# Chapter 6

## Data Types

# Topics

- Introduction
- Primitive Data Types
- Character String Types
- Enumeration Types
- Array Types
- Associative Arrays
- Record Types
- Tuple Types
- List Types
- Union Types
- Pointer and Reference Types
- Optional Types
- Type Checking
- Strong Typing
- Type Equivalence
- Theory and Data Types

# Introduction

- A *data type* defines a *collection of data objects* and a *set of predefined operations* on those objects.
- A *descriptor* is the collection of the attributes of a variable.
- An *object* represents an *instance* of a user-defined (abstract data) type.
- One design issue for all data types:

    What *operations* are defined and

    how are they *specified*?

# Primitive Data Types

- Almost all programming languages provide a set of *primitive data types.*
- *Primitive* data types: Those not defined in terms of other data types.

  E.g.) integer, float-point, boolean, character, etc.
- Some primitive data types are merely reflections of the *hardware*.
- Others require only a little *non-hardware support* for their implementation.

# Primitive Data Types: Integer

- Almost always an *exact reflection* of the hardware

  so the mapping is trivial.
- There may be as many as *eight* different integer types in a language.
- Java's signed integer sizes:

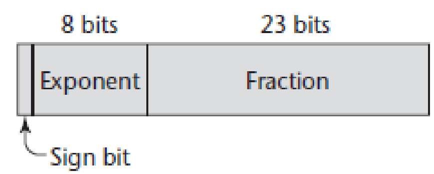
  **`byte, short, int, long`**

  (1 byte – 2 byte – 4 byte – 8 byte)

# Primitive Data Types: Floating Point

- Model real numbers, but only as approximations
- Languages for scientific use support at least two floating-point types
  - e.g., **float** (4 byte) and **double** (8 byte)
  - Usually exactly like the hardware.
- IEEE Floating-Point standard 754 format:

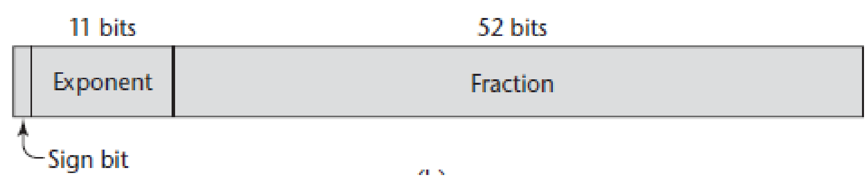
single-precision (32 bits)



(a)

vs.

double-precision (64 bits)




encoding - https://www.sciencedirect.com/topics/computer-science/single-precision-format

# Primitive Data Types: Complex

- Some languages support a complex type.

  e.g.) C99, Fortran, and Python

- Each value consists of two floats:

  a *real* part + an *imaginary* part.

- Complex literal form:
  - In Python: $(7 + 3i)$,

    where 7 is the real part

    and 3 is the imaginary part

# Primitive Data Types: Decimal

- For business applications (money)
  - Essential to COBOL.  C# offers a decimal data type.
- Store a *fixed number of decimal digits* with the implied decimal point at a fixed position in the value, in coded form (BCD – binary coded decimal) – supports up to 29 significant digits and can represent values > $7.9228 * 10^{28}$.
- *Advantage*: accuracy
- *Disadvantages*: limited range, wastes memory

# Primitive Data Types: Boolean

- Simplest of all
- Range of values: two elements
  - 1 for "true" and 0 for "false"
- Could be implemented as bits, but often as bytes
  - Advantage: readability

# Primitive Data Types: Character

- Stored as numeric codings
- Most commonly used coding: ASCII (7-bit coding)
- An alternative, 16-bit coding: Unicode (UCS-2)
  - UCS: universal coded character
  - Includes characters from most natural languages
  - Originally used in Java
  - Now supported by many languages
- 32-bit Unicode (UCS-4)
  - Supported by Fortran, starting with 2003

# Character String Types

- Values are *sequences of characters.*
- Design issues:
  - Is it a primitive type or just a special kind of array?
  - Should the length of strings be static or dynamic?

# Character String Types: Operations

- Typical operations:
  - Assignment and copying
  - Comparison (=, >, etc.)
  - Catenation
  - Substring reference: a reference to a substring of a given string, called a *slice*.
  - Pattern matching:
    - Regular expression
    - Included in the class libraries of C++, Java, Python, C#, F#.
- Example:
  - C, C++: `strcpy, strcat, strcmp, strlen`

# Character String Type: in Certain Languages

- C and C++
  - Not primitive
  - Use **`char`** arrays and a library of functions that provide operations: e.g.) **`char`** `str[] = "apples"`
    - `#include <string>`, **`string`** `greeting = "Hello"`
- SNOBOL4 (a string manipulation language)
  - Primitive
  - Many operations, including elaborate *pattern matching*.
- Fortran and Python
  - Primitive type with assignment and several operations.
- Java, C#, Ruby, Swift
  - Primitive via the **`String`** and **`StringBuffer`** class
- Perl, JavaScript, Ruby, and PHP

  \- Provide built-in pattern matching, using regular expressions

# Character String: Length Options in Design

- *Static length string*:
  - the length is static and set when a string is created.
  - COBOL, Python, Ruby, Java's built-in `String` class
  - C++: string class library
- *Limited Dynamic Length*: C and C++
  - Varying length *up to* a fixed *maximum* length in definition.
  - A special character is used to indicate the *end of a string*'s characters, rather than maintaining the length,
    - e.g.) the null character, which is simply the character with the value 0.
- *Dynamic* (no maximum): Perl, JavaScript.

# Character String Type: Evaluation

- Aid to writability
- As a primitive type with static length, they are inexpensive to provide--why not have them?
- Dynamic length is nice, but is it worth the expense?

# Character String: Implementation

- Static length: *compile-time descriptor*
- Limited dynamic length: may need a *run-time descriptor* for length (but not in C and C++)
- Dynamic length: need *run-time descriptor*; allocation/deallocation is the biggest implementation problem
- 3 approaches to support the dynamic (de)allocation.
  - Store strings in a *linked list*
  - Store them as *arrays of pointers* to individual characters allocated in the heap.
  - Store complete strings *in adjacent storage cells*.
    - What if a string grows? A new area of memory is found/stores the complete new string and the old part is moved to this area.

# Compile- and Run-Time Descriptors

- Name of the type
- Type’s length
- Address of the 1st character

| |
|---|
| Static string |
| Length |
| Address |

Compile-time descriptor for static strings

| |
|---|
| Limited dynamic string |
| Maximum length |
| Current length |
| Address |

Run-time descriptor for limited dynamic strings

# Enumeration Types

- A data type in which all possible values, which are *named constants*, are provided/enumerated in the definition.
  - a way of defining and *grouping collections of named constants* - enumeration constants.
- 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?
  - All of the above are related to type checking.

# Enumerated Type: Designs

- In a language with no enumeration type, simulate it with integer values.
  - `enum days {Sun, Mon, Tue, Wed, Thr, Fri, Sat};`

    `where` `Sun` is state=0, `Mon` is state=1, etc.
  - https://www.geeksforgeeks.org/enumeration-enum-c/
- C++ includes C's enumeration type.
  - `enum colors {red, blue, green, yellow, black};`
  - `colors myColor = blue, yourColor = red;`
  - `myColor++` → `myColor? is green`
- In ML, it's defined as new type with datatype declaration.
  - `datatype colors = red | blue | green | yellow |black`
- Swift has an enumeration type.

```
enum fruit {
    case orange
    case apple
    case banana
}
```

# Enumerated Type: Evaluation

- Aid to readability
  - e.g.) no need to code a color as a number.
- Aid to reliability, e.g., compiler can check:
  - operations (don't allow colors to be added)
  - No enumeration variable can be assigned a value outside its defined range.
  - Better support in C#, F#, Swift, and Java 5.0 than C++:
    - because their enumeration type variables are not coerced into integer types.

# Array Types

- An array is a *homogeneous* aggregate of data elements in which an individual element is identified by its *position* in the aggregate, relative to the first element.
- The individual data elements of an array are of the *same type*.
- References to individual array elements are specified using *subscript/index expressions*.
- If any of the subscript expressions in a reference include *variables*, then the reference will require an additional run-time calculation to determine the address of the memory location being referenced.
  - E.g.) sum = sum + A[*i*]

# Array Design Issues

- What *types* are legal for *subscripts*?
- Are subscripting expressions in element references *range* checked?
- When are subscript ranges bound?
- When does allocation take place?
- Are ragged or rectangular multidimensional arrays allowed, or both?
- What is the maximum number of subscripts?
- Can array objects be initialized?
- Are any kind of slices supported?

# Array Indexing

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

  array_name (index_value_list) $\rightarrow$ an element

- Index Syntax
  - Fortran and Ada use parentheses '( )'.
    - Ada explicitly uses parentheses to show uniformity between array references and function calls

      because both are *mappings.*

    e.g.) `Sum := Sum + B(I)`
  - Most other languages use brackets [ ].

# Arrays Index (Subscript) Types

- Often integer types.
- Index range checking
  - C, C++, Perl, and Fortran do not specify the range checking of subscripts.
  - Java, ML, C# specify range checking.
- In Perl:
  - a name of an array begins with @ sign.
  - Reference to array elements uses $ sign.
  - E.g.) For the array `@age`, the 2nd element is referenced with `$age[1]`.
  - Ref: slide #15-chap.5

# Subscript Binding and Array Categories

- *Static array*:
  - subscript *ranges* are statically bound and storage allocation is static (before run-time).
  - Advantage: efficiency (no dynamic allocation)
- *Fixed stack-dynamic array*:
  - subscript ranges are *statically bound*, but the allocation is done at *declaration time* during execution.
  - Advantage: space efficiency with *space sharing*
    - A large array in one subprogram/block can use the same space as a large array in a different subprogram/block, as long as both subprograms are not active at the same time.

# Subscript Binding and Array Categories (cont.)

- *Fixed heap-dynamic array*:
  - Subscript ranges and storage binding are *fixed* after allocation.
  - Both the subscript ranges and storage *bindings* are done when *requested* (by an instruction) and storage is allocated from a *heap*, not stack.
  - Advantage: flexibility – the array size always fits the prob.
  - Disadvantage: storage allocation time is longer than from a stack.
- *Heap-dynamic* array:
  - binding of subscript ranges and storage allocation is dynamic and can change any number of times during the array's lifetime.
  - Advantage: flexibility
    - the arrays can grow or shrink during program execution.

# Subscript Binding and Array Categories (cont.)

- C and C++ arrays with **`static`** modifier are *static;* otherwise, *fixed stack-dynamic*.
- C, C++, C#, Java provide *fixed heap-dynamic arrays*.
  - C: library functions `malloc` and `free` for C arrays.
  - C++: `new` and `delete` to manage heap storage.
  - Java: all non-generic arrays are fixed heap-dynamic.
- Perl, JavaScript, Python, Ruby support *heap-dynamic* arrays.
  - C#: objects of `List` class.

```
List<String> stringList = new List<String>();
stringList.Add(“Michael”);
```

- The array object is created with no element; then, add element.

where `List` is a generic heap-dynamic collection class.

# Heterogeneous Arrays

- A *heterogeneous array* is one in which the elements need not be of the same type.
  - e.g.)
```
var = 10
A = (2, “apple”, var, ‘A’)
```
- The elements are all *references* to data objects that reside in *scattered locations*, often on the heap.
- Supported by:
  - Perl, Python, JavaScript, and Ruby

# Array: Initialization

- Some languages allow initialization *at the time of storage allocation*.
- C, C++, Java, Swift, and C#
- Example: C-based languages
  - `int list [] = {1, 3, 5, 7}`
  - `char name [] = "Freddie"`

    - character array as string constant with 8 elements which is terminated with a null character (zero) `Freddie0`
  - `char *names [] = {"Bob", "Fred", "Mary"};`

    - an array of pointers to characters where the literals are pointers to characters. E.g.) names[0] is a pointer to the letter 'B' in the literal character array that contains the characters 'B', 'o', 'b', and the null.
- Example: Java initialization of String objects
  - `String[] names = {"Bob", "Jake", "Joe"};`
    - the array is an array of references to string objects.

# Array: Initialization (cont.)

- Python
  - Python’s array is a list.
  - Through assignment statement

```
names = ['Daniel', 'Roxanna', 'Jean']
```

  - List comprehensions

```
num = [x ** 2 for x in range(12) if x % 3 == 0]
```

**output:** puts [0, 9, 36, 81] in num

# Array: Operations

- Operation that operates on an array as a unit.
  - E.g.) assignment, catenation, comparison for (in)equality, slices,
- C-based languages provide NO array operation, except through the *methods* of Java, C++, and C#.
  - E.g.)

```
string firstName = "John ";
string lastName = "Doe";
string name = string.Concat(firstName, lastName);
Console.WriteLine(name);
```

- C# (method)

```
strcat( str1, str2);
```

- C (function)

- Python:
  - Supports array assignment though it's only reference change.
  - Array catenation (`+`), element membership (`in`).
  - Comparison: `is` (do two variables reference the same object?), `==` (compare all corresponding objects in the referenced objects)
- Ruby's array is reference to objects.
  - It supports catenation with an Array method

# Rectangular and Jagged Arrays

- A rectangular array:
  - a *multi-dimensioned array* in which all of the rows have the *same* number of elements and all columns have the *same* number of elements.
  - Reference in a *single pair* of [ ]: e.g.) `myArray[3, 7]`
- A *jagged array*:
  - one that has rows with *varying number* of elements.
  - Possible when multi-dimensioned arrays actually appear as *arrays of arrays*.
  - A reference uses a *separate pair* of [ ] for each dimension. E.g.) `myArray[3][7]` ≠ `myArray[3,7]` (rectangular array)
- C, C++, Java: support jagged arrays.
- F#, C#: support both rectangular arrays and jagged arrays.

# Slices

- A slice is some *substructure* of an array; a referencing mechanism.
- Slices are only useful in languages that have array operations.
- Example: Python

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

`vector[x:y:z]` returns elements from index `x` to `y-1` with step size `z`

e.g.) `vector[1:7:2]`→`[4,8,12]`

`mat[x][y:z]:` returns elements from index `y` to `z-1` at the row `x`.

e.g.) `mat[0][0:2]` →`[1,2]`

- Ruby supports slices with the `slice` *method*

`list.slice(x,y)` returns `y` elements of `list` from index `x`.

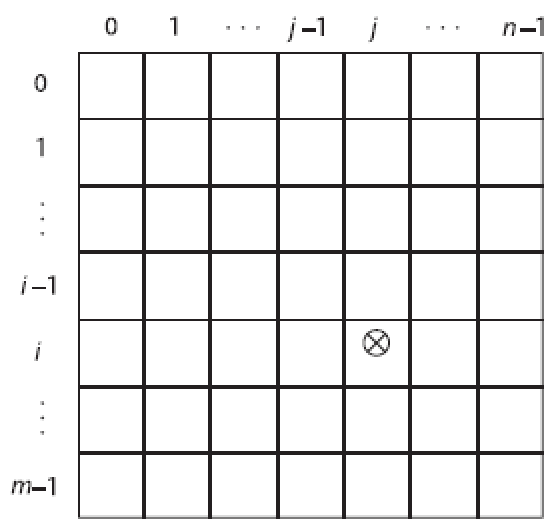
# Implementation of Arrays

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

address(`list[k]`) = address (`list[`lower_bound`]`)

+ ((`k` – lower_bound) * element_size)

Often, address(`list[k]`) = address (`list[0]`) + `k` * 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
  - A compile-time descriptor for a multidimensional array

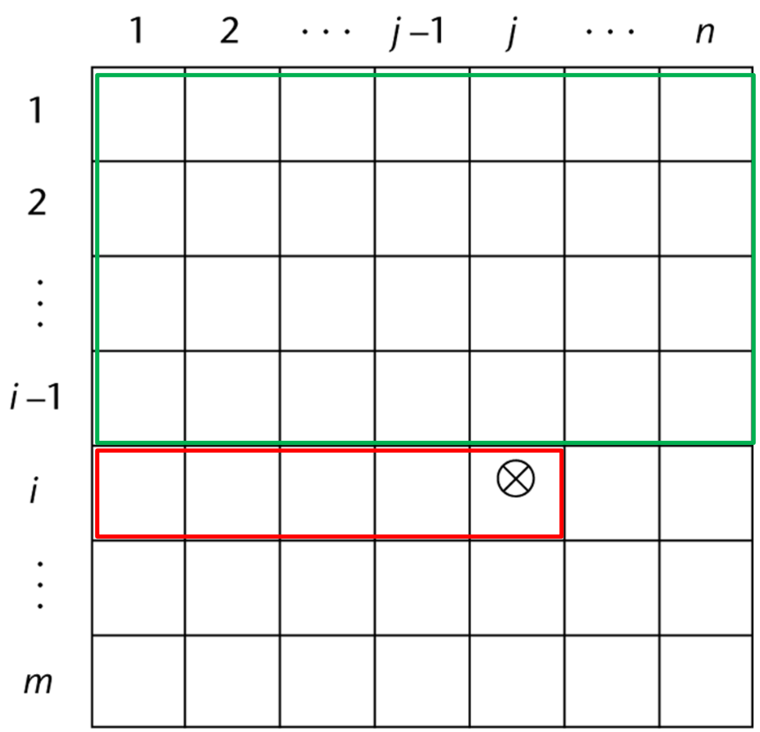
| |
|---|
| Multidimensioned array |
| Element type |
| Index type |
| Number of dimensions |
| Index range 0 |
| ⋮ |
| Index range n – 1 |
| Address |

# Locating an Element in a Multi-dimensioned Array

- General format

$$\text{Location } (A[i, j]) = \text{address of } A\ [\text{row\_lb}, \text{col\_lb}]$$

$$+ \left\{ \underbrace{\left( \boxed{(i - \text{row\_lb}) * n} \right) + \boxed{(j - \text{col\_lb})}}_{\text{i.e. the \# of elements in the array}} \right\} \times \text{element\_size}$$



# Compile-Time Descriptors

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

Single-dimensioned array

| |
|---|
| Multidimensioned array |
| Element type |
| Index type |
| Number of dimensions |
| Index range 0 |
| ⋮ |
| Index range n – 1 |
| Address |

Multidimensional array

# Associative Arrays

- An *associative array* is an *unordered collection* of data elements that are indexed by an equal number of values called *keys.*
  - User-defined keys must be stored.
    - key : *element*
  - Python 3.7+ : ordered vs. ~ Python 3.6: unordered
- Design issues:
  - What is the form of references to elements?
  - Is the size static or dynamic?
- Built-in type in:
  - Perl, Ruby - hash
  - Python, Swift - dictionary,
- Standard class library in Java, C++, C#, F#.

# Associative Arrays: Structure & Operations

In Perl:

- Names begin with **%** (#15-chap.5)
  - literals are delimited by *parentheses* ( )

```
%hi_temps=("Mon” => 77, "Tue" => 79, "Wed" => 65, … );
```

- Subscripting begins with $ and is done using *braces*{ } and *keys*

```
$hi_temps{"Wed” } = 83;
```

  - Element removal with **`delete`**

e.g.) `delete $hi_temps{"Tue"};`

In Python:

- E.g.)

```
car = {
        "brand": "Ford",
        "model": "Mustang",
        "year": 1964 }
car[“year”] = 1970;   del car[“year”]
```

# Record Types

- A *record* is a possibly *heterogeneous* aggregate of data elements in which the individual elements are identified by *names*.
- The elements are of potentially different sizes and reside in *adjacent memory locations*. Cf) heterogeneous array (#27)
- Design issues:
  - What is the syntactic form of references to the field?
  - Are elliptical references allowed? (ref. #42)
- Supported in COBOL, Pascal
  - Pascal:

```
type
record-name = record
   field-1: field-type1;
   field-2: field-type2;
   ...
   field-n: field-typen;
end;
```

```
type
Books = record
   title: packed array [1..50] of char;
   author: packed array [1..50] of char;
   subject: packed array [1..100] of char;
   book_id: integer;
end;
```

# Record Types (cont.)

- In C, C++, C#, Swift: supported with the **`struct`** data type.
- E.g.) In C:

```
struct MyStructure {      // Structure declaration
     int myNum;           // Member (int variable)
     char myLetter;       // Member (char variable)
 } C++ ;                  // End the structure with a semicolon
```

- In C++:
  - `struct` is a Stack-allocated value type and default access is public.
  - `class` object is a heap-allocated reference type and default is private.
- In Python ?
- Used as encapsulation structures, rather than data structures – chap. 11

# Definition of Records in COBOL

- COBOL uses level numbers to show *nested records*; others use a recursive definition.

```
01 EMP-REC.
   02 EMP-NAME.
      05 FIRST PIC X(20).
      05 MID   PIC X(10).
      05 LAST  PIC X(20).
   02 HOURLY-RATE PIC 99V99.
```

Level numbers for the *hierarchical structure* of the record.

# References to Records

- Record field references

  1. COBOL

  field_name OF record_name_1 OF ... OF record_name_n

  -- innermost record that contains the field to outermost record

  2. Others (*dot* notation) – outermost to innermost

  record_name_n.record_name_*n*-1.....record_name_1.field_name

- *Fully qualified references must include all record names.*
- E.g.) `MID OF EMP-NAME OF EMP-REC`

  or `EMP-REC.EMP-NAME.MID`

- *Elliptical references* allow *omitting enclosing record names* as long as the reference is unambiguous. E.g.) in COBOL

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

# Evaluation and Comparison: Record vs. Array

- Records are used

  when a collection of data values is *heterogeneous*

  *vs.*

  *A*rrays are used for the *homogeneous* data values.

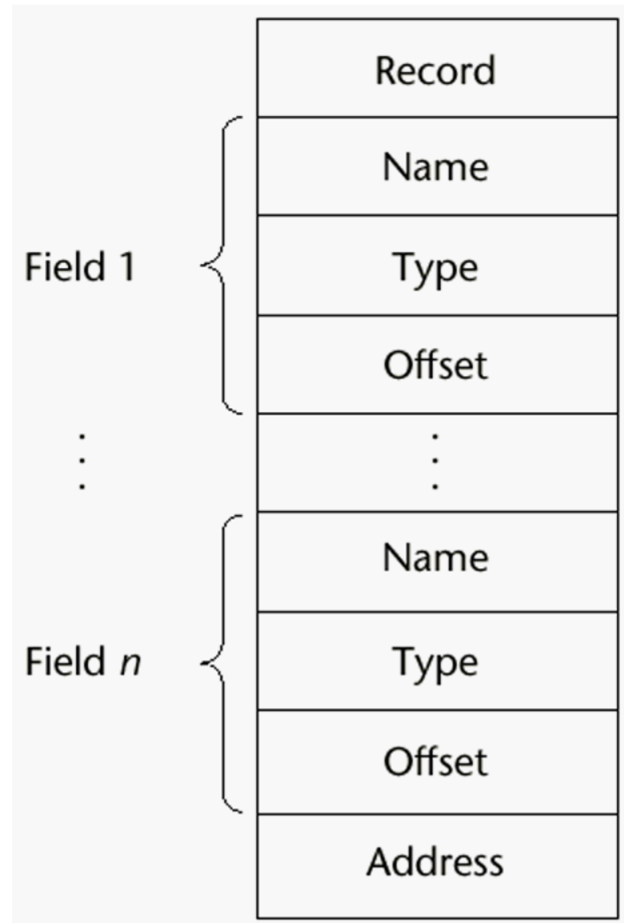
- Access to array elements is much slower because *subscripts are dynamic*

  *A*ccess to record fields is faster -- *field names are static*.

- Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower.

# Implementation of Record Type

- The fields of records are stored in *adjacent memory locations*.
- Since 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 offset address.



# Tuple Types

- A *tuple* is a data type that is similar to a record, except that the *elements are not named*.
- Used in Python, ML, and F# to allow functions to *return multiple values*.
  - Python
    - Closely related to its lists, but *immutable* - *its elements cannot be changed*
    - Create with a tuple literal: `myTuple =(3, 5.8, 'apple')`
    - Referenced with subscripts (begin at 1), `myTuple[1]`
    - Concatenation with '`+`' operator and deleted with **`del`**

# Tuple Types (cont.)

In Python, (ref. zybooks)

- *Named tuple* allows a user to define a new simple data type that consists of named attributes.
- namedtuple: a name of the package must be *imported* to create a new named tuple
  - from collections import namedtuple

| Car | | | | | |
|---|---|---|---|---|---|
| | make | model | price | hosepower | seats |
| chevy_blazer | Chevrolet | Blazer | 32000 | 275 | 8 |
| chevy_impala | Chevrolet | Impala | 37495 | 305 | 5 |

```
from collections import namedtuple

Car = namedtuple('Car', ['make','model','price','horsepower','seats'])  # Create the named tuple

chevy_blazer = Car('Chevrolet', 'Blazer', 32000, 275, 8)  # Use the named tuple to describe a car
chevy_impala = Car('Chevrolet', 'Impala', 37495, 305, 5)  # Use the named tuple to describe a different car

print(chevy_blazer)
print(chevy_impala)
```

```
Car(make='Chevrolet', model='Blazer', price=32000, horsepower=275, seats=8)
Car(make='Chevrolet', model='Impala', price=37495, horsepower=305, seats=5)
```

- A data object's attributes can be accessed using dot notation, as in name*.attribute*.
  - E.g.) chevy_blazer*.make*, chevy_blazer*.model*, .. chevy_impala*.seats*.
  - E.g.) chevy_impala.horsepower → 305

# Tuple Types (cont.)

- ML

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

  - Reference: `#n(Tuple_name)` – access $n^{th}$ field of tuple.

    e.g.) `#1(myTuple)` is the first element → 3.

  - A new tuple type can be defined.

    `type intReal = int * real;` a type with integer and real

    (The asterisk is just a separator)

- F#

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

  `let a, b, c = tup` - assigns a tuple to a tuple pattern `(a,b,c).`

  So, `a = 3, b = 5, c = 7.`

# List Types

- Lists was first supported in functional language.
- List in *Lisp* and *Scheme* are delimited by parentheses and use *no commas*.

  `(A B C D)` and `(A (B C) D)`

- Data and code have the same form:
  - As data: `(A B C)` is literally what it is.
  - As code: `(A B C)` is the function `A` applied

    to the parameters `B` and `C`.

- The interpreter needs to distinguish them

  → if it is data, we quote it with an *apostrophe ':*

  `'(A B C)` is data.

# List Types (cont.)

- List Operations in Scheme
  - `CAR:` returns the *1st element (atom)* of its list parameter:

    `(CAR '(A B C))` returns `A`
  - `CDR:` returns the *remainder of its list parameter* after the 1st element has been removed:

    `(CDR '(A B C))` returns `(B C)`
  - `CONS:` puts its 1st parameter into its 2nd parameter, a list, to make a new list:

    `(CONS 'A (B C))` returns `(A B C)`
  - `LIST:` returns a new list of its parameters

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

# List Types (cont.)

- List Operations in ML
  - Lists are written in *brackets* and the elements are separated by *commas*.
  - List elements must be of the same type.
  - The Scheme `CONS` function is a binary operator in ML, **::**

    `3 :: [5, 7, 9]` evaluates to `[3, 5, 7, 9]`
  - The **`hd`** and **`tl`** are `CAR` and `CDR` functions in Scheme, respectively.

    `hd [5, 7, 9]` → `5`, `tl [5, 7, 9]` → `[7, 9]`

# List Types (cont.)

- F# Lists
  - Like those of ML, except elements are separated by *semicolons* and `hd` and `tl` are *methods* of the `List` class.
  - `List.hd [1; 3; 5; 7]` → `1`
- Python Lists
  - The list data type also serves as Python's arrays.
  - Python's lists are *mutable*, unlike Scheme, Common Lisp, ML, and F# in FL: cf) tuple (ref. #45)
  - Elements can be of any type – *heterogeneous aggregation*.
  - Create a list with an assignment

```
myList = [3, 5.8, "grape"]
myList[1] = “apple” → [3, “apple”, "grape"]
```

# List Types (cont.)

- Python Lists (cont.)

```
myList = [3, 5.8, "grape"]
```

  - List elements are referenced with subscripting, with indices beginning at *zero*.

    `x = myList[1]` sets `x` to `5.8`

  - List elements can be deleted with `del`

    ```
    del myList[1]
    ```

  - List Comprehensions – derived from set notation

    ```
    [x * x for x in range(6) if x % 3 == 0]
    ```

    `range(12)` creates `[0, 1, 2, …, 11]`

    Constructed list: `[0, 9, 36, 81]`

# List Types (cont.)

- Haskell's List Comprehensions
  - The original: `[body | quantifiers]`

    `[n * n | n <- [1..10]]`
- F#'s List Comprehensions

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

→ `[1; 4; 9; 16; 25;]`

- Both C# and Java support lists through their generic *heap-dynamic collection classes*, `List` and `ArrayList`, respectively. -- ref. #26 for an example.

# Summary

- The data types of a language are a large part of what determines that language's style and usefulness
- The primitive data types of most imperative languages include numeric, character, and Boolean types
- The user-defined enumeration and subrange types are convenient and add to the readability and reliability of programs
- Arrays and records are included in most languages
- Pointers are used for addressing flexibility and to control dynamic storage management