Data Abstraction and Object Orientation

Why abstractions?
• Reduce conceptual load
  – Hide implementation details
• Independence among program components
  – Replacement of pieces without rewriting others
  – Organizational compartmentalization
• Fault containment
  – Enforce division of labor
  – Prevent access to things you shouldn’t see

Object-Oriented Programming
• Fundamental concepts
  – Encapsulation
  – Inheritance
  – Dynamic method binding
• Class – module as the abstract type
  including data and method definition
• Object – instance of a class

Encapsulation
• Allows reasoning at the level of the interface
• Namespace (C++) or packages (Java)
  – Modules that span multiple files
  – Collection of objects (subroutines, types, variables) visible to each other but visible to the outside only if explicitly exported
  – Class definitions visible only within module

Initialization and Finalization
• Constructors and destructors
  – Storage allocation and deallocation
  – Call base constructor before derived class constructor

Data Abstractions
• Scopes and lifetime
  – Global variables (introduced by Basic)
  – Lifetime and scope spans program execution
• Local variables (introduced by Fortran)
• Lifetime and scope limited to execution of subroutine
• Nested scopes (Algol 60)
  – Allows subroutines or blocks to themselves be local
• Static variables (Fortran)
• Lifetime spans execution, names visible in a single scope
• Modules (Modula-2)
  – Allow a collection of subroutines to share a set of static variables
• Module types (Eurid)
  – Allow instantiation of multiple instances of a given abstraction
• Classes (Smalltalk, C++, Java)
  – Allow definition of families of related abstractions
Visibility Rules

- Parts of an object declaration/definition (e.g., C++)
  - Public
    - Accessible to users of the class
  - Private
    - Accessible to members of this class
  - Protected
    - Accessible to members of this or derived classes
- Derived classes can restrict visibility of members of a base class in C++ (but not in Java)

Implementation of Classes

- Dynamic method binding
  - Virtual methods to dispatch appropriate implementation at run time (dynamic)
  - Abstract classes – contain virtual methods with no body
  - Virtual method table (vtable)
    - Array whose ith entry indicates the address of the code for the object’s ith virtual method
    - FST field of record of each object contains address of vtable, shared by all objects of a given class
- Static method binding – version called based on type of the variable or reference being used rather than class to which object referred to belongs
- Reflection – mechanism by which type information can be obtained at run time

Virtual Method Table (vtable)

Dynamic Method Binding

Generics

- Dynamic method binding introduces polymorphism
- Base class methods return references of base class type
- Type-specific operations?Use generics

```
template<class V>
class list_node {
  list_node* prev;
  V val;
  ...
  typedef list_node<int> int_list_node;
  ...
  int_list_node* first_int;
```

Inheritance

- Multiple inheritance - Inherit from more than one base class
  - E.g.,
    - class student: public person, public gp_list_node { ...
  - Replicated vs. shared inheritance when deriving from the same base
- Mix-in inheritance
  - Java – base class composed entirely of abstract methods (an interface)
  - Inherit from one real base class and an arbitrary number of interfaces
  - Facilitates code reuse through polymorphism
Multiple Inheritance

Replicated Multiple Inheritance

Shared Multiple Inheritance