Multidimensional Arrays as Parameters: **Ada**

- **Ada** – not a problem
  - Constrained arrays – size is part of the array’s type
  - Unconstrained arrays - declared size is part of the object declaration

**Example:**

```ada
Function Sumer(Mat: in Mat_type) return Float is
  Sum: Float := 0.0;
Begin
  ...
End Sumer;
```

Multidimensional Arrays as Parameters: **Fortran**

- Formal parameter that are arrays have a declaration after the header
  - For single-dimension arrays, the subscript is irrelevant
  - For multidimensional arrays, the sizes are sent as parameters and used in the declaration of the formal parameter, so those variables are used in the storage mapping function

Multidimensional Arrays as Parameters: **Java & C#**

- Similar to **Ada**
- Arrays are objects; they are all single-dimensioned, but the elements can be arrays
- Each array inherits a named constant (`length` in Java, `Length` in C#) that is set to the length of the array when the array object is created
Design Considerations for Parameter Passing

- Two important considerations
  - Efficiency
  - One-way or two-way data transfer
- But the above considerations are in conflict
  - Good programming suggests limited access to variables, which means one-way whenever possible
  - But pass-by-reference is more efficient to pass structures of significant size

Parameters that are Subprogram Names

- It is sometimes convenient to pass subprogram names as parameters
  - **Issues:**
    1. Are parameter types checked?
    2. What is the correct referencing environment for a subprogram that was sent as a parameter?

Parameters that are Subprogram Names: Parameter Type Checking

- **C** and **C++**: functions cannot be passed as parameters but pointers to functions can be passed and their types include the types of the parameters, so parameters can be type checked
- **FORTRAN 95** type checks
- **Ada** does not allow subprogram parameters; an alternative is provided via Ada's generic facility
- **Java** does not allow method names to be passed as parameters

Parameters that are Subprogram Names: Referencing Environment

- **Shallow binding**: The environment of the call statement that enacts the passed subprogram
  - Most natural for dynamic-scoped languages
- **Deep binding**: The environment of the definition of the passed subprogram
  - Most natural for static-scoped languages
- **Ad hoc binding**: The environment of the call statement that passed the subprogram
Overloaded Subprograms

• An overloaded subprogram is one that has the same name as another subprogram in the same referencing environment.
  – Every version of an overloaded subprogram has a unique protocol.
• C++, Java, C#, and Ada include predefined overloaded subprograms.
• C++, Java, C#, and Ada allow users to write multiple versions of subprograms with the same name.
• In Ada, the return type of an overloaded function can be used to disambiguate calls (thus two overloaded functions can have the same parameters).

Generic Subprograms

• A generic or polymorphic subprogram takes parameters of different types on different activations.
• Overloaded subprograms provide ad hoc polymorphism.
• A subprogram that takes a generic parameter that is used in a type expression that describes the type of the parameters of the subprogram provides parametric polymorphism.
  – A cheap compile-time substitute for dynamic binding.

Generic Subprograms (cont’d)

• Ada
  – Versions of a generic subprogram are created by the compiler when explicitly instantiated by a declaration statement.
  – Generic subprograms are preceded by a generic clause that lists the generic variables, which can be types or other subprograms.

Generic Subprograms (cont’d)

• C++
  – Versions of a generic subprogram are created implicitly when the subprogram is named in a call or when its address is taken with the & operator.
  – Generic subprograms are preceded by a template clause that lists the generic variables, which can be type names or class names.
Generic Subprograms (cont’d)

• Java 5.0
  – Differences between generics in Java 5.0 and those of C++ and Ada:
    1. Generic parameters in Java 5.0 must be classes
    2. Java 5.0 generic methods are instantiated just once as truly generic methods
    3. Restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters
    4. Wildcard types of generic parameters

Examples of parametric polymorphism: C++

```cpp
template <class Type>
Type max(Type first, Type second) {
    return first > second ? first : second;
}

int max (int first, int second) {
    return first > second ? first : second;
}
```

Design Issues for Functions

• Are side effects allowed?
  – Parameters should always be in-mode to reduce side effect (like Ada)
• What types of return values are allowed?
  – Most imperative languages restrict the return types
  – C allows any type except arrays and functions
  – C++ is like C and also allows user-defined types
  – Ada subprograms can return any type (but Ada subprograms are not types, so they cannot be returned)
  – Java and C# methods can return any type (but because methods are not types, they cannot be returned)
  – Python and Ruby treat methods as first-class objects, so they can be returned, as well as any other class
User-Defined Overloaded Operators

- Operators can be overloaded in Ada, C++, Python, and Ruby
- An Ada example

```ada
function "*" (A,B: in Vec_Type): return Integer is
  Sum: Integer := 0;
  begin
  for Index in A'range loop
    Sum := Sum + A(Index) * B(Index)
  end loop
  return sum;
end "*";
```

...c = a * b;

Coroutines

- A coroutine is a subprogram that has multiple entries and controls them itself – supported directly in Lua
- Also called symmetric control: caller and called coroutines are on a more equal basis
- A coroutine call is named a resume
- The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine
- Coroutines repeatedly resume each other, possibly forever
- Coroutines provide quasi-concurrent execution of program units (the coroutines); their execution is interleaved, but not overlapped
Coroutines Illustrated: Possible Execution Controls with Loops

Summary

- A subprogram definition describes the actions represented by the subprogram
- Subprograms can be either functions or procedures
- Local variables in subprograms can be stack-dynamic or static
- Three models of parameter passing: in mode, out mode, and inout mode
- Some languages allow operator overloading
- Subprograms can be generic
- A coroutine is a special subprogram with multiple entries