V22.0490.001
Special Topics: Programming Languages

B. Mishra
New York University.

Lecture # 20

—Slide 1—

Generic

A generic abstract data type definition

allows certain attribute of the type to be specified separately so that one base type definition may be given, with the attributes as parameters, and then several specialized types derived from the same base type may be created.

- Like type definitions with parameters
- Parameters may be type names as well as values
- Parameters may affect the operations in the abstract data type definition

Generic Facility in Ada

- Provides means of parameterizing subprograms and packages
- Generic packages provide families of abstract data types
- Generic subprograms provide families of abstract operations

—Slide 2—

Use of `generic` *in Ada*

```
package QUEUE_PACKAGE is
  procedure APPEND(C: in Character);

  procedure REMOVE(C: out Character);

  function IS_EMPTY return Boolean;

  function IS_FULL return Boolean;

  procedure INIT_QUEUE;

  procedure DESTROY_QUEUE;
end;
```

—Slide 3—

Generic QUEUE_PACKAGE

```
generic
  type Elem is private;
  MAXELTS: in Natural;
package QUEUE_PACKAGE is
  type Queue is limited private;

  procedure APPEND(Q: in out Queue; C: in Elem);
  procedure REMOVE(Q: in out Queue; C: out Elem);
  function IS_EMPTY(Q: in Queue) return Boolean;
  function IS_FULL(Q: in Queue) return Boolean;
  procedure INIT_QUEUE(Q: in out Queue);
  procedure DESTROY_QUEUE(Q: in out Queue);

  FULL_QUEUE, EMPTY_QUEUE: exceptions;
private
  type Queue is
    record
      FIRSTELT, LASTELT: Integer range 0..MAXELTS - 1 := 0;
      CURSIZE : Integer range 0..MAXELTS := 0;
      ELEMENTS: array (0..MAXELTS - 1) of Elem;
    end record;
end QUEUE_PACKAGE;
```

—Slide 4—

Generic Formal Parameters

Can be: *Object parameters*, *Type Parameters*, *Subprogram parameters*, or *Package Parameters*

- **Object Parameters**
  – Similar to parameters of subprograms
  – Modes: `in` & `out`

```
generic
  MAX_LEN : in Natural; --Local Constants
```

- **Type Parameters**
  – Necessary to specify some of the properties of the actual type parameters to allow syntactic and semantic checks to be performed on the body of the generic subprogram independently of the generic instantiation.

```
generic
  type Item is private;
  type Sort_Array is array (positive range <>) of Item;
   with function "<"(u,v: Item) return Boolean is <>;
function HEAP_SORT(A: Sort_Array) return Sort_Array is
   ...
end HEAP_SORT;
```

—Slide 5—

Generic Subprogram Parameters

Three Forms:

- `with function SUM(X, Y: Item) return Item;`
  – An actual function of the appropriate function type

  $$\texttt{Item} \times \texttt{Item} \rightarrow \texttt{Item}$$

  must be provided.

- `with function "*"(X, Y: Item) return Item is <>;`
  – An actual function may be omitted
  – The default is obtained from the instantiation.

- `type Enum is (<>);`

  `with function NEXT(A: Enum) return Enum is Enum'Succ;`
  – An actual function may be omitted
  – The default is then the one specified in the declaration: (e.g., `Enum'Succ`)

—Slide 6—

Generic Package Parameters

Two Forms:

- `with package P is new Q(<>);`
  – Actual parameters corresponding to `P` must be a package—obtained by instantiating a generic package `Q`.

- `with package R is new Q(P1, P2, ...);`
  – Actual parameters corresponding to `R` must have been instantiated with the given parameters `P1`, `P2`, etc.

—Slide 7—

Generic Package Parameters (Example)

```
generic
  type Floating is digits <>;
package Generic_Complex_Numbers is
  type Complex is private;
  ...
end;

generic
  type Index is (<>);
  with package Complex_Number is new Genric_Complex_Numbers(<>);
package Generic_Complex_Vectors is
  use Complex_Numbers;
  type Vector is array (Index range <>) of Complex;
  ...
end;

package Long_Complex is new Generic_Complex_Numbers(Long_Float);
use Long_Complex;
package Long_Complex_Vectors is
  new Generic_Complex_Vectors(Integer, Long_Complex);
```

—Slide 8—

Polymorphism

Polymorphism means the ability of a single operator or a subprogram name to refer to any number of functions, depending on the data types of the arguments and results.

- Usually, parameters are assumed to have L-values. Polymorphism allows one to go beyond simply those objects with L-values.
- Note `+` in `1 + 2` and `1.1 + 2.1` is *overloaded*, signifying polymorphism.
- Examples: `generic` in Ada and `template` in C++ allow limited forms of polymorphism

—Slide 9—

Inheritance

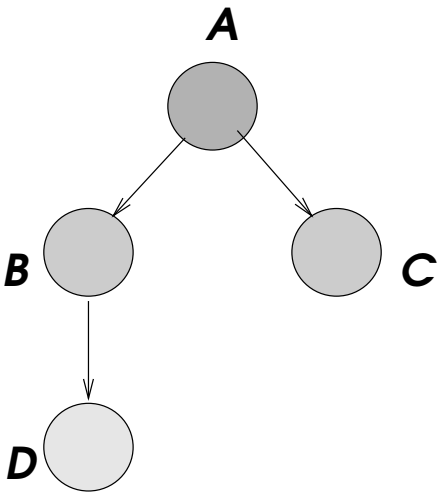
Information passed implicitly among program components

- **Inheritance is**
  the receiving in one program component of properties or characteristics of another component *according to the special relationship* that exists between the two components.
  - Passing of data and functions between independent modules.
  - $A \Rightarrow B$ — inheritance relationship between class A and class B is established.
  - If a certain object X is declared within class A and is not redefined in class B, then a reference to X in B actually refers to object X of class A via *inheritance*.

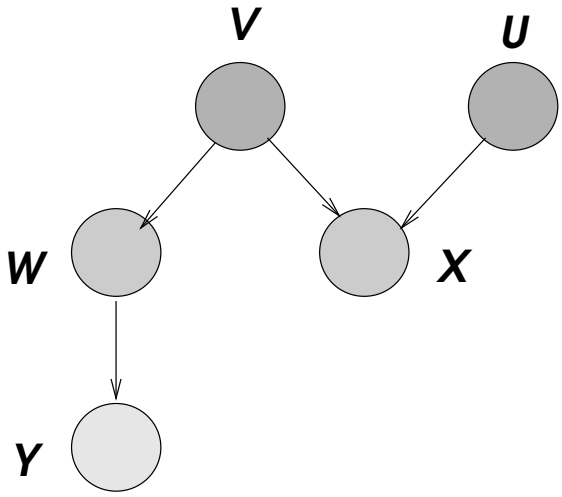
—Slide 10—

Single & Multiple Inheritance

- $A \Rightarrow B$
  - A is the *parent* class, *base* class or *superclass*
  - B is the *child* class, *dependent* class or *subclass*



Single Inheritance



Multiple Inheritance

- *Single Inheritance*: Class can only have a single parent.
- *Multiple Inheritance*: Class can have multiple parents.

—Slide 11—

Example: C++

```
class elem {
public:
  elem(){v = 0}
  void ToElem(int b){v = b;}
  int FromElem(){ return v;}
private:
  int v;}

class ElemStack: elem {
public:
  ElemStack() { size=0;}
  void push(elem i)
    {storage[++size] = i;}
  void pop()
    {return storage[size--];}
  void MyType() {printf("I am type ElemStack\n")}
protected:
  int size;
  elem storage[100];}
```

—Slide 12—

Derived Class and Method Inheritance

```
class NewStack: ElemStack {
public:
  int peek(){ return storage[size].FromElem()}
}

. . .

{ elem x;
  ElemStack y;
  int i;
  read(i);
  x.ToElem(i);   // Coercion to elem type
  y.push(x);
  ...
}
```

—Slide 13—

Mixin Inheritance

- $A \Rightarrow B$ — Class B is derived from and is a modification of class A.
- *Mixin Inheritance*: Define only the difference between the base class and the new derived class.

```
deltaclass StackMod{
  int peek(){return storage[size].FromElem();}
}// Does not exist in C++

class NewStack
  = class ElemStack + deltaclass StackMod;
```

—Slide 14—

History of Inheritance

- Emerged from Artificial Intelligence research in the 1970's on Knowledge representation.
  Scott Fahlman's `NETL`.
- Developed as a way to organize knowledge efficiently.
- Adopted by object-oriented programming as a way to organize programs and facilitate reuse.
- ```
  Elelphant extends Mammal, PhysicalObject;
        //Elephant is a (kind of) mammal
        //Elephant is a (kind of) physical object
  Clyde extends Elephant;
        //Clyde is an (instance of) elephant
  ```
  - *Does Clyde have big ears?*

  —Yes, because it's an elephant.
  - *Can Clyde lay eggs?*

  —No, elephant is a mammal.
  - *What will happen to Clyde, if you shoot it while it's flying?*

  —It'll fall, since it's a physical object.
  ```

—Slide 15—

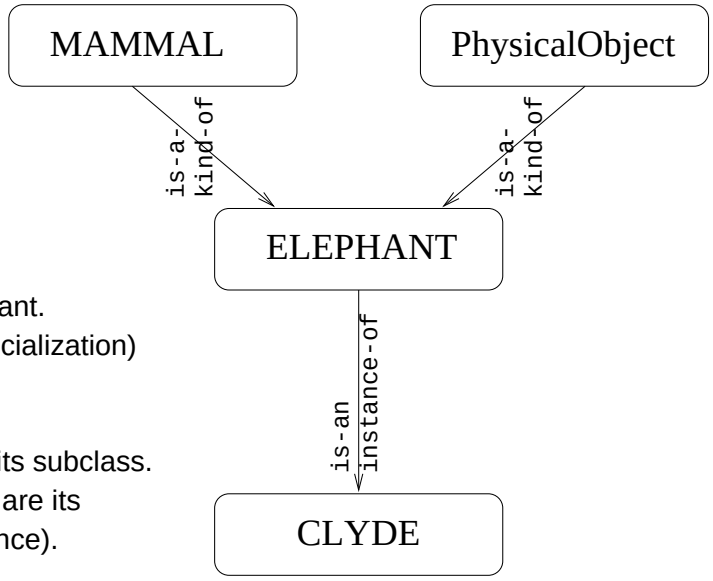
Clyde's Story

1. Clyde inherits all the properties of elephants.

2. Elephants inherit all the properties of mammals.
   Elephants inherit all the properties of physical objects.

3. Clyde is an instance of elephant.
   But elephant is a kind of (specialization) mammal and physical object

4. Elephant is a class. Clyde is its subclass.
   Mammal and PhysicalObject are its superclass. (Multiple inheritance).



—Slide 16—

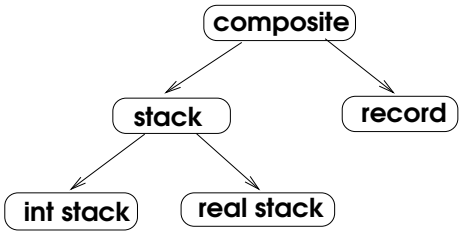
Inheritance Fosters Reuse

- Build a Taxi by starting with an existing car and modifying it.
- A new legal contract can often be drafted by starting with existing boilerplate for that type of agreement.
- A new graphical user interface (GUI) control can often be built by basing it on an existing control.

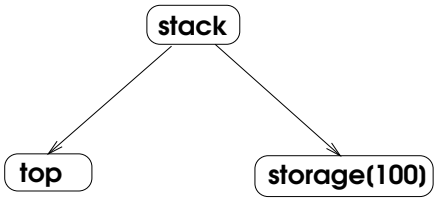
—Slide 17—

Abstraction Concepts
Specialization & Decomposition

If $A \Rightarrow B$ means B is a related class to A
what is the relationship between objects of A and B?



Specialization

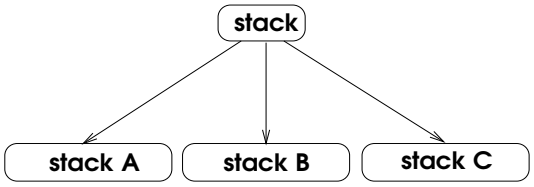


Decomposition

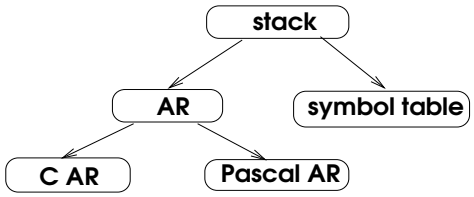
- **Specialization**
  - Allows the derived object B to obtain more precise properties than A.
  - Class `NewStack` is a specialization of `ElemStack`
  - *Converse process,* **Generalization**.
- **Decomposition**
  - Separates an abstraction into its components
  - *Converse process,* **Aggregation**.

—Slide 18—

Abstraction Concepts
Instantiation & Individualization



Instantiation



Individualization

- **Instantiation**
  - Creates instances of a class (copy operation)
  - *Converse process,* **Classification**.
- **Individualization**
  - Groups similar objects with common purposes
  - *Converse process,* **Grouping**.

—Last Slide—

Inheritance in Ada 95

- **Single Inheritance**

  Note: `Adventure` has all the "methods" of `Tried_And_Trusted`, but none of `Wizard_Stuff`

- **Mixin Inheritance**

  – Combines *inheritance* with *static polymorphism*

- **Containers**

- **Iterators**

- **Multiple Views**

[End of Lecture #20]