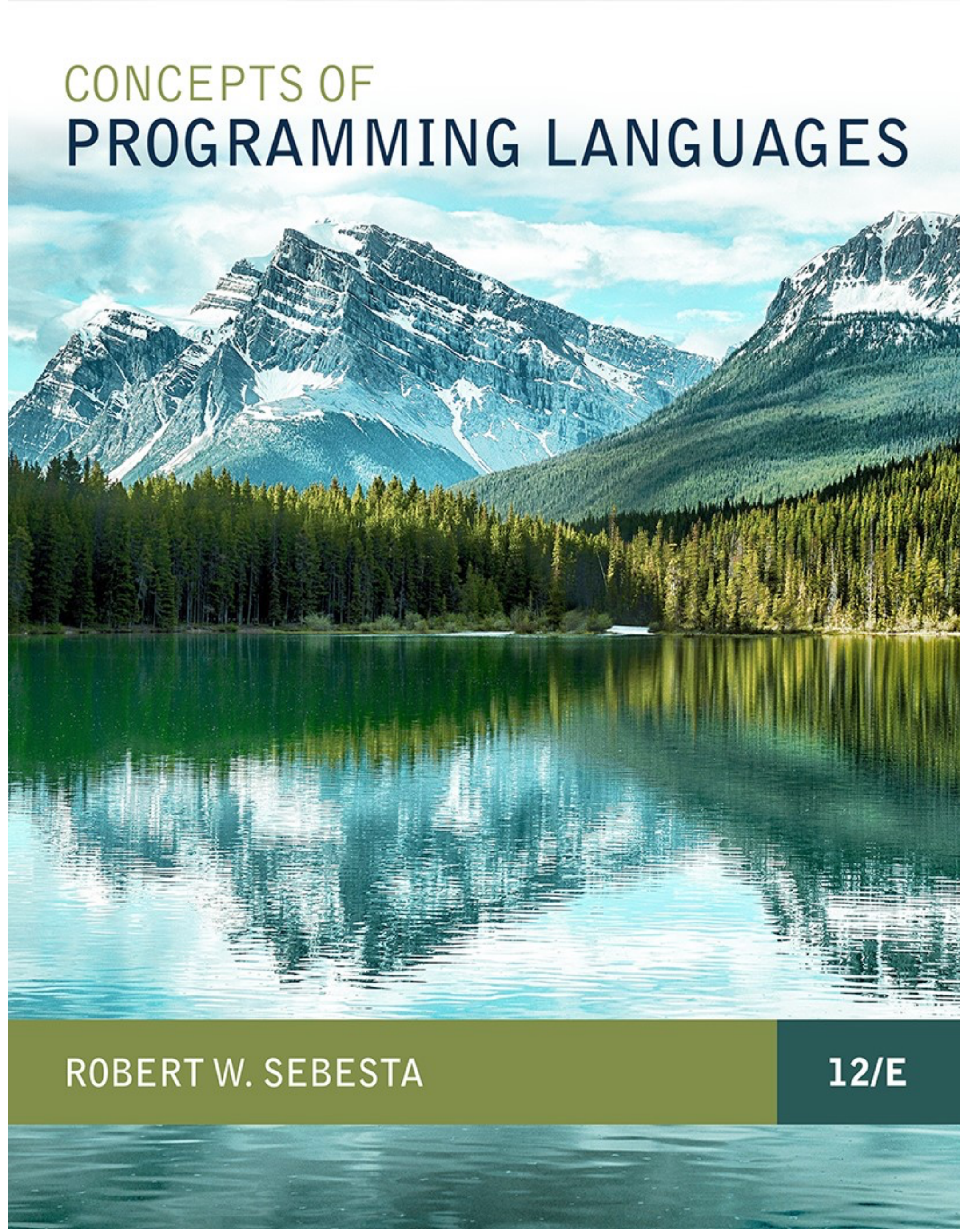
Chapter 6

Data Types



ISBN 0-321—49362-1

Chapter 6 Topics

- Introduction
- Primitive Data Types
- Character String Types
- Enumeration Types
- Array Types
- Associative Arrays
- Record Types
- Tuple Types
- List Types
- Union Types
- Pointer and Reference Types
- Optional 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:*
    - is the collection of the attributes of a variable (e.g., file descriptor)
  - *Object:*
    - represents an instance of a user-defined (abstract data) type
- One design issue for all data types:
  - What **operations** are defined
  - how are they **specified**?

Primitive Data Types

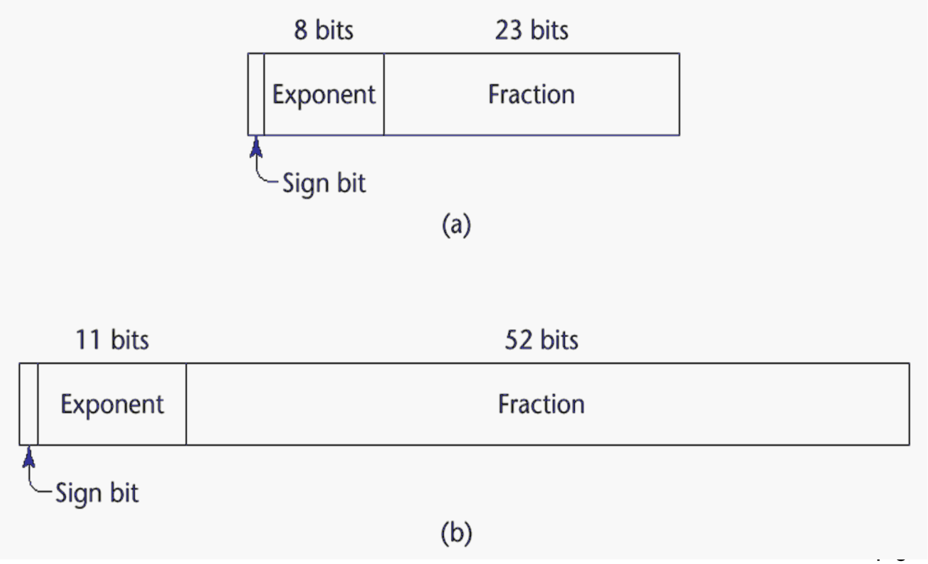
- Almost all programming languages provide a set of *primitive data types*
- Primitive data types:
  - those basic or built-in data types

Primitive Data Types: Integer

- There may be as many as eight different integer types in a language
- E.g., Java's signed integer: **`byte`**, **`short`**, **`int`**, **`long`**

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`
- 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
  - the imaginary part
- Literal form (in Python):

  `(7 + 3j)`, where **7** is the real part and **3** is the imaginary part [imaginary number: $i^2 = -1$]

Primitive Data Types: Decimal

- For business applications
  - Essential to COBOL
  - C# offers a decimal data type
  - java.math.BigDecimal in Java
- Store a fixed number of decimal digits, in coded form--Binary-Coded Decimal (BCD)
  - Ref: https://en.wikipedia.org/wiki/Binary-coded_decimal
  - E.g., 1001 = 9
- *Advantage*: accuracy
- *Disadvantages*: limited range, wastes memory

Primitive Data Types: Boolean

- Simplest of all
- Range of values: two elements:
  - "true" and "false"
- Could be implemented as bits, but often as bytes
  - Advantage: readability

Primitive Data Types: Character

- Stored as numeric encodings
- Most commonly used encoding: ASCII
- Alternatives
  - 16-bit Unicode-2 byte Universal Character Set (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
  - Universal Coded Character Set + Transformation Format—8-bit (UTF-8)
    - Dominant character encoding for the World Wide Web

Character String Type

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

Character String Type: Operations

- 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
- Fortran and Python
  - Primitive type with assignment and several operations
- Java
  - Primitive via the java.lang.**`String`** class
- Perl, JavaScript, Ruby, and PHP
  - Provide built-in pattern matching, using regular expressions

Character String Length Options

- Static: 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—null character '\0', rather than maintaining the length
- *Dynamic* (no maximum): Perl, JavaScript

Character String Type Evaluation

- Aid to writability
- Static length:
  - 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/ deallocation is the 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

Note: length of Java String = Integer.MAX_VALUE = 2,147,483,647 (2^{31} - 1)

User-Defined Ordinal Types

- An ordinal type:
  - range of possible **values** that can be easily associated with the set of positive integers
- E.g., in Java
  - **`integer`**
  - **`char`**
  - **`boolean`**

Enumeration Types

- All possible values--named constants, are provided in the definition
- e.g., in C#:

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

- Design topics
  - 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?
  - Can enumeration values be converted to integer?
  - Any other type converted to an enumeration type?

Enumeration Type Evaluation

- Aid to readability, e.g., no need to code a color as a number
- Aid to reliability, e.g., compiler can check:
  - **operations** (e.g., don't allow colors to be added)
  - No enumeration variable can be assigned a value **outside** its **defined range**
  - 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

Ref: https://stackoverflow.com/questions/3990319/storing-integer-values-as-constants-in-enum-manner-in-java

Array Types

- Array:
  - a homogeneous aggregate of data elements
  - each individual element is **identified** by its position in the aggregate, relative to the first element.

Array Design Topics

- Subscripts / indices
  - What types are valid for **subscripts/indices**?
  - Are subscripting expressions references **range checked**?
  - When are subscript ranges bound?
  - What is the **maximum** number of subscripts?
- Allocation/initialization
  - When does **allocation** take place?
  - Can array objects be **initialized**?
  - Are any kind of **slices** supported?
- Array Type
  - Are ragged or rectangular multidimensional arrays allowed, or both?

Array Indexing

- *Indexing* (or subscripting)
  - a **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, Java:
  - **integer only**
- Index range checking
  - C, C++, Perl, and Fortran do **not** specify range checking
  - Java, ML, C# specify range checking
- **Non-integer** index
  - E.g., PhP:

```
$array = array(
        "foo" => "bar",
        "bar" => "foo",
    );
```

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 time
  - advantage: space efficiency

Subscript Binding and Array Categories

- *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)

Subscript Binding and Array Categories

- Static:
  - C and C++ arrays that include `static` modifier
  - e.g., **static** myClass anAry[100];
- Fixed stack-dynamic:
  - C and C++ arrays without `static` modifier
  - e.g., myClass anAry[100];
- Fixed heap-dynamic:
  - C, C++ arrays
  - e.g., myClass* heapAry = new myClass[100];
- Heap-dynamic:
  - Perl, JavaScript, Python, and Ruby arrays
  - e.g., Perl: @schools=("GMU","GWU");

Array Initialization

- Some language allow **initialization** at the time of **storage allocation**
  - C and C++:

```
char myString[] = "freddie"; // char string
char *myStrings[] = {"Bob", "Jake", "Peter" ];
                            //array of string
```

  - Java:

```
int[] myList = {4, 5, 7, 83};
String[] myStrings={"Bob", "Jake", "Peter" };
```

Array Initialization

- **Swift**

  e.g., array of **String** and **Int** elements

```
let myClasses = [“csci6011”, “csci6221”]
let myGrades = [ 100, 99, 98, 97, 95]
```

- **Python**

  e.g.,

```
import array as arr
ary = arr.array(‘I’,[x ** 2 for x in range(12) if
x % 3 == 0])
```

  ➔ assign `[0, 9, 36, 81]` to **ary**

Python Array vs. List

```
import array as arr
myArray = arr.array('I', [x ** 2 for x in range(12) if x % 3 == 0])
   ## 'I' :  unsigned inta  ;  array: same data type
myArray2 = arr.array('I', [x ** 2 for x in range(12) if x % 4 == 0])
   ## 'I' :  unsigned inta  ;  array: same data type
print type(myArray)
print (myArray[3])
finalArray= myArray + myArray2
print finalArray[6]

myList = [1, 2, 3, 4, 5, 6.0, 'I']  # list, mutable
print type(myList)
print (myList[5])
print (myList[6])

myTuple = ('T', [1, 2, 3, 4, 5], 'abc')   # tuple, immutable
print type(myTuple)
print (myTuple[1])
```

Output:

```
<type 'array.array'>
81
64
<type 'list'>
6.0
I
<type 'tuple'>
[1, 2, 3, 4, 5]
```

Heterogeneous Arrays

- *Heterogeneous array*
  - the elements need **not** be of the **same** type
- Supported by Perl, JavaScript, and Ruby
- E.g., Javascript:

```
var garde = [100, 3.0, "A"];
```

Arrays Operations

- Python:
  - array catenation and element membership operations
  - e.g., a=[1,2], b=[3, 4]

    c= a + b = [1, 2, 3, 4]
- Ruby
  - provides array catenation
  - e.g., a=[“a”, “b”] b=[“c”, “d”]

    c = a + b

Rectangular and Jagged Arrays

- Rectangular array: multi-dimensional array
  - all **rows, columns** have same number of elements
- Jagged array:
  - row with different number of elements--columns
  - arrays of arrays
- C, C++, and Java support jagged arrays
- Java

```
int  csci6221[][] = new int[2][];   // jagged array
csci6221[0] = new int[20];  // first row
csci6221[1] = new int[10];  // second row
```

Slice

- Slice
  - is sub-structure of an array
- Python:

  `vector = [2, 4, 6, 8, 10, 12, 14, 16]`

  `mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`

  `vector[3:6]=[8, 10, 12]` is a 3-element array

  `mat[0][0:2]=[1,2]` is the *first* and *second* element of the first row of `mat (i.e., mat[0])`
- Ruby: supports slices with the **`slice`** method

  `ary.slice(2, 2)` returns the third and fourth elements of `ary` `[ ary.slice(`**`start, length`**`) ]`

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 topics:
  - 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" =>
  83, …);
```

- Subscripting is done using braces and keys

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

  - Elements can be removed with **delete**

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

Record Types

- *Record*
  - a heterogeneous aggregate of data elements in which the individual elements are identified by **names**
- Design topics:
  - What is the syntactic form of references to the field?
  - Are **elliptical** references allowed ?
    - elliptical references : example on next 2 slides

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.
```

```
PIC: Picture clause
X:   alphanumeric
V:   implicit decimal
```

Ref: https://www.tutorialspoint.com/cobol/cobol_data_types.htm

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

- Elliptical references allow leaving out record names as long as the reference is **unambiguous**,
  - E.g., in COBOL example on previous page:

  `FIRST`, `FIRST OF EMP-NAME`, and `FIRST` of EMP-REC are elliptical references to the employee's first name

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 (i.e., cannot change it after its creation)
    - Create with a tuple literal

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

    - Referenced with subscripts (begin at 0)

Tuple Types

- Python
  - myTuple = ('l', [1, 2, 3, 4, 5]);

- F#
  - **`let`** `tup = (3, 5, 7)`
  - **`let`** `a, b, c = tup` This assigns a tuple to a tuple pattern `(a, b, c)`

List Types

- Lisp and Scheme:
  - Lists are delimited by parentheses

    `(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

List Types: in Scheme

- List Operations
  - **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/**concatenates** its first parameter into its second 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: in ML

- List Operations
  - 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]`
  - The Scheme `CAR` and `CDR` functions are named `hd` and `tl` in ML, respectively.

List Types

- F# Lists
  - Like those of ML, except elements are 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/changeable
  - Elements can be of any type
  - Create a list with an assignment

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

List Types

- Python Lists (continued)
  - List elements are referenced with subscript/indices beginning at **zero**

    ```
    myList = [3, 5.8, "grape"]
    x = myList[1]      # Set x to 5.8
    ```

  - List elements can be deleted with `del`

    ```
    del myList[1]
    ```

  - List Comprehensions–derived from set notation
    - `range(7)` creates `[0, 1, 2, 3, 4, 5, 6]`
    - `[x * x for x in range(7) if x % 3 == 0]` construct a list: `[0, 9, 36]`

List Types

- Haskell's List

```
[n * n | n <- [1..10]]
```

- F#'s List

```
let myArray = [ for i in 1 .. 5 -> (i * i) ]
printf "%A" myArray
```

- 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 topic
  - Should type checking be required?

Discriminant vs. Free Unions

- Free Union:
  - no type checking
  - Supported by C and C++
- Discriminant Union*:*
  - Type checking
  - Supported by ML, Haskell, and F#

Unions in F#

- Defined with a type statement using “|”

```
type intReal =                    // int or float
    | IntValue of int
    | RealValue of float;;
```

`intReal` is the new type

`IntValue` and `RealValue` are constructors

To create a value of type `intReal`:

```
let ir1 = IntValue 17;;
let ir2 = RealValue 3.4;;
```

Unions in F#

- Accessing the value of a union is done with pattern matching

```
match pattern with
    | expression_list1 -> expression1
    | ...
    | expression_listn -> expressionn
```

  - Pattern can be **any** data type
  - The expression list can have wildcards _

Unions in F#

Examples:

```
let a = 7;;
let b = "grape";;
let x = match (a, b) with
     | 4, "apple" -> apple
     | _, "grape" -> grape
     | _ -> fruit;;
// apple, grape, fruit: variable
```

```
let filter123 x =
    match x with
    | 1 | 2 | 3 -> printfn "Found 1, 2, or 3"
    | a -> printfn "%d" a
```

Unions in F#

To display the type of the `intReal` union:

```
let printType value =
  match value with
      | IntVale value -> printfn "int"
      | RealValue value -> printfn "float";;
```

If `ir1` and `ir2` are defined on p.49,

`printType ir1` returns `int`

`printType ir2` returns `float`

Unions Evaluation

- 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

Pointer and Reference Types

- *Pointer* :
  - 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
  - use to access a location in the area where storage is dynamically created (usually called a *heap*)

Design Topics of Pointers

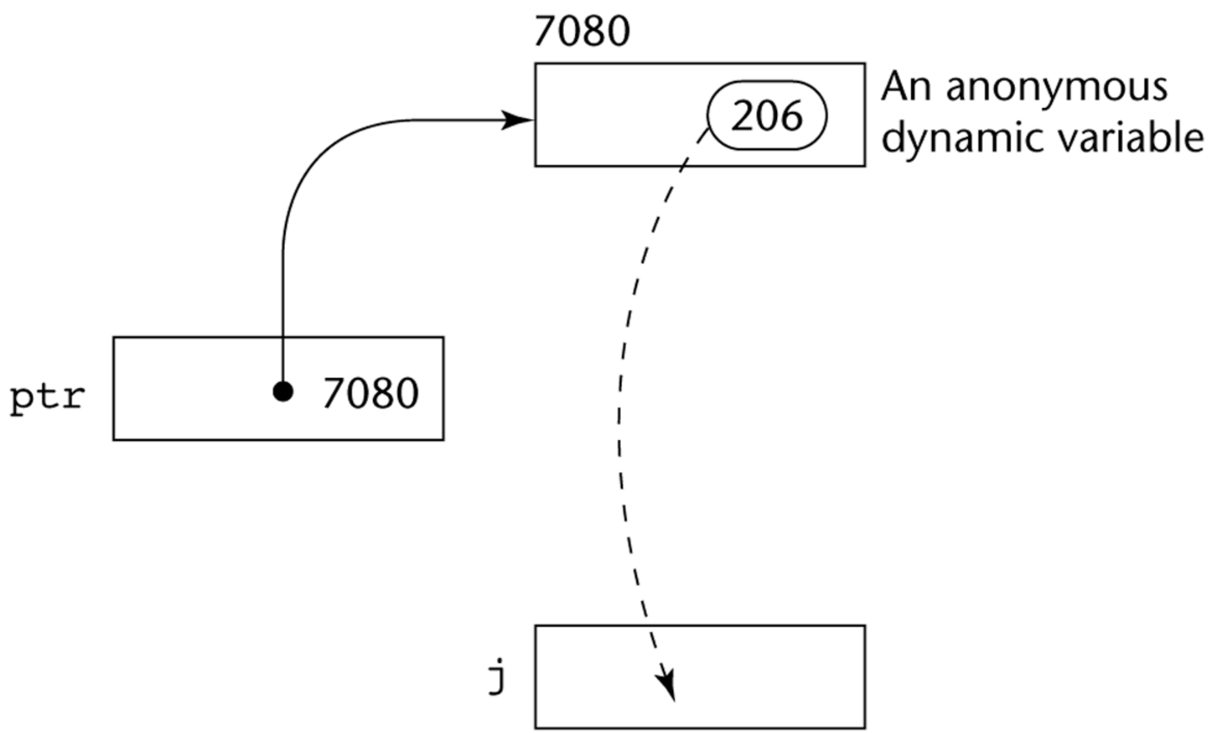
- What are the scope of and lifetime of a pointer variable?
- What is the lifetime of a heap-dynamic variable?
- Are pointers restricted as to the type of value to which they can point?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer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

Pointer Operations

- Two fundamental operations:
  - assignment and dereferencing
- Assignment
  - used to set a pointer to some address
- Dereferencing
  - yields the value stored at the address--pointer's value
  - can be explicit or implicit
  - C++ uses an explicit operation via *

    `j = *ptr` // set j = the value located at `ptr`

Pointer Assignment Illustrated



The assignment operation j = *ptr = 206

Problems with Pointers

- Dangling pointers
  - pointer points to a heap-dynamic variable that has been deallocated
  - E.g., int *p1 = new int;   int *p2=p1;   delete p1;

    p2: dangling pointer
- Lost heap-dynamic variable (memory Leakage)
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)
    - E.g., int *p1 = new int;   //obj-1
    -         p1= new int;   //obj-2
    - The process of losing heap-dynamic variable (e.g., obj-1) is called *memory leakage*

Pointers in C and C++

- Pointer extremely flexible, can point at any variable
- Used for dynamic storage management, addressing
- Explicit **dereferencing** and **address-of** operators
- **type** does not need to be fixed (`void *`)
  - `void *` can point to any type and can be type-checked (but cannot be de-referenced)
  - E.g., in C: int a = 8; void* ptr= &a; int* b = ptr;
- Pointer arithmetic is possible

```
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

- Reference:
  - a special kind of pointer type in C++
  - is used primarily for formal parameters
- Java extends C++'s reference variables and allows them to replace pointers entirely
- C# includes both the references of Java and the pointers of C++

Pointers Evaluation

- Dangling pointers and dangling objects are problems
- Pointers can be accessed by a variable
- Pointers or references are necessary for dynamic data structures--so we can't design a language **without** them

Problems with Pointers

- Dangling pointers
  - pointer points to a heap-dynamic variable that has been deallocated
  - E.g., int *p1 = new int;   int *p2=p1;   delete p1;

    p2: dangling pointer
- Lost heap-dynamic variable (memory Leakage)
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)
    - E.g., int *p1 = new int;   //obj-1
    - p1= new int;   //obj-2
    - The process of losing heap-dynamic variable (e.g., obj-1) is called *memory leakage*

Dangling Pointer Problem

- *Tombstone*: extra heap cell--pointer to a heap-dynamic variable
  - The actual pointer variable points only at tombstones, when heap-dynamic variable (object) **de-allocated**, tombstone is set to nil.
  - [ptr var → tombstone (nil) → heap-dynamic var]

. *Locks-and-keys*:pointer value is represented as (key, address) **pair**
  - Heap-dynamic variables are represented as variable plus a cell for integer **lock** value [ptr var+key (xxx) → heap-dynamic var+lock)]
  - When heap-dynamic variable **allocated**, a **lock** value is created/placed in ***lock*** *cell* of the variable and in ***key*** *cell* of pointer.
  - When a heap-dynamic variable is **deallocated**, the key of its pointer is modified to hold a value different from the variable's cell--lock.
  - Any attempt to dereference the pointer can be flagged as an error.

Ref : *Tombstones*: https://en.wikipedia.org/wiki/Tombstone_(programming)
Locks-and-keys : https://en.wikipedia.org/wiki/Locks-and-keys

Heap Management

- A very complex run-time process
- 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

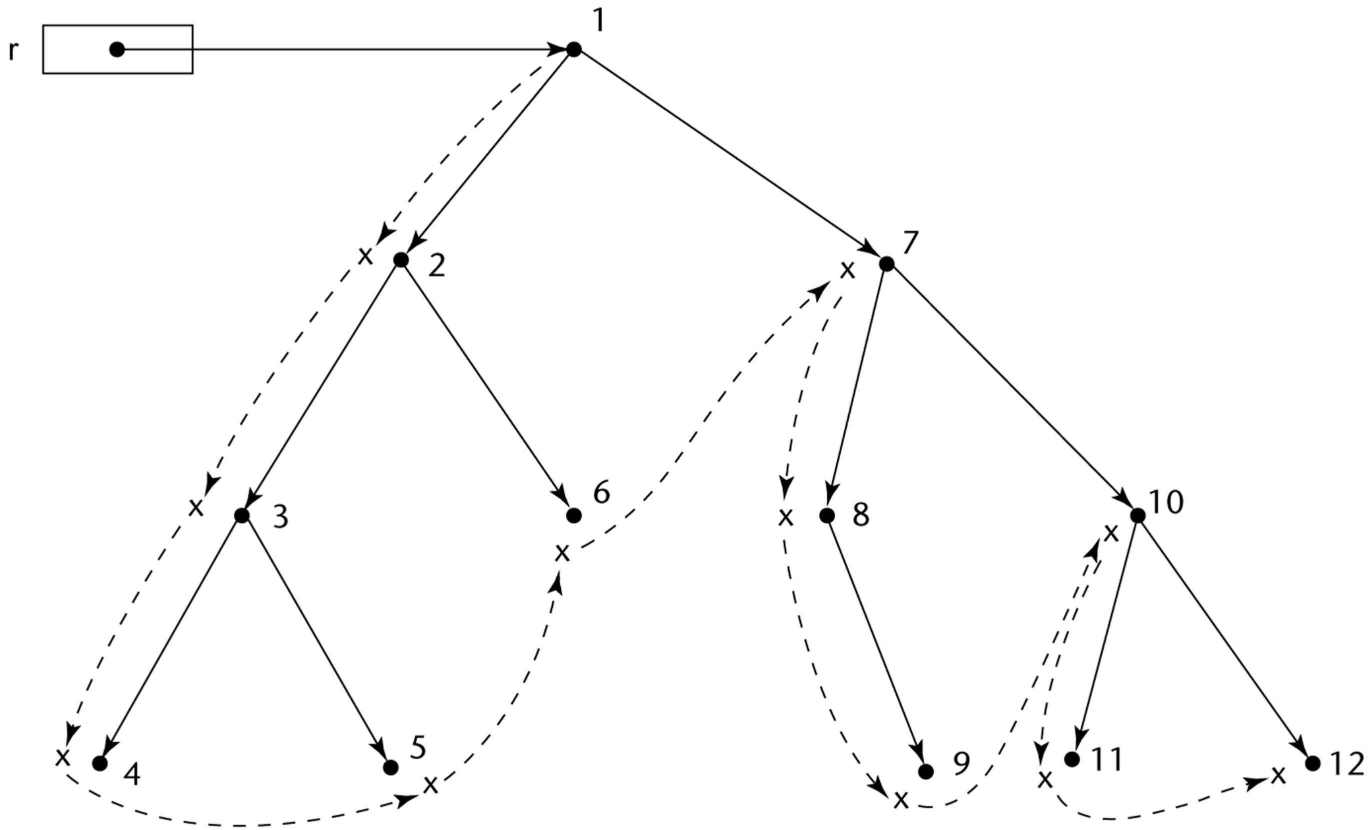
- Reference counters: maintain a **counter** in every cell that store the number of pointers **currently pointing** at the memory cell (object)
- If an object's reference count reaches **zero**, the object has become inaccessible, and can be destroyed.
  - *Disadvantages*: space required, execution time required, complications for cells connected circularly
  - *Advantage*: it is intrinsically incremental, so significant delays in the application execution are avoided

- Ref: https://en.wikipedia.org/wiki/Reference_counting

Mark-Sweep

- mark-sweep
  - 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

Optional Types

- Optional types : a variable that could have no value

  - **Swift (optional type)**

```
var  hw1: Int? = 100       // set 100 to optional var hw1
hw1  = nil                 // reset hw1 to nil--no value
                           // cannot use nil for non-optional var
```

  - **C# (nullable type)**

```
int?  hw1  = 100;             // set 100 to nullable var hw1
Nullable<int> hw1  = 100; // System.Nullable<T> : struct
hw1  = null;                  // reset hw1 to null--no value
                              // can use null for all variables

hw1.HasValue                  // return true if has a value; otherwise false
```

Type Checking

- *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

- 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
- Java and C# are, almost (because of explicit type casting)
- ML and F# are

Strong Typing

- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus ML and F#)

- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of **Ada**