Programming Languages

Sessions 7 & 8 – Main Theme
Program Structure
and
Object-Oriented Programming

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Adapted from course textbook resources
Programming Language Pragmatics (3rd Edition)
Michael L. Scott, Copyright © 2009 Elsevier

Agenda

1 **Session Overview**

2 **Program Structure**

3 **Object-Oriented Programming**

4 **Conclusion**

What is the course about?

- Course description and syllabus:
  - » http://www.nyu.edu/classes/jcf/CSCI-GA.2110-001
  - » http://www.cs.nyu.edu/courses/summer14/CSCI-GA.2110-001/index.html
- Textbook:

  - » ***Programming Language Pragmatics (3rd Edition)***
    Michael L. Scott
    Morgan Kaufmann
    ISBN-10: 0-12374-514-4, ISBN-13: 978-0-12374-514-4, (04/06/09)
- Additional References:
  - » Osinski, Lecture notes, Summer 2010
  - » Grimm, Lecture notes, Spring 2010
  - » Gottlieb, Lecture notes, Fall 2009
  - » Barrett, Lecture notes, Fall 2008

Session Agenda

- Session Overview
- Program Structure
- Object-Oriented Programming
- Conclusion

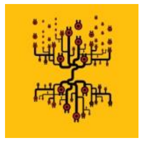
Icons / Metaphors

Information

Common Realization

Knowledge/Competency Pattern

Governance


Alignment

Solution Approach

Session 6 Review

- Data Types
  - Strong vs. Weak Typing
  - Static vs. Dynamic Typing
- Type Systems
  - Type Declarations
- Type Checking
  - Type Equivalence
  - Type Inference
  - Subtypes and Derived Types
- Scalar and Composite Types
  - Records, Variant Records, Arrays, Strings, Sets
- Pointers and References
  - Pointers and Recursive Types
- Function Types
- Files and Input / Output
- Conclusions

Agenda

1 Session Overview

2 Program Structure

3 Object-Oriented Programming

4 Conclusion

Program Structure

- Key Concepts
  - » Modules
  - » Packages
  - » Interfaces
  - » Abstract types and information hiding
- Review Session 2
  - » Textbook Sections 3.3.4, 3.3.5, 3.7

Software Complexity

- Tony Hoare:
  - » *here are two ways of constructing a software design: one way is to make it so simple that there are* obviously *no deficiencies, and the other is to make it so complicated that there are no* obvious *deficiencies.*
- Edsger Dijkstra:
  - » *Computing is the only profession in which a single mind is obliged to span the distance from a bit to a few hundred megabytes, a ratio of* 1 *to* 10^9*, or nine orders of magnitude. Compared to that number of semantic levels, the average mathematical theory is almost flat. By evoking the need for deep conceptual hierarchies, the automatic computer confronts us with a radically new intellectual challenge that has no precedent in our history.*
- Steve McConnell:
  - » *Software's Primary Technical Imperative has to be managing complexity.*

Dealing with Complexity

- *Problem Decomposition:* Minimize the amount of essential complexity that has to be dealt with at any one time. In most cases, this is the *top priority*
- *Information Hiding:* Encapsulate complexity so that it is not accessible outside of a small part of the program
  - » Additional benefits of information hiding:
    - Reduces risk of name conflicts
    - Safeguards integrity of data
    - Helps to compartmentalize run-time errors

Modules

- Programs are built out of components
- Each component:
  - » has a public interface that defines entities exported by the component
  - » may depend on the entities defined in the interface of another component (weak external coupling)
  - » may include other (private) entities that are not exported
  - » should define a set of logically related entities (strong internal coupling)
  - » “Strong (internal) cohesion – Low (external) coupling”
- We call these components modules

Modules - Definition

- Different languages use different terms
  - » different languages have different semantics for this construct (sometimes very different)
  - » a module is somewhat like a record, but with an important distinction:
    - record => consists of a set of names called fields, which refer to values in the record
    - module => consists of a set of names, which can refer to values, types, routines, other language-specific entities, and possibly other modules
- Note that the similarity is between a record and a module, not a record type and a module

Language Constructs for Modularity

- Issues:
  - public interface
  - private implementation
  - dependencies between modules
  - naming conventions of imported entities
  - relationship between modules and files
- Language Choices
  - Ada : package declaration and body, with and use clauses, renamings
  - C : header files, #include directives
  - C++ : header files, #include directives, namespaces, using declarations/directives, namespace alias definitions
  - Java : packages, import statements
  - ML : signature, structure and functor definitions

Ada Packages

```
package Queues is
  Size: constant Integer := 1000;

  type Queue is private; -- information hiding

  procedure Enqueue (Q: in out Queue, Elem: Integer);
  procedure Dequeue (Q: in out Queue; Elem: out Integer);
  function Empty (Q: Queue) return Boolean;
  function Full (Q: Queue) return Boolean;
  function Slack (Q: Queue) return Integer;
  -- overloaded operator "=":
  function "=" (Q1, Q2: Queue) return Boolean;

private
  ... -- concern of implementation, not of package client
end Queues;
```

Private Parts and Information Hiding

```
package Queues is
  ... -- visible declarations
private
  type Storage is
    array (Integer range <>) of Integer;
  type Queue is record
    Front: Integer := 0; -- next elem to remove
    Back: Integer := 0;  -- next available slot
    Contents: Storage (0 .. Size-1); -- actual contents
    Num: Integer := 0;
  end record;
end Queues;
```

Implementation of Queues

```
package body Queues is
  procedure Enqueue (Q: in out Queue;
                     Elem: Integer) is
  begin
    if Full(Q) then
      -- need to signal error: raise exception
    else
      Q.Contents(Q.Back) := Elem;
    end if;
    Q.Num := Q.Num + 1;
    Q.Back := (Q.Back + 1) mod Size;
  end Enqueue;
```

Predicates on Queues

```
function Empty (Q: Queue) return Boolean is
begin
  return Q.Num = 0;      -- client cannot access
                         --   Num directly
end Empty;

function Full (Q: Queue) return Boolean is
begin
  return Q.Num = Size;
end Full;

function Slack (Q: Queue) return Integer is
begin
  return Size - Q.Num;
end Slack;
```

Operator Overloading

```
function "=" (Q1, Q2 : Queue) return Boolean is
begin
  if Q1.Num /= Q2.Num then
    return False;
  else
    for J in 1 .. Q1.Num loop
      -- check corresponding elements
      if Q1.Contents((Q1.Front + J - 1) mod Size) /=
         Q2.Contents((Q2.Front + J - 1) mod Size)
      then
        return False;
      end if;
    end loop;
    return True; -- all elements are equal
  end if;
end "=";    -- operator "/=" implicitly defined
            --   as negation of "="
```

Client Can Only Use Visible Interface

```
with Queues;  use Queues;  with Text_IO;

procedure Test is
  Q1, Q2: Queue; -- local objects of a private type
  Val : Integer;
begin
  Enqueue(Q1, 200); -- visible operation
  for J in 1 .. 25 loop
    Enqueue(Q1, J);
    Enqueue(Q2, J);
  end loop;
  Deqeue(Q1, Val); -- visible operation
  if Q1 /= Q2 then
    Text_IO.Put_Line("lousy␣implementation");
  end if;
end Test;
```

Notes: The “use” keyword specifies that a function name which cannot be resolved locally should be searched for in this library. “with” is approximately equal to “#include”: in the above example, it means that you want to work with the functions available in the “Ada.Text_IO” package. The rest is pretty straightforward: you want to put out the text “lousy implementation”, and the Put_Line function you are interested in is the one in Ada.Text_IO.

Implementation

- package body holds bodies of subprograms that implement interface
- package may not require a body:

```
package Days is
   type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

   subtype Weekday is Day range Mon .. Fri;

   Tomorrow: constant array (Day) of Day
       := (Tue, Wed, Thu, Fri, Sat, Sun, Mon);

   Next_Work_Day: constant array (Weekday) of Weekday
       := (Tue, Wed, Thu, Fri, Mon);
end Days;
```

Syntactic Sugar: Use and Renames

- Visible entities can be denoted with an expanded name:

```
with Text_IO ;
...
Text_IO . Put_Line (" hello ");
```

- Use clause makes name of entity directly usable:

```
with Text_IO ; use Text_IO ;
...
Put_Line (" hello ");
```

- Renames clause makes name of entity more manageable:

```
with Text_IO ;
package T renames Text_IO ;
...
T. Put_Line (" hello ");
```

Sugar Can Be Indispensable

```
with Queues ;
procedure Test is
    Q1, Q2: Queues.Queue ;
begin
    if Q1 = Q2 then ...
     -- error : "=" is not directly visible
     -- must write instead : Queues ."="( Q1 , Q2)
```

Two solutions:

- import all entities:
  - » use Queues ;
- import operators only:
  - » use type Queues.Queue ;

C++ Namespaces

- Late addition to the language
- an entity requires one or more declarations and a single definition
- A namespace declaration can contain both, but definitions may also be given separately

```
// in .h file
namespace util {
  int f (int); /* declaration of f */
}

// in .cpp file
namespace util {
  int f (int i) {
    // definition provides body of function
    ...
  }
}
```

Dependencies Between Modules in C++

- Files have semantic significance: #include directives means textual substitution of one file in another
- Convention is to use header files for shared interfaces

```
#include <iostream> // import declarations

int main () {
  std::cout << "C++␣is␣really␣different"
            << std::endl;
  return 0;
}
```

Header Files Are Visible Interfaces

```
namespace stack {  // in file stack.h
  void push (char);
  char pop ();
}
```

```
#include "stack.h"  // import into client file

void f () {
  stack::push('c');
  if (stack::pop() != 'c') error("impossible");
}
```

Namespace Definitions

```
#include "stack.h" // import declarations

namespace stack {  // the definition
  const unsigned int MaxSize = 200;
  char v[MaxSize];
  unsigned int numElems = 0;

  void push (char c) {
    if (numElems >= MaxSize)
      throw std::out_of_range("stack␣overflow");
    v[numElems++] = c;
  }

  char pop () {
    if (numElems == 0)
      throw std::out_of_range("stack␣underflow");
    return v[--numElems];
  }
}
```

Syntactic Sugar Using Declarations

```
namespace queue { // works on single queue
  void enqueue (int);
  int dequeue ();
}
```

```
#include "queue.h"  // in client file

using queue::dequeue;  // selective: a single entity

void f () {
  queue::enqueue(10);  // prefix needed for enqueue
  queue::enqueue(-999);
  if (dequeue() != 10)  // but not for dequeue
    error("buggy␣implementation");
}
```

Wholesale Import: The Using Directive

```
#include "queue.h"  // in client file

using namespace queue;  // import everything

void f () {
  enqueue(10);  // prefix not needed
  enqueue(-999);
  if (dequeue() != 10)  // for anything
    error("buggy␣implementation");
}
```

Shortening Names

- Sometimes, we want to qualify names, but with a shorter name.
- In Ada:

```
package PN renames A.Very_Long.Package_Name;
```

- In C++:

```
namespace pn = a::very_long::package_name;
```

- We can now use PN as the qualifier instead of the long name.

Visibility: Koenig Lookup

- When an unqualified name is used as the postfix-expression in a function call (expr.call), other namespaces not considered during the usual unqualified look up (basic.lookup.unqual) may be searched; this search depends on the types of the arguments.
- For each argument type T in the function call, there is a set of zero or more associated namespaces to be considered
  - » The set of namespaces is determined entirely by the types of the function arguments. typedef names used to specify the types do not contribute to this set

Koenig Lookup Details

- The set of namespaces are determined in the following way:
  - » If T is a fundamental type, its associated set of namespaces is empty.
  - » If T is a class type, its associated namespaces are the namespaces in which the class and its direct and indirect base classes are defined.
  - » If T is a union or enumeration type, its associated namespace is the namespace in which it is defined.
  - » If T is a pointer to U, a reference to U, or an array of U, its associated namespaces are the namespaces associated with U.
  - » If T is a pointer to function type, its associated namespaces are the namespaces associated with the function parameter types and the namespaces associated with the return type. [recursive]

Koenig Lookup

Example

```
namespace NS
{
   class A {};
   void f( A ) {}
}

int main()
{
   NS::A a;
   f( a );     //calls NS::f
}
```

Example

```
#include<iostream>

int main()
{
  // Where does operator<<() come from?
  std::cout << "Hello,␣World" << std::endl;
  return 0;
}
```

Linking

- An external declaration for a variable indicates that the entity is defined elsewhere

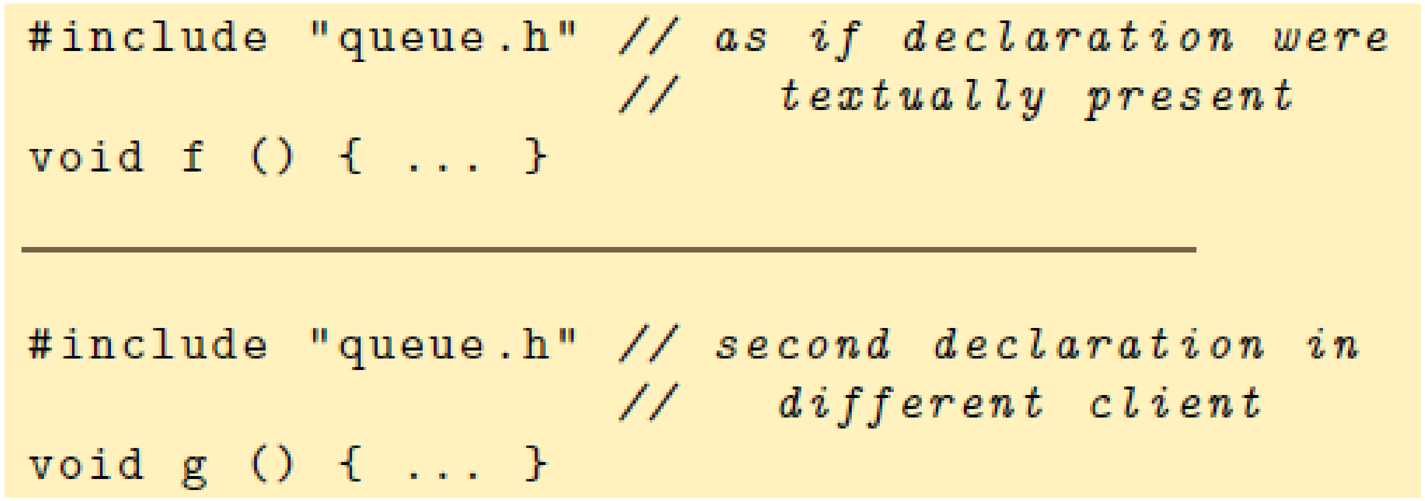
```
extern int x; // will be found later
```

- A function declaration indicates that the body is defined elsewhere
- Multiple declarations may denote the same entity

```
extern int x; // in some other file
```

- An entity can only be defined once
- Missing/multiple definitions cannot be detected by the compiler: link-time errors

Include Directives = Multiple Declarations



```
#include "queue.h" // as if declaration were
                   //    textually present
void f () { ... }
```

```
#include "queue.h" // second declaration in
                   //    different client
void g () { ... }
```

- Definitions are legal if textually identical (but compiler can’t check!)
- Headers are safer than cut-and-paste, but not as good as a proper module system

Modules in Java

- Package structure parallels file system
- A package corresponds to a directory
- A class is compiled into a separate object file
- Each class declares the package in which it appears (open structure)

```
package polynomials;
class poly {
  ... // in file .../alg/polynomials/poly.java
}
```

```
package polynomials;
class iterator {
  ... // in file .../alg/polynomials/iterator.java
}
```

Default: anonymous package in current directory

Dependencies Between Classes

- Dependencies indicated with import statements:

```
import java.awt.Rectangle; // declared in java.awt

import java.awt.*;     // import all classes
                       //    in package
```

- No syntactic sugar across packages: use expanded names
- None needed in same package: all classes in package are directly visible to each other

Modules in ML

- There are three entities:
  - » signature : an interface
  - » structure : an implementation
  - » functor : a parameterized structure
- A structure implements a signature if it defines everything mentioned in the signature (in the correct way)

ML Signature

- An ML signature specifies an interface for a module

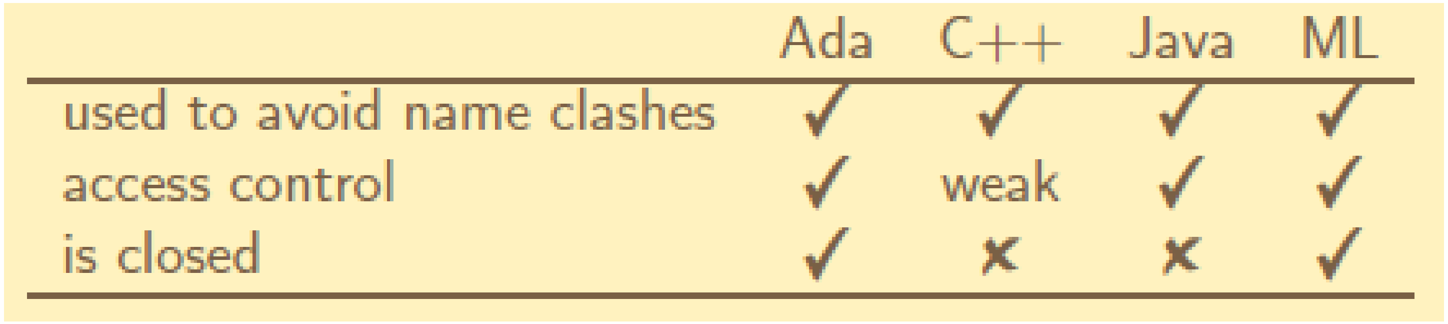
```
signature STACKS =
sig
    type stack
    exception Underflow
    val empty : stack
    val push : char * stack -> stack
    val pop   : stack -> char * stack
    val isEmpty : stack -> bool
end
```

ML Structure

- A structure provides an implementation

```
structure Stacks : STACKS =
struct
    type stack = char list
    exception Underflow
    val empty = [ ]
    val push = op::
    fun pop (c::cs) = (c, cs)
      | pop []      = raise Underflow
    fun isEmpty [] = true
      | isEmpty _  = false
end
```

Comparison



	Ada	C++	Java	ML
used to avoid name clashes	✓	✓	✓	✓
access control	✓	weak	✓	✓
is closed	✓	✗	✗	✓

- Relation between interface and implementation:
  - Ada :
    - one package (interface) , one package body
  - ML :
    - one signature can be implemented by many structures
    - one structure can implement many signatures

Agenda

Object-Oriented Data Types and Representation

- Key Concepts
  - » Objects
  - » Classes
- Review Session 6
  - » Textbook Section 7.7

What is OOP ? (Part I)

- The *object* idea:
  - » bundling of data (*data members*) and operations (*methods*) on that data
  - » restricting access to the data
- An object contains:
  - » data members : arranged as a set of named fields
  - » methods : routines which take the object they are associated with as an argument (known as *member functions* in C++)
  - » constructors : routines which create a new object
- A class is a construct which defines the data, methods and constructors associated with all of its instances (objects)

What is OOP ? (Part II)

- The *inheritance* and *dynamic binding* ideas:
  - » classes can be extended (*inheritance*):
    - by adding new fields
    - by adding new methods
    - by *overriding* existing methods (changing behavior)

    If class B extends class A, we say that B is a *subclass* or *derived* class of A, and A is a *superclass* or *base* class of B
  - » dynamic binding : wherever an instance of a class is required, we can also use an instance of any of its subclasses; when we call one of its methods, the overridden versions are used
  - » There should be an *is-a* relationship between a derived class and its base class

Styles of OOLs

- In class-based OOLs, each object is an instance of a class (Java, C++, C#, Ada95, Smalltalk, OCaml, etc.)
- In prototype-based OOLS, each object is a clone of another object, possibly with modifications and/or additions (Self, Javascript)

Other Common OOP Features

- Multiple inheritance
  - C++
  - Java (of interfaces only)
  - problem: how to handle diamond shaped inheritance hierarchy
- Classes often provide package-like capabilities:
  - visibility control
  - ability to define types and classes in addition to data fields and methods

Object-Oriented Programming

- Control or PROCESS abstraction is a very old idea (subroutines!), though few languages provide it in a truly general form (Scheme comes close)
- 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

Object-Oriented Programming

- 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

Object-Oriented Programming

- We talked about data abstraction some back in the session on 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

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

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
- C++ base classes can also be public, private, or protected

Encapsulations 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*

Encapsulation and Inheritance – Classes (C++)

- Disadvantage of the module-as-manager approach: include explicit create/initialize & destroy/finalize routines for every abstraction
  - Even w/o dynamic allocation inside module, users don't have necessary knowledge to do initialization
  - Ada 83 is a little better here: you can provide initializers for pieces of private types, but this is NOT a general approach
  - Object-oriented languages often give you constructors and maybe destructors
    - Destructors are important primarily in the absence of garbage collection

Encapsulations and Inheritance – Classes (C++)

- A few C++ features you may not have learned:
  - » classes as members

    ```
    foo::foo (args0) : member1
    (args1), member2 (args2) { ...
    ```

    args1 and args2 need to be specified in terms of args0
    - The reason these things end up in the header of foo is that they get executed *before* foo's constructor does, and the designers consider it good style to make that clear in the header of foo::foo

Encapsulations and Inheritance – Classes (C++)

- A few C++ features (2):
  - initialization v. assignment

```
foo::operator=(&foo) v.
foo::foo(&foo)
     foo b;
     foo f = b;
         // calls constructor
     foo b, f;
         // calls no-argument constructor
     f = b;
         // calls operator=
```

Encapsulations and Inheritance – Classes (C++)

- A few C++ features (3):
  - virtual functions (see the next dynamic method binding section for details):
    Key question: if child is derived from parent and I have a parent* p (or a parent& p) that points (refers) to an object that's actually a child, what member function do I get when I call p->f (p.f)?
    - Normally I get p's f, because p's type is parent*.
    - But if f is a virtual function, I get c's f.

Encapsulations and Inheritance – Classes (C++)

- A few C++ features (4):
  - virtual functions (continued)
    - 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 declare 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)
    - multiple inheritance in Standard C++ (see next)
  - friends
    - functions
    - classes

Initialization and Finalization

- Texbook’s section 3.2 defines 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*
    - 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

Initialization and Finalization

Issues:

- choosing a constructor
- 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++, the constructors for any base classes will be executed before the constructor for the derived class
- garbage collection

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
- Generics were added to Java in 2004 and are implemented as (checked) dynamic casts
  - » http://www.jelovic.com/articles/why_java_is_slow.htm

Dynamic Method Binding – More on Generics

```
// Removes 4-letter words from c. Elements must be strings
// (using inconvenient/unsafe casting to the type of element that is stored
    in the collection)
static void expurgate(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); )
        if (((String) i.next()).length() == 4)
            i.remove();
}

// Removes the 4-letter words from c
// (using clear and safe code based on generics that eliminates an unsafe
    cast and a number of extra parentheses)
static void expurgate(Collection<String> c) {
    for (Iterator<String> i = c.iterator(); i.hasNext(); )    if (i.next().length() == 4)
i.remove();
}
```

Note: “Collection<String> c” above reads as “Collection of String c”

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 (see next slide)
  - » 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
    - 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;
```

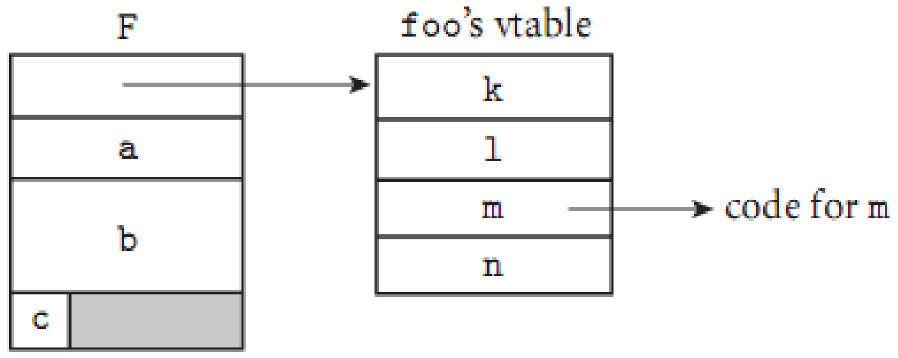


Figure 9.3 Implementation of virtual methods. The representation of object F begins with the address of the vtable for class foo. (All objects of this class will point to the same vtable.) The vtable itself consists of an array of addresses, one for the code of each virtual method of the class. The remainder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

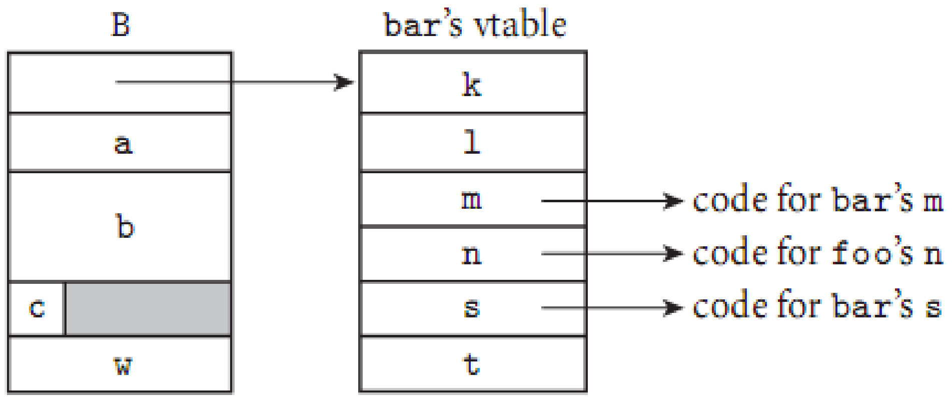


Figure 9.4 Implementation of single inheritance. As in Figure 9.3, the representation of object B begins with the address of its class's vtable. The first four entries in the table represent the same members as they do for foo, except that one —m— has been overridden and now contains the address of the 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
  - » 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

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

- 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 - Java

- An imperative language (like C++, Ada, C, Pascal)
- is interpreted (like Scheme, APL)
- is garbage-collected (like Scheme, ML, Smalltalk, Eiffel, Modula-3)
- can be compiled
- is object-oriented (like Eiffel, more so than C++, Ada)
- a successful hybrid for a specific-application domain
- a reasonable general-purpose language for non-real-time applications

- Work in progress: language continues to evolve
- C# is latest, incompatible variant

Object-Oriented Programming - Java

- Original Design Goals (1993 White Paper):
  - simple
  - object-oriented (inheritance, polymorphism)
  - distributed
  - interpreted
  - multi-threaded
  - robust
  - secure
  - architecture-neutral
- Obviously, “simple” was dropped

Object-Oriented Programming - Java

- Portability:
  - » Critical concern: write once – run everywhere
  - » Consequences:
    - portable interpreter
    - definition through virtual machine: the JVM
    - run-time representation has high-level semantics
    - supports dynamic loading
    - high-level representation can be queried at run-time to provide reflection
    - dynamic features make it hard to fully compile, safety requires numerous run-time checks

Object-Oriented Programming - Java

- Contrast with Conventional Systems Languages
  - » Conventional imperative languages are fully compiled:
    - run-time structure is machine language
    - minimal run-time type information
    - language provides low-level tools for accessing storage
    - safety requires fewer run-time checks because compiler (least for Ada and somewhat for C++) can verify correctness statically
    - languages require static binding, run-time image cannot be easily modified
    - different compilers may create portability problems
  - » Notable omissions:
    - no operator overloading (syntactic annoyance)
    - no separation of specification and body
    - no enumerations until latest language release
    - no generic facilities until latest language release

Object-Oriented Programming - Java

- Statements:
  - Most statements are like their C counterparts:
    - switch (including C’s falling through behavior)
    - for
    - if
    - while
    - do ... while
    - break and continue
      - Java also has *labeled* versions of break and continue, like Ada.
    - return
    - Java has no goto!

Object-Oriented Programming – The Simplest Java Program

```
class HelloWorld {
  public static void main (String[] args) {
    System.out.println("Hello,␣world");
  }
}
```

Object-Oriented Programming – Classes in Java

- Encapsulation of type and related operations

```
class Point {
  private double x, y;  // private data members

  public Point (double x, double y) { // constructor
    this.x = x;   this.y = y;
  }

  public void move (double dx, double dy) {
    x += dx;  y += dy;
  }

  public double distance (Point p) {
    double xdist = x - p.x, ydist = y - p.y;
    return Math.sqrt(xdist * xdist + ydist * ydist);
  }

  public void display () { ... }
}
```

Object-Oriented Programming – Extending a Class in Java

```
class ColoredPoint extends Point {
  private Color color;

  public ColoredPoint (double x, double y,
                       Color c) {
    super(x, y);
    color = c;
  }

  public ColoredPoint (Color c) {
    super(0.0, 0.0);
    color = c;
  }

  public Color getColor () { return color; }

  public void display () { ... }  // now in color!
}
```

Object-Oriented Programming – Dynamic Dispatching in Java

```
Point p1 = new Point(2.0, 3.0);
ColoredPoint cp1 = new ColoredPoint(2.0, 3.0, Blue);

Point p2 = p1;               // OK
Point p3 = cp1;              // OK

ColoredPoint cp2 = cp1;      // OK
ColoredPoint cp3 = p1;       // Error

cp1.move(1.0, 1.0);   // cp1 and p3 affected

p1.display();    // Point's display
cp1.display();   // ColoredPoint's display
p3.display();    // ColoredPoint's display
```

Object-Oriented Programming – Java Interfaces

- A Java interface allows otherwise unrelated classes to satisfy a given requirement
- This is orthogonal to inheritance.
  - inheritance: an A *is-a* B (has the attributes of a B, and possibly others)
  - interface: an A *can-do* X (and possibly other unrelated actions)
  - interfaces are a better model for multiple inheritance
- See textbook section 9.4.3 for implementation details

Object-Oriented Programming – Java Interface Comparable

```
public interface Comparable {
  public int CompareTo (Object x) throws
    ClassCastException;
  // returns -1 if this < x,
  //          0 if this = x,
  //         +1 if this > x
};

// Implementation needs to cast x to the proper class.

// Any class that may appear in a container should
//  implement Comparable, so the container can support
//  sorting.
```

Object-Oriented Programming – C++

- 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 friends
  - » Has static type checking

Object-Oriented Programming – Classes in C++

- The same classes, translated into C++:

```
class Point {
  double m_x, m_y;   // private data members

public:

  Point (double x, double y)   // constructor
    : m_x(x), m_y(y) { }

  virtual ~Point () { }

  virtual void move (double dx, double dy) {
    m_x += dx;  m_y += dy;
  }

  virtual double distance (const Point& p) {
    double xdist = m_x - p.m_x, ydist = m_y - p.m_y;
    return sqrt(xdist * xdist + ydist * ydist);
  }

  virtual void display () { ... }
};
```

Object-Oriented Programming – Extending a Class in C++

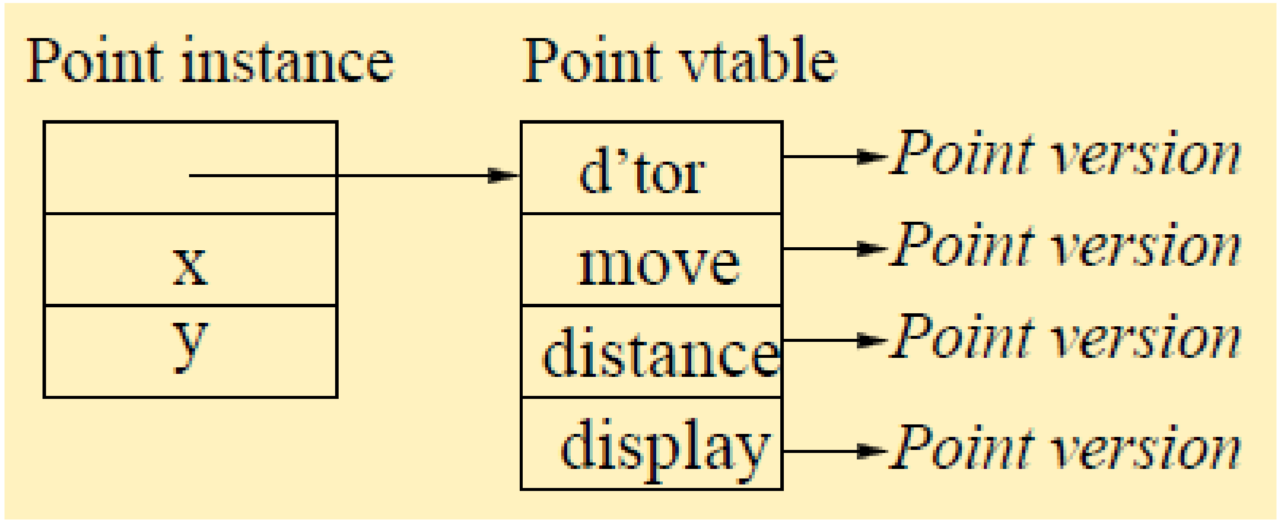
```
class ColoredPoint : public Point {
  Color color;

public:

  ColoredPoint (double x, double y,
                Color c) : Point(x, y), color(c) {
    color = c;
  }

  ColoredPoint (Color c) : Point(0.0, 0.0), color(c) { }

  virtual Color getColor () { return color; }

  virtual void display () { ... }  // now in color!
};
```

Object-Oriented Programming – Dynamic Dispatching in C++

```
Point *p1 = new Point(2.0, 3.0);
ColoredPoint *cp1 = new ColoredPoint(2.0, 3.0, Blue);

Point *p2 = p1;              // OK
Point *p3 = cp1;             // OK

ColoredPoint *cp2 = cp1;     // OK
ColoredPoint *cp3 = p1;      // Error

cp1->move(1.0, 1.0);   // cp1 and p3 affected

p1->display();    // Point's display
cp1->display();   // ColoredPoint's display
p3->display();    // ColoredPoint's display
```

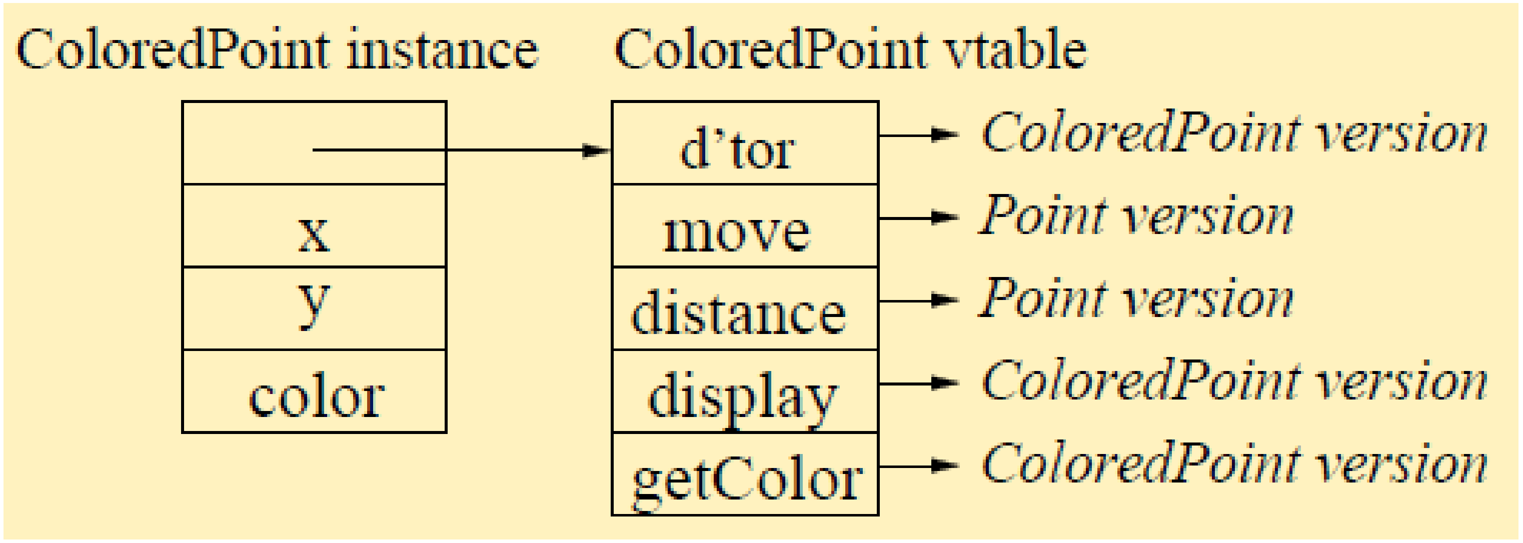
Object-Oriented Programming – C++ Implementation of a Class

- A typical implementation of a class in C++; using Point as an example:



Object-Oriented Programming – Extended Vtable in C++

- For ColoredPoint, we have:

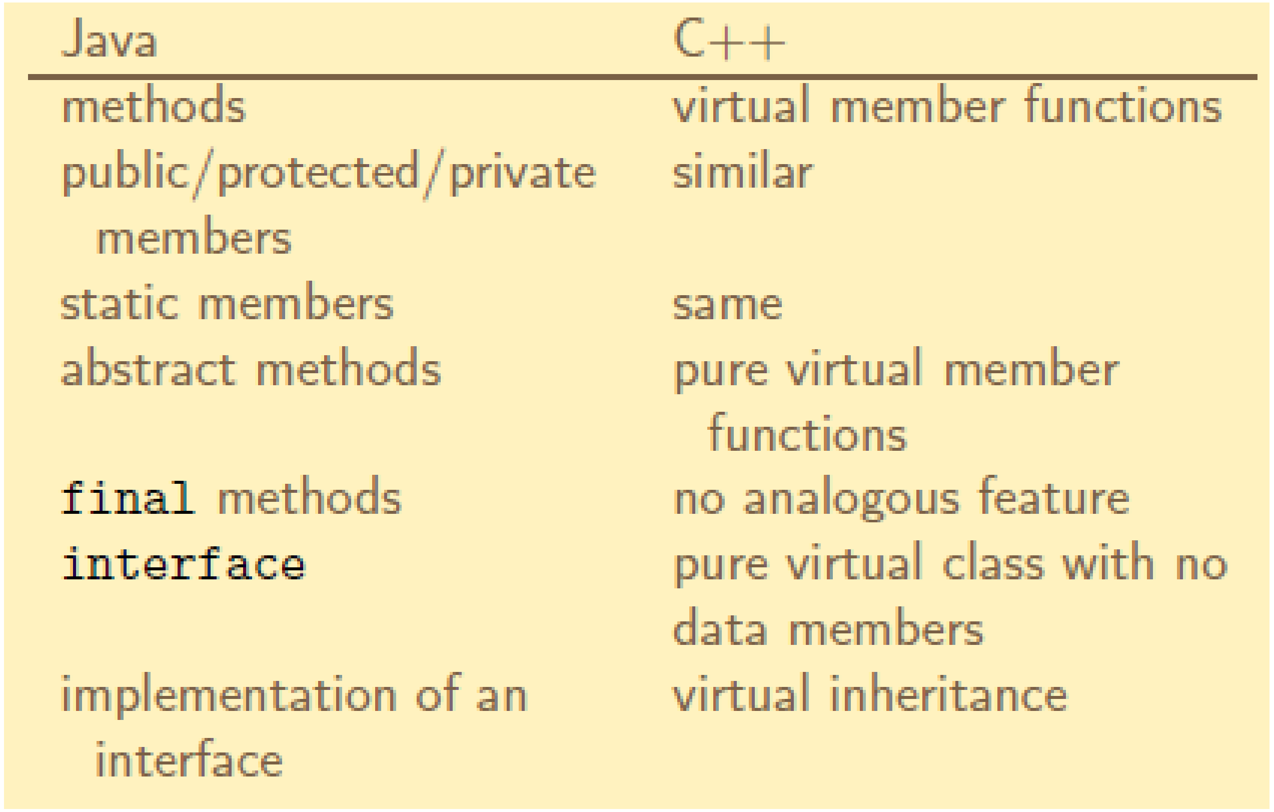


- Non-virtual member functions are never put in the vtable

Object-Oriented Programming – Method Modifiers in C++

- Access modifiers:
  - public
  - protected
  - package
  - private
- abstract
- static
- final
- synchronized
- native
- strictfp (strict floating point)

Object-Oriented Programming – Java and C++ Comparison



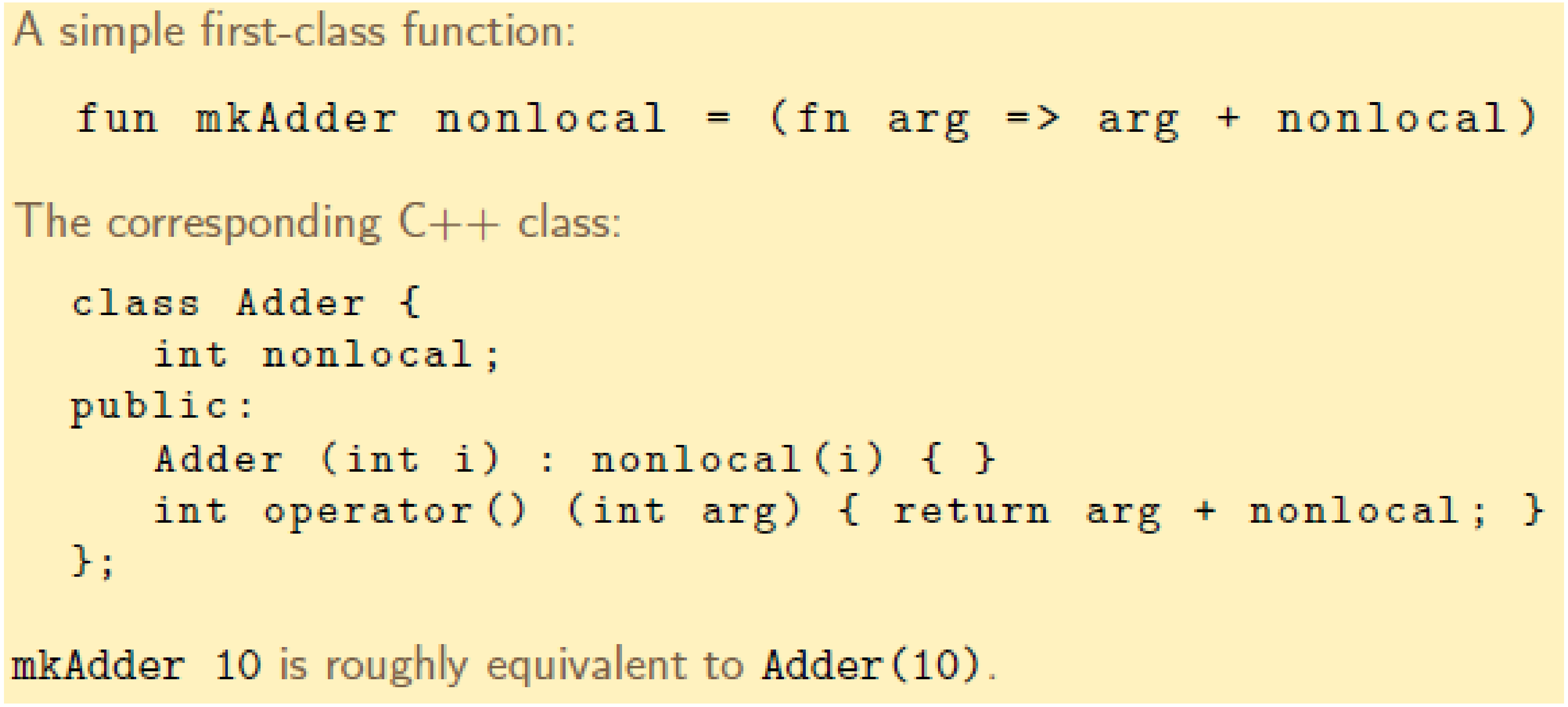
Java	C++
methods	virtual member functions
public/protected/private members	similar
static members	same
abstract methods	pure virtual member functions
`final` methods	no analogous feature
`interface`	pure virtual class with no data members
implementation of an interface	virtual inheritance

Object-Oriented Programming

- In the same category of questions related to classifying languages:
  - » 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

Object-Oriented Programming – First-Class Functions

- Simulating a first-class function with an object:



A simple first-class function:

```
fun mkAdder nonlocal = (fn arg => arg + nonlocal)
```

The corresponding C++ class:

```
class Adder {
    int nonlocal;
public:
    Adder (int i) : nonlocal(i) { }
    int operator() (int arg) { return arg + nonlocal; }
};
```

`mkAdder 10` is roughly equivalent to `Adder(10)`.

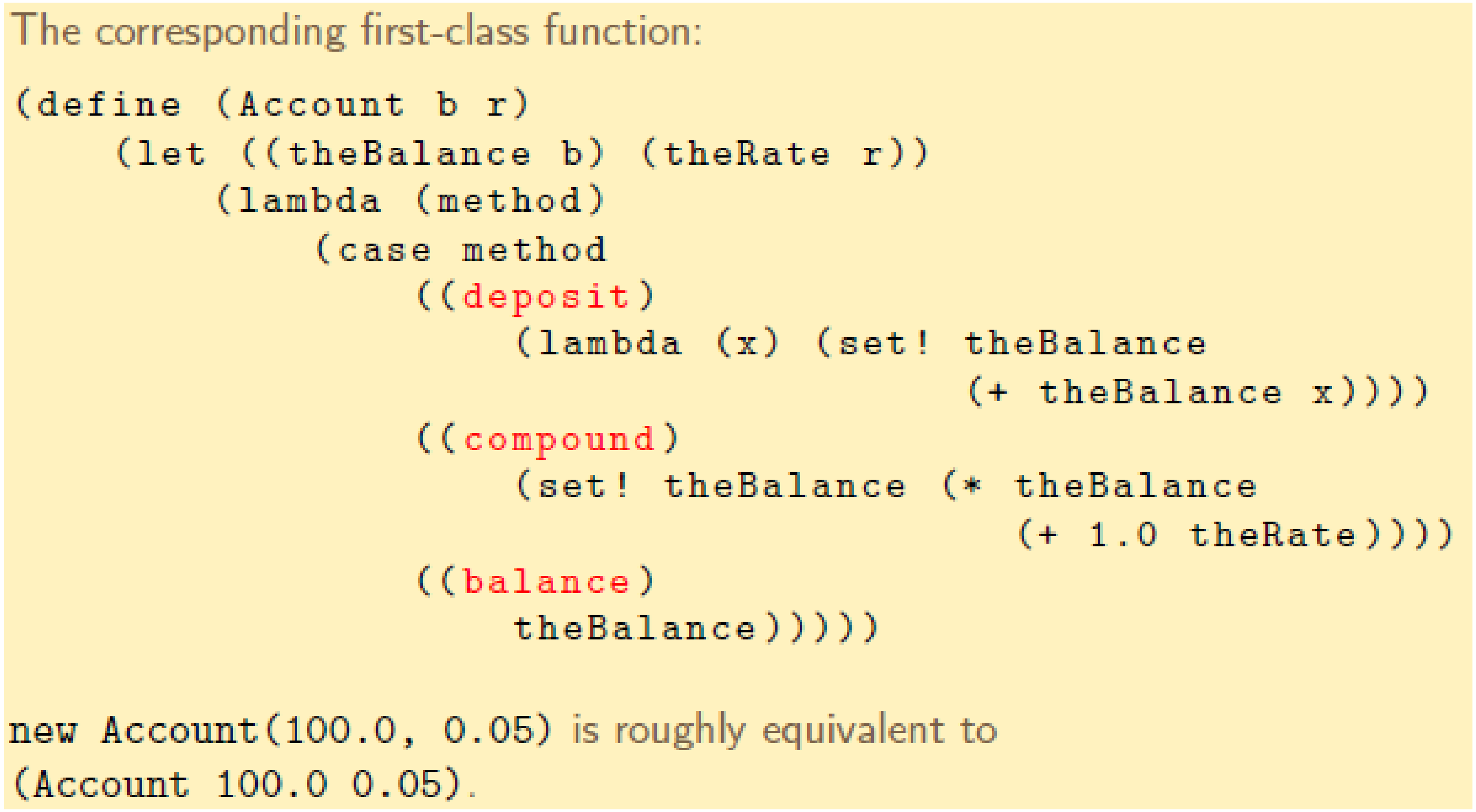
Object-Oriented Programming – First-Class Functions

- A simple unsuspecting object (in Java, for variety):

```
class Account {
   private float theBalance;
   private float theRate;

   Account (float b, float r) { theBalance = b;
                                theRate = r; }

   public void deposit (float x) {
     theBalance = theBalance + x;
   }
   public void compound () {
     theBalance = theBalance * (1.0 + rate);
   }
   public float balance () { return theBalance; }
}
```

Object-Oriented Programming – First-Class Functions

- The corresponding first-class function:



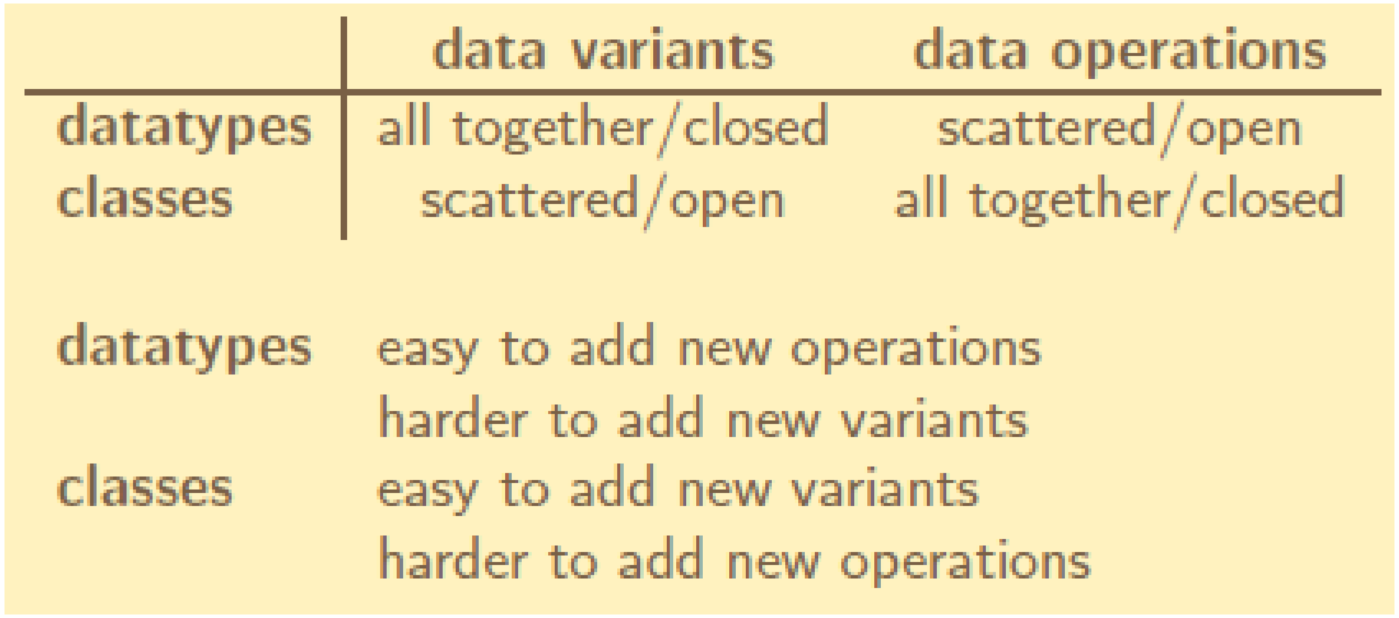
The corresponding first-class function:

```
(define (Account b r)
    (let ((theBalance b) (theRate r))
        (lambda (method)
            (case method
                ((deposit)
                    (lambda (x) (set! theBalance
                                      (+ theBalance x))))
                ((compound)
                    (set! theBalance (* theBalance
                                        (+ 1.0 theRate))))
                ((balance)
                    theBalance)))))
```

`new Account(100.0, 0.05)` is roughly equivalent to `(Account 100.0 0.05)`.

Object-Oriented Programming – ML Data Types vs. Inheritance

- ML datatypes and OO inheritance organize data and routines in orthogonal ways:



	data variants	data operations
datatypes	all together/closed	scattered/open
classes	scattered/open	all together/closed

datatypes	easy to add new operations
	harder to add new variants
classes	easy to add new variants
	harder to add new operations

Object-Oriented Programming – OOP Pitfalls

- A couple of facts:
  - In mathematics, an ellipse (from the Greek for absence) is a curve where the sum of the distances from any point on the curve to two fixed points is constant. The two fixed points are called foci (plural of focus).

    from http://en.wikipedia.org/wiki/Ellipse
  - A circle is a special kind of ellipse, where the two foci are the same point.
- If we need to model circles and ellipses using OOP, what happens if we have class Circle inherit from class Ellipse?

Object-Oriented Programming – OOP Pitfalls

```
class Ellipse {
   ...

   public move (double dx, double dy) { ... }

   public resize (double x, double y) { ... }
}

class Circle extends Ellipse {
   ...

   public resize (double x, double y) { ??? }
}
```

We can't implement a resize for `Circle` that lets us make it asymmetric!

Object-Oriented Programming – OOP Pitfalls

- In Java, if class B is a subclass of class A, then Java considers “array of B” to be a subclass of “array of A”:

```
class A { ... }
class B extends A { ... }

B[] b = new B[5];
A[] a = b;          // allowed (a and b are now aliases)

a[1] = new A();  // Bzzzt!  (Type error)
```

- The problem is that arrays are *mutable*; they allow us to replace an element with a different element.

Agenda

1 Session Overview

2 Program Structure

3 Object-Oriented Programming

4 Conclusion

Assignments & Readings

- Readings

  - Chapter Section 9
- Programming Assignment #3
  - TBA
  - Due on: TBA

Next Session: Control Abstractions and Concurrency

Appendix – Quick Survey of Various Languages

FORTRAN

- The earliest algorithmic language (50’s)
- Invented the idea of a*b+c*d
- Multi-dimensional arrays
- Subprograms (but no recursion)
- Separate and independent compilation
- Control structures depend heavily on goto

FORTRAN, an oddity

- One oddity in Fortran, no separation between tokens, blanks ignored
- Following are equivalent
  - DO 10 I = 1,100
  - DO10I=1,100
- More diabolical
  - DO10I = 1.10
    DO 10 I = 1.10

FORTRAN, later history

- FORTRAN-66
  - » First standardized version
- Fortran-77
  - » Updated, but no very major changes
- Fortran-90
  - » Big language with lots of extensions
  - » Including comprehensive aggregates and slicing notations.
- HPF (high performance Fortran)
  - » Parallelization constructs

Algol-60

- European contemporary of Fortran
  - » A bit later
- Designed then implemented
  - » Algol-60 report
- Features
  - » Structured programming (if, loop)
  - » Beginnings of a type system
  - » Recursive procedures
  - » I/O etc provided by library procedures

Algol-60 Call by Name

- Call by name means that an expression is passed and then evaluated with current values of variables when referenced.
- Jensen's device
  - » Sum (j*j + 1, j, 1, 10)
  - » Means sum j*j+1 for j from 1 to 10
  - » Can even call Sum recursively
    - Sum (sum (j*k+j+j, j, 1, 10), k, 2, 100)

Algol-60 Call by Name

- Here is how sum is coded
  - »
```
real procedure sum (x, i, from, to);
   integer x, i
   integer from, to;
begin
    integer s;
    s := 0;
    for i := from step 1 until to do
        s := s + x;
    sum := s;
 end
```

LISP

- LISP is quite early, developed during 60’s
- Invented
  - Functional programming
  - Use of lambda forms
  - Higher order functions
  - Lists
  - Garbage collection
- Pretty much what scheme is today

LISP grew up on the IBM 709(0)

- This machine had 36 bit words
- Divided into prefix/CDR/index/CAR
  - » CDR and CAR could hold pointers
- So why is CAR the first of the list
  - » Because in the assembly language, the CAR field was given as the first operand, and the CDR as the second
  - » CAR = **C**ontents of the **A**ddress **R**egister
  - » CDR = **C**ontents of the **D**ecrement **R**egister

LISP has dynamic scoping

- In a block structured language like Pascal, if you reference a non-local variable, you get the statically enclosing one.
- With dynamic scoping, you get the most recently declared one whether or not you are statically enclosed in its scope

Lisp, Dynamic Scoping

- Define a function f with a parameter x
- f calls separate function g (not nested)
- g has no x declared but references x
- It gets the x that is the parameter value passed to f

COBOL

- Another early language (late 50’s)
- Designed for information processing
- Important features
  - » Scaled decimal arithmetic, 18 digits
  - » Dynamic binding (at runtime) for subroutines
  - » Can add new subroutines at runtime
  - » CALL string-expression USING parameters
    - String-expression is evaluated at run time

COBOL

- Uses english language flavor
  - Idea: managers can read code (a bit bogus)
- Example

  -
    ```
    PROCESS-BALANCE.
        IF BALANCE IS NEGATIVE
            PERFORM SEND-BILL
        ELSE
            PERFORM RECORD-CREDIT
        END-IF.
    SEND-BILL.
         …
    RECORD-CREDIT.
         …
    ```

COBOL, A horrible feature

- PARA.
      GOTO .
  …
  ALTER PARA TO PROCEED TO LABEL-1
- UGH!
  - Copied from machine code for 1401
- ANSI committed tried to remove this
  - Were threatened with multiple law suits
  - So it is still there ☹

COBOL High Level Features

- Built in indexed files with multiple indexes
- Built in high level functions
  - Search and replace strings
  - Sort
  - Edit
    - Uses pictures
    - PIC ZZZ,ZZ9.99
    - If you move 1234.56 to above field you get
    - 1,234.56 (with two leading blanks)

COBOL History

- COBOL-66 First standard
- COBOL-74 upgrade, nothing too major
- COBOL-91 full structured programming
- Latest COBOL (date?)
  - » Full very elaborate object oriented stuff
- Still very widely used
  - » Particularly on mainframes
  - » Mainframes are still widely used!

PL/1

- Systematic attempt by IBM to combine the ideas in
  - Fortran
  - COBOL
  - Algol-60
  - Also added concurrency (a la IBM mainframe OS)
- Not very successful, but still used
- Widely derided as kitchen sink, committee work
- PL/1 is not as bad as its reputation
- Hurt badly by poor performance of compilers

Algol Developments

- Algol-X, Algol-Y, Algol-W
  - » Variants adding various features including notably records.
- Burroughs built Algol machines and used only Algol for all work (no assembler!)
- JOVIAL
  - » **J**ules **O**wn **V**ersion of the **I**nternational **AL**goritmic Language
  - » Widely used by DoD (still used in some projects)
- Algol-68
  - » Major update

Algol-68

- Designed by distinguished committee
- Under auspices of IFIP WG2.1
  - First comprehensive type system
  - Garbage collection required
  - Full pointer semantics
  - Includes simple tasking facilities
- Used in the UK, but not really successful
  - Lack of compilers
  - Building compilers was a really hard task

Algol-68 Modules

- An interesting addition to Algol-60
- Module facility
- First comprehensive attempt at separate compilation semantics
- Influenced later languages including Ada

Algol-68 Reactions

- Several important people thought Algol-68 had got far too complex, voted against publication, lost vote, and stormed out
  - Wirth
  - Hoare
  - Djikstra
  - Per Brinch Hansen (sp?)
  - And several others

Pascal

- Designed by Wirth as a reaction to A68
- Retained reasonably comprehensive type system
  - » Pointers, records, but only fixed length arrays
- Emphasis on simplicity
- Widely used for teaching

Pascal follow ons

- Borland picked up Pascal
- And developed it into a powerful language
- This is the language of Delphi, added:
  - Modules
  - String handling
  - Object oriented facilities
- Still in use (e.g. MTA)

Simula-67

- Another Algol development
- First object oriented language
- Objects are concurrent tasks
- So message passing involves synchronization
- Widely used in Europe in the 70’s

Another thread, BCPL

- BCPL is a low level language
- Simple recursive syntax
- But weakly typed
- Really has only bit string type
- Quite popular in the UK

B came from BCPL

- Don’t know much about this
- Seems to have disappeared into the mists of time ☺
- Important only for the next slide

C, partly inspired by B

- An attempt to create a nice simple language
- Powerful
- But easy to compile
- Formed the basis of Unix
- 32 users simultaneously using C and Unix on a PDP-11 (equivalent in power to a very early 5MHz PC, with 128K bytes memory!)

C++

- C catches the complexity bug ☺
- Adds
  - Abstraction
  - Comprehensive type system
  - Object oriented features
  - Large library, including STL

Eiffel

- Pure object oriented language
- In the Pascal tradition
- Emphasizes Programming-By-Contract ™
  - » The idea is to include assertions that illuminate the code, and form the basis of proof by correctness
  - » Code is correct if the implementation conforms to the contract

Interpreted Languages

- UCSD Pascal
  - » First language to use a byte code interpretor
  - » Widely implemented on many machines
  - » Lots of applications that could run anywhere
  - » Widely used commercially
  - » Died because of transition to PC
    - Which it did not make successfully

C#

- Designed by Microsoft
- Similar goals to Java, but
  - » Design virtual machine (.NET) first
  - » Then derive corresponding language
  - » Object oriented
  - » Cleans up C
    - Garbage collection
    - Full type safety

C# continued

- Very similar to Java
- But tries to be closer to C and C++
  - For example, overloading is retained
- Full interface to COM (what a surprise ☺)
- A nice comparison of C# and Java is at
  - http://www.csharphelp.com/archives/archive96.html

Very High Level Languages

- Another thread entirely
- Languages with high level semantics and data structures

String Processing Languages

- Languages where strings are first class citizens (intended for language processing etc)
  - COMMIT (language translation project at University of Chicago)
  - SNOBOL and SNOBOL3 (Bell Labs)
  - SNOBOL4
    - Comprehensive pattern matching
  - ICON
    - Particularly develops notion of back tracking

SETL

- A language designed in the 70’s
- Based on ZF Set theory
  - » Here is printing of primes up to 100
  - » Print ({x in 2 .. 100 |

```
                notexists d in 2 .. X-1 |
                  x mod d = 0} )
```

  - » Notexists here should use nice math symbol!
  - » General mappings and sets

Functional Languages

- MIRANDA
  - » Introduced notion of lazy evaluation
  - » Don’t evaluate something till needed
  - » Provides pure referential transparency
  - » Suppose we have
    - define f (x, y) = y
    - Can replace f (expr1, expr2) by expr2
    - Even if computing expr1 would cause infinite loop with strict semantics (strict as opposted to lazy)

HASKEL

- Comprehensive attempt at defining modern usable functional language
- Uses more familiar syntax (not so reliant on parens ☺)
- Has lazy evaluation
- And large library of stuff