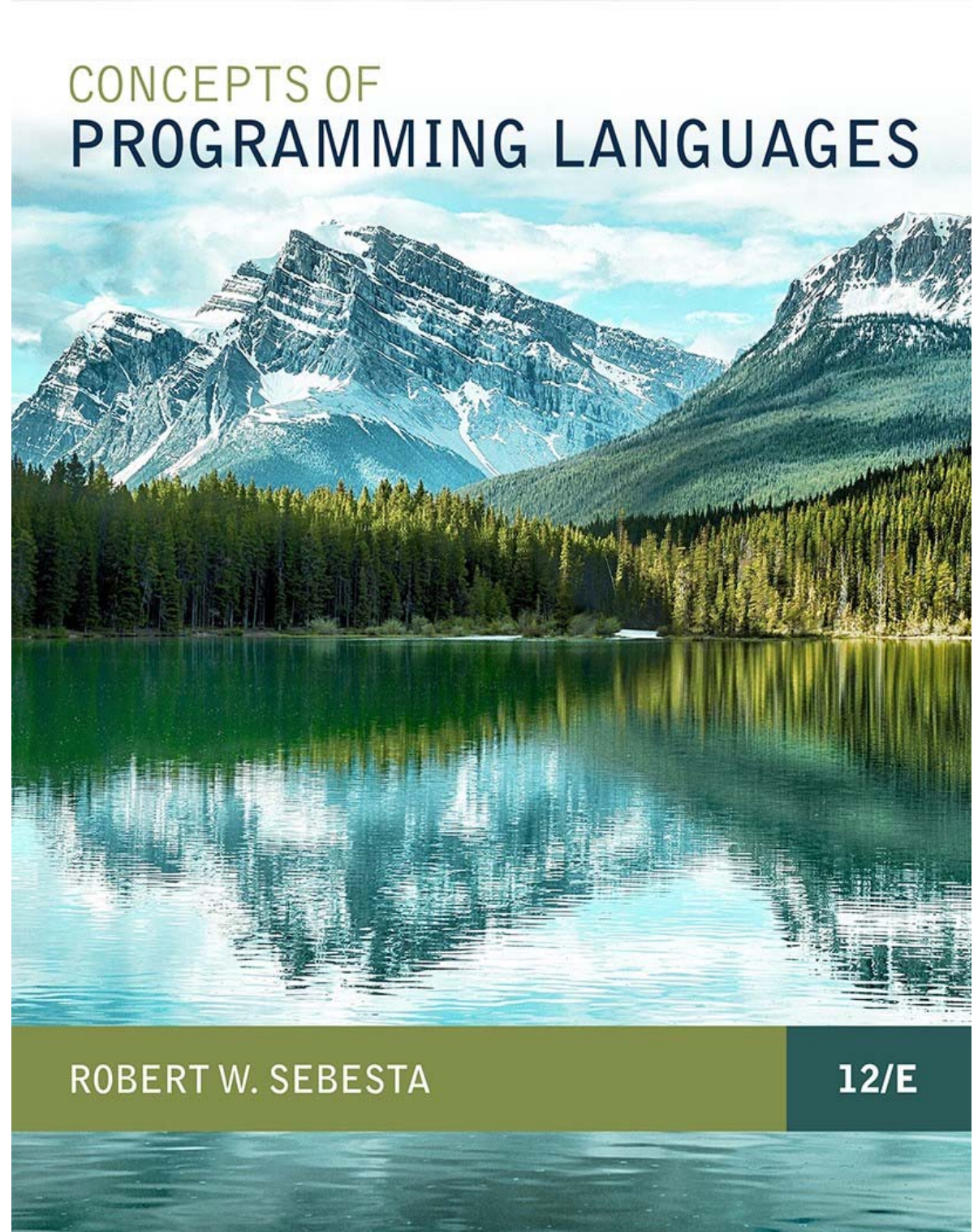


# Chapter 6

## 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

---

- 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

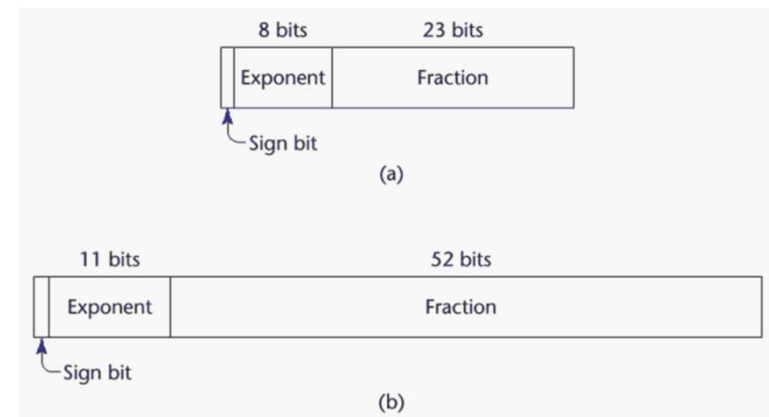
---

- 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**

# 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
  - Now supported by many languages
- 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?

# Character String Types 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
- SNOBOL4 (a string manipulation language)
  - Primitive
  - Many operations, including elaborate pattern matching
- Fortran and Python
  - Primitive type with assignment and several operations
- Java (and C#, Ruby, and Swift)
  - 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

# 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/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

# 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
  - C#, F#, Swift, and Java 5.0 provide better support for enumeration than C++ because enumeration type variables in these languages are not coerced into integer types

# Array Types

---

- 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.

# 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 initialized?
- Are any kind of slices supported?



# Array Indexing

---

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

array\_name (index\_value\_list) → 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
- Java: integer types only
- Index range checking
  - C, C++, Perl, and Fortran do not specify range checking
  - Java, ML, C# specify range checking

# 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 (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)

# Subscript Binding and Array Categories (continued)

---

- 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 (continued)

---

- 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, Swift, and C#

- 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"};`

- Python

- List comprehensions

```
list = [x ** 2 for x in range(12) if x % 3 == 0]
puts [0, 9, 36, 81] in list
```

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

# Rectangular and Jagged Arrays

---

- A rectangular array is a multi-dimensional array in which all of the rows have the same number of elements and all columns have the same number of elements
- A jagged matrix has rows with varying number of elements
  - Possible when multi-dimensional arrays actually appear as arrays of arrays
- C, C++, and Java support jagged arrays
- F# and C# support rectangular arrays and 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

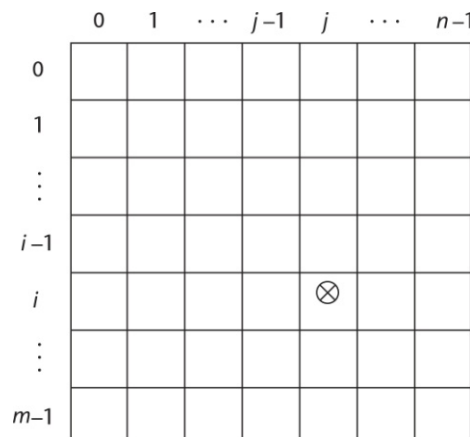
`list.slice(2, 2)` returns the third and fourth elements of `list`

# Implementation of Arrays

---

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

$$\text{address}(\text{list}[k]) = \text{address}(\text{list}[\text{lower\_bound}]) + ((k - \text{lower\_bound}) * \text{element\_size})$$





# Accessing Multi-dimensional 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 an Element in a Multi-dimensional Array

---

- General format

Location ( $a[l,j]$ ) = address of  $a[\text{row\_lb}, \text{col\_lb}] + (((l - \text{row\_lb}) * n) + (j - \text{col\_lb})) * \text{element\_size}$

	1	2	...	$j-1$	$j$	...	$n$
1							
2							
⋮							
$i-1$							
$i$					⊗		
⋮							
$m$							

# 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 Swift

# 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 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.
```

# 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, for example in COBOL

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



# Evaluation and Comparison to Arrays

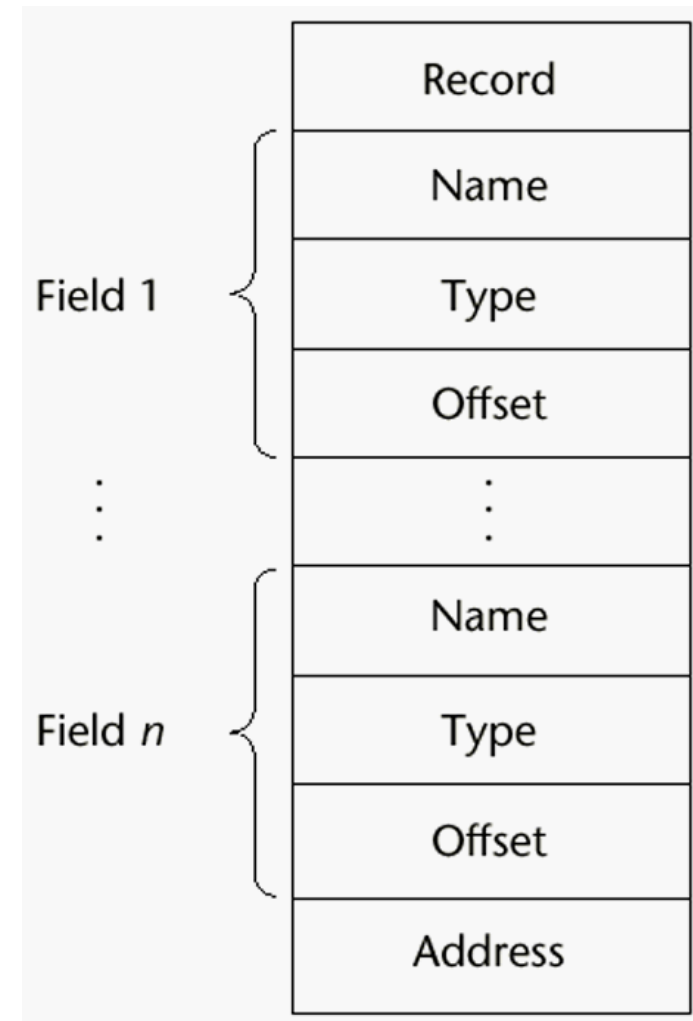
---

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

- (The asterisk is just a separator)

- F#

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

```
let a, b, c = tup
```

- This assigns a tuple to a tuple pattern (a, b, c)

# 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

# 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 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 (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]`
  - The Scheme `CAR` and `CDR` functions are named `hd` and `tl`, respectively

# 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]`

# 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) |]
```

- 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 issue
  - Should type checking be required?

# Discriminated vs. Free Unions

---

- 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 ML, Haskell, and F#

# Unions in F#

---

- Defined with a type statement using OR

```
type intReal =  
    | 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# (continued)

---

- 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 wild cards (`_`)

# Unions in F# (continued)

---

## Example:

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

# Unions in F# (continued)

---

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 as previously,

```
printType ir1 returns int  
printType ir2 returns float
```



# 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

# 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?

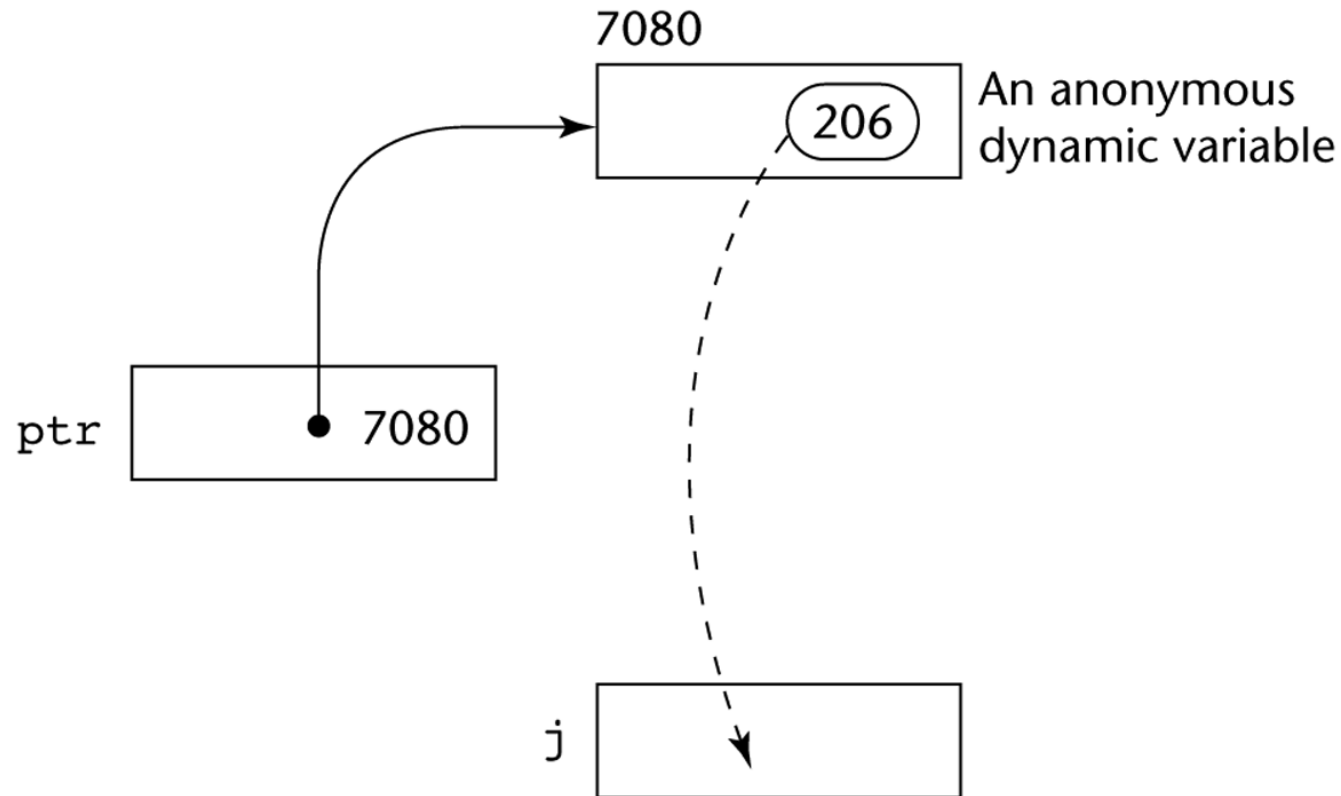
# Pointer Operations

---

- 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
- 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 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;
```

$\ast(p+5)$  is equivalent to `stuff[5]` and `p[5]`  
 $\ast(p+i)$  is equivalent to `stuff[i]` and `p[i]`



# Reference Types

---

- C++ includes a special kind of pointer type called a *reference type* that is used primarily for formal parameters
  - Advantages of both pass-by-reference and pass-by-value
- 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

# 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

---

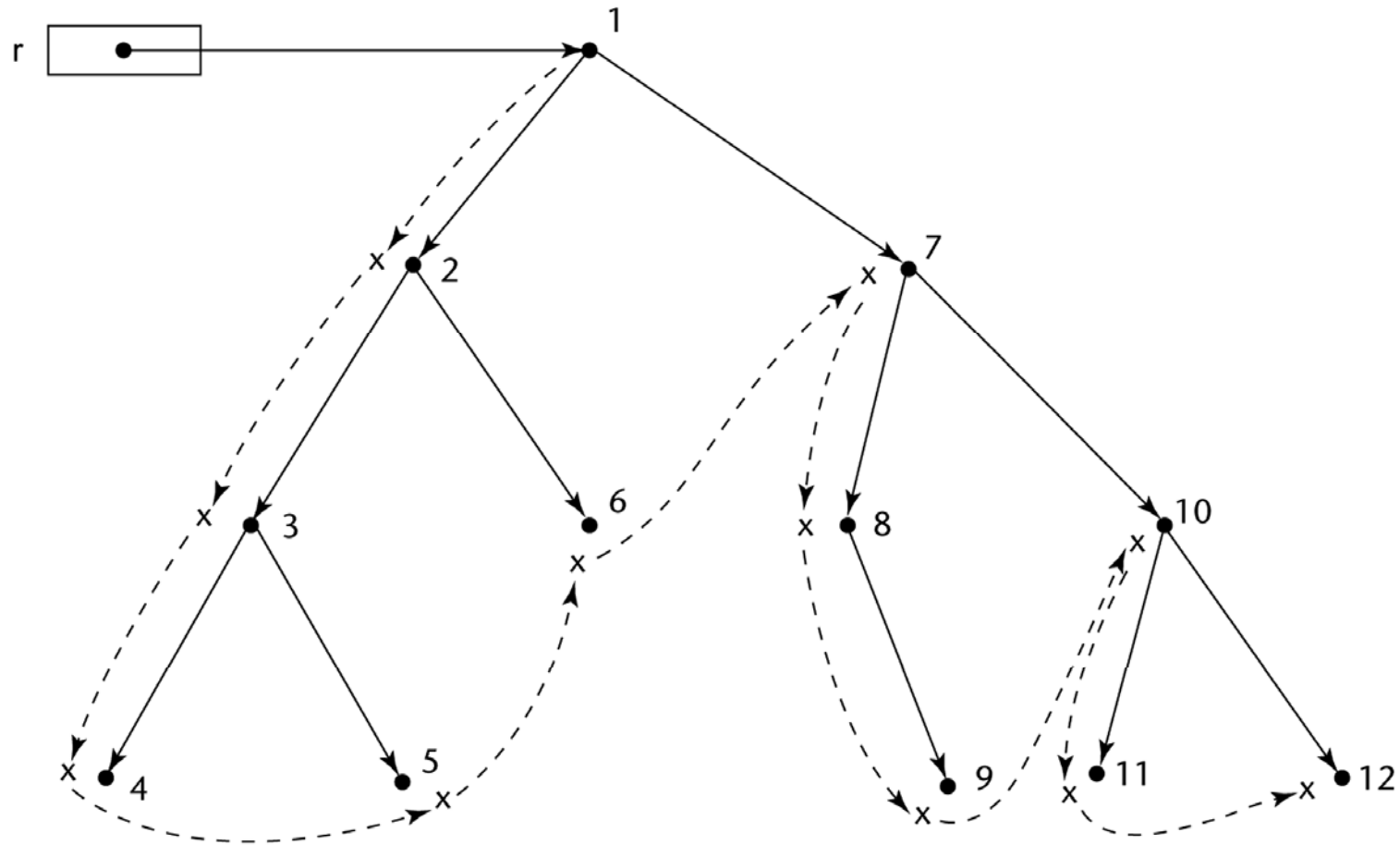
- Reference counters: maintain a counter in every cell that store the number of pointers currently pointing at the cell
  - *Disadvantages*: space required, execution time required, complications for cells connected circularly
  - *Advantage*: it is intrinsically incremental, so significant delays in 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 is nontrivial
  - Maintaining the list of available space is another source of overhead

# Optional Types

---

- Optional types are useful when there is a need for a variable to indicate that it currently has no value
- C#, F#, and Swift, among others, have optional types
- Reference types in C# are already optional types (use `null` for no value)
- Value types in C# (struct types) can be declared to be optional by attaching a question mark to the type name in their declaration

```
int? x;
```

- The no-value is `null`, which can be assigned to `x` and `x` can be tested for it
- In Swift, `nil` is used instead of `null`

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

# Strong Typing (continued)

---

- 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

# Name 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

---

- *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 is a 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

---

- 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