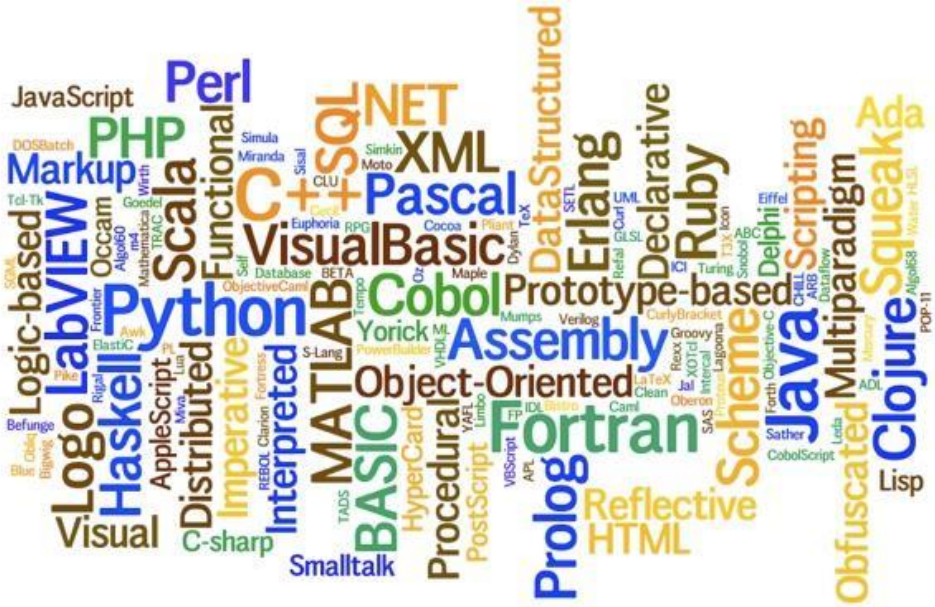


Programming Languages
Data Types

Programming Languages
Module 7

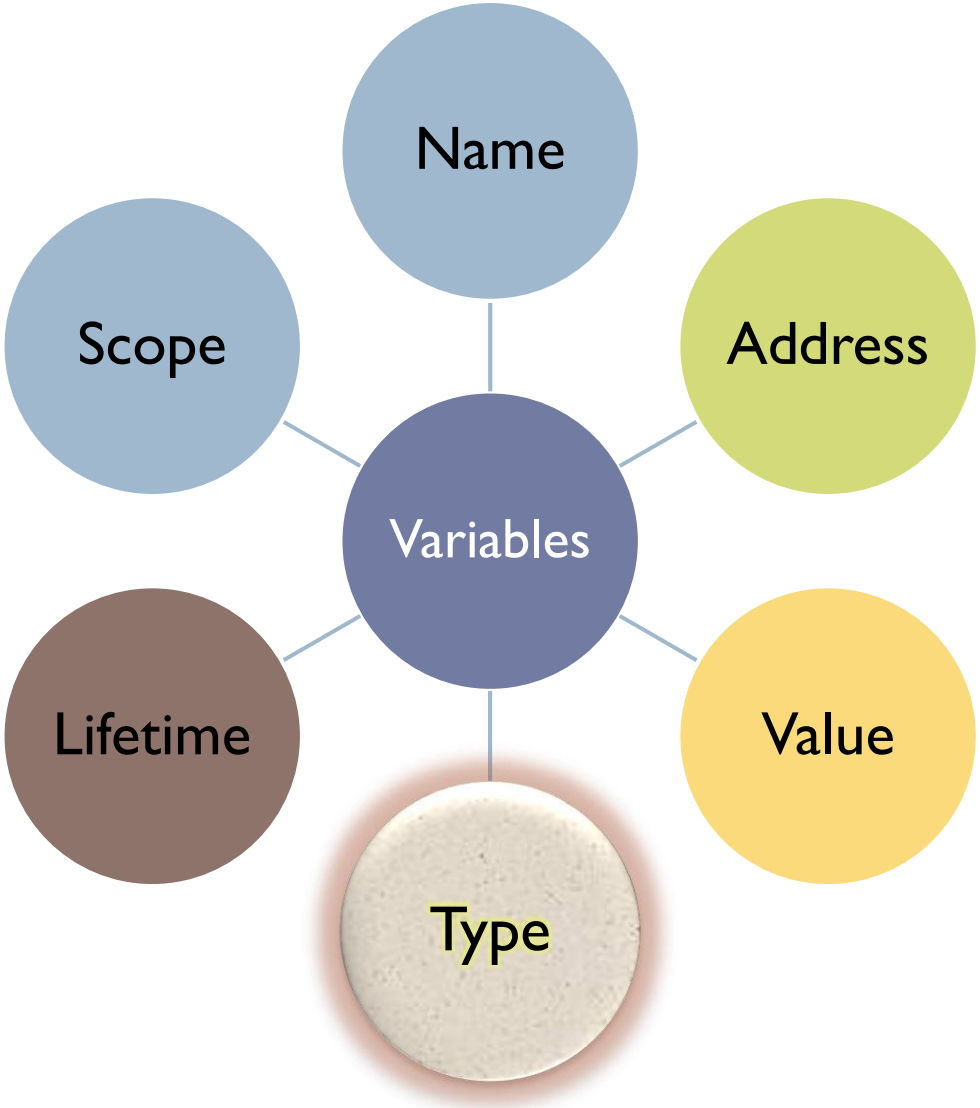
Dr. Tamer ABUHMED
College of Computing

SUNG KYUN KWAN UNIVERSITY

Important Topics (Chapter6)

- What is a "type system"?
- What are common data types?
- How are numeric types stored and operated on?
- Compound types: arrays, struct, records
- Enumerated types
- character and string types
- Tuple Types and List Types
- Pointer and Reference Types
- Strong type checking versus not-so-strong
  - enumerations, type compatibility, and type safety
- Advantages & disadvantages of compile-time checking

Variable Attributes



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

Symbol	Kind	Type	Address	Value	Attributes
b	var	int	2350CA00	5	....

- An *object* represents an instance of a user-defined (abstract data) type

  Student engStudent = new Student(……..);

- One design issue for all data types: What operations are defined and how are they specified?

Primitive Data Types

- Almost all programming languages provide a set of *primitive data types*
- Primitive data types: Those not defined in terms of other data types
- Some primitive data types are merely reflections of the hardware
- Others require only a little non-hardware support for their implementation

Primitive Data Types: 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
- C++: short int, unsigned short int, unsigned int, int, long int, unsigned long int, long long int, unsigned long long int
- Java's signed integer sizes: **`byte, short, int, long`**
- Python: only one type of **signed integers:** int

Integer Data Types

C/C++ support both "unsigned" and "signed" integer types.

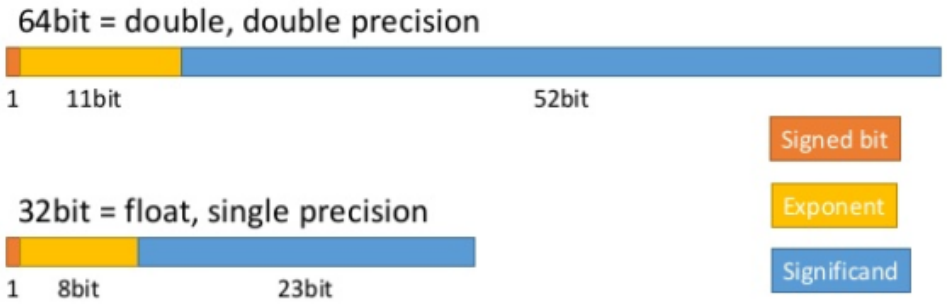
Type	# Bytes	Range of values
short int	2	-32,768 (-2^{15}) to 32,767 ($2^{15} - 1$)
unsigned short	2	0 to 65,535 ($2^{16} - 1$)
int	4	-2,147,483,648 (-2^{31}) to 2,147,483,647 ($2^{31} - 1$)
unsigned int	4	0 to 4,294,967,295 ($2^{32} - 1$)
long int	same as "int" on Pentium and Athlon CPU	

C permits "char" type for integer values, too…

char	1	-128 to 127
unsigned char	1	0 to 255

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`** and **`double`**; sometimes more
- Usually exactly like the hardware, but not always
- IEEE Floating-Point Standard 754



Floating point type	Memory requirement	Range
Float	4 bytes	±3.40282347E+38F i.e. 6-7 significant digits
Double	8 bytes	±1.79769313486231570E+308 i.e. 15-16 significant digits

Primitive Data Types: Complex

- Some languages support a complex type, e.g., C99, Fortran, and Python
- Each value consists of two floats, the real part and the imaginary part
- Literal form (in Python):

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

Primitive Data Types: Decimal

- For business applications (money)
  - Essential to COBOL
  - C# offers a decimal data type
- Store a fixed number of decimal digits, in coded form (BCD)
- *Advantage*: accuracy
- *Disadvantages*: limited range, wastes memory

Decimal Number	BCD Code
0	0000 0000
1	0000 0001
2	0000 0010
3	0000 0011
4	0000 0100
5	0000 0101
6	0000 0110
7	0000 0111
8	0000 1000
9	0000 1001
10 (1+0)	0001 0000
11 (1+1)	0001 0001
12 (1+2)	0001 0010
…	…
20 (2+0)	0010 0000
21 (2+1)	0010 0001
22 (2+2)	0010 0010
etc, continuing upwards in groups of four	

Primitive Data Types: Boolean

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

Primitive Data Types: Character

- Stored as numeric coding
- Most commonly used coding: ASCII
- An alternative, 16-bit coding: Unicode (UCS-2)
  - Includes characters from most natural languages
  - Originally used in Java
  - C# and JavaScript also support Unicode
- 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
  - Assignment and copying
  - Comparison (=, >, etc.)
  - Catenation
  - Substring reference
  - Pattern matching

Character String Type in Certain Languages

- C and C++
  - Not primitive
  - Use **`char`** arrays and a library of functions that provide operations
- SNOBOL4 (a string manipulation language)
  - Primitive
  - Many operations, including elaborate pattern matching
- Fortran and Python
  - Primitive type with assignment and several operations
- Java
  - Primitive via the `String` class
- Perl, JavaScript, Ruby, and PHP

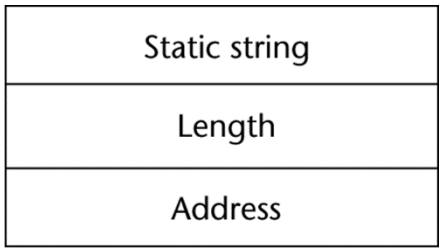
  - Provide built-in pattern matching, using regular expressions

Character String Length Options

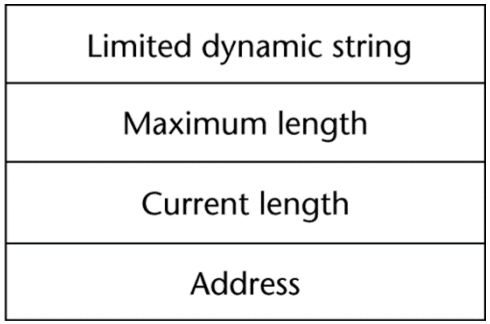
- Static: COBOL, Java's `String` class, Python
- *Limited Dynamic Length*: C and C++
  - In these languages, a special character is used to indicate the end of a string's characters, rather than maintaining the length
- *Dynamic* (no maximum): SNOBOL4, Perl, JavaScript
- Ada supports all three string length options

Compile- and Run-Time Descriptors

- 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



Compile-time descriptor for static strings



Run-time descriptor for limited dynamic strings

User-Defined Ordinal Types

- An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers
- Examples of primitive ordinal types in Java
  - `integer`
  - `char`
  - `boolean`
- There are two kinds of ordinal types: **enumeration** and **subrange**

Enumeration Types

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

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

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

Evaluation of Enumerated Type

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

Enumerated Type

- C++

```
enum week_days {
Sunday,
Monday=1,
Tuesday,
Wednesday=1,
Thursday,
Friday=9,
Saturday
};

int main() {
 week_days variable = Thursday;
 //variable= 5; error
 // variable++; error
 variable= (week_days)5;
 int i = Sunday;
 // i becomes 0
 int j = 3 + Monday;
 // j becomes 4
 return 0;
}
```

Java

```
class Ideone
{

enum BookType { HARDCOPY,EBOOK };
//where the semicolon is optional
public static void main (String[] args)
  {

BookType bt1 = BookType.HARDCOPY;
BookType bt2 = BookType.EBOOK;
System.out.println(bt1);

  }
}
```

Enumerated Type

Java Code Example

```
public class MainClass {
  enum Week {
  Monday, Tuesday, Wednesday, Thursday, Friday, Saturaday, Sunday  }

public static void main(String args[]) {
 // Obtain all ordinal values using ordinal().
  System.out.println("Here are all week constants"
  + " and their ordinal values: ");
  for (Week day : Week.values())
     System.out.println(day + " " + day.ordinal());
    }
 }
```

Output

```
Here are all week constants and their ordinal values:
Monday 0
Tuesday 1
Wednesday 2
Thursday 3
Friday 4
Saturaday 5
Sunday 6
```

Subrange type

- Set up the type to support range checking.
- Contiguous subsequence of an ordinal type.
- It is for readability and writability.
- e.g. Ada's code

```
type Days is (mon, tue, wed, thu, fri, sat, sun);
subtype Weekdays is Days range mon..fri;
subtype Index is Integer range 1..100;

Day1: Days;
Day2: Weekday;
Day2 := Day1;
```

Subrange Evaluation

- Aid to readability
  - Make it clear to the readers that variables of subrange can store only certain range of values
- Reliability
  - Assigning a value to a subrange variable that is outside the specified range is detected as an error

Implementation of User-Defined Ordinal Types

- Enumeration types are implemented as integers
- Subrange types are implemented like the parent types with code inserted (by the compiler) to restrict assignments to subrange variables

Important Topics

- type compatibility
  - what are compatibility rules in C, C++, and Java?
  - when are user-defined types compatible?
- type conversion
  - what conversions are automatic in C, C++, Java?
  - what conversions are allowed using a cast?

Type Compatibility for built-in types

- Operations in most languages will automatically convert ("promote") some data types:

  2 * 1.75 convert 2 (int) to floating point

- Assignment compatibility: what automatic type conversions are allowed on assignment?

```
int n = 1234567890;
float x = n;  // OK is C or Java
n = x;        // allowed in C?  Java?
```

- char -> short -> int -> long -> double

  short -> int -> float -> double

- What about long -> float ?
- Rules for C/C++ not same as Java.

C/C++ Arithmetic Type Conversion

- For +, -, *, /, both operands must be the **same type**
- C/C++ compiler "promotes" mixed type operands to make all operands same using the following rules:

Operand Types	Promote	Result
short *op* int	short => int	int
long *op* int	int => long	long
int *op* float	int => float	float
int *op* double	int => double	double
float *op* double	float => double	double
etc...		

"op" is any arithmetic operation: + - * /

Assignment Type Conversion is not Arithmetic Type Conversion (1)

- What is the result of this calculation?

```
int m = 15;
int n = 16;
double x = m / n;
```

Forcing Type Conversion

- Since arguments are integer, integer division is used:
  double x = 15 / 16; // = 0 !
- you must coerce "int" values to floating point.
  There are two ways:

```
int m = 15;
int n = 16;
/** Efficient way: cast as a double */
double x = (double)m / (double)n ;

/** Clumsy way: multiply by a float (ala Fortran) */
double x = 1.0*m / n;
```

Assignment Type Conversion is not Arithmetic Type Conversion (2)

- Many students wrote this in Fraction program:

```
public class Fraction {
     int numerator;          // numerator of the fraction
     int denominator;        // denominator of the fraction
...etc...
     /** compare this fraction to another. */
     public int compareTo( Fraction frac ) {
          double r1 = this.numerator / this.denominator;
          double r2 = frac.numerator / frac.denominator;
          if ( r1 > r2 ) return 1;
          else if ( r1 == r2 ) return 0;
          else return -1;
     }
```

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.

1	2	3	4	5	9	10	25

a	b	n
t	o	u
m	c	v

1	2	3	4	5	9	10	25
8	5	7	65	9	7	11	40
50	58	8	99	2	1	0	30

Array Design Issues

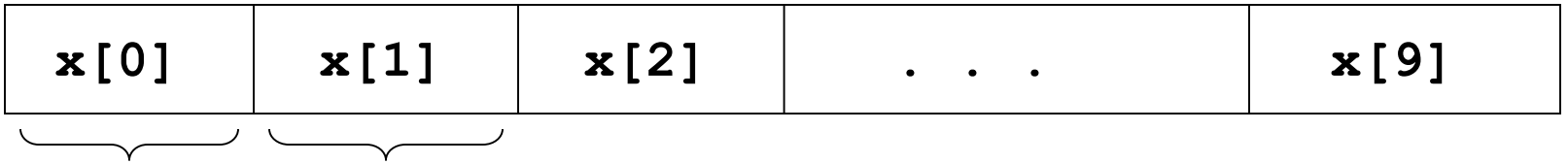
- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does allocation take place?
- Are ragged or rectangular multidimensional arrays allowed, or both?
- What is the maximum number of subscripts?
- Can array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

Arrays

An array is a series of elements of the same type, with an index, which occupy consecutive memory locations.

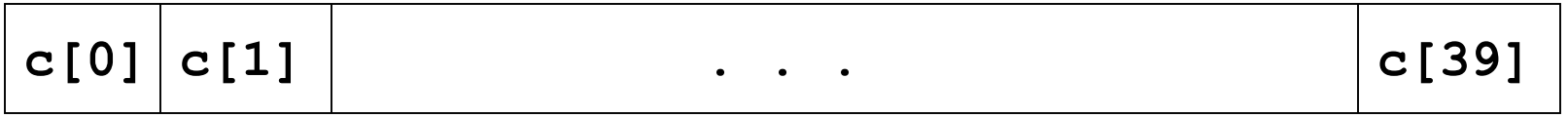
```
float x[10];                    // C: array of 10 “float” vars
char [] c = new char[40];       // Java: array of 40 "char"
```

Array x[ ] in memory:



4 Bytes = sizeof(float)

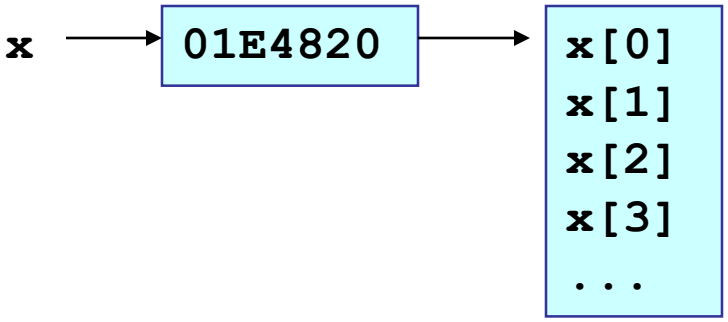
Array c[ ] in memory :



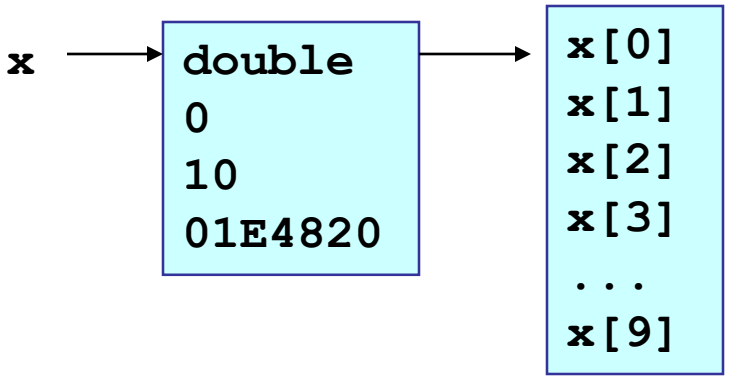
Array "dope vector"

- In C or Fortran an array is just a set on continuous elements. No type or length information is stored.
- Some languages store a "dope vector" (aka *array descriptor*) describing the array.

```
/* C language */
double x[10];
```




```
/* Language with dope */
double x[10];
```

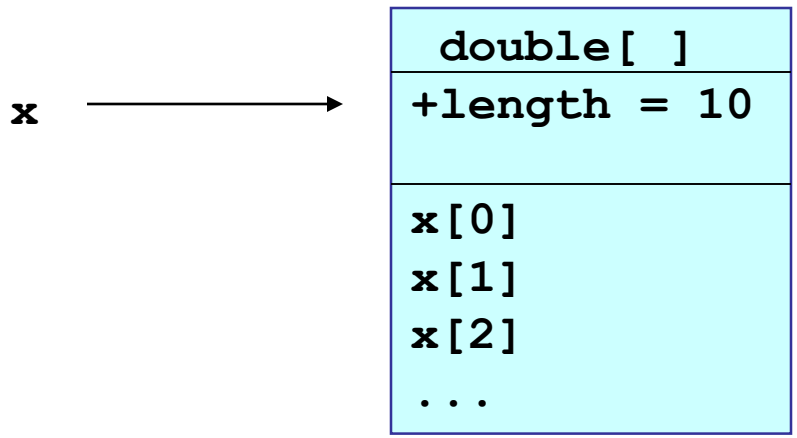


Array as Object

- In Java, arrays are objects:

  `double [ ] x = new double[10];`

- **`x`** is an Object; **`x[10]`** is a double (primitive type).



`x.getClass( ).toString( )` returns `"[D"`

Array Indexing

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

  array_name (index_value_list) → an element

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

Arrays Index (Subscript) Types

- FORTRAN, C: integer only
- Ada: integer or enumeration (includes Boolean and char)
- Java: integer types only
- Index range checking
  - C, C++, Perl, and Fortran do not specify range checking
  - Java, ML, C# specify range checking
  - In Ada, the default is to require range checking, but it can be turned off

Subscript Binding and Array Categories

- **Static array**: an array whose size is known, and whose storage is allocated, at compile time. In C, you might write at global (file) scope:

```
static int static_array[7];
```

- Advantage: efficiency (no dynamic allocation)

- **Fixed stack-dynamic array**: you know the size of your array at compile time, but allow it to be allocated automatically on the stack (the size is fixed at compile time but the storage is allocated when you enter its scope, and released when you leave it)
- Advantage: space efficiency

```
void foo() {
int fixed_stack_dynamic_array[7];
      /* ... */ }
```

Subscript Binding and Array Categories (continued)

- *Stack-dynamic*: subscript ranges are dynamically bound and the storage allocation is dynamic (done at run-time)
- **For Example:** you don't know the size until runtime, eg. C99 allows this:

```
void foo(int n)
{ int stack_dynamic_array[n];
   /* ... */ }
```

  - Advantage: flexibility (the size of an array need not be known until the array is to be used)

Subscript Binding and Array Categories (continued)

- *Fixed heap-dynamic*: similar to fixed stack-dynamic: storage binding is dynamic but fixed after allocation (i.e., binding is done when requested and storage is allocated from heap, not stack)

```
int * fixed_heap_dynamic_array = malloc(7 * sizeof(int));
```

  - using explicit heap allocation

- Heap-dynamic: binding of subscript ranges and storage allocation is dynamic and can change any number of times
  - Advantage: flexibility (arrays can grow or shrink during program execution)

```
void foo(int n)
{ int * heap_dynamic_array = malloc(n * sizeof(int));
 }
```

Subscript Binding and Array Categories (continued)

- C and C++ arrays that include `static` modifier are static
- C and C++ arrays without `static` modifier are fixed stack-dynamic
- C and C++ provide fixed heap-dynamic arrays
- C# includes a second array class `ArrayList` that provides fixed heap-dynamic
- Perl, JavaScript, Python, and Ruby support heap-dynamic arrays

Array Initialization

- Some language allow initialization at the time of storage allocation
  - C, C++, Java, C# example

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

  - Character strings in C and C++

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

  - Arrays of strings in C and C++

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

  - Java initialization of String objects

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

Heterogeneous Arrays

- A *heterogeneous array* is one in which the elements need not be of the same type
- Supported by Perl, Python, JavaScript, and Ruby
- Python Example:

I	R	hello	4	5.7	9	false	25

```
>>> def a(): pass
>>>
lst=[1,'one',{1:'one'},a,[1,1],(1,),True,set((1,))]
>>> for each in lst:
...     print type(each), str(each)
...
<type 'int'> 1
<type 'str'> one
<type 'dict'> {1: 'one'}
<type 'function'> <function a at 0x100496938>
<type 'list'> [1, 1]
<type 'tuple'> (1,)
<type 'bool'> True
<type 'set'> set([1])
```

T	**4**	**n**
t	**5**	**F**
5	**c**	**v**

Rectangular and Jagged Arrays

- A rectangular array is 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
- A jagged matrix has rows with varying number of elements
  - Possible when multi-dimensioned arrays actually appear as arrays of arrays
- C, C++, and Java support jagged arrays
- Fortran, Ada, and C# support rectangular arrays (C# also supports jagged arrays)

Slice

- Python

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

`vector (3:6)` is a three-element array

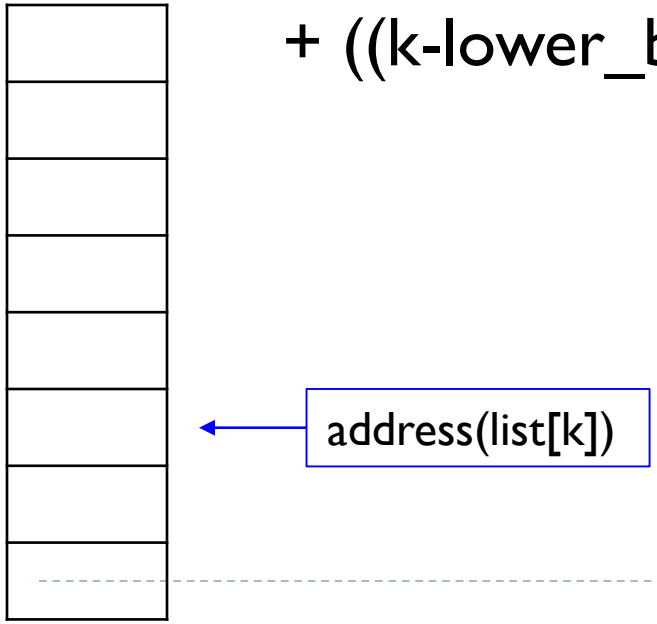
`mat[0][0:2]` is the first and second element of the first row of `mat`

- Ruby supports slices with the `slice` method

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

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



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

Compile-time descriptor for single-dimensioned arrays

Accessing Multi-dimensioned Arrays

- Two common ways:
  - Row major order (by rows)
    – used in most languages
  - Column major order (by columns)
    – used in Fortran
  - For example:

$$\begin{matrix} 3 & 4 & 7 \\ 6 & 2 & 5 \\ 1 & 3 & 8 \end{matrix}$$

- it would be stored in row major order as
  - 3, 4, 7, 6, 2, 5, 1, 3, 8
- In column major order, it would have the following order in memory:
  - 3, 6, 1, 4, 2, 3, 7, 5, 8

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 1
⋮
Index range *n*
Address

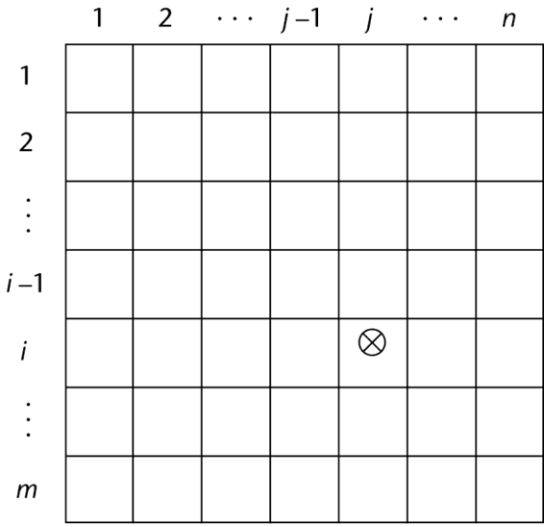
A compile-time descriptor for a multidimensional array

Locating an Element in a Multi-dimensioned Array

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)

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

- General format
  Location (a[I,j]) = address of a [row_lb,col_lb] + (((I – row_lb) * n) + (j – col_lb)) * element_size



Associative Arrays

- An *associative array* is an unordered collection of data elements that are indexed by an equal number of values called *keys*
  - User-defined keys must be stored
- Design issues:

  - What is the form of references to elements?

  - Is the size static or dynamic?
- Built-in type in Perl, Python, Ruby, and Lua
  - In Lua, they are supported by tables
  - Python's associative arrays, are called **dictionaries**

Tuple Types

- A tuple is a data type used in Python, ML, and F# to allow functions to return multiple values
  - Python
    - Closely related to its lists, but immutable
    - Create with a tuple literal

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

      Referenced with subscripts (begin at `1`)

    Catenation with + and deleted with **`del`**

```
def f(x):
        y0 = x + 1
        y1 = x * 3
        y2 = y0 ** y1
        return (y0,y1,y2)  #return multiple values

A,B,C = f(2)

print(A,B,C)               # 3  6  729
```

List Types

- Lists in LISP and Scheme are delimited by parentheses and use no commas

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

- F# Lists
  - elements are separated by semicolons and have same type
- Python Lists
  - The list data type also serves as Python's arrays
  - Python's lists are mutable
  - Elements can be of any type
  - Create a list with an assignment

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

Pointer and Reference Types

- A *pointer* type variable has a range of values that consists of memory addresses and a special value, *nil*
- Provide the power of indirect addressing
- Provide a way to manage dynamic memory
- A pointer can be used to access a location in the area where storage is dynamically created (usually called a *heap*)

Design Issues of Pointers

- What are the scope of and lifetime of a pointer variable?
- What is the lifetime of a heap-dynamic variable?
- Are pointers restricted as to the type of value to which they can point?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer types, reference types, or both?

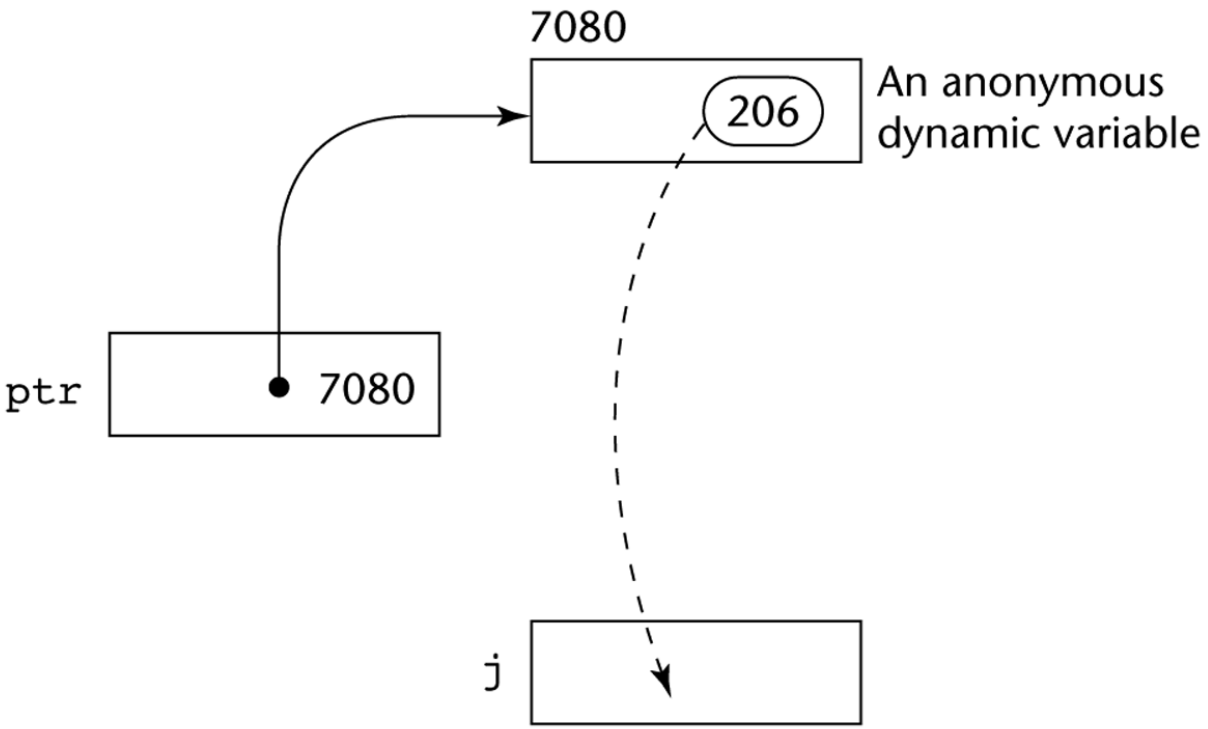
Pointer Operations

- Two fundamental operations: assignment and dereferencing
- Assignment is used to set a pointer variable's value to some useful address
- Dereferencing yields the value stored at the location represented by the pointer's value
  - Dereferencing can be explicit or implicit
  - C++ uses an explicit operation via *

    `j = *ptr`

    sets j to the value located at `ptr`

Pointer Assignment Illustrated



The assignment operation j = *ptr

Problems with Pointers

- Dangling pointers (dangerous)
  - A pointer points to a heap-dynamic variable that has been deallocated
- Lost heap-dynamic variable
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)
    - Pointer `p1` is set to point to a newly created heap-dynamic variable
    - Pointer `p1` is later set to point to another newly created heap-dynamic variable
    - The process of losing heap-dynamic variables is called *memory leakage*

Pointers in C and C++

- Extremely flexible but must be used with care
- Pointers can point at any variable regardless of when or where it was allocated
- Used for dynamic storage management and addressing
- Pointer arithmetic is possible
- Explicit dereferencing and address-of operators
- Domain type need not be fixed (`void *`)

  `void *` can point to any type and can be type checked (cannot be de-referenced)

Pointer Arithmetic in C and C++

```
float stuff[100];
float *p;
p = stuff;
```

`*(p+5)` is equivalent to `stuff[5]` and `p[5]`

`*(p+i)` is equivalent to `stuff[i]` and `p[i]`

Reference Types

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

Evaluation of Pointers

- Dangling pointers and dangling objects are problems as is heap management
- Pointers are like `goto`'s--they widen the range of cells that can be accessed by a variable
- Pointers or references are necessary for dynamic data structures--so we can't design a language without them

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments
- *Type checking* is the activity of ensuring that the operands of an operator are of compatible types
- A *compatible type* is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler- generated code, to a legal type
  - This automatic conversion is called a *coercion*.
- A *type error* is the application of an operator to an operand of an inappropriate type

Type Checking (continued)

- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic
- A programming language is *strongly typed* if type errors are always detected
- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors

Strong Typing

Language examples:

- C and C++ are not
- Ada is, almost (`UNCHECKED CONVERSION` is loophole)

  (Java and C# are similar to Ada)

- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus Ada)

- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada

Theory and Data Types

- Type theory is a broad area of study in mathematics, logic, computer science, and philosophy
- Two branches of type theory in computer science:
  - Practical – data types in commercial languages
  - Abstract – typed lambda calculus

- A type system is a set of types and the rules that govern their use in programs

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