Object-Oriented Programming

- *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

Object-Oriented Programming

- *Module types* allow the module to *be* the abstract data type - you can declare a bunch of them
  - This is generally more intuitive
    - It avoids explicit object parameters to many operations
    - One minor drawback: If you have an operation that needs to look at the innards of two different types, you'd define both types in the same manager module in Modula-2
    - In C++ you need to make one of the classes (or some of its members) "friends" of the other class

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 (data hiding)
  - Inheritance
  - Dynamic method binding

Encapsulation and Inheritance

- Visibility rules
  - Public and Private parts of an object declaration/definition
  - 2 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

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

Dynamic Method Binding

- Note that inheritance does not obviate the need for generics
  - You might think: hey, I can define an abstract list class and then derive int_list, person_list, etc. from it, but the problem is you won't be able to talk about the elements because you won't know their types
  - That's what generics are for: abstracting over types
- Java doesn't have generics, but it does have (checked) dynamic casts

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

Dynamic Method Binding

- Virtual functions are the only thing that requires any trickiness (Figure 9.4)
  - 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 (Figure 10.5)
    - In the dispatch table, functions defined in the parent come first, though some of the pointers point to overridden versions
    - You could put the whole dispatch table in the object itself
      - That would save a little time, but potentially waste a LOT of space

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;
```

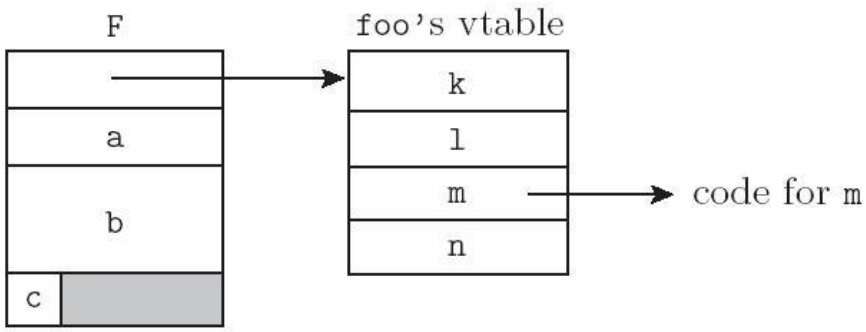


Figure 9.4: **Implementation of virtual methods.** The representation of object `F` begins with the address of the vtable for class `foo`. (All objects of this class will point to the same vtable.) The vtable itself consists of an array of addresses, one for the code of each virtual method of the class. 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;
```

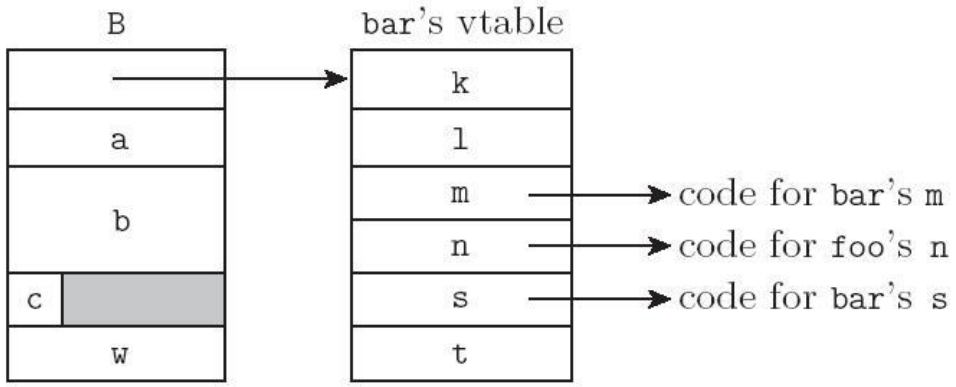


Figure 9.5: **Implementation of single inheritance.** As in Figure 9.4, the representation of object `B` begins with the address of its class's vtable. The first four entries in the table represent the same members as they do for `foo`, except that one—`m`—has been overridden and now contains the address 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 `bar`.

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
  - Of course 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

Historical Origins

- Church's model of computing is called the *lambda calculus*
  - based on the notion of parameterized expressions (with each parameter introduced by an occurrence of the letter λ—hence the notation's name.
  - Lambda calculus was the inspiration for functional programming
  - one uses it to compute by substituting parameters into expressions, just as one computes in a high level functional program by passing arguments to functions

Historical Origins

- Mathematicians established a distinction between
  - *constructive* proof (one that shows how to obtain a mathematical object with some desired property)
  - *nonconstructive* proof (one that merely shows that such an object must exist, e.g., by contradiction)
- Logic programming is tied to the notion of constructive proofs, but at a more abstract level
  - the logic programmer writes a set of *axioms* that allow the *computer* to discover a constructive proof for each particular set of inputs

Functional Programming Concepts

- Functional languages such as Lisp, Scheme, FP, ML, Miranda, and Haskell are an attempt to realize Church's lambda calculus in practical form as a programming language
- The key idea: do everything by composing functions
  - no mutable state
  - no side effects

Functional Programming Concepts

- Necessary features, many of which are missing in some imperative languages
  - 1st class and high-order functions
  - serious polymorphism
  - powerful list facilities
  - structured function returns
  - fully general aggregates
  - garbage collection

Functional Programming Concepts

- So how do you get anything done in a functional language?
  - Recursion (especially tail recursion) takes the place of iteration
  - In general, you can get the effect of a series of assignments

```
x := 0          ...
x := expr1      ...
x := expr2      ...
```

from f3(f2(f1(0))), where each f expects the value of x as an argument, f1 returns expr1, and f2 returns expr2

Functional Programming Concepts

- Recursion even does a nifty job of replacing looping

```
x := 0; i := 1; j := 100;
while i < j do
      x := x + i*j;
      i := i + 1;
      j := j - 1
end while
return x
```

becomes `f(0,1,100)`, where

```
f(x,i,j) == if i < j then
      f (x+i*j, i+1, j-1) else x
```

Functional Programming Concepts

- Thinking about recursion as a direct, mechanical replacement for iteration, however, is the wrong way to look at things
  - One has to get used to thinking in a recursive style
- Even more important than recursion is the notion of ***higher-order functions***
  - Take a function as argument, or return a function as a result
  - Great for building things

15.1 Logic and Horn Clauses

A Horn clause has a head h, which is a predicate, and a body, which is a list of predicates $p_1, p_2, \ldots, p_n$.

It is written as:

$$h \leftarrow p_1, p_2, \ldots, p_n$$

This means, "h is true only if $p_1, p_2, \ldots,$ and p_n are simultaneously true."

E.g., the Horn clause:

$$snowing(C) \leftarrow precipitation(C), freezing(C)$$

says, "it is snowing in city C only if there is precipitation in city C and it is freezing in city C."

Horn Clauses and Predicates

Any Horn clause

$$h \leftarrow p_1, p_2, \ldots, p_n$$

can be written as a predicate:

$$p_1 \wedge p_2 \wedge \ldots \wedge p_n \supset h$$

or equivalently:

$$\neg(p_1 \wedge p_2 \wedge \ldots \wedge p_n) \vee h$$

But not every predicate can be written as a Horn clause.

E.g., $literate(x) \supset reads(x) \vee writes(x)$

Resolution and Unification

If h is the head of a Horn clause

$h \leftarrow terms$

and it matches one of the terms of another Horn clause:

$t \leftarrow t_1, h, t_2$

then that term can be replaced by h's terms to form:

$t \leftarrow t_1, terms, t_2$

During resolution, assignment of variables to values is called *instantiation*.

Unification is a pattern-matching process that determines what particular instantiations can be made to variables during a series of resolutions.

15.2 Logic Programming in Prolog

In logic programming the program declares the goals of the computation, not the method for achieving them.

Logic programming has applications in AI and databases.

- *Natural language processing (NLP)*
- *Automated reasoning and theorem proving*
- *Expert systems (e.g., MYCIN)*
- *Database searching, as in SQL (Structured Query Language)*

Prolog emerged in the 1970s. Distinguishing features:

- *Nondeterminism*
- *Backtracking*

15.2.1 Prolog Program Elements

Prolog programs are made from *terms*, which can be:

- *Variables*
- *Constants*
- *Structures*

Variables begin with a capital letter, like Bob.

Constants are either integers, like 24, or atoms, like the, zebra, C , and .

Structures are predicates with arguments, like:

n(zebra), speaks(Y, English), and np(X, Y)

- The *arity* of a structure is its number of arguments (1, 2, and 2 for these examples).

Facts, Rules, and Programs

A Prolog *fact* is a Horn clause without a right-hand side. Its form is (note the required period .):

term.

A Prolog *rule* is a Horn clause with a right-hand side. Its form is (note :- represents ← and the required period .):

term :- $term_1$, $term_2$, ... $term_n$.

A Prolog *program* is a collection of facts and rules.

Searching for Success: Queries

A *query* is a fact or rule that initiates a search for success in a Prolog program. It specifies a search goal by naming variables that are of interest. E.g.,

```
?- speaks(Who, russian).
```

asks for an instantiation of the variable Who for which the query speaks(Who, russian) succeeds.

A program is loaded by the query consult, whose argument names the program. E.g.,

```
?- consult(speaks).
```

loads the program named speaks given on the previous slide

Answering the Query: Unification

To answer the query:

?- speaks(Who, russian).

Prolog considers every fact and rule whose head is speaks. (If more than one, consider them in order.)

Resolution and unification locate all the successes:

Who = allen ;

Who = mary ;

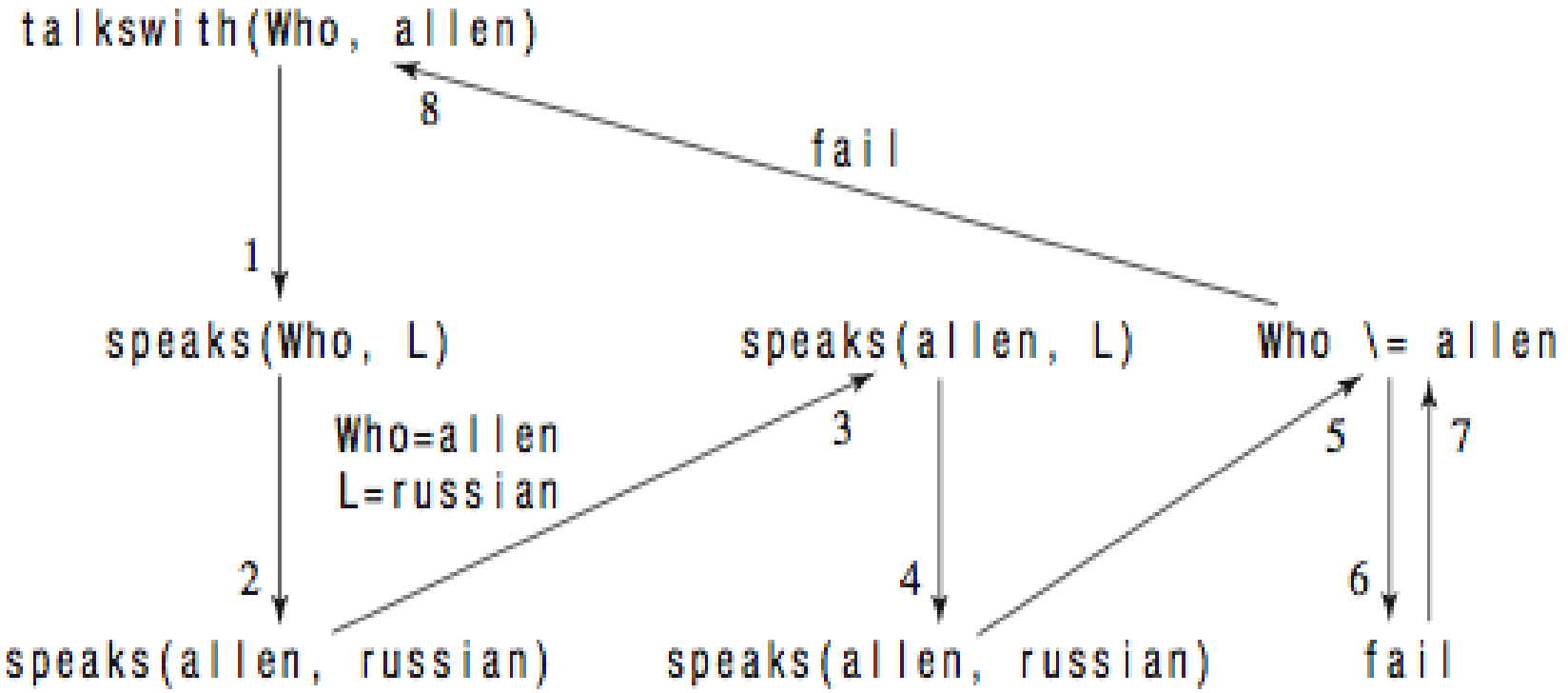
No

– *Each semicolon (;) asks, "Show me the next success."*

Search Trees

First attempt to satisfy the query ?- talkswith(Who, allen).

Fig 15.2



Prolog Program

Fig 15.3

```
mother(mary, sue).
mother(mary, bill).
mother(sue, nancy).
mother(sue, jeff).
mother(jane, ron).

father(john, sue).
father(john, bill).
father(bob, nancy).
father(bob, jeff).
father(bill, ron).

parent(A,B) :- father(A,B).
parent(A,B) :- mother(A,B).
grandparent(C,D) :- parent(C,E), parent(E,D).
```

Some Database Queries

Who are the parents of jeff?

```
?- parent(Who, jeff).
Who = bob;
Who = sue
```

Find all the grandparents of Ron.

```
?- grandparent(Who, ron).
```

What about siblings? Those are the pairs who have the same parents.

```
?- sibling(X, Y) :- parent(W, X), parent(W, Y), X\=Y.
```

Lists

A *list* is a series of terms separated by commas and enclosed in brackets.

- *The empty list is written* [].
- *The sentence "The giraffe dreams" can be written as a list:*
  [the, giraffe, dreams]
- *A "don't care" entry is signified by _, as in*
  [_, X, Y]
- *A list can also be written in the form:*
  [Head | Tail]
- *The functions*
  append *joins two lists, and*
  member *tests for list membership.*