Chapter 6: Data Types

Principles of Programming Languages

Contents

- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Associative Arrays
- Record Types
- Union Types
- Pointer and Reference Types
- Type Checking

Introduction

- A *data type* defines
  - a collection of data objects
  - a set of predefined operations
- Abstract Data Type: Interface (visible) are separated from the representation and operation (hidden)
- Uses of type system:
  - Error detection
  - Program modularization assistant
  - Documentation

The Next Part

- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Associative Arrays
- Record Types
- Union Types
- Pointer and Reference Types
- Type Checking

Primitive Data Types

- Data types that are not defined in terms of other data types
- Some primitive data types are merely reflections of the hardware
- Others require little non-hardware support

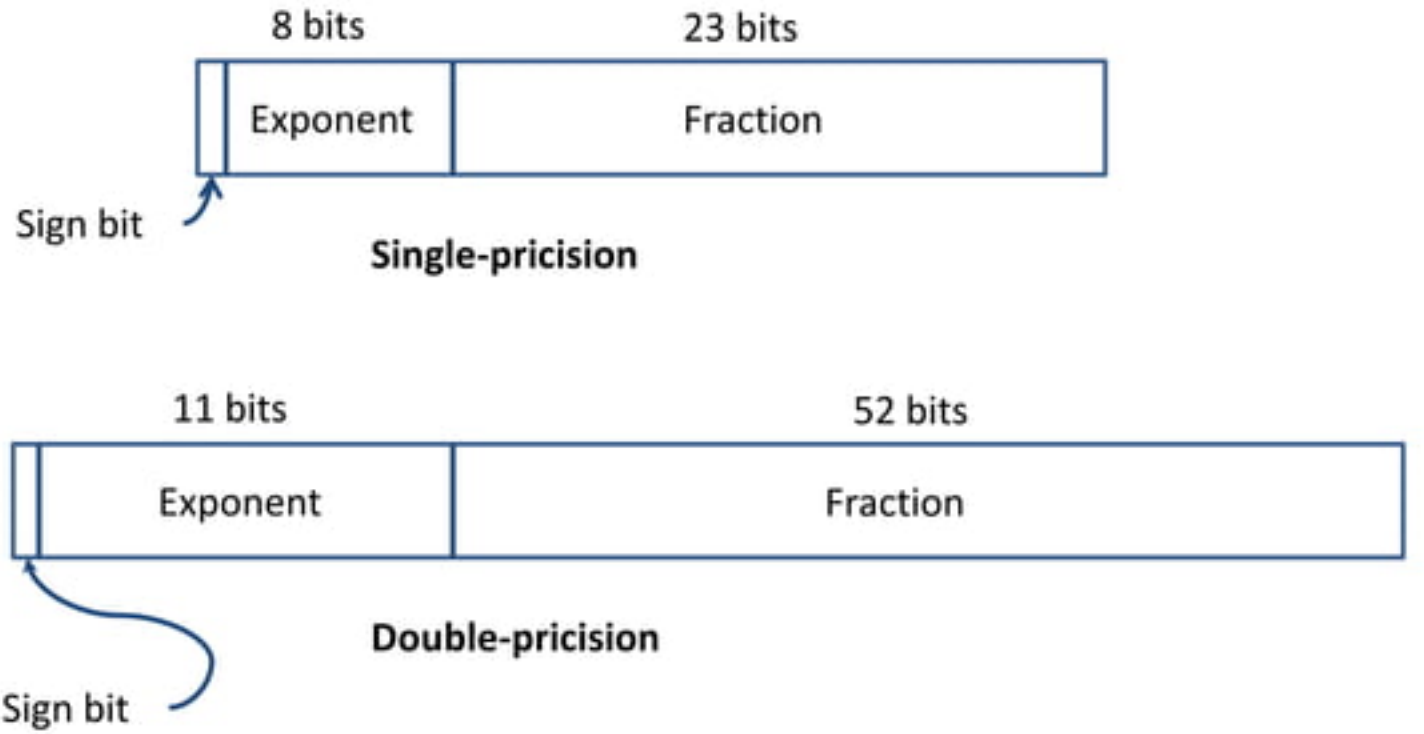
Primitive Data Types: Integer

- Languages may support several sizes of integer
  - Java's signed integer sizes: `byte`, `short`, `int`, `long`
- Some languages include unsigned integers
- Supported directly by hardware: a string of bits
- To represent negative numbers: **twos complement**

Primitive Data Types: Floating Point

- Model real numbers, but only as approximations
- Languages for scientific use support at least two floating-point types (e.g., `float` and `double`)
- Precision and range
- IEEE Floating-Point Standard 754

IEEE 754


8 bits
23 bits
Exponent
Fraction
Sign bit
Single-pricision
11 bits
52 bits
Exponent
Fraction
Double-pricision
Sign bit

Primitive Data Types: Decimal

- For business applications (money)
  - Essential to COBOL
  - C# offers a decimal data type
- Store a fixed number of decimal digits
- *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
  - Includes characters from most natural languages
  - Originally used in Java
  - C# and JavaScript also support Unicode

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
  - Comparison (=, >, etc.)
  - Catenation
  - Substring reference
  - Pattern matching
    - Regular Expression

Character String in Some Languages

- C and C++
  - Not primitive
  - Use **`char`** arrays and a library of functions that provide operations
- Java
  - Primitive via the `String` class
  - Immutable

Character String Length Options

- *Static*: Python, Java's `String` class
- *Limited Dynamic Length*: C
  - In C, a special character is used to indicate the end of a string's characters, rather than maintaining the length
- *Dynamic* (no maximum): Perl, JavaScript, standard C++ library
- Ada supports all three string length options

Character String Type Evaluation

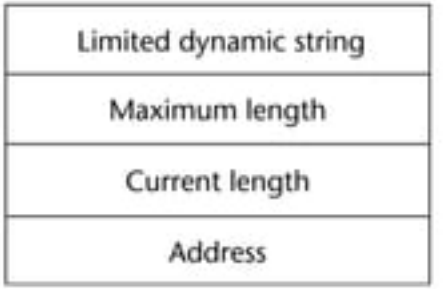
- Aid to writability
- As a primitive type with static length, they are inexpensive to provide--why not have them?
- Dynamic length is nice, but is it worth the expense?

Character String Implementation

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

Descriptor

Compile-time descriptor for static strings



Run-time descriptor for limited dynamic strings

User-Defined Ordinal Types

- An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers
- Examples of primitive ordinal types in Java
  - `integer`
  - `char`
  - `boolean`

Enumeration Types

- All possible values, which are named constants, are provided in the definition
- C# example

```
enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
days myDay = Mon, yourDay = Tue;
```

- Design issues:
  - Is an enumeration constant allowed to appear in more than one type definition?
  - Are enumeration values coerced to integer?
  - Are any other types coerced to an enumeration type?

Evaluation of Enumerated Type

- Readability
  - no need to code a color as a number
- Reliability
  - Ada, C# and Java 5.0
    - operations (don't allow colors to be added)
    - No enumeration variable can be assigned a value outside its defined range
    - Better support for enumeration than C++: enumeration type variables are not coerced into integer types

Implementation of User-Defined Ordinal Types

- Enumeration types are implemented as integers

The Next Part

- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Associative Arrays
- Record Types
- Union Types
- Pointer and Reference Types
- Type Checking

Array Types

- Collection of homogeneous data elements
- Each element is identified by its position relative to the first element
- Homogeneous: data elements are of same type
- Referenced using subscript expression

Array Indexing

- *Indexing* (or subscripting) is a mapping from indices to elements

  `array_name (index_value_list) → an element`

- Index Syntax
  - Fortran, Ada use parentheses
  - Most other languages use brackets

Arrays Index (Subscript) Types

- *What type are legal for subscripts?*
- Pascal, Ada: any ordinal type (integer, boolean, char, enumeration)
- Others: subrange of integers
- *Are subscripting expressions range checked?*
- Most contemporary languages do not specify range checking but Java, ML, C#
- Unusual case: Perl

Subscript Binding & Array Categories

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

Subscript Binding & Array Categories

- *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 (i.e., binding is done when requested and storage is allocated from heap, not stack)

Subscript Binding & Array Categories

- *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 Categories

- C and C++ arrays that include `static` modifier are static
- C and C++ arrays without `static` modifier are fixed stack-dynamic
- Ada arrays can be stack-dynamic

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

Subscript Binding & Array Categories

- C and C++ provide fixed heap-dynamic arrays
  - `malloc` and `free`, `new` and `delete`
- All arrays in Java are fixed heap-dynamic
- C# includes a array class `ArrayList` that provides heap-dynamic

```
ArrayList intList = new ArrayList();
intList.Add(nextOne);
```

- Perl and JavaScript support heap-dynamic arrays

Array Initialization

- 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"};
```

Arrays Operations

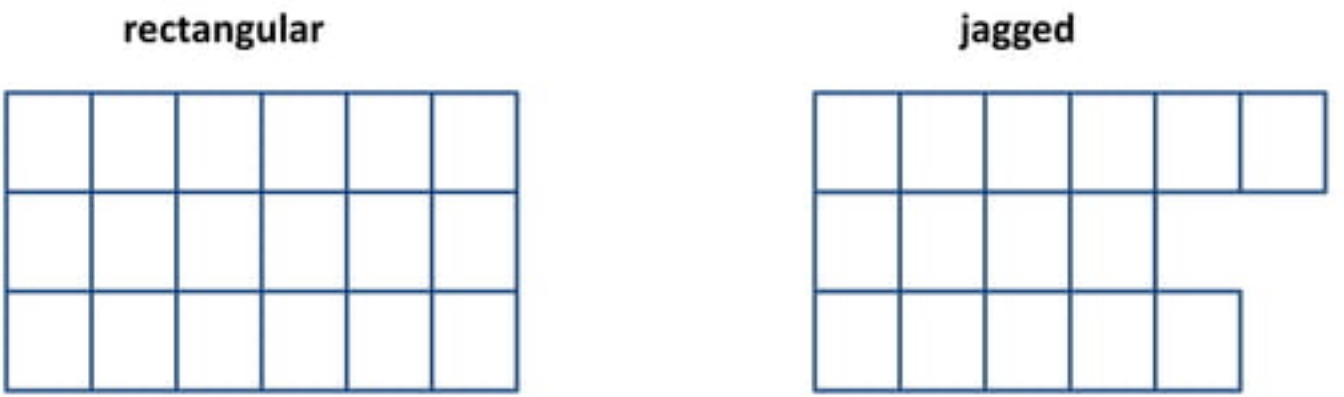
- Assignment, catenation, comparison for equality, and slices
- C-based languages do not provide any array operations
- Java, C++ and C# use methods to manipulate

Rectangular and Jagged Arrays

- C, C++, Java, C#: jagged arrays

  `myArray[3][7]`
- Fortran, Ada, C#: rectangular array

  `myArray[3,7]`



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
- E.g. Python

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
mat = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]
vector[3:6], mat[1], mat[0][0:2]
```

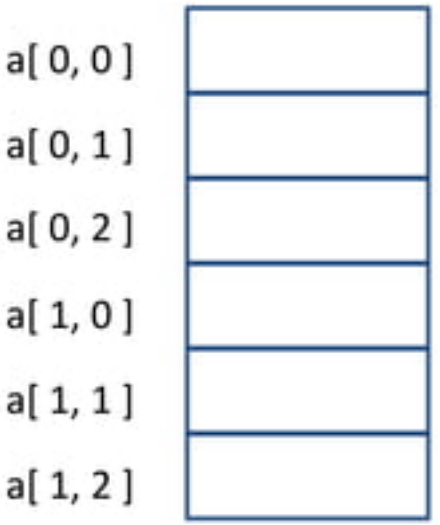
Implementation of Arrays

- Access function maps subscript expressions to an address in the array
- Single-dimensioned: list of adjacent memory cells
- Access function for single-dimensioned arrays:

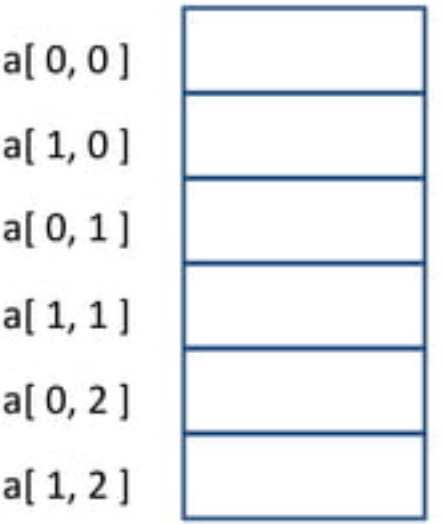
  address(list[k]) = address (list[lower_bound])

  + ((k-lower_bound) * element_size)

Accessing Multi-dimensioned Arrays



Row major order – used in most languages

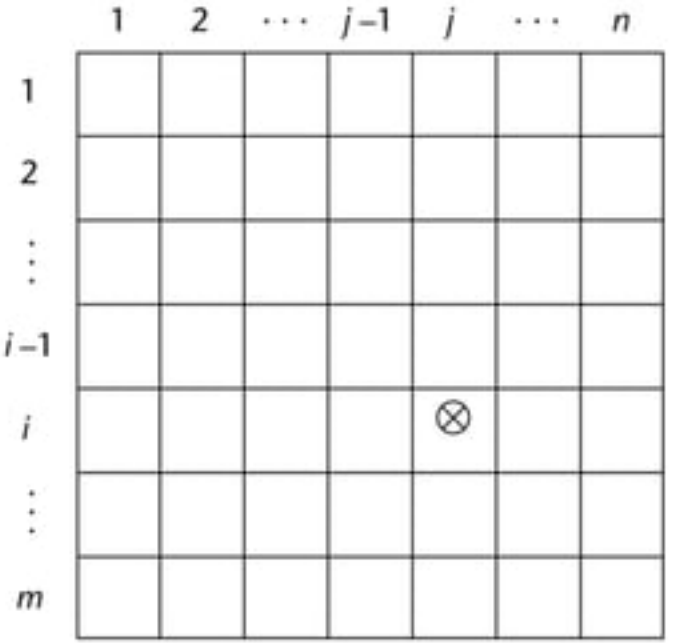


Column major order – used in Fortran

Accessing Multi-dimensioned Arrays

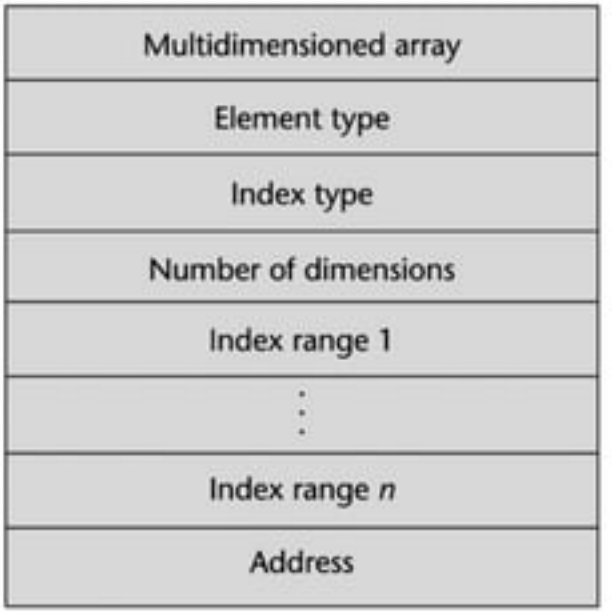
- General format

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



Compile-Time Descriptors

Single-dimensioned array



Multi-dimensional array

Associative Arrays

- An *associative array* is an unordered collection of data elements that are indexed by an equal number of values called *keys*
  - User defined keys must be stored
- Similar to `Map` in Scala
- Design issues: What is the form of references to elements

```
phonebook = {"John": "555-1234", "Mary": "555-6789", "Bob": "555-4321"}
```

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*:
  - heterogeneous aggregate of data elements
  - individual elements are identified by names
- Popular in most languages, OO languages use objects as records
- Design issues:
  - What is the syntactic form of references to the field?
  - Are elliptical references allowed

Definition of Records in Ada

- Record structures are indicated in an orthogonal way

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

References to Records

- Most language use dot notation
  Emp_Rec.Emp_Name.Mid
- Fully qualified references must include all record names
- Elliptical references allow leaving out record names as long as the reference is unambiguous, for example in COBOL
  `FIRST, FIRST OF EMP-NAME`, and `FIRST OF EMP-REC` are elliptical references to the employee's first name

Elliptical reference is also known as loop or cycle reference. For example,

```
struct Node{
   int data;
   Node* next;  //reference of its own type
```

Operations on Records

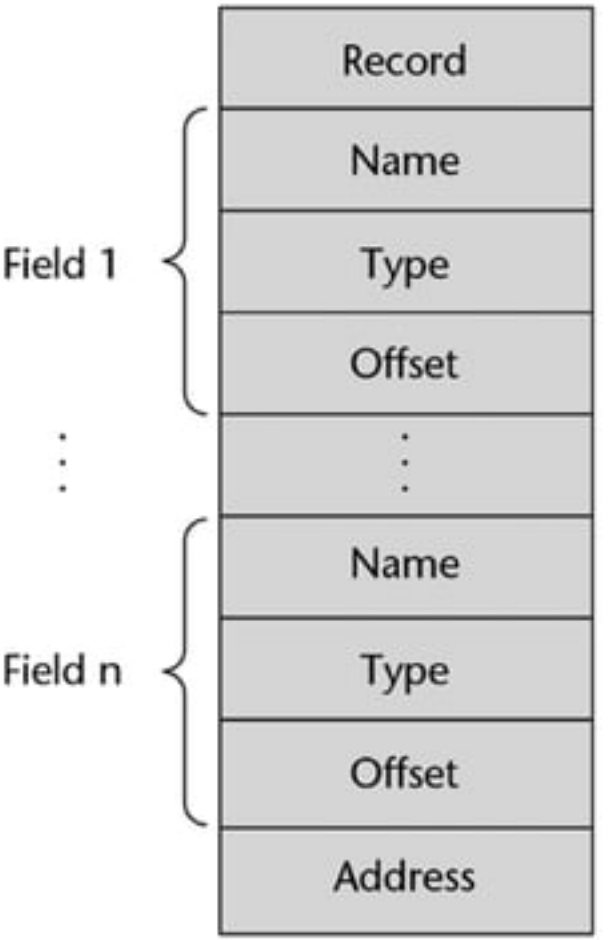
- Assignment is very common if the types are identical
- Ada allows record comparison
- Ada records can be initialized with aggregate literals
- COBOL provides `MOVE CORRESPONDING`
  - Copies fields which have the same name

Evaluation

- Straight forward and safe design
- Arrays are used when all data values have the same type and/or are processed in the same way
- Records are opposite
- Arrays: dynamic subscripting
- Records: static subscripting

Implementation of Record Type

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



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?

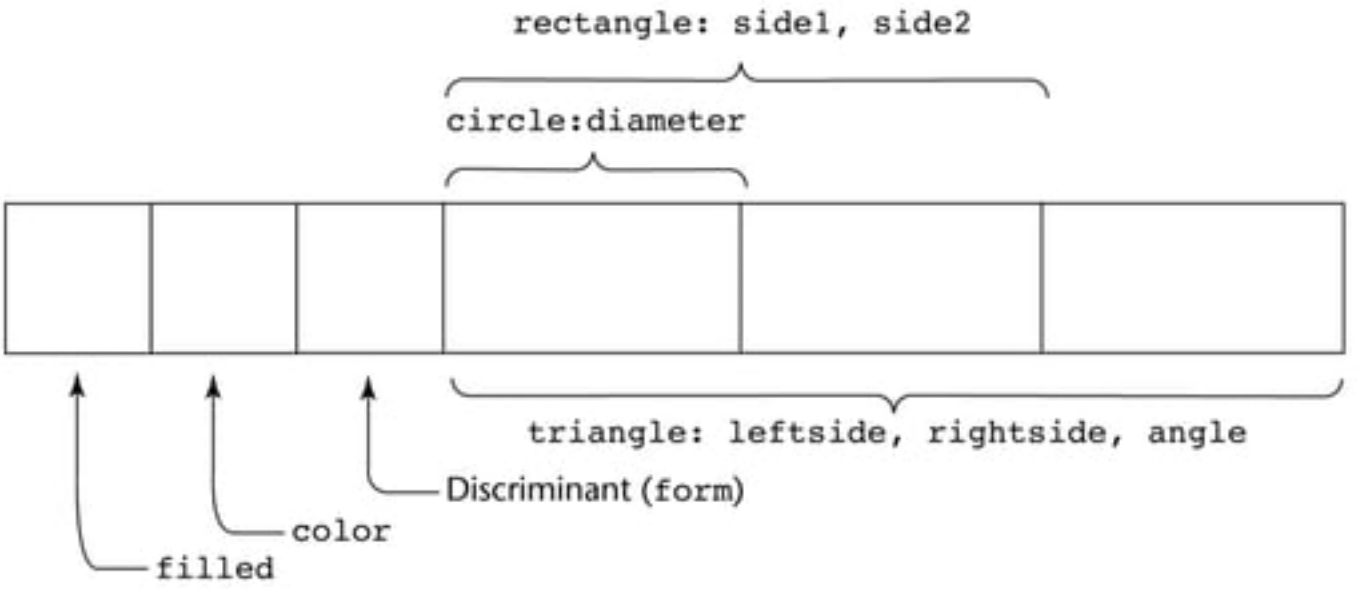
Discriminated vs. Free Unions

- Fortran, C, and C++ provide union constructs in which there is no language support for type checking; the 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

Evaluation of Unions

- Potentially unsafe construct
  - Do not allow type checking
- Java and C# do not support unions
  - Reflective of growing concerns for safety in programming language

Pointer and Reference 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 (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?

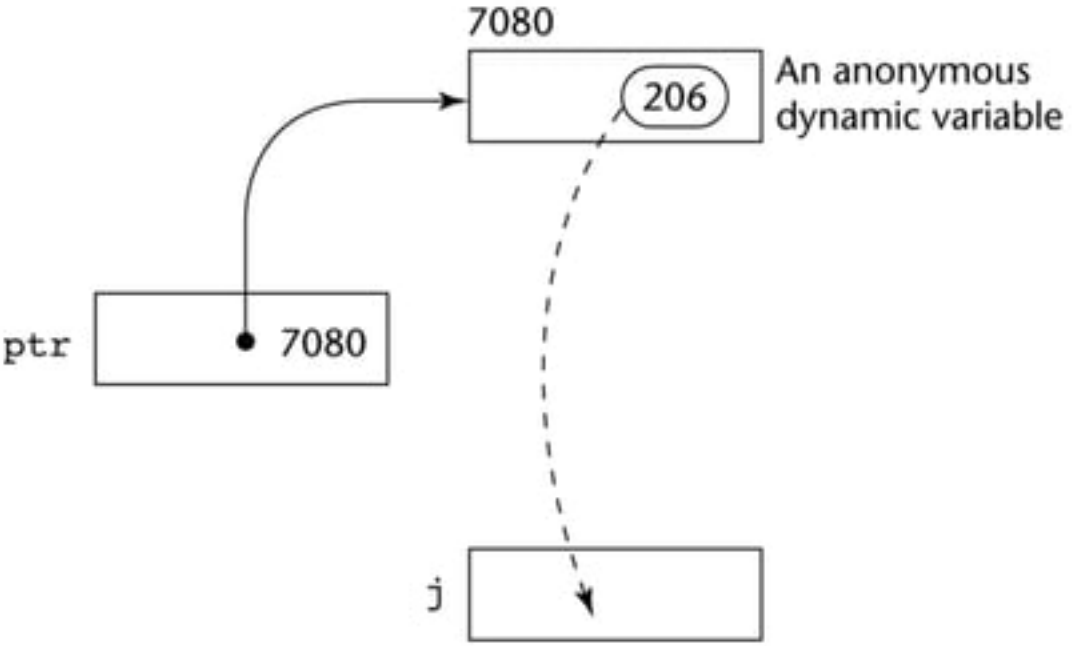
Pointer Operations

- Two fundamental operations: assignment and dereferencing
- Assignment is used to 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`

Pointer Operations

- Pointer points to a record in C/C++
  - Explicit: `(*p).name`
  - Implicit: `p -> name`
- Management of heap use explicit allocation
  - C: subprogram `malloc`
  - C++: `new` and `delete` operators

Problems with Pointers

- Dangling pointers (dangerous)
  - A pointer points to a heap-dynamic variable that has been de-allocated
- Lost heap-dynamic variable
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)

Pointers in Ada

- **access** types
- Some dangling pointers are disallowed because dynamic objects can be automatically de-allocated at the end of pointer's type scope
- The lost heap-dynamic variable problem is not eliminated by Ada

Pointers in C and C++

```
int *ptr;
int count, init;
...
ptr = &init;
count = *ptr;
```

- Extremely flexible but must be used with care
- Pointers can point at any variable regardless of when it was allocated
- Used for dynamic storage management and addressing

Pointers in C and C++

- Pointer arithmetic is possible

```
int list [10]; int *ptr; ptr = list;
*(ptr + 1)
*(ptr + index)
*ptr[index]
```

- 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)

Reference Types

- C++ includes a special kind of pointer type called a *reference type* that is used primarily for formal parameters
- Java extends C++'s reference variables and allows them to replace pointers entirely
  - References refer to call instances
- 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--they widen the range of cells that can be accessed by a variable
- Essential in some kinds of programming applications, e.g. device drivers
- Using references provide some of the flexibility and capabilities of pointers, without the hazards

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

Type Checking

- Generalize 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

Strong Typing

- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors
- Language examples:
  - FORTRAN 77 is not: `EQUIVALENCE`
  - C and C++ are not: unions are not type checked
  - Java, C#: strongly typed

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 the two variables have compatible 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 compatible with integer types
  - Formal parameters must be the same type as their corresponding actual parameters (Pascal)

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

Structure Type Equivalence

- 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)

Summary

- The data types of a language are a large part of what determines that language's style and usefulness
- The primitive data types of most imperative languages include numeric, character, and Boolean types
- The user-defined enumeration and subrange types are convenient and add to the readability and reliability of programs
- Arrays and records are included in most languages
- Pointers are used for addressing flexibility and to control dynamic storage management
- Strong typing means detecting all type errors