

# 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
- 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
  - In an implementation, a descriptor is an area of memory that stores the attributes of a variable.
  - If the attributes are all static, descriptors are required only at compile time.
  - These descriptors are built by the compiler, usually as a part of the symbol table, and are used during compilation

# Introduction

---

- For dynamic attributes, part or all of the descriptor must be maintained during execution. In this case, the descriptor is used by the run-time system.
  - In all cases, descriptors are used for type checking and building the code for the allocation and deallocation operations.
- 
- 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?

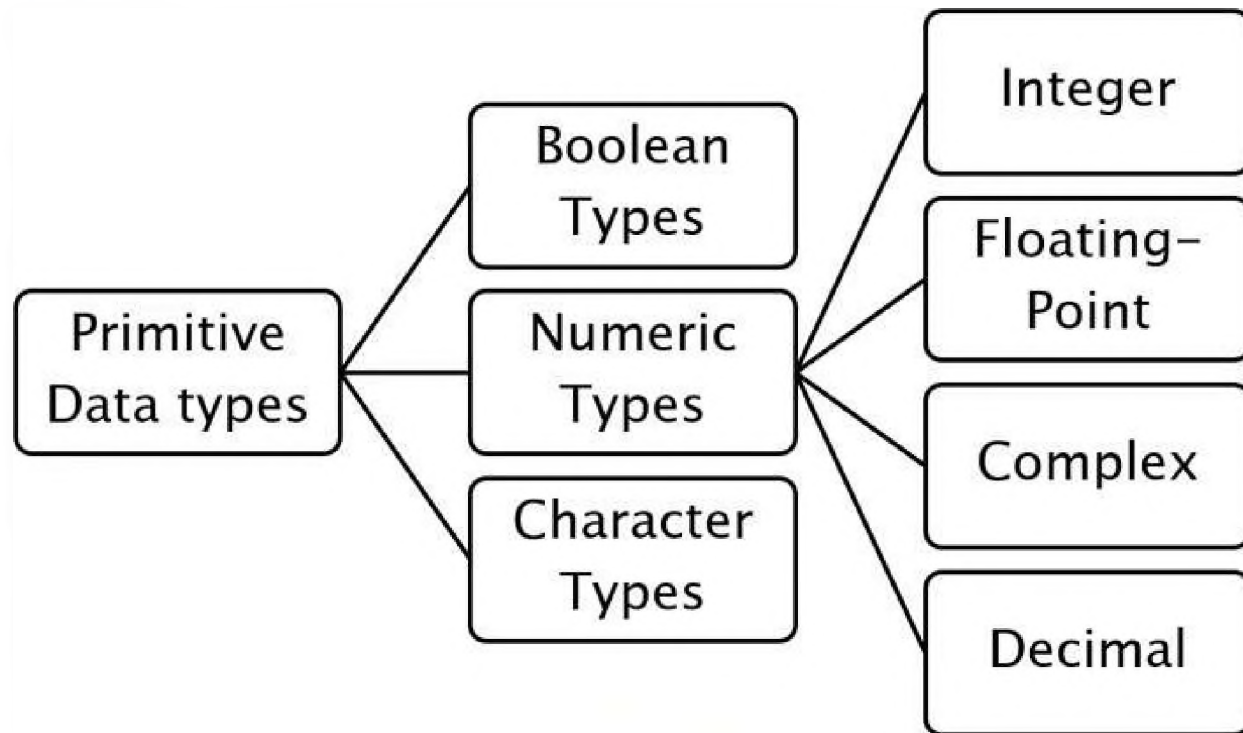
# Primitive Data Types

---

- 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

---



# Primitive Data Types: Integer

---

- Almost always an exact reflection of the hardware so the mapping is trivial
- The hardware of many computers supports several sizes of integers
- There may be as many as eight different integer types in a language
  - Java's signed integer sizes: `byte`, `short`, `int`, `long`
  - C++ and C#, include unsigned integer types, which are types for integer values without signs.



# Primitive Data Types: Floating Point

---

- **Floating Point** data types model real numbers, but the representations are only approximations for many real values.
- Floating-point problems:
  - Representation of infinite number (i.e.,  $\pi$ ) in finite amount of computer memory. On most computers, floating-point numbers are stored in binary, which exacerbates the problem.
    - E.g.,  $(0.1)_{10} = (0000.0001100110011001101)_2$
  - Floating-point types is the loss of accuracy through arithmetic operations.

# Primitive Data Types: Floating Point

---

- Floating-point values are represented as fractions and exponents, a form that is borrowed from scientific notation.
  - $346.7893 = 3.467893E2$
- Languages for scientific use support at least two floating-point types (e.g., `float` and `double`; sometimes more)

# Primitive Data Types: Floating Point

---

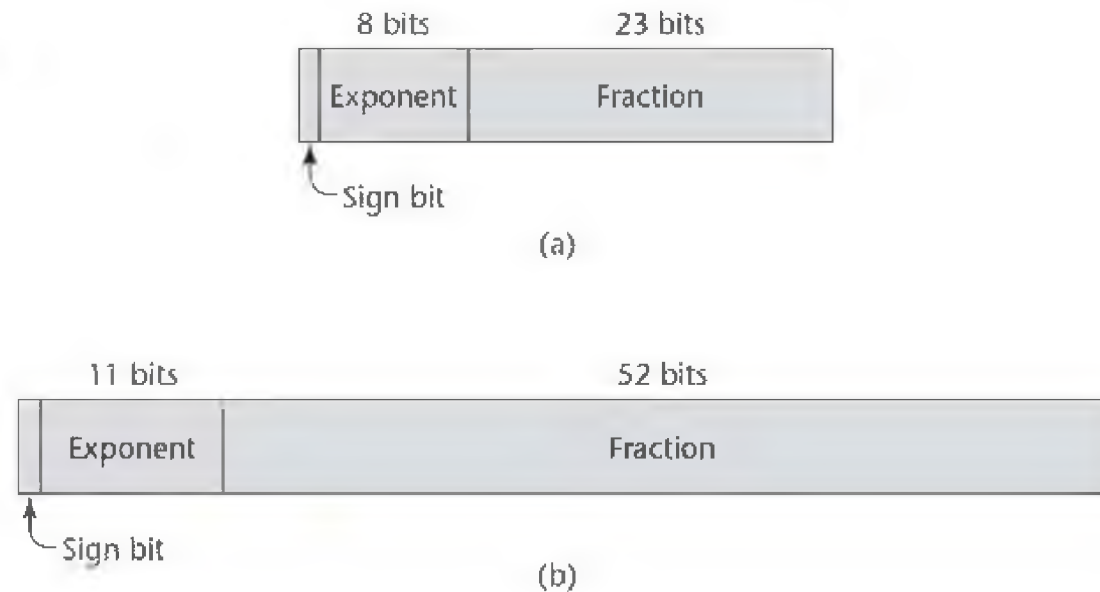
- float type is the standard size, usually stored in four bytes of memory.
- Double-precision variables usually occupy twice as much storage as float variables and provide at least twice the number of bits of fraction.
- Usually exactly like the hardware, but not always

# Primitive Data Types: Floating Point

---

**Figure 6.1**

IEEE floating-point formats: (a) single precision, (b) double precision



# 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 Binary Coded Decimal (BCD)
- *Advantage:* accuracy
  - the number 0.1 (in decimal) can be exactly represented in a decimal type, but not in a floating-point type
- *Disadvantages:* limited range, wastes memory
  - because no exponents are allowed

# Primitive Data Types: Boolean

---

- Simplest of all
- Range of values: two elements, one for “true” and one for “false”
- Boolean types are often used to represent switches or flags in programs. Although other types, such as integers, can be used for these purposes, the use of Boolean types is more readable
- Could be implemented as bits, but often as bytes

# Primitive Data Types: Boolean

---

- C99 and C++ have a Boolean type, they also allow numeric expressions to be used as if they were Boolean. In such expressions, all operands with nonzero values are considered true, and zero is considered false.
  - Example: `If(x=3)`
- –This is not in Java and C#



# Primitive Data Types: Character

---

- Character data are stored in computers as numeric codings.
- Most commonly used coding: American Standard Code for Information Interchange (ASCII) which includes 128 characters
- An alternative, 16-bit coding: Unicode (UCS-2)
  - Includes characters from most natural languages
  - Originally used in Java
  - C#, JavaScript, Python, Perl, and F# also support Unicode
- 32-bit Unicode (UCS-4)
  - Supported by Fortran, starting with 2003

# Character String Types

---

- A character string type is one in which the values consist of 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
    - A substring reference is a reference to a substring of a given string.
  - Pattern matching

# Character String Types Operations

---

- For example, consider the following declaration:  
`char str[] = "apples";`
- `str` is an array of `char` elements, specifically `apples0`, where `0` is the null character.
- `strcpy`, which moves strings;
- `strcat`, which catenates one given string onto another
- `strcmp`, which lexicographically compares (by the order of their character codes) two given strings.
- `strlen`, which returns the number of characters, not counting the null, in the given string

# Character String Types Operations

---

- The string manipulation functions of the C standard library, which are also available in C++, are inherently unsafe and have led to numerous programming errors.
- The problem is that the functions in this library that move string data do not guard against overflowing the destination. For example, consider the following call to `strcpy`:  

```
strcpy(str1, str2);
```
- If the length of `str1` is 20 and the length of `str2` is 50, `strcpy` will write over the 30 bytes that follow `str1`. The point is that `strcpy` does not know the length of `str1`, so it cannot ensure that the memory following it will not be overwritten.
- The same problem can occur with several of the other functions in the C string library

# 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

---

1. **Static length strings:** COBOL, Python, Java's `String` class, Ruby, and the .NET class library available to C# and F#
  - The length **can** be static and set when the string is created.
2. **Limited Dynamic Length strings:** 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

# Character String Length Options

---

## 3. Dynamic (no maximum) length strings: SNOBOL4, Perl, JavaScript, and the standard C++ library

- allow strings to have varying length with no maximum.
- This kind requires the overhead of dynamic storage allocation and deallocation but provides maximum flexibility.



# Character String Type Evaluation

---

- String types are important to the writability of a language.
- Dealing with strings as arrays can be more cumbersome than dealing with a primitive string type.
  - For example, consider a language that treats strings as arrays of characters and does not have a predefined function that does what `strcpy` in C does.
  - Then, a simple assignment of one string to another would require a loop.

# Character String Type Evaluation

---

- Strings as a primitive type to a language is not costly in terms of either language or compiler complexity.
- 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
  - the name of the type.
  - the type's length (in characters).
  - the address of the first character.

**Figure 6.2**

Compile-time descriptor  
for static strings

Static string
Length
Address

# Character String Implementation

---

- **Limited dynamic length**: require a run-time descriptor to store both the fixed maximum length and the current length.
  - C and C++ do not require run-time descriptors, because the end of a string is marked with the null character.

**Figure 6.3**

Run-time descriptor for limited dynamic strings

Limited dynamic string
Maximum length
Current length
Address

# Character String Implementation

---

- **Dynamic length:** require a simpler run-time descriptor because only the current length needs to be stored.
- Dynamic length strings require more complex storage management than others. The length of a string, and therefore the storage to which it is bound, must grow and shrink dynamically.

# Character String Implementation

---

Three approaches to supporting the dynamic allocation and deallocation that is required for dynamic length strings:

## 1. Linked List.

- The drawbacks to this method are the extra storage occupied by the links in the list representation and the necessary complexity of string operations.

## 2. Arrays of pointers

- This method still uses extra memory, but string processing can be faster than with the linked-list

# Character String Implementation

---

## 3. Adjacent storage cells

- The problem with this method arises when a string grows: How can storage that is adjacent to the existing cells continue to be allocated for the string variable?
- This problem solved by moving the string to another storage place.
- If no adjacent storage cells enough to store the sting?

# Enumeration Types

---

- All possible values, which are named constants, are provided in the definition
- C# example
  - The enumeration constants are typically implicitly assigned the integer values, 0, 1, . . .

```
enum days {mon=5, tue, wed, thu, fri, sat, sun};
```



# Enumeration Types

---

- 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?
- If an enumeration variable is coerced to a numeric type, then there is little control over its range of legal operations or its range of values.
- If an `int` type value is coerced to an enumeration type, then an enumeration type variable could be assigned any integer value, whether it represented an enumeration constant or not.

# Enumeration Types

---

- In C++, we could have the following:

```
enum colors {red, blue, green, yellow, black};
```

```
colors myColor = blue, yourColor = red;
```

- The expression `myColor++`, would assign green to `myColor`.
- The expression `myColor = 4`, is illegal in C++.
- C# enumeration types are like those of C++, except that they are never coerced to integer.

# Enumeration Types

---

- In ML, enumeration types are defined as new types with **datatype** declarations.

```
datatype weekdays = Monday | Tuesday  
                  | Wednesday | Thursday | Friday
```

- The type of the elements of `weekdays` is integer.
- F# has enumeration types that are similar to those of ML, except the reserved word **type** is used instead of **datatype** and the first value is preceded by an **OR** operator (`|`).

# Enumeration Types

---

- Interestingly, none of the relatively recent scripting languages include enumeration types. These include Perl, JavaScript, PHP, Python, Ruby, and Lua.
- Even Java was a decade old before enumeration types were added.

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

# Evaluation of Enumerated Type

---

```
enum colors {red =1, blue =1000, green =100000}
```

- In this example, a value assigned to a variable of `colors` type will only be checked to determine whether it is in the range of `1...100000`.


# 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.
- The individual data elements of an array are of the same type.
- References to individual array elements are specified using subscript expressions

# 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*. (This reduced the readability)  
`Sum := Sum + B(I);`
  - Most other languages use brackets

# Arrays Index (Subscript) Types

---

- Two distinct types are involved in an array type: the **element type** and the **type of the subscripts**. The type of the subscripts is often integer.
  - E.g. `double temp[10];` (**elements type is double**)
  - `Temp[3] = 70.8;` (**3 is index**)
- FORTRAN, C: integer only
- Java: integer types only (byte, short, int, and long)



# Arrays Index (Subscript) Types

---

- Index range checking
  - C, C++, Perl, and Fortran do not specify range checking
  - Java, ML, C# specify range checking
  - For example in Perl, for the array `@list`, the second element is referenced with `$list[1]`.
  - One can reference an array element in Perl with a negative subscript, in which case the subscript value is an offset from the end of the array. `$list[-2]`.

# Subscript Binding and Array Categories

---

- The binding of the subscript type to an array variable is usually static, but the subscript value ranges are sometimes dynamically bound.
- In some languages, the lower bound of the subscript range is implicit. For example, in the C-based languages, the lower bound of all subscript ranges is fixed at 0.
- In some other languages, the lower bounds of the subscript ranges must be specified by the programmer.

# Subscript Binding and Array Categories

---

1. **Static** subscript ranges are statically bound and storage allocation is static (before run-time)
  - Advantage: efficiency (i.e. No dynamic allocation or deallocation is required)
  - The disadvantage is that the storage for the array is fixed for the entire execution time of the program.

# Subscript Binding and Array Categories

---

2. **Fixed stack–dynamic**: subscript ranges are statically bound, but the allocation is done at execution time

- Advantage: space efficiency
  - large array in one subprogram can use the same space as a large array in a different subprogram, as long as both subprograms are not active at the same time.
- The disadvantage is the required allocation and deallocation time

# Subscript Binding and Array Categories

---

3. **Fixed heap-dynamic**: is similar to a fixed stack-dynamic array, in that the subscript ranges and the storage binding are both fixed after storage is allocated.

- The differences are that both the subscript ranges and storage bindings are done when the user program requests them during execution,
- The storage is allocated from the heap, rather than the stack.
- Advantage: flexibility
- Disadvantages: allocation time from the heap, which is longer than allocation time from the stack.

# Subscript Binding and Array Categories

---

4. **Heap-dynamic**: binding of subscript ranges and storage allocation is dynamic and can change any number of times during the array's lifetime.

- Advantage: flexibility (arrays can grow or shrink during program execution as the need for space changes)
- Disadvantages: is that allocation and deallocation take longer and may happen many times during execution of the program.



# Subscript Binding and Array Categories

---

- 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 (By using `new` and `delete` )
- In Java, arrays are fixed heap-dynamic.
- C# provides fixed heap-dynamic.
- 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";
```

The array name will have eight elements, because all strings are terminated with a null character (zero)

- 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
- Example in Python:

```
mixedList = [1, 2, "three", 4]
```

# 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)
- The C-based languages do not provide any array operations.
- Perl supports array assignments but does not support comparisons.
- Python's array assignments, but they are only reference changes. Python also supports array catenation and element membership operations
- Ruby also provides array catenation

# Arrays Operations

---

- In APL , the four basic arithmetic operations are defined for vectors (single-dimensional arrays) and matrices.

$C = A * B$

$C = A + B$

- APL includes a collection of unary operators for vectors and matrices, some of which are as follows (where V is a vector and M is a matrix):

$\phi V$  reverses the elements of V

$\phi M$  reverses the columns of M

$\theta M$  reverses the rows of M

$\emptyset M$  transposes M (its rows become its columns and vice versa)

$\div M$  inverts M

# Rectangular and Jagged Arrays

---

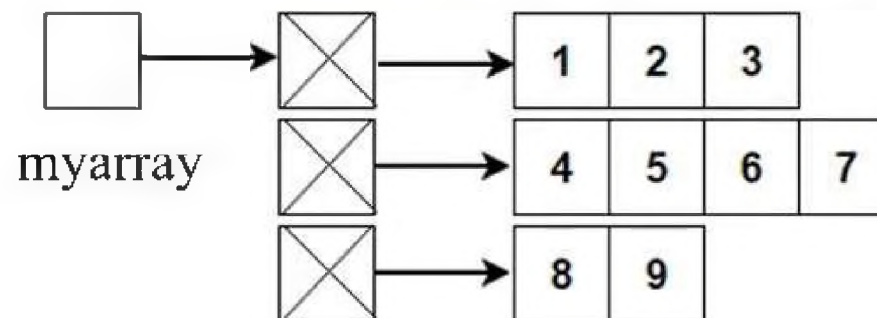
- A **rectangular array** is a multi-dimensional array in which all of the rows have the same number of elements and all columns have the same number of elements
- A **jagged matrix** has rows with varying number of elements
  - Possible when multi-dimensional arrays actually appear as arrays of arrays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

# Rectangular and Jagged Arrays

---

- Jagged Arrays

```
int myarray[][] = new int[][]{  
    new int[] { 1, 2, 3 };  
    new int[] { 4, 5, 6, 7 };  
    new int[] { 8, 9 };  
};
```



# 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
- For example, if **A** is a matrix, then the first row of **A** is one possible slice, as are the last row and the first column.



# 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 (those elements with the subscripts 3, 4, and 5)

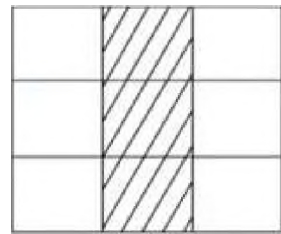
`mat[0][0:2]` is the first and second element of the first row of `mat`

- Ruby supports slices with the `slice` method

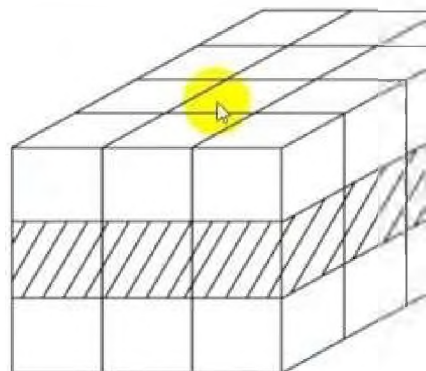
`list.slice(2, 2)` returns the third and fourth elements of `list`

# Slice Examples

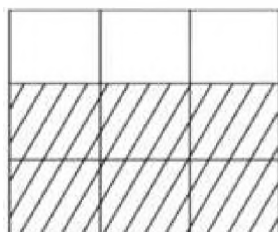
- Fortran 95



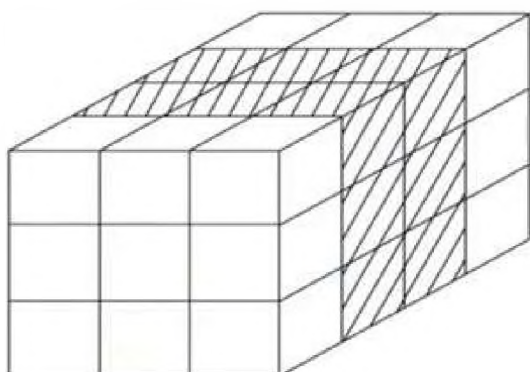
`MAT (1:3, 2)`



`CUBE (2, 1:3, 1:4)`



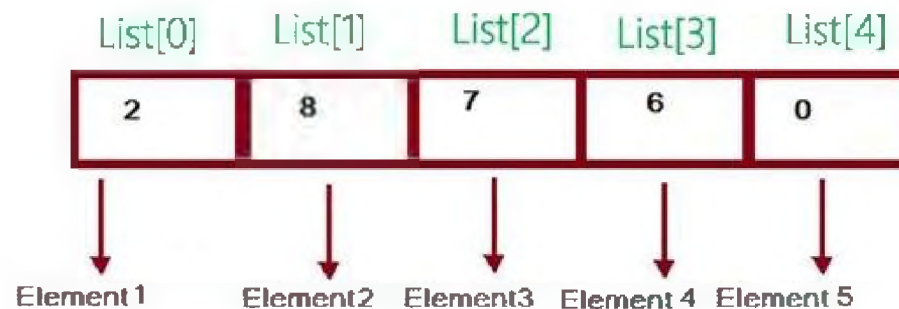
**MAT (2:3, 1:3)**



**CUBE (1:3, 1:3, 2:3)**

# 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 Single-dimensional Arrays

---

- The descriptor includes information required to construct the access function.

A compile-time descriptor  
for a Single- dimensional  
array

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

# Accessing Multi-dimensional Arrays

---

- Two common ways:
  - Row major order (by rows) – used in most languages
    - For example, if the matrix had the values  
3 4 7  
6 2 5  
1 3 8  
it would be stored in row major order as  
3, 4, 7, 6, 2, 5, 1, 3, 8
  - Column major order (by columns) – used in Fortran

# Accessing Multi-dimensional Arrays

---

A compile-time descriptor  
for a Multi-dimensional  
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[i,j]) = address of a [row\_lb,col\_lb] +  
(((i - row\_lb) \* n) + (j - col\_lb)) \* 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

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
- 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

---

- Associative arrays in Perl are called **hashes**, because in the implementation their elements are stored and retrieved with hash functions.
  - Every hash variable name must begin with a percent sign (%).
  - Each hash element consists of two parts:
    - a key, which is a string,
    - a value, which is a scalar (number, string, or reference).

# 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"};
```

- Elements can be added using

```
$hi_temps{"Sun"} = 75;
```

- To empty array using

```
$hi_temps = ();
```

# Associative Arrays in Perl

---

- The `exists` operator returns true or false, depending on whether its operand key is an element in the hash.

```
if (exists $salaries{"Shelly"})
```

- The size of a Perl hash is dynamic, It grows when an element is added and shrinks when an element is deleted, and also when it is emptied by assignment of the empty literal.
- `Python's` associative arrays, which are called `dictionaries`, are similar to those of `Perl`, except the values are all references to objects.
- The keys in `PHP's` arrays, can be integers or strings.

# Record Types

---

- A *record* is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names
- The individual elements in the record are not of the same type or size.
- For example, information about a college student might include name, student number, grade point average, and so forth.

# Record Types

---

- In C, C++, and C#, records are supported with the struct data type.
  - In C++, structures are a minor variation on classes.
  - C# structs are stack-allocated value types, as opposed to class objects, which are heap-allocated reference types.
  - In Python and Ruby, records can be implemented as hashes, which themselves can be elements of arrays.
- Design issues:
  - What is the syntactic form of references to the field?
  - Are elliptical references allowed



# Record Types

---

- The main difference between Arrays and Records:
  - The fundamental difference between a record and an array is that record elements, or fields, are not referenced by indices.
  - Another difference between arrays and records is that records in some languages are allowed to include unions.



# 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.
```

- The PIC clauses show the formats of the field storage locations, with X(20) specifying 20 alphanumeric characters and 99V99 specifying four decimal digits with the decimal point in the middle.

# Definition of Records

---

- 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
- 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

# Evaluation and Comparison to Arrays

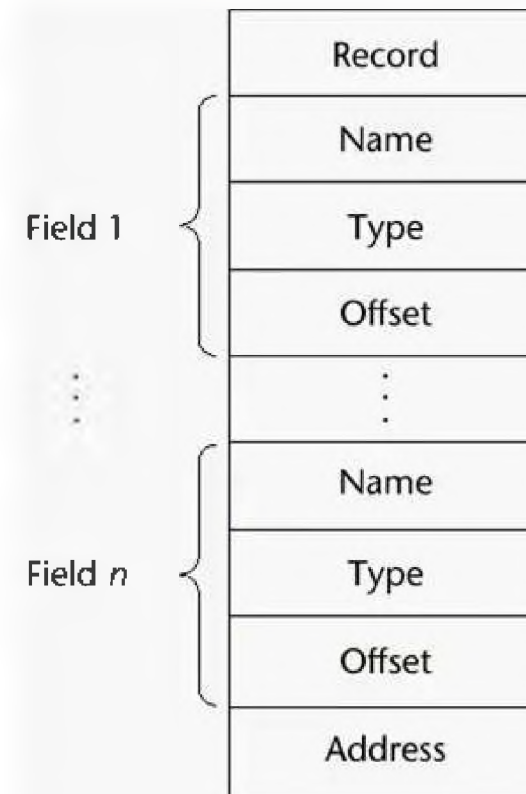
---

- 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)

```
myTuple[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;
```

- Values of this type consist of an integer and a real.

- 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

`hd [5, 7, 9]` is 5

`tl [5, 7, 9]` is `[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

`List.hd [1; 3; 5; 7]`, which returns 1

- 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(6)` 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]]
```

This defines a list of the squares of the numbers from 1 to 10.

- F#'s List Comprehensions

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

This statement creates the array [1; 4; 9; 16; 25] and names it 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 issue
  - Should type checking be required?
  - Should unions be embedded in records?



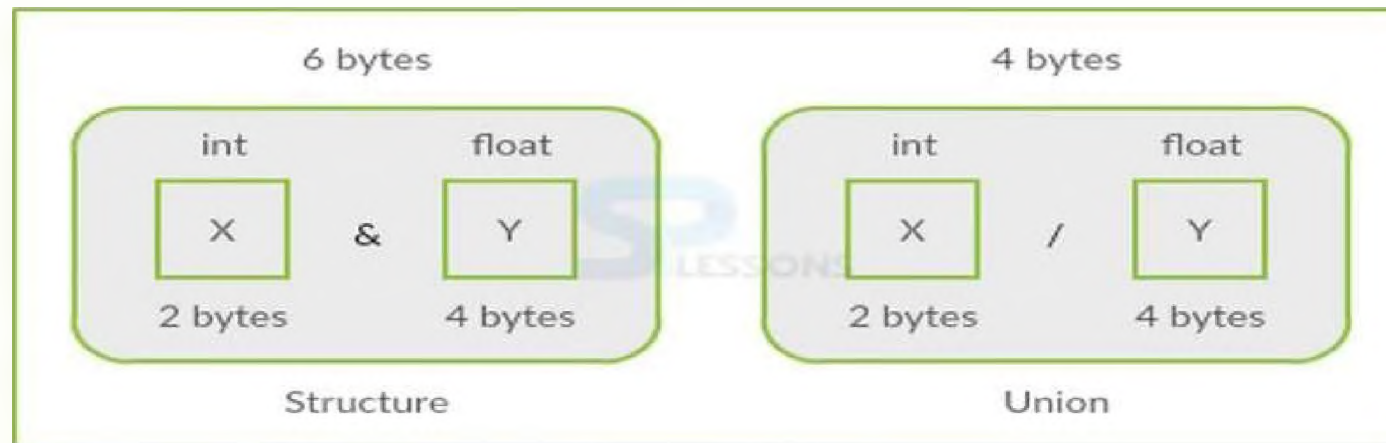
# Unions Types

## Structure

```
struct Emp  
{  
    int X;  
    float Y;  
}
```

## Unions

```
union Emp  
{  
    int X;  
    float Y;  
}
```





# 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 C


---

```
union flexType {  
    int intEl;  
    float floatEl;  
};
```

```
union flexType e11;  
float x;
```

```
...
```

```
e11.intEl = 27;  
x = e11.floatEl;
```



---

This last assignment is not type checked, because the system cannot determine the current type of the current value of `ell`, so it assigns the bit string representation of 27 to the `float` variable `x`, which of course is nonsense.

# 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
```

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

# Evaluation of Unions

---

- Free unions are unsafe
  - Do not allow type checking
- unions can be safely used, as in their design in ML, Haskell, and F#.
- 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*
- The value *nil* is not a valid address and is used to indicate that a pointer cannot currently be used to reference a memory cell.
- 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*)

# Pointer and Reference Types

---

- Pointers, unlike arrays and records, are not structured types, although they are defined using a type operator (\* in C and C++).
- They are also different from scalar variables because they are used to reference some other variable, rather than being used to store data.
- Two categories of variables are called **reference types** and **value types**.

# 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

---

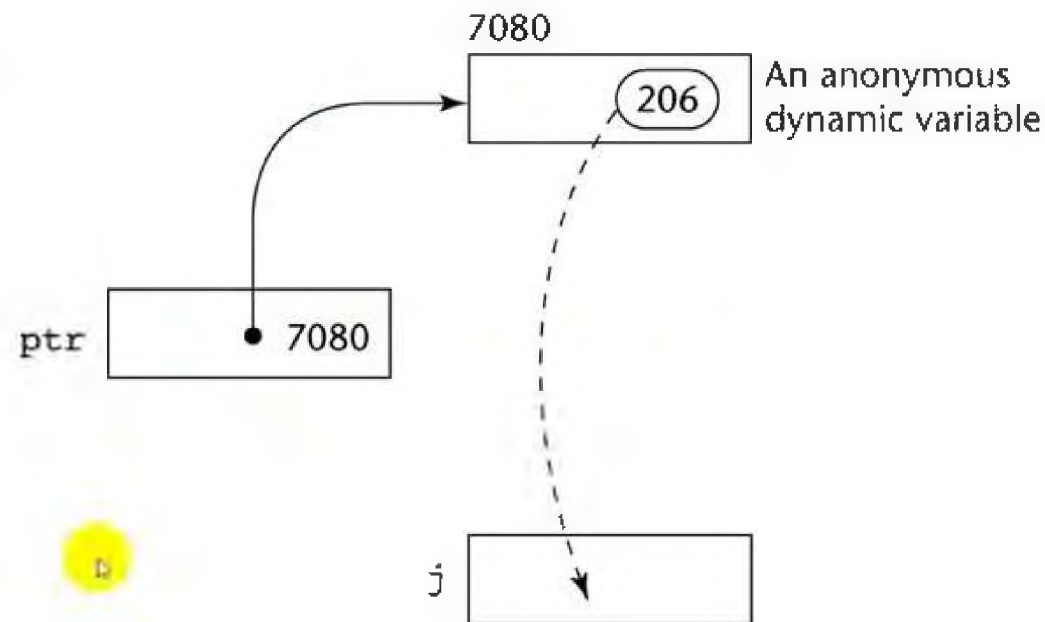
- Two fundamental operations: assignment and dereferencing
- **Assignment** is used to set a pointer variable's value to some useful address
- **Dereferencing** yields the value stored at the location represented by the pointer's value
  - Dereferencing can be explicit or implicit
- In languages that support object-oriented programming, allocation of heap objects is specified with the **new** operator.
- C++, which does not provide implicit deallocation, uses **delete** as its deallocation operator.

# Pointer Assignment Illustrated

- C++ uses an explicit operation via `*`

`j = *ptr`

sets `j` to the value located at `ptr`



# Problems with Pointers

---

## 1. 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.
```

# Problems with Pointers

---

- Dangling pointers are dangerous for several reasons.
  - If the new variable is not the same type as the old one, type checks of uses of the dangling pointer are invalid.
  - If the new dynamic variable is the same type, its new value will have no relationship to the old pointer's dereferenced value.
  - If the dangling pointer is used to change the heap-dynamic variable, the value of the new heap-dynamic variable will be destroyed.
  - it is possible that the location now is being temporarily used by the storage management system, possibly as a pointer in a chain of available blocks of storage, thereby allowing a change to the location to cause the storage manager to fail.

# Problems with Pointers

---

## 2. 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*



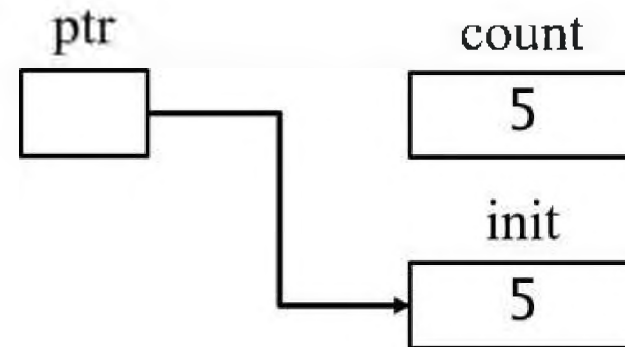
# Pointers in C and C++

---

- 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++

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



```
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]`

# Pointer Arithmetic in C and C++

---

- Some recent languages, such as **Java**, have replaced pointers completely with reference types, which, along with implicit deallocation, minimize the primary problems with pointers.
- A **reference type** is really only a pointer with restricted operations.

# Reference Types

---

- A **reference type** variable is similar to a pointer, with one difference:
  - A pointer refers to an address in memory,
  - reference refers to an object or a value in memory.
- 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

```
int result = 0;  
int &ref_result = result;
```

```
* * *  
ref_result = 100;
```

In this code segment, `result` and `ref_result` are aliases.

# Reference Types

---

- Java extends C++'s reference variables and allows them to replace pointers entirely

- References are references to objects, rather than being addresses

```
String str1;
```

```
...
```

```
str1 = "This is a Java literal string";
```

- In this code, `str1` is defined to be a reference to a String class instance or object. It is initially set to `null`.
  - The subsequent assignment sets `str1` to reference the String object, "This is a Java literal string".

# Reference Types

---

- C# includes both the references of Java and the pointers of C++
- All variables in the object-oriented languages Smalltalk, Python, Ruby, and Lua are references. They are always implicitly dereferenced.

# 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
- The references of Java and C# provide some of the flexibility and the capabilities of pointers, without the hazards.

# Representations of Pointers

---

- In most larger computers, pointers and references are single values stored in memory cells.
- Intel microprocessors use segment and offset

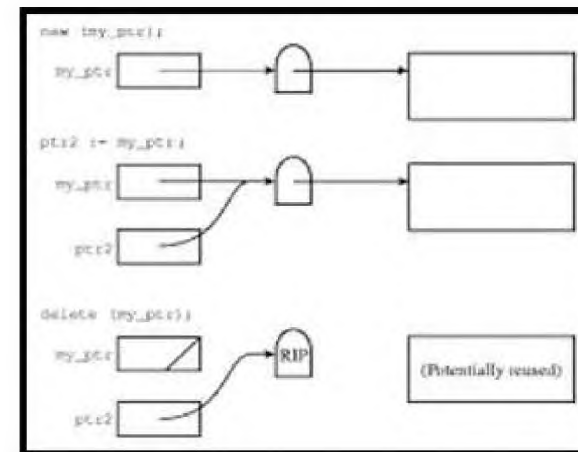


# Solutions of Dangling Pointer Problem

## 1. Tombstone:

every heap-dynamic variable includes a special cell, called a **tombstone**, that is itself 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, indicating that the heap-dynamic variable no longer exists.
- Costly in time and space



# Solutions of Dangling Pointer Problem

---

## 1. Tombstone: (continued)

Costly in time and space

- tombstones are never deallocated, their storage is never reclaimed.
- Every access to a heap-dynamic variable through a tombstone requires one more level of indirection.
- no widely used for the most of the programming languages.

# Solutions of Dangling Pointer Problem

---

## 2. 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 a heap-dynamic variable is allocated, a lock value is created and placed both in the lock cell of the heap-dynamic variable and in the key cell of the pointer that is specified in the call to **new**.
- Every access to the dereferenced pointer compares the key value of the **pointer** to the lock value in the heap-dynamic variable.
  - If they match, the access is legal;
  - otherwise the access is treated as a run-time error.

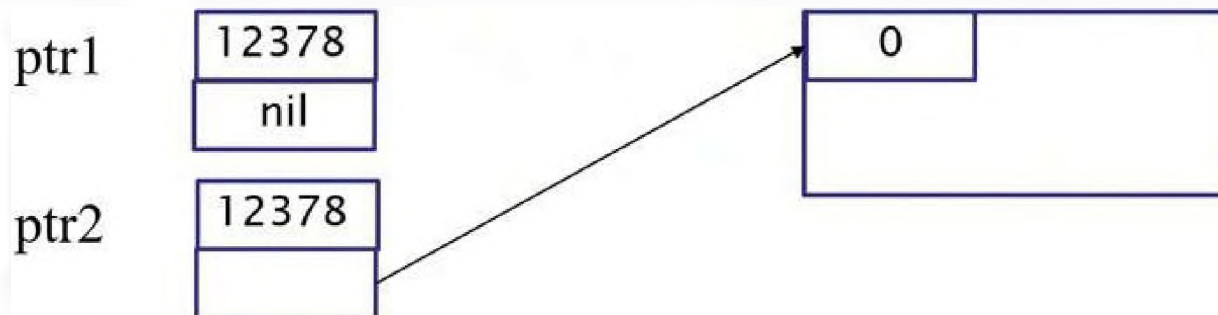
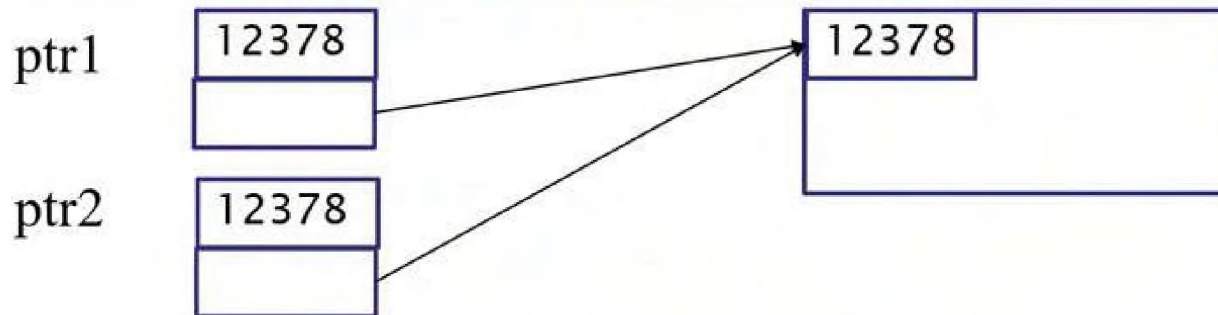
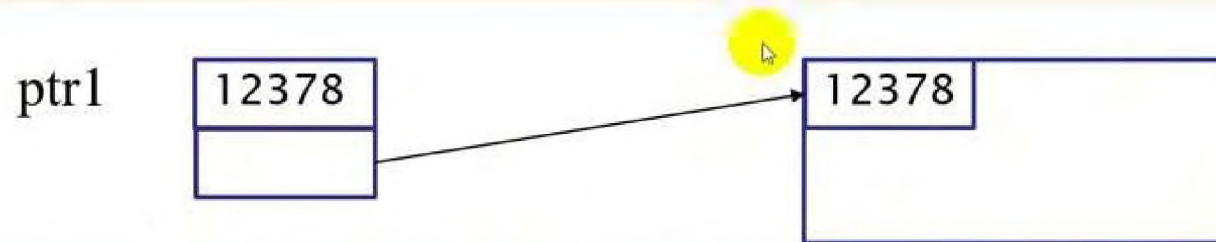
# Solutions of Dangling Pointer Problem

---

## 2. Locks-and-keys: (continued)

- Any copies of the pointer value to other pointers must copy the key value.
- When a heap- dynamic variable is deallocated with dispose, its lock value is cleared to an illegal lock value.
- Then, if a pointer other than the one specified in the dispose is dereferenced, its address value will still be intact, but its key value will no longer match the lock, so the access will not be allowed.
- Lisp, Java and C# use this approach for their reference variables

# Solutions of Dangling Pointer Problem



# Heap Management

---

- A very complex run-time process
- Two separate situations: all heap storage is allocated and deallocated in units of **Single-size** or **variable-size**.

- **Single-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
- If the reference counter reaches zero, it means that no program pointers are pointing at the cell, and it has thus become garbage and can be returned to the list of available space.
- Advantage: it is intrinsically incremental, so significant delays in the application execution are avoided

# Reference Counter

---

- Disadvantages:
  - space required for the counters is significant for each cell.
  - execution time required to maintain the counter values.
  - complications for cells connected circularly. The problem here is that each cell in the circular list has a reference counter value of at least 1, which prevents it from being collected and placed back on the list of available space.



# 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

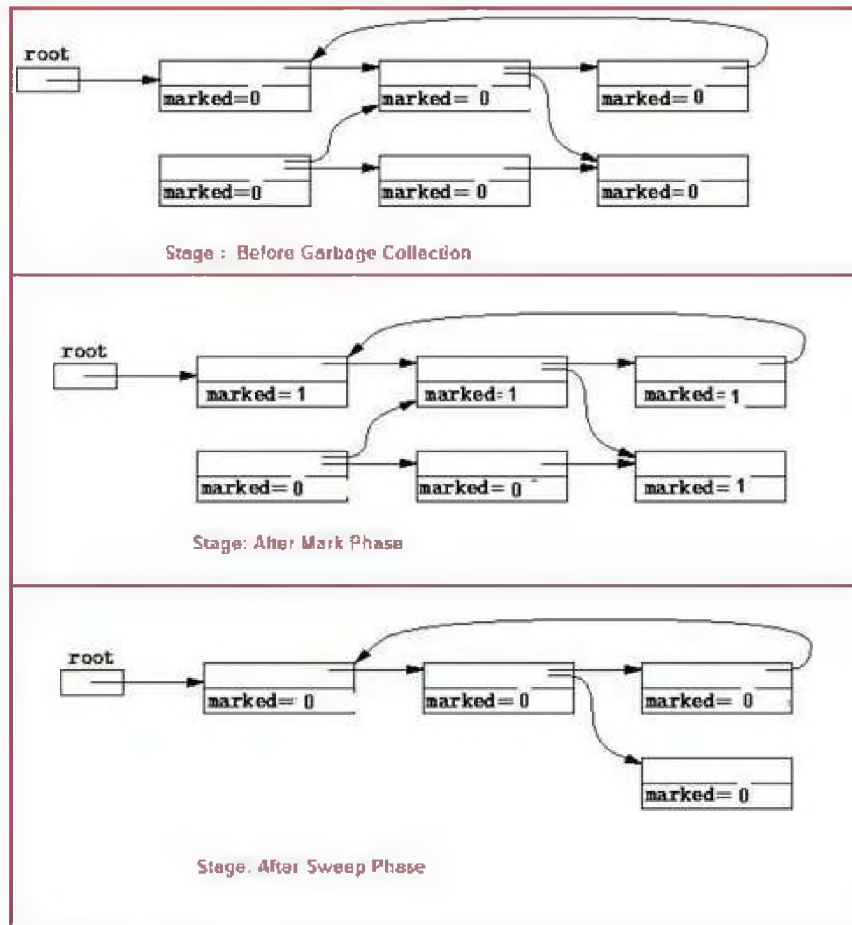


# Mark–Sweep

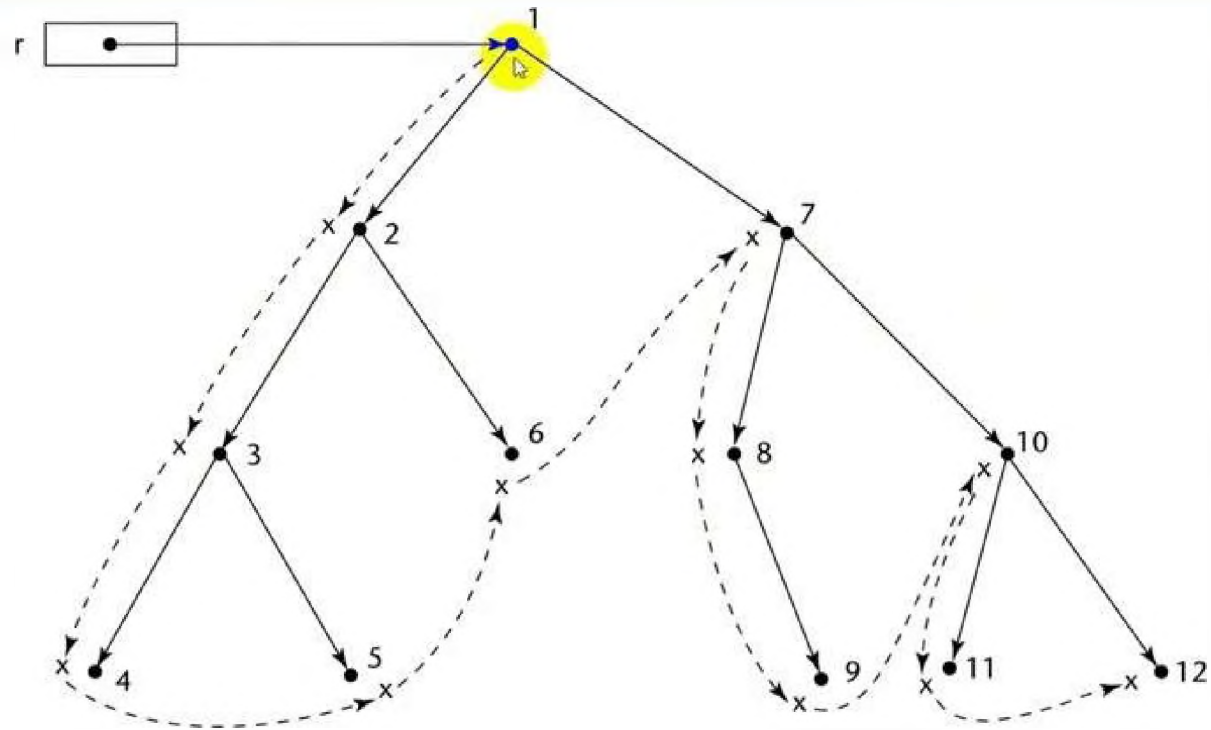
---

- The mark– sweep process consists of three distinct phases.
  - The first phase: all cells in the heap have their indicators set to indicate they are garbage.
  - The second phase: called the **marking** phase, Every pointer in the program is traced into the heap, and all reachable cells are marked as not being garbage.
  - The third phase, called the **sweep** phase, is executed: All cells in the heap that have not been specifically marked as still being used are returned to the list of available space.

# Mark-Sweep



# Marking Algorithm



Dashed lines show the order of node\_marking

**The method reverses pointers as it traces out linked structures.**

# Mark–Sweep

---

- Disadvantages: in its original form, it was done too infrequently.
  - a significant delay in the progress of the application to find the available cells.
  - Contemporary mark–sweep algorithms avoid this by doing it more often—called incremental mark–sweep
    - Incremental mark–sweep garbage collection occurs more frequently, long before memory is exhausted, making the process more effective in terms of the amount of storage that is reclaimed. Furthermore, time of required for each run of the process.
    - to perform the mark–sweep process on parts, rather than all of the memory associated with the application, at different times.

# 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
    - Because the cells are different sizes, scanning them is a problem.
    - solution is to require each cell to have the cell size as its first field.
    - Some times need more time than its counterpart for fixed-size cells.
  - The marking process is nontrivial
  - Maintaining the list of available space is another source of overhead

## Variable-Size Cells (continued)

---

- Maintaining the list of available space is another source of overhead
  - The list can begin with a single cell consisting of all available space.
  - Requests for segments simply reduce the size of this block. Reclaimed cells are added to the list.
  - The problem is that before long, the list becomes a long list of various-size segments, or blocks.
  - This slows allocation because requests cause the list to be searched for sufficiently large blocks.
  - Eventually, the list may consist of a large number of very small blocks, which are not large enough for most requests.
  - At this point, adjacent blocks may need to be collapsed into larger blocks.

# 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*.
- For example, if an *int* variable and a *float* variable are added in *Java*, the value of the *int* variable is coerced to *float* and a *floating-point add is done*.



## Type Checking (continued)

---

- A type error is the application of an operator to an operand of an inappropriate type
- For example, in the *original version of C*, if an **int** value was passed to a function that expected a **float** value, a *type error would occur*.
  - Because compilers for that language did not check the types of parameters.



## 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
  - JavaScript and PHP, because of their dynamic type binding, allow only dynamic type checking.
- It is better to detect errors at compile time than at run time, because the earlier correction is usually less costly.

# Type Checking (continued)

---

- Type checking is complicated when a language allows a memory cell to store values of different types at different times during execution.
  - Such memory cells can be created with C and C++ unions and the discriminated unions of ML, Haskell, and F#.
  - In these cases, type checking, if done, must be dynamic and requires the run-time system to maintain the type of the current value of such memory cells.

# Strong Typing

---

- 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
- 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)

---

- The coercion rules of a language have an important effect on the value of type checking.
  - For example, expressions are strongly typed in Java. However, an arithmetic operator with one floating-point operand and one integer operand is legal. The value of the integer operand is coerced to floating-point, and a floating-point operation takes place.
  - However, the coercion also results in a loss of one of the benefits of strong typing—error detection.
  - For example, suppose a program had the `int` variables `a` and `b` and the `float` variable `d`. Now, if a programmer meant to type `a + b`, but mistakenly typed `a + d`, the error would not be detected by the compiler. The value of `a` would simply be coerced to float. So, the value of strong typing is weakened by coercion.

## Strong Typing (continued)

---

- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus ML and F#)
- Java and C# have half as many assignment type coercions as C++, so their error detection is better than that of C++, but still not nearly as effective as that of ML and F#.

