Theory of Programming Languages

Data Types

Sajid Anwer

Department of Computer Science,
FAST-NUCES, CFD Campus

Lecture Outline

- Data types fundamentals
- 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 Type
- Type Checking
- Strong Typing
- Type Equivalence
- Theory and Data Types

Introduction

- A *data type* defines a collection of data objects and a set of predefined operations on those objects

- A *descriptor* is the collection of the attributes of a variable

- An *object* represents an instance of a user-defined (abstract data) type

- One design issue for all data types: What operations are defined and how are they specified?

Primitive Data Types

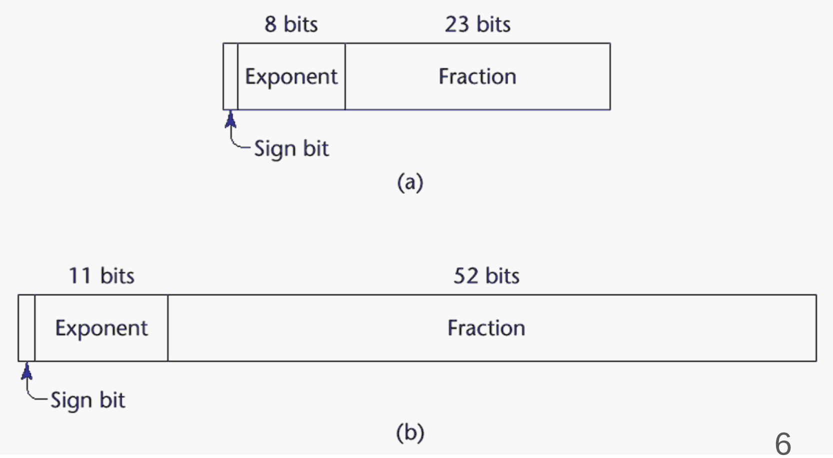
- Almost all programming languages provide a set of *primitive data types*
- Primitive data types: Those not defined in terms of *other data types*
- Some primitive data types are merely reflections of the hardware
- Others require only a little non-hardware support for their implementation

Primitive Data Types: Integer

- Almost always an exact reflection of the hardware so the mapping is trivial
- There may be as many as eight different integer types in a language
- Java's signed integer sizes: **`byte, short, int, long`**
- Most of computers used *tow's complement* to store negative value.

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`**; sometimes more
- Usually exactly like the hardware, but not always
- IEEE Floating-Point Standard 754



Primitive Data Types: Complex

- Some languages support a complex type, e.g., C99, Fortran, and Python

- Each value consists of two floats, the real part and the 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, starting with 2003

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

Character String Type in Certain Languages

- C and C++
  - » Not primitive
  - » Use **`char`** arrays and a library of functions that provide operations

- SNOBOL4 (a string manipulation language)
  - » Primitive
  - » Many operations, including elaborate pattern matching

- Fortran and Python
  - » Primitive type with assignment and several operations

- Java
  - » Primitive via the `String` class

- Perl, JavaScript, Ruby, and PHP

  - Provide built-in pattern matching, using regular expressions

Character String Length Options

- Static: COBOL, Java's `String` class
- *Limited Dynamic Length*: C and C++
  - » In these languages, a special character is used to indicate the end of a string's characters, rather than maintaining the length
- *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*--why not have them?
- Dynamic length is nice, but is it worth *the expense*?

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

- 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/deallocation is the biggest implementation problem

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};
```

  - » 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

National University
Of Computer and Emerging Sciences

Subrange Types

- An ordered contiguous subsequence of an ordinal type
  - » 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
- 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 Design Issues

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

- 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 initialized?
  - Are any kind of slices supported?

Array Indexing

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

  array_name (index_value_list) $\to$ 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 and Array Categories

- *Static*: subscript ranges are statically bound and storage *allocation is static* (before run-time)
  - » Advantage: efficiency (no dynamic allocation)

- *Fixed stack-dynamic*: subscript ranges are statically bound, but the allocation is done *at declaration elaboration time*
  - » Advantage: space efficiency

- *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)

Subscript Binding and Array Categories (continued)

- *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)

- *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)

- 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 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"};
```

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

Arrays Operations

- APL provides the 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's array assignments, but they are only reference changes. Python also supports array catenation and element membership operations

- Ruby also provides 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

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

- A jagged matrix has 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, *why?*

- 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`

National University
Of Computer and Emerging Sciences

Implementation of Arrays

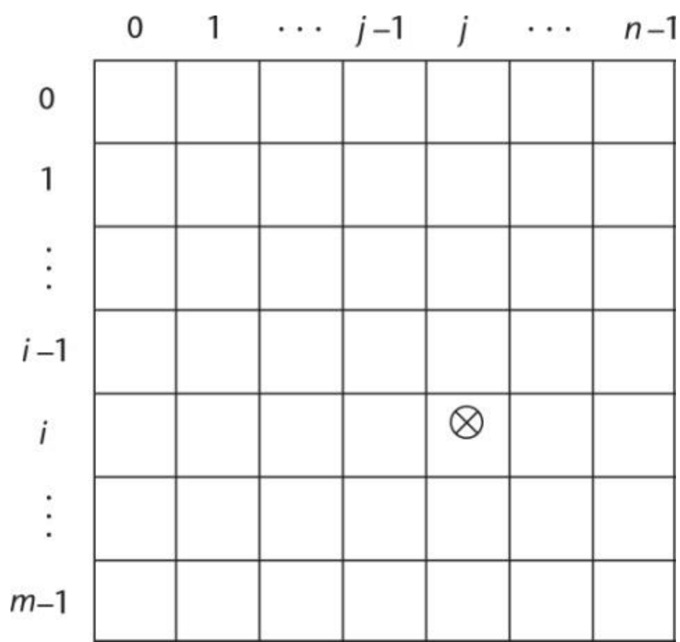
- Access function maps subscript expressions to an address in the array

- Access function for single-dimensioned arrays:

address(list[k]) = address (list[lower_bound])
+ ((k-lower_bound) * element_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
- General format
  - » Location (a[I,j]) = address of a [row_lb,col_lb] + (((I - row_lb) * n) + (j - col_lb)) * element_size



National University
Of Computer and Emerging Sciences

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

- 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

- 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

```
struct myStruct
{
        int var1;
        char var2[8];
        float var3;
}struct_var;
```

- Design issues:
  - » What is the syntactic form of references to the field?

Definition of Records in COBOL

- COBOL uses level numbers to show nested records; others use recursive definition

```
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.
```

Definition of Records in Ada

- Record structures are 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

  1. COBOL

  field_name `OF` record_name_1 `OF` ... `OF` record_name_n

  2. Others (dot notation)

  record_name_1.record_name_2. ... record_name_n.field_name

- Fully qualified references must include all record names

Operations on Records

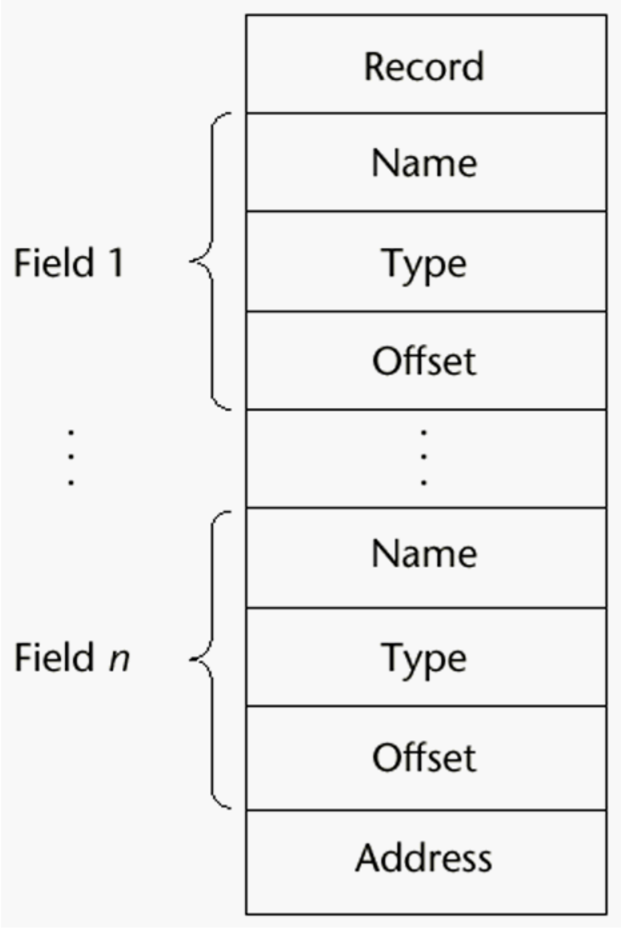
- Assignment is very common if the types are identical
- Ada allows record comparison
- Ada records can be initialized with aggregate literals
- COBOL provides `MOVE CORRESPONDING`
  - » Copies a field of the source record to the corresponding field in the target record

Evaluation and Comparison to Arrays

- Records are used when collection of data values is *heterogeneous*
- Access to array elements is much slower than access to record fields, *why?*
  - » 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`**

List Types

- Lists in LISP and Scheme are 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 literally what it is

  As code, `(A B C)` is the function `A` applied to the parameters `B` and `C`

- The interpreter needs to know which a list is, so if it is data, we quote it with an *apostrophe* `'(A B C)` is data

National University
Of Computer and Emerging Sciences

List Types (continued)

- List Operations in ML
  - » Lists are written in brackets and the elements are 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]`

List Types (continued)

- F# Lists
  - » Like those of ML, except elements are separated by semicolons and `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]`

- Both C# and Java supports lists through their 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*

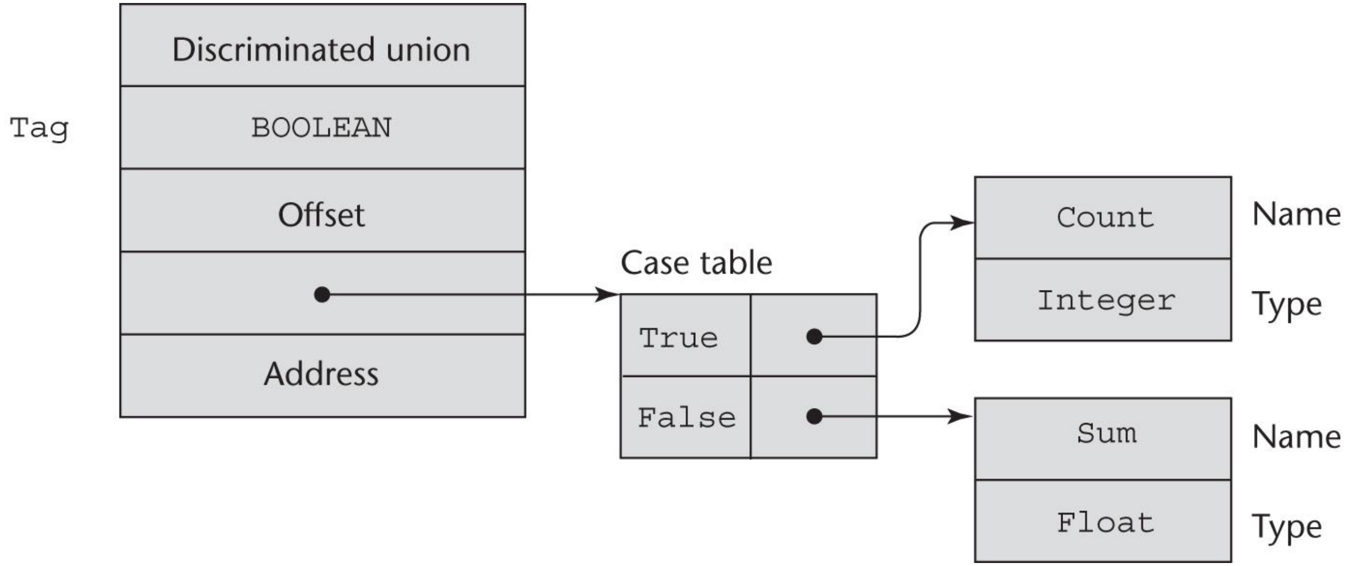
- Design issues
  - » Should type checking be required?
  - » Should unions be embedded in records?

- 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

Implementation of Unions

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



National University
Of Computer and Emerging Sciences

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 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

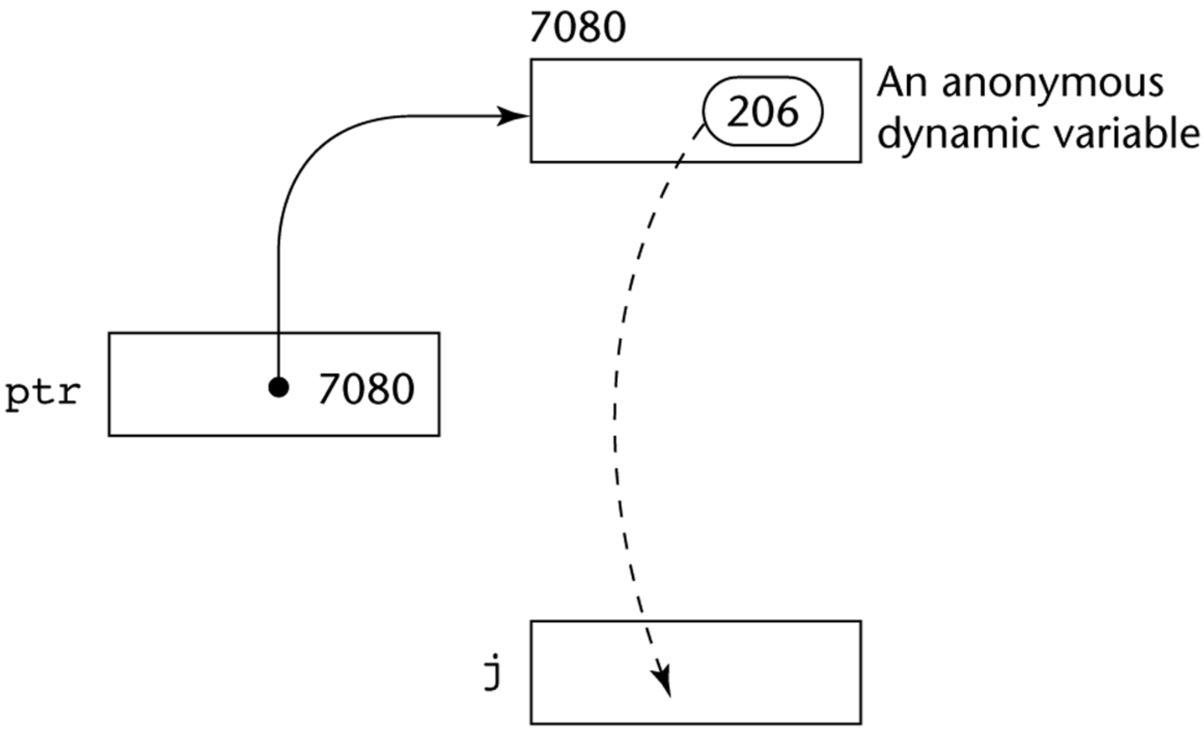
  » Dereferencing can be explicit or implicit

  » C++ uses an explicit operation via *

  `j = *ptr`

  sets j to the value located at `ptr`

Pointer Assignment Illustrated



The assignment operation j = *ptr

Problems with Pointers

- Dangling pointers (dangerous)
  - » A pointer points to a heap-dynamic variable that has been *deallocated*

```
int * arrayPtr1;
int * arrayPtr2 = new int[100];
arrayPtr1 = arrayPtr2;
delete [] arrayPtr2;
// Now, arrayPtr1 is dangling, because the heap storage
// to which it was pointing has been deallocated.
```

- Some dangling pointers are disallowed because dynamic objects can be automatically deallocated at the end of pointer's type scope

Problems with Pointers

- 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*

National University
Of Computer and Emerging Sciences

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

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]`

National University Of Computer and Emerging Sciences

Reference Types

- C++ includes a special kind of pointer type called a *reference type* that is used primarily for formal parameters
  - » Support pass-by-reference
- 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--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

Implementation: 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

```
int * arrayPtr1;
int * arrayPtr2 = new int[100];
arrayPtr1 = arrayPtr2;
delete [] arrayPtr2;
// Now, arrayPtr1 is dangling, because the heap storage
// to which it was pointing has been deallocated.
```

Implementation: Dangling Pointer Problem

- *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

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

- 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*
- 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)
- *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

Type Equivalence

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

National University
Of Computer and Emerging Sciences

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*
    - different units of speed, both float)