4. DATA TYPES

T: Ch. 6

R1: 10.7, 10.8

Topics

- Introduction
- Primitive Data Types
- Character and String Types
- Enumeration Types
- Array Types
- Record Types
- Union Types
- Pointer and Reference Types
- Type Checking
- C++ Classes
- Classes in Java

Introduction

- A *data type* defines a collection of data objects and a set of predefined operations on those objects
  - The stored values (instructions and data) in memory hardware are untyped: the hardware on most machines makes no attempt to keep track of which interpretations correspond to which locations in memory.
  - One design issue for all data types: What operations are defined and how are they specified?

- *Type checking* is the activity of ensuring that the operands of an operator are of compatible types
  - A *type error* is the application of an operator to an operand of an inappropriate type
  - A programming language is *strongly typed* if type errors are always detected

Introduction

- Type System of a programming language:
  - Defines how a type is associated with each expression in the language and includes its rules for type equivalence and type compatibility.
  - Uses of the type system of a programming language
    - Error detection
    - Assistance for program modularization: ensuring consistency of the interfaces among modules.
    - Documentation: provides information about a program's data and its behavior.

Introduction

- A *descriptor* is the collection of the attributes of a variable that are used for type checking and performing allocation and deallocation operations.
  - Descriptors of static attributes are needed only at compile-time, built as part of the symbol table.
  - Descriptors for dynamic attributes must be used and maintained by the run-time system.

- Data Types are either scalar (primitive types), or structured (composite).
  - Scalar types: ordinal types (integer, char, enumerated), real, complex, etc.
  - Structure types: records/classes, unions, arrays, pointers, lists, sets, and files.

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
  - numeric types (integer, floating point, complex, decimal)
    - Allowed operators: Arithmetic operations
  - Boolean, charácter types.

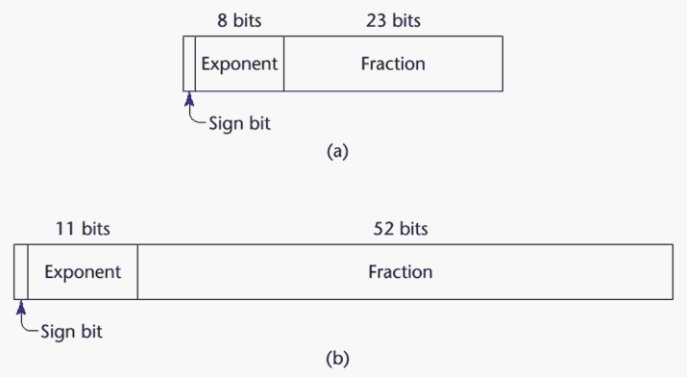
- Some primitive data types are merely reflections of the hardware.
- Others require only a little non-hardware support for their implementation.

Primitive Data Types: Integer

- Almost always an exact reflection of the hardware so the mapping is trivial.
- C and C++ integer types: char, short, int, long, long long
  - These are listed in nondecreasing order
  - An `int` is the size of an integer on the target processor
- Java's signed integer sizes: **`byte, short, int, long`**
  - These have fixed sizes: 1, 2, 4, and 8 bytes
- Unsigned and Signed
  - "Positive" and "negative" is usually represented by designating one of the bits in the integer as a sign bit
  - C++ allows an integer to be declared to be unsigned.
- Hardware representation of signed intergers in binary is based on using the two's complement notation.
  - A negative integer is the complement of the positive versión of the number plus 1.

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
  - Floating-point type is defined in terms of precision and range.
  - IEEE Floating-Point Standard 754
    - Value= sign * mantissa * 2^exp
    - Mantissa = 1.fract
  - C++ includes float, double and long double with increasing order of size



Primitive Data Types: Boolean

- Range of values: two elements, one for “true” and one for “false”
- Could be implemented as bits, but often as bytes
  - Advantage: readability of code
- The allowable operations are the logical (**!**, **&&**, ||) and relational operations.

Primitive Data Types: Character

- Stored as numeric codings
- Most commonly used coding: ASCII (128 characters) or extended ASCII (256 characters)
  - In C++ char type is a 1 byte ASCII coding
  - A **byte** in Java is an 8 bit quantity
- An alternative, 16-bit coding: Unicode (UCS-2 or UTF-16)
  - Includes characters from most natural languages
  - Originally used in Java. A **char** in Java is a 16 bit quantity.
  - In C++ the data type **wchar_t** is used for Unicode characters.
- 32-bit Unicode (UCS-4 or UTF-32)
  - Supported by Fortran, starting with 2003

Character String Types

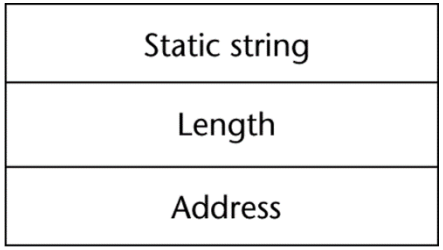
- Values are sequences of characters.
- Design issues:
  - Is it a primitive type or just a special kind of array?
  - Should the length of strings be static or dynamic?
- C and C++
  - Not a primitive type
  - Uses **char** arrays and a library of functions to provide operations.
    - String manipulation functions in the C standard library are inherently unsafe and source of numerous programming errors.
  - C++ has a string class which is a sequence of char. There are also strings of UTF-16 and UTF-32 available.
- Fortran and Python
  - Primitive type with assignment and several operations.

Character String Types

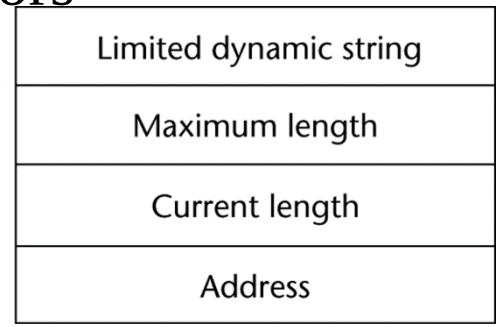
- Java
  - Provides a String class, which is a sequence of Java char, and which uses UTF-16 encoding.
- Typical operations:
  - Assignment and copying
  - Comparison (=, >, etc.)
  - Catenation
  - Substring reference
  - Pattern matching

Character String Length Options

- Static: Java's `String` class, C++ standard class, and strings in 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
- Compile- and Run-Time Descriptors



Compile-time descriptor for static strings



Run-time descriptor for limited dynamic strings

Enumeration Types

- All possible values, which are named constants, are provided in the definition
- Example

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

- Values in an enumeration type are also represented internally as integers. By default, the first enumerator has the integer value 0, the second has the value 1, and so forth.
- Design issues
  - An enumeration constant is not allowed to appear in more than one type definition
  - Enumeration values can be coerced to integer

```
int x = mon;
```

  - Any other type cannot be coerced to an enumeration type

```
days myday = 4;//not allowed in C++
myday = (days) 4; //in C++ casting to enumerated type is allowed
```

Enumeration Types

- Comparison: ordering is determined by the order in which the enumerators in the type declaration are listed.
- Values of enumeration types must be input or output indirectly
- Incrementing or decrementing can be applied using explicit type conversion (i.e., casting). For example,

```
myday = (days)(myday+ 1);
```

  - None of the following works

```
myday = myday + 1;
myday += 1;
myday++;
++myday;
```

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

Arrays

- 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.
- Array index types:
  - FORTRAN, C: integer only
  - Java: integer types only
  - Array index type C/C++ must be an ordinal type such as **char**, **short**, **int**, **long**, or **bool.**

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

  array_name (index_value_list) → an element

Arrays

- Index range checking
  - C, C++, Perl, and Fortran do not specify range checking
  - Java, C# specify range checking
- Array operations: Operations that operate on an array as a unit
  - Such as assignment, catenation, comparison for equality or inequality, and slices.
  - What are the operations allowed on arrays in C++, Java, and Python?
- Subscript Bindings
  - Binding of subscript type to an array variable is usually static.
  - In some languages the lower bound of the subscript range is implicit (as in C/C++ and Java).
- Categories of arrays
  - Four categories based on binding to subscript ranges, binding to storage, and the location of the storage.

Subscript Binding and Array Categories

- *Static*: subscript ranges are statically bound and storage allocation is static (before run-time).
  - Advantage: efficiency (no dynamic allocation)
- *Fixed stack-dynamic*: subscript ranges are statically bound, but the allocation is done at declaration time.
  - Advantage: space efficiency
- *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).
- 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).

Subscript Binding and Array Categories (continued)

- C and C++ arrays that include **`static`** modifier are static.
- C and C++ arrays declared in functions without **`static`** modifier are fixed stack-dynamic.
- C and C++ provide fixed heap-dynamic arrays
  - C library functions malloc and free for allocation and deallocation.
  - C++ uses operators new and delete.
- In Java all non-generic arrays are fixed-heap-dynamic.
  - Java includes a generic second array class `ArrayList` that provides fixed heap-dynamic.
    - Does not support indexing (using get and set methods).
- Perl, JavaScript, Python, and Ruby support heap-dynamic arrays
  - Objects of C# List class are generic heap-dynamic arrays: elements are added with Add method.

Arrays

- Some languages allow initialization at the time of storage allocation
  - C++ example:

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

  - Character strings in C and C++

```
char name [] = "freddie";//length is 8
```

  - Arrays of strings in C and C++

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

  - Java initialization of String objects

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

Arrays

- Operations that operate on an array as a unit
  - Such as assignment, catenation, comparison for equality or inequality, and slices.
  - C-based languages do not provide any array operations
    - Except through methods of classes as in Java, C++, and C#.
  - Python provides array assignments, but they are only reference changes. Python also supports array catenation and element membership operations
    - Array catenation using (+) and element membership (using in operator)

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 is one in which the lengths of the rows need not be the same. For example, 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
- F# and C# support rectangular arrays and jagged arrays

Slices

- A slice is some substructure of an array; nothing more than a referencing mechanism
- Slices are only useful in languages that have array operations
- Python

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

  - The syntax of a Python slice reference is a pair of numeric expressions. Examples:
    - `vector (3:6)` is a three-element array: the elements at indices 3, 4, and 5 of vector.
    - First expression is the first subscript of the slice, and the second is the first subscript after the last subscript of the slice.
    - `mat[0][0:2]` is the first and second element of the first row of `mat`

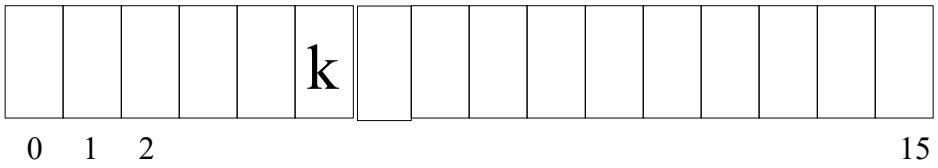
Implementation of Arrays

- 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)

(NOTE in C, C++, and Java, lower_bound is == 0)



Implementation of 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
  - Example: Assuming a matrix having the following values

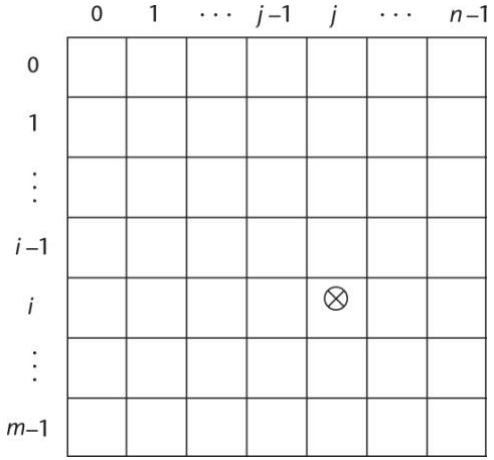
    3 4 7

    6 2 5

    1 3 8

    Its storage in row major order is

    3 4 7 6 2 5 1 3 8



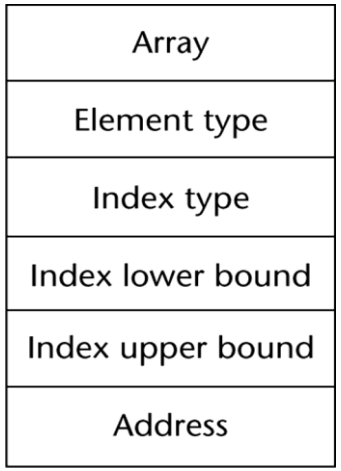
  - Address of an element mat[2] [1] is the base address of the array plus
    - The number of rows times the row size, plus the number of elements to the left of the element in its row.
  - Location(a[i, j]) = address of a [0, 0] +(((i * n) + j) * element_size)

Implementation of Arrays

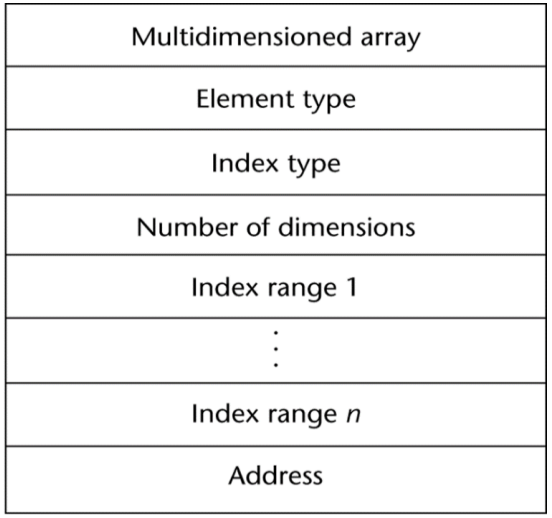
- General format

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

- Compile-Time Descriptors
  - A compile-time descriptor for a single –dimensional and multidimensional arrays
  - If no index range checking is done and all attributes are static, then only the access function is required during execution.



Single-dimensioned array



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.
- Built-in type in Perl, Python, Ruby, and Swift
  - In Python it is called dictionaries, where values are references to objects.
- Related Implementations in Java/C++ by standard class libraries
  - C++ map or Java HashMap
  - These are parameterized using < > and type names, so the type of the key and value are fixed.
    - Example:

```
map <string, float> myMap = {{"smith", 2.5},{"George", 3.25}};
```

    - The values in a map can be accessed directly by their corresponding key using the *bracket operator* ((operator[]).

Record Types

- A *record* is a possibly heterogeneous aggregate of data elements in which the individual elements, called fields, are identified by names.
  - Fields are stored in adjacent memory locations.
  - Access is through offsets from the beginning of the structure.
- struct: a record of a collection of elements of possibly different types, supported in C, C++, and C#.
  - In C++, structures are similar to classes. The fields are called members, where references to individual fields are done using dot notation.
  - Declaration of a data type and a variable (compact way)

```
struct employeeType{
    int id;
    char name[25];
    float salary;
} employee;
```

Record Types

```
struct employeeType{ //declaration of a data type
    int id;
    char name[25];
    float salary;
};
struct employeeType employee;//in C definition variable
employeeType employee;//in C++ definition of variable
...
employee.salary = 5545;//access of member using dot notation
employeeType * empPtr = & employee;//access using a pointer
(*empPtr).name = "Smith";//access using the dot notation
empPtr -> salary = 4500.50;//access using arrow form
```

Record Types

- Union: A union in C/C++ is like a struct in that it generally has several fields, all of which are public by default. Unlike a struct, however, only one of the fields is used at any given time. In other words, it is a structure that allows the same storage space to be used to store values of different data types at different times.
  - A union is a special class type that can hold only one of its non-static data members at a time.
  - Example:

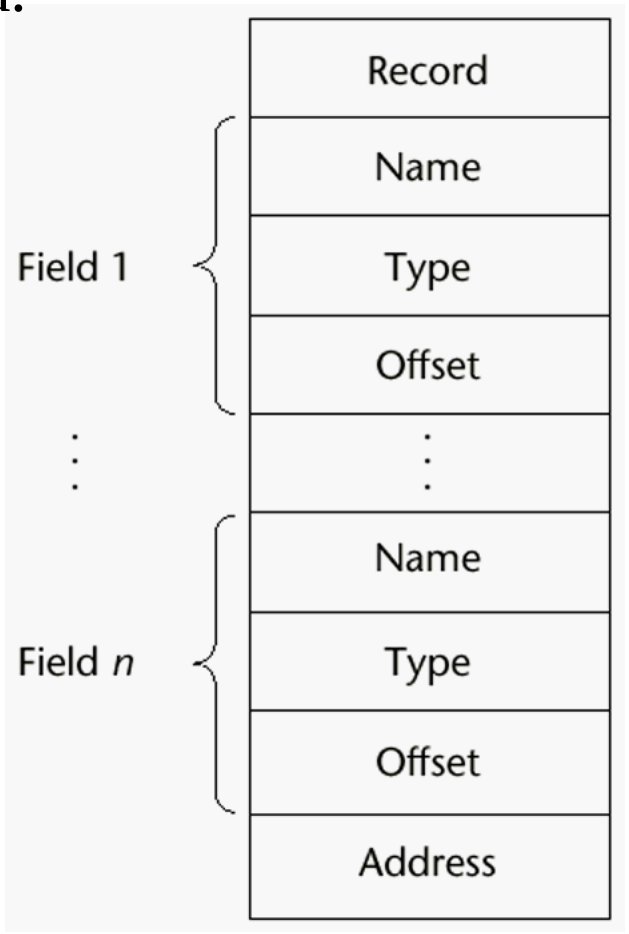
```
union UnionTyp {
        int i;
        double d;
        char c;
    };
    UnionType u;
u.i = 5;
. . . .
u.c ='M';
```

Comparison between Arrays and Records

- Records are used when collection of data values is heterogeneous
  - The fields of records are stored in adjacent memory locations.
  - Offset address relative to the beginning of the records is associated with each field.
  - Assignment operator is defined on struct variables of the same type.
  - Struct variable can be passed by value or reference to functions or to be the return type of a function.
  - I/O operations are not defined on struct variables.
- Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static).

Implementation of Record Type

- A compile-time descriptor for a record.



- The fields of records are stored in adjacent memory locations.
- Offset address relative to the beginning of the records is associated with each field

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 they are immutable.
    - If a tuple needs to be changed, it can be converted to an array using the list function. Then it can be converted back to tuple using the tuple function.
    - a tuple is created by assigning a tuple literal, for example:

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

    - Elements are referenced by indexing, where first index is at 1. For example `myTuple[2]` refers to the value 5.8.
    - Catenation using + operator, and deleted with **`del`** statement.

List Types (continued)

- Python Lists
  - The list data type also serves as Python's arrays
  - Unlike Scheme, Common Lisp, ML, and F#, Python's lists are mutable.
  - Elements can be of any type.
  - Create a list with an assignment

    ```
    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: Powerful mechanism for creating arrays. Syntax of Python List Comprehensions :
    [expression for iterate_var in array if condition]

    ```
    [x * x for x in range(6) if x % 3 == 0]
    ```
    range(6) creates [0, 1, 2, 3, 4, 5, 6]
    Constructed list: [0, 9, 36]

Pointer and Reference Types

- A *pointer* type variable has a range of values that consists of memory addresses and a special value, *NULL (value of 0)*
  - A pointer variable indicates the location of a variable **of a given** type.

- Provide the power of indirect addressing
  - **Indirect addressing** Accessing a variable in two steps by first using a pointer variable that gives the location of the variable.
  - **Direct addressing** Accessing a variable in one step by using the variable name.
- Provide a way to manage dynamic memory (e.g., allocation and deallocation).

Pointer Arithmetic in C and C++

- A pointer can be used to access a location in the area where storage is dynamically created (usually called a *heap*)
  - null pointer is special pointer value that does not refer to any valid memory location, it is provided by **nullptr** keyword.
- C++ operators that are valid for pointers include: =, *****, **->, and relational operators as (== and !=)**
- **Arithmetic operations as addition and subtraction. For 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]`

Problems with Pointers

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

```
int * arrPtr1;
int * arrPtr2 = new int[10];

arrPtr1 = arrPtr2;
delete [] arrPtr2;
```

    - Now arrPtr1 and arrPtr2 are dangling pointers

Problems with Pointers

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

```
int *ptr1 = new int;
*ptr1 = 5;
int * ptr2 = new int;
*ptr2 = 7;
ptr1 = ptr2;
cout << *ptr1 << " " <<    *ptr2 << endl;
//first heap-dynamic variable is garbage now.
```

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 (i.e., &).
- Domain type need not be fixed (`void *`)

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

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
  - Reference type variable refers to an object or a value in memory. While, a pointer variable refers to an address in memory.
- References in C++ must be initialized when declared:

```
int x;
int& xr = x;
```

  - Similar to a constant pointer, it cannot be reassigned and it cannot be assigned 0.
  - No dereferencing applied by the * operator
  - References of reference are not posible
  - Pointer of a reference is not posible: int & * xr = x;

Reference Types

- 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 statements--they widen the range of references to memory 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.
  - They are essential in some kinds of applications (e.g., device drivers).
- References of Java and C# provide some of the flexibility and capabilities of pointers without the hazards.

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 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*.
  - Example:
    - In an assignment statement, the type of the right-hand side must be compatible with that of the left-hand side.
    - The types of the operands of + must both be compatible with some common type that supports addition (e.g., numeric types).
- 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 at compile-time.
- If type bindings are dynamic, type checking must be done at run-time, which is called dynamic checking.
  - Javascript and PHP allow only dynamic type checking.

Strong Typing

- 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).
- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus ML and F#)
  - Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada.

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