alan Kay at Utah in the late 1960 ‘s
  - It went through 5 generations at Xerox PARC, where Kay worked after graduating
  - Smalltalk-80 is the current standard

Object-Oriented Programming

- Other languages are described in what follows:
- Modula-3
  - single inheritance
  - all methods virtual
  - no constructors or destructors

Object-Oriented Programming

- Ada 95
  - *tagged* types
  - single inheritance
  - no constructors or destructors
  - *class-wide* parameters:
    - methods static by default
    - can define a parameter or pointer that grabs the object-specific version of all methods
      - base class doesn't have to decide what will be virtual
  - notion of *child* packages as an alternative to friends

Object-Oriented Programming

- Java
  - interfaces, *mix-in* inheritance
  - alternative to multiple inheritance
    - basically you inherit from one real parent and one or more interfaces, each of which contains ***only*** virtual functions and no data
    - this avoids the contiguity issues in multiple inheritance above, allowing a very simple implementation
  - all methods virtual

Object-Oriented Programming

- Is C++ object-oriented?
  - Uses all the right buzzwords
  - Has (multiple) inheritance and generics (templates)
  - Allows creation of user-defined classes that look just like built-in ones
  - Has all the low-level C stuff to escape the paradigm
  - Has static type checking

Object-Oriented Programming

- In the same category of questions:
  - Is Prolog a logic language?
  - Is Common Lisp functional?
- However, to be more precise:
  - Smalltalk is really pretty purely object-oriented
  - Prolog is primarily logic-based
  - Common Lisp is largely functional
  - C++ can be used in an object-oriented style