DATA TYPES

DATA TYPES

- A **data type** defines a collection of data values and a set of predefined operations on those values. Computer programs produce results by manipulating data.
- It is crucial that a language supports an appropriate collection of data types and structures.

Primitive Data Types

- Data types that are not defined in terms of other types are called **primitive data types**. All programming languages provide a set of primitive data types. Some of the primitive types are merely reflections of the hardware—for example, most integer types.
- **Numeric Types**
- Many early programming languages had only numeric primitive types. Numeric types still play a central role among the collections of types supported by contemporary languages.

Integer

- The most common primitive numeric data type is **`integer`**. Many computers now support several sizes of integers. These sizes of integers, and often a few others, are supported by some programming languages. For example, Java includes four signed integer sizes: **`byte`**, **`short`**, **`int`**, and **`long`**.
- 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.
- Integer arithmetic operations in Python that produce values too large to be represented with `int` type store them as long integer type values.
- 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.
- Ones-complement notation has the disadvantage that it has two representations of zero.

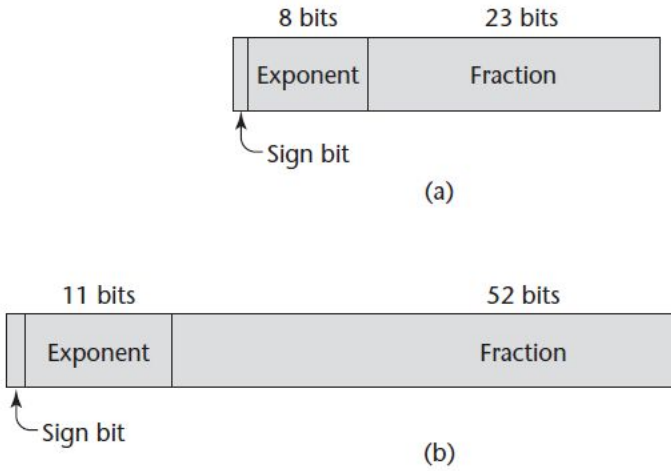
Floating-Point

- **Floating-point** data types model real numbers, but the representations are only approximations for many real values. For example, neither of the fundamental numbers or *e* (the base for the natural logarithms) can be correctly represented in floating-point notation.
- On most computers, floatingpoint numbers are stored in binary.
- Another problem with floating-point types is the loss of accuracy through arithmetic operations.
- Floating-point values are represented as fractions and exponents, a form that is borrowed from scientific notation.
- Most languages include two floating-point types, often called **float** and stored in four bytes of memory. The double type is provided for situation exponents is needed.
- The collection of values that can be represented by a floating-point type is defined in terms of precision and range.

Precision is the accuracy of the fractional part of a value, measured as the number of bits. **Range** is a combination of the range of fractions and, more important, the range of exponents.

Figure 6.1

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



Complex

- Some programming languages support a complex data type—for example, Fortran and Python. Complex values are represented as ordered pairs of floating-point values. In Python, the imaginary part of a complex literal is specified by following it with a `j` or `J`—for example, `(7 + 3j)`

Decimal

- Most larger computers that are designed to support business systems applications have hardware support for **decimal** data types. 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.
- Decimal types have the advantage of being able to precisely store decimal values, at least those within a restricted range, which cannot be done with floating-point.
- Decimal types are stored very much like character strings, using binary codes for the decimal digits. These representations are called **binary coded decimal (BCD).**
- The disadvantages of decimal types are that the range of values is restricted because no exponents are allowed.

- **Boolean Types**
- **Boolean** types are perhaps the simplest of all types. Their range of values has only two elements: one for true and one for false.
- They were introduced in ALGOL 60.
- Boolean types are often used to represent switches or flags in programs. A Boolean value could be represented by a single bit.

Character Types

- Character data are stored in computers as numeric codings. The 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.

Character String Types

- A **character string type** is one in which the values consist of sequences of characters.

Design Issues

- The two most important design issues that are specific to character string types are the following:
- Should strings be simply a special kind of character array or a primitive type?
- Should strings have static or dynamic length?

- **Strings and Their Operations**
- The most common string operations are assignment, catenation, substring reference, comparison, and pattern matching.
- A **substring reference** is a reference to a substring of a given string. Substring references are discussed in the more general context of arrays, where the substring references are called **slices**.
- Pattern matching is another fundamental character string operation. In some languages, pattern matching is supported directly in the language. In others, it is provided by a function or class library.
- If strings are not defined as a primitive type, string data is usually stored in arrays of single characters and referenced as such in the language. This is the approach taken by C and C++. C and C++ use **`char`** arrays to store character strings.
- C and C++ use **`char`** arrays to store character strings. These languages provide a collection of string operations through standard libraries.
- For example, consider the following declaration:

```
char str[] = "apples";
```

- In this example, `str` is an array of **`char`** elements, specifically `apples0`, where `0` is the null character.
- Some of the most commonly used library functions for character strings in C and C++ are

  `strcpy`, which moves strings;

  `strcat`, which catenates one given string onto another;

  `strcmp`, which lexicographically compares (by the order of their character codes) two given strings;

  `strlen`, which returns the number of characters, not counting the null, in the given string.

- In Java, strings are supported by the `String` class, whose values are constant strings, and the `StringBuffer` class, whose values are changeable and are more like arrays of single characters.
- In F#, strings are a class. Individual characters, which are represented in Unicode UTF-16, can be accessed, but not changed. Strings can be catenated with the `+` operator.
- In ML, string is a primitive immutable type. It uses `^` for its catenation operator and includes functions for substring referencing and getting the size of a string
- Perl, JavaScript, Ruby, and PHP include built-in pattern-matching operations.In these languages, the pattern-matching expressions are called **regular expressions.**
- Consider the following pattern expression:

```
/[A-Za-z][A-Za-z\d]+/
```

- This pattern matches (or describes) the typical name form in programming languages. The brackets enclose character classes. The first character class specifies all letters; the second specifies all letters and digits (a digit is specified with the abbreviation `\d`)
- consider the following pattern expression:

```
/\d+\.?\d*|\.\d+/
```

- This pattern matches numeric literals. The `\.` specifies a literal decimal point. The question mark quantifies what it follows to have zero or one appearance.
- The vertical bar (`|`) separates two alternatives in the whole pattern. The first alternative matches strings of one or more digits, possibly followed by a decimal point, followed by zero or more digits;
- the second alternative matches strings that begin with a decimal point, followed by one or more digits.

String Length Options

- First, the length can be static and set when the string is created. Such a string is called a **static length string**.
- This is the choice for the strings of Python, the immutable objects of Java's `String` class, as well as similar classes in the C++.
- The second option is to allow strings to have varying length up to a declared and fixed maximum set by the variable's definition, as exemplified by the strings in C and the C-style strings of C++. These are called **limited dynamic length strings**. Such string variables can store any number of characters between zero and the maximum.
- The third option is to allow strings to have varying length with no maximum, as in JavaScript, Perl, and the standard C++ library. These are called **dynamic length strings**. This option requires the overhead of dynamic storage allocation and deallocation but provides maximum flexibility.
- Ada 95+ supports all three string length options.

Implementation of Character String Types

- A descriptor for a static character string type, which is required only during compilation, has three fields. 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.
- Dynamic length strings require a simpler run-time descriptor because only the current length needs to be stored.

- 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 dynamic

- There **<u>are three approaches</u>** to supporting the dynamic allocation and deallocation that is required for dynamic length strings.

Figure 6.2

Compile-time descriptor for static strings

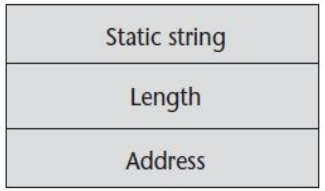
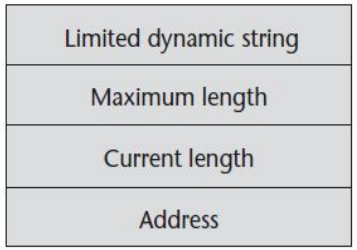


Figure 6.3

Run-time descriptor for limited dynamic strings



1) First, strings can be stored in a linked list, so that when a string grows, the newly required cells can come from anywhere in the heap. The drawbacks to this method are the extra storage occupied by the links in the list representation and the necessary complexity of string operations.

2) The second approach is to store strings as arrays of pointers to individual characters allocated in the heap. This method still uses extra memory, but string processing can be faster.

3) The third alternative is to store complete strings in adjacent storage cells. The problem with this method arises when a string grows:

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.
- There are two user-defined ordinal types that have been supported by programming languages: **enumeration and subrange.**

Enumeration Types

- An **enumeration type** is one in which all of the possible values, which are named constants, are provided, or enumerated, in the definition.
- Enumeration types provide a way of defining and grouping collections of named constants, which are called **enumeration constants**.
- C# example:

```
enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

- The enumeration constants are typically implicitly assigned the integer values, 0, 1, . . .

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 in the program checked?

- • Are enumeration values coerced to integer?
- • Are any other types coerced to an enumeration type?

Designs

- 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 . 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++,

```
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.
- For example, if the current value of `myColor` is `blue`, then the expression

`myColor++` would assign `green` to `myColor`

- For example,

`myColor = 4;` **is illegal in C++.**

- C++ enumeration constants can appear in only one enumeration type in the same referencing environment

- In Ada, enumeration literals are allowed to appear in more than one declaration in the same referencing environment. These are called **overloaded literals.**
- 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
```

- F# has enumeration types that are similar to those of ML, except the reserved word **`type`** is used instead of **`datatype`**
- 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.

- For example,
- **`enum`** `colors {red = 1, blue = 1000, green = 100000}`
- In this example, a value assigned to a variable of `colors` type will only be checked to determine whether it is in the range of `1..100000`.

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.
- There are no design issues

Ada's Design

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

```
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

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

- the assignment is legal unless the value of `Day1` is `Sat` or `Sun`

Implementation of User-Defined Ordinal Types

- enumeration types are usually implemented as integers.
- 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. References to individual array elements are specified using subscript expressions.
- In many languages, such as C, C++, Java, Ada, and C#, all of the elements of an array are required to be 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?

Arrays and Indices

- 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**.
- If all of the subscripts in a reference are constants, the selector is static; otherwise, it is dynamic.
- arrays are sometimes called **finite mappings**.

  array_name(subscript_value_list) element

- For example, consider the following Ada assignment statement:

```
Sum := Sum + B(I);
```

- Because parentheses are used for both subprogram parameters and array subscripts in Ada, both program readers and compilers are forced to use other information to determine whether `B(I)` in this assignment is a function call or a reference to an array element. This results in reduced readability
- Most languages other than Fortran and Ada use brackets to delimit their array indices.
- Two distinct types are involved in an array type: the element type and the type of the subscripts.
- Subscripting in Perl is a bit unusual in that although the names of all arrays begin with at signs (`@`), because array elements are always scalars and the names of scalars always begin with dollar signs (`$`), references to array elements use dollar signs rather than at signs in their names. For example, for the array `@list`, the second element is referenced with `$list`[1].

Subscript Bindings and Array Categories

- There are five categories of arrays, based on the binding to subscript ranges, the binding to storage, and from where the storage is allocated.
- A **static array** is one in which 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.
- A **fixed stack-dynamic array** is one in which 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.
- A **stack-dynamic array** is one in which both the subscript ranges and the storage allocation are dynamically bound at elaboration time. The advantage of stack-dynamic arrays over static and fixed stack-dynamic arrays is flexibility.
- A **fixed heap-dynamic array** is similar to a fixed stack-dynamic array, in that the subscript ranges and the storage binding are both fixed after storage is allocated. The 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.

- A **heap-dynamic array** is one in which 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. 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.
- C++ uses the operators **`new`** and **`delete`** to manage heap storage.
- C# also provides generic heap-dynamic arrays, which are objects of the `List` class. These array objects are created without any elements, as in

```
List<String> stringList = new List<String>();
```

- Elements are added to this object with the `Add` method, as in

```
stringList.Add("Michael");
```

- A Perl array can be made to grow by using the `push` (puts one or more new elements on the end of the array) and `unshift` (puts one or more new elements on the beginning of the array)
- JavaScript arrays can be sparse, meaning the subscript values need not be contiguous.

Array Initialization

- In the C declaration `int list [] = {4, 5, 7, 83};`
- character strings in C and C++ are implemented as arrays of **`char`**. These arrays can be initialized to string constants, as in

```
char name [] = "freddie";
```

- The array `name` will have eight elements, because all strings are terminated with a null character (zero).
- Arrays of strings in C and C++ can also be initialized with string literals.

```
char *names [] = {"Bob", "Jake", "Darcie"};
```

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

```
String[] names = ["Bob", "Jake", "Darcie"];
```

Array Operations

- The most common array operations are assignment, catenation, comparison for equality and inequality, and slices.
- Ada allows array assignments, including those where the right side is an aggregate value rather than an array name. Ada also provides catenation, specified by the ampersand (`&`). Catenation is defined between two single dimensioned arrays and between a single-dimensioned array and a scalar.

- Python's arrays are called lists. Python provides array assignment, although it is only a reference change.
- Python also has operations for array catenation (+) and element membership (**`in`**). It includes two different comparison operators: one that determines whether the two variables reference the same object (**`is`**) and one that compares all corresponding objects in the referenced objects, for equality (==).
- F# includes many array operators in its `Array` module.

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

⌽V reverses the elements of V
⌽M reverses the columns of M
⊖M reverses the rows of M
⍉M transposes M (its rows become its columns and vice versa)
⌹M inverts M

APL also includes several special operators that take other operators as operands. One of these is the inner product operator, which is specified with a period (.). It takes two operands, which are binary operators. For example,

`+.×`

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.
- Fortran, Ada, C#, and F# support rectangular arrays. For example,

```
myArray[3, 7]
```

- 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. For example,

```
myArray[3][7]
```

Slices

- A **slice** of an array is some substructure of that array. It is important to realize that a slice is not a new data type. Consider the following Python declarations:

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
mat = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]
```

- In Python `vector[3:6]` is a three-element array with the fourth through sixth elements of `vector`.
- Perl supports slices of two forms, a list of specific subscripts or a range of subscripts. For example,

```
@list[1..5] = @list2[3, 5, 7, 9, 13];
```

- Ruby supports slices with the `slice` method of its `Array` object, For example, suppose `list` is defined as follows: `list = [2, 4, 6, 8, 10]`

`list.slice(2, 2)` returns `[6, 8]`

Implementation of Array Types

- There is no way to precompute the address to be accessed by a reference such as

```
list[k]
```

- 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)

The compile-time descriptor for single-dimensioned arrays

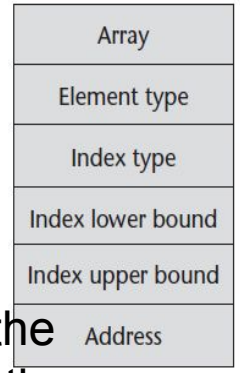


Figure 6.4
Compile-time descriptor for single-dimensioned arrays

There are two ways in which multidimensional arrays can be mapped to one dimension: row major order and column major order.
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

- the access function can be written as

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

- Because the number of rows above the `i`*th* row is `i` and the number of elements to the left of the `j`*th* column is `j`, we have

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

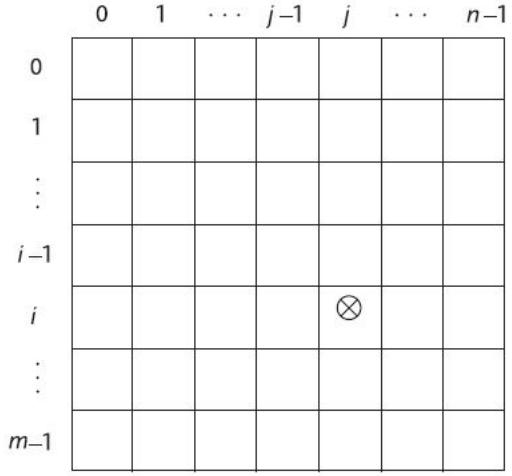
- The generalization to arbitrary lower bounds results in the following access function:

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.

Figure 6.5

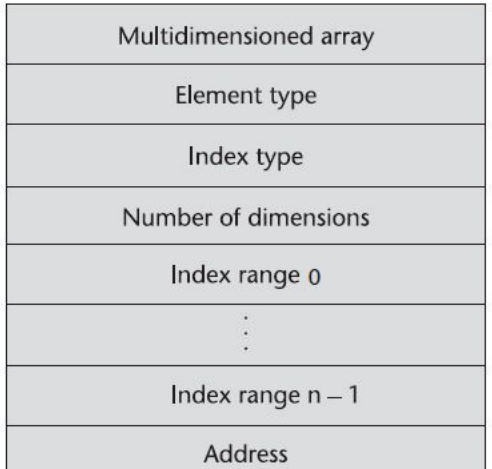
The location of the `[i,j]` element in a matrix



The compile-time descriptor for a multidimensional array

Figure 6.6

A compile-time descriptor for a multidimensional array



Eg 1:Given an array [1..8, 1..5, 1..7] of integers. Calculate address of element A[5,3,6], by using rows and columns methods, if BA=900?

Solution:-

A[i ,j,k]=Address of A+(i-L1)*S1+(j-L2)*S2 +(k-L3)*S3

Let ***i=5, j=3, k=6,L1=L2=L3=1, U1=8,U2=5,U3=7***

S3 = 4 (size of element type)
S2=(U3-L3+1)*S3=(7-1+1)*4=28

S1=(U2-L2+1)*S2=(5-1+1)*28=140

Location(A[5,3,6])= 900 +(5-1)*S1+(3-1)*S2+(6-1)*S3

=900+4*140+2*28+5*4

=1536

Eg:2 Given an array [1..8, 1..5, 1..7] of integers. Calculate address of element A[5,3,6], by using rows and columns methods, if BA=900?

Solution:- The dimensions of A are :

D1= U1-L1+1*=8-1+1=8 ,* ***D2= U2-L2+1****=5-1+1=5,* ***D3 = U3-L3+1****=7-1+1=7,*

i=5, j=3, k=6

Rows - wise :

Location (A[i,j,k]) = BA + D1D2(k-1) + D2(i-1) + (j-1)

Location(A[5,3,6])= 900 + 8x5(6-1) + 5(5-1) + (3-1)
= 900 + 40 x 5 +5 x 4 + 2
= 900 + 200 +20 +2
= 1122

Columns - wise :

Location (A[i,j,k]) = BA + D1D2(k-1) + D1(j-1) + (i-1)

Location (A[5,3,6]) = 900 + 8x5(6-1) + 8(3-1) + (5-1)
= 900 + 40 x 5 +8 x 2 + 4
= 900 + 200 +16 +4
= 1120

Associative Arrays

- An **associative array** is an unordered collection of data elements that are indexed by an equal number of values called **keys**.
- In the case of non-associative arrays, the indices never need to be stored (because of their regularity).
- In an associative array, however, the user-defined keys must be stored in the structure.
- The only design issue that is specific for associative arrays is the form of references to their elements.

Structure and Operations

- In Perl, associative arrays are called **hashes**, because in the implementation their elements are stored and retrieved with hash functions. The namespace for Perl hashes is distinct: Every hash variable name must begin with a percent sign (%).
- Each hash element consists of two parts: a key, which is a string, and a value, which is a scalar (number, string, or reference). Hashes can be set to literal values with the assignment statement, as in

```
%salaries = ("Gary" => 75000, "Perry" => 57000,
       "Mary" => 55750, "Cedric" => 47850);
```

- The key value is placed in braces and the hash name is replaced by a scalar variable name that is the same except for the first character.
- scalar variable names begin with dollar signs ($). For example,

```
$salaries{"Perry"} = 58850;
```

- A new element is added using the same assignment statement form. An element can be removed from the hash with the **`delete`** operator, as in

```
delete $salaries{"Gary"};
```

- The entire hash can be emptied by assigning the empty literal to it, as in

```
@salaries = ();
```

- PHP's arrays are both normal arrays and associative arrays.

Record Types

- A **record** is an aggregate of data elements in which the individual elements are identified by names and accessed through offsets from the beginning of the structure.
- In C, C++, and C#, records are supported with the `struct` data type.
- In Python and Ruby, records can be implemented as hashes, which themselves can be elements of arrays.

design issues are :

- 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

```
01 EMPLOYEE-RECORD.
    02 EMPLOYEE-NAME.
    05 FIRST PICTURE IS X(20).
    05 MIDDLE PICTURE IS X(10).
    05 LAST PICTURE IS X(20).
   02 HOURLY-RATE PICTURE IS 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.
- Ada declaration:

```
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;
```

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.
- 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.

```
Employee_Record.Employee_Name.Middle
```

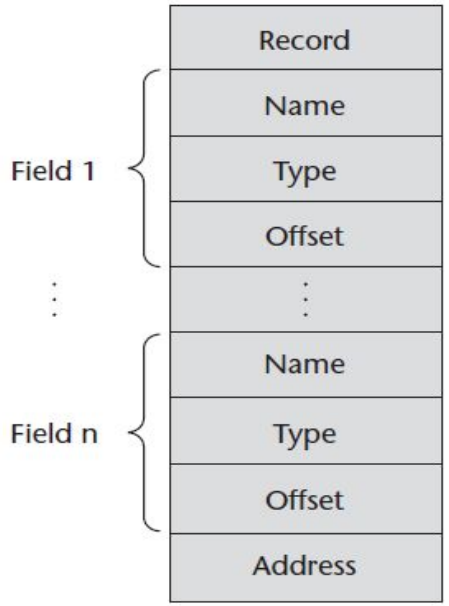
- A **fully qualified reference** to a record field is one in which all intermediate record names, from the largest enclosing record to the specific field, are named in the reference. Both the COBOL and the Ada example field references above are fully qualified.
- COBOL allows **elliptical references** to record fields. 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

Implementation of Record Types

- The fields of records are stored in adjacent memory locations. But because the sizes of the fields are not necessarily the same, the access method used for arrays is not used for records. Instead, the offset address, relative to the beginning of the record, is associated with each field.

Figure 6.7

A compile-time descriptor for a record



Syntax and Operations

- In C, a simple record might be defined as follows.

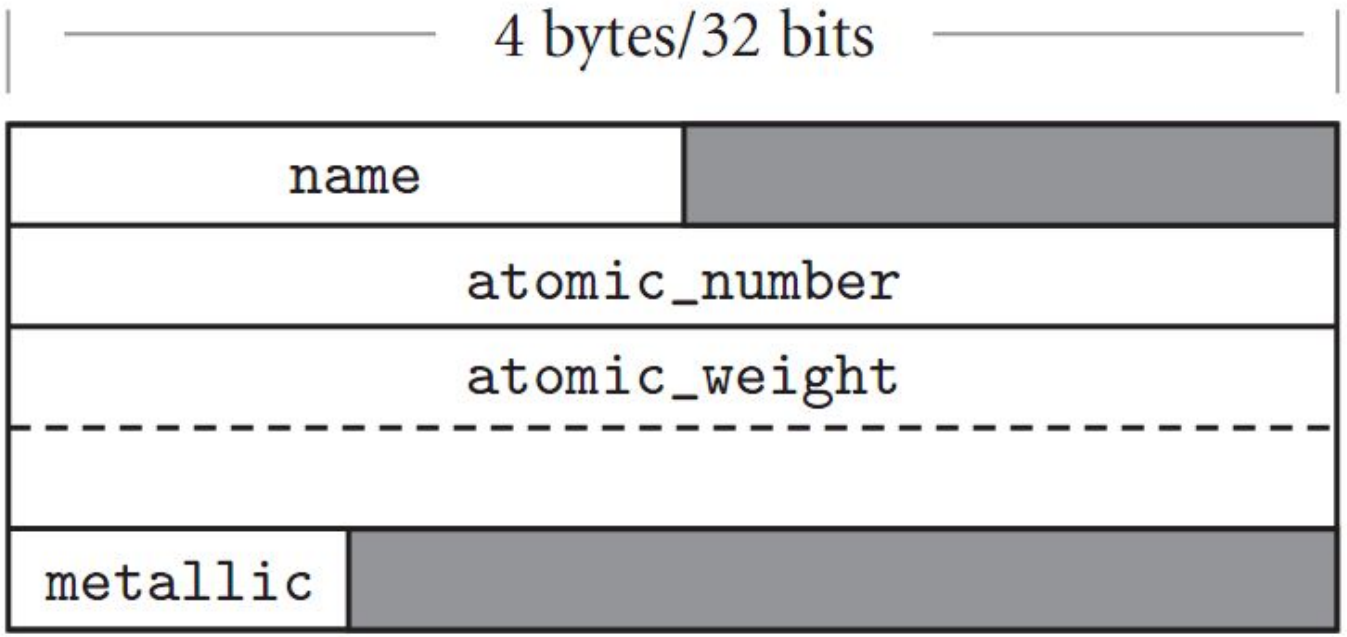
```
struct element {
char name[2];
int atomic_number;
double atomic_weight;
_Bool metallic;
};
```

– **In Pascal**

```
type element = record
name : two_chars;
atomic_number : integer;
atomic_weight : real;
metallic : Boolean
end;
```

Memory Layout and Its Impact

- The fields of a record are usually stored in adjacent locations in memory.

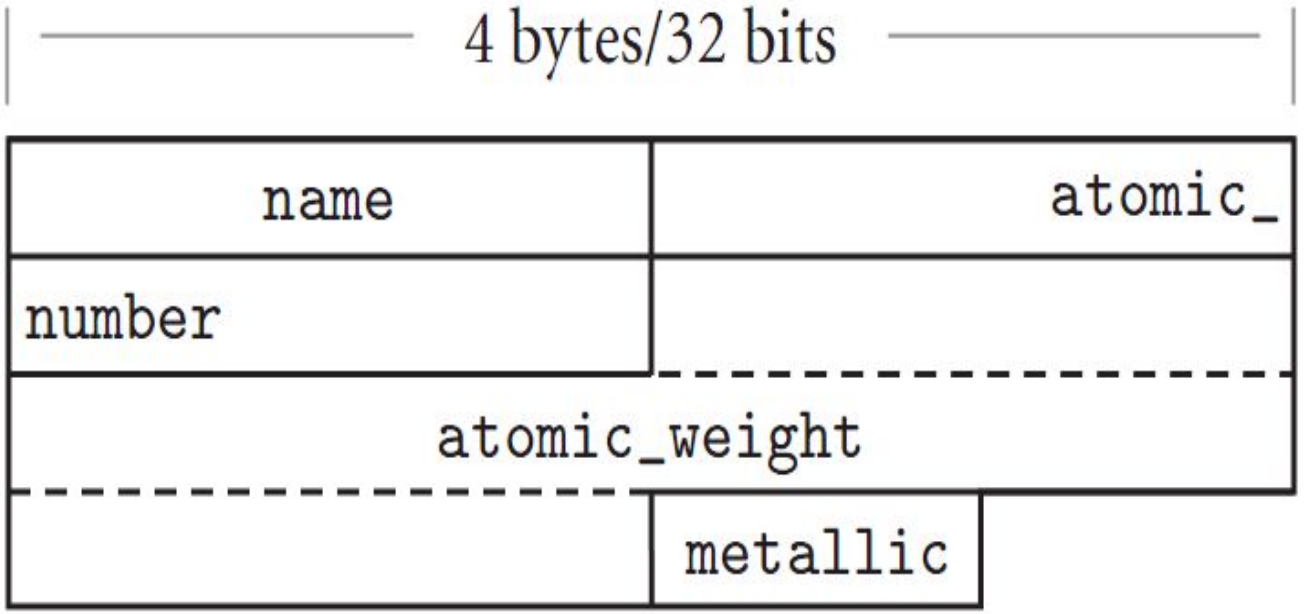


- layout in memory for objects of type element on a 32-bit machine
- Name field is only two characters long,it occupies 2 bytes in memory.
- Since atomic_number is integer,there is a 2byte hole between the end of name and beginning of atomic_number
- As Boolean is variable occupy a single byte,there are three bytes of empty space between the end of metallic field and next aligned location

- In Pascal—allow the programmer to specify that a record type (or an array, set, or file type) should be *packed:*

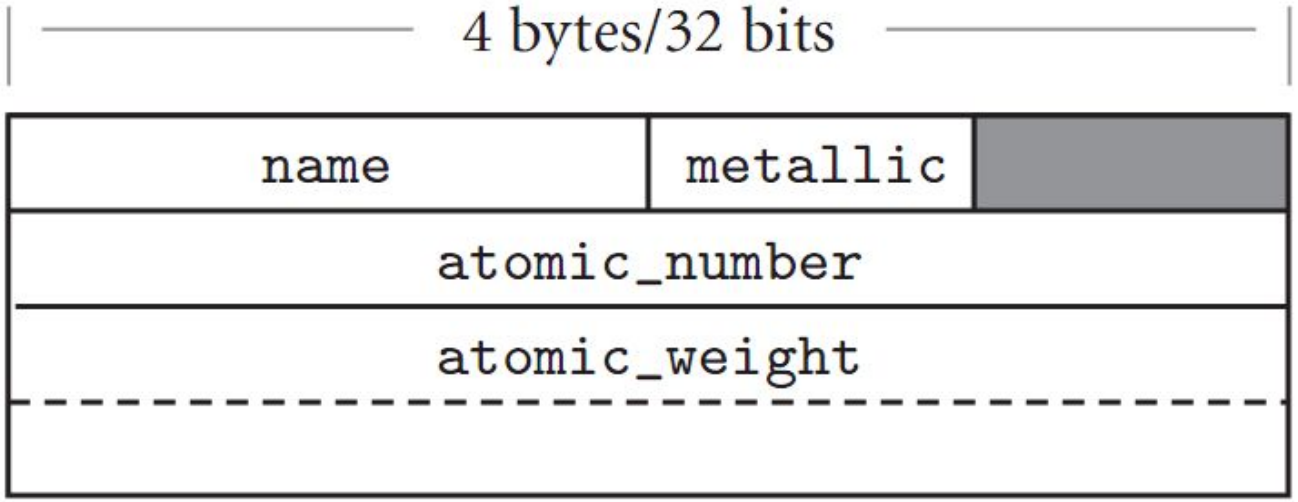
```
type element = packed record
name : two_chars;
atomic_number : integer;
atomic_weight : real;
metallic : Boolean
end;
```

- The keyword packed indicates that the compiler should optimize for space instead of speed.
- a compiler will implement a packed record without holes, by simply“pushing the fields together.”



Likely memory layout for packed element records(putting the fields together ,without holes). The atomic_number and atomic_weight fields are nonaligned, and can only be read or written (on most machines) via multi-instruction sequences.

- holes in records waste space.
- Packing eliminates holes, but at potentially heavy cost in access time.
- Another solution is that some compilers, is to sort a record's fields according to the size of their alignment constraints



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.[byte aligned , half word aligned , word aligned]

Variant Records (Unions)

- In C

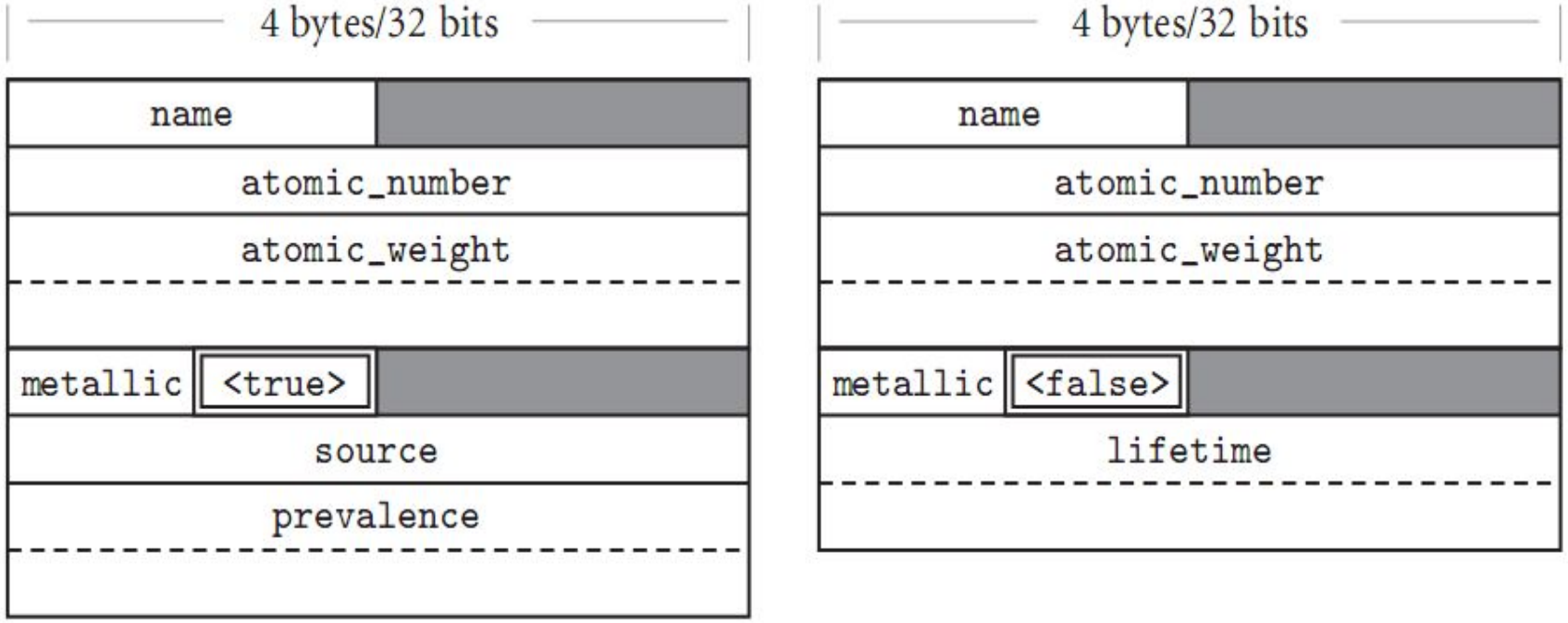
```
union {
int i;
double d;
_Bool b;
};
```

The overall size of this *union would be that of its largest member*

- Allows the programmer to specify that certain variables should be allocated “on top of one another ” sharing the same bytes in memory
- overlay space . only one member can contain value at any given time

- 2 main purposes of union

1. In system programs , **unions allows the same set of bytes to be interpreted in different ways at different times**
2. To **represent alternative sets of fields within a record**



Likely memory layouts for element variants. The value of the naturally occurring field (shown here with a double border) **determines which of the interpretations of the remaining space is valid.** Type string_ptr is assumed to be represented by a (four-byte) pointer to dynamically allocated storage.

Tuple Types

- A tuple is a data type that is similar to a record, except that the elements are not named.
- Python includes an immutable tuple type. If a tuple needs to be changed, it can be converted to an array with the `list` function. After the change, it can be converted back to a tuple with the `tuple` function.
- Python's tuples are closely related to its lists, except that tuples are immutable. A tuple is created by assigning a tuple literal, as in the following example:

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

- The elements of a tuple need not be of the same type.
- The elements of a tuple can be referenced with indexing in brackets, as in the following:

```
myTuple[1]
```

- Tuples can be catenated with the plus (+) operator. They can be deleted with the **`del`** statement.
- ML includes a tuple data type. An ML tuple must have at least two elements, whereas Python's tuples can be empty or contain one element.
- The following statement creates a tuple:

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

- The syntax of a tuple element access is as follows:

```
#1(myTuple);
```

This references the first element of the tuple

- F# also has tuples. A tuple is created by assigning a tuple value, which is a list of expressions separated by commas and delimited by parentheses, to a name in a **`let`** statement

```
let tup = (3, 5, 7);;
let a, b, c = tup;;
```

This assigns `3` to `a`, `5` to `b`, and `7` to `c`.

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)`
- The fundamental list operations in Scheme are two functions that take lists apart and two that build lists. The `CAR` function returns the first element of its list parameter. For example, consider the following example: `(CAR '(A B C))` 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)`.
- 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))`.

- ML has lists and list operations. Lists are specified in square brackets, with the elements separated by commas, as in the following list of integers: `[5, 7, 9]`
- In ML `CONS` function is implemented as a binary infix operator, represented as `::`. For example,

  `3 :: [5, 7, 9]` returns the following new list: `[3, 5, 7, 9]`.
- The elements of a list must be of the same type, so the following list would be illegal:

  ```
  [5, 7.3, 9]
  ```
- ML has functions that correspond to Scheme's `CAR` and `CDR`, named `hd` (head) and `tl` (tail). For example,

  ```
  hd [5, 7, 9] is 5
  tl [5, 7, 9] is [7, 9]
  ```
- Python includes a list data type, which also serves as Python's arrays. For example, consider the following statement:

  ```
  myList = [3, 5.8, "grape"]
  ```
- The elements of a list are referenced with subscripts in brackets, as in the following example:

  `x = myList[1]` This statement assigns `5.8` to `x`.
- A list element can be deleted with `del`, as in the following statement:

  `del myList[1]` This statement removes the second element of `myList`.

- Python includes a powerful mechanism for creating arrays called **list comprehensions.** A list comprehension is an idea derived from set notation. It first appeared in the functional programming language Haskell.
- The syntax of a Python list comprehension is as follows:

  [expression **`for`** iterate_var **`in`** array **`if`** condition]

- Consider the following example:

```
[x * x for x in range(12) if x % 3 == 0]
```

- The **`range`** function creates the array `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10, 11, 12]`. The conditional filters out all numbers in the array that are not evenly divisible by `3`.
- This list comprehension returns the following array: `[0, 9, 36, 81]`
- Slices of lists are also supported in Python.
- Haskell's list comprehensions have the following form: `[`body `|` qualifiers`]`
- For example, consider the following definition of a list: `[n * n | n <- [1..10]]`
- This defines a list of the squares of the numbers from `1` to `10`.

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 bindings of variables to types are static in a language, then type checking can nearly always be done statically. Dynamic type binding requires type checking at run time, which is called **dynamic type checking**.
- JavaScript and PHP, allow only dynamic type checking.

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. The importance of strong typing lies in 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. Ada is nearly strongly typed
- C and C++ are not strongly typed languages .ML and F# are strongly typed.

Arithmetic Expressions

- Expressions are the fundamental means of specifying computations in a programming language. It is crucial for a programmer to understand both the syntax and semantics of expressions of the language being used.
- In programming languages, arithmetic expressions consist of operators, operands, parentheses, and function calls. An operator can be **unary**, meaning it has a single operand, **binary**, meaning it has two operands, or **ternary**, meaning it has three operands.
- In most programming languages, binary operators are **infix**, which means they appear between their operands. One exception is Perl, which has some operators that are **prefix**, which means they precede their operands.
- design issues for arithmetic expressions
  - What are the operator precedence rules?
  - What are the operator associativity rules?
  - What is the order of operand evaluation?
  - Are there restrictions on operand evaluation side effects?
  - Does the language allow user-defined operator overloading?
  - What type mixing is allowed in expressions?

Operator Evaluation Order

- The operator precedence and associativity rules of a language dictate the order of evaluation of its operators.

Precedence

- The value of an expression depends at least in part on the order of evaluation of the operators in the expression. Consider the following expression:

```
a + b * c
```

- Suppose the variables `a`, `b`, and `c` have the values `3`, `4`, and `5`, respectively. If evaluated left to right the result is `35`. If evaluated right to left, the result is `23`.
- The **operator precedence rules** for expression evaluation partially define the order in which the operators of different precedence levels are evaluated.The operator precedence rules for expressions are based on the hierarchy of operator priorities, as seen by the language designer.
- In all of the common imperative languages, the unary minus operator can appear in an expression either at the beginning or anywhere inside the expression, as long as it is parenthesized to prevent it from being next to another operator. For example,

```
a + (- b) * c
```
is legal, but

```
a + - b * c
```
illegal

- Of the common programming languages, only Fortran, Ruby, Visual Basic, and Ada have the exponentiation operator. In all four, exponentiation has higher precedence than unary minus, so
  - `A ** B` is equivalent to `-(A ** B)`

The precedences of the arithmetic operators of Ruby and the C-based languages are as follows:

	Ruby	*C-Based Languages*
Highest	**	postfix ++, --
	unary +, -	prefix ++, --, unary +, -
	*, /, %	*, /, %
Lowest	binary +, -	binary +, -

Associativity

- When an expression contains two adjacent occurrences of operators with the same level of precedence, the question of which operator is evaluated first is answered by the **associativity** rules of the language. An operator can have either left or right associativity, meaning that when there are two adjacent operators with the same precedence, the left operator is evaluated first or the right operator is evaluated first, respectively.
- Associativity in common languages is left to right, except that the exponentiation operator sometimes associates right to left. In the Java expression

  `a - b + c` the left operator is evaluated first.
- Exponentiation in Fortran and Ruby is right associative, so in the expression

  `A ** B ** C` the right operator is evaluated first.
- In Ada, exponentiation is nonassociative, which means that the expression
- `A ** B ** C` is illegal. Such an expression must be parenthesized to show the desired order,
- as in either `(A ** B) ** C` or

The associativity rules for a few common languages are given here:

Language	*Associativity Rule*
Ruby	Left: *, /, +, -
	Right: **
C-based languages	Left: *, /, %, binary +, binary -
	Right: ++, --, unary -, unary +
Ada	Left: all except **
	Nonassociative: **

Parentheses

- Programmers can alter the precedence and associativity rules by placing parentheses in expressions. A parenthesized part of an expression has precedence over its adjacent unparenthesized parts. For example, although multiplication has precedence over addition, in the expression

  `(A + B) * C` the addition will be evaluated first

Ruby Expressions

- Recall that Ruby is a pure object-oriented language. For example, the expression `a + b` is a call to the + method of the object referenced by `a`, passing the object referenced by `b` as a parameter.

Expressions in LISP

- In LISP, the expression `(+ a (* b c))` , `+` and `*` are the names of functions.

Conditional Expressions

- **`if-then-else`** statements can be used to perform a conditional expression assignment. For example, consider

```
if (count == 0)
average = 0;
else
average = sum / count;
```

- In the C-based languages, this code can be specified more conveniently in an assignment statement using a conditional expression, which has the form

  expression_1 ? expression_2 : expression_3

- where expression_1 is interpreted as a Boolean expression. If expression_1 evaluates to true, the value of the whole expression is the value of expression_2; otherwise, it is the value of expression_3.

Operand Evaluation Order

- If neither of the operands of an operator has side effects, then operand evaluation order is irrelevant.

Side Effects

- A **side effect** of a function, naturally called a functional side effect, occurs when the function changes either one of its parameters or a global variable.
- Consider the expression

```
a + fun(a)
```

- If fun does not have the side effect of changing `a`, then the order of evaluation of the two operands, `a` and `fun(a)`, has no effect on the value of the expression.
- However, if `fun` changes `a`, there is an effect. Consider the following situation:
- `fun` returns `10` and changes the value of its parameter to `20`. Suppose we have the following:

```
a = 10;
b = a + fun(a);
```

- Then, if the value of `a` is fetched first, its value is `10` and the value of the expression is `20`. But if the second operand is evaluated first, then the value of the first operand is `20` and the value of the expression is `30`.

- The following C program illustrates the same problem when a function changes a global variable that appears in an expression:

```
int a = 5;

int fun1() {

a = 17;

return 3;

} /* end of fun1 */

void main() {

a = a + fun1();

} /* end of main */
```

- The value computed for `a` in `main` depends on the order of evaluation of the operands in the expression `a + fun1()`. The value of `a` will be either `8` (if `a` is evaluated first) or `20`.
- There are two possible solutions to the problem of operand evaluation order and side effects.
- First, the language designer could disallow function evaluation from affecting the value of expressions by simply disallowing functional side effects.
- Second, the language definition could state that operands in expressions are to be evaluated in a particular order and demand that implementors guarantee that order.
- Disallowing functional side effects in the imperative languages is difficult, and it eliminates some flexibility for the programmer.

Referential Transparency and Side Effects

- A program has the property of **referential transparency** if any two expressions in the program that have the same value can be substituted for one another anywhere in the program, without affecting the action of the program.
- The value of a referentially transparent function depends entirely on its parameters.
- For example:

```
result1 = (fun(a) + b) / (fun(a) - c);
temp = fun(a);
result2 = (temp + b) / (temp - c);
```

- If the function `fun` has no side effects, `result1` and `result2` will be equal, because the expressions assigned to them are equivalent.
- Suppose `fun` has the side effect of adding 1 to either `b` or `c`. Then `result1` would not be equal to `result2`.
- So, that side effect violates the referential transparency of the program in which the code appears.
- **advantages**
- the semantics of such programs is much easier to understand than the semantics of programs that are not referentially transparent.
- ease of understanding

Overloaded Operators

- Multiple use of an operator is called **operator overloading.**
- consider the use of the ampersand (`&`) in C++. As a binary operator, it specifies a bitwise logical AND operation. As a unary operator with a variable as its operand, the expression value is the address of that variable.
- For example, the execution of

  `x = &y;` causes the address of `y` to be placed in `x`.
- two problems with this multiple use of the ampersand.
- First, using the same symbol for two completely unrelated operations is detrimental to readability.
- Second, the simple keying error of leaving out the first operand for a bitwise AND operation can go undetected by the compiler, because it is interpreted as an address-of operator.
- `+` usually is used to specify integer addition and floating-point addition. Some languages—Java, for example—also use it for string catenation.

Type Conversions

- Type conversions are either narrowing or widening.
- A **narrowing conversion** converts a value to a type that cannot store even approximations of all of the values of the original type. For example, converting a `double` to a `float` in Java is a narrowing conversion, because the range of `double` is much larger than that of `float`.
- A **widening conversion** converts a value to a type that can include at least approximations of all of the values of the original type. For example, converting an `int` to a `float` in Java is a widening conversion.
- Type conversions can be either explicit or implicit.

Coercion in Expressions

- Languages that allow such expressions, which are called **mixed-mode expressions**, must define conventions for implicit operand type conversions.
- coercion was defined as an implicit type conversion that is initiated by the compiler. Type conversions explicitly requested by the programmer are referred to as explicit conversions, or casts, not coercions.
- For overloaded operators in a language that uses static type binding, the compiler chooses the correct type of operation on the basis of the types of the operands.
- When the two operands of an operator are not of the same type and that is legal in the language, the compiler must choose one of them to be coerced and supply the code for that coercion

- consider the following Java code:

```
int a;
float b, c, d;
. . .
d = b * a;
```

- Assume that the second operand of the multiplication operator was supposed to be `c`, but because of a keying error it was typed as `a`. Because mixed-mode expressions are legal in Java, the compiler would not detect this as an error. It would simply insert code to coerce the value of the **`int`** operand, `a`, to **`float.`**
- If mixed-mode expressions were not legal in Java, this keying error would have been detected by the compiler as a type error.
- Ada allows very few mixed type operands in expressions.

```
A : Integer;
B, C, D : Float;
. . .
C := B * A;
```

- then the Ada compiler would find the expression erroneous, because `Float` and `Integer` operands cannot be mixed for the `*` operator.
- ML and F# do not coerce operands in expressions
- In Java, they are **`byte`** and **`short`** are coerced to **`int`** whenever virtually any operator is applied to them.

```
byte a, b, c;
. . .
a = b + c;
```

- The values of `b` and `c` are coerced to **`int`** and an **`int`** addition is performed. Then, the sum is converted to **`byte`** and put in `a`.

Explicit Type Conversion

- Most languages provide some capability for doing explicit c onversions, both widening and narrowing. In some cases, warning messages are produced when an explicit narrowing conversion results in a significant change to the value of the object being converted.
- In the C-based languages, explicit type conversions are called **casts**. To specify a cast, the desired type is placed in parentheses just before the expression to be converted, as in

```
(int) angle
```

- In ML and F#, the casts have the syntax of function calls. For example, in F# we could have the following:

```
float(sum)
```

Errors in Expressions

- The most common error occurs when the result of an operation cannot be represented in the memory cell where it must be stored. This is called **overflow** or **underflow**, depending on whether the result was too large or too small.

Relational and Boolean Expressions

Relational Expressions

- A **relational operator** is an operator that compares the values of its two operands.
- A relational expression has two operands and one relational operator. The value of a relational expression is Boolean, except when Boolean is not a type included in the language.

Highest	postfix `++`, `--`
	unary `+`, `-`, prefix `++`, `--`, `!`
	`*`, `/`, `%`
	binary `+`, `-`
	`<`, `>`, `<=`, `>=`
	`=`, `!=`
	`&&`
Lowest	`||`

- Versions of C prior to C99 have no Boolean type and thus no Boolean values. Instead, numeric values are used to represent Boolean values.
- One odd result of C's design of relational expressions is that the following expression is legal:

```
a > b > c
```

- The leftmost relational operator is evaluated first because the relational operators of C are left associative, producing either 0 or 1.
- Some languages, including Perl and Ruby, provide two sets of the binary logic operators, `&&` and **`and`** for AND and `||` and **`or`** for OR. One difference between `&&` and **`and`** (and `||` and **`or`**) is that the spelled versions have lower precedence. Also, **`and`** and **`or`** have equal precedence, but `&&` has higher precedence than `||`.

Short-Circuit Evaluation

- A **short-circuit evaluation** of an expression is one in which the result is determined without evaluating all of the operands and/or operators. For example, the value of the arithmetic expression

```
(13 * a) * (b / 13 - 1)
```

- is independent of the value of `(b / 13 - 1)` if `a` is `0`, because `0 * x = 0` for any `x`. So, when `a` is `0`, there is no need to evaluate `(b / 13 - 1)` or perform the second multiplication
- The value of the Boolean expression

```
(a >= 0) && (b < 10)
```

- is independent of the second relational expression if `a < 0`, because the expression

  (`FALSE && (b < 10)`) is `FALSE` for all values of `b`.
- In the C-based languages, the usual AND and OR operators, `&&` and `||`, respectively, are short-circuit.
- All of the logical operators of Ruby, Perl, ML, F#, and Python are shortcircuit evaluated.
- The inclusion of both short-circuit and ordinary operators in Ada is clearly the best design, because it provides the programmer the flexibility of choosing short-circuit evaluation for any Boolean expression for which it is appropriate

- Code generation for a Boolean condition

```
if ((A > B) and (C > D)) or (E <> F) then
   then_clause
  else
   else_clause
```

In **Pascal**, **which does not use short-circuit evaluation**, the output code
r1 := A – – load
r2 := B
r1 := r1 > r2
r2 := C
r3 := D
r2 := r2 > r3
r1 := r1 & r2
r2 := E
r3 := F
r2 := r2 = r3
r1 := r1 | *r2*
if r1 = 0 goto L2
L1: *then clause – – (label not actually used)*
goto L3
L2: *else clause*
L3:

- The root of the subtree for ((A > B) and (C > D)) or (E = F) would name r1 as the
register containing the expression value

- Code generation for short-circuiting

```
if ((A > B) and (C > D)) or (E <> F) then
  then_clause
else
  else_clause
```

r1 := A
r2 := B
if r1 <= r2 goto L4
r1 := C
r2 := D
if r1 > r2 goto L1
L4: r1 := E
r2 := F
if r1 = r2 goto L2
L1: *then clause*
goto L3
L2: *else clause*
L3:

- Here **the value of the Boolean condition is never explicitly placed into a registe**r.It is

implicit in the flow of control.Moreover for most values of A, B, C, D,and E, the

execution path through **the jump code is shorter and therefore faster**

than the straight-line code that calculates the value of every subexpression

Assignment Statements

Simple Assignments

- Nearly all programming languages currently being used use the equal sign for the assignment operator. All of these must use something different from an equal sign for the equality relational operator to avoid confusion with their assignment operator.
- ALGOL 60 pioneered the use of `:=` as the assignment operator, which avoids the confusion of assignment with equality. Ada also uses this assignment operator.

Conditional Targets

- Perl allows conditional targets on assignment statements. For example, consider

```
($flag ? $count1 : $count2) = 0;
```

- which is equivalent to

```
if ($flag) {
$count1 = 0;
} else {
$count2 = 0;
}
```

Compound Assignment Operators

- A **compound assignment operator** is a shorthand method of specifying a commonly needed form of assignment .
- The form of assignment has the destination variable also appearing as the first operand in the expression on the right side, as in

```
a = a + b
```

- Compound assignment operators were introduced by ALGOL 68.
- The syntax of these assignment operators is the catenation of the desired binary operator to the = operator. For example,

`sum += value;` is equivalent to `sum = sum + value;`

Unary Assignment Operators

- The C-based languages, Perl, and JavaScript include two special unary arithmetic operators that are actually abbreviated assignments.
- They combine increment and decrement operations with assignment.
- The operators `++` for increment, and `--` for decrement, can be used either in expressions or to form stand-alone single-operator assignment statements

- In the assignment statement

```
sum = ++ count;
```

- the value of `count` is incremented by `1` and then assigned to `sum`. This operation could also be stated as

```
count = count + 1;
sum = count;
```

- If the same operator is used as a postfix operator, as in

```
sum = count ++;
```

- the assignment of the value of `count` to `sum` occurs first; then `count` is incremented.
- The effect is the same as that of the two statements

```
sum = count;
count = count + 1;
```

- When two unary operators apply to the same operand, the association is right to left. For example, in

```
- count ++
```

- `count` is first incremented and then negated. So, it is equivalent to `- (count ++)`

Assignment as an Expression

- In the C-based languages, Perl, and JavaScript, the assignment statement produces a result, which is the same as the value assigned to the target. It can therefore be used as an expression and as an operand in other expressions. This design treats the assignment operator much like any other binary operator, except that it has the side effect of changing its left operand. For example, in C, it is common to write statements such as

```
while ((ch = getchar()) != EOF) { ... }
```

- In this statement, the next character from the standard input file, usually the keyboard, is gotten with `getchar` and assigned to the variable `ch`. The result, or value assigned, is then compared with the constant `EOF`.
- The disadvantage of allowing assignment statements to be operands in expressions is that it provides yet another kind of expression side effect.
- There is a loss of error detection in the C design of the assignment operation that frequently leads to program errors. In particular, if we type

```
if (x = y) ...
```

- instead of

```
if (x == y) ...
```

- which is an easily made mistake, it is not detectable as an error by the compiler.

Multiple Assignments

- Several recent programming languages, including Perl, Ruby, and Lua, provide multiple-target, multiple-source assignment statements. For example, in Perl we can write

```
($first, $second, $third) = (20, 40, 60);
```

- The semantics is that `20` is assigned to `$first`, `40` is assigned to `$second`, and `60` is assigned to `$third`. If the values of two variables must be interchanged, this can be done with a single assignment, as with `($first, $second) = ($second, $first);`

Assignment in Functional Programming Languages

- All of the identifiers used in pure functional languages and some of them used in other functional languages are just names of values. As such, their values never change. For example, in ML
- names are bound to values with the **`val`** declaration, whose form is exemplified in the following:

```
val cost = quantity * price;
```

Mixed-Mode Assignment

- Fortran, C, C++, and Perl use coercion rules for mixed-mode assignment that are similar to those they use for mixed-mode expressions.
- C++, Java and C# allow mixed-mode assignment only if the required coercion is widening.8 So, an `int` value can be assigned to a `float` variable, but not vice versa.