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: *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*

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: 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
- IEEE Floating-Point Standard 754

![Floating-Point Representation](image)

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**):
  \[(7 \ + \ 3j), \text{ where } 7 \text{ is the real part and } 3 \text{ is the imaginary part}\]

Primitive Data Types: Decimal

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

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 codings
- 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 extensions

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?
- **Typical operations:**
  - Assignment and copying
  - Comparison (=, >, etc.)
  - Concatenation
  - 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
- SNOBOL 4 (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, C++ standard class library, and .NET library
- **Limited Dynamic Length:**
  - C and C++ (C-Style strings)
  - 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): SNOBOL 4, Perl, JavaScript
- Ada 95 supports all three string length options
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**:
  - Implemented using link lists (pointers)
  - Implemented using adjacent storage cells
  - Need run-time descriptor; allocation/de-allocation is the biggest implementation problem

Compile- and Run-Time Descriptors

<table>
<thead>
<tr>
<th>Static string</th>
<th>Limited dynamic string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>Maximum length</td>
</tr>
<tr>
<td>Address</td>
<td>Current length</td>
</tr>
</tbody>
</table>

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
- User-Defined Ordinal Types
  - **Enumeration Types**
  - **Subrange Types**
Enumeration Types

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

Subrange Types

- An ordered **contiguous** subsequence of an ordinal type
  - Example: 12..18 is a subrange of integer type
- **Ada’s** design
  ```plaintext
  type Days is (mon, tue, wed, thu, fri, sat, sun);
  subtype Weekdays is Days range mon..fri;
  subtype Index is Integer range 1..100;
  ```
  ```plaintext
  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
  - Usually without restrictions
- Subrange types are implemented like the parent types
  - with code inserted (by the compiler) to restrict assignments to subrange variables

Array Types

- An array is an aggregate of usually homogeneous 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 subcripting expressions in element references range checked?
- When are subscript ranges bound?
- When does allocation take place?
- 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) → an element

Index Syntax
  - **FORTRAN, PL/I, 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**: 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

Subscript Binding and Array Categories (cont’d)

- **Stack-dynamic**: subscript ranges are dynamically bound and the storage allocation is dynamic (done at run-time)
  - Advantage: flexibility (the size of an array need not be known until the array is to be used)
- **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)

Subscript Binding and Array Categories (cont’d)

- **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 (cont’d)

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