CS49K Programming Languages

Chapter 8: Composite Types

LECTURE 10: COMPOSITE TYPES

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

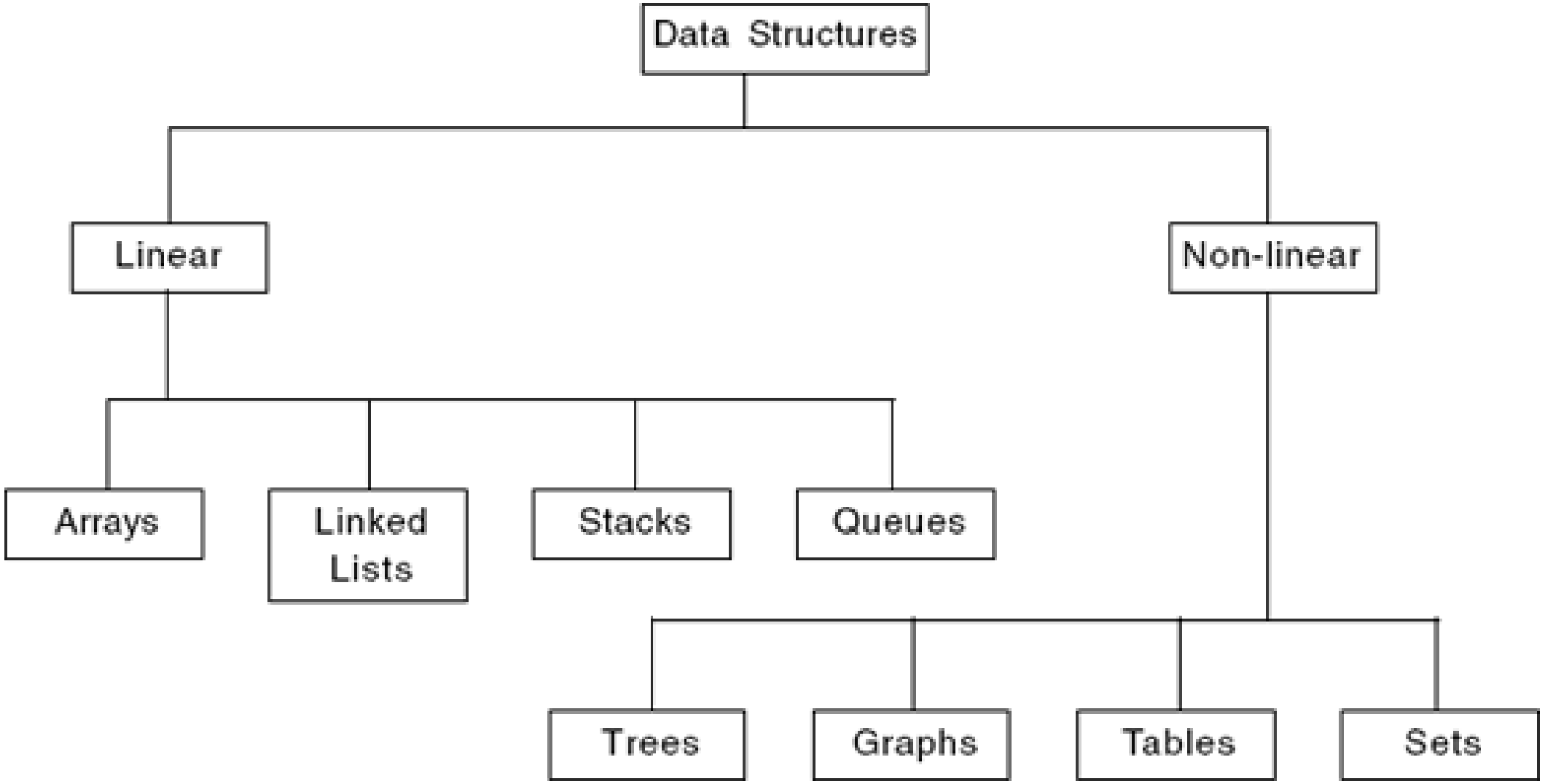
- Records (Structures) and Variants (Unions)
- Array
- Strings
- Sets
- Pointer/Reference/Alias
- Dangling Reference
- Garbage Collection
- File

Composite Data Types

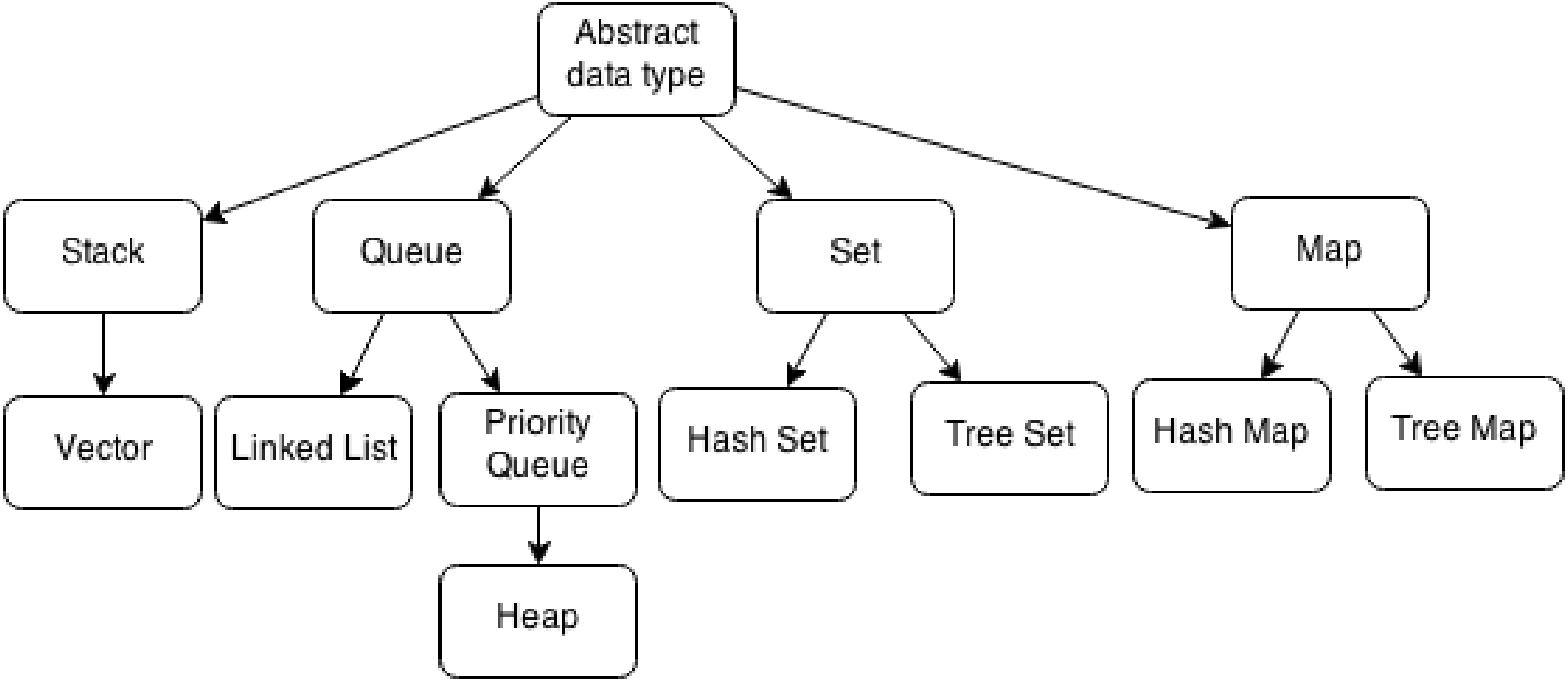
SECTION 1

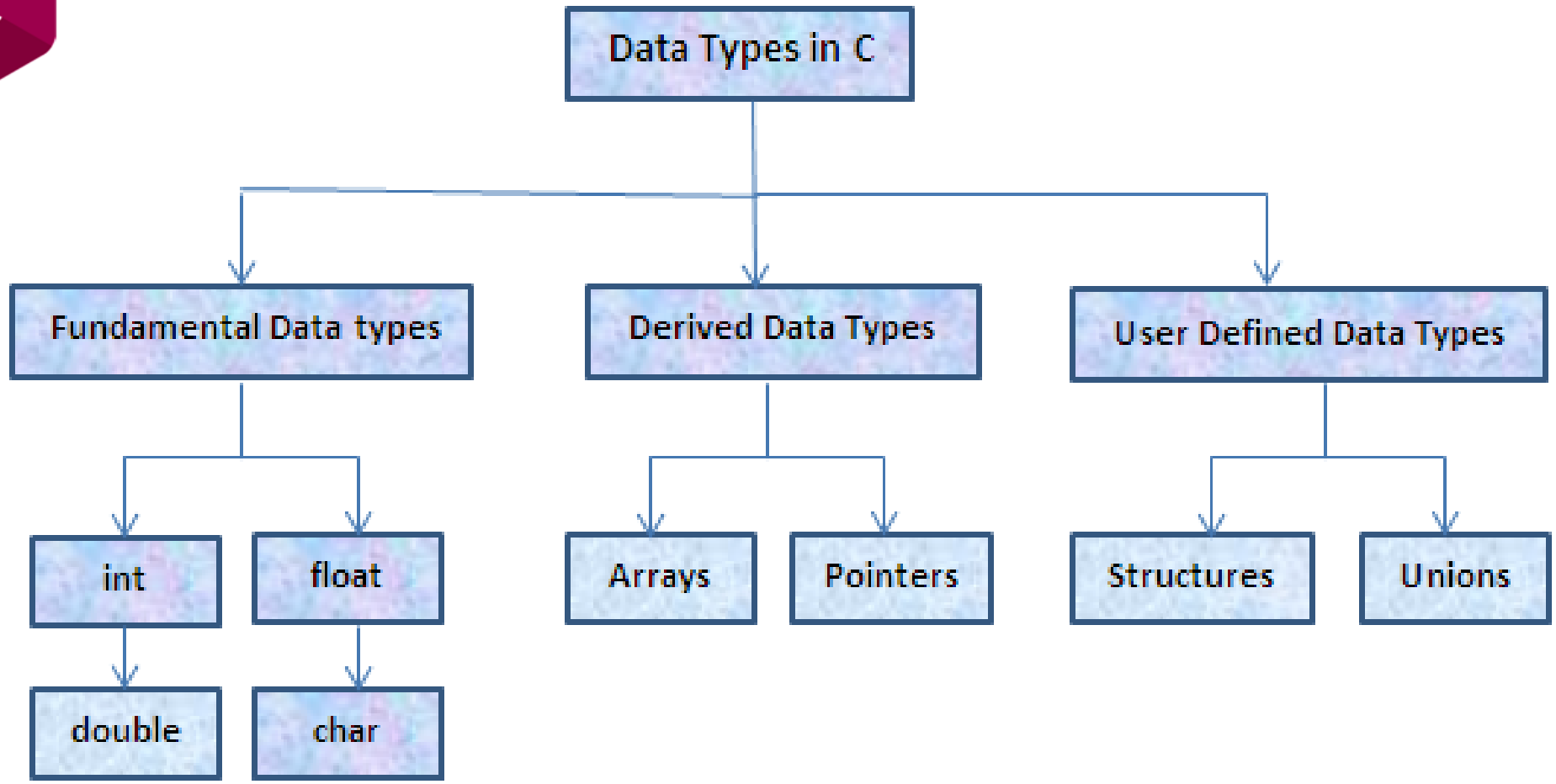
Composite Types

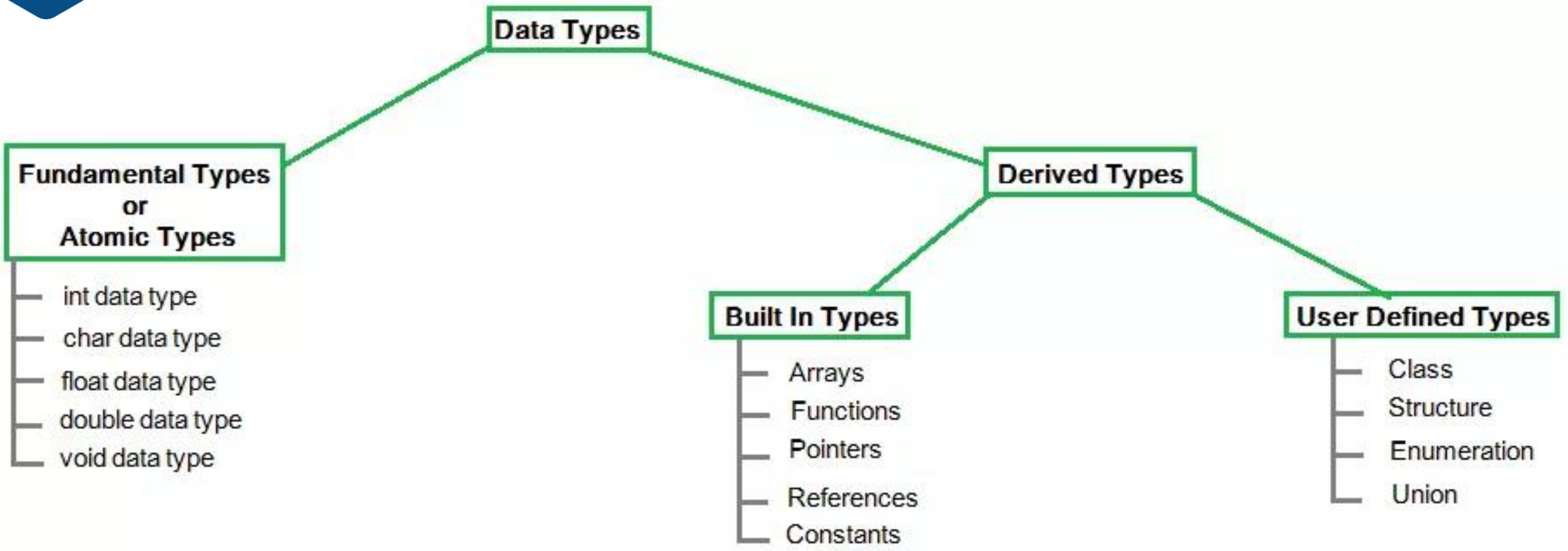
- Records (struct)
- Variant records (union)
- Arrays
  - Strings (Sometime array, sometime object, and sometimes a separate type.)
- Sets
- Dictionary (map)
- Pointers
- Lists
- Files

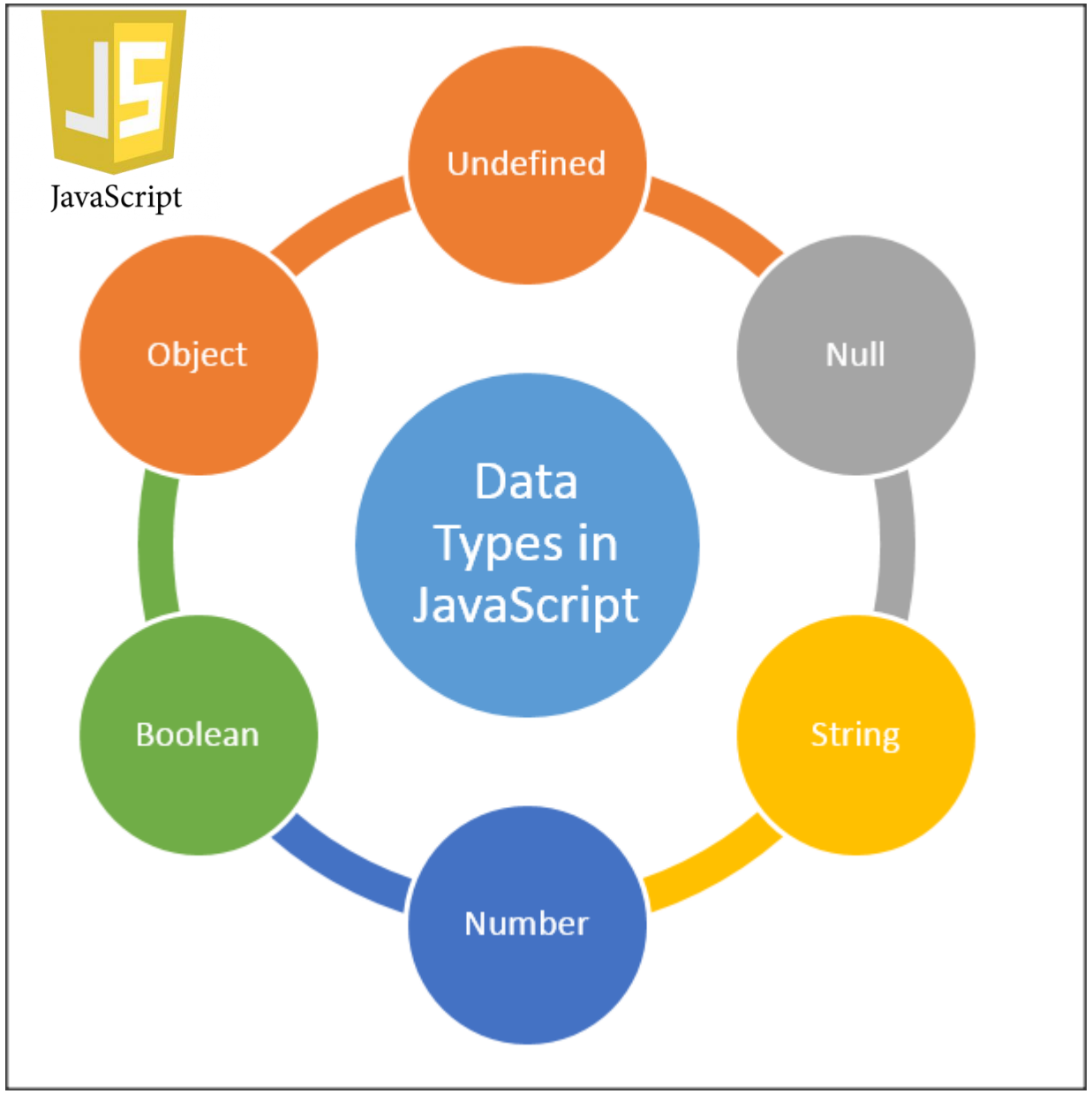

Data Structures
Linear
Non-linear
Arrays
Linked Lists
Stacks
Queues
Trees
Graphs
Tables
Sets

ADT (Derived)


Abstract data type
Stack
Queue
Set
Map
Vector
Linked List
Priority Queue
Hash Set
Tree Set
Hash Map
Tree Map
Heap

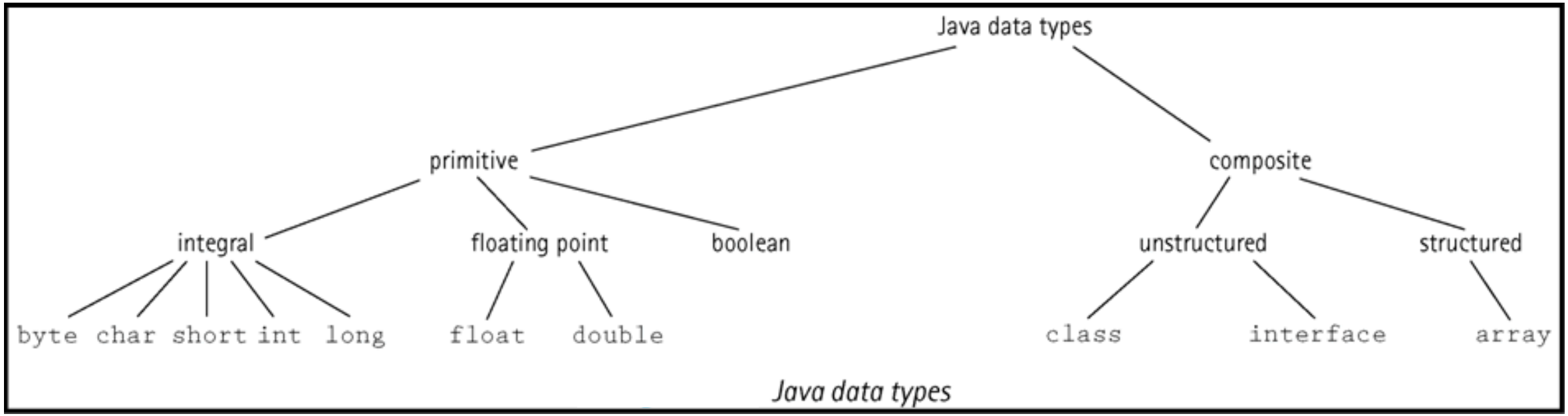

Data Types in C
Fundamental Data types
Derived Data Types
User Defined Data Types
int
float
double
char
Arrays
Pointers
Structures
Unions


Data Types
Fundamental Types or Atomic Types
int data type
char data type
float data type
double data type
void data type
Derived Types
Built In Types
Arrays
Functions
Pointers
References
Constants
User Defined Types
Class
Structure
Enumeration
Union



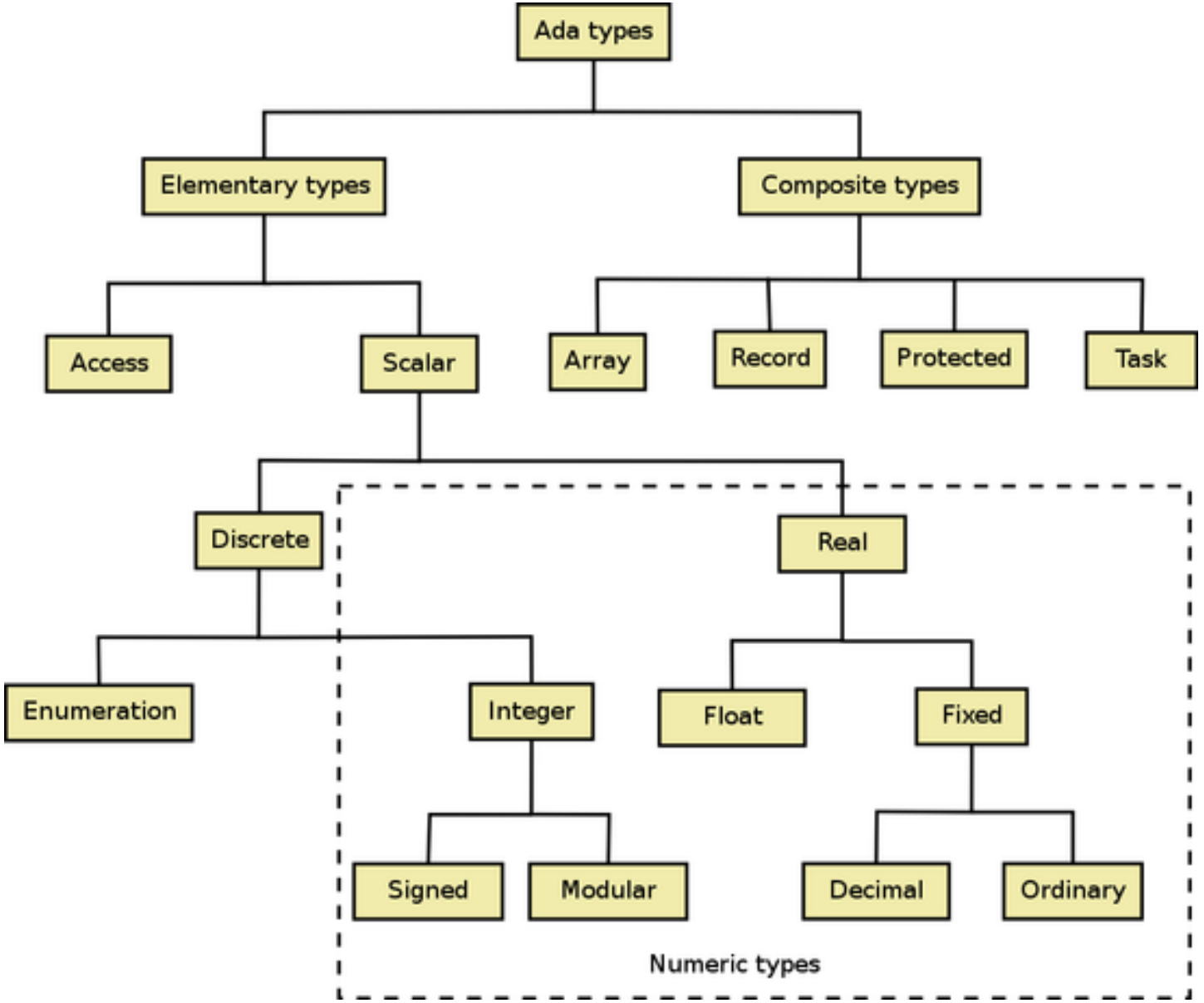
Variables

Dictionary
List
Numbers
Tuple
Set
Strings



Java data types


IN STRONG TYPING WE TRUST
Ada
1983 1995 2005 2012


Ada types
Elementary types
Composite types
Access
Scalar
Array
Record
Protected
Task
Discrete
Real
Enumeration
Integer
Float
Fixed
Signed
Modular
Decimal
Ordinary
Numeric types

Records (Structures)

SECTION 2

Records(Structure)

- usually laid out contiguously
- possible holes for alignment reasons
- smart compilers may rearrange fields to minimize holes (C compilers promise not to)
- implementation problems are caused by records containing dynamic arrays
  - we won't be going into that in any detail

Records (Structures) and Variants (Unions)

- Unions (variant records)
  - overlay space
  - cause problems for type checking
- Lack of tag means you don't know what is there
- Ability to change tag and then access fields hardly better
  - can make fields "uninitialized" when tag is changed (requires extensive run-time support)
  - can require assignment of entire variant, as in Ada

Syntax for Record(Structure)

C:

```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    _Bool metallic;
};
```

Pascal:

```
type two_chars = packed array [1..2] of char;
    (* Packed arrays will be explained in Example 7.43.
       Packed arrays of char are compatible with quoted strings. *)
type element = record
    name : two_chars;
    atomic_number : integer;
    atomic_weight : real;
    metallic : Boolean
end;
```

Accessing Record Fields

```
element copper;
const double AN = 6.022e23;      /* Avogadro's number */
...
copper.name[0] = 'C'; copper.name[1] = 'u';
double atoms = mass / copper.atomic_weight * AN;
```

Syntax for Record(Structure)

C:

```
struct ore {
    char name[30];
    struct {
        char name[2];
        int atomic_number;
        double atomic_weight;
        _Bool metallic;
    } element_yielded;
};
```

Alternatively, one could say

```
struct ore {
    char name[30];
    struct element element_yielded;
};
```

Records (Structures)

Memory layout and its impact (structures)

```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    _Bool metallic;
};
```

- The fields of a record are usually stored in adjacent locations in memory.
- In an array of **`element`**s, most compilers would devote 20 bytes to every member of the array.

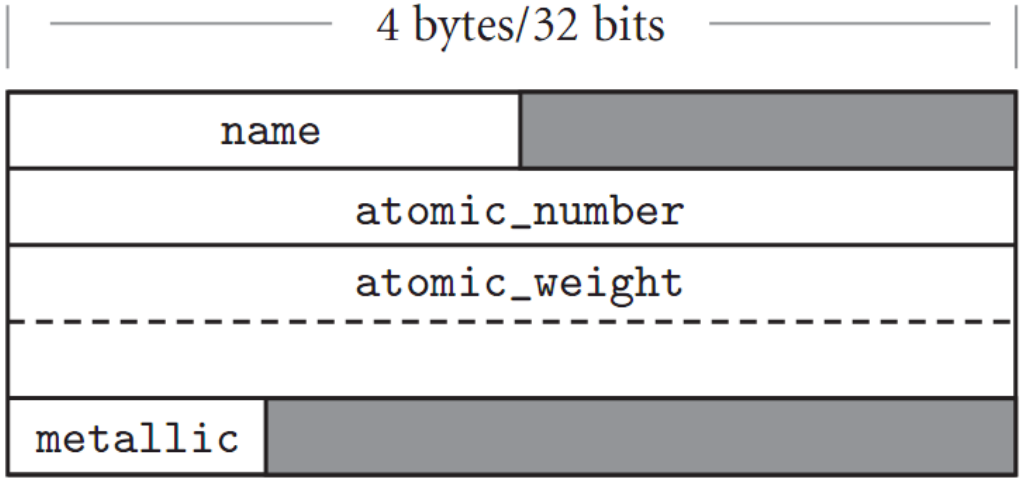
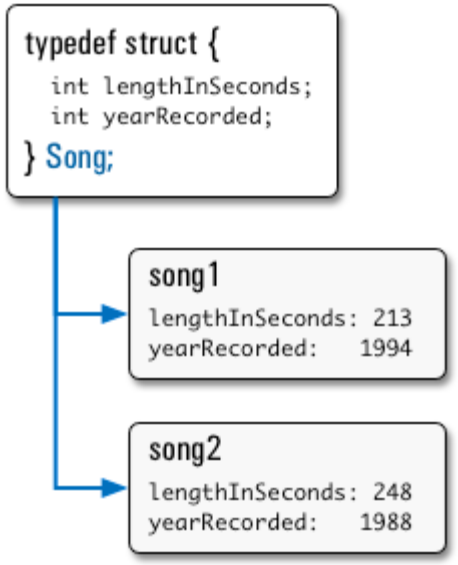


Figure 8.1 Likely layout in memory for objects of type element on a 32-bit machine. Alignment restrictions lead to the shaded “holes.”

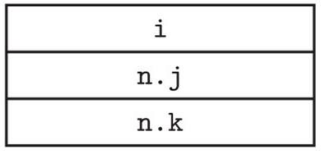
struct Tag and typedef in C/C++

- One of the peculiarities of the **C** type system is that **struct** tags are not exactly type names. In Example 8.1, the name of the type is the two-word phrase struct element. We used this name to declare the **`element_yielded`** field of second **`struct`** in Example 8.3.
- To obtain one-word name, one can say **`typedef struct element element_t`**, or even **`typedef struct element element;`** **`struct`** tags and **`typedef`** names have separate name spaces, so the same name can be used in each.
- C++ eliminates this idiosyncrasy by allowing the **`struct`** tag to be used as the type name without the struct prefix; in effect, it performs the **`typedef`** implicitly.



Nested Record as Value

```
struct T {
    int j;
    int k;
};
struct S {
    int i;
    struct T n;
};
```




```
class T {
    public int j;
    public int k;
}
class S {
    public int i;
    public T n;
}
```

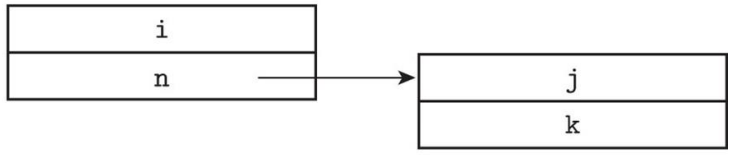


Figure 8.2 Layout of a nested struct(class) in C(top) and Bottom(Java)

C:

```
struct S s1;
struct S s2;
s1.n.j = 0;
s2 = s1;                    /* copy the struct data fields */
s2.n.j = 7;
printf("%d\n", s1.n.j);     /* prints */
```

Java:

```
S s1 = new S();
s1.n = new T();                    // fields initialized to 0
S s2 = s1;
s2.n.j = 7;
System.out.println(s1.n.j);        // prints 7
```

Record (structure)

Layout of Packed-Type

Ada:

```
type element = record
      …
end;
pragma  Pack(element);
```

- A few language allow the programmer to specify that a record type should be packed. (In Ada, like above.)
- When compiling with **gcc**, one uses an attribute:

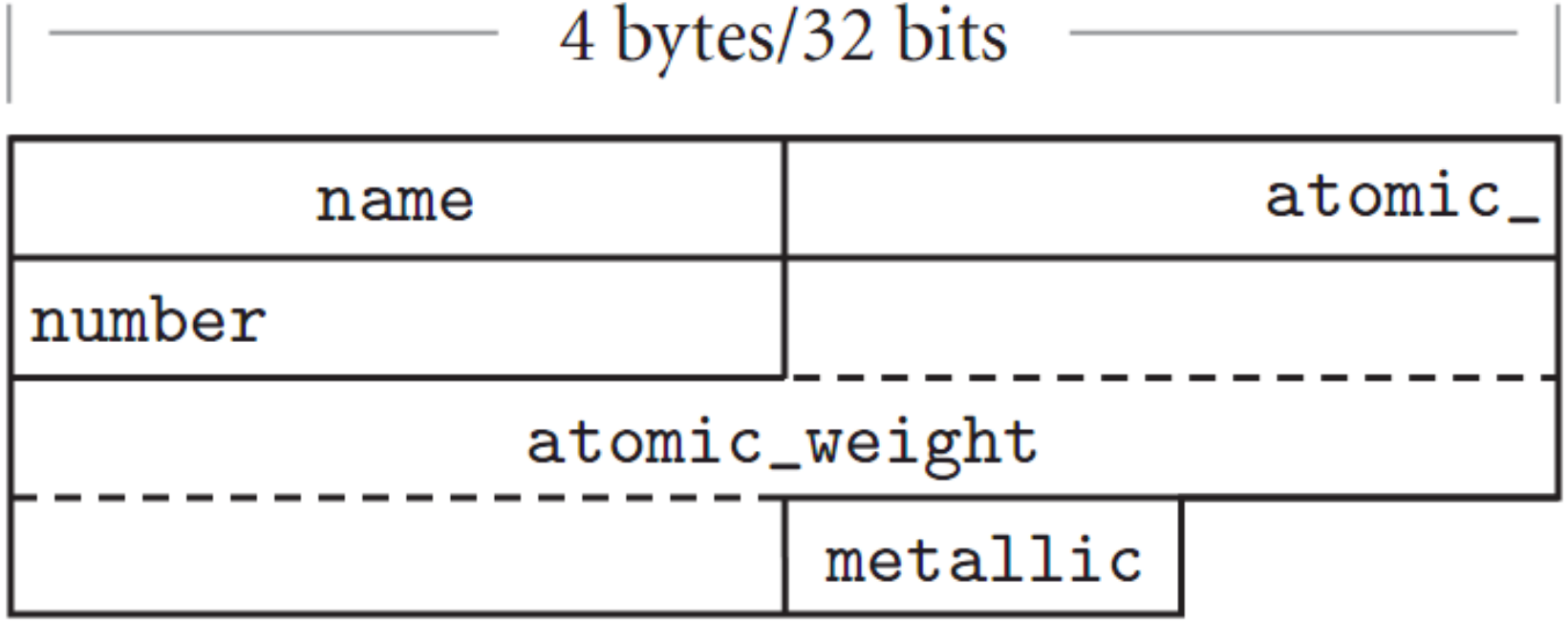
```
struct __attribute__ ((__packed__))
element {}
      …
}
```

- The Ada syntax is built into the language; the **gcc** syntax is a GNU extension.
- The directive asks the compiler to optimize for space instead of speed.
- Compiler will implement a packed record without holes.
- See Figure 8.3.

Records (Structures)

Memory layout and its Impact (packed record)

Figure 8.3 Likely memory layout for packed element records. The atomic_number and atomic_weight fields are nonaligned, and can only be read or written (on most machines) via multi-instructiosequences. (15 bytes)



Record (structure)

Assignment and Comparison of Records

- Most languages allow a value to be assigned to an entire record in a single operation:

```
my_element := copper;
```

- Ada allows records to be compared for equality.
- Many other languages (C-family) support though C++ allows the programmer to define the latter for individual record types.
- assignment but not equality testing, For small records, both copies and comparisons can be performed in-line on field-by-field basis.
- For longer records, we can save significantly on code space b deferring to a library routine.
- A `block_copy` routine can take source address, destination address, and length as arguments, but the analogous `block_compare` routine would fail on records with different data in the holes.
- One solution is to arrange for all holes to contain some predictable value, but this requires code in every elaboration point.
- Another is to have the compiler generate a customized field-by-field comparison routine for every record type.
- Different routines would be called to compare records f different types.
- Packing eliminates holes, but maybe costly.

Records (Structures) and Variants (Unions)

Memory layout and its impact (structures)

- A compromise it to sort a record's fields according to the size of their alignment constraints.
- All type-alignment fields might come first, followed by any half-word fields, word-aligned fields, and double-word-aligned fields. (Figure 8.4)
- C/C++ in order declared
- When a "packed" directive is essentially a nonbinding indication of the programmers priorities, bit lengths on field declarations are a binding specification of assembly-level layout.

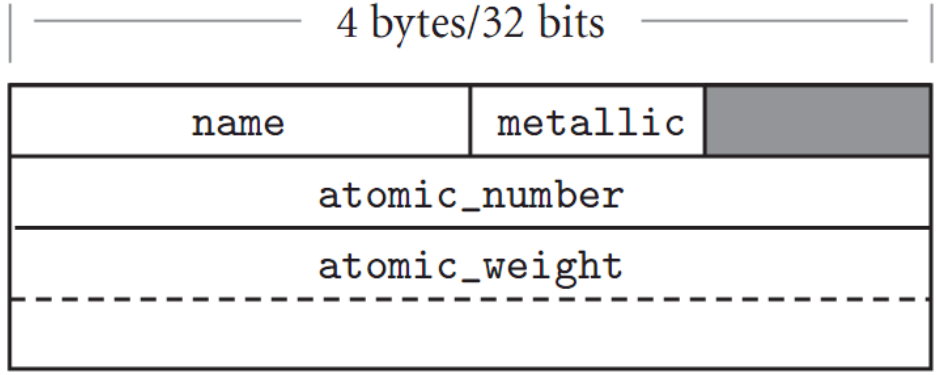


Figure 8.4 Rearranging record fields to minimize holes. By sorting fields according to the size of their alignment constraint, a compiler can minimize the space devoted to holes, while keeping the fields aligned.

Variants (Union)

SECTION 3

Variant Records (Unions)

C:

```
union {
        int i;
        double d;
        _Bool b;
};
```

- The overall size of this union would be that of its largest member (presumably **`d`**).
- Exactly which bytes of **`d`** would be overlapped by **`i`** and **`b`** is implementation dependent, and presumably influenced by the relative sizes of types, their alignment constraints, and the endian-ness of the hardware.

Variant Records (Unions)

- **Unions have been used for two main purposes:**
  1. Unions allow the same set of bytes to be interpreted in different ways at different times in system programs:
     - Non-converting type casts can be used to implement heap management routines.
     - Bits are not being reinterpreted, they are being used for independent purposes.
  2. Represents alternative sets of fields within a record.

Note: Used at input/output buffer of various interpretations.

Variant Records (Unions)

- Traditional C Unions were awkward when used for this purpose.
- A much cleaner syntax appeared in the variant records of Pascal and its successors, which allow the programmer to specify that certain fields within a record should overlap in memory.
- Similar functionality was added to C11 and C++11 in the form of anonymous unions.

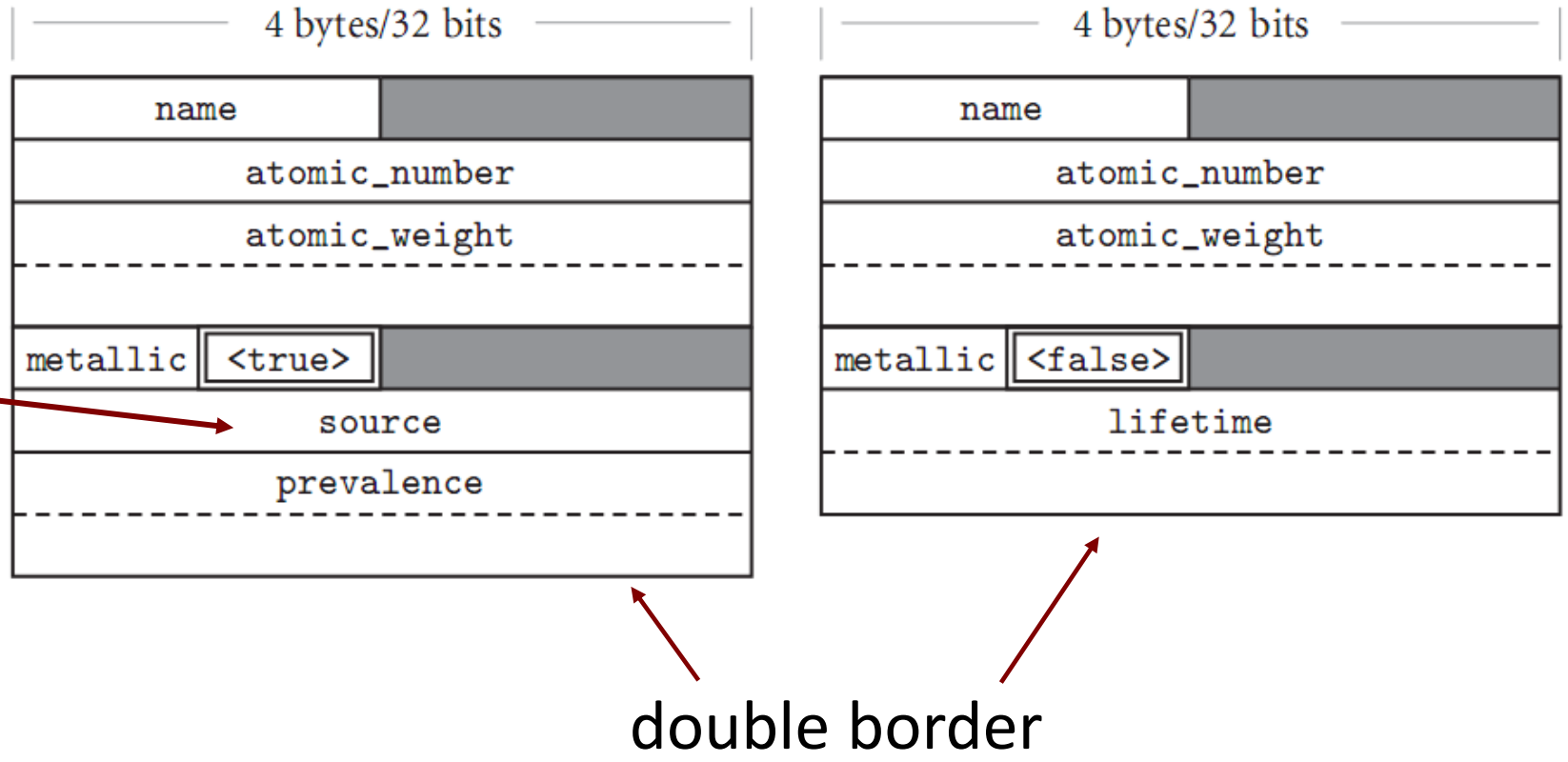
```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    _Bool metallic;
    _Bool naturally_occurring;
    union {
        struct {
            char *source;
                /* textual description of principal commercial source */
            double prevalence;
                /* fraction, by weight, of Earth's crust */
        } natural_info;
        double lifetime;
            /* half-life in seconds of most stable known isotope */
    } extra_fields;
} copper;
```

From Supplemental Material of the textbook.

Variants (Unions)

Memory layout and its impact (Unions)

Figure 8.16 (CD) Likely memory layouts for element variants. The value of the naturally_ occurring field (shown here with a double border) is intended to indicate which of the interpretations of the remaining space is valid. **Field source** is assumed to point to a string that has been independently allocated.

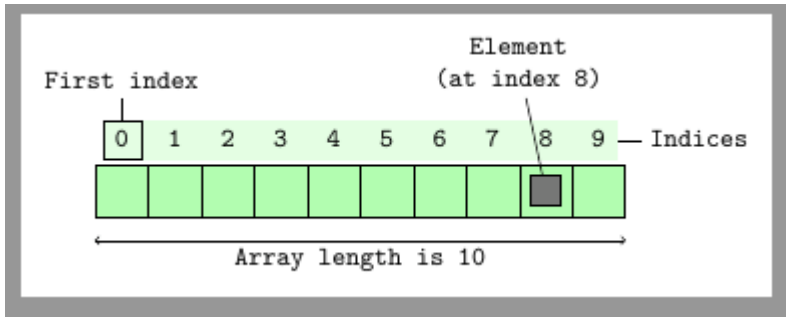


Arrays I

Definition

SECTION 4

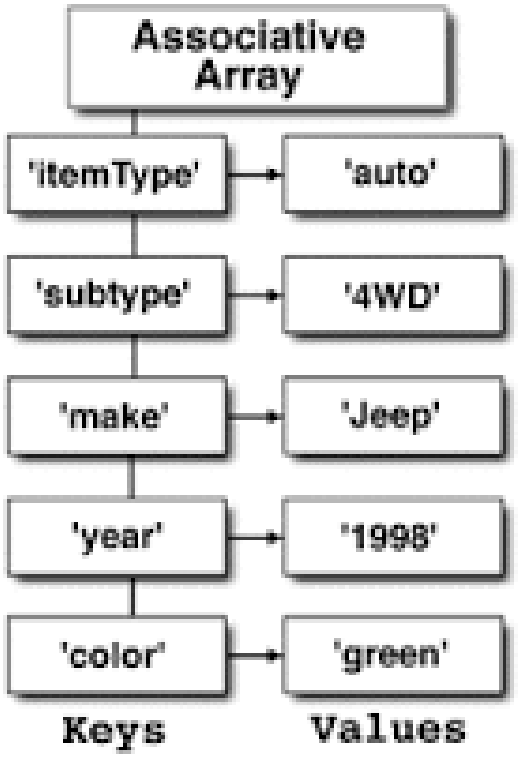
Arrays



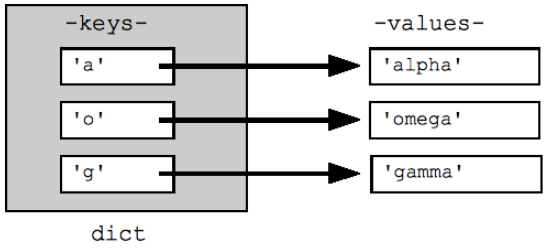
- Arrays are the most common and important composite data types
- Unlike records, which group related fields of disparate types, arrays are usually homogeneous
- Semantically, they can be thought of as a mapping from an **index type** to a **component** or **element type**
- A **slice or section** is a rectangular portion of an array (See figure 8.5)

Associative Arrays

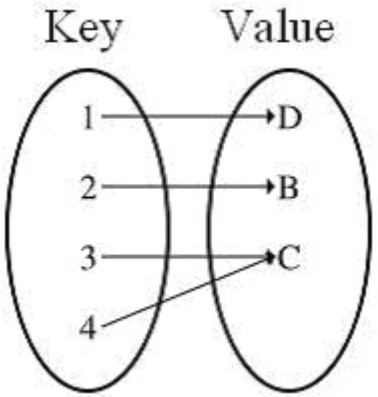
Associative Memory



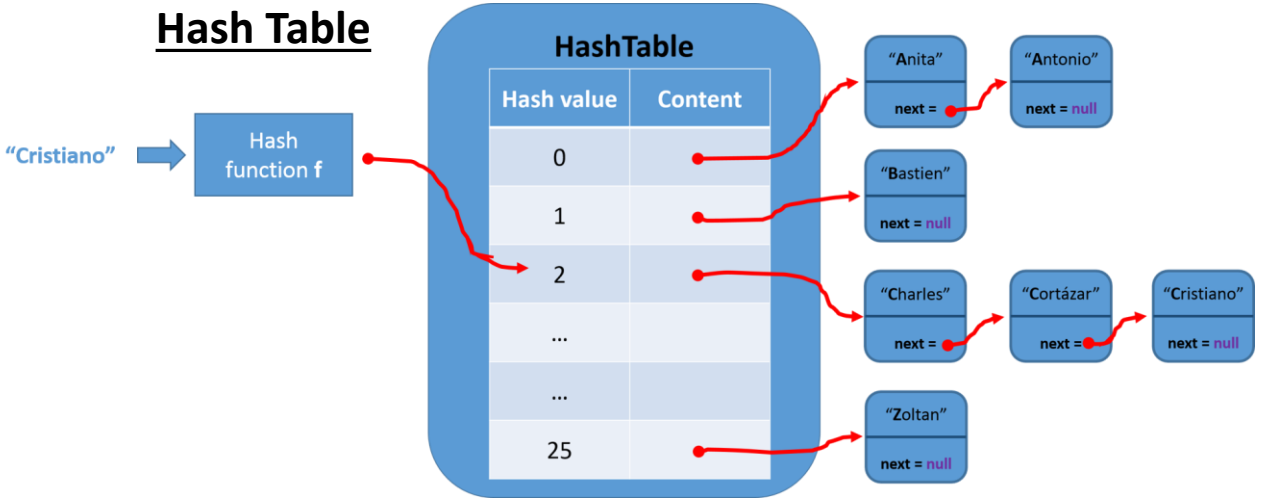
Python Dictionary



Java HashMap



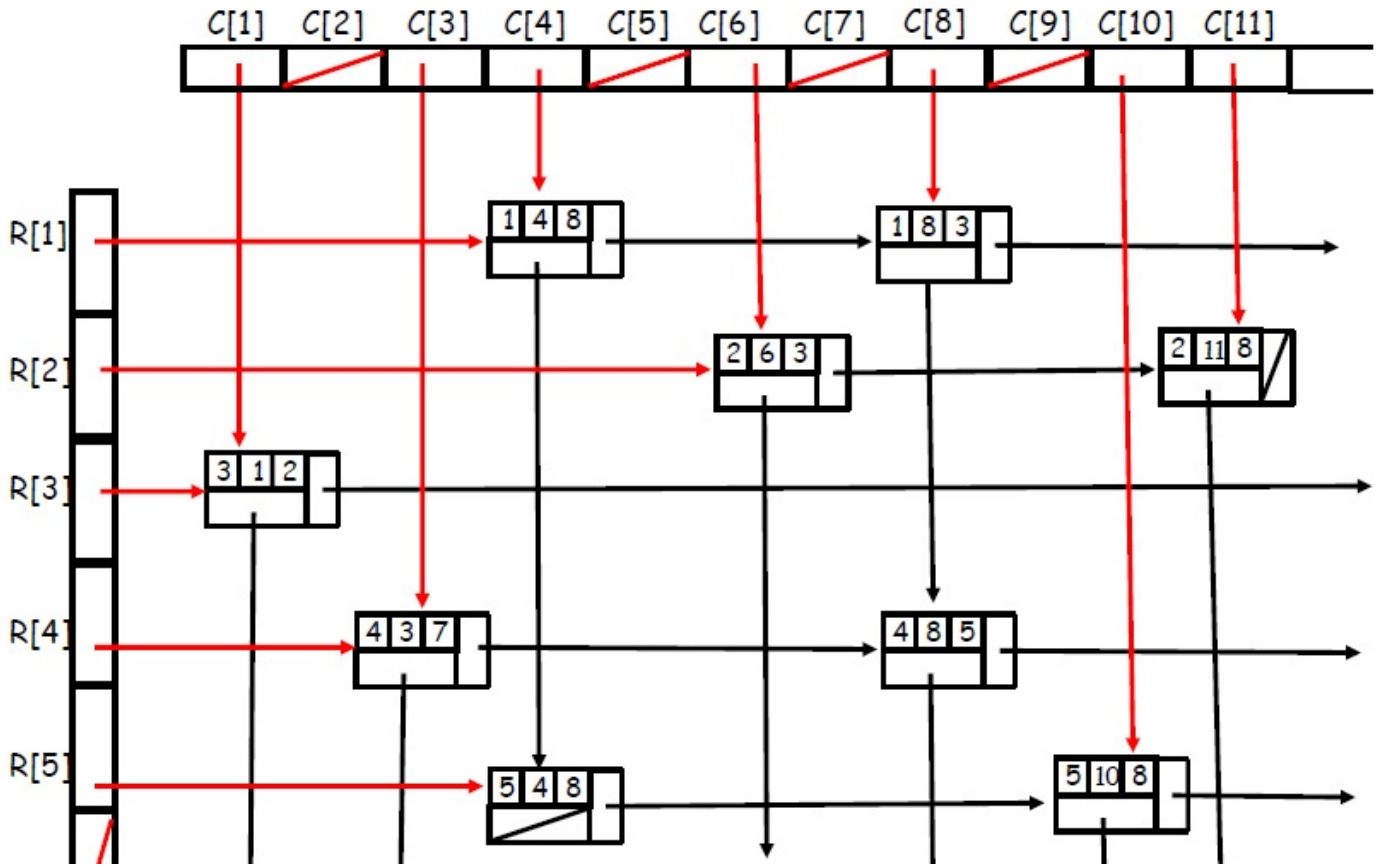
Hash Table



SQL Table (QUERY)

Sparse Matrix

C/C++ Constructed by Array and Nodes



Array Declarations

C:

```
char upper[26];
```

Fortran:

```
character, dimension (1:26) :: upper
character (26) upper    ! shorthand notation
```

Ada: (Use Array Constructor, Algo-descedents)

```
upper : array (character range 'a'..'z') of character;
```

Index:

- In C, the lower bound of an index range is always zero: the indices of an n-element array are 0 . . . n−1.
- In Fortran, the lower bound of the index range is one by default.

Multi-Dimensional Arrays

Ada:

```
mat : array (1..10, 1..10) of real;      -- Ada
```

Fortran:

```
real, dimension (10,10) :: mat           ! Fortran
```

Modula-3: 2-D => Array of Arrays

```
VAR mat : ARRAY [1..10], [1..10] OF REAL;
VAR mat : ARRAY [1..10] OF ARRAY [1..10] OF REAL;
```

Ada:

In Ada, by contrast,

```
mat1 : array (1..10, 1..10) of real;
```

is not the same as

```
type vector is array (integer range <>) of real;
type matrix is array (integer range <>) of vector (1..10);
mat2 : matrix (1..10);
```

Array of Arrays in C/C++, Java, C#

Figure: Showing jagged array.

```
int[][] jagArray = new int[5][];
```

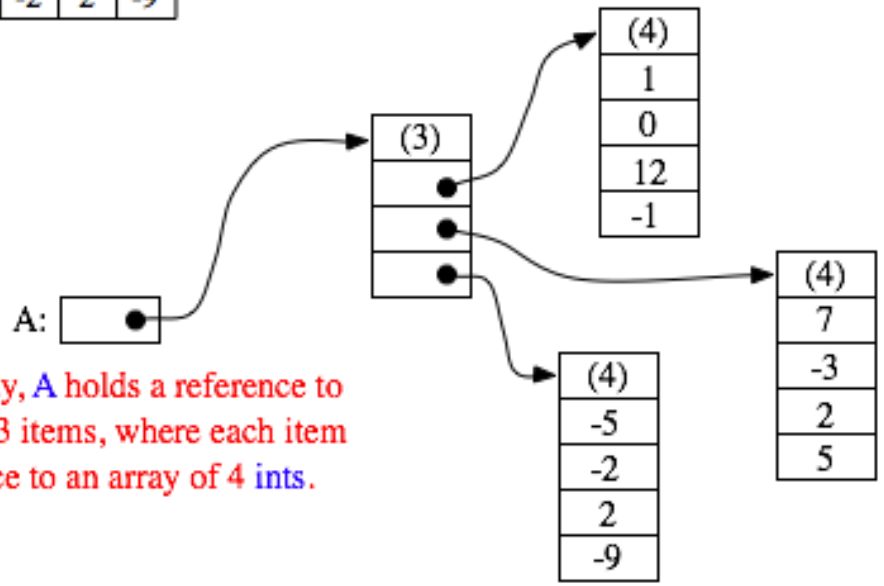
0	int[]
1	int[]
2	int[]
3	int[]
4	int[]

On each index of jagged array another array reference is stored.

If you create an array A = new int[3][4], you should think of it as a "matrix" with 3 rows and 4 columns.

A:

1	0	12	-1
7	-3	2	5
-5	-2	2	-9



But in reality, A holds a reference to an array of 3 items, where each item is a reference to an array of 4 ints.

Arrays II

Arrays in C-family

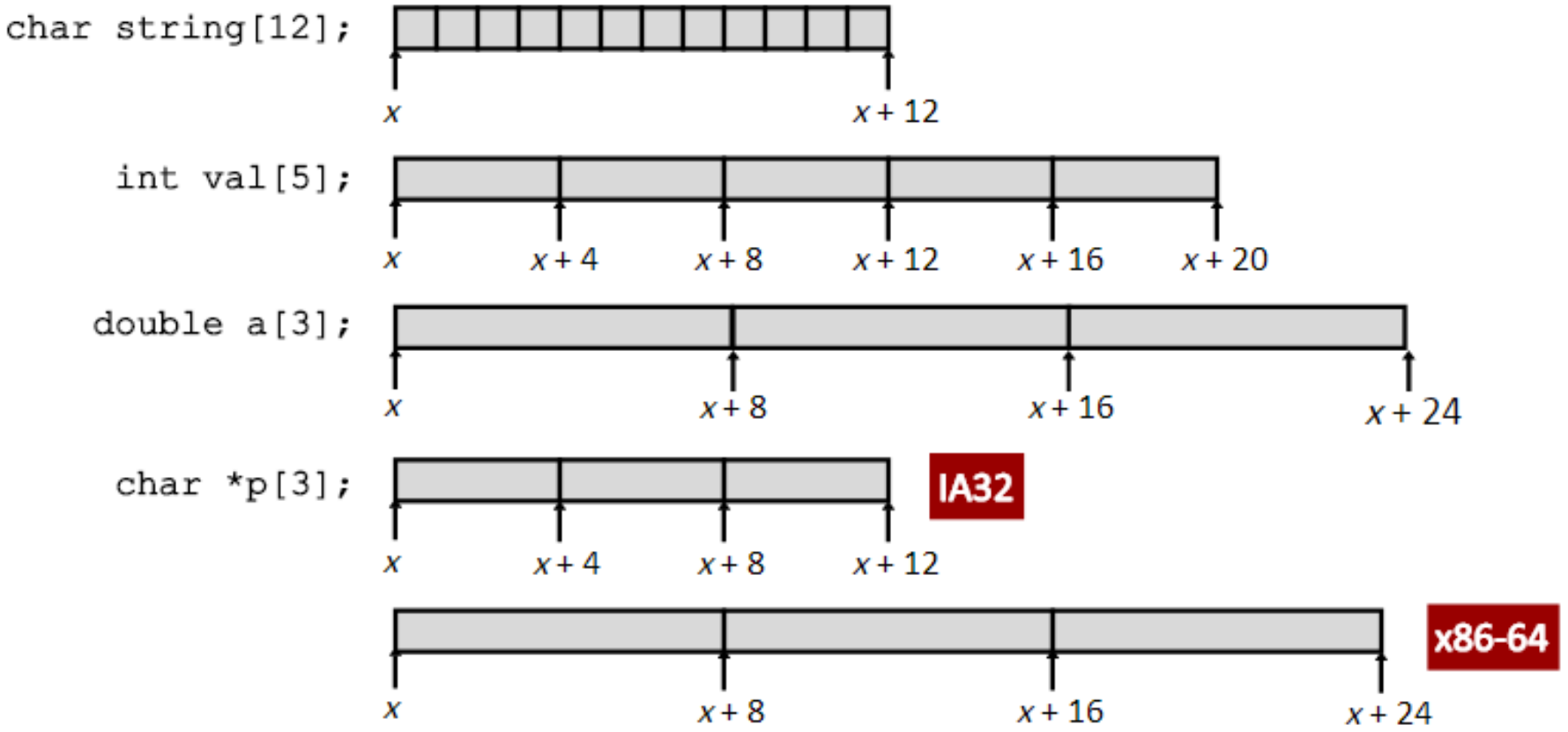
SECTION 4

Array Allocation

- **Basic Principle**

  T `A[`*N*`];`

  - Array of data type *T* and length *N*
  - Contiguously allocated region of *N* * `sizeof(`*T*`)` bytes

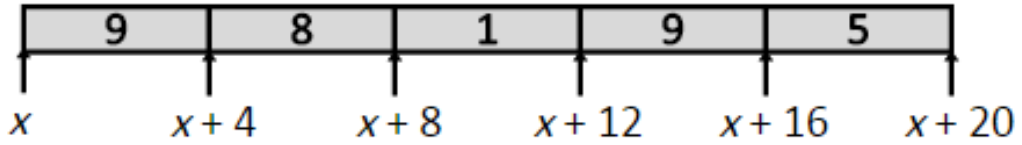


Basic Principle

T `A[`*N*`];`

- Array of data type *T* and length *N*
- Identifier `A` can be used as a pointer to array element 0: Type *T**

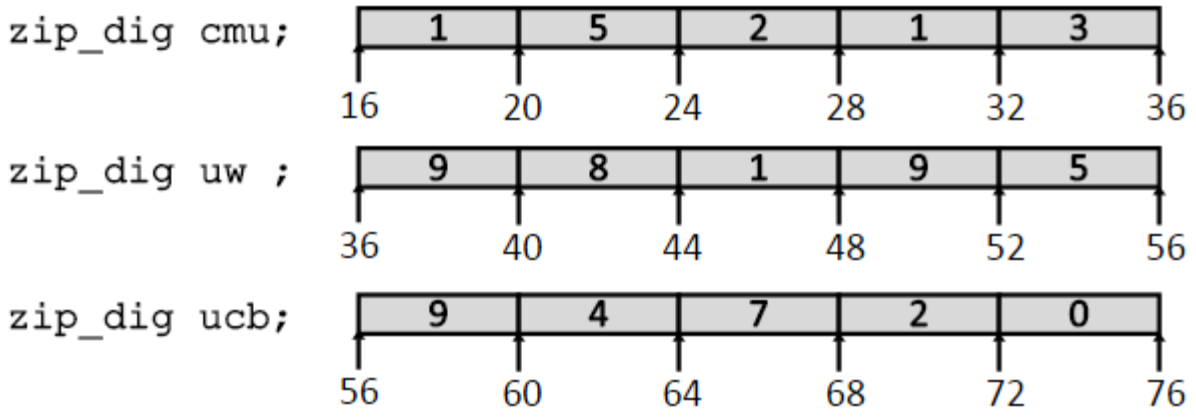
`int val[5];`



Reference	Type	Value
`val[4]`	`int`	5
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	8
`val + i`	`int *`	*x* + 4 *i*

Array Example

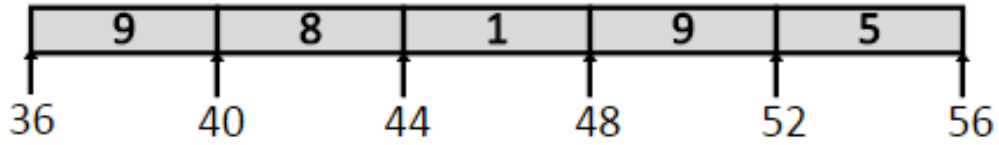
```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uw  = { 9, 8, 1, 9, 5 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



- Declaration "`zip_dig uw`" equivalent to "`int uw[5]`"
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

Array Accessing Example

```
zip_dig uw;
```



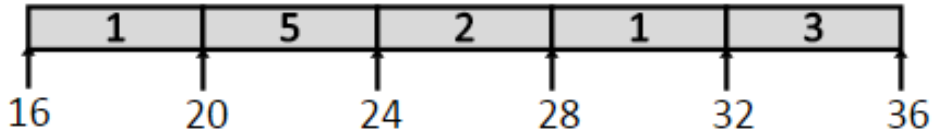

```
int get_digit
  (zip_dig z, int dig)
{
  return z[dig];
}
```

IA32

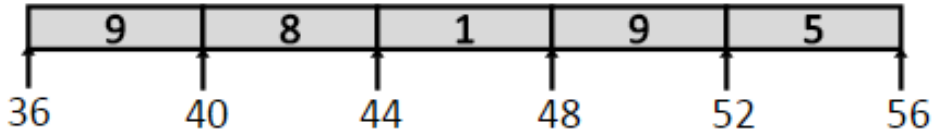
```
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax  # z[dig]
```

Referencing Examples

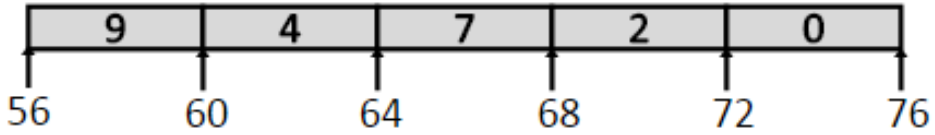
```
zip_dig cmu;
```




```
zip_dig uw ;
```




```
zip_dig ucb;
```



Reference	Address	Value	Guaranteed?
`uw[3]`	`36 + 4* 3 = 48`	`9`	Yes
`uw[6]`	`36 + 4* 6 = 60`	`4`	No
`uw[-1]`	`36 + 4*-1 = 32`	`3`	No
`cmu[15]`	`16 + 4*15 = 76`	`??`	No

- No bound checking
- Out-of-range behavior implementation-dependent
- No guaranteed relative allocation of different arrays

Index, *, & are just ways to calculate reference address

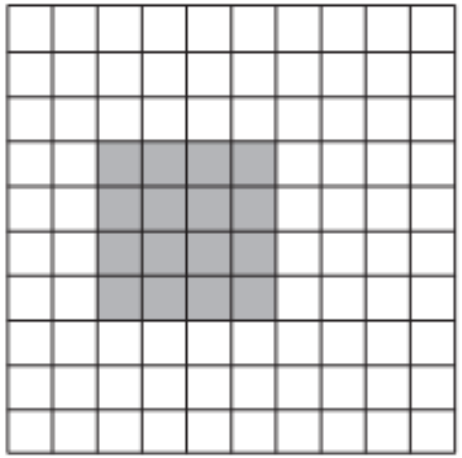
Is [] an operator?

Associative arrays in C++ are typically defined by overloading `operator[]`. C#, like C++, provides extensive facilities for operator overloading, but it does not use these facilities to support associative arrays. Instead, the language provides a special *indexer* mechanism, with its own unique syntax:

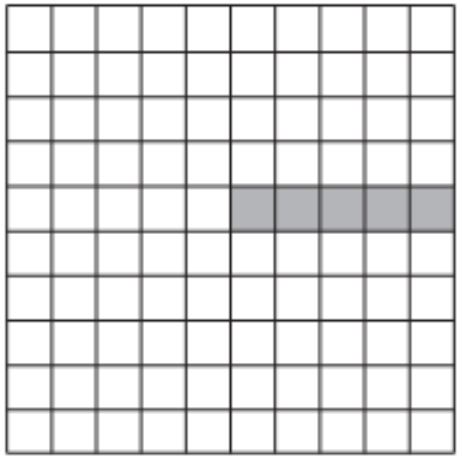
```
class directory {
    Hashtable table;                          // from standard library
    ...
    public directory() {                      // constructor
        table = new Hashtable();
    }
    ...
    public string this[string name] {         // indexer method
        get {
            return (string) table[name];
        }
        set {
            table[name] = value;              // value is implicitly
}   }   }                                     // a parameter of set
...
directory d = new directory();
...
d["Jane Doe"] = "234-5678";
Console.WriteLine(d["Jane Doe"]);
```

Why the difference? In C++, `operator[]` can return a reference (an explicit l-value—see Section 8.3.1), which can be used on either side of an assignment. C# has no comparable notion of reference, so it needs separate methods to `get` and `set` the value of `d["Jane Doe"]`.

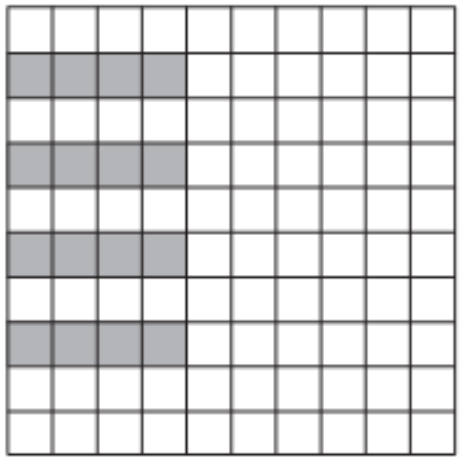
Arrays



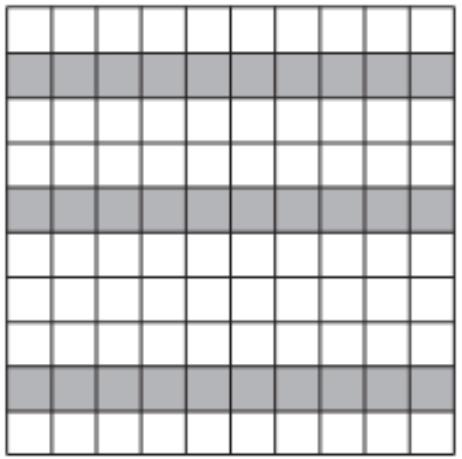
matrix(3:6, 4:7)



matrix(6:, 5)



matrix(:4, 2:8:2)



matrix(:, (/2, 5, 9/))

Slices and Array Operations

A *slice* or *section* is a rectangular portion of an array. Fortran 90 and Single Array slice operations Assignment C provide extensive facilities for slicing, as do many scripting languages, including Perl, Python, Ruby, and R.

Figure 8.5 Array slices(sections) in Fortran90. Much like the values in the header of an enumeration-controlled loop (Section6.5.1), a: b: c in a subscript indicates positions a, a+c, a+2c, ...through b. If a or b is omitted, the corresponding bound of the array is assumed. If c is omitted, 1 is assumed. It is even possible to use negative values of c in order to select positions in reverse order. The slashes in the second subscript of the lower right example delimit an explicit list of positions.

Arrays III

Call stack and Heap

SECTION 5

Arrays

Dimensions, Bounds, and Allocation (Size, Scope, Memory Allocation)

- **global lifetime, static shape —** If the shape of an array is known at compile time, and if the array can exist throughout the execution of the program, then the compiler can allocate space for the array in static global memory
- **local lifetime, static shape —** If the shape of the array is known at compile time, but the array should not exist throughout the execution of the program, then space can be allocated in the subroutine's stack frame at run time.
- **local lifetime, shape bound at elaboration time**

Arrays: Lifetime and Shape

- Lifetime (**Scope**: how long object exists) and shape (**Size**: dimensions and bounds), common options:
  - global lifetime, static shape
    - Pascal, C globals
  - local lifetime, static shape
    - Pascal, C locals
  - local lifetime, shape bound at elaboration
    - Ada locals
  - arbitrary lifetime, shape bound at elaboration
    - Java arrays
  - arbitrary lifetime, dynamic shape
    - Icon strings, APL, Perl arrays

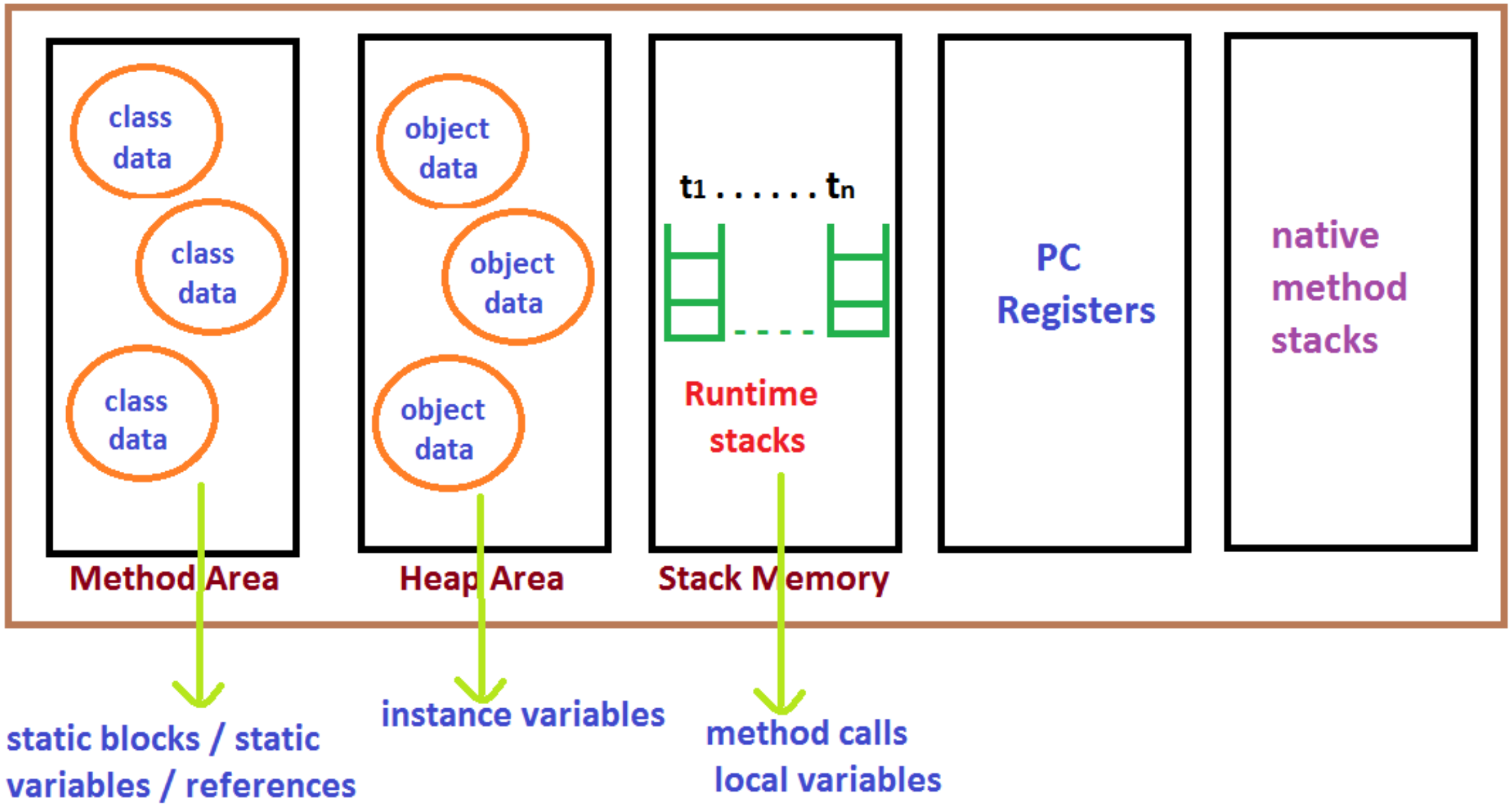
Local/Pointer: use stack
Instance: use heap (GC)
Static/Global: use memory pool

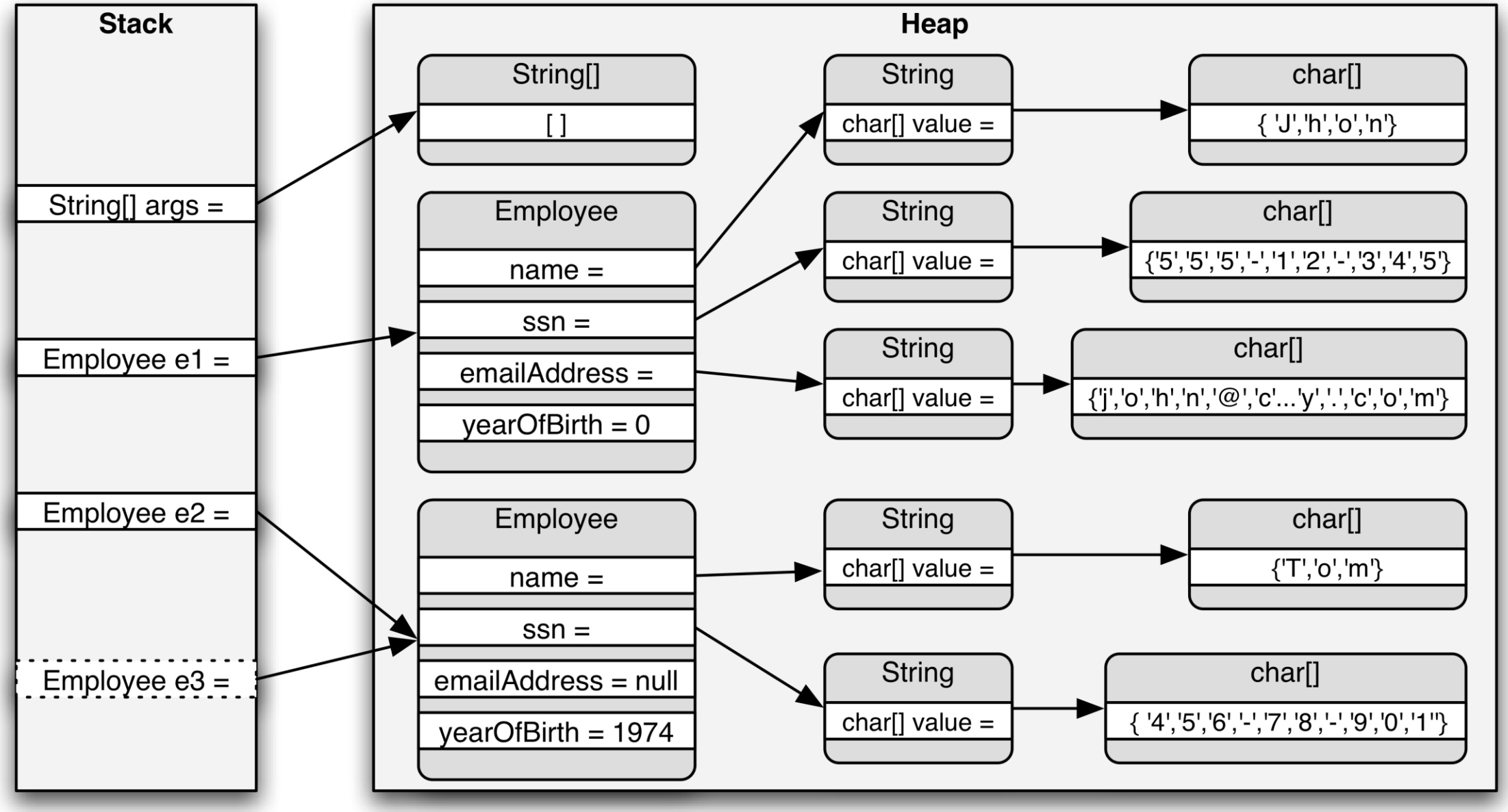
Dope Vectors

Descriptors, or Dope vectors, used to hold shape information at run time.

VO(Virtual Origin)
Lower bound for subscript 1
Upper bound for subscript 1
. . . .
Lower bound for subscript n
Upper bound for subscript n
Size of Component

- **Descriptors (dope vectors)** are required when array bounds are not known at compile time.
- **A dope vector** is usually stored next to the pointer to data.
- Typically, a dope vector for an array of dynamic scope will contain the lower bound for each dimension and the size of every dimension.
- Upper bounds are also need to support dynamics out-of-bounds checking
- When bounds are know the arithmetic can be done at compile time


class data
class data
class data
Method Area
object data
object data
object data
Heap Area
t1 . . . . . tn
Runtime stacks
Stack Memory
PC Registers
native method stacks
static blocks / static variables / references
instance variables
method calls
local variables


Stack
String[] args =
Employee e1 =
Employee e2 =
Employee e3 =
Heap
String[]
[ ]
Employee
name =
ssn =
emailAddress =
yearOfBirth = 0
Employee
name =
ssn =
emailAddress = null
yearOfBirth = 1974
String
char[] value =
String
char[] value =
String
char[] value =
String
char[] value =
String
char[] value =
char[]
{ 'J','h','o','n'}
char[]
{'5','5','5','-','1','2','-','3','4','5'}
char[]
{'j','o','h','n','@','c'...'y','.','c','o','m'}
char[]
{'T','o','m'}
char[]
{ '4','5','6','-','7','8','-','9','0','1''}

Arrays – Stack Allocation

```
void square(int n, double M[n][n]) {
    double T[n][n];
    for (int i = 0; i < n; i++) {          // compute product into T
        for (int j = 0; j < n; j++) {
            double s = 0;
            for (int k = 0; k < n; k++) {
                s += M[i][k] * M[k][j];
            }
            T[i][j] = s;
        }
    }
    for (int i = 0; i < n; i++) {          // copy T back into M
        for (int j = 0; j < n; j++) {
            M[i][j] = T[i][j];
        }
    }
}
```

Figure 8.6 A dynamic local array (conformant) in C. Function square multiplies a matrix by itself and replaces the original with the product. To do so it needs a scratch array of the same shape as the parameter. Note that the declarations of **M** and **T** both rely on parameter **n**.

Call Frame

(Dope Vector Included)

```
-- Ada:
procedure foo(size : integer) is
M : array (1..size, 1..size)
        of long_float;
...
begin
    ...
end foo;
```

```
// C99:
void foo(int size) {
    double M[size][size];
    ...
}
```

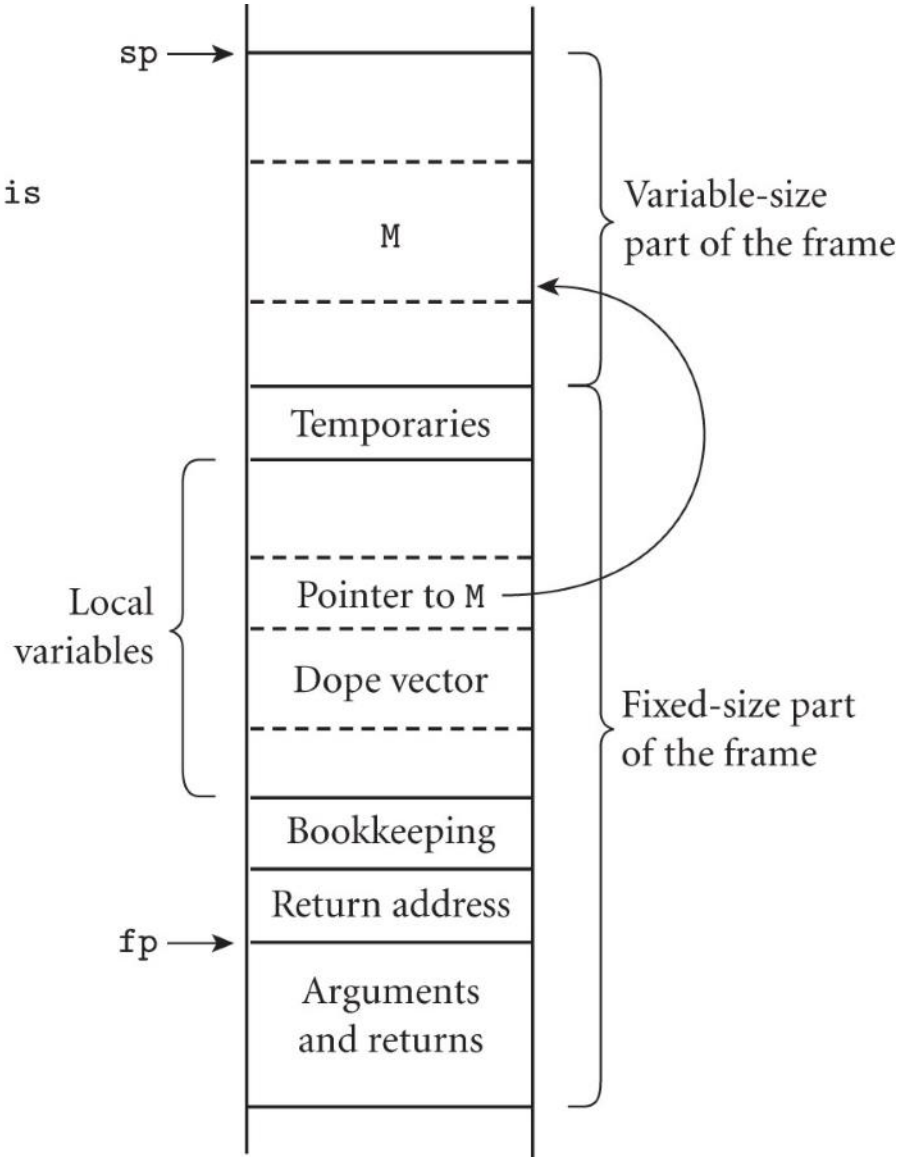
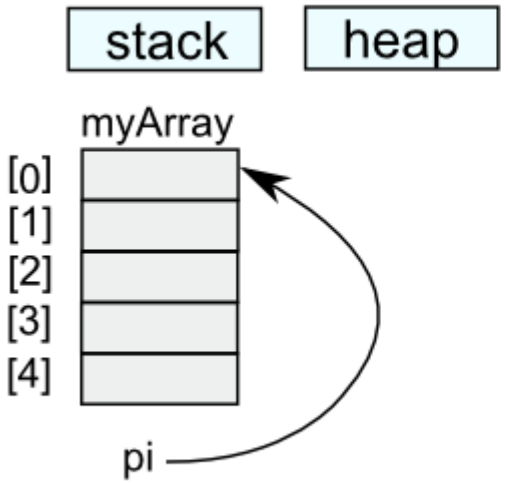
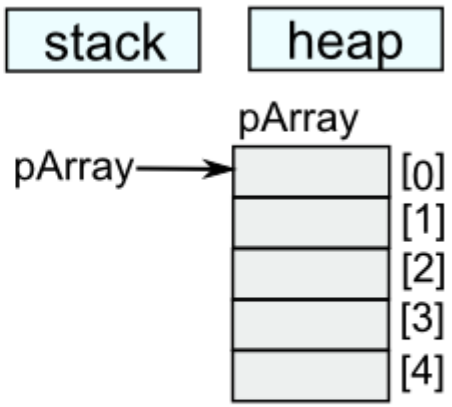


Figure 8.7 Elaboration-time allocation of arrays in Ada or C99. Here M is a square two-dimensional array whose bound are determined by parameter passed to foo at run time. The compiler arranges for a pointer to M and a dope vector to reside at static offsets from the frame pointer. M cannot b placed among the other local variables because it would prevent those higher in the frame from having static offsets. Additional variable-size arrays or records are easily accommodated.

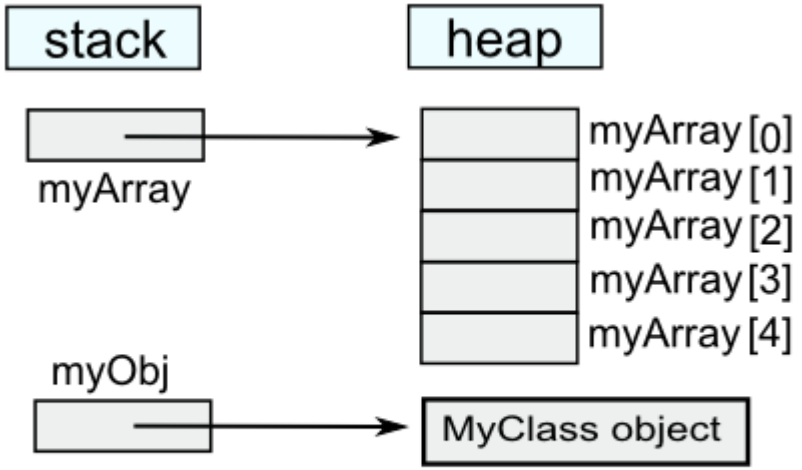
Heap Allocation C++



int myArray[5]



int *pArray = new int[5]



double *myArray = new double[5]
MyClass *myObj = new MyClass

Heap Allocation

- Arrays that can change shape at arbitrary times are sometimes said to be **fully dynamic**. Because changes in size do not in general occur in FIFO order, stack allocation will not suffice; fully dynamic arrays must be allocated in the heap.
- Several languages, including **Snobol**, **Icon**, and all the **scripting languages**, allow strings—arrays of characters—to change size after elaboration time. **Java** and **C#** provide a similar capability

```
String s = "short";    // This is Java; use lowercase 'string' in C#
...
s = s + " but sweet";  // + is the concatenation operator
```

Java String is immutable.

- Dynamically resizable arrays (other than strings) appear in APL, Common Lisp, and the various scripting languages. They are also supported by the vector, **Vector**, and **ArrayList** classes of the **C++**, **Java**, and **C#** libraries, respectively.
- If the number of dimensions of a fully dynamic array is statically known, the dope vector can be kept, together with a pointer to the data, in the stack frame of the subroutine in which the array was declared. If the number of dimensions can change, **the dope vector must generally be placed at the beginning of the heap block instead**.
- In the absence of garbage collection, the compiler must arrange to reclaim the space occupied by fully dynamic arrays when control returns from the subroutine in which they were declared. Space for stack-allocated arrays is of course reclaimed automatically by popping the stack.

Arrays IV

Layout and Address Calculation

SECTION 6

Arrays

Contiguous elements (see Figure 8.8)

- column major - only in Fortran
- row major
  - used by everybody else
  - makes array [a..b, c..d] the same as array [a..b] of array [c..d]

Arrays

- Arrays are usually stored in contiguous locations
- Two common orders

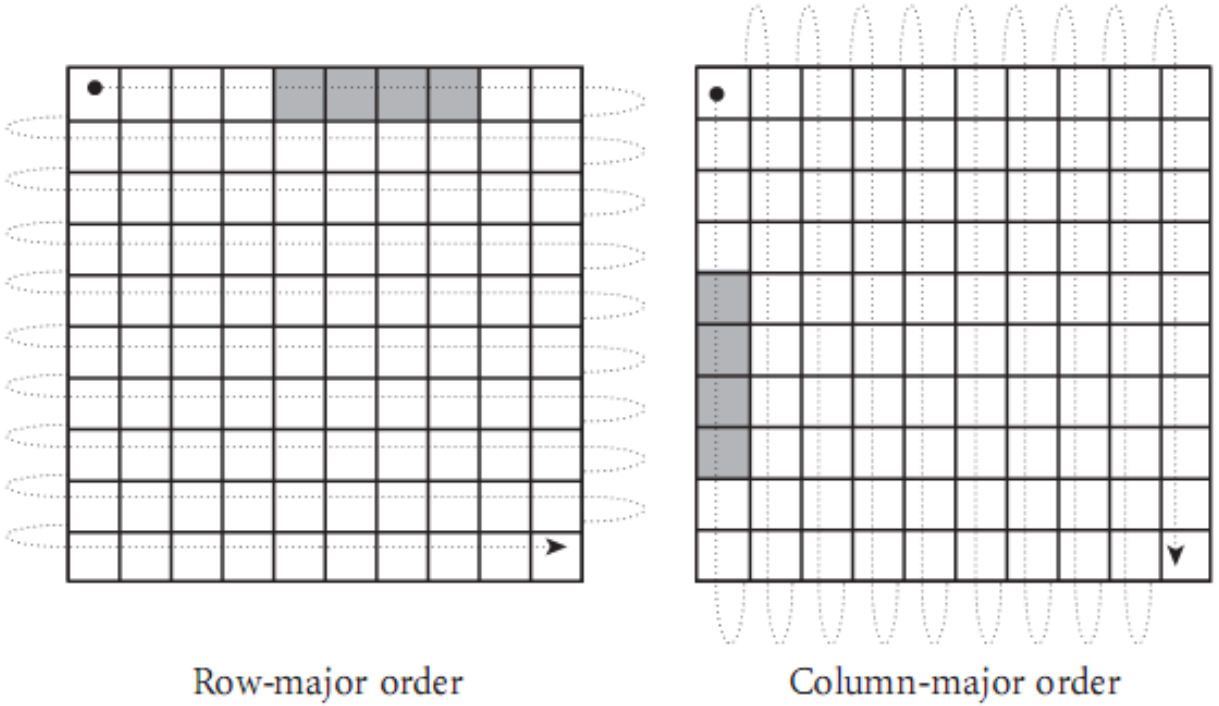


Figure 8.8 Row- and column-major memory layout for two-dimensional arrays. In row-major order, the elements of a row are contiguous in memory; in column-major order, the elements of a column are contiguous. The second cache line of each array is shaded, on the assumption that each element is an eight-byte floating-point number, that cache lines are 32 bytes long (a common size), and that the array begins at a cache line boundary. If the array is indexed from A[0,0] to A[9,9], then in the row-major case elements A[0,4] through A[0,7] share a cache line; in the column-major case elements A[4,0] through A[7,0] share a cache line.

Arrays

- Two layout strategies for arrays (Figure 8.9):
  - Contiguous elements
  - Row pointers
- Row pointers
  - an option in C
  - allows rows to be put anywhere - nice for big arrays on machines with segmentation problems
  - avoids multiplication
  - nice for matrices whose rows are of different lengths
    - e.g. an array of strings
  - requires extra space for the pointers

Arrays

Contiguous allocation vs. row pointers

```
char days[][10] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

```
char *days[] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

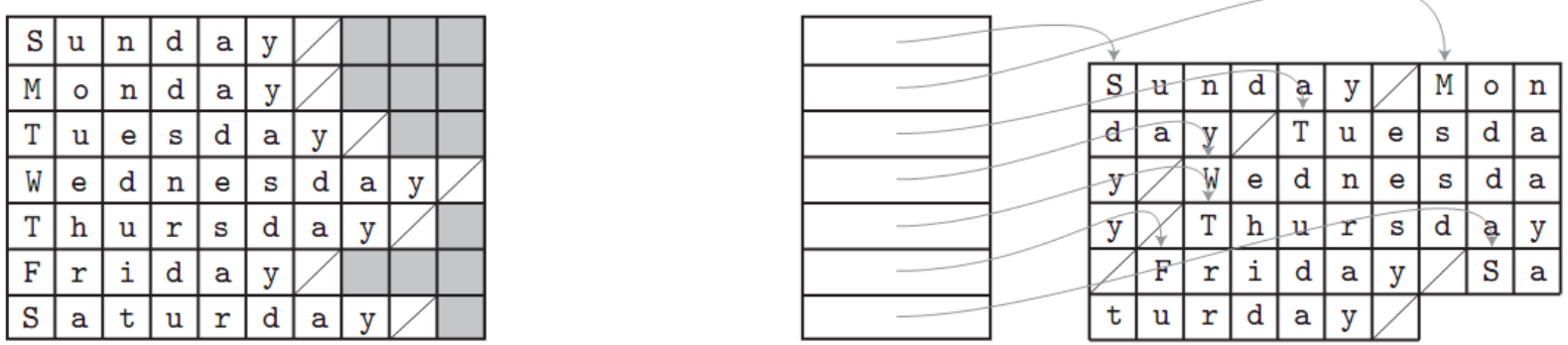


Figure 8.9 Contiguous array allocation v. row pointers in C. The declaration on the left is a true two-dimensional array. The slashed boxes are NUL bytes; the shaded areas are holes. The declaration on the right is a ragged array of pointers to arrays of character s. In both cases, we have omitted bounds in the declaration that can be deduced from the size of the initializer (aggregate). Both data structures permit individual characters to be accessed using double subscripts, but the memory layout (and corresponding address arithmetic) is quite different.

Arrays

Example 8.25 Indexing a contiguous array

Suppose

```
A : array [L1..U1] of array [L2..U2] of array
  [L3..U3] of elem;
  D1 = U1-L1+1
  D2 = U2-L2+1
  D3 = U3-L3+1
      Let
  S3 = size of elem
  S2 = D3 * S3
  S1 = D2 * S2
```

Arrays

Address Calculations

- Code generation for a 3D array
  - A: array $[L_1..U_1]$ of array $[L_2..U_2]$ of array $[L_3..U_3]$ of elem_type

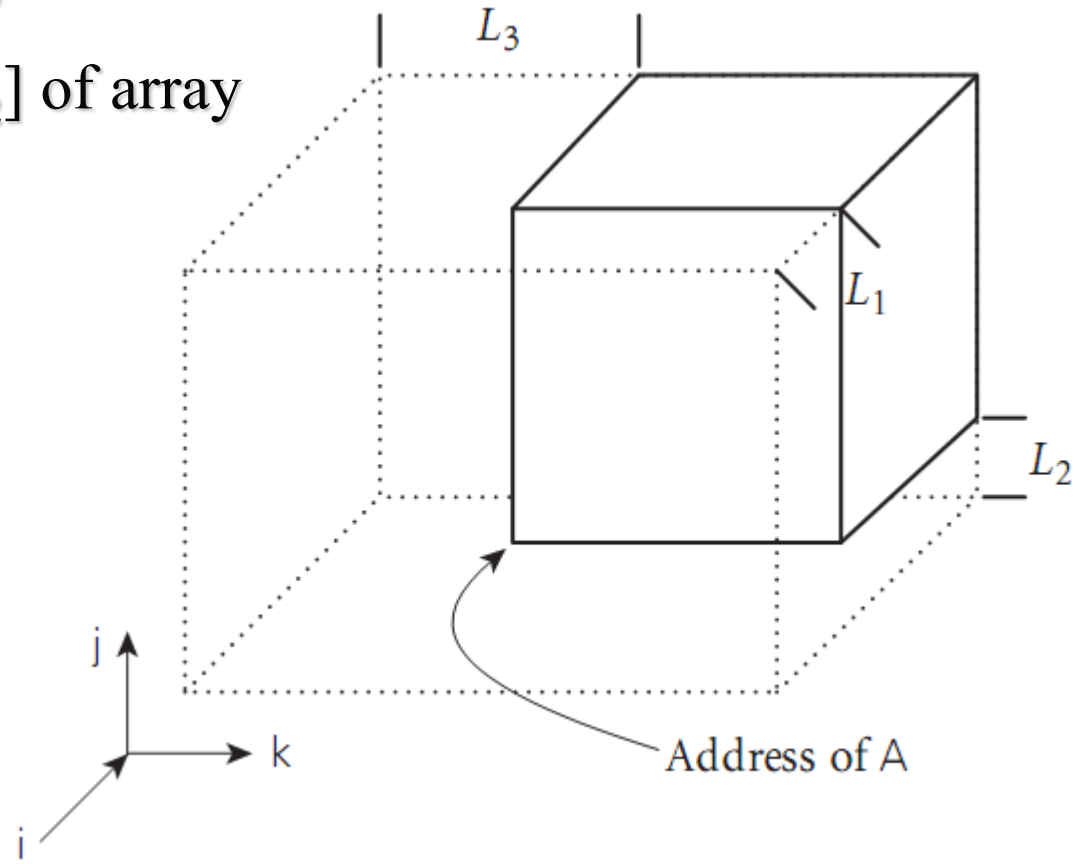


Figure 8.10 Virtual location of an array with nonzero lower bounds. By computing the constant portions of an array index at compile time, we effectively index into an array whose starting address is offset in memory, but whose lower bounds are all zero.

Arrays

Address Calculations

define

$$S_3 = \text{size of elem`type}$$
$$S_2 = (U_3 - L_3 + 1) \times S_3$$
$$S_1 = (U_2 - L_2 + 1) \times S_2$$

The address of A[i, j, k] is

$$\begin{aligned} &\text{address of A} \\ &+(i - L_1) \times S_1 \\ &+(j - L_2) \times S_2 \\ &+(k - L_3) \times S_3 \end{aligned}$$

Row-Major Order

Optimization

$$= (i \times S_1) + (j \times S_2) + (k \times S_3) + \text{address of A} - [(L_1 \times S_1) + (L_2 \times S_2) + (L_3 \times S_3)]$$

Compile-Time Constant

Arrays

Address Calculations

Instruction sequence to load A[i,j,k] into a register:

```
      -- assume i is in r1, j is in r2, and k is in r3
1:    r4 := r1 × S1
2:    r5 := r2 × S2
3:    r6 := &A − L1 × S1 − L2 × S2 − L3 × 4   -- one or two instructions
4:    r6 := r6 + r4
5:    r6 := r6 + r5
6:    r7 := *r6[r3]                           -- load
```

Strings

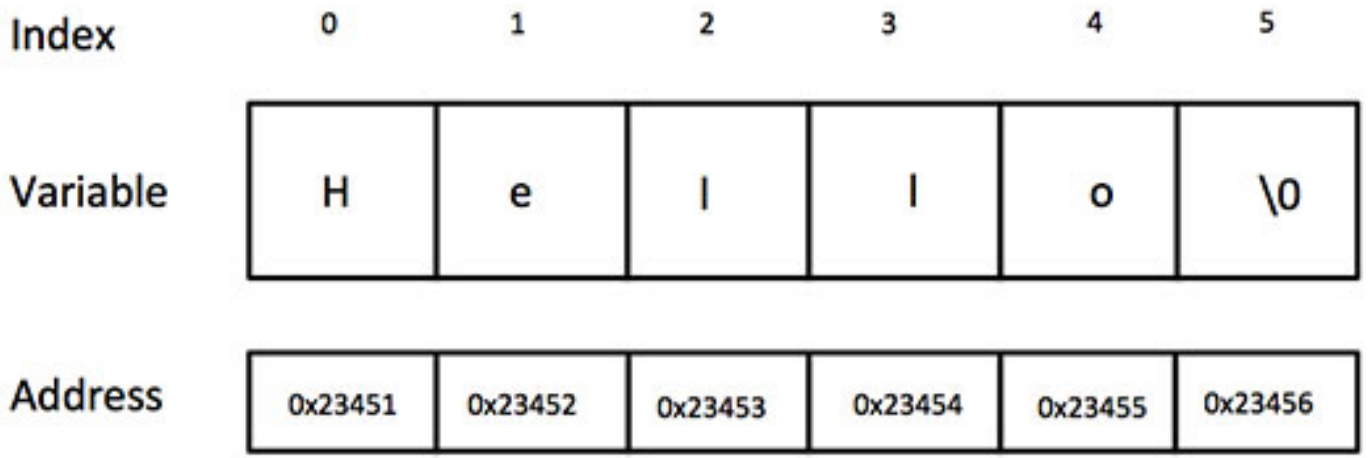
SECTION 7

Strings

- In some languages, strings are really just arrays of characters
- In others, they are often special-cased, to give them flexibility (like polymorphism or dynamic sizing) that is not available for arrays in general
  - It's easier to provide these things for strings than for arrays in general because strings are one-dimensional and (more importantly) non-circular

String as Array of Characters

C-language



std::string

```
#include <iostream>
#include <string>

std::string str("Hello!");
const char* p1 = "Hello!";
const char* p2 = p1 + 6;

std::string s1;
std::string s2("Hello!");
std::string s3("Hello!", 2);
std::string s4(5, 'H');
std::string s5(str.begin(), str.end());
std::string s6(p1, p2);
```

String in C++

- As an array // char charArray[10];
- As a char* // char *charString;
- As a string // stringTypeArray;

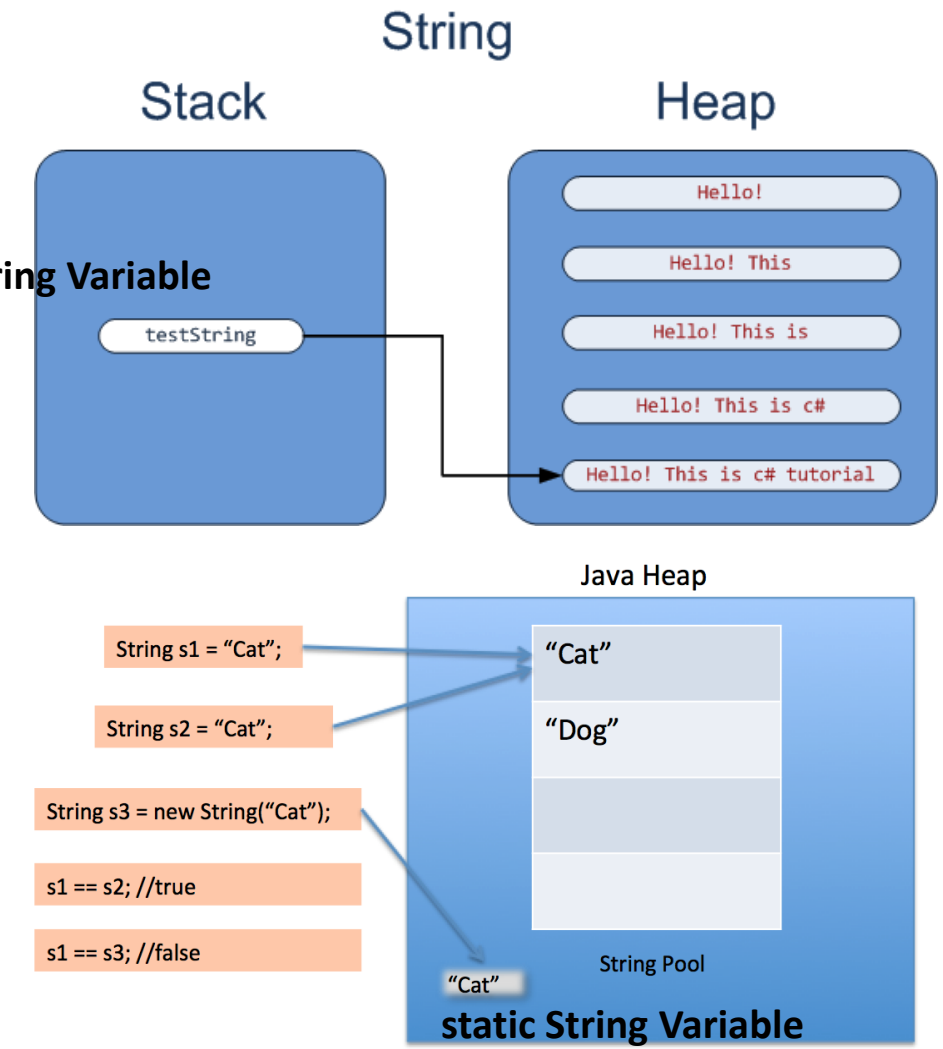
Constructor	Description
string()	creates an empty string object
string(const string s, int n, int n2)	creates a string and initializes it to contents starting at index n spanning n2 characters in s
string(const char *cs)	creates a string and initializes it to cs
string(int n, char c)	creates a string and initializes it to ns c
string(const char *cs, int n)	creates a string and initializes it to the first n elements in cs; if n is too big the string will contain garbage data
string(pt b, pt e)	initializes string object to the values of range [b, e); b and e can be pointers pointing to some element in a char array

String as Object

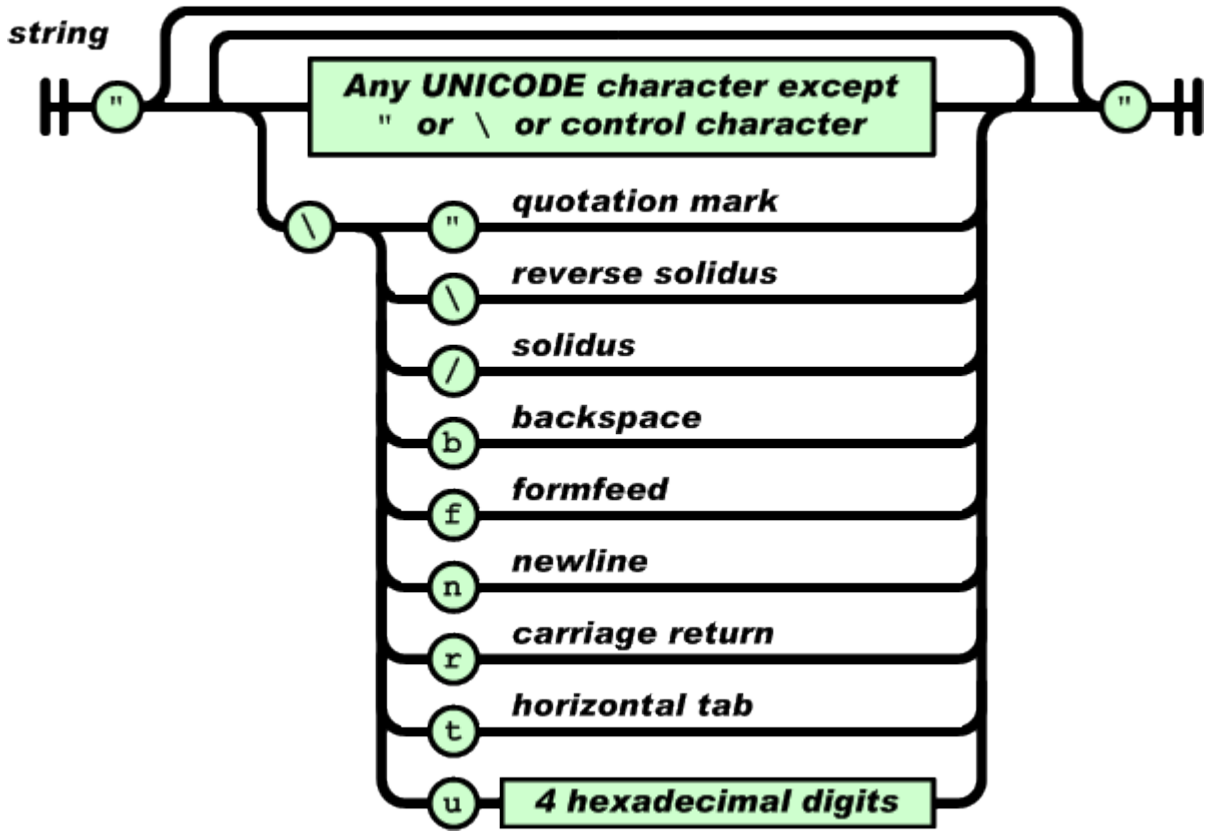
Java-Language

TYPE	NAME	VALUE	
int	number	1	Stored only Integer
int	sum	500500	Stored only Integer
double	radius	5.5	Stored only floating-point number
double	area	95.0334	Stored only floating-point number
String	greeting	Hello	Stored only texts
String	statusMsg	Game Over	Stored only texts

A *variable* has a ***name***, stores a ***value*** of the declared ***type***.



Java	C#	CLR Type
String	string	String
Object	object	Object
boolean	bool	Boolean
char	char	Char
short	short	Int16
int	int	Int32
long	long	Int64
double	double	Double
float	float	Single
byte	sbyte	SByte
—	byte	Byte
—	ushort	UInt16
—	uint	UInt32
—	ulong	UInt64



Escape Sequence

Escape Sequence	Function
\t	Tab (Unicode 0x0009)
\r	Carriage return (0x000d)
\n	Newline (line feed) (0x000a)
\v	Vertical tab (0x000b)
\a	Alert (0x0007)
\b	Backspace (0x0008)
\f	Form feed (0x000c)
\0	Null (0x0000)
\\	Backslash (0x005c)
\'	Single quotation mark (0x0027)
\"	Double quotation mark (0x0022)
\uABCD	Unicode character 0xABCD (where A, B, C, and D are valid hexadecimal digits 0-9, a-f, A-F)
\x0058	Hexadecimal character

Regular Expression in String

Expression	Description
`[013]`	A single digit 0, 1, or 3.
`[0-9][0-9]`	Any two-digit number from 00 to 99.
`[0-9&&[^4567]]`	A single digit that is 0, 1, 2, 3, 8, or 9.
`[a-z0-9]`	A single character that is either a lowercase letter or a digit.
`[a-zA-z][a-zA-Z0-9_$]*`	A valid Java identifier consisting of alphanumeric characters, underscores, and dollar signs, with the first character being an alphabet.
`[wb](ad|eed)`	Matches `wad`, `weed`, `bad`, and `beed`.
`(AZ|CA|CO)[0-9][0-9]`	Matches `AZxx`,`CAxx`, and `COxx`, where `x` is a single digit.

public class String (Java string data type)

	`String(String s)`	*create a string with the same value as* `s`
`int`	`length()`	*string length*
`char`	`charAt(int i)`	*i*`th character`
`String`	`substring(int i, int j)`	*i*`th through (`*j*`-1)st characters`
`boolean`	`contains(String sub)`	*does string contain* `sub` *as a substring?*
`boolean`	`startsWith(String pre)`	*does string start with* `pre`*?*
`boolean`	`endsWith(String post)`	*does string end with* `post`*?*
`int`	`indexOf(String p)`	*index of first occurrence of* `p`
`int`	`indexOf(String p, int i)`	*index of first occurrence of* `p` *after* `i`
`String`	`concat(String t)`	*this string with* `t` *appended*
`int`	`compareTo(String t)`	*string comparison*
`String`	`replaceAll(String a, String b)`	*result of changing* `a`*s to* `b`*s*
`String[]`	`split(String delim)`	*strings between occurrences of* `delim`
`boolean`	`equals(String t)`	*is this string's value the same as* `t`*'s?*

Sets

SECTION 8

Sets

Set is an unordered collection of an arbitrary number of distinct values of a common type.

Sets in Pascal:

- Set of <ordinal type>
  (enumeration type(char,Boolean),
  subrange type)
- Var S, T: set of 1..10;
- S := [1, 2, 3, 5, 7];
- T := [1 ..6];
- If T = [1, 2, 3, 5] then …

Pascal Sets Operators:

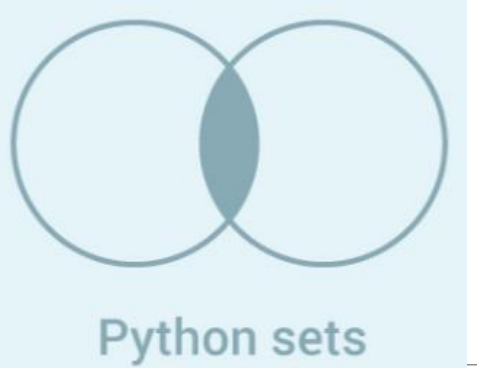
- =
- <>
- <= subset or equal
- >=
- But: no < or > !
- *(Intersection)
- +(Union)
- -(Difference)

Sets

We learned about a lot of possible implementations

- Bit sets are what usually get built into programming languages
- Things like intersection, union, membership, etc. can be implemented efficiently with bitwise logical instructions
- Some languages place limits on the sizes of sets to make it easier for the implementor
  - There is really no excuse for this

Python Sets




```
>>> SET={'new','old','list','new'}
>>> SET
set(['new', 'old', 'list'])
```

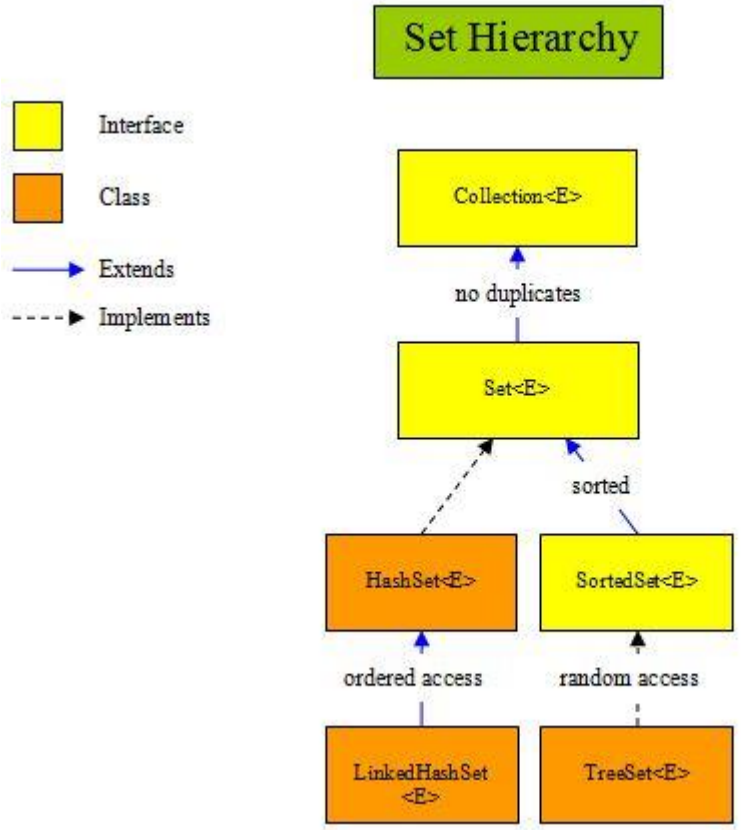
Operation	Equivalent	Result
len(s)		cardinality of set s
x in s		test x for membership in s
x not in s		test x for non-membership in s
s.issubset(t)	s <= t	test whether every element in s is in t
s.issuperset(t)	s >= t	test whether every element in t is in s
s.union(t)	s | t	new set with elements from both s and t
s.intersection(t)	s & t	new set with elements common to s and t
s.difference(t)	s - t	new set with elements in s but not in t
s.symmetric_difference(t)	s ^ t	new set with elements in either s or t but not both
s.copy()		new set with a shallow copy of s

Operation	Equivalent	Result
s.update(t)	s |= t	return set s with elements added from t
s.intersection_update(t)	s &= t	return set s keeping only elements also found in t
s.difference_update(t)	s -= t	return set s after removing elements found in t
s.symmetric_difference_update(t)	s ^= t	return set s with elements from s or t but not both
s.add(x)		add element x to set s
s.remove(x)		remove x from set s; raises KeyError if not present
s.discard(x)		removes x from set s if present
s.pop()		remove and return an arbitrary element from s; raises KeyError if empty
s.clear()		remove all elements from set s

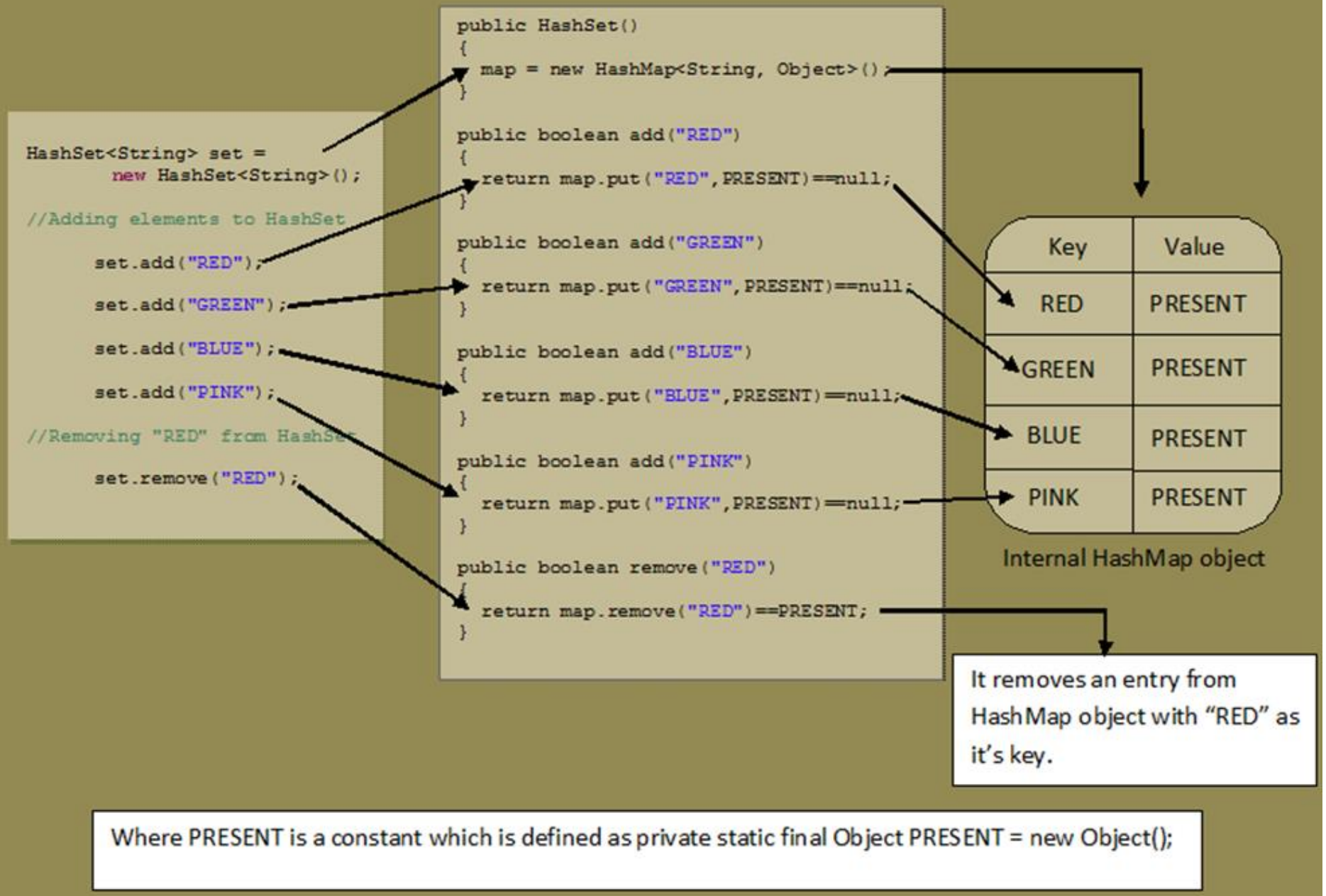
Python Set Operations and methods

Set in Java

Set
+add(element : Object) : boolean +addAll(collection : Collection) : boolean +clear() : void +contains(element : Object) : boolean +containsAll(collection : Collection) : boolean +equals(object : Object) : boolean +hashCode() : int +iterator() : Iterator +remove(element : Object) : boolean +removeAll(collection : Collection) : boolean +retainAll(collection : Collection) : boolean +size() : int +toArray() : Object[ ] +toArray(array : Object[ ]) : Object[ ]



Emulating Set Using Map In Java



Pointers And Recursive Types I

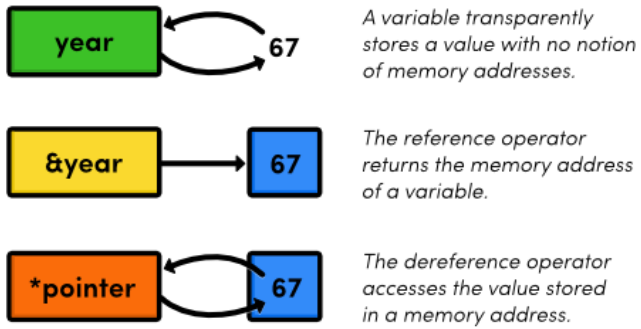
Basics and Reference Model

SECTION 9

Basic Operators for Pointer, Reference, Node and Recursive Type

- Address: location where data stored
- Pointer: variable that holds an address

```
int i = 10;
int *j = &i;
int k = 2 * (*j); /* dereference j */
```

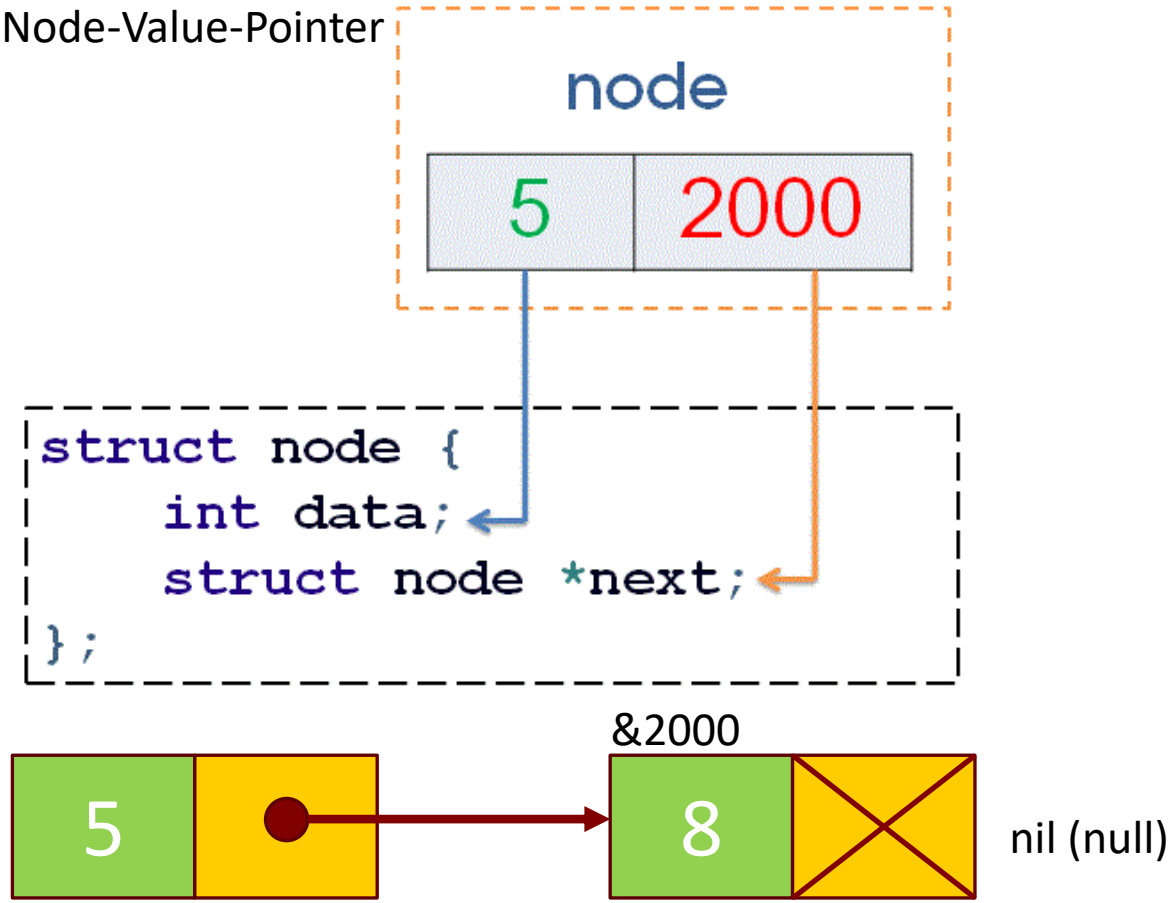


A variable transparently stores a value with no notion of memory addresses.

The reference operator returns the memory address of a variable.

The dereference operator accesses the value stored in a memory address.

- Node-Value-Pointer



Java Node Class

BlueJ: Terminal Window - Test

Options

```
5
8
```

```
public class Node
{   int value=0;
    Node next;
    Node(){}
    void setValue(int v){value = v;}
    int getValue() {return value;}
    void setNext(Node n){next = n;}
    Node getNext() {return next;}
}
```


```
public class TestNode
{   public static void main(String[] args){
        Node a = new Node();
        a.setValue(5);
        Node b = new Node();
        b.setValue(8);
        a.setNext(b);
        b.setNext(null);
        Node head = a;
        while (head != null){
            System.out.println(head.getValue());
            head = head.getNext();
        }
    }
}
```

Note:

Java have only reference type (actually is pointer type) but no address type data. Use pointer to complete the Whole tree/graph type projects.

Pointers And Recursive Types

- Pointers serve two purposes:
  - efficient (and sometimes intuitive) access to elaborated objects (as in C)
  - dynamic creation of linked data structures, in conjunction with a heap storage manager
- Several languages (e.g. Pascal, Ada 83) restrict pointers to accessing things in the heap
- Pointers are used with a value model of variables
  - They aren't needed with a reference model

Syntax and Operations

Content (Body)

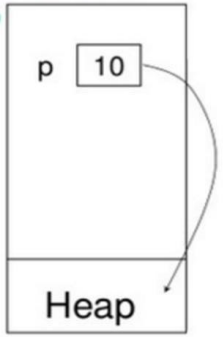
- Operations on pointers include:
  - allocation and deallocation of objects in the heap,
  - dereferencing of pointers to access the objects to which they point, and
  - assignment of one pointer into another.

- The behavior of these operations depends heavily on whether the language is functional or imperative, and on whether it employs a reference or value model for variables/names.

THE C PROGRAMMING LANGUAGE

Dynamic Allocation with malloc:

```
{
int *p;
p = (int *) malloc (sizeof (int));
scanf("%d", p);
printf("%d", *p);
printf("%d" &p);
}
```



```
class Super {
    String s;

    public Super(String s) {
        this.s = s;
    }
}

public class Sub extends Super {
    int x = 200;
    public Sub(String s) {

    }

    public Sub(){
        System.out.println("Sub");
    }

    public static void main(String[] args){
        Sub s = new Sub();
    }
}
```

C# Programming

```
class Program
{
  string name;
  int age;

  public void setdata()
  {
   name = "Ram";
   age = 20;
  }
  public void showdata()
  {
Console.WriteLine(name+" "+age);
  }
}
```

Member Variables

Member Functions

```
class mainClass
  {
 public static void Main()
   {
    Program obj;
     obj = new Program();
     obj.setdata();
     obj.showdata();
     Console.ReadLine();
    }
  }
```

Reference

Ram 20


```
#include <iostream>

class CBool
{
public:
    bool miBoolean;
};

int main() {
    CBool qMyBool;
    CBool qCopy(qMyBool);
    CBool qInit = {true};
    qCopy = qInit;
    return 0;
}
```

```
class Vector{
    int sz;
    double *p;
public:
    Vector(int n = 3);
    ~Vector( );
};

Vector::Vector(int n){
    p = new double[n];
    assert(p != 0);
    sz = n;
}

Vector::~Vector( ){
    delete[ ] p;
}
```

Reference Model

- In ML-family, the variants mechanism can be used to declare recursive types:

```
datatype chr_tree = empty | node of char * chr_tree * chr_tree;
```

- **`chr_tree`** is either an Empty leaf or a Node consisting of a character and two child trees.
- The tree node (#"R", node (#"X", empty, empty), node (#"Y", node (#"Z", empty, empty), node (#"W", empty, empty))) would most likely be represented in memory as shown in Figure 8.11. **[OCaml]**
- **`'(#\R (#\X () ()) (#\Y (#\Z () ()) (#\W () ())))`** Each level of parentheses brackets the elements of a list. (The prefix #\ notation serves the same purpose as surrounding quotes in other languages.) **[Lisp]**

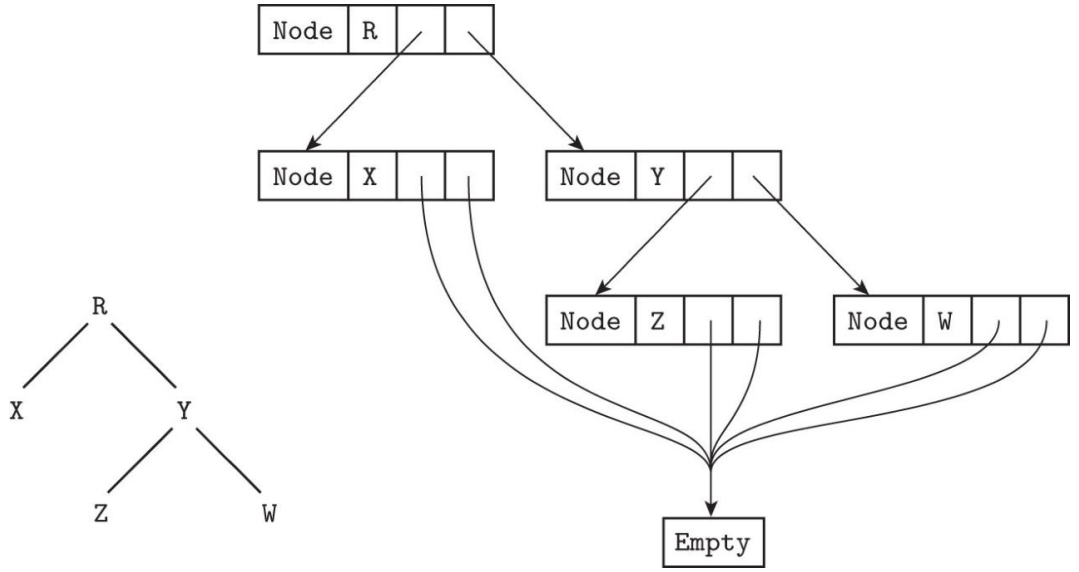


Figure 8.11 Implementation of a tree in ML. The abstract (conceptual) tree is shown at the lower left.

Pointers And Recursive Types

Lisp Binary Tree

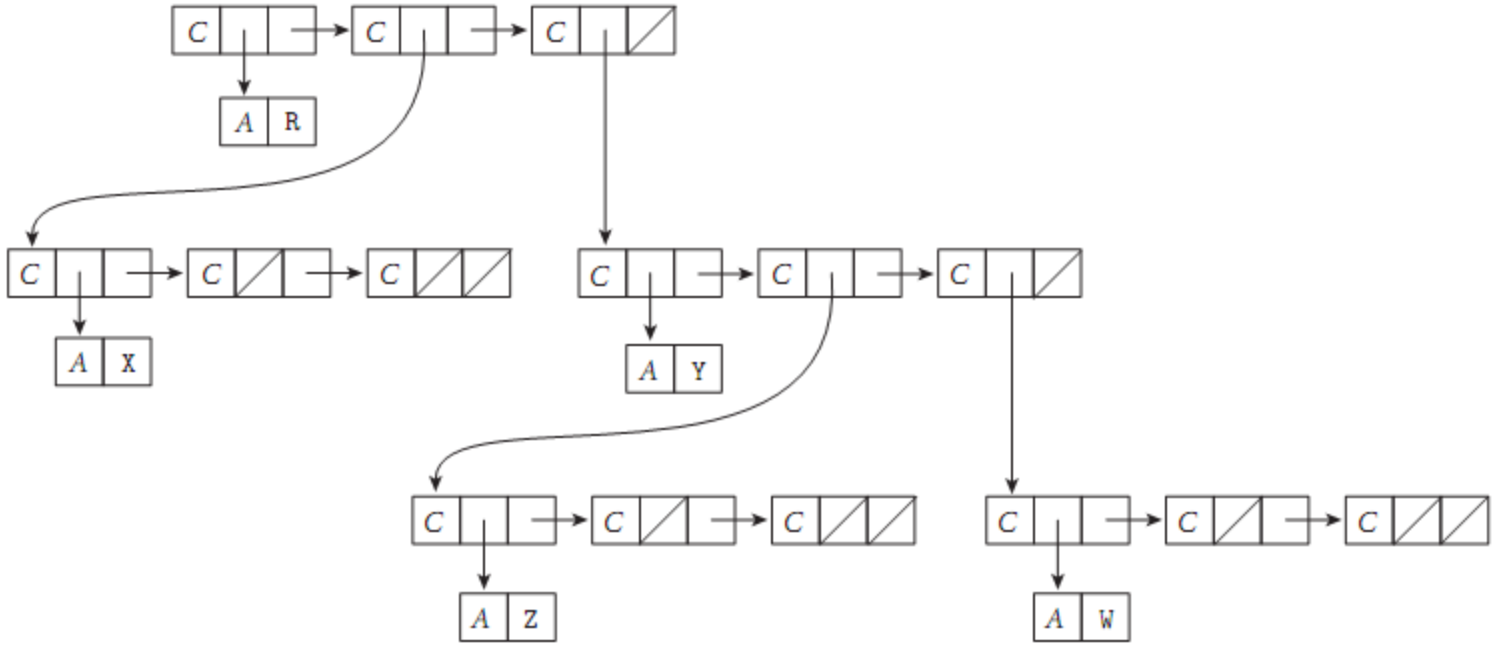


Figure 8.12 Implementation of a tree in Lisp. A diagonal slash through a box indicates a null pointer. The *C* and *A* tags serve to distinguish the two kinds of memory blocks: cons cells and blocks containing atoms.

Pointers And Recursive Types

Mutually Recursive Types in OCamel

- In a compiler, for example, it is likely that **symbol table records** and **syntax tree nodes** will need to refer to each other.
- A syntax tree node that represents a subroutine call will need to refer to the symbol table record that represents the subroutine.
- The symbol table record, for its part, will need to refer to the syntax tree node at the root of the subtree that represents the subroutine's code.
- If types are declared one at a time, and if names must be declared before they can be used, then whichever mutually recursive type is declared first will be unable to refer to the other.

- ML addresses this problem by allowing type declared together in as a group. Using **OCaml** syntax:

```
type subroutine_info = {code: syn_tree_node; …} (* record *)
 and subr_call_info   = {subr: sym_tab_rec; …}     (* record *)
 and sym_tab_rec =                                  (* variant *)
       Variable of …
   | Type of …
   |  …
   | subroutine of subroutine_info
 and syn_tree_node =
     Expression of …
   | Loop of …
   | …
   | Subr_call of subr_call_info;;
```

Pointers And Recursive Types II

Value Model

SECTION 10

Value Model

Tree Type in Ada and C

Ada: (Node Declaration)

```
type chr_tree;
type chr_tree_ptr is access chr_tree;
type chr_tree is record
    left, right : chr_tree_ptr;
    val : character;
end record;
```

C: (Node Declaration)

```
struct chr_tree {
    struct chr_tree *left, *right;
    char val;
};
```

Both Ada and C rely on incomplete type declarations for recursive definition.

Allocating Heap Nodes:

In Ada:

```
my_ptr := new chr_tree;
```

In C:

```
my_ptr = malloc(sizeof(struct chr_tree));
```

C++, Java, and C# replace `malloc` with a built-in, type-safe `new`:

```
my_ptr = new chr_tree( arg_list );
```

C's malloc is defined as a library function, not a built-in part of the language. The programmer must specify the size of the allocated object explicitly, and while the return value (of type void*) can be assigned into any pointer, the assignment is not type-safe.

Pointers And Recursive Types

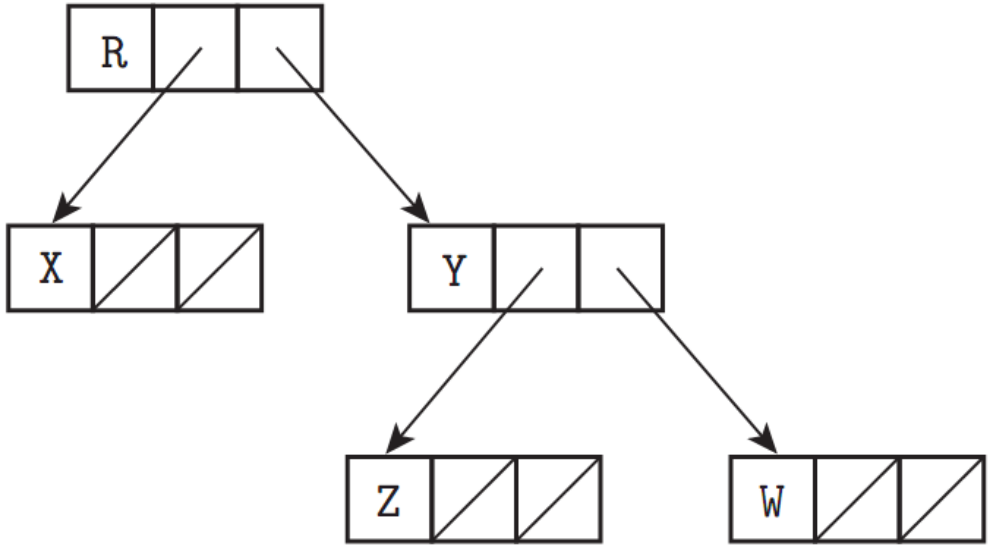


Figure 8.13 Typical implementation of a tree in a language with explicit pointers. As in Figure 8.12, a diagonal slash through a box indicates a null pointer.

Pointer Dereferencing

Pointer dereferencing (find the body of a pointer) :

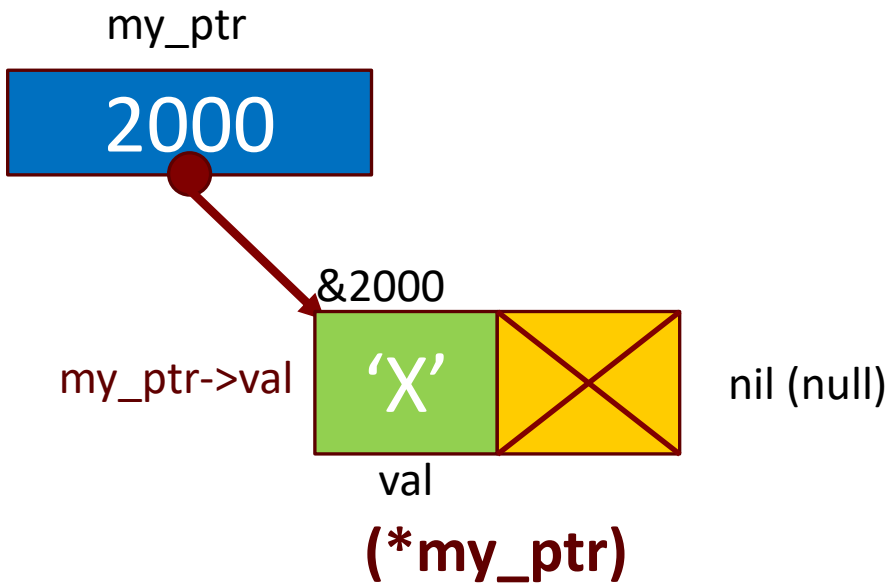
To access the object referred to by a pointer, most languages use an explicit dereferencing operator. In Pascal and Modula this operator takes the form of a postfix “up-arrow”:

Pascal/Modula-3:

```
my_ptr^.val := 'X';
```

C:

```
(*my_ptr).val = 'X';
```



C: (dereferencing to: value that the pointer referenced to)

```
my_ptr->val = 'X';
```

Implicit dereferencing in Ada

- On the assumption that EXAMPLE 8.40 pointers almost always refer to records, Ada dispenses with dereferencing altogether.
- The same dot-based syntax can be used to access either a field of the record foo or a field of the record pointed to by foo, depending on the type of foo:

```
T : chr_tree;
P : chr_tree_ptr;
...
T.val := 'X';
P.val := 'Y';
```

all for entire body:

```
T := P.all;
```

Java's Reference Type is Pointer with Implicit Dereferencing

Demo Program: PrintHashCode.java

- In Java language, there is no such terminology as pointer.
- There is only one data type Reference. But the reference type is actually a pointer (in the heap). If you print the pointer, it will show the Heap Hash Code if there is no .toString() overriding method.

Example:

Using the same Node as the previous lecture.

1. head is reference type to Node.
2. If .toString() is defined, to print head will print out head's body's toString() return value.
3. If .toString() is not defined, the Class@hashCode will be printed.
4. head.value actually means head->value in C. Java has the implicit dereferencing (auto-dereferencing)
5. head is a pointer. It stores the address of the object (body) in the heap.
6. Java's pointer is called **Reference** Type.

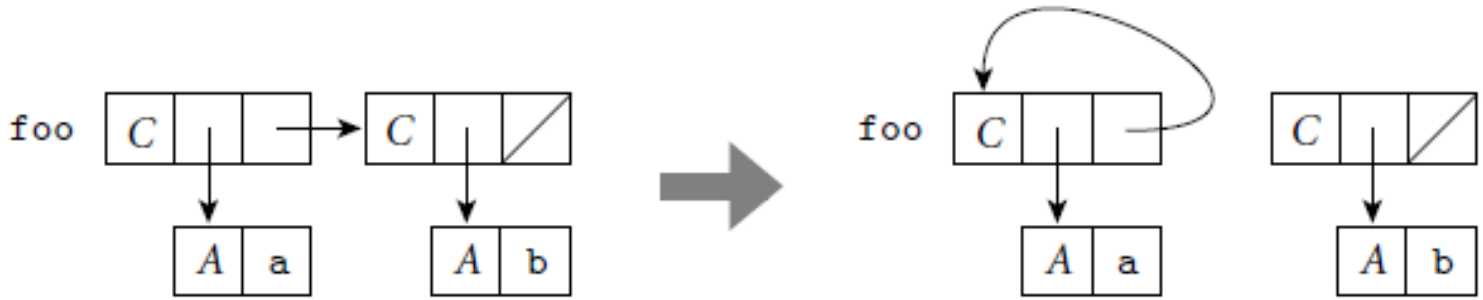
```
8  public class PrintHashCode
9  {   public static void main(String[] args){
10         Node a = new Node();
11         a.setValue(5);
12         Node b = new Node();
13         b.setValue(8);
14         a.setNext(b);
15         b.setNext(null);
16         Node head = a;
17
18         System.out.println("Class@hashCode    = "+head);
19         System.out.println("Decimal hashCode = "+head.hashCode());
20     }
21 }
```

```
BlueJ: Terminal Window - chapter8
Options
Class@hashCode    = Node@1e16b20
Decimal hashCode = 31550240
```

Pointers And Recursive Types

Assignment in Lisp

- The imperative features of **Lisp** do not include a dereferencing operator. Since every object has a self-evident type, and assignment is performed using a small set of built-in operators, there is never any ambiguity as to what is intended.
- Assignment in Common Lisp employs the **setf** operator (Scheme uses **set!**, **set-car!**, and **set-cdr!**), rather than the more common **:=**. For example, if **foo** refers to a list, then **(cdr foo)** is the right-hand **("rest of list")** pointer of the first node in the list, and the assignment **(set-cdr! foo foo)** makes this pointer refer back to **foo**, creating a one-node circular list:



C Pointers And Array Reference

- C pointers and arrays

```
int *a == int a[]
int **a == int *a[]
```

- BUT equivalences don't always hold
  - Specifically, a declaration allocates an array if it specifies a size for the first dimension
  - otherwise it allocates a pointer

int **a, int *a[] pointer to pointer to int

int *a[n], n-element array of row pointers

int a[n][m], 2-d array

C Pointers And Array Reference

- Compiler has to be able to tell the size of the things to which you point
  - So the following aren't valid:

```
int a[][]          bad
int (*a)[]
```

  - C declaration rule: read right as far as you can (subject to parentheses), then left, then out a level and repeat

`int *a[n],` n-element array of pointers to integer

`int (*a)[n],` pointer to n-element array of integers

C Pointers And Array Reference

```
int n;
int *a;          /* pointer to integer */
int b[10];       /* array of 10 integers */
```

Array Reference in integer Type (address calculation in the unit of integer (4 bytes))

The third integer array element

The body of a pointer + 3 integer away location

```
1.  a = b;          /* make a point to the initial element of b */
2.  n = a[3];
3.  n = *(a+3);     /* equivalent to previous line */
4.  n = b[3];
5.  n = *(b+3);     /* equivalent to previous line */
```

DESIGN & IMPLEMENTATION

Pointers and arrays

Many C programs use pointers instead of subscripts to iterate over the elements of arrays. Before the development of modern optimizing compilers, pointer-based array traversal often served to eliminate redundant address calculations, thereby leading to faster code. With modern compilers, however, the opposite may be true: redundant address calculations can be identified as common subexpressions, and certain other code improvements are easier for indices than they are for pointers. In particular, as we shall see in Chapter 16, pointers make it significantly more difficult for the code improver to determine when two l-values may be *aliases* for one other.

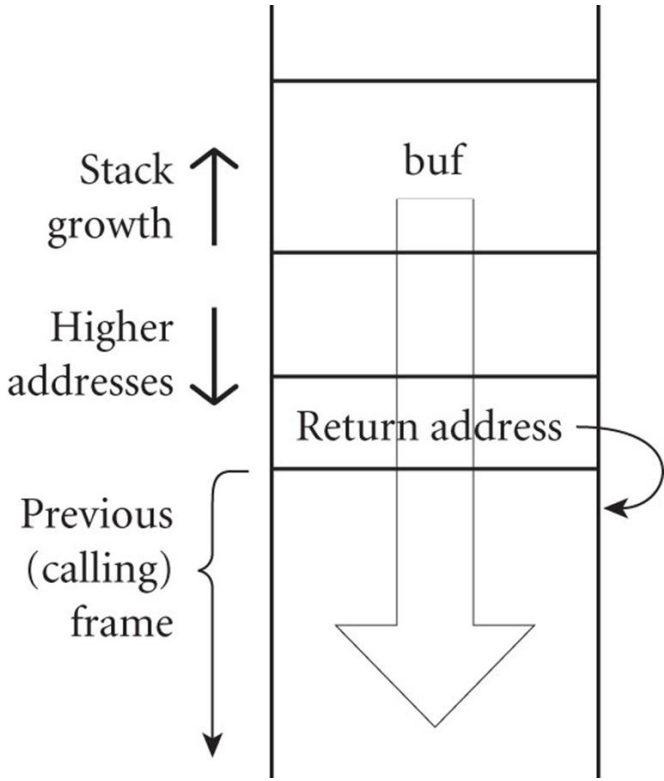
Today the use of pointer arithmetic is mainly a matter of personal taste: some C programmers consider pointer-based algorithms to be more elegant than their array-based counterparts; others simply find them harder to read. Certainly the fact that arrays are passed as pointers makes it natural to write subroutines in the pointer style.

Stack smashing

The lack of bounds checking on array subscripts and pointer arithmetic is a major source of bugs and security problems in C. Many of the Internet viruses have propagated by means of stack smashing, a nasty form of buffer overflow attack. Consider a routine designed to read a number from an input stream:

If the stream provides more than 100 characters without a newline ('\n'), those characters will overwrite memory beyond the confines of **buf**, as shown by the large white arrow in the figure. A careful attacker may be able to invent a string whose bits include both a sequence of valid machine instructions and a replacement value for the subroutine's return address. When the routine attempts to return, it will jump into the attacker's instructions instead. **Stack smashing** can be prevented by manually checking array bounds in C, or by configuring the hardware to prevent the execution of instructions in the stack.

```
int get_acct_num(FILE *s) {
    char buf[100];
    char *p = buf;
    do {
        /* read from stream s: */
        *p = getc(s);
    } while (*p++ != '\n');
    *p = '\0';
    /* convert ascii to int: */
    return atoi(buf);
}
```



Dangling References

SECTION 11

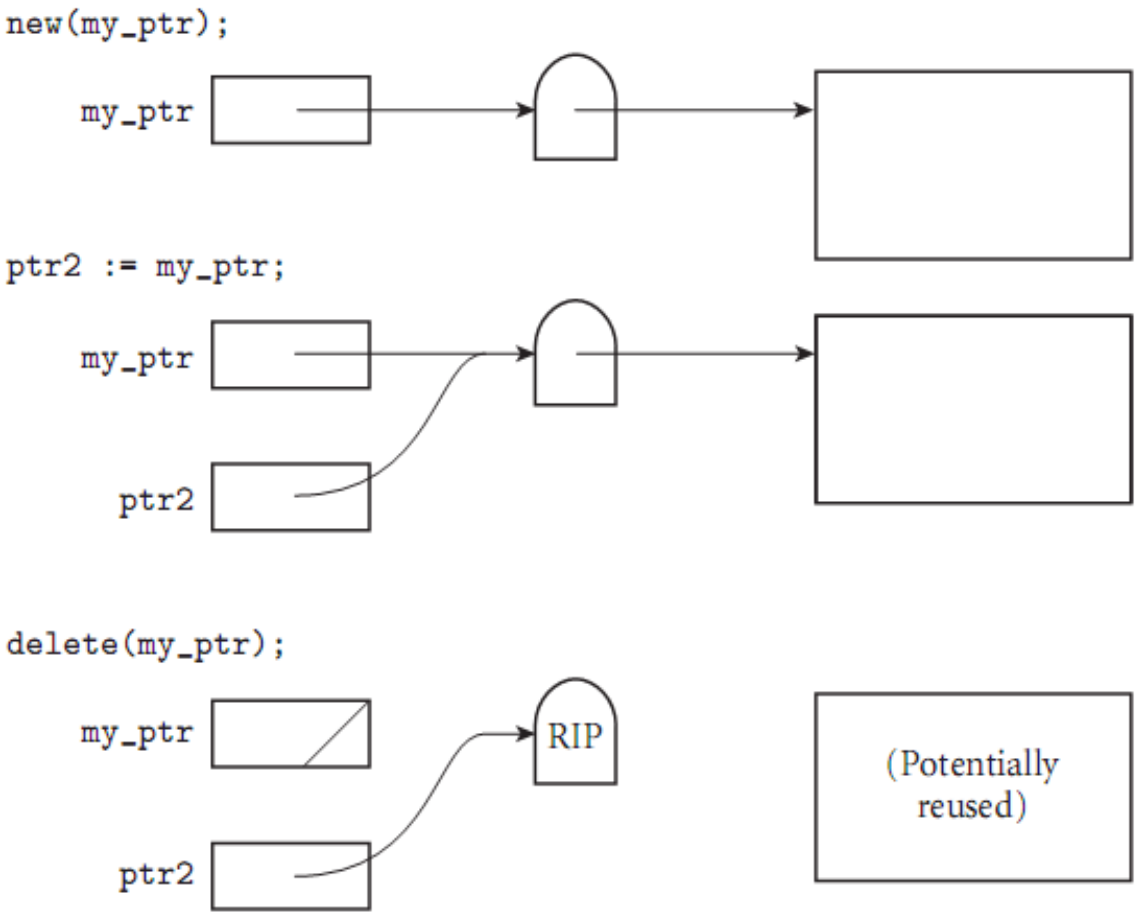
Dangling References

- Problems with dangling pointers are due to
  - **explicit deallocation** of heap objects (C/C++ language)
    - only in languages that *have* explicit deallocation
  - implicit deallocation of elaborated objects (Java/C#)
- Two implementation mechanisms to catch dangling pointers
  - Tombstones
  - Locks and Keys

Dangling References

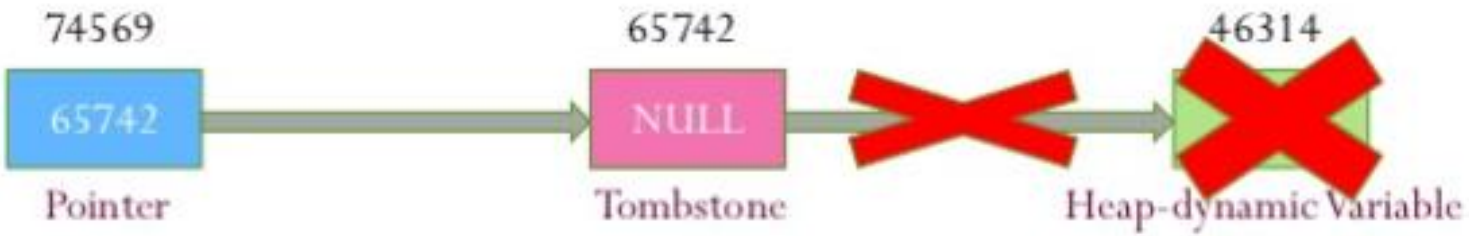
Tombstone

Figure 8.17 (CD) Tombstones. A valid pointer refers to a tombstone that in turn refers to an object. A dangling reference refers to an "expired" tombstone.



Solution: Tombstone

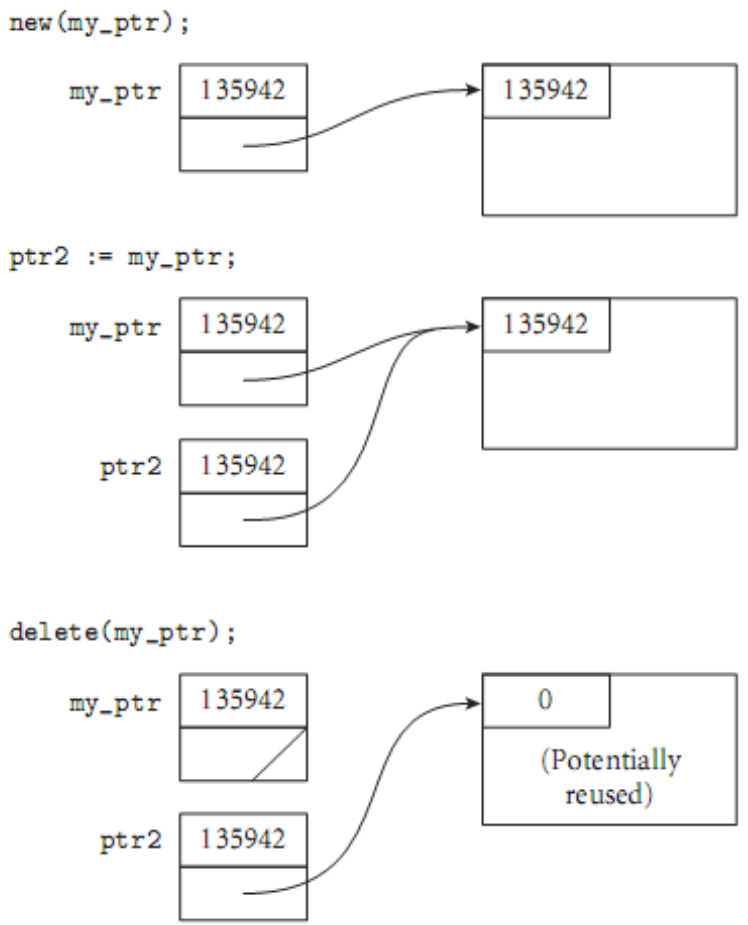
- Tombstones:
  - Every heap-dynamic variable includes an extra cell called tombstone
  - The Tombstone is a pointer to the heap dynamic variable.
  - When the heap-dynamic variable is deallocated, the tombstone is set to NULL.
- Disadvantages :
  - Costly in memory and time.



Dangling References

Locks and Keys

Figure 8.18 (CD) Locks and Keys. A valid pointer contains a key that matches the lock on an object in the heap. A dangling reference is unlikely to match.



Solution: Lock-and-Keys

- Pointers instead of only being an address, also have a key and are made into a pair. (key, address)
- Heap-dynamic variables also have a header containing a lock.
- When allocated, the keya and the lock are set the same. (key=lock)
- Access is granted only if the lock and key match. Otherwise it's an error.
- In deallocating, the lock is cleared to an illegal lock value. When other pointers try to access it, the key doesn't match th e it is not allow

Garbage Collection I

Theory and Reference Counting

SECTION 12

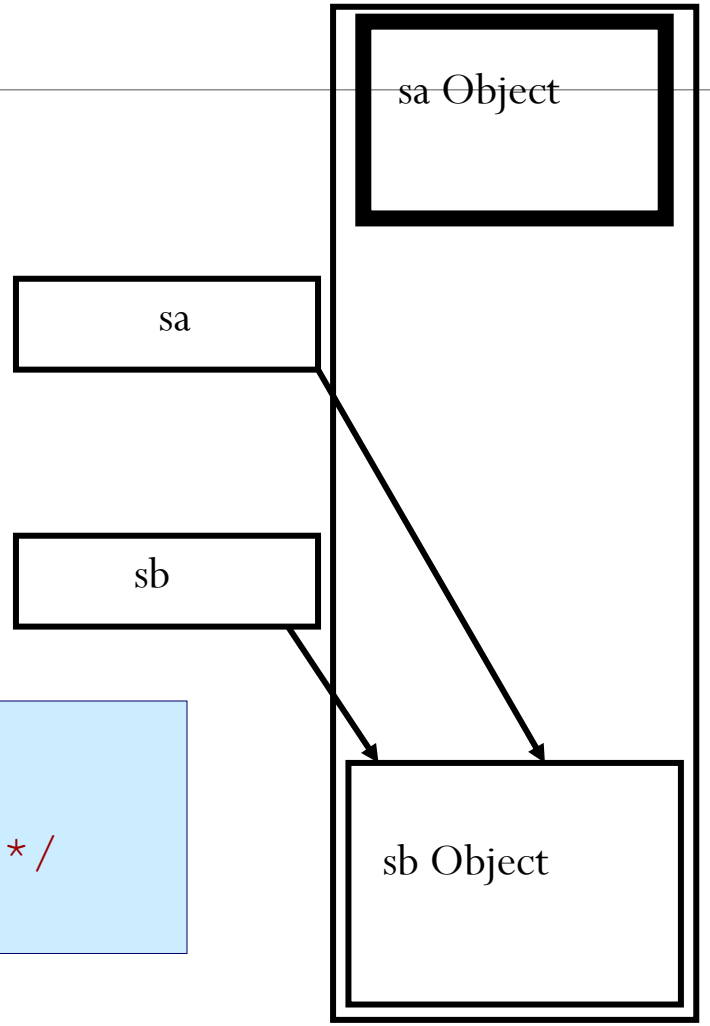
What is Garbage?

Garbage: unreferenced objects

```
Student sa= new Student();
Student sb= new Student();
sa=sb;
```

Now sa Object becomes a garbage,

It is unreferenced Object.




```
void foo() {
int *p = malloc(128);
   return; /* p block is now garbage */
}
```

What is Garbage Collection?

- What is Garbage Collection?
  - Finding garbage and reclaiming memory allocated to it.
- Why Garbage Collection?
  - the heap space occupied by an un-referenced object can be recycled and made available for subsequent new objects
- When is the Garbage Collection process invoked?
  - When the total memory allocated to a Java program exceeds some threshold.
- Is a running program affected by garbage collection?
  - Yes, the program suspends during garbage collection.

Advantages of Garbage Collection

- GC eliminates the need for the programmer to deallocate memory blocks explicitly.
- Garbage collection helps ensure program integrity.
- Garbage collection can also dramatically simplify programs.

Disadvantages of Garbage Collection

- Garbage collection adds an overhead that can affect program performance.
- GC requires extra memory.
- Programmers have less control over the scheduling of CPU time.

Garbage Collection

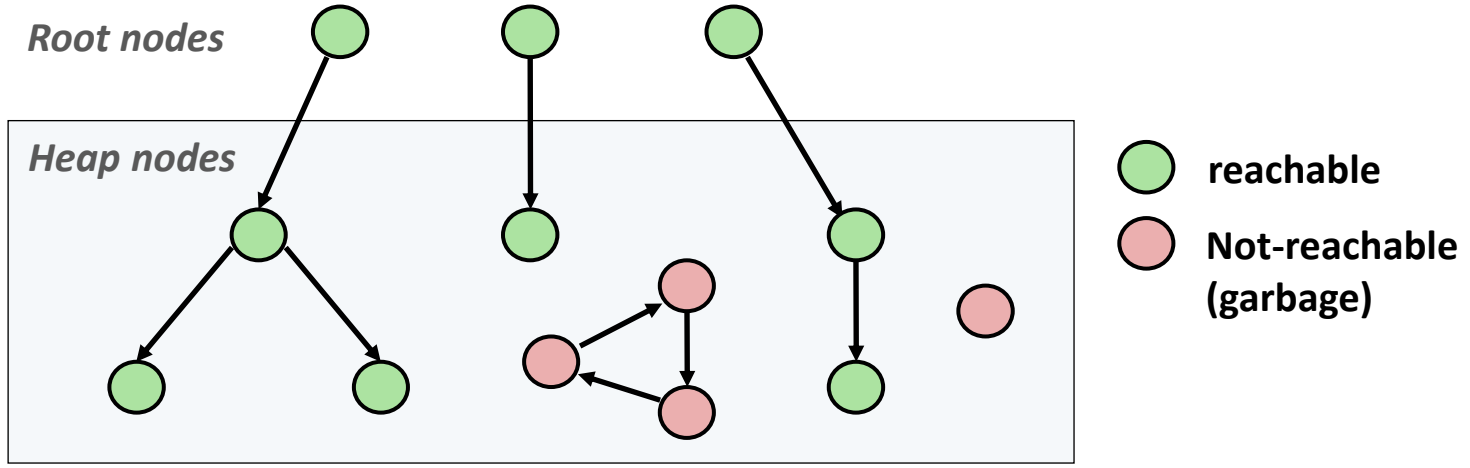
Issues with Garbage Collection Mechanism Design

- many languages leave it up to the programmer to design without garbage creation - this is VERY hard
- others arrange for automatic garbage collection
- reference counting
  - does not work for circular structures
  - works great for strings
  - should also work to collect unneeded tombstones

Java, C#, Scala, Go and all major scripting languages.

Memory as a Graph

- We view memory as a directed graph
  - Each block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called ***root*** nodes (e.g. registers, locations on the stack, global variables)



A node (block) is **reachable** if there is a path from any root to that node.

Non-reachable nodes are **garbage**(cannot be needed by the application)

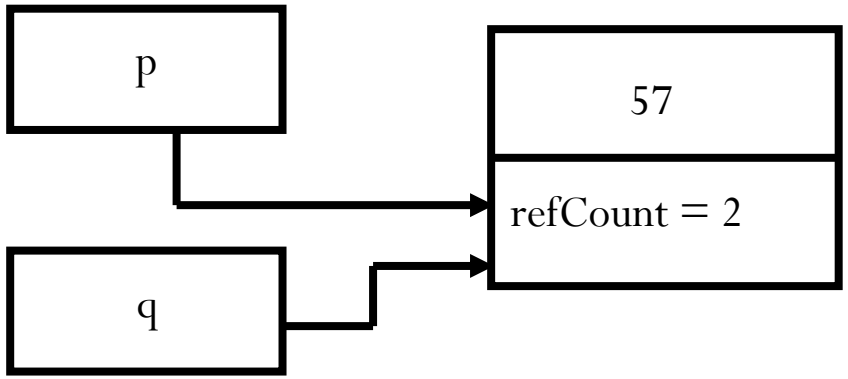
Reference Counting Garbage Collection

Main Idea: Add a reference count field for every object.

This Field is updated when the number of references to an object changes.

Example

```
Object p= new Integer(57);
Object q = p;
```



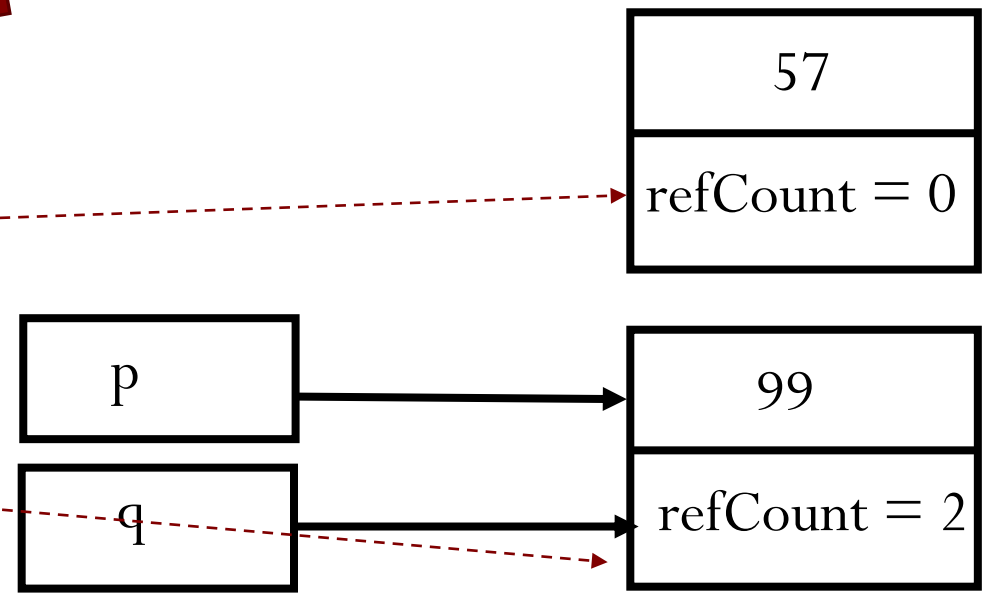
Reference Counting (cont'd)

- The update of reference field when we have a reference assignment (i.e p=q) can be implemented as follows:

Example:
Object p = new Integer(57);
Object q= new Integer(99);
p=q

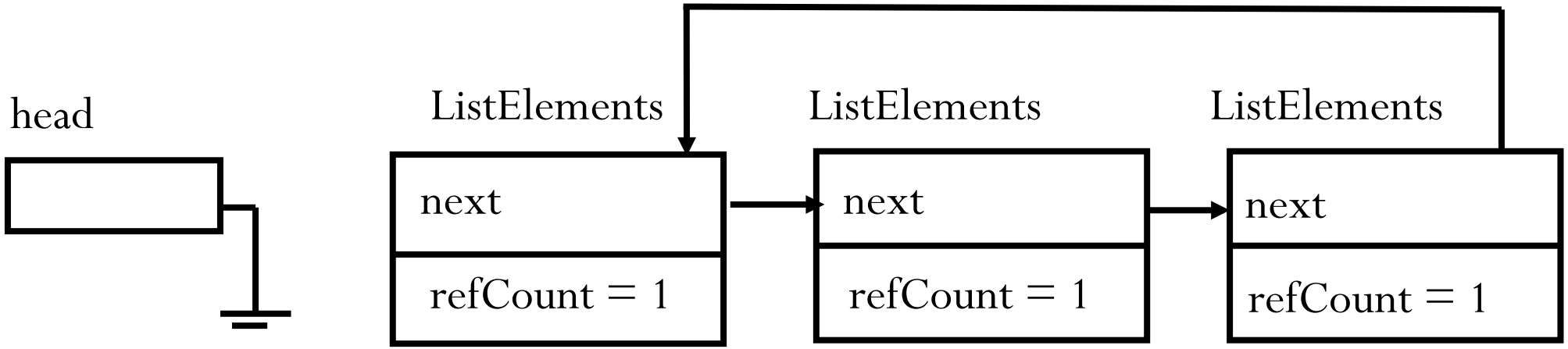
Code Substitution

```
if (p!=q)
{
    if (p!=null)
      --p.refCount;
    p=q;
    if (p!=null)
      ++p.refCount;
}
```



Reference Counting (cont'd)

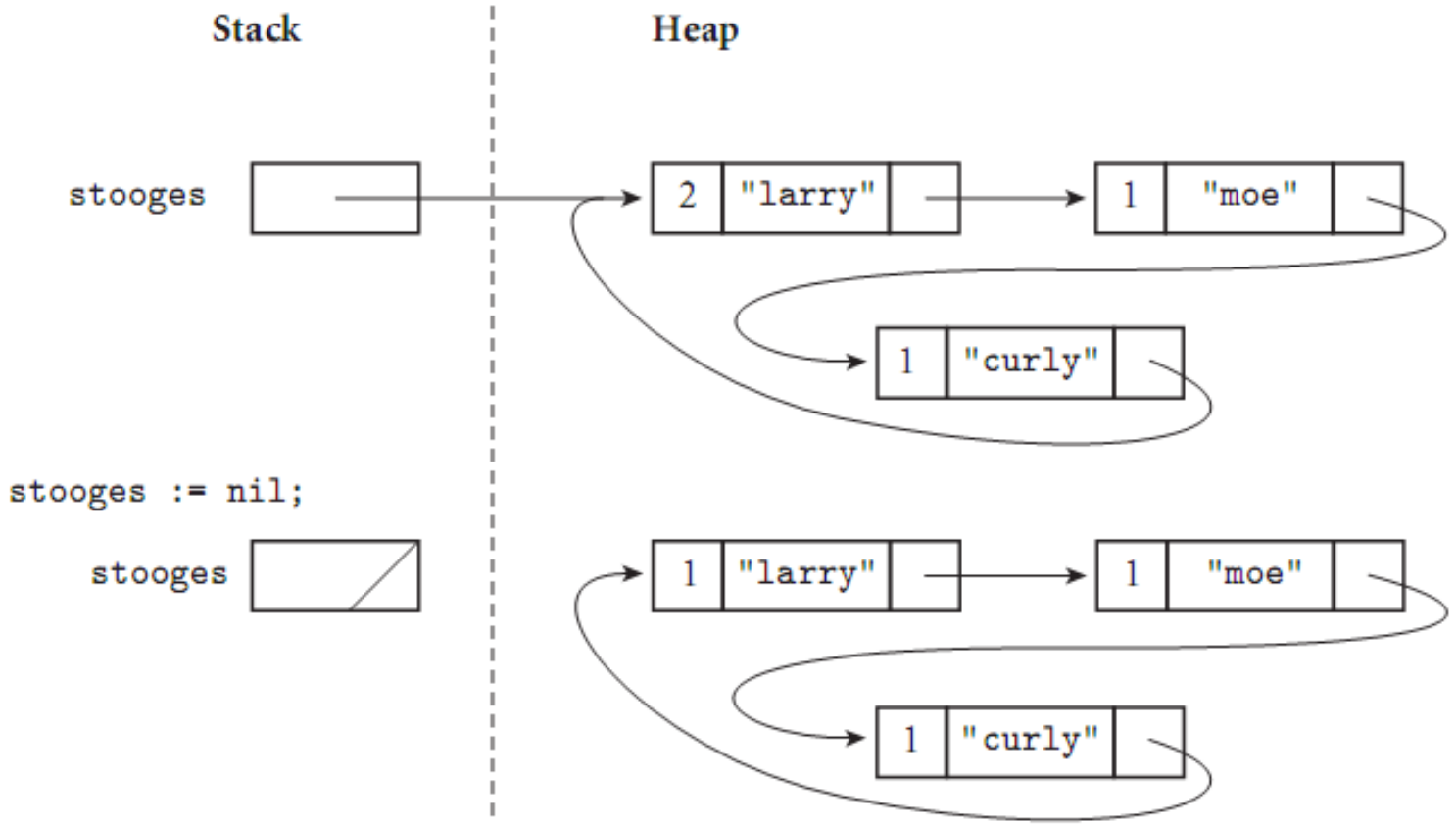
- Reference counting will fail whenever the data structure contains a cycle of references and the cycle is not reachable from a global or local reference.



Garbage Collection

Garbage collection with reference counts

Figure 8.14 Reference counts and circular lists. The list shown here cannot be found via any program variable, but because it is circular, every cell contains a nonzero count.



Reference Counting (cont'd)

- Advantages
  - Conceptually simple: Garbage is easily identified
  - It is easy to implement.
  - Immediate reclamation of storage

- Disadvantages
  - Reference counting does not detect garbage with cyclic references.
  - The overhead of incrementing and decrementing the reference count each time.
  - Extra space: A count field is needed in each object.
  - It may increase heap fragmentation.

Garbage Collection II

Mark and Sweep

SECTION 13

Garbage Collection

Mark-and-Sweep

- commonplace in Lisp dialects
- complicated in languages with rich type structure, but possible if language is strongly typed
- achieved successfully in Java, C#, Scala, Go
- complete solution impossible in languages that are not strongly typed
- conservative approximation possible in almost any language (Xerox Portable Common Runtime approach)

Mark and Sweep Algorithm

Positive Marking

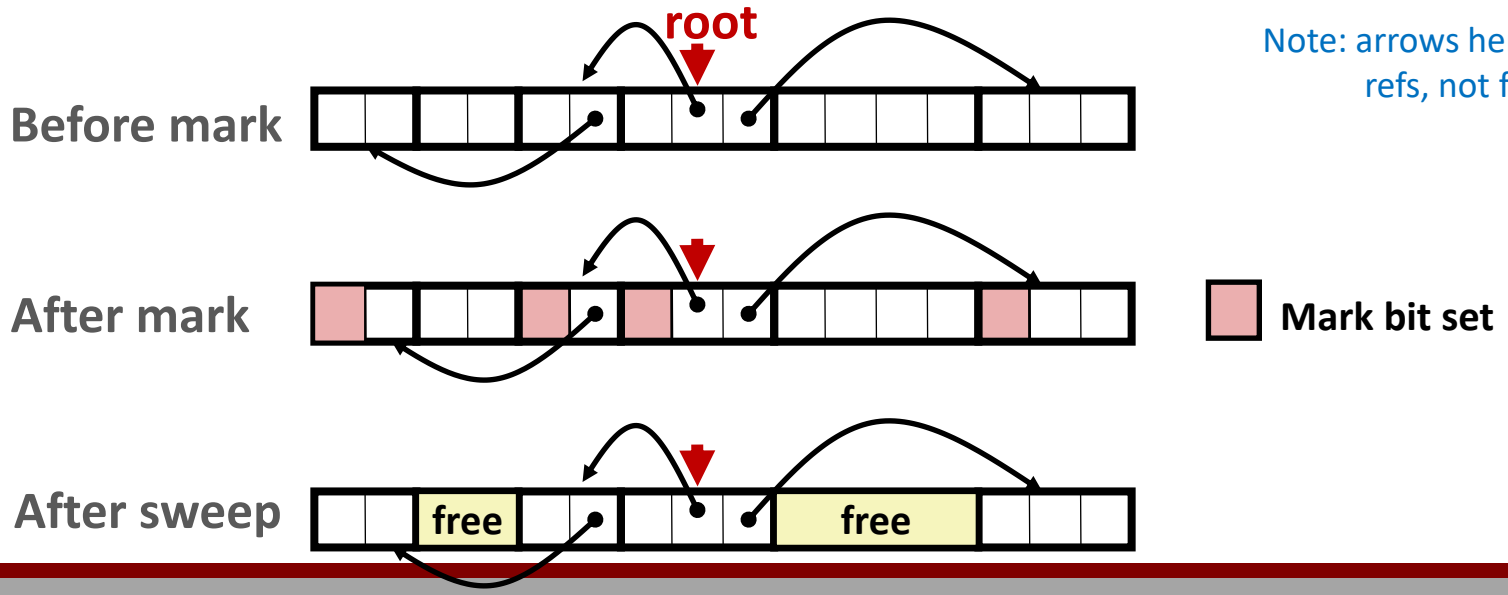
1. The collector walks through the heap, tentatively marking every block as "useless."

2. Beginning with all pointers outside the heap, the collector recursively explores all linked data structures in the program, marking each newly discovered block as "useful." (When it encounters a block that is already marked as "useful," the collector knows it has reached the block over some previous path, and returns without recursing.)

3. The collector again walks through the heap, moving every block that is still marked "useless" to the free list.

Mark and Sweep Collecting

- Can build on top of malloc/free package
  - Allocate using `malloc` until you "run out of space"
- When out of space:
  - Use extra **mark bit** in the head of each block
  - **Mark:** Start at roots and set mark bit on each reachable block
  - **Sweep:** Scan all blocks and free blocks that are not marked



Note: arrows here denote memory refs, not free list ptrs.

Assumptions For a Simple Implementation

- Application
  - **`new(n)`:** returns pointer to new block with all locations cleared
  - **`read(b,i)`:** read location **`i`** of block **`b`** into register
  - **`write(b,i,v)`:** write **`v`** into location **`i`** of block **`b`**

- Each block will have a header word
  - addressed as **`b[-1]`**, for a block **`b`**
  - Used for different purposes in different collectors

- Instructions used by the Garbage Collector
  - **`is_ptr(p)`:** determines whether `p` is a pointer
  - **`length(b)`:** returns the length of block `b`, not including the header
  - **`get_roots()`:** returns all the roots

Mark and Sweep (cont.)

Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // do nothing if not pointer
   if (markBitSet(p)) return;       // check if already marked
   setMarkBit(p);                   // set the mark bit
   for (i=0; i< length(p); i++)   // call mark on all words
     mark(p[i]); //    in the block
   return;
}
```

Mark and Sweep (cont.)

Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p);
}
```

Pointer reversal

Can be used in Mark phase to save time.

Problem: The exploration step (Step 2) of mark-and-sweep collection is naturally recursive.

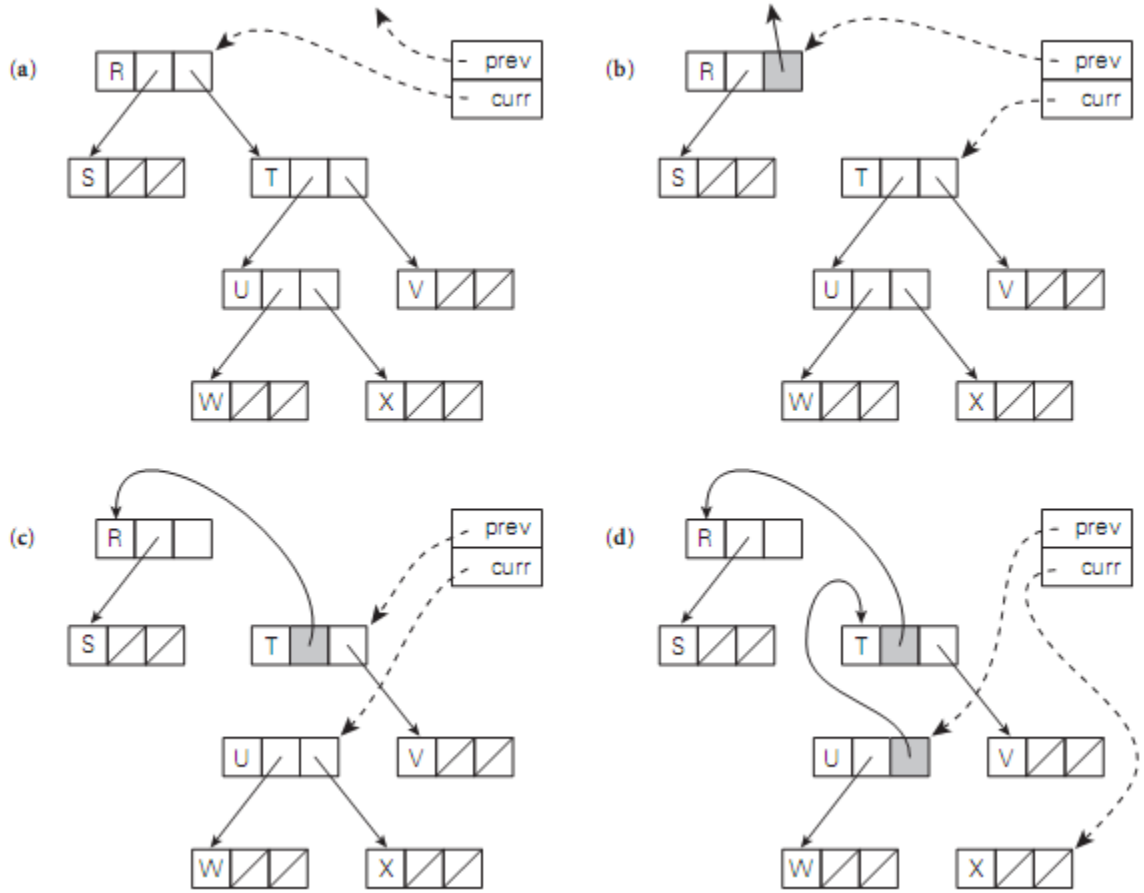
Idea: during DFS, each pointer only followed once. Can *reverse pointers* after following them -- no recursion needed! (Deutsch-Waite-Schorr alg.)

Implication: objects are broken while being traversed; all computation over objects must be halted during mark phase (oops)

Garbage Collection

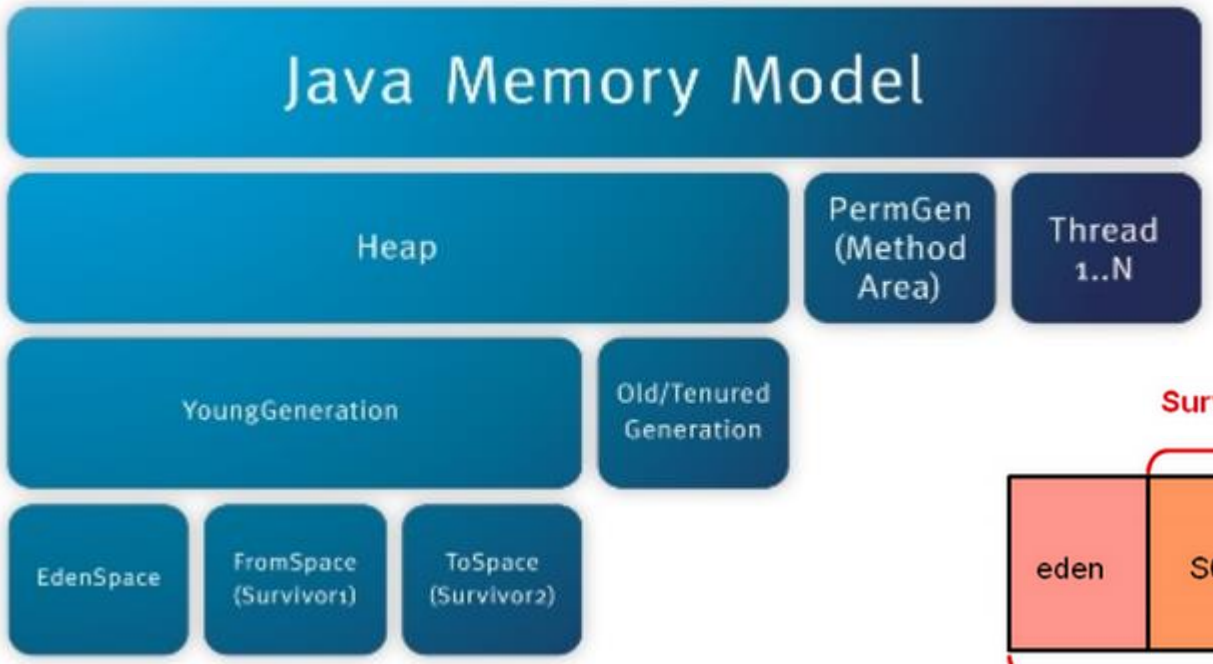
Heap exploration via pointer reversal

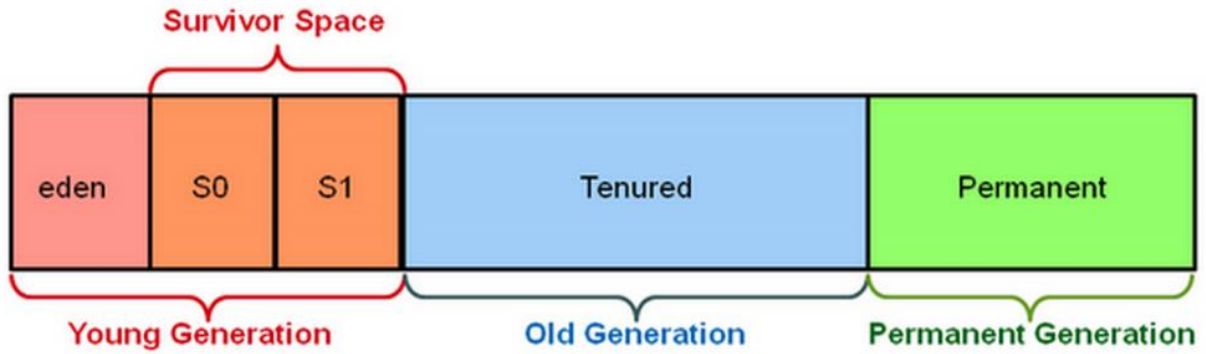
- The block currently under examination is indicated by the **curr** pointer.
- The previous block is indicated by the **prev** pointer. As the garbage collector moves from one block to the next, it changes the pointer it follows to refer back to the previous block. When it returns to a block it restores the pointer.
- Each reversed pointer must be marked (indicated with a shaded box), to distinguish it from other, forward pointers in the same block.
- Unmarked one can be removed.



Real World Garbage Collector

GC1 Garbage Collector
http://www.oracle.com/technetwork/tutorials/tutorials-1876574.html





Lists

SECTION 14

Lists

- A list is defined recursively as either the empty list or a pair consisting of an object (which may be either a list or an atom) and another (shorter) list
  - Lists are ideally suited to programming in functional and logic languages
    - In Lisp, in fact, a program *is* a list, and can extend itself at run time by constructing a list and executing it
  - Lists can also be used in imperative programs

Basic list operations in Lisp

- The most fundamental operations on lists are those that construct them from their components or extract their components from them. In Lisp:

```
(cons 'a '(b))              ⟹ (a b)
(car '(a b))                ⟹ a
(car nil)                   ⟹ ??
(cdr '(a b c))              ⟹ (b c)
(cdr '(a))                  ⟹ nil
(cdr nil)                   ⟹ ??
(append '(a b) '(c d))      ⟹ (a b c d)
```

- Here we have used ⇒ to mean “evaluates to.” The **car** and **cdr** of the empty list(nil) are defined to be nil in Common Lisp; in Scheme they result in a dynamic semantic error.

Basic list operations in ML

- In ML the equivalent operations are written as follows:

```
a :: [b]              ⟹ [a, b]
hd [a, b]             ⟹ a
hd []                 ⟹ run-time exception
tl [a, b, c]          ⟹ [b, c]
tl [a]                ⟹ nil
tl []                 ⟹ run-time exception
[a, b] @ [c, d]       ⟹ [a, b, c, d]
```

- Run-time exceptions may be caught by the program if desired; further details will appear in Section 8.5.

ML and Lisp

Both ML and Lisp provide many additional list functions, including:

- ones that test a list to see if it is empty;
- return the length of a list; return the nth element of a list, or a list consisting of all but the first n elements;
- reverse the order of the elements of a list;
- search a list for elements matching some predicate; or
- apply a function to every element of a list, returning the results as a list.

List comprehensions

- Miranda, Haskell, Python, and F# provide lists that resemble those of ML, but with an important additional mechanism, known as list comprehensions. These are adapted from traditional mathematical set notation. A common form comprises an expression, an enumerator, and one or more filters. In Haskell, the following denotes a list of the squares of all odd numbers less than 100:

```
[i*i | i <- [1..100], i `mod` 2 == 1]
```

In Python we would write

```
[i*i for i in range(1, 100) if i % 2 == 1]
```

In F# the equivalent is

```
[for i in 1..100 do if i % 2 = 1 then yield i*i]
```

All of these are the equivalent of the mathematical

$$\{i \times i \mid i \in \{1, \ldots, 100\} \wedge i \bmod 2 = 1\}$$

We could of course create an equivalent list with a series of appropriate function calls. The brevity of the list comprehension syntax, however, can sometimes lead to remarkably elegant programs

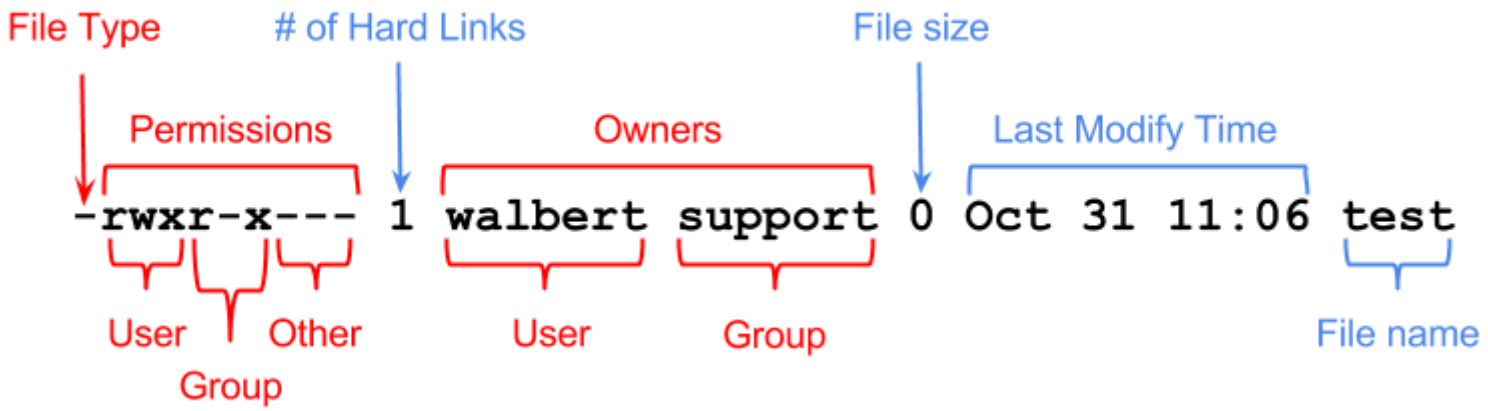
File and Input/Output

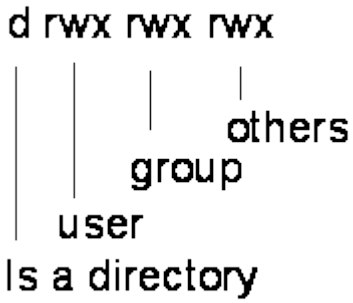
SECTION 15

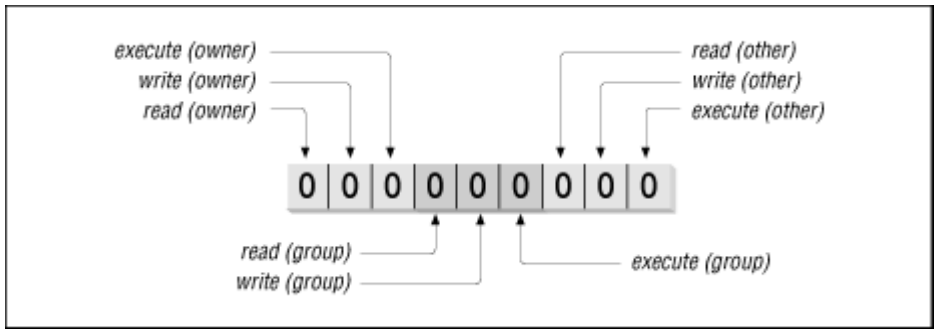
Files and Input/Output

- Input/output (I/O) facilities allow a program to communicate with the outside world
  - *interactive* I/O and I/O with files
- Interactive I/O generally implies communication with human users or physical devices
- Files generally refer to off-line storage implemented by the operating system
- Files may be further categorized into
  - *temporary*
  - *persistent*

File Permissions (Linux/Unix)







Layers of File System Abstraction

Application Programs

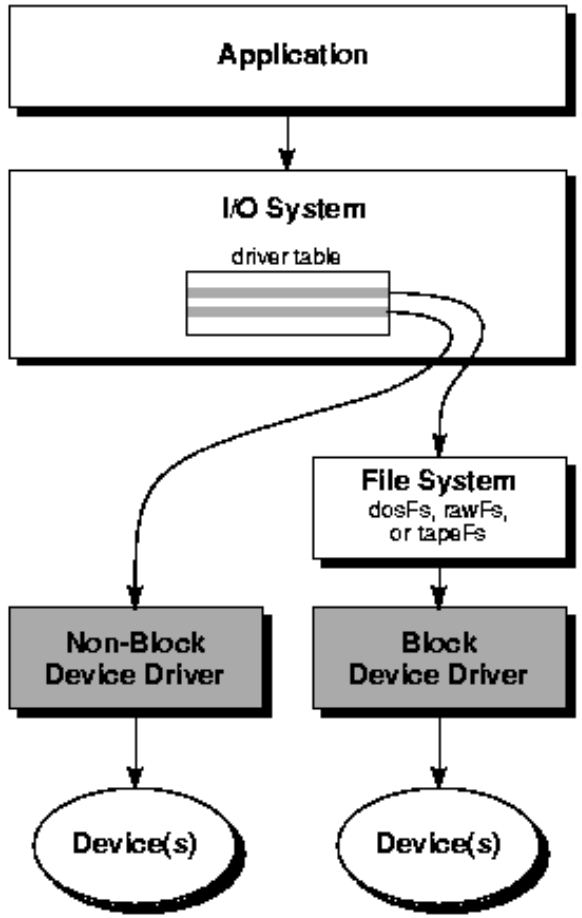
Logical File System: manages metadata (all of the file-system structure except the actual contents of files) and directory structure

File-Organization Module: translates logical addresses to physical addresses for the basic file system

Basic File System: able to issue generic commands to device drivers, manages memory buffers and caches

I/O Control: device drivers and interrupt handlers to transfer information between main memory and disk system

Storage Devices



Handling Files

1. Declare FILE type variable
   - Use standard library :
   - #include <stdio.h>
     FILE* fp;
2. Open a file
   - Connect FILE type variable with actual file using fopen function
     fp=fopen("test.txt","r");
3. Perform I/O with the open file
   Text file : fscanf(), fprintf(), fgets(), fputs(), …
   Binary file : fread(), fwrite()
4. Close a file
   - Break the connection between FILE type variable and actual file
     fclose(fp);

File Open

Before using a file, you must open the file

`fopen()`

- FILE *fopen(const char *filename, const char *mode);

Example 1

```
FILE *fp;
fp = fopen("c:\work\text.txt", "r");
if (fp == NULL) {
    printf(“file open error!\n");
}
```

Example 2

```
fp = fopen("outdata.txt", "w");
fp = fopen("outdata.txt", "a");
```

File Open Modes

r or **rb**

- Open file for reading.

w or **wb**

- Truncate to zero length or create file for writing.

a or **ab**

- Append; open or create file for writing at end-of-file.

r+ or **rb+** or **r+b**

- Open file for update (reading and writing).

b stands for “binary”

Text File Processing

```
char* fgets(char *s, int n, FILE *fp);

int fputs(const char *s, FILE *fp);

int fprintf(FILE *fp, const char *format, ...);

int fscanf(FILE *fp, const char *format, ...);
```

deleteLine.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *fp1;
  FILE *fp2;
  char buffer[100];

  fp1 = fopen(argv[1], "r");
    fp2 = fopen(argv[2], "w");
    if (fp1 == NULL) {
                printf("file not found");
                exit(1);
     }

  while(fgets(buffer, 100, fp1) != NULL)
                if (*buffer != '\n')
                                fputs(buffer, fp2);
  fclose(fp1);
  fclose(fp2);
}
```

Binary File Processing

```
int fread(void *buf, int size, int n, FILE *fp);

int fwrite(const void *buf, int size, int n, FILE *fp);
```

student1.c

output:
```
Student_id  Name  year  Major
200601001 Mike    1 Computer
200601002 Tom     1 Computer
...
^Z
```

```
#include <stdio.h>
#include <stdlib.h>
#include "student.h"

main()
{
  struct student st, *stp = &st;
  FILE *fp = fopen("st_file", "wb");
  if(fp == NULL ) {
                printf("file open error\n");
                exit(1);
  }

  printf("student_id     Name  Year  Major\n“);
  while (scanf("%d %s %d %s", &st.stud_id, st.name, &st.year, st.major) == 4)
  {
                fwrite(stp, sizeof(struct student), 1, fp);
  }
  fclose(fp);
}
```

student2.c

output:

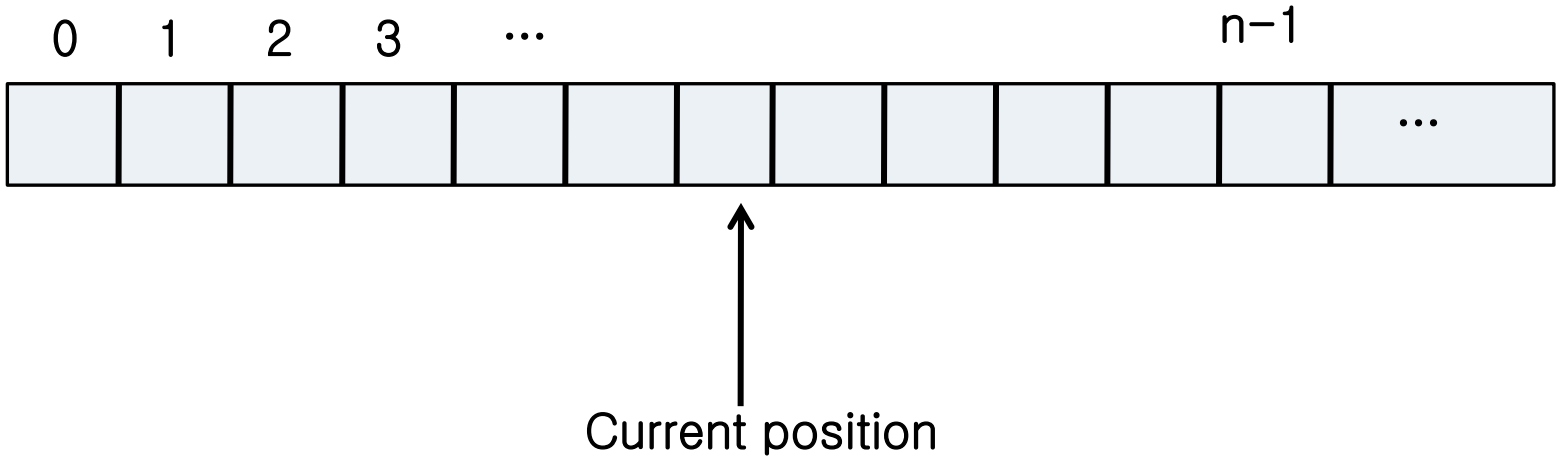
```
---------------------------------
  Student_ID  Name  Year  Major
---------------------------------
   0601001   Mike     1  Computer
   0601002   Tom      1  Computer
        ...


---------------------------------
```

```
#include <stdio.h>
#include <stdlib.h>
#include "student.h"
int main({
  struct student st, *stp = &st;
  FILE *fp = fopen("st_file", "rb");

  if (fp == NULL ) {
                printf("File Open Error.\n");
                exit(1);
  }
  printf("---------------------------------\n");
  printf("%10s %6s %6s %10s\n", "Student_ID", "Name", "Year", "Major");
  printf("---------------------------------\n");

  while (fread(stp, sizeof(struct student), 1, fp) > 0)
  {
                printf("%10d %6s %6d %10s\n", st.stud_id, st.name, st.year, st.major);
  }
  printf("---------------------------------\n");
  fclose(fp);
  return 0;
}
```

Random Access

Current file position

- Updated after file I/O is performed



Related Functions

int fseek(FILE *fp, long offset, int mode)

void rewind(FILE *fp)

int ftell(FILE *fp)

fseek mode

mode	val	meaning
SEEK_SET	0	file start
SEEK_CUR	1	Current
SEEK_END	2	file end

Example

- fseek(fp, 0L, SEEK_SET)
- fseek(fp, 100L, SEEK_CUR)
- fseek(fp, 0L, SEEK_END)

```c
#include <stdio.h>
#include <stdlib.h>
#define  MAX  256

int main(int argc, char *argv[]){
        FILE  *fp;
        char w[MAX];

        if ((fp = fopen(argv[1], "a+") ) == NULL ) {
                fprintf(stderr, "file open error!\n");
                exit(1);
        }

        puts("Insert a sentence to a file. ");
        puts("(Just press enter to end the input.)\n");

        while (gets(w) != NULL && w[0] != '\0')
                fprintf(fp, "%s", w);

        puts("file contents :");
        rewind(fp);

        while (fscanf(fp, "%s", w) == 1)
                puts(w);

        if (fclose(fp) != 0)
        fprintf(stderr, "file close error! \n");
        return 0;
}
```

output:

```
C:> wordAppend message.txt
Insert a sentence to a file.
Just press enter to end the input.)
Hello world.
Thank you very much.
```

File Contents:

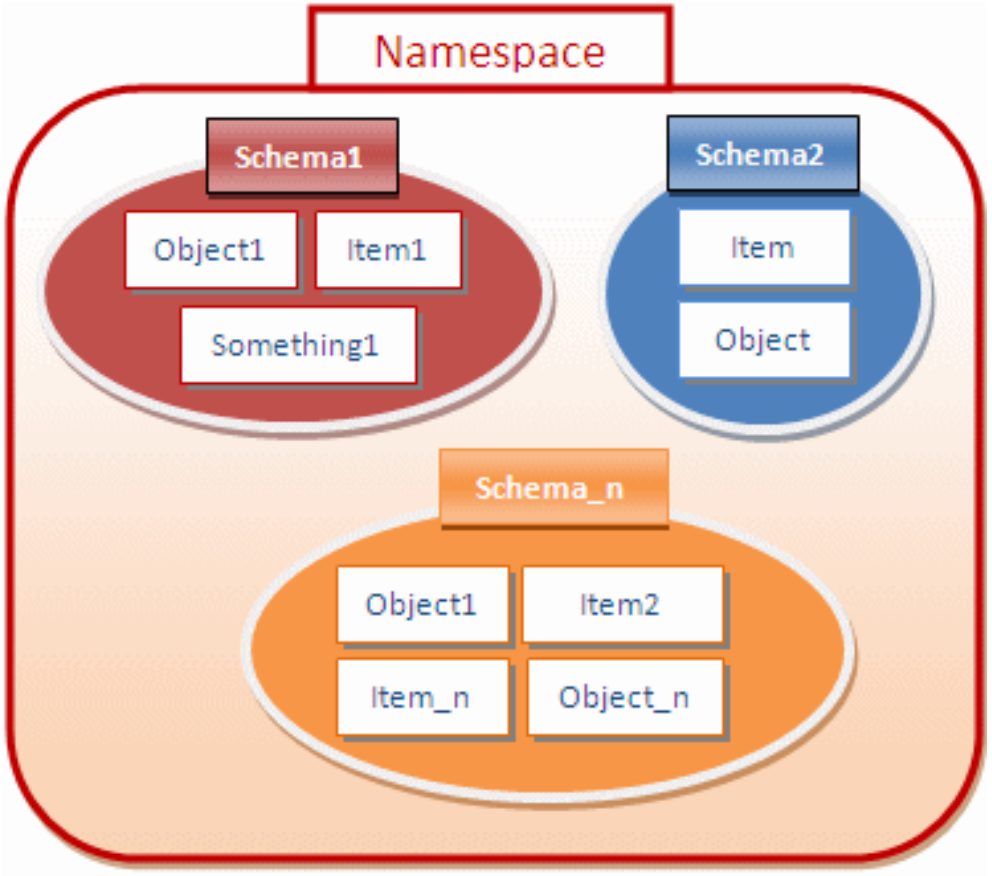
```
Hello
world.
Thank
you
very
much.
```

Namespace

SECTION 16

What is Namespaces in C++ Programming Language?

Using namespace we can have class, global variables, functions under one name. We can change global scope to Sub-scope.



Namespace

1. Namespace is a logical compartment used to avoid naming collisions.
2. The naming conflict or Collision always Occurs may occur –
3. When same program is using more than one library with some variables or functions having same name.
4. The name collision may occur for global variables, global functions, classes etc.
5. Default namespace is global namespace and can access global data and functions by proceeding (::) operator.
6. We can create our own namespace. And anything declared within namespace has scope limited to namespace.

Creating a namespace

- Creation of namespace is similar to creation of class.
- Namespace declarations appear only at global scope.
- Namespace declarations can be nested within another namespace.
- Namespace declarations don’t have access specifiers. (Public or private)
- No need to give semicolon after the closing brace of definition of namespace.
- We can split the definition of namespace over several units.

Syntax

```
namespace namespace_name
{
//member declarations

}
```

```
//In firstHeader.h
namespace ns1
{
   class one
   {
      //----Declarations---
   };
   class two
   {
      //----Declarations---
   };
}

//In secondHeader.h
namespace ns1    // can continue the defination of ns1 over multiple header
{
   class three
   {
      //----Declarations---
   };
}
```

What is Unnamed Namespace ?

- Unnamed namespaces are the replacement for the static declaration of variables.
- They are directly usable in the same program and are used for declaring unique identifiers.
- In unnamed namespaces, name of the namespace in not mentioned in the declaration of namespace.
- The name of the namespace is uniquely generated by the compiler.
- The unnamed namespaces you have created will only be accessible within the file you created it in.

Demo Program: ns1/main.cpp

Go Code::Blocks!

main.cpp

```
1   #include <iostream>
2
3   using namespace std;
4
5   namespace
6   {
7       int i;
8   }
9
10  int main()
11  {
12      i=10;
13      cout<< "Value : "<<i;
14      return 0;
15  }
16
```

```
"C:\Eric_Chou\CppDev\ProgrammingLanguages\chapter 8\ns1\bin\Debug\ns1.exe"
Value : 10
Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

How to Use Namespaces in C++ Programming Language ?

Way 1 : Using Scope Resolution :

In this method, we use the namespace name, scope resolution operator (::) and member of that namespace.

Way 2 : Using Directive:

We can use ‘using’ directive to specify the namespace.

Demo Program: ns2/main.cpp

```
1   #include<iostream>
2   using namespace std;
3
4   namespace first
5   {
6   int i;
7   }
8
9   namespace second
10  {
11  int i;
12  }
13
14  int main()
15  {
16  first::i=1;    //scope resolution
17  second::i=2;
18
19  using first::i; //using directive
20  cout<<i;
21  }
22
```

```
"C:\Eric_Chou\CppDev\ProgrammingLanguages\chapter 8\ns2\bin\Debug\ns2.exe"
1
Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

How to Use Namespaces in C++ Programming Language ?

Way 3 : Using declaration:

It is a declaration within the current scope. This means it can override names from a using directive.

Demo Program: ns3/main.cpp

```
1   #include<iostream>
2   using namespace std;
3
4   namespace first
5   {
6   int i;
7   }
8   using namespace first;// using declaration
9   int main()
10  {
11  i=2;
12  cout<<"Value : "<<i<<endl;
13  }
14
```

```
"C:\Eric_Chou\CppDev\ProgrammingLanguages\chapter 8\ns3\bin\Debug\ns3.exe"
Value : 2

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

Different Verities of Using Namespace :

1.We can Have Class Definition inside Namespace

- •num1,num2 are two variables under same namespace name "MyNamespace".
- •We can Initialize them in main function by using Scope Resolution Operator.
- •We can Declare Class inside Namespace and thus we can have multiple classes with same name but they must be in different namespace.

Demo Program: ns4/main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  namespace MyNameSpace {
5     int num1;
6     int num2;
7
8     class Student
9     {
10       int marks;
11    public:
12       Student() {
13          marks = num1;
14       }
15    };
16 }
17
18 int main()
19 {
20    MyNameSpace::num1 = 70;
21    MyNameSpace::num2 = 90;
22
23    MyNameSpace::Student ob1();
24    cout<< MyNameSpace::num1<< "   " << MyNameSpace::num2 << "   " << endl;
25 }
26
```

```
"C:\Eric_Chou\CppDev\ProgrammingLanguages\chapter 8\ns4\bin\Debug\ns4.exe"
70  90

Process returned 0 (0x0)   execution time : 0.074 s
Press any key to continue.
```

Different Verities of Using Namespace

2.We can Have Same Class Definition inside different Namespace

Demo Program: ns5/main.cpp

```cpp
#include <iostream>
using namespace std;

namespace MyNameSpace1 {
  int num1;
  int num2;

  class Student
  {
     int marks;
   public:
     Student() {
        marks = num1;
     }
  };
}
```

```cpp
namespace MyNameSpace2 {
  int num1;
  int num2;

  class Student
  {
     int marks;
   public:
     Student() {
        marks = num1;
     }
  };
}

int main()
{
  MyNameSpace1::num1 = 80;
  MyNameSpace1::num2 = 50;

  MyNameSpace1::Student ob1();
}
```