Lecture 4: Data Types

Chapter 6

Lecture 4 (Chapter 6) Topics

- Introduction
- Primitive Data Types
- Character String Types
- Enumeration Types
- Array Types
- Associative Arrays
- Tuple Types
- List Types
- Pointers and Reference Types
- Type Checking
- Strong Typing
- Type Equivalence

Primitive Types

- Introduction
  - A *data type* defines a collection of data objects and a set of predefined operations on those objects, e.g
    - Java int type
      - A collection of data objects: –maxint .. Maxint
      - A set of predefined operations: +, –, *, /, …
    - Java String type
      - A collection of data objects: all valid strings
      - A set of predefined operations: + (concatenation), length, [],…
  - An *object* represents an instance of a type, in some languages, referring to a user-defined abstract data type

- Design Issues
  - How data types are specified? What operations are defined on a data type?

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
  - byte, integer, float, double, …
- Others require only a little non-hardware support for their implementation
  - character, Boolean, …

Primitive Data Types: Integer and Floating Point

- Integer
  - Reflection of the hardware so the mapping is trivial
  - Some language support many different types of integer
    - Java's signed integer sizes: **`byte, short, int, long`**
    - **`C/C++?`**

- 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`**
  - IEEE Floating-Point Standard 754 single and double precision (CS3650)

Primitive Data Types: Complex and Decimal

- Complex
  - Some languages support a complex type, e.g. Fortran, and Python
  - Each value consists of two floats, the real part and the imaginary part, e.g. in Python 7 + 3j
  - Question: How do you code complex numbers in Java?

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

Primitive Data Types: Boolean and Character

- Boolean
  - Simplest of all
  - Range of values: two elements, “true” and “false”
  - Could be implemented as bits, but often as bytes
  - Advantage: readability
- Character
  - Stored as numeric coding, originally in ASCII
  - Unicode now most popular
    - Includes characters from most natural languages
    - Originally used in Java, now supported by many languages
    - UTF-8, UTF-16, UTF-32 (less common)

Character String Types

- Values are sequences of characters
- Design issues:
  - Is it a primitive type or an array of characters?
  - Should the length of strings be static or dynamic?
- Typical operations:
  - Assignment and copying
  - Comparison (=, >, etc.)
  - Catenation
  - Substring reference
  - Pattern matching
- Advantage
  - Aid to writability

Character String Types: Language Support

- C and C++: Not primitive, use *char arrays* and a library of functions that provide operations
- Python: Primitive type with assignment and several operations
- Java (and C#, Ruby, and Swift): Primitive via the *String* class
- Perl et al: built-in pattern matching, using regular expressions

- Character String Length Options
  - *Static*: COBOL, Java's `String` class
  - *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
    - Increased cost in implementation & efficiency

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: char, Boolean
- User-defined Ordinal Types
  - Subrange Types
    - Pascal: type positive = 0 .. MAXINT;
  - 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};

Enumeration Types

- 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?
- Evaluations
  - Aid to readability, e.g. code color as name not as 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
- Language support
  - C/C++
  - C# and Java provide better support for enumeration than C/C++

Primitive types: 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

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.
- 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

  a(i) = 5 //Ada, Fortran use ()

  a[i] = 5 //C++, Java use []

- Index/Subscript Types
  - integer commonly used, some may use ordinal type such as characters (e.g. Pascal, arr['a'] = 5;)
- Index range checking
  - C, C++, Perl, and Fortran do not specify range checking
  - Java, ML, C# specify range checking

- Range Checking

```
int sum[5];
//int[] sum = new int[5] in Java
for (int j=0; j<=5; j++)
        sum[j] = j*2;
… display sum[6] …
```

sum[0]
sum[4]
sum[5]
***

Observation: Java run-time error; C++ no error; C++ displays a garbage value for sum[6]. (Java enforces range-checking while C++ not.)

Subscript Binding and Array Categories

- Based on the subscript binding and storage allocation
  - Subscript binding time: when we know subscript range, i.e. how many elements in the array
  - Storage allocation: on stack or on heap
- Four categories of arrays
  - Static
  - Fixed stack dynamic
    - Stack dynamic available in Ada, obsolete now
  - Fixed heap dynamic
  - Heap dynamic

Array Categories

- *Static*: subscript ranges are statically bound and storage allocation is static (before run-time)

  ```
  static int mdays[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31} ; //C
  ```

  - Advantage: efficiency (no dynamic allocation)

- *Fixed stack-dynamic*: subscript ranges are statically bound, but the allocation is done at array elaboration (i.e. creation) time

  ```
  void getInput() {     //C++
          int data[100];
          ...
  }
  ```

  - No storage allocation until the method is called.
  - Advantage: space efficiency

Stack dynamic vs. fixed stack dynamic

- C++ example
- The following okay?

```
void getData (int size ) {
        double data[size];
        for (int i=0; i<size; i++) {
                data[i] = 2.1 * i;
                cout << data[i] << endl;
        }
}
```

Should use:

```
double *data = new double [size];
```

Theoretically such stack dynamic arrays are NOT supported by C++ (Ada does) but it works on many C++ compiler without warning or error (some GNU C++ version may support it, overall it's not safe to use it.)

Array Categories (continued)

- *Fixed heap-dynamic*: storage binding is dynamic but fixed after allocation (i.e., binding is done when array elaboration/created) and storage is allocated from heap, not stack.

```
double *arr = new double [5];   //C++
```

- *Heap-dynamic*: binding of subscript ranges and storage allocation is dynamic and can change any number of times

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];   //JavaScript
fruits[6] = "Lemon";     // adds a new element (Lemon) to fruits
```

  - Advantage: flexibility (arrays can grow or shrink during program execution)

Questions

(1) What category for data and result arrays respectively?

```
public static void test (int size) {            //Java
        int data[5] = {1, 2, 3, 4, 5};
        int[] result = new int[size];
… }
```

(2) What category a Java ArrayList belong to?

```
ArrayList<String> cars = new ArrayList<String>();
cars.add("BMW");
cars.add("Ford");
cars.add("Mazda");
System.out.println(cars);
```

Java arrays vs. C++ arrays

- All Java arrays on heap

- C++: arrays on stack as well as on heap

```
int a[100];              //stack
int *ha = new int [100];  //heap
```

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
  - e.g. dataArr = [35, 3.5, “n/a”]
  - Traditionally we restrict “arrays” to homogeneous elements only, but the trend now extends to heterogeneous elements

Language Support

- C and C++
  - static arrays: static modifier
  - fixed stack-dynamic: popular ones
  - fixed heap-dynamic arrays: new operator
- Perl, JavaScript, Python, and Ruby support heap-dynamic arrays
  - Heap dynamic array good at supporting heterogeneous elements

- Questions:
  - What category does Java ArrayList belong? C# ArrayList?
  - What is the primary challenge for supporting heterogeneous arrays in fixed stack/heap dynamic categories?

Array Initialization

- Some language allow initialization at the time of storage allocation (e.g. C, C++, Java, Swift, and C#)

```
int list [] = {4, 5, 7, 83}; //C-based
char name [] = "freddie";    //C-based
char *names [] = {"Bob", "Jake", "Joe"]; //C-based
String[] names = {"Bob", "Jake", "Joe"};  //Java
```

- Python
  - List comprehensions

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

#this initialization will put [0, 9, 36, 81] in `list`

Arrays Operations

- Traditionally very limited array operations supported
  - In C, array assignments and comparisons are based on pointer operations. Have to write functions to copy or compare arrays.
  - With the development of object-oriented programming, arrays could be treated as objects and many operations now can be built upon array objects.
- Language Examples
  - APL provides the most powerful array processing operations for vectors and matrixes as well as unary operators (for example, to reverse column elements)
  - Python's array assignments, but they are only reference changes. Python also supports array catenation and element membership operations
  - Ruby also provides array catenation

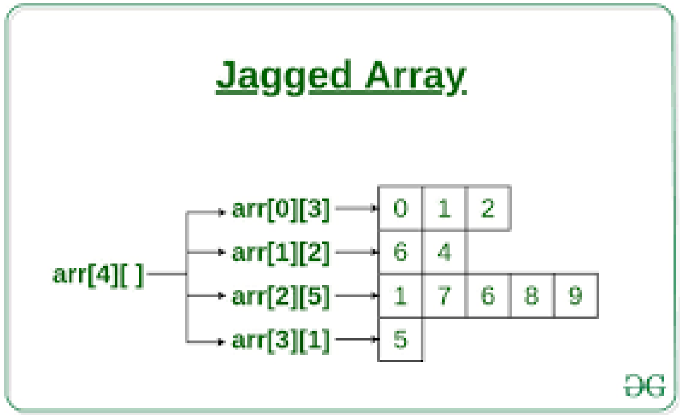
```
int arr1[] = { 10, 20, 30, 40, 50 };
int arr2[] = {0, 20, 30, 40, 50};
arr2[0] = 10;

if (arr1==arr2) {
        … "same" …;
}
else {
        … "not same"   …
}

//same or not same?  -- answer: not same
//== compares two array object references not array elements
```

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 array* has rows with varying number of elements
  - multi-dimensioned arrays may appear as arrays of arrays
- Many languages (C, C++, Java etc) support both rectangular arrays and jagged arrays

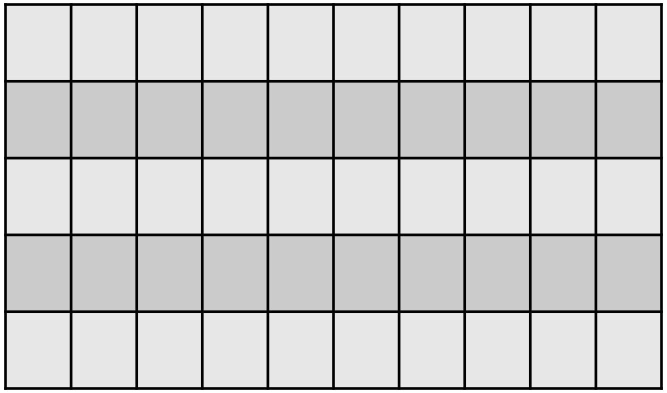


Source

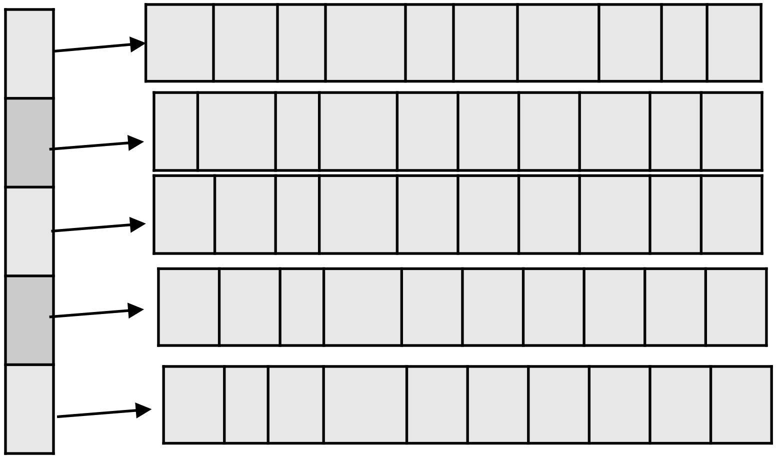
Compare/Contrast

- Any difference between the following two C++ arrays?
- Can you create Java arrays similar to each of the following?

```
double a[5][10];
```



```
double *a[5];
for (int j=0; j<5; j++) a[j] = new double [10];
```



Compare & Contrast

```
double a[5][10];
double *a[5];
for (int j=0; j<5; j++) a[j] = new double[10];
```

- Memory allocation: (1) stack; (2) heap   (C++)

- Memory organization:

(1) consecutive 50 memory locations (row-majored in C++)
(2) Rows may spread in different memory locations

- Access (in program): no difference

```
cout << a[i][j] ;
```

Slices

- A slice is some substructure of an array; nothing more than a referencing mechanism
- Example – Python

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
vector (3:6)  #a three-element array
mat[0][0:2]   #the 1st and 2nd element of the 1st row
```

- Ruby supports slices with the `slice` method

```
> fruit = ["apple", "banana", "orange", "grapefruit", "tomato"]
> fruit.slice(1,3)      # ["banana", "orange", "grapefruit"]
```

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
  - Also called dictionary, map, hashmap, etc.
- Design issues:
  - What is the form of references to elements?
  - Is the size static or dynamic?
- Built-in type in Perl, Python, Ruby, and Swift
- Example: Perl

```
%hi_temps = ("Mon" => 77, "Tue" => 79, "Wed" => 65, …);
$hi_temps{"Wed"} = 83;
```

Tuple Types

- A *tuple* is a data type for an ordered set of values
  - Similar to an old data type Record (in Ada)
  - In tuple, values don't need to be named
- Used in Python, ML, and F# to allow functions to return multiple values
  - Python
    - Closely related to Python lists, but immutable

```
myTuple = (3, 5.8, 'apple')
print(tup[1])    //displays 5.8
```

List Types

- First used in Lisp (1960)
  - Lists in Lisp and Scheme are delimited by parentheses and use no commas

    `(3 2.5 "a")` and `(A (B C) D)`
  - A set of operations (concatenate, element access, …) provided on lists
- Many newer languages incorporated lists
  - Python, ML, F#, Haskell, …
  - A rich set of operations on lists provided
- Variation
  - Both C# and Java supports lists through their generic heap-dynamic collection classes, List and ArrayList, respectively

Array Types: Summary

- The data types of a language are a large part of what determines that language's style and usefulness
- Arrays are included in most languages
- Traditional arrays
  - Homogeneous elements
  - one-dimensional as well as multi-dimensional arrays
  - Mostly row-major with very few language support column major
  - Arrays of arrays provide jagged arrays
- Newer array-like data types
  - Associative arrays (dictionary etc)
  - Lists and Tuples
  - (discriminant) union
    - Free union is unsafe, obsolete now.

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 access a location in the area where storage is dynamically created (usually called a *heap*)

Design Issues

- What are the scope of and lifetime of *a pointer variable*?
- What is the lifetime of *a heap-dynamic variable*?
  - Please identify pointer variable and heap-dynamic variable in the following C++ code:

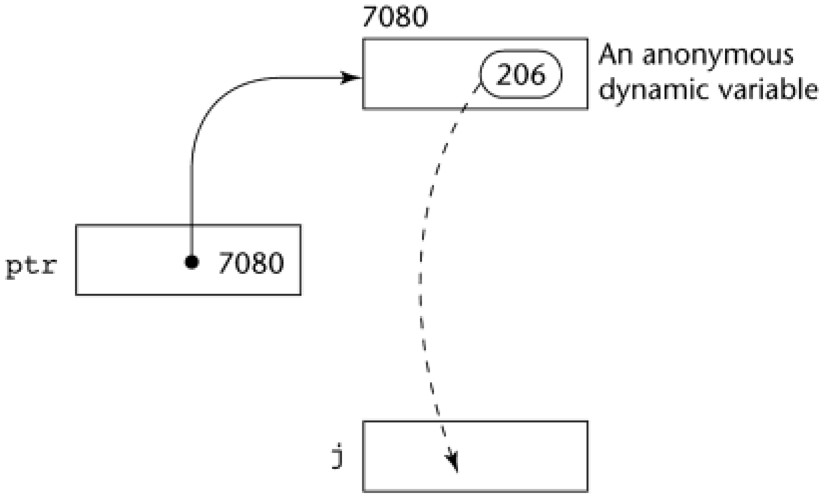
    ```
    double *p = new double (5.0)
    ```
- Are pointers restricted as to the type of value to which they can point?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer types, reference types, or both?
- How are pointers represented in machines?

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 *

```
int *ptr = new int (206);
int j;
...
j = *ptr;

j=ptr; //type error;
```



Problems with Pointers

- Dangling pointers (dangerous)
  - A pointer points to a heap-dynamic variable that has been deallocated

```
int *p1 = new int[5];
int *p2;
p2 = p1;
delete []p1; //p2 now?
```

- Lost heap-dynamic variable
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)

```
//Example: C++
int *p1 = new int(5);
int *p2 = new int (50);
p1 = p2;          //which location now becomes garbage?
```

  - The process of losing heap-dynamic variables is called *memory leakage*

Does the lifetime of pm ends at delete?

```
double *pm = new double[10];
…
delete []pm;
pm = new double [100]; //pm still alive

int *pf = new int [5];
for (int j = 0; j<5; j++) pf[j] = j;
cout << pf[6] << endl;  //what will be displayed?
                        //garbage value
```

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** * can point to any type)
- Example

```
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
  - In C++, both pointer type and reference type exist
- Java extends C++'s reference variables and allows them to replace pointers entirely
  - References are references to objects, rather than being addresses
  - Pointers do not exist in Java
- 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
- Heap Management is a very complex run-time process; Two approaches to reclaim garbage:
  - Reference counters (*eager approach*): reclamation is gradual
  - Mark-sweep (*lazy approach*): reclamation occurs when the list of variable space becomes empty
  - Details omitted

Pointer and Reference Types: Summary

- Pointers are used for addressing flexibility and to control dynamic storage management
- Pointers may be unsafe; many pointer-related problems:
  - Dangling pointer
  - Memory leakage
- Reference types are introduced trying to remedy the pointer problems
  - Conceptually "references to objects" are more abstract than pointers as not directly associated with memory addresses
- Optional type and optional values (null, nil)
  - Can help write clear code and enhance type checking in case of optional values

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
  - Language Examples
    - C and C++ are not: parameter type checking can be avoided; unions are not type checked
    - Java and C# are, almost (because of explicit type casting)
    - ML and F# are

Type Equivalence

- Name and structure equivalences
- *Name type equivalence* means the two variables have equivalent types if they are in either the same declaration or in declarations that use the same type name
  - Easy to implement but highly restrictive:
    - Subranges of integer types are not equivalent with integer types
    - Formal parameters must be the same type as their corresponding actual parameters
- *Structure type equivalence* means that two variables have equivalent types if their types have identical structures
  - More flexible, but harder to implement

Name vs. Structure Equivalence

//C++: variables s1 and s2 of Stack type and q1 and q2 of Queue type.

```
typedef struct {
        int data[100];
        int count;
} Stack;

typedef struct {
        int number[100];
        int size;
} Queue;

Stack s1, s2;
Queue q1, q2;
```

Name equivalence:

s1 and s2

q1 and q2

Structure equivalence:

s1, s2, q1, q2

Type Equivalence

- Many problems may arise with the use of structure equivalence
- Consider the problem of two structured types:
  - Are two array types equivalent if they are the same except that the subscripts are different? (e.g. `[1..10]` and `[0..9]`)
  - Are two enumeration types equivalent if their components are spelled differently?
  - With structural type equivalence, you cannot differentiate between types of the same structure
    - e.g. different units of speed, both float

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
- Formal model of a type system is a set of types and a collection of functions that define the type rules
- Details omitted here

Summary

- The data types of a language are a large part of what determines that language's style and usefulness
- Type checking is important to enhance the reliability of the language
- Many theoretical research in the field of type system
  - Our primary interest is in the practical branch, on data types of modern, popular programming languages

– End of Lecture