V22.0490.001
Special Topics: Programming Languages

B. Mishra
New York University.

Lecture # 19

—Slide 1—

Data Abstraction Encapsulation and Inheritance

- Specification of the *Abstract Data Types* (Design of attributes and operation)
- Implementation of *Concrete Data Types*
- Encapsulation — Usage does not depend on implementation details.
- Mechanisms
  - *Subprograms*
  - *Type Definitions*
  - *Inheritance*
- **History**
  - (Pascal 1970) Type Definition: defines structure of a data object with its possible value binding.
  - Type can be then bound (as an attribute) to a variable name
  - (Ada, Modula, etc.) Further extended to *set of data objects + set of operations*

—Slide 2—

Data Abstraction and Encapsulation of Programmer Defined Data Types

Abstract Data Types

- *Set of Data Objects* (uses other data type definitions)
- *Set of Abstract Operations* on those data objects
- *Encapsulation of the Whole* in such a way that the user of the new type cannot manipulate data objects except by the user defined operations.

1) **Package** 2) **Module** 3) **Class**
4) **Flavor** 5) **Form** 6) **Structure**
7) **Cluster**

- Distinction between

  Ada Packages — Partition the static program text

  C++ Class — Also describes dynamic objects (at compile time)

—Slide 3—

Stack Example in Ada 95

```
package STACK_PACKAGE is                          --specification
  procedure PUSH(C: in Character);

  function POP return Character;

  function EMPTY return Boolean;

  function FULL return Boolean;

  procedure INIT;
end STACK_PACKAGE;

package body STACK_PACKAGE is                      --body
  MAX_LEN: constant Integer := 1000;

  TOP: Integer := -1
  ELEMENTS: array (0..MAX_LEN -1) of Character;
    ...
```

—Slide 4—

STACK_PACKAGE *Continued*

```
  procedure PUSH(C: in Character) is
    begin if TOP < MAX_LEN then
          TOP := TOP + 1; ELEMENTS(TOP) := C;
    end if; end PUSH;

  procedure POP return Character is
    begin if TOP > -1 then
          TOP := TOP - 1; return ELEMENTS(TOP+1);
    end if; end POP;

  function EMPTY return Boolean is
    begin return TOP = -1; end;

  function FULL return Boolean is
    begin return TOP = MAX_LEN - 1; end;

  procedure INIT is begin null; end;

begin
  Top := -1;                                    --initialization
end STACK_PACKAGE;
```

—Slide 5—

STACK *in C++*

```
#define MAX_LEN 1000

struct STACK {
  char ELEMENTS[MAX_LEN];

  int TOP = -1;
  enum {EMPTY = -1; FULL = MAX_LEN -1};

  void PUSH(char C){TOP++; ELEMENTS[TOP] = C;}

  char POP(){TOP--; return (ELEMENT[TOP+1]);}

  boolean EMPTY(){return (boolean)(TOP == EMPTY);}

  boolean FULL(){return (boolean)(TOP == FULL);}

  void INIT(){}
};
```

—Slide 6—

Abstraction Facilities

Set of Modules:

Each module performs a limited set of operations on a limited amount of data.

- Information is encapsulated. Implementation hiding. Representation independence.
  1. User *does not need to know* the hidden information to use the abstraction.
  2. User *is not permitted* to directly use or manipulate hidden information.
- Ada, C++, Smalltalk provide language support for data encapsulation.
- In Ada, the declaration "`is private`" for type makes the internal structure inaccessible. Only subprograms within the package have access to "`private`" data.

—Slide 7—

Modules & Classes

- **Modules**:
  - *Static*
    – cannot create new modules or copies of existing modules at run-time
  - Interface (*Public View*)
    – Collection of declarations: Types, Variables, Procedures, etc.
  - Implementation (*Private View*)
    – Code for the procedures ... may be changed without affecting the interface.
- **Classes**:
  - *Dynamic*
    – Class of objects may be created and deleted at run time
  - *Constructor & Destructor*
    – Constructor contains initialization process
    – Destructor contains clean-up and "last-rites" (will-and-testament) process

—Slide 8—

Example in C++

```
const int max_len = 255;

class string {
  char *s;
  int len;                              \\PRIVATE

public:                                 \\PUBLIC form here
  string(){s = new char[max_len]; len = max_len - 1;}
  string(char *p){len = strlen(p); s = new char[len+1];
                  strcpy(s,p);}         \\CONSTRUCTORS

  ~string() {delete s;}                 \\DESTRUCTOR

  void operator = (string&);
  void operator += (string&);
  string operator + (string&);

  void assign(char* st){strcpy(s,st); len=strlen(st);}
  void print(){cout << s << "\nLength" << len << "\n";}
};
```

—Slide 9—

`string` *continued*

```
void string::operator = (string& a)
{strcpy(s,a.s); len = a.len;}

void string::operator += (string& a)
{strcpy(s+len,a.s); len += a.len;}

string string::operator + (string& a)
{
  string temp(len+a.len);
  strcpy(temp.s,s);
  strcpy(temp.s+len, a.s);

  return(temp);
}
```

—Slide 10—

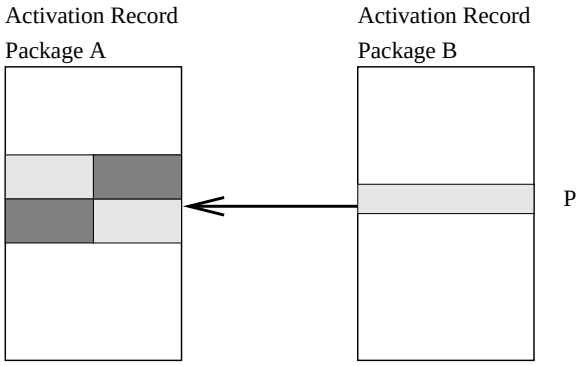
`string` *Usage*

```
main()
{
  string one("Bud_Why_Err"),
         two("Why ask Why"),
         both, four;

  both = one;
  both.print();
  both += two;
  both.print();
  both = both + both;
  both.print();
  four.assign("This Bud's for you");
  both = four + two;
  both.print();
}
```

—Slide 11—

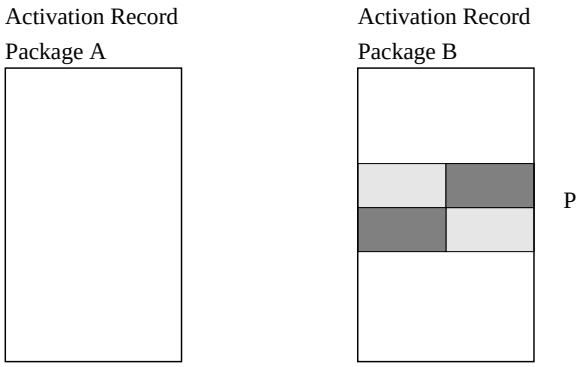
Implementation Issues
Indirect Encapsulation



- The structure of the abstract data type is defined by the package specification A
- Actual storage is maintained in an activation record for package A
- In package B, which declares and uses the objet P, the run-time activation record contains a pointer to the actual data storage.
- Huge run-time cost — Low recompilation cost

—Slide 12—

Implementation Issues
Direct Encapsulation



- The structure of the abstract data object is defined by the specification for package A
- The actual storage for object P is maintained within the activation record for package B
- High run-time execution efficiency — High compile-time cost. Use of an abstract data type requires the compiler to know the details of representation (including the private components)
- Ada uses *direct encapsulation*

—Slide 13—

Key Concepts of OOPS

- **Abstraction:** Interfaces abstract away from implementation details.
- **Encapsulation:** Ccontrolled access to the internal states. Consistent states and consistent state transition.
- **Modularity:** Each object has minimal dependence on other objects. (Inheritance, Friends, Public Access, Import/Export)
- **Reuse:** Code reuse through inheritance and polymorphism. Minimal Code modification.

—Last Slide—

Key Concepts of OOPS, Contd.

- **Invariants:** Universal properties that hold of an instance at all the time (except inside the class's method).
- **Mutability:** Allows one to design immutable objects (Java `final`, e.g., `Strings`) that cannot be changed once it is created.

[End of Lecture #19]