CS2510
Modern Programming Languages

Lecture 6

Data Types

Topics

- Introduction
- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Associative Arrays
- Record Types
- Tuple Types
- List Types
- Union Types
- Pointer and Reference Types
- Type Checking
- Strong Typing
- Type Equivalence
- Theory and Data Types

Introduction

- *Data type* defines a collection of data objects and a set of predefined operations on those objects
- *Descriptor*: collection of attributes of a variable
- *Object*: an instance of a user-defined (abstract data) type

Design issue for all data types:

- What operations are defined?
- How are operations specified?

Primitive Data Types

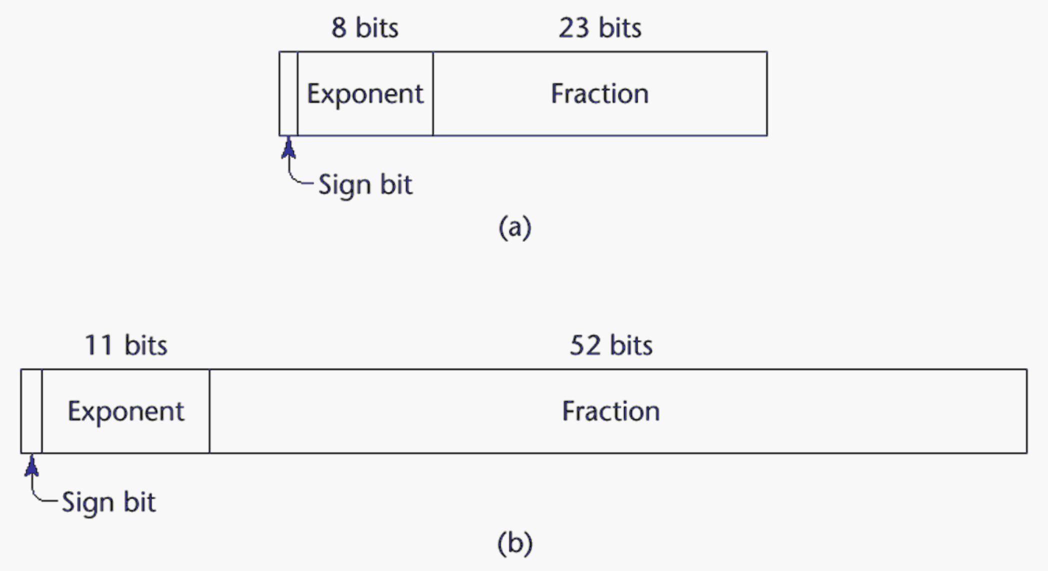
- Almost all programming languages provide a set of *primitive data types*
- Primitive data types: not defined in terms of other data types
- Some primitive data types come from hardware
- Others require only a little non-hardware support for their implementation

Primitive Data Types: Integer

- Almost always an exact reflection of the hardware
  - Mapping is trivial
- As many as 8 different integer types in some languages
- Java's signed integer types:
  - **`byte`**,
  - **`short`**,
  - **`int`**,
  - **`long`**

Primitive Data Types: Floating Point

- Represent real numbers
  - Only an approximation!
- Scientific languages support at least 2 floating-point types (sometimes more)
  - E.g., **`float`** and **`double`**
- Usually exactly like the hardware, but not always
- IEEE Floating-Point Standard 754



Primitive Data Types: Complex

- Some languages (Python) support complex type
- Each value consists of two floating-point values
  - Real part
  - Imaginary part
- Literal form (in Python):

```
(7 + 3j)
```

where **7** is the real part and **3** is the imaginary part

Primitive Data Types: Decimal

- For business applications (money)
  - Essential to COBOL
  - C# offers a decimal data type
- Store a fixed number of decimal digits, in coded form (BCD)
- *Advantage*: accuracy
- *Disadvantages*: limited range, wastes memory

Primitive Data Types: Boolean

- Simplest of all
- Range of values:
  - Two elements, one for “true” and one for “false”
- Could be implemented as bits, but often as bytes
- Advantage: readability

Primitive Data Types: Character

- Stored as numeric codings
- Most commonly used coding: ASCII
- An alternative, 16-bit coding: Unicode (UCS-2)
  - Includes characters from most natural languages
  - Originally used in Java
  - C# and JavaScript also support Unicode
- 32-bit Unicode (UCS-4)
  - Supported by Fortran (from 2003 onwards)

Character String Types

- Values are sequences of characters
- Design issues:
  - Is it a primitive type or just a special kind of array?
  - Should the length of strings be static or dynamic?

Character String Types Operations

- Typical operations:
  - Assignment and copying
  - Comparison (=, >, etc.)
  - Catenation
  - Substring reference
  - Pattern matching

Character String Type in Languages

- C and C++
  - Not primitive
  - **`char`** arrays and a library of operations
- SNOBOL4 (a string manipulation language)
  - Primitive
  - Many operations, including elaborate pattern matching
- Fortran and Python
  - Primitive with assignment and several operations
- Java
  - Primitive via the **`String`** class
- Perl, JavaScript, Ruby, and PHP
  - Built-in pattern matching via regular expressions

Character String Length Options

- *Static*
  - COBOL, Java's **`String`** class
- *Limited Dynamic Length*
  - C and C++
  - A special character indicates the end of the string
- *Dynamic* (no maximum)
  - SNOBOL4, Perl, JavaScript
- Ada supports all three string length options

Character String Type Evaluation

- Aid to writability
- As a primitive type with static length, they are inexpensive to provide so why not have them?
- Dynamic length is nice, but is it worth the expense?

Character String Implementation

- Static length:
  - Compile-time descriptor
- Limited dynamic length:
  - May need a run-time descriptor for length
  - … But not in C or C++…
- Dynamic length:
  - Need run-time descriptor
  - (De)allocation biggest implementation problem

Compile- and Run-Time Descriptors

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

User-Defined Ordinal Types

- Ordinal type
  - A range of values associated with the positive integers
- Examples of primitive ordinal types in Java
  - **`integer`**
  - **`char`**
  - **`boolean`**

Enumeration Types

- All possible values (named constants) provided
- C# example

```
enum days {mon, tue, wed, thu, fri, sat, sun};
```

- Design issues
  - Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant checked?
  - Are enumeration values coerced to integer?
  - Any other type coerced to an enumeration type?

Evaluation of Enumerated Type

- Aid to readability
  - E.g., no need to code a color as a number
- Aid to reliability, e.g., compiler can check:
  - Operations (don’t allow colors to be added)
  - No enumeration variable can be assigned a value outside its defined range
  - Ada, C#, and Java 5.0 provide better support for enumeration than C++ because enumeration type variables in these languages are not coerced into integer types

Subrange Types

- Ordered contiguous subsequence of ordinal types
  - Example: 12..18 is a subrange of integer type
- Ada's design

```
type Days is (mon, tue, wed, thu, fri, sat, sun);
subtype Weekdays is Days range mon..fri;
subtype Index is Integer range 1..100;

Day1: Days;
Day2: Weekday;
Day2 := Day1;
```

Subrange Evaluation

- Aid to readability
  - Make it clear to the readers that variables of subrange can store only certain range of values
- Reliability
  - Assigning a value to a subrange variable that is outside the specified range is detected as an error

Implementing User-Defined Ordinal Types

- Enumeration types are implemented as integers
- Subrange types are implemented like the parent types with code inserted (by the compiler) to restrict assignments to subrange variables

Array Types

- An array is a homogeneous aggregate of data elements
- An individual element is identified by its position in the aggregate, relative to the first element

Array Design Issues

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does allocation take place?
- Are ragged or rectangular multidimensional arrays allowed, or both?
- What is the maximum number of subscripts?
- Can array objects be initialised?
- Are any kind of slices supported?

Array Indexing

Indexing (or subscripting)

- Mapping from indices to elements

  array_name (index_value_list) $\rightarrow$ an element

Index Syntax

- Fortran and Ada use parentheses
  - Ada explicitly uses parentheses to show uniformity between array references and function calls because both are *mappings*
- Most other languages use brackets

Arrays Index (Subscript) Types

- FORTRAN, C: integer only
- Ada: integer or enumeration (includes Boolean and char)
- Java: integer types only
- 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

Subscript Binding & Array Categories

- *Static*:
  - Subscript ranges statically bound and storage allocation is static (before run-time)
  - Advantage: efficiency (no dynamic allocation)
- *Fixed stack-dynamic*:
  - Subscript ranges statically bound, but the allocation is done at declaration time
  - Advantage: space efficiency

Subscript Binding & Array Categs. (cont')

- *Stack-dynamic*:
  - Subscript ranges are dynamically bound and the storage allocation is dynamic (done at run-time)
  - Advantage: flexibility (the size of an array need not be known until the array is to be used)
- *Fixed heap-dynamic*:
  - Similar to fixed stack-dynamic: storage binding is dynamic but fixed after allocation
  - That is, binding is done when requested and storage is allocated from heap, not stack

Subscript Binding & Array Categs. (cont')

- Heap-dynamic:
  - Binding of subscript ranges and storage allocation is dynamic and can change any number of times
  - Advantage: flexibility (arrays can grow or shrink during program execution)

Subscript Binding & Array Categs. (cont')

- C and C++ arrays that include `static` modifier are static
- C and C++ arrays without `static` modifier are fixed stack-dynamic
- C and C++ provide fixed heap-dynamic arrays
- C# includes a second array class `ArrayList` that provides fixed heap-dynamic
- Perl, JavaScript, Python, and Ruby support heap-dynamic arrays

Array Initialisation

- Some language allow initialization at the time of storage allocation
  - C, C++, Java, C# example

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

  - Character strings in C and C++

```
char name [] = "freddie";
```

  - Arrays of strings in C and C++

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

  - Java initialization of String objects

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

Heterogeneous Arrays

- A *heterogeneous array* is one in which the elements need not be of the same type
- Supported by Perl, Python, JavaScript, and Ruby

Array Initialization

- C-based languages

```
int list [] = {1, 3, 5, 7}
char *names [] = {"Mike", "Fred", "Mary Lou"};
```

- Ada

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

- Python (list comprehensions)

```
list = [x ** 2 for x in range(12) if x % 3 == 0]
```

  – assigns `[0, 9, 36, 81]` to `list`

Arrays Operations

- APL
  - Most powerful array processing operations for vectors and matrixes as well as unary operators
  - For example, to reverse column elements
- Ada allows array assignment but also catenation
- Python/Ruby support array catenation
- Fortran provides *elemental* operations because they are between pairs of array elements
  - For example, + operator between two arrays results in an array of the sums of the element pairs of the two arrays

Rectangular and Jagged Arrays

- Rectangular array
  - Multi-dimensioned array in which all rows and columns have same number of elements
- Jagged matrix
  - Rows with varying number of elements
  - Possible when multi-dimensioned arrays actually appear as arrays of arrays
- C, C++, and Java support jagged arrays
- Fortran, Ada, and C# support rectangular arrays (C# also supports jagged arrays)

Slices

- A slice is some substructure of an array; nothing more than a referencing mechanism
- Slices are only useful in languages that have array operations

Slice Examples

- Python

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

  `vector (3:6)` is a three-element array

  `mat[0][0:2]` is the first and second element of the first row of `mat`

- Ruby supports slices with the `slice` method

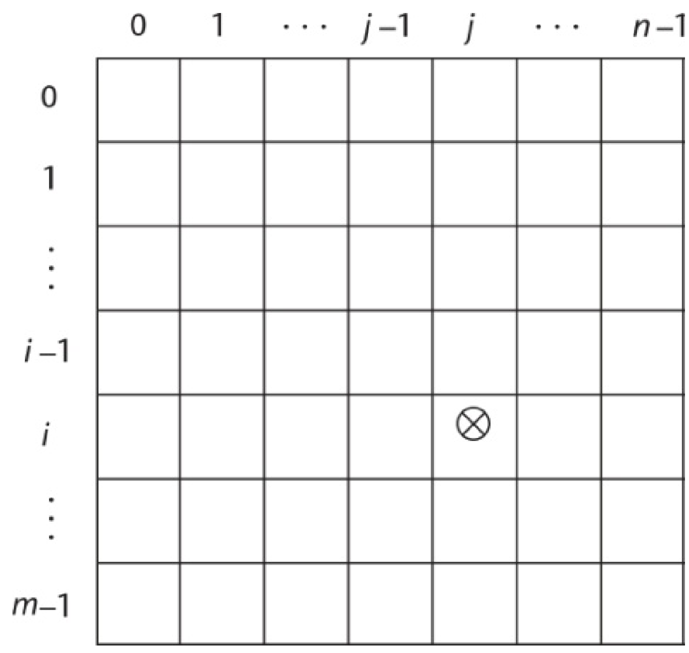
```
list.slice(2, 2)
```

  returns the third and fourth elements of `list`

Implementation of Arrays

- Access function maps subscript expressions to an address in the array
- Access function for single-dimensioned arrays:

address(list[k]) = address (list[lower_bound]) + ((k-lower_bound) * elem_size)



Accessing Multi-dimensioned Arrays

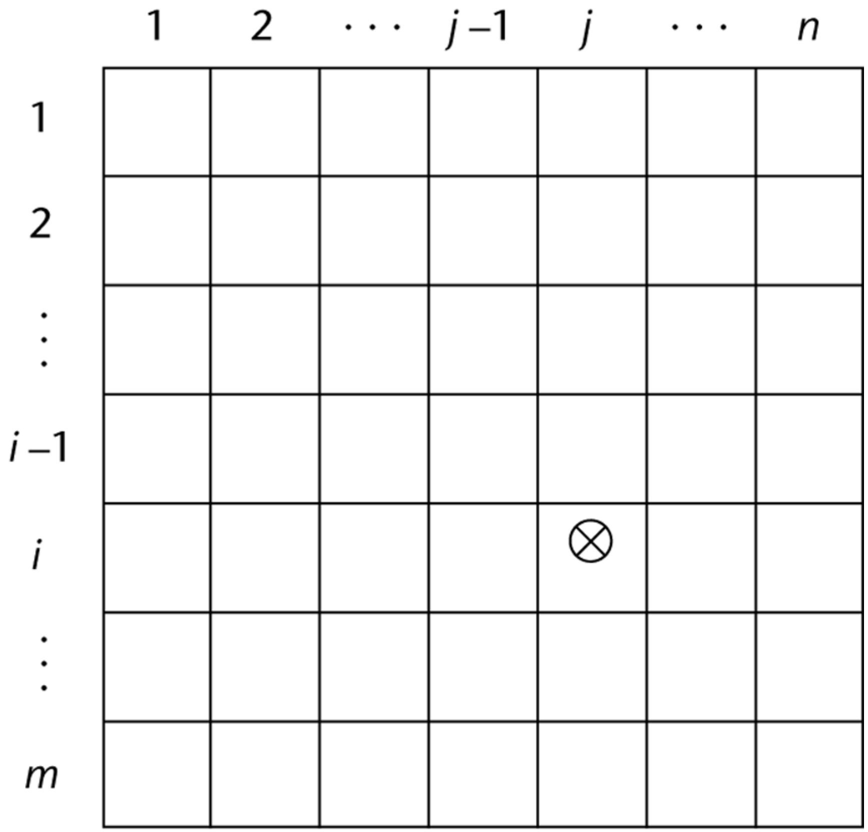
- Two common ways:
  - Row major order (by rows) – used in most languages
  - Column major order (by columns) – used in Fortran
  - A compile-time descriptor for a multidimensional array

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 0
⋮
Index range n – 1
Address

Locating elems (multi-dimension arrays)

- General format

  Location (a[I,j]) = address of a [row_lb,col_lb] + (((I - row_lb) * n) + (j - col_lb)) * element_size



Compile-Time Descriptors

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

Single-dimensioned array

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

Multidimensional array

Associative Arrays

- *Associative array*
  - Unordered collection of data elements indexed by an equal number of values called *keys*
  - User-defined keys must be stored
- Design issues:
  - What is the form of references to elements?
  - Is the size static or dynamic?
- Built-in type in Perl, Python, Ruby, and Lua
  - In Lua, they are supported by tables

Associative Arrays in Perl

- Names begin with "%"; literals are delimited by parentheses

```
%hi_temps = ("Mon" => 77, "Tue" => 79, "Wed" => 65, …);
```

- Subscripting is done using braces and keys

```
$hi_temps{"Wed"} = 83;
```

  - Elements can be removed with **`delete`**

```
delete $hi_temps{"Tue"};
```

Record Types

- A *record* is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names
- Design issues:
  - What is the syntactic form of references to the field?
  - Are elliptical references allowed

Definition of Records in COBOL

- COBOL uses level numbers to represent nesting

```
01 EMP-REC.
   02 EMP-NAME.
      05 FIRST PIC X(20).
      05 MID   PIC X(10).
      05 LAST  PIC X(20).
   02 HOURLY-RATE PIC 99V99.
```

- Other languages may use recursive definitions

Definition of Records in Ada

- Record structures indicated in an orthogonal way

```
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;
```

References to Records

Record field references

- COBOL

  field_name **OF** record_name_1 **OF** ... **OF** record_name_n

- Others (dot notation)

  record_name_1.record_name_2. ... record_name_n.field_name

- Fully qualified references must include all record names
  - Elliptical references allow leaving out record names as long as the reference is unambiguous
  - In COBOL , `FIRST`, `FIRST OF EMP-NAME`, and `FIRST` of `EMP-REC` are elliptical references to the employee's first name

Operations on Records

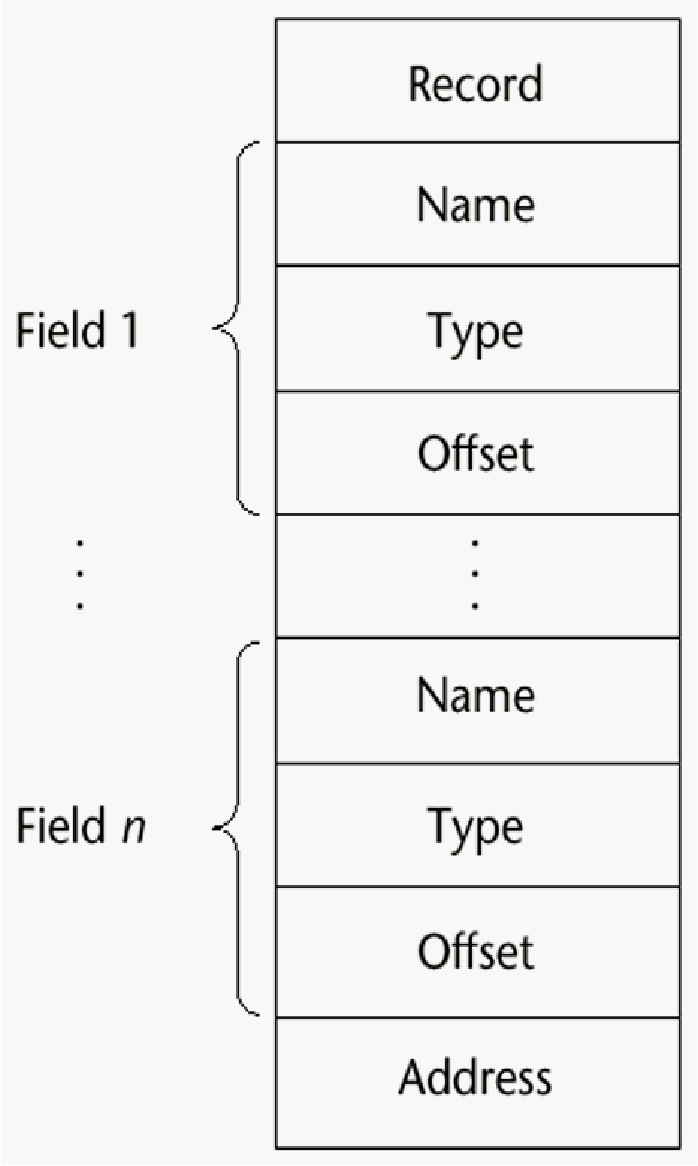
- Assignment is very common if types are identical
- Ada allows record comparison
- Ada records can be initialised with aggregate literals
- COBOL provides `MOVE CORRESPONDING`
  - Copies a field of the source record to the corresponding field in the target record

Record vs. Arrays

- Records are used when collection of data values is heterogeneous
- Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)
- Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower

Implementation of Record Type

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



Tuple Types

- A tuple is a data type that is similar to a record, except that the elements are not named
- Used in Python, ML, and F# to allow functions to return multiple values
- Python
  - Closely related to its lists, but immutable
  - Create with a tuple literal

```
myTuple = (3, 5.8, 'apple')
```

  - Referenced with subscripts (begin at `1`)
  - Catenation with + and deleted with **`del`**

Tuple Types (continued)

- ML

```
val myTuple = (3, 5.8, 'apple');
```

  - Access as follows:

    `#1(myTuple)` is the first element

  - A new tuple type can be defined

```
type intReal = int * real;
```

- F#

```
let tup = (3, 5, 7)
```

`let a, b, c = tup` assigns tuple to tuple pattern `(a, b, c)`

List Types

- Lists in LISP and Scheme delimited by parentheses and use no commas

  `(A B C D)` and `(A (B C) D)`

- Data and code have the same form
  - As data `(A B C)` is what it is (a collection with 3 elements)
  - As code `(A B C)` is function A applied to parameters `B` and `C`
- Interpreter needs to differentiate these
  - If it is data, we quote it with an apostrophe

    `'(A B C)`

List Types (continued)

- List Operations in Scheme
  - `CAR` returns the first element of its list parameter

    `(CAR '(A B C))` returns `A`

  - `CDR` returns the remainder of its list parameter after the first element has been removed

    `(CDR '(A B C))` returns `(B C)`

  - `CONS` puts its 1st parameter into its 2nd parameter, a list, to make a new list

    `(CONS 'A (B C))` returns `(A B C)`

  - `LIST` returns a new list of its parameters

    `(LIST 'A 'B '(C D))` returns `(A B (C D))`

List Types (continued)

- List Operations in ML
  - Lists are written in brackets and elements separated by commas
  - List elements must be of the same type
  - The Scheme `CONS` function is a binary operator in ML, `::`

    `3 :: [5, 7, 9]` evaluates to `[3, 5, 7, 9]`
  - The Scheme `CAR` and `CDR` functions are named `hd` and `tl`, respectively

List Types (continued)

- F# Lists
  - As ML, except elements separated by semicolons
  - `hd` and `tl` are methods of the `List` class
- Python Lists
  - The list data type also serves as Python's arrays
  - Unlike Scheme, Common LISP, ML, and F#, Python's lists are mutable
  - Elements can be of any type
  - Create a list with an assignment

```
myList = [3, 5.8, "grape"]
```

List Types (continued)

- Python Lists (continued)
  - List elements are referenced with subscripting, with indices beginning at zero

    `x = myList[1]` Sets `x` to `5.8`
  - List elements can be deleted with `del`

    `del myList[1]`
  - List Comprehensions – derived from set notation

    `[x * x for x in range(6) if x % 3 == 0]`

    `range(12)` creates `[0, 1, 2, 3, 4, 5, 6]`

    Constructed list: `[0, 9, 36]`

List Types (continued)

- Haskell's List Comprehensions
  - The original

    `[n * n | n <- [1..10]]`
- F#'s List Comprehensions

```
let myArray = [|for i in 1 .. 5 -> [i * i) |]
```

- C# and Java support lists through generic heap-dynamic collection classes
  - `List` and `ArrayList`, respectively

Unions Types

- A *union* is a type whose variables are allowed to store different type values at different times during execution
- Design issues
  - Should type checking be required?
  - Should unions be embedded in records?

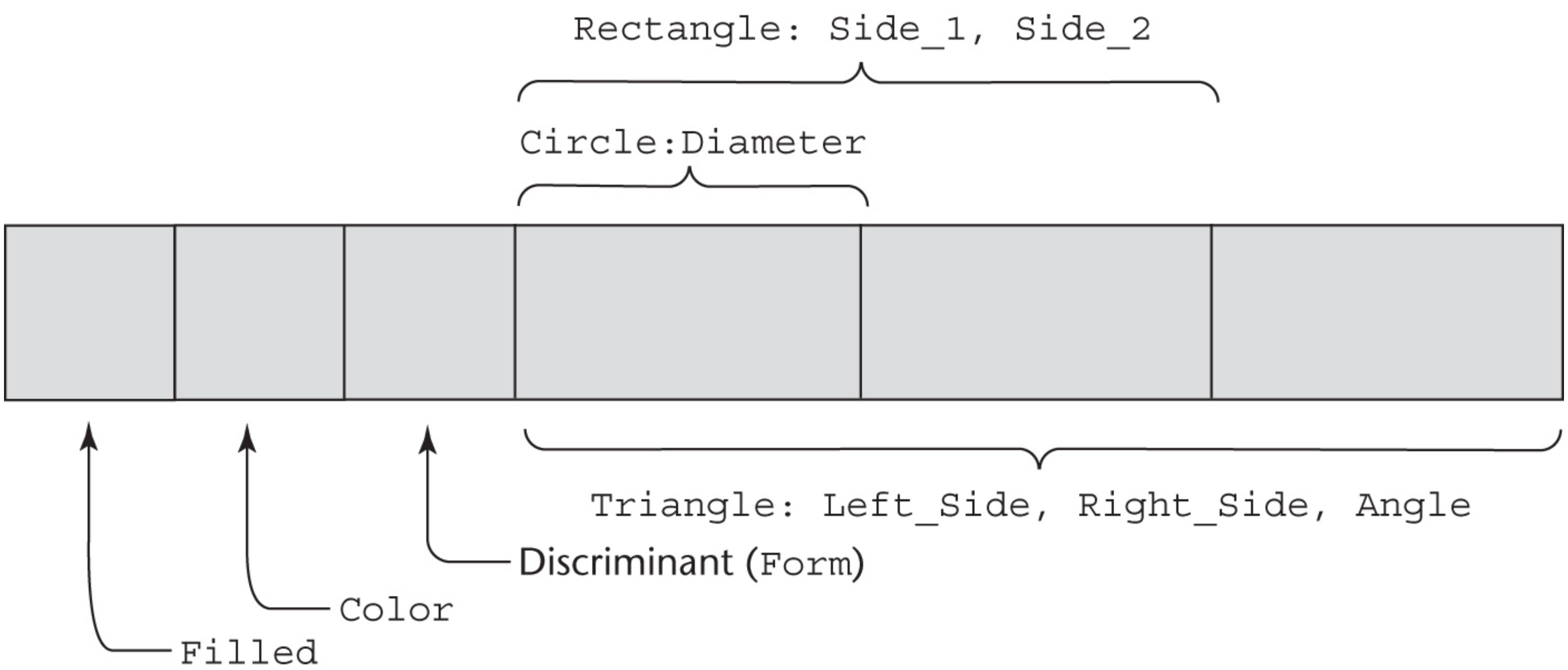
Discriminated vs. Free Unions

- Fortran, C, and C++ provide union constructs in which there is no language support for type checking;
  - Union in these languages is called *free union*
- Type checking of unions require that each union include a type indicator called a *discriminant*
  - Supported by Ada

Ada Union Types

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
  Filled: Boolean;
  Color: Colors;
  case Form is
     when Circle => Diameter: Float;
     when Triangle =>
          Leftside, Rightside: Integer;
          Angle: Float;
     when Rectangle => Side1, Side2: Integer;
  end case;
end record;
```

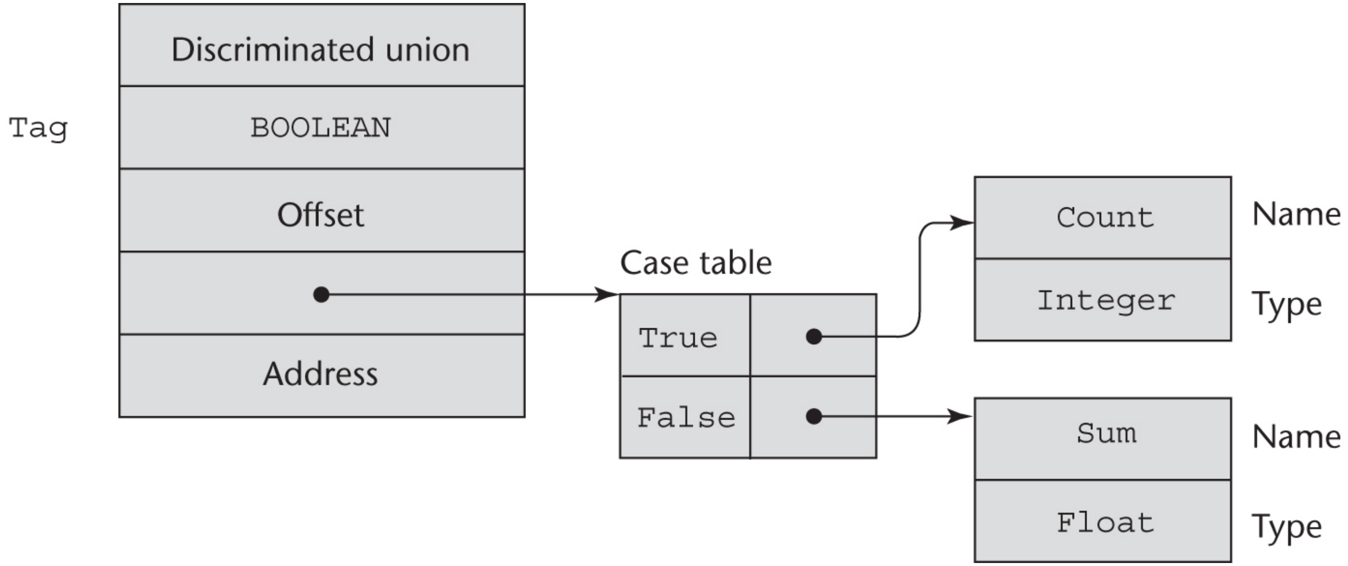
Ada Union Type Illustrated



A discriminated union of three shape variables

Implementation of Unions

```
type Node (Tag : Boolean) is
  record
    case Tag is
      when True => Count : Integer;
      when False => Sum : Float;
    end case;
  end record;
```



Evaluation of Unions

- Free unions are unsafe
  - Do not allow type checking
- Java and C# do not support unions
  - Reflective of growing concerns for safety in programming language
- Ada's descriminated unions are safe

Pointer and Reference Types

- A *pointer* type variable has a range of values (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
  - Usually called a *heap*

Design Issues of Pointers

- What are the scope of and lifetime of a pointer variable?
- What is the lifetime of a heap-dynamic variable?
- Are pointers restricted as to the type of value to which they can point?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer types, reference types, or both?

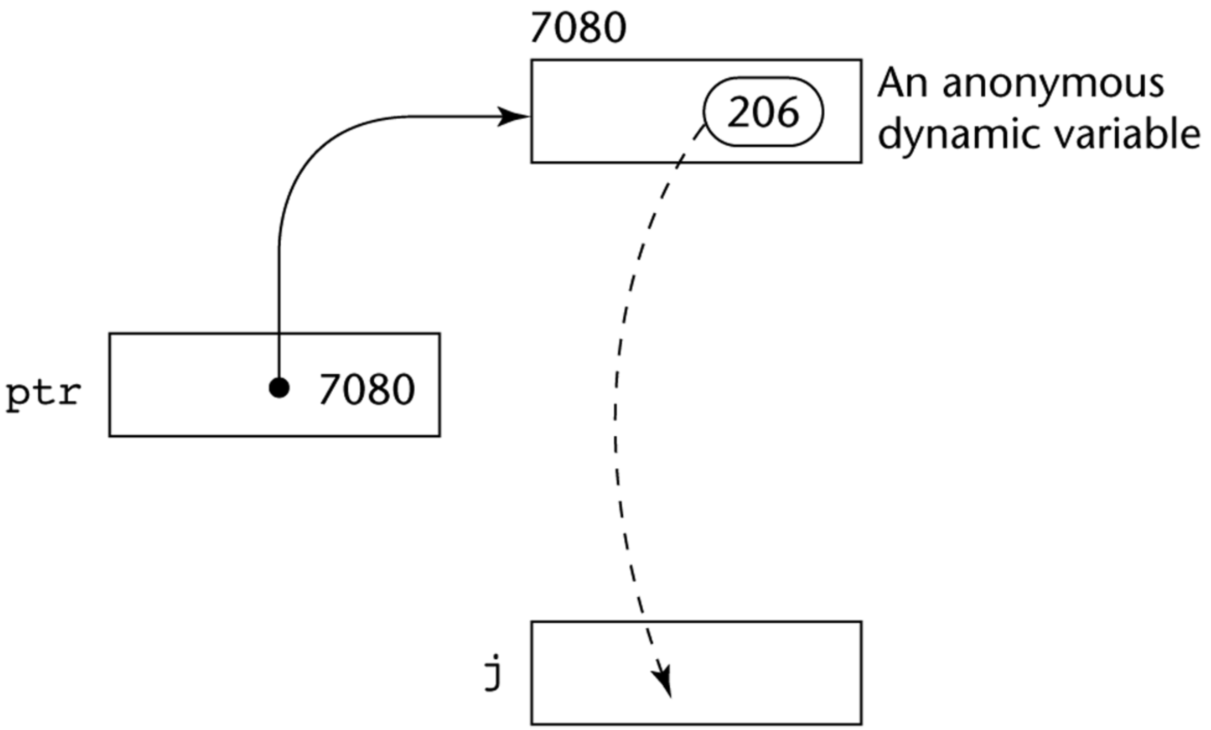
Pointer Operations

- Two fundamental operations:
  - Assignment
  - Dereferencing
- Assignment: set a pointer variable's value to some useful address
- Dereferencing: yields the value stored at the location represented by the pointer's value
  - Dereferencing can be explicit or implicit
  - C++ uses an explicit operation via *

    ```
    j = *ptr
    ```

    sets j to the value located at `ptr`

Pointer Assignment Illustrated



The assignment operation j = *ptr

Problems with Pointers

- Dangling pointers (dangerous)
  - A pointer points to a heap-dynamic variable that has been deallocated
- Lost heap-dynamic variable
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)
    - 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
  - The process of losing heap-dynamic variables is called *memory leakage*

Pointers in Ada

- Some dangling pointers are disallowed because dynamic objects can be automatically deallocated at the end of pointer's type scope
- The lost heap-dynamic variable problem is not eliminated in Ada
  - It is possible with `UNCHECKED_DEALLOCATION`

Pointers in C and 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
- Pointer arithmetic is possible
- Explicit dereferencing and address-of operators
- Domain type need not be fixed (`void *`)
- `void *` can point to any type and can be type-checked (cannot be de-referenced)

Pointer Arithmetic in C and 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++ has a special kind of pointer type called a *reference type* used primarily for formal parameters
  - Advantages of both pass-by-reference and pass-by-value
- Java extends C++'s reference variables and allows them to replace pointers entirely
  - References are references to objects, rather than being addresses
- C# includes both the references of Java and the pointers of C++

Evaluation of Pointers

- Dangling pointers and dangling objects are problems as is heap management
- Pointers are like `goto`'s in that they 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

Representations of Pointers

- Large computers use single values
- Intel microprocessors use segment and offset

Dangling Pointer Problem

- *Tombstone*: extra heap cell that is a pointer to the heap-dynamic variable
  - The actual pointer variable points only at tombstones
  - When heap-dynamic variable de-allocated, tombstone remains but set to nil
  - Costly in time and space
- *Locks-and-keys*: Pointer values are represented as (key, address) pairs
  - Heap-dynamic variables are represented as variable plus cell for integer lock value
  - When heap-dynamic variable allocated, lock value is created and placed in lock cell and key cell of pointer

Heap Management

- A very complex run-time process
- Single-size cells vs. variable-size cells
- Two approaches to reclaim garbage
  - Reference counters (*eager approach*): reclamation is gradual
  - Mark-sweep (*lazy approach*): reclamation occurs when the list of variable space becomes empty

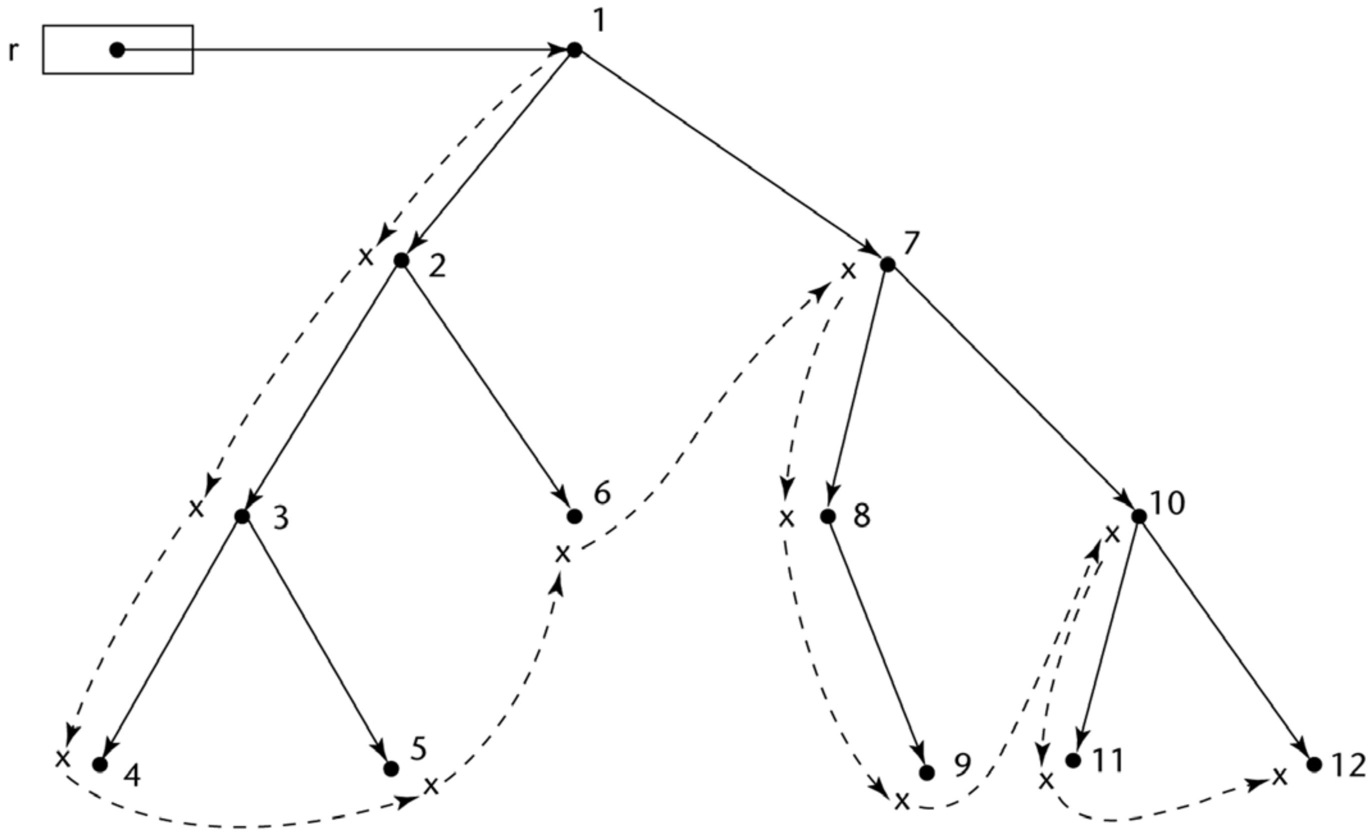
Reference Counter

- Reference counters:
  - maintain a counter in every cell that store the number of pointers currently pointing at the cell
- *Disadvantages*:
  - space required,
  - execution time required,
  - complications for cells connected circularly
- *Advantage*:
  - it is intrinsically incremental, so significant delays in the application execution are avoided

Mark-Sweep

- The run-time system allocates storage cells as requested and disconnects pointers from cells as necessary; mark-sweep then begins
  - Every heap cell has an extra bit used by collection algorithm
  - All cells initially set to garbage
  - All pointers traced into heap, and reachable cells marked as not garbage
  - All garbage cells returned to list of available cells
  - Disadvantages: in its original form, it was done too infrequently. When done, it caused significant delays in application execution. Contemporary mark-sweep algorithms avoid this by doing it more often (called incremental mark-sweep)

Marking Algorithm



Dashed lines show the order of node_marking

Variable-Size Cells

- All the difficulties of single-size cells plus more
- Required by most programming languages
- If mark-sweep is used, additional problems occur
  - The initial setting of the indicators of all cells in the heap is difficult
  - The marking process in nontrivial
  - Maintaining the list of available space is another source of overhead

Type Checking

- Generalise the concept of operands and operators to include subprograms and assignments
- *Type checking* is the activity of ensuring that the operands of an operator are of compatible types
- A *compatible type* is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler- generated code, to a legal type
  - This automatic conversion is called a *coercion*.
- A *type error* is the application of an operator to an operand of an inappropriate type

Type Checking (continued)

- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic
- A programming language is *strongly typed* if type errors are always detected
- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors

Strong Typing

Language examples:

- C and C++ are not: parameter type checking can be avoided; unions are not type checked
- Ada is, almost (`UNCHECKED CONVERSION` is loophole)
  (Java and C# are similar to Ada)

Strong Typing (continued)

- Coercion rules strongly affect strong typing; they can weaken it considerably (C++ versus Ada)
- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada

Name Type Equivalence

- *Name type equivalence* means 2 variables have equivalent types if
  - They are in either the same declaration or
  - In declarations that use the same type name
- Easy to implement but highly restrictive:
  - Subranges of integer types are not equivalent with integer types
  - Formal parameters must be the same type as their corresponding actual parameters

Structure Type Equivalence

- *Structure type equivalence* means that two variables have equivalent types if their types have identical structures
- More flexible, but harder to implement

Type Equivalence (continued)

Consider the problem of two structured types:

- Are two record types equivalent if they are structurally the same but use different field names?
- Are two array types equivalent if they are the same except that the subscripts are different?

  (e.g. `[1..10]` and `[0..9]`)
- Are two enumeration types equivalent if their components are spelled differently?
- With structural type equivalence, you cannot differentiate between types of the same structure (e.g. different units of speed, both float)

Theory and Data Types

- Type theory: broad area of study in mathematics, logic, computer science, and philosophy
- Two branches of type theory in computer science:
  - Practical – data types in commercial languages
  - Abstract – typed lambda calculus
- A type system is a set of types and the rules that govern their use in programs

Theory and Data Types (continued)

- Formal model of a type system is a set of types and a collection of functions that define the type rules
  - Either an attribute grammar or a type map could be used for the functions
  - Finite mappings – model arrays and functions
  - Cartesian products – model tuples and records
  - Set unions – model union types
  - Subsets – model subtypes

Summary

- Data types of a language are a large part of what determines that language's style and usefulness
- Primitive data types of most imperative languages include numeric, character, and Boolean types
- User-defined enumeration & subrange types are convenient;
  - Improve readability & reliability of programs
- Arrays and records included in most languages
- Pointers for addressing flexibility and to control dynamic storage management