

Organization of Programming Languages

CS320/520N

Lecture 06

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

Data Types

- A **data type** defines a collection of data objects and a set of predefined operations on those objects.
- **Primitive data types** are those not defined in terms of other data types:
 - Some primitive data types are merely reflections of the hardware.
 - Others require only a little non-hardware support for their implementation.
- **User defined types** are created with flexible *structure* defining operators (ALGOL 68).
- **Abstract data types** separate the interface of a type (visible) from the representation of that type (hidden).

Primitive Data Types

- **Integers** – almost always an exact reflection of the hardware.
 - Java's signed integers: `byte`, `short`, `int`, `long`.
- **Floating Point** – model real numbers, but only as approximations.
 - Support for two types: `float` and `double`.
- **Complex** – two floats, the real and the imaginary.
 - Supported in Fortran and Python.
- **Boolean** – two elements, `true` and `false`.
 - Implemented as bits or bytes.
- **Character** – stored as numeric codings.
 - ASCII 8-bit encoding, UNICODE 16-bit encoding.

Character String Types

- **Character Strings** – values are sequences of characters.
- Typical operations:
 - Assignment.
 - Comparison.
 - Concatenation.
 - Substring reference.
 - Pattern matching.
- Design issues:
 - Is it a primitive type or just a special kind of array?
 - Should the length of strings be static or dynamic?

Strings in Programming Languages

- C and C++:
 - Implemented as null terminated char arrays.
 - A library of functions in `string.h` that provide string operations.
 - Many operations are inherently unsafe (ex: `strcpy`).
 - C++ `string` class from the standard library is safer.
- Java (C# and Ruby):
 - Primitive via the `String` class (immutable).
 - Arrays via the `StringBuffer` class (mutable, w/ subscripting).
- Fortran:
 - Primitive type.

Strings in Programming Languages

- Python:
 - Primitive type that behaves like an array of characters:
 - indexing, searching, replacement, character membership.
 - Immutable.
- Pattern Matching:
 - built-in for Perl, JavaScript, Ruby, and PHP, using regular expressions.
 - class libraries for C++. Java, Python, C#.

String Length

- **Static Length** – set when the string is created:
 - Java String, C++ STL, Ruby String, C# .NET.
- **Limited Dynamic Length** – length can vary between 0 and a maximum set when the string is defined:
 - C/C++ null terminated strings.
- **Dynamic Length** – varying length with no maximum:
 - JavaScript and Perl (overhead of dynamic allocation/deallocation).
- Ada supports all three types:
 - String, Bounded_String, Unbounded_String.

Character String Implementation

- Static length:
 - compile-time descriptor storing the length and the address.
- Limited dynamic length:
 - may need a run-time descriptor for length (but not in C and C++).
- Dynamic length:
 - need run-time descriptor;
 - allocation/de-allocation is the biggest implementation problem.

Character String Implementation

Static string
Length
Address

Compile-time
descriptor for
static strings

Limited dynamic string
Maximum length
Current length
Address

Run-time descriptor
for limited dynamic
strings

Array Types

- An **array** is an aggregate of homogeneous data elements in which an individual element is identified by its position in the aggregate, relative to the first element.
- Indexing is a mapping from indices to elements:
`array_name[index_value_list] → an element`
- Index range checking:
 - C, C++, Perl, and Fortran do not specify range checking.
 - Java, ML, C# specify range checking.
 - In Ada, the default is to require range checking, but it can be turned off.

Array Categories

- **Static:** subscript ranges are statically bound and storage allocation is static (before run-time)
 - Advantage: efficiency – no dynamic allocation/deallocation.
 - Example: arrays declared as `static` in C/C++ functions.
- **Fixed Stack-Dynamic:** subscript ranges are statically bound, but the allocation is done at declaration time (at run-time)
 - Advantage: space efficiency – stack space is reused.
 - Example: arrays declared in C/C++ functions without the `static` modifier.

Array Categories

- **Stack-Dynamic:** subscript ranges are dynamically bound and the storage allocation is dynamic (at run-time):
 - Advantage: flexibility – the size of an array need not be known until the array is to be used.
 - Example: Ada arrays.

```
Get(List_Len);  
declare  
  List: array(1. . List_len) of Integer;  
begin  
  ...  
end;
```

Array Categories

- **Fixed Heap-Dynamic:** similar to fixed stack-dynamic i.e. subscript range and storage binding are fixed after allocation:
 - Binding is done when requested by the program.
 - Storage is allocated from the heap.
 - Examples:
 - C/C++ using malloc/free or new/delete.
 - Fortran 95.
 - In Java all arrays are fixed heap-dynamic.
 - C#.

Array Categories

- **Heap-dynamic:** binding of subscript ranges and storage allocation is dynamic and can change any number of times:
 - Advantage: flexibility, as arrays can grow or shrink during program execution.
 - Examples:
 - C#:

```
ArrayList intList = new ArrayList();  
intList.add(nextOne);
```
 - Java has a similar class, but no subscripting (use methods `get()/set()` instead).
 - Perl, JavaScript, Python, Ruby

Array Initialization

- Some languages allow initialization at the time of storage allocation:

- C, C++, Java, C# example:

```
int list [] = {4, 5, 7, 83}
```

- Arrays of strings in C and C++

```
char *names [] = {"Bob", "Jake", "Joe"};
```

- Java initialization of String objects:

```
String[] names = {"Bob", "Jake", "Joe"};
```

- Ada initialization using *arrow* operator:

```
Bunch : array (1..5) of Integer := (1 => 17,  
3 => 34, others => 0)
```

Heterogeneous Arrays

- A **heterogeneous array** is one in which the elements need not be of the same type.
- Supported by:
 - Perl: any mixture of scalar types (numbers, strings, and references).
 - JavaScript: dynamically typed language \Rightarrow any type.
 - Python and Ruby: references to objects of any type

Rectangular and Jagged Arrays

- A rectangular array is a multi-dimensional array in which all of the rows have the same number of elements and all columns have the same number of elements:

- Fortran, Ada, and C# support rectangular arrays.

```
myArray[ 3 , 7 ]
```

- A jagged matrix has rows with varying number of elements:

- Possible when multi-dimensional arrays actually appear as arrays of arrays
- C, C++, C# and Java support jagged arrays.

```
myArray[ 3 ][ 7 ]
```

Slices

- A **slice** is some substructure of an array:
 - nothing more than a referencing mechanism.
 - only useful in languages that have array operations.
- Fortran 95 (also Perl, Python, Ruby, restricted in Ada):

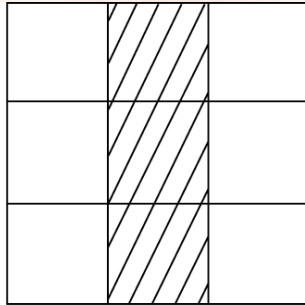
```
Integer, Dimension (10) :: Vector
```

```
Integer, Dimension (3, 3) :: Mat
```

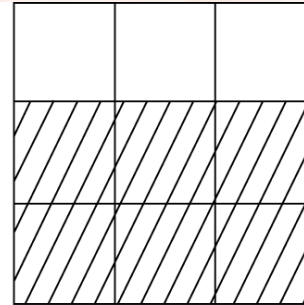
```
Integer, Dimension (3, 3) :: Cube
```

`Vector (3:6)` is a four element array

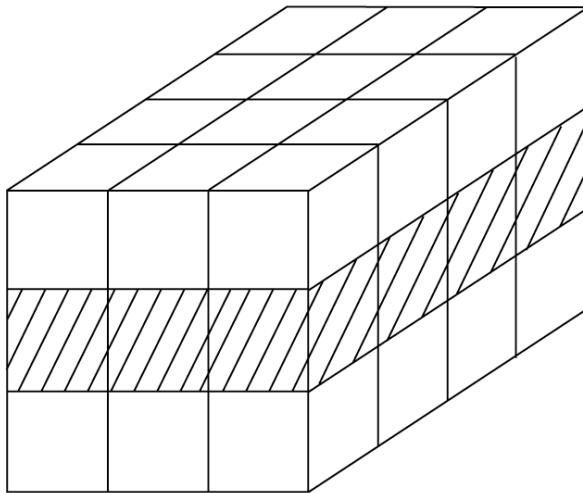
Slices Examples in Fortran 95



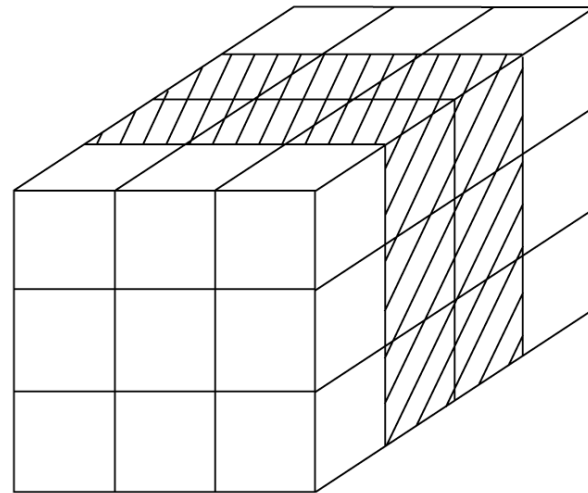
MAT (1:3, 2)



MAT (2:3, 1:3)



CUBE (2, 1:3, 1:4)



CUBE (1:3, 1:3, 2:3)

Implementation of Arrays

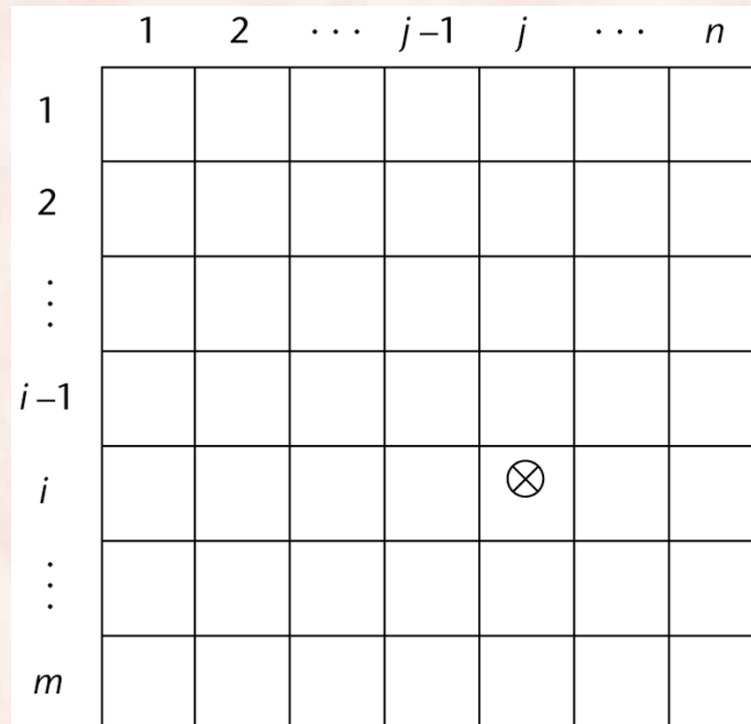
- **Access function** maps subscript expressions to the address of an element in the array.
- Single-Dimensional Arrays:
 - implemented as a block of adjacent memory cells.
 - access function for single-dimensioned arrays:
$$\text{address}(\text{list}[k]) = \text{address}(\text{list}[\text{lower_bound}]) + ((k - \text{lower_bound}) * \text{element_size})$$

Implementation of Arrays

- Multi-dimensional arrays can be accessed in two common ways:
 - Row major order (by rows) – used in most languages.
 - column major order (by columns) – used in Fortran.
- C based languages do not support true multi-dimensional arrays:
 - represented as arrays of arrays.

Access Function for a Multi-Dimensioned Array

$$\text{address}(a[i,j]) = \text{address}(a[\text{row_lb}, \text{col_lb}]) + (((i - \text{row_lb}) * n) + (j - \text{col_lb})) * \text{elem_size}$$



Compile-Time Descriptors

Array
Element type
Index type
Index lower bound
Index upper bound
Address

Single-dimensional array

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 1
⋮
Index range n
Address

Multi-dimensional array

Record Types

- A **record** is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names.
- A record type in Ada:

```
type Emp_Rec_Type is record
    First: String (1..20);
    Mid: String (1..10);
    Last: String (1..20);
    Hourly_Rate: Float;
end record;

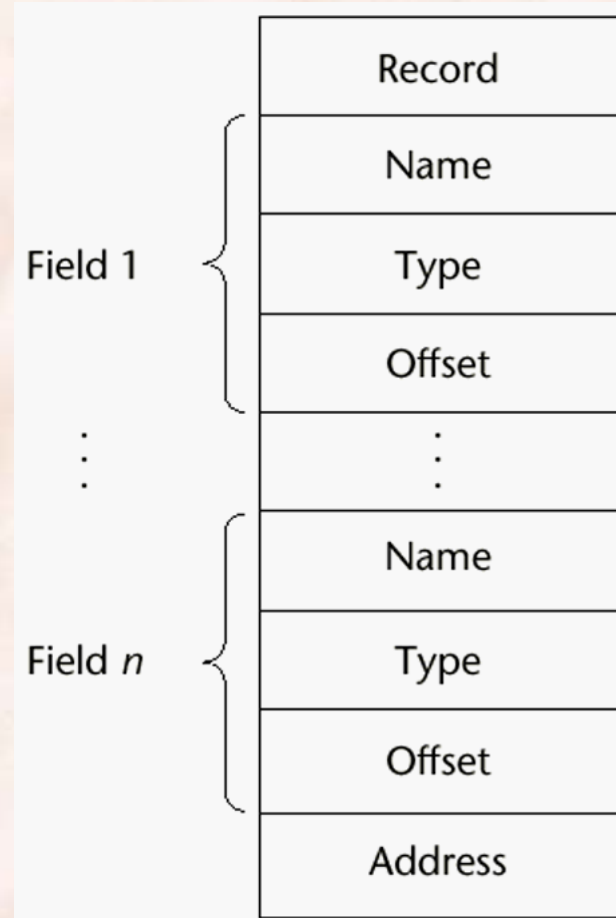
Emp_Rec: Emp_Rec_Type;
```

Record Types

- C, C++, C#: supported with the `struct` data type.
 - In C++ structures are minor variations on classes.
 - In C# structures are related to classes, but also quite different.
 - In C++ and C# structures are also used for *encapsulation*.
- Python, Ruby: implemented as hashes.

Implementation of Record Types

Offset address relative to the beginning of the records is associated with each field



Records vs. Arrays

- Arrays mostly used when:
 - collection of data values is homogenous.
 - values are process in the same way.
 - order is important.
- Records are used when:
 - collection of data values is heterogeneous.
 - values are not precessed in the same way.
 - unordered.
- Access to array elements is much slower than access to record fields:
 - array subscripts are dynamic.
 - record field names are static.

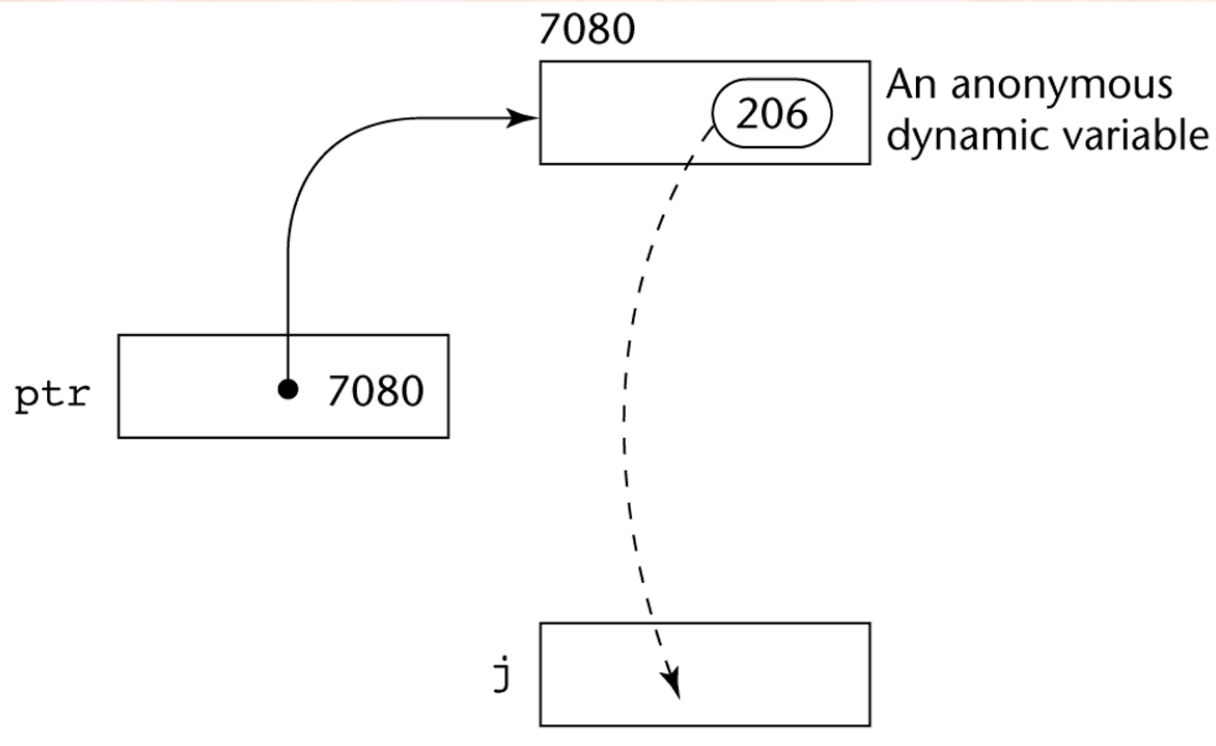
Pointer Types

- A **pointer** type variable has a range of values that consists of memory addresses and a special value **nil**.
 - Provide the power of indirect addressing.
 - Provide a way to manage dynamic memory
 - a pointer can be used to access a location in the area where storage is dynamically created i.e. the heap.
 - variables that are dynamically allocated on the heap are **heap-dynamic variables**.
- Pointer types are defined using a type operator:
 - C/C++: `int *ptr = new int;`

Pointer Operations

- Two fundamental operations:
 - assignment.
 - dereferencing.
- Assignment is used to set a pointer variable's value to some useful address:
 - `int *ptr = &counter; // indirect addressing.`
 - `int *ptr = new int; // heap-dynamic variable.`
- Dereferencing yields the value stored at the location represented by the pointer's value
 - C++ uses an explicit operation via unary operator `*`:
`j = *ptr; // sets j to the value located at ptr`

Pointer Dereferencing



The dereferencing operation $j = *ptr;$

Problems with Pointers

- Dangling pointers:
 - A pointer points to a heap-dynamic variable that has been deallocated.
 - Dangerous: the location may be assigned to other variables.
- Lost heap-dynamic variable:
 - An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage* or *memory leak*):
 - Pointer `p1` is set to point to a newly created heap-dynamic variable
 - Pointer `p1` is later set to point to another newly created heap-dynamic variable, without deallocating the first one.

Pointers in C/C++

- Extremely flexible but must be used with care:
 - Pointers can point at any variable regardless of when or where it was allocated.
 - Used for dynamic storage management and addressing.
 - Explicit dereferencing (*) and address-of (&) operators.
 - Domain type need not be fixed:
 - `void *` can point to any type and can be type checked.
 - `void *` cannot be de-referenced.
 - Pointer arithmetic is possible.

Pointer Arithmetic in C/C++

```
float stuff[100];  
float *p;  
p = stuff;
```

`*(p+5)` is equivalent to `stuff[5]` and `p[5]`

`*(p+i)` is equivalent to `stuff[i]` and `p[i]`

Reference Types

- C++ includes a special kind of pointer type called a **reference type** that is used primarily for formal parameters:
 - Advantages of both pass-by-reference and pass-by-value.
 - No arithmetic on references.
- Java extends C++'s reference variables and allows them to replace pointers entirely:
 - References are handles to objects, rather than being addresses.
- C# includes both the references of Java and the pointers of C++.

Evaluation of Pointers & References

- Problems due to dangling pointers and memory leaks.
- Heap management can be complex and costly.
- Pointers are analogous to goto's:
 - goto's widen the range of statements that can be executed next.
 - pointers widen the range of cells that can be accessed by a variable.
- Pointers or references are necessary for dynamic data structures, so we can't design a language without them:
 - pointers are essential for writing device drivers.
 - references in Java and C# provide some of the capabilities of pointers, without the hazards.

Reading Assignment

- Chapter 6.9 (also ordinal types, subrange types, associative arrays, unions).