MODULE 2
Data Types

Data Types

- Introduction
- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Record Types
- List Types
- Pointer and Reference Types

Introduction

- A *data type* defines a collection of data objects and a set of predefined operations on those objects
- A *descriptor* is the collection of the attributes of a variable
- An *object* represents an instance of a user-defined abstract data types
- 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

1. Numeric Types

a). Integer

- There may be as many as eight different integer types in a language

- Java includes four signed integer sizes: byte, short, int, and long.

- The most common primitive numeric data type is integer.
- Some languages, for example, C++ and C#, include **unsigned integer types**, which are simply types for integer values without signs. Unsigned types are often used for binary data.
- A **signed integer value** is represented in a computer by a string of bits, with one of the bits (typically the leftmost) representing the sign.
- Most integer types are supported directly by the hardware. One example of an integer type that is not supported directly by the hardware is the long integer type of Python (F# also provides such integers). Values of this type can have unlimited length.

- A negative integer could be stored in sign-magnitude notation, in which the sign bit is set to indicate negative and the remainder of the bit string represents the absolute value of the number.

- Most computers now use a notation called **twos complement** to store negative integers, which is convenient for addition and subtraction.

- In twos-complement notation, the representation of a negative integer is formed by taking the logical complement of the positive version of the number and adding one.
- Ones-complement notation is still used by some computers.

- In ones-complement notation, the negative of an integer is stored as the logical complement of its absolute value. Ones-complement notation has the disadvantage that it has two representations of zero.

b). Floating Point

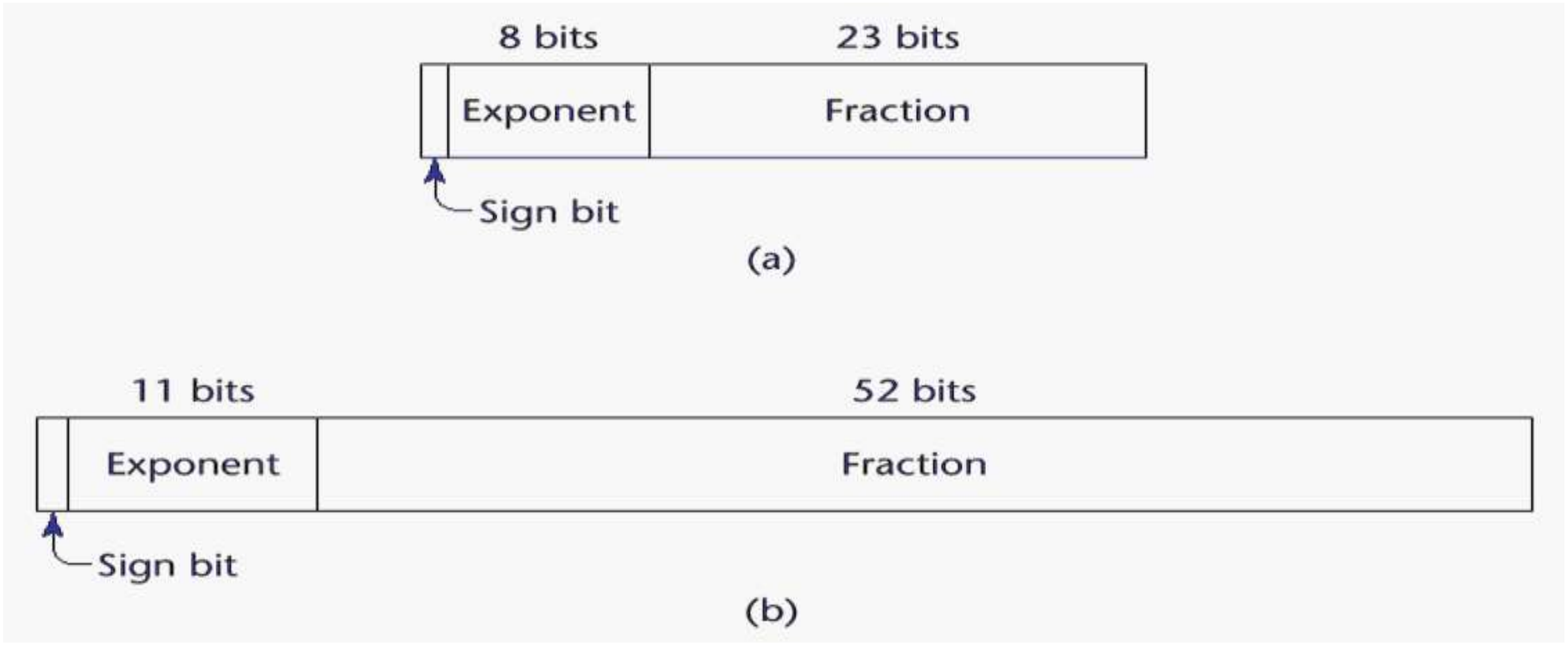
- Floating-point data types model real numbers, but the representations are only approximations for many real values. For example, neither of the fundamental numbers Π or e (the base for the natural logarithms) can be correctly represented in floating-point notation.

- Floating-point values are represented as fractions and exponents, a form that is borrowed from scientific notation.

- Older computers used a variety of different representations for floating-point values. However, most newer machines use the IEEE Floating-Point Standard 754 format.

- Most languages include two floating-point types, often called **float** and **double**.
- The float type is the standard size, usually being 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.

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



c). Complex

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

- Each value consists of two parts, the real part and the imaginary part

- In Python, the imaginary part of a complex literal is specified by following it with a j or J—for example,

  `(7 + 3j)`, where 7 is the real part and 3 is the imaginary part

d). Decimal

- Decimal data types store a fixed number of decimal digits, with the decimal point at a fixed position in the value.
- These are the primary data types for business data processing and are therefore essential to COBOL.
- C# and F# also have decimal data types.
- Store a fixed number of decimal digits, in coded form (BCD)

- *Advantage*: accuracy
- *Disadvantages*: limited range, wastes memory
  - The disadvantages of decimal types are that the range of values is restricted because no exponents are allowed, and their representation in memory is mildly wasteful

2. Boolean Types

- Simplest of all
- Range of values: two elements, one for "true" and one for "false"
- A Boolean value could be represented by a single bit, but because a single bit of memory cannot be accessed efficiently on many machines, they are often stored in the smallest efficiently addressable cell of memory, typically a byte.
- Advantage: readability

3. Character Types

- Character data are stored in computers as numeric codings.

- Most commonly used coding was the 8-bit code ASCII (American Standard Code for Information Interchange), which uses the values 0 to 127 to code 128 different characters.

- ISO 8859-1 is another 8-bit character code, but it allows 256 different characters. Ada 95+ uses ISO 8859-1.

- An alternative, 16-bit coding: Unicode (UCS-2)
  - Includes characters from most natural languages
  - The first 128 characters of Unicode are identical to those of ASCII.
  - Java was the first widely used language to use the Unicode character set.
  - JavaScript, Python, Perl, C# 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.

- Character string constants are used to label output, and the input and output of all kinds of data are often done in terms of strings.

➢Design Issues

The two most important design issues are:

- Should strings be simply a special kind of character array or a primitive type?
- Should strings have static or dynamic length?

➢Character String Types Operations

- Typical operations:
  - Assignment and copying
  - Comparison (=, >, etc.)
  - Catenation
  - Substring reference
  - Pattern matching
  - A substring reference is a reference to a substring of a given string.

➢Character String Type in Certain Languages

- C and C++
  - strings are not defined as a primitive type
  - Use **`char`** arrays
  - These languages provide a collection of string operations through standard libraries.
  - Some of the most commonly used library functions for character strings in C and C++ are **strcpy**, which moves strings; s**trcat**, which catenates one given string onto another; **strcmp**, which lexicographically compares (by the order of their character codes) two given strings; and **strlen**, which returns the number of characters

- SNOBOL4 (a string manipulation language)
  - Primitive
  - Many operations, including elaborate pattern matching
  - SNOBOL 4 was the first widely known language to support 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

There are several design choices regarding the length of string values.

- *Static length string* : Python, Java's `String` class

  Set when the string is created.

- *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
  - to allow strings to have varying length up to a declared and fixed maximum set by the variable's definition
  - string variables can store any number of characters between zero and the maximum
- *Dynamic Length strings* : JavaScript, Perl, and the standard C++ library
  - to allow strings to have varying length with no maximum
- Ada 95+ supports all three string length options

➢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.
- Strings as a primitive type to a language is not costly in terms of either language or compiler complexity
- Dynamic length strings are obviously the most flexible.

➢Implementation of Character String Types

- Character string types could be supported directly in hardware; but in most cases, software is used to implement string storage, retrieval, and manipulation.

- Static length: compile-time descriptor
  - The first field of every descriptor is the name of the type.
  - In the case of static character strings, the second field is the type's length (in characters).
  - The third field is the address of the first character.
- Limited dynamic length: require a run-time descriptor to store both the fixed maximum length and the current length
  - The limited dynamic strings of C and C++ do not require run-time descriptors, because the end of a string is marked with the null character
- Dynamic length: require a simpler run-time descriptor because only the current length needs to be stored.

Figure 6.2

Compile-time descriptor for static strings

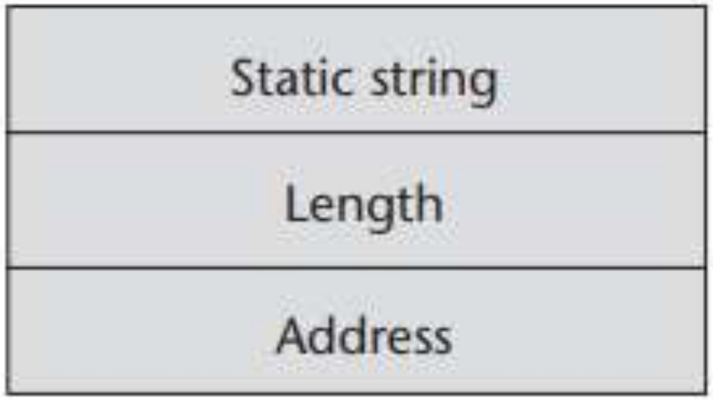
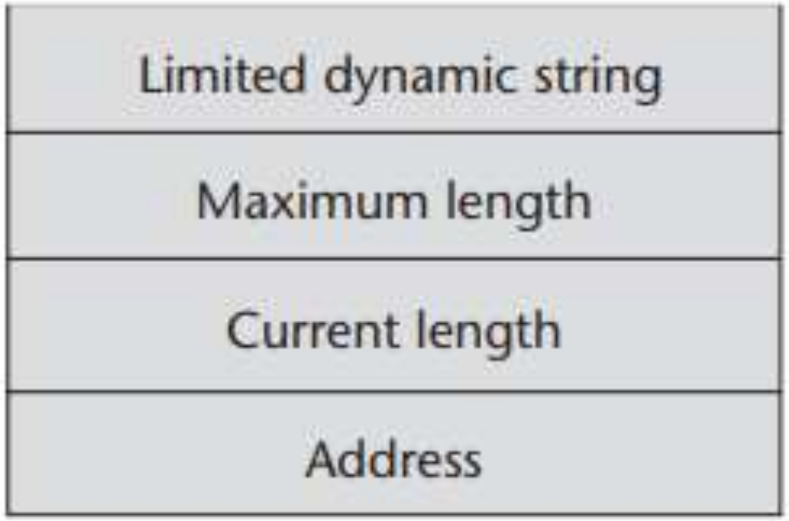


Figure 6.3

Run-time descriptor for limited dynamic strings



- Static length and limited dynamic length strings require no special dynamic storage allocation.

- Dynamic length strings require more complex storage management.
- The length of a string, and therefore the storage to which it is bound, must grow and shrink dynamically.
- There are three approaches to supporting the dynamic allocation and deal location that is required for dynamic length strings.

- First, strings can be stored in a linked list
- The second approach is to store strings as arrays of pointers to individual characters allocated in the heap.
- The third alternative is to store complete strings in adjacent storage cells.

User-Defined Ordinal Types

- An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers
- Examples of primitive ordinal types in Java
  - `integer`
  - `char`
  - `boolean`
- There are two user-defined ordinal types that have been supported by programming languages:

  1. enumeration
  2. subrange

1.Enumeration Types

- All possible values, which are named constants, are provided in the definition
- C# example

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

- Design issues
  - Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant checked?
  - Are enumeration values coerced to integer?
  - Any other type coerced to an enumeration type?

➢Design of Enumerated Type

- In languages that do not have enumeration types, programmers usually simulate them with integer values.

- For example, suppose we needed to represent colors in a C program and C did not have an enumeration type. We might use Data Types 0 to represent blue, 1 to represent red, and so forth.

  These values could be defined as follows:

  ```
  int red = 0, blue = 1;
  ```

- C and Pascal were the first widely used languages to include an enumeration data type.

- C++ includes C's enumeration types.

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

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

  colors myColor = blue, yourColor = red;

- The colors type uses the default internal values for the enumeration constants, 0, 1, . . . , although the constants could have been assigned any integer literal (or any constant-valued expression).

- The enumeration values are coerced to int when they are put in integer context.
- For example, if the current value of myColor is blue, then the expression myColor++ would assign green to myColor.

- myColor = 4; is illegal in C++

- In ML, enumeration types are defined as new types with datatype declarations.
- For example, we could have the following:

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

- The types 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 (|).

➢Evaluation of Enumerated Type

- Enumeration types can provide advantages in both readability and reliability.

- **Readability** is enhanced very directly: Named values are easily recognized, whereas coded values are not.

- In the area of **reliability**, the enumeration types of Ada, C#, F#, and Java 5.0 provide two advantages:
  - (1) No arithmetic operations are legal on enumeration types; this prevents adding days of the week, for example, and
  - (2) second, no enumeration variable can be assigned a value outside its defined range. If the colors enumeration type has 10 enumeration constants and uses 0..9 as its internal values, no number greater than 9 can be assigned to a colors type variable.

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

2.Subrange Types

- A subrange type is a contiguous subsequence of an ordinal type.
- For example, 12..14 is a subrange of integer type.

- Subrange types were introduced by Pascal and are included in Ada.

➢ **Ada's Design:**

- In Ada, subranges are included in the category of types called subtypes.

- For example, consider the following declarations:

  type Days **is** (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

  subtype Weekdays **is** Days **range** Mon..Fri;

  subtype Index **is** Integer **range** 1..100;

- In these examples, the restriction on the existing types is in the range of possible values.
- All of the operations defined for the parent type are also defined for the subtype, except assignment of values outside the specified range.
- For example, in

```
Day1 : Days;
Day2 : Weekdays;
. . .
Day2 := Day1;
```

the assignment is legal unless the value of Day1 is Sat or Sun.

Subrange Evaluation:

- Subrange types enhance **readability**

  by making it clear to readers that variables of subtypes can store only certain ranges of values.

- **Reliability** is increased with subrange types,

  because assigning a value to a subrange variable that is outside the specified range is detected as an error, either by the compiler (in the case of the assigned value being a literal value) or by the run-time system (in the case of a variable or expression).

Implementation of User-Defined Ordinal Types

- Enumeration types are usually implemented as integers.
- Without restrictions on ranges of values and operations, this provides no increase in reliability.

- Subrange types are implemented in exactly the same way as their parent types.
- except that range checks must be implicitly included by the compiler in every assignment of a variable or expression to a subrange variable.

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.

➢Design Issues

The primary design issues specific to arrays are the following:

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does array allocation take place?
- Are ragged or rectangular multidimensioned arrays allowed, or both?
- Can arrays be initialized when they have their storage allocated?
- What kinds of slices are allowed, if any?

➢Array Indexing

- Specific elements of an array are referenced by means of a two-level syntactic mechanism, where the first part is the aggregate **name**, and the second part is a possibly dynamic selector consisting of one or more items known as **subscripts or indices**.
- Indexing (or subscripting) is a mapping from indices to elements

- The syntax of array references is fairly universal: The array name is followed by the list of subscripts, which is surrounded by either parentheses or brackets.
- array_name(subscript_value_list) → element

- Index Syntax
  - FORTRAN, PL/I, Ada use parentheses
    - Ada explicitly uses parentheses to show uniformity between array references and function calls because both are *mappings*
  - Most other languages use brackets

➢Arrays Index (Subscript) Types

- FORTRAN, C: integer only
- Ada: integer or enumeration (includes Boolean and char)
- Java: integer types only
- Index range checking

  - C, C++, Perl, and Fortran do not specify range checking

  - Java, ML, C# specify range checking

  - In Ada, the default is to require range checking, but it can be turned off

➢Subscript Binding and Array Categories

- 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.

- There are ***five categories of arrays***, based on the binding to subscript ranges, the binding to storage, and from where the storage is allocated.

- **Static array:** the subscript ranges are statically bound and storage allocation is static (done before run time).
  - The advantage of static arrays is efficiency: 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.
- **Fixed stack-dynamic array:** the subscript ranges are statically bound, but the allocation is done at declaration elaboration time during execution.
  - The advantage of fixed stack-dynamic arrays over static arrays is space efficiency.
  - The disadvantage is the required allocation and deallocation time.

- **Stack-dynamic array:** both the subscript ranges and the storage allocation are dynamically bound at elaboration time.
  - Advantage: flexibility (the size of an array need not be known until the array is to be used)
- **Fixed heap-dynamic array:** similar to a fixed stack-dynamic array, in that the subscript ranges and the storage binding are both fixed after storage is allocated. (i.e., binding is done when requested and storage is allocated from heap, not stack)
  - The *advantage* of fixed heap-dynamic arrays is flexibility—the array's size always fits the problem.
  - The *disadvantage* is allocation time from the heap, which is longer than allocation time from the stack.

- **Heap-dynamic array:** the binding of subscript ranges and storage allocation is dynamic and can change any number of times during the array's lifetime.
  - The *advantage* of heap-dynamic arrays over the others is flexibility: Arrays can grow and shrink during program execution as the need for space changes.
  - The *disadvantage* is that allocation and deallocation take longer and may happen many times during execution of the program.

- Arrays declared in C and C++ functions that include the static modifier are static.
- Arrays that are declared in C and C++ functions (without the static specifier) are examples of fixed stack-dynamic arrays.
- C and C++ also provide fixed heap-dynamic arrays.
- Perl, JavaScript, Python, and Ruby support heap-dynamic arrays

➢Array Initialization

- Some languages provide the means to initialize arrays at the time their storage is allocated.
  - C, C++, Java, C# example

```
int list [] = {4, 5, 7, 83}
```

  - Character strings in C and C++

```
char name [] = “freddie”;
```

  - Arrays of strings in C and C++

```
char *names [] = {“Bob”, “Jake”, “Joe”];
```

  - In Java, similar syntax is used to define and initialize an array of references to String objects.

```
String[] names = {“Bob”, “Jake”, “Joe”};
```

- Ada provides two mechanisms for initializing arrays in the declaration statement:
  - by listing them in the order in which they are to be stored,

  or

  - by directly assigning them to an index position using the => operator, which in Ada is called an arrow.
  - For example, consider the following:

    List : array (1..5) of Integer := (1, 3, 5, 7, 9);

    Bunch : array (1..5) of Integer := (1 => 17, 3 => 34, others => 0);

  - In the first statement, all the elements of the array List have initializing values, which are assigned to the array element locations in the order in which they appear. In the second, the first and third array elements are initialized using direct assignment, and the others clause is used to initialize the remaining elements.

➢Array Operations

- The most common array operations are assignment, catenation, comparison for equality and inequality, and slices.
- The C-based languages do not provide any array operations, except through the methods of Java, C++, and C#.
- Perl supports array assignments but does not support comparisons.
- Ada allows array assignments. Ada also provides catenation.
- F# includes many array operators in its Array module. Among these are Array.append, Array.copy, and Array.length.

- APL provides the most powerful array processing operations.
  - In APL, the four basic arithmetic operations are defined for vectors (single-dimensioned arrays) and matrices, as well as scalar operands.
  - For example, A + B is a valid expression, whether A and B are scalar variables, vectors, or matrices.

➢Rectangular and Jagged Arrays

- A **rectangular array** is a multidimensioned array in which all of the rows have the same number of elements and all of the columns have the same number of elements.
- Rectangular arrays model rectangular tables exactly.

- A **jagged array** is one in which the lengths of the rows need not be the same.
- For example, a jagged matrix may consist of three rows, one with 5 elements, one with 7 elements, and one with 12 elements
- This also applies to the columns and higher dimensions.

- C, C++, and Java support jagged arrays but not rectangular arrays.
- Fortran, Ada, C#, and F# support rectangular arrays. (C# and F# also support jagged arrays.)

➢Slices

- A slice of an array is some substructure of that array. For example, if A is a matrix, then the first row of A is one possible slice.

- It is important to realize that a slice is not a new data type. Rather, it is a mechanism for referencing part of an array as a unit.

- Consider the following Python declarations:

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

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

  - Recall that the default lower bound for Python arrays is 0.
  - The syntax of a Python slice reference is a pair of numeric expressions separated by a colon.
  - The first is the first subscript of the slice; the second is the first subscript after the last subscript in the slice.
  - Therefore, vector[3:6] is a three-element array with the fourth through sixth elements of vector (those elements with the subscripts 3, 4, and 5).
  - A row of a matrix is specified by giving just one subscript. For example, mat[1] refers to the second row of mat;
  - a part of a row can be specified with the same syntax as a part of a single dimensioned array. For example, mat[0][0:2] refers to the first and second element of the first row of mat, which is [1, 2].

- Ruby supports slices with the `slice` method
  - For example, suppose list is defined as follows:

    list = [2, 4, 6, 8, 10]
  - list.slice(2, 2) returns the third and fourth elements of `list.` That is [6, 8]

- The third parameter form for slice is a range, which has the form of an integer expression, two periods, and a second integer expression.
- With a range parameter, slice returns an array of the element with the given range of subscripts.
  - For example, list.slice (1..3) returns [4, 6, 8]

➢Evaluation of Array Types

- Arrays have been included in virtually all programming languages.
- The primary advances since their introduction in Fortran I have been the inclusion of all ordinal types as possible subscript types, slices, and, of course, dynamic arrays.
- The latest advances in arrays have been in associative arrays.

➢Implementation of Array Types

- Implementing arrays requires considerably more compile-time effort than does implementing primitive types.

- A single-dimensioned array is implemented as a list of adjacent memory cells.
  - Suppose the array list is defined to have a subscript range lower bound of 0.
  - The access function for list is often of the form

    address(list[k]) = address(list[0]) + k * element_size
- The generalization of this access function for an arbitrary lower bound is

  address(list[k]) = address(list[lower_bound]) + ((k - lower_bound) * element_size)

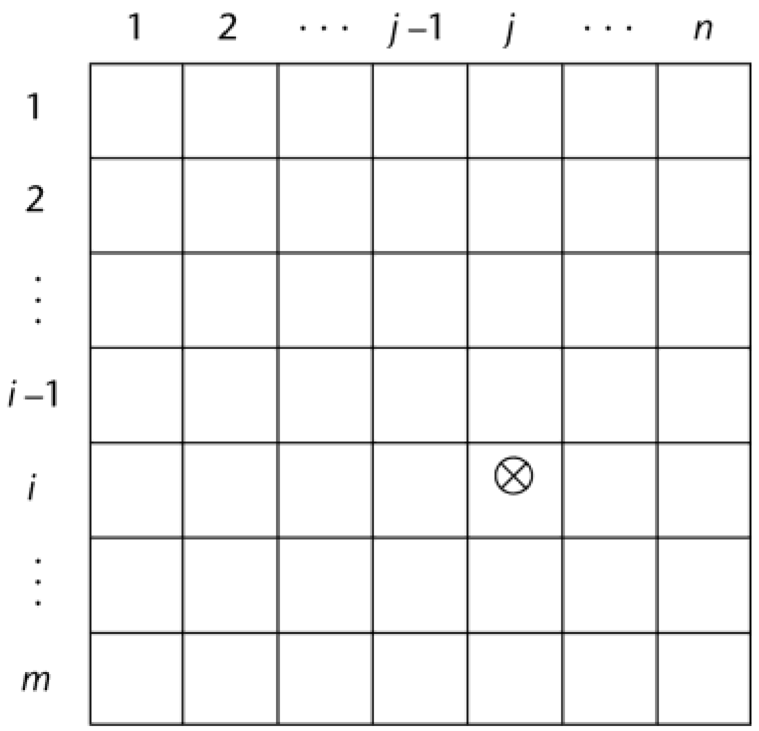
- Accessing Multi-dimensioned Arrays, two common ways:
  - Row major order (by rows) – used in most languages
  - column major order (by columns) – used in Fortran
- In **row major order**, the elements of the array that have as their first subscript the lower bound value of that subscript are stored first, followed by the elements of the second value of the first subscript, and so forth.
  - If the array is a matrix, it is stored by rows.
  - 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
- In **column major order**, the elements of an array that have as their last subscript the lower bound value of that subscript are stored first, followed by the elements of the second value of the last subscript, and so forth.
  - If the array is a matrix, it is stored by columns.
  - If the example matrix were stored in column major order, it would have the following order in memory: 3, 6, 1, 4, 2, 3, 7, 5, 8

- Locating an Element in a Multi-dimensioned Array:



The location of the [i,j] element in a matrix

- The access function can be written as

  location(a[i,j]) = address of a[0, 0] + ((((number of rows above the ith row) * (size of a row)) + (number of elements left of the jth column)) * element size)

- Because the number of rows above the ith row is i and the number of elements to the left of the jth column is j, we have

  location(a[i, j]) = address of a[0, 0] + (((i * n) + j) * element_size)

- The generalization to arbitrary lower bounds results in the following access function:

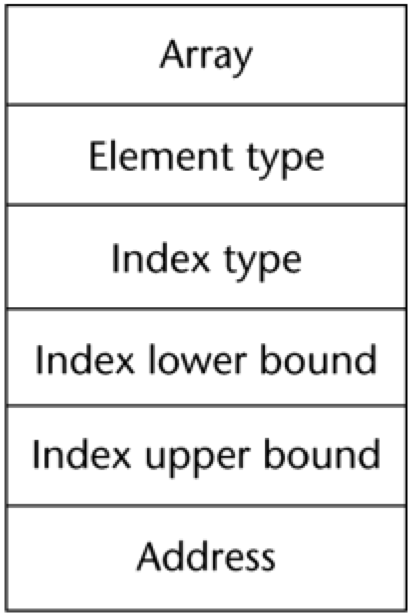
  location(a[i, j]) = address of a[row_lb, col_lb] + (((i - row_lb) * n) + (j - col_lb)) * element_size

  where row_lb is the lower bound of the rows and col_lb is the lower bound of the columns.

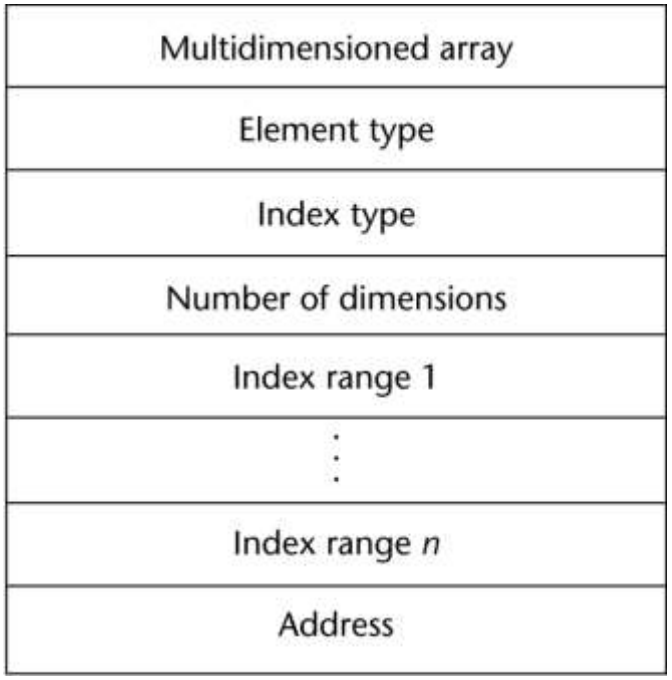
- This can be rearranged to the form

  location(a[i, j]) = address of a[row_lb, col_lb] - (((row_lb * n) + col_lb) * element_size) + (((i * n) + j) * element_size)

- Compile-Time Descriptors:



Single-dimensioned array



Multi-dimensional array

Record Types

- A record is a heterogeneous aggregate of data elements in which the individual elements are identified by names and accessed through offsets from the beginning of the structure.

- For example, information about a college student might include name, student number, grade point average, and so forth. A data type for such a collection might use a character string for the name, an integer for the student number, a floatingpoint for the grade point average, and so forth. Records are designed for this kind of need.

- The following design issues are specific to records:
  - What is the syntactic form of references to fields?
  - Are elliptical references allowed?

➢Definitions of Records

- The fundamental difference between a record and an array is that record elements, or fields, are not referenced by indices.
- Instead, the fields are named with identifiers, and references to the fields are made using these identifiers.

- 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
- Example:

```
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 EMPLOYEE-RECORD record consists of the EMPLOYEE-NAME record and the HOURLY-RATE field.
- The numerals 01, 02, and 05 that begin the lines of the record declaration are level numbers, which indicate by their relative values the hierarchical structure of the record. Any line that is followed by a line with a higher-level number is itself a record.
- The PICTURE 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 in Ada:

- Record structures are indicated in an orthogonal way
- In Ada, records must be named types.

- Example:

```
type Employee_Name_Type is record
        First : String (1..20);
        Middle : String (1..10);
        Last : String (1..20);
end record;
type Employee_Record_Type is record
        Employee_Name: Employee_Name_Type;
        Hourly_Rate: Float;
end record;
Employee_Record: Employee_Record_Type;
```

- These statements create a table (record) named employee with two elements (fields) named Employee_Name and Hourly_Rate.

➢References to Record Fields

- References to the individual fields of records are syntactically specified by several different methods, two of which name the desired field and its enclosing records.

- COBOL field references have the form
  - field_name OF record_name_1 OF . . . OF record_name_n
  - where the first record named is the smallest or innermost record that contains the field. The next record name in the sequence is that of the record that contains the previous record, and so forth.

- For example, the MIDDLE field in the COBOL record example above can be referenced with MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE-RECORD

- Most of the other languages use dot notation for field references, where the components of the reference are connected with periods.
  - For example, the following is a reference to the field Middle in the earlier Ada record example: Employee_Record.Employee_Name.Middle
  - C and C++ use this same syntax for referencing the members of their structures.

- A **fully qualified reference** to a record field is one in which all intermediate record names, from the largest enclosing record to the specific field, are named in the reference.

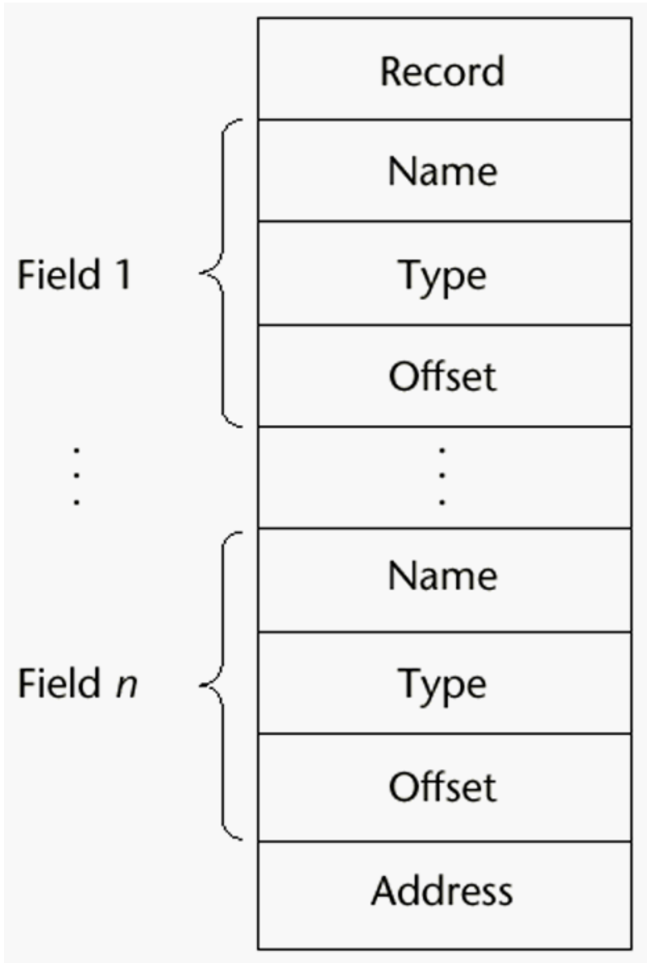
- In an **elliptical reference**, the field is named, but any or all of the enclosing record names can be omitted, as long as the resulting reference is unambiguous in the referencing environment.
  - For example, FIRST, FIRST OF EMPLOYEE-NAME, and FIRST OF EMPLOYEE-RECORD are elliptical references to the employee's first name in the COBOL record declared above.

➢Evaluation and Comparison to Arrays

- Arrays are used when all the data values have the same type and/or are processed in the same way.
- Records are used when collection of data values is heterogeneous and the different fields are not processed in the same way.
- 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 Types

- The fields of records are stored in adjacent memory locations.
- The sizes of the fields are not necessarily the same.
- The offset address, relative to the beginning of the record, is associated with each field. Field accesses are all handled using these offsets.
- The compile-time descriptor for a record has the general form as:
- Run-time descriptors for records are unnecessary.



A compile-time descriptor for a record

List Types

- Lists were first supported in the first functional programming language, LISP.

- Lists in Scheme and Common LISP are delimited by parentheses and the elements are not separated by any punctuation.
    - For example, (A B C D)
- Nested lists have the same form, so we could have

        (A (B C) D)

    - In this list, (B C) is a list nested inside the outer list.

- If the list (A B C) is interpreted as code, it is a call to the function A with parameters B and C.

The fundamental list operations in Scheme are two functions :

- The **CAR function** returns the first element of its list parameter.
  - For example, consider the following example:

    ```
    (CAR '(A B C))
    ```

  - The quote before the parameter list is to prevent the interpreter from considering the list a call to the A function with the parameters B and C, in which case it would interpret it.
  - This call to CAR returns A.
- The **CDR function** returns its parameter list minus its first element.
  - For example, consider the following example:

    ```
    (CDR '(A B C))
    ```

  - This function call returns the list (B C).

- New lists are constructed with the CONS and LIST functions.
- The function **CONS** takes two parameters and returns a new list with its first parameter as the first element and its second parameter as the remainder of that list.
  - For example, consider the following:

    ```
    (CONS 'A '(B C))
    ```

  - This call returns the new list (A B C).
- The **LIST** function takes any number of parameters and returns a new list with the parameters as its elements.
  - For example, consider the following call to LIST:

    ```
    (LIST 'A 'B '(C D))
    ```

  - This call returns the new list (A B (C D)).

Pointer and Reference Types

- A pointer type is one in which the variables have 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.

- Pointers are designed for two distinct kinds of uses.
  - First, pointers provide some of the power of **indirect addressing**, which is frequently used in assembly language programming.
  - Second, pointers provide a way to **manage dynamic storage**. A pointer can be used to access a location in an area where storage is dynamically allocated called a heap.

- Variables that are dynamically allocated from the heap are called **heap dynamic variables**. They often do not have identifiers associated with them and thus can be referenced only by pointer or reference type variables.
- Variables without names are called **anonymous variables**.
- **heap dynamic variables** and **anonymous variables** are accessed using Pointers

➢Design Issues of Pointers

- What are the scope and lifetime of a pointer variable?
- What is the lifetime of a heap-dynamic variable (the value a pointer references)?
- 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?

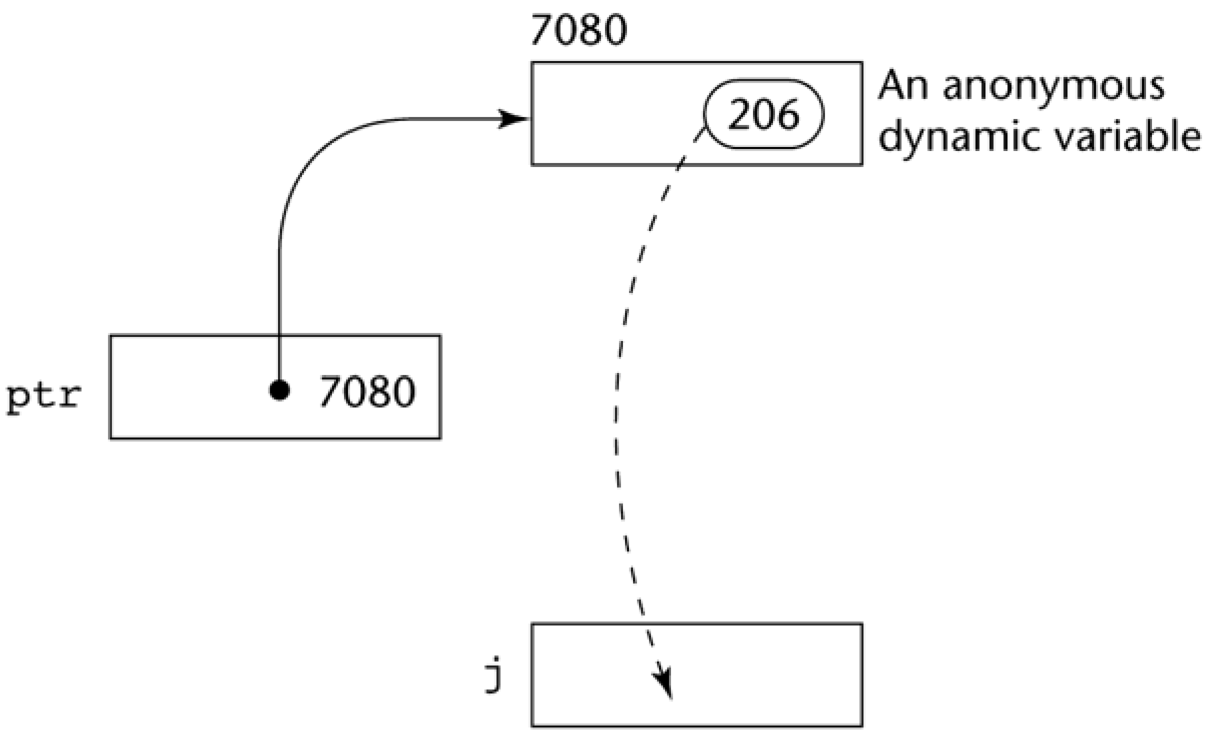
➢Pointer Operations

- 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 C++, it is explicitly specified with the asterisk (*) as a prefix unary operator.

- Consider the following example of dereferencing:

  j = *ptr

  sets j to the value located at ptr



If ptr is a pointer variable with the value 7080 and the cell whose address is 7080 has the value 206, then the assignment

➢Problems with Pointers

- **Dangling pointers**
  - A pointer that contains the address of a heap-dynamic variable that has been deallocated.
  - Dangling pointers are dangerous for several reasons.
    - First, the location being pointed to may have been reallocated to some new heap-dynamic variable. If the new variable is not the same type as the old one, type checks of uses of the dangling pointer are invalid.
    - Even if the new dynamic variable is the same type, its new value will have no relationship to the old pointer's dereferenced value. Furthermore, if the dangling pointer is used to change the heap-dynamic variable, the value of the new heap-dynamic variable will be destroyed.
  - The following sequence of operations creates a dangling pointer in many languages:

    1. A new heap-dynamic variable is created and pointer p1 is set to point at it.

    2. Pointer p2 is assigned p1's value.

    3. The heap-dynamic variable pointed to by p1 is explicitly deallocated (possibly setting p1 to nil), but p2 is not changed by the operation. p2 is now a dangling pointer. If the deallocation operation did not change p1, both p1 and p2 would be dangling. (Of course, this is a problem of aliasing—p1 and p2 are aliases.

- **Lost heap-dynamic variable(**often called *garbage***)**
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)
  - Lost heap-dynamic variables are most often created by the following sequence of operations:
    1. Pointer `p1` is set to point to a newly created heap-dynamic variable
    2. 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*
    - Memory leakage is a problem, regardless of whether the language uses implicit or explicit deallocation.

➢Pointers in Ada

- Ada's pointers are called access types.
- The dangling-pointer problem is partially alleviated by Ada's design, at least in theory.
- A heap-dynamic variable may be (at the implementor's option) implicitly deallocated at the end of the scope of its pointer type.
- The Ada language also has an explicit deallocator, Unchecked_Deallocation.
  - Unchecked_Deallocation can cause dangling pointers.
- The lost heap-dynamic variable problem is not eliminated by Ada's design of pointers.

➢Pointers in C and C++

- Extremely flexible but must be used with care. This design offers no solutions to the dangling pointer or lost heap-dynamic variable problems.
- 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
- C and C++ include pointers of type **void ***, which can point at values of any type.
  - Type checking is not a problem with void * pointers, because these languages disallow dereferencing them.
  - One common use of void * pointers is as the types of parameters of functions that operate on memory.

- In C and C++, the asterisk (*) denotes the dereferencing operation, and the ampersand (&) denotes the operator for producing the address of a variable.
- For example, consider the following code:

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

  - The assignment to the variable ptr sets it to the address of init.
  - The assignment to count dereferences ptr to produce the value at init, which is then assigned to count.
  - So, the effect of the two assignment statements is to assign the value of init to count.
- The two assignment statements above are equivalent in their effect on count to the single assignment
- count = init;

➢Reference Types

- A reference type variable is similar to a pointer, with one important and fundamental difference: A pointer refers to an address in memory, while a reference refers to an object or a value in memory.
- Reference type variables are specified in definitions by preceding their names with ampersands (&).

  For example,

  ```
  int result = 0;
  int &ref_result = result;
  . . .
  ref_result = 100;
  ```

  In this code segment, result and ref_result are aliases.

- C++ includes a special kind of reference type called a *reference type* that is used primarily for formal parameters
  - Advantages of both pass-by-reference and pass-by-value
- Java extends C++'s reference variables and allows them to replace pointers entirely
  - References are references to objects, rather than being addresses
- C# includes both the references of Java and the pointers of C++

➢Evaluation of Pointers

- Dangling pointers and garbage are problems as is heap management
- Pointers have been compared with the goto.
  - The goto statement widens the range of statements that can be executed next.
  - Pointer variables widen the range of memory cells that can be referenced by a variable.
- Pointers or references are necessary for dynamic data structures--so we can't design a language without them

➢Implementation of Pointer and Reference Types

- In most languages, pointers are used in heap management.
- The same is true for Java and C# references, as well as the variables in Smalltalk and Ruby, so we cannot treat pointers and references separately.

1. Representations of Pointers and References

- In most larger computers, pointers and references are single values stored in memory cells.
- However, in early microcomputers based on Intel microprocessors, addresses have two parts: a segment and an offset.
  - So, pointers and references are implemented in these systems as pairs of 16-bit cells, one for each of the two parts of an address.

2. Solutions to the Dangling-Pointer Problem

- There have been several proposed solutions to the dangling-pointer problem.

➢ **Tombstones** :

- 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 and never to heap-dynamic variables.
- When a heap-dynamic variable is deallocated, the tombstone remains but is set to nil, indicating that the heap-dynamic variable no longer exists.
- This approach prevents a pointer from ever pointing to a deallocated variable. Any reference to any pointer that points to a nil tombstone can be detected as an error.
- Tombstones are costly in both time and space. Because tombstones are never deallocated, their storage is never reclaimed.

- **locks-and-keys approach**:

- pointer values are represented as ordered pairs (key, address), where the key is an integer value.
- Heap-dynamic variables are represented as the storage for the variable plus a header cell that stores an 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.
- 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.

3. Heap Management

- Heap management can be a very complex run-time process.
- We examine the process in two separate situations:
- one in which all heap storage is allocated and deallocated in units of a *single size*,
- and one in which *variable-size* segments are allocated and deallocated.
- Two approaches to reclaim garbage
  - **Reference counters (*eager approach*):** reclamation is incremental and is done when inaccessible cells are created
  - **Mark-sweep (*lazy approach*):** reclamation occurs only when the list of variable space becomes empty

- Many variations of these two approaches have been developed.
- The **reference counter method** of storage reclamation accomplishes its goal by maintaining in every cell a counter that stores the number of pointers that are currently pointing at the cell.
- *Disadvantage*: There are three distinct problems with the reference counter method.
    - First, if storage cells are relatively small, the space required for the counters is significant.
    - Second, some execution time is obviously required to maintain the counter values.
- Some of the inefficiency of reference counters can be eliminated by an approach named **deferred reference counting**, which avoids reference counters for some pointers.
    - The third problem is that complications arise when a collection of cells is 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.
- *Advantage*: it is intrinsically incremental, so significant delays in the application execution are avoided

- The **mark-sweep process** of garbage collection operates as follows:
- The run-time system allocates storage cells as requested and disconnects pointers from cells as necessary, without regard for storage reclamation (allowing garbage to accumulate), until it has allocated all available cells.
- The mark-sweep process consists of three distinct phases.
  - First, all cells in the heap have their indicators set to indicate they are garbage.
  - The second part, called the ***marking phase***, is the most difficult. Every pointer in the program is traced into the heap, and all reachable cells are marked as not being garbage.
  - After this, 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.
- Disadvantages: in its original form, it was done too infrequently. When done, it caused significant delays in application execution. Contemporary mark-sweep algorithms avoid this by doing it more often—called incremental mark-sweep

Variable-Size Cells:

- Managing a heap from which variable-size cells are allocated has all the difficulties of managing one for single-size cells, but also has additional problems.
- The additional problems posed by variable-size cell management depend on the method used.
- If mark-sweep is used, the following additional problems occur:
  - The initial setting of the indicators of all cells in the heap to indicate that they are garbage is difficult.
  - The marking process is nontrivial.
  - Maintaining the list of available space is another source of overhead.

Type Checking

- Type checking is the activity of ensuring that the operands of an operator are of compatible types.
- A compatible type is one that either is legal for the operator or is allowed under language rules to be implicitly converted by compiler-generated code (or the interpreter) 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.
- 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

- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic(type checking at run time)
- Type checking is complicated when a language allows a memory cell to store values of different types at different times during execution.

Strong Typing

- A programming language is strongly typed if type errors are always detected.
- This requires that the types of all operands can be determined, either at compile time or at run time.
- Advantage of strong typing: its ability to detect all misuses of variables that result in type errors.
- A strongly typed language also allows the detection, at run time, of uses of the incorrect type values in variables that can store values of more than one type.

Language examples:

- Ada is nearly strongly typed, (`UNCHECKED CONVERSION` is loophole)
- Java and C#, although they are based on C++, are strongly typed in the same sense as Ada.
- C and C++ are not strongly typed languages because both include union types, which are not type checked.
- ML is strongly typed, even though the types of some function parameters may not be known at compile time.
- F# is strongly typed.

Type Equivalence

- Type equivalence is a strict form of type compatibility—compatibility without coercion.
- There are two approaches to defining type equivalence: name type equivalence and structure type equivalence.
- **Name type equivalence** means that two variables have equivalent types if they are defined either in the same declaration or in declarations that use the same type name.
  - Easy to implement but highly restrictive:
    - Subranges of integer types are not equivalent with integer types
    - Formal parameters must be the same type as their corresponding actual parameters

- **Structure type equivalence** means that two variables have equivalent types if their types have identical structures.
  - More flexible, but harder to implement
    - Under name type equivalence, only the two type names must be compared to determine equivalence. Under structure type equivalence, however, the entire structures of the two types must be compared. This comparison is not always simple.
  - Another difficulty with structure type equivalence is that it disallows differentiating between types with the same structure

- There are some variations of these two approaches, and many languages use combinations of them.

- **Ada** uses a restrictive form of name type equivalence but provides two type constructs, subtypes and derived types, that avoid the problems associated with name type equivalence.
- **C** uses both name and structure type equivalence.
- In languages that do not allow users to define and name types, such as **Fortran** and **COBOL**, name equivalence obviously cannot be used.