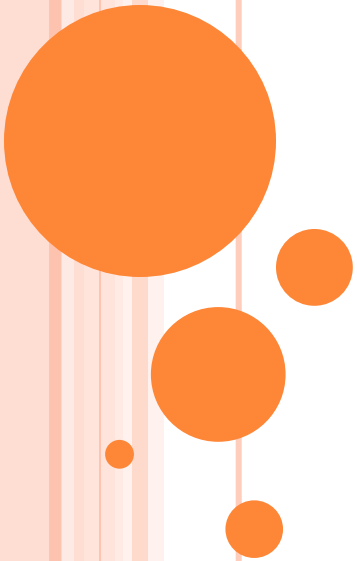


# Chapter 5

## Data Types

# Chapter 6 Topics

- Introduction
- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Associative Arrays
- Record Types
- Union Types
- Pointer and Reference 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

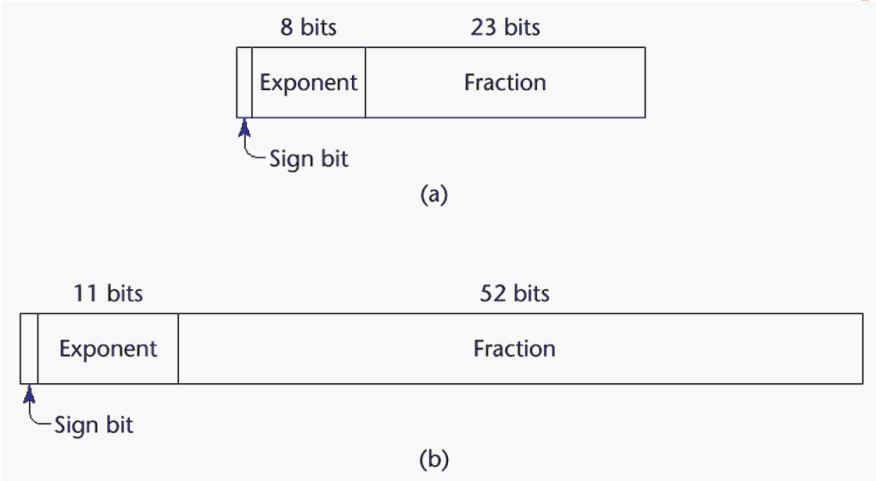
- Almost all programming languages provide a set of *primitive data types*
- Primitive data types: Those not defined in terms of other data types
- Some primitive data types are merely reflections of the hardware
- Others require little non-hardware support

# 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: 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



# Primitive Data Types: Decimal

- For business applications (money)
  - Essential to COBOL
  - C# offers a decimal data type
- Store a fixed number of decimal digits
- *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
  - Includes characters from most natural languages
  - Originally used in Java
  - C# and JavaScript also support Unicode

# 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
  - 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
- Java
  - Primitive via the `String` class

# Character String Length Options

- Static: COBOL, Java's `String` class
- *Limited Dynamic Length*: C and C++
  - In C-based language, 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

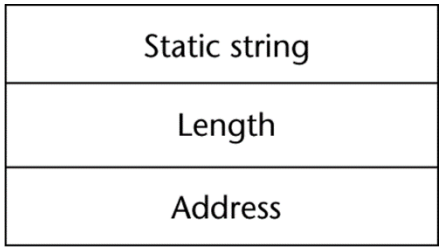
# Character String Type Evaluation

- 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?

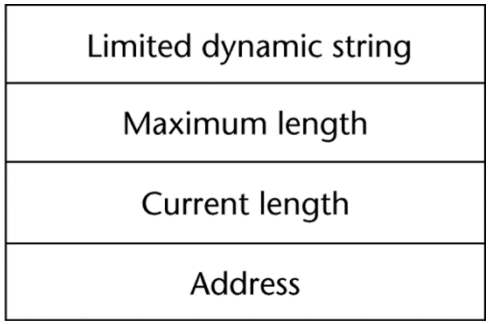
# Character String Implementation

- 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/de-allocation is the biggest implementation problem

# Compile- and Run-Time Descriptors




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`

# 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

# Subrange Types

- An ordered contiguous subsequence of an ordinal type
  - Example: 12..18 is a subrange of integer type
- Ada's design

```
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

# Array Types

- An array is an aggregate of homogeneous data elements in which an individual element is identified by its position in the aggregate, relative to the first element.

# 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?
- What is the maximum number of subscripts?
- Can array objects be initialized?
- Are any kind of slices allowed?

# Array Indexing

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

  `array_name (index_value_list)` → `an element`

- Array reference : array name followed by list of subscripts which surrounded by brackets or parentheses.
- 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
- Pascal: any ordinal type (integer, Boolean, char, enumeration)
- Ada: integer or enumeration (includes Boolean and char)
- Java: integer types only
- C, C++, Perl, and Fortran do not specify range checking
- Java, ML, C# specify range checking

# 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 (continued)

- *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 (continued)

- 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 without `static` modifier are fixed stack-dynamic
- Ada arrays can be stack-dynamic
- C and C++ provide fixed heap-dynamic arrays
- C# includes a second array class `ArrayList` that provides fixed heap-dynamic
- Perl and JavaScript 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"};
```

# Arrays Operations

- APL provides the most powerful array processing operations for vectors and matrices as well as unary operators (for example, to reverse column elements)
- Ada allows array assignment but also catenation
- Fortran provides *elemental* operations because they are between pairs of array elements
  - For example, + operator between two arrays results in an array of the sums of the element pairs of the two arrays

# 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

# 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

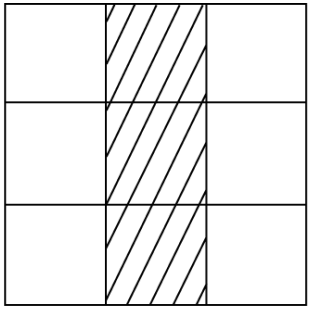
# Slice Examples

- Fortran 95

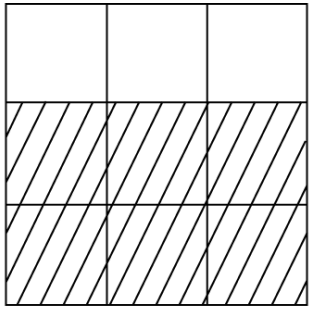
```
Integer, Dimension (10) :: Vector
Integer, Dimension (3, 3) :: Mat
Integer, Dimension (3, 3) :: Cube
```

`Vector (3:6)` is a four element array

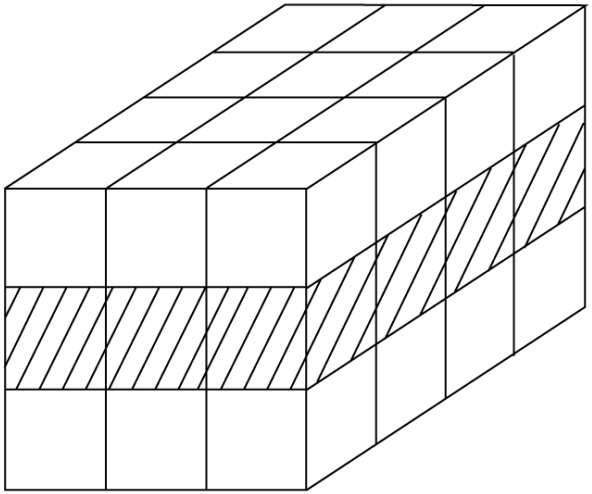
# Slices Examples in Fortran 95



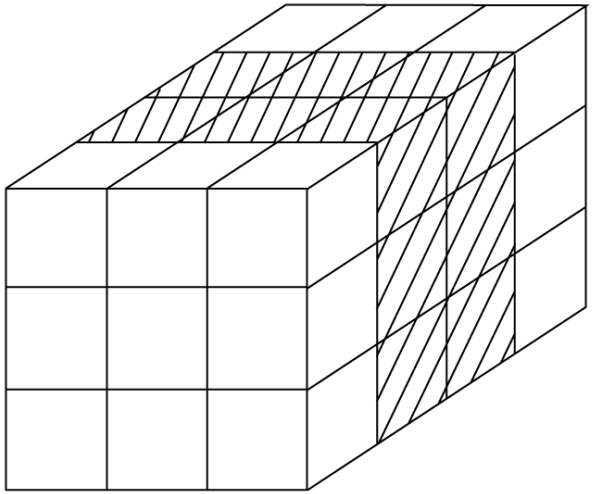
MAT (1:3, 2)



MAT (2:3, 1:3)



CUBE (2, 1:3, 1:4)



CUBE (1:3, 1:3, 2:3)

# 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 * element_size

# Pointer and Reference Types

- A *pointer* type variable has a range of values that consists of memory addresses and a special value, *nil*
- Advantages of pointers:
  - 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?

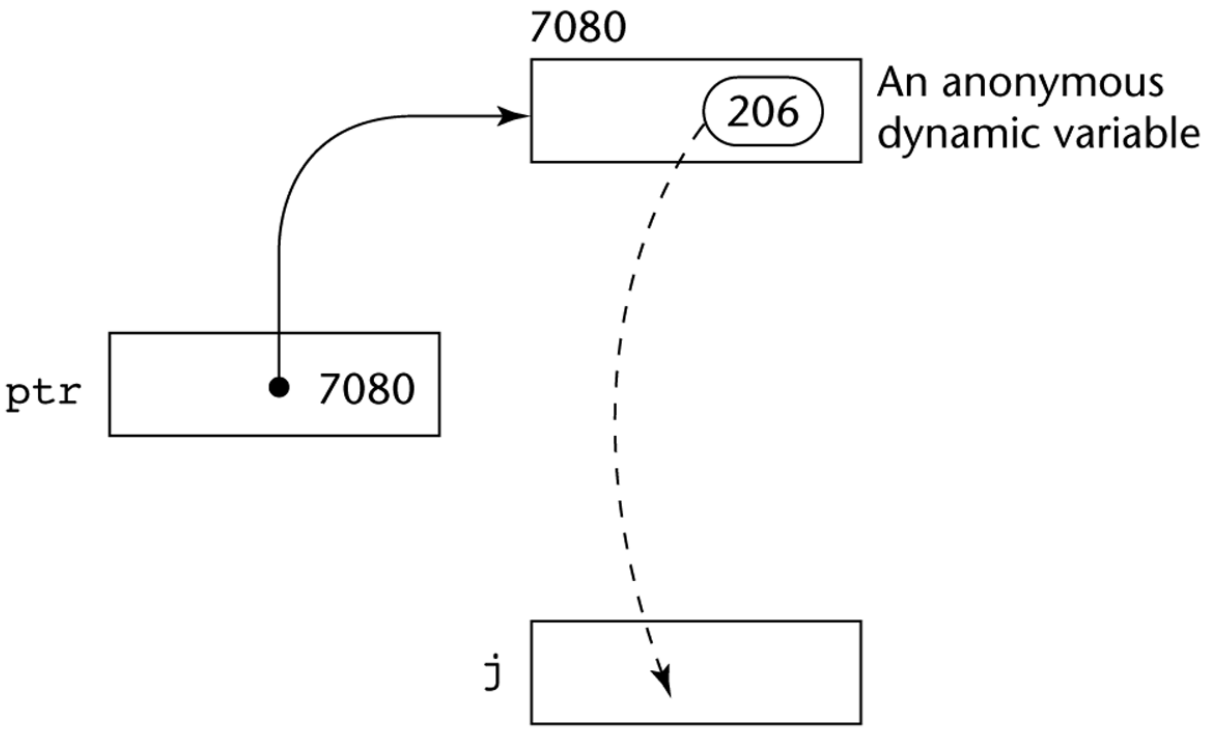
# Pointer Operations

- 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 de-allocated
  - Dangerous for several reasons because location being pointed to may have been reallocated to some new heap-dynamic variable.
  - Example:

```
int *arrayPtr1;
int *arrayPtr2 = new int[100];
arrayPtr1 = arrayPtr2;
delete [] arrayPtr2;
```

# Problems with Pointers

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

# Pointers in Ada

- Some dangling pointers are disallowed because dynamic objects can be automatically de-allocated at the end of pointer's type scope
- The lost heap-dynamic variable problem is not eliminated by Ada

# POINTERS IN C AND C++

- Extremely flexible but must be used with care
- Pointers can point at any variable regardless of when 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; // assign address of stuff[0] to p
```

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

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

# Reference Types

- Similar to pointer except that it is referring to an object or a value in memory rather than an address.
- 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 refer to call instances
- 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
  - Subprogram as operator, parameters as operands
  - Assignment symbol as binary operator, target variable and expression as operands
- *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***. For example, adding an **`int`** to a **`float`** in Java causes **`int`** to be coerced to **`float`**

# Type Checking (continued)

- A *type error* is the application of an operator to an operand of an inappropriate type. For examples, in passing values to functions.
- 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

# Strong Typing

- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors
- Language examples:
  - FORTRAN 77 is not: parameters, `EQUIVALENCE`
  - Pascal is not: variant records
  - C and C++ are not: unions are not type checked
  - ML is strongly typed
  - Ada is, almost: type checking can be suspended with the calling of function `Unchecked_Conversion`. Java and C# are similar.

# Strong Typing (continued)

- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus Ada)
- Languages with a great deal of coercion, like Fortran, C, and C++ are les reliable than those with little coercion, such as Ada.
- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada

# Type Equivalence

- Type compatibility influences the design of the data types and the operation provided for values of those types.
- Two different types of compatibility methods :
  - Name type equivalence
  - Structure type equivalence

# Name Type Equivalence

- *Name type equivalence* means the two variables have compatible types if they are in either the same declaration or in declarations that use the same type name
- Easy to implement but more restrictive:
  - Subranges of integer types are not equivalence with integer types. In the following example, types of variable count and index are not equivalence.

```
type Indextype is 1..100;
count : Integer;
index : Indextype;
```

  - Formal parameters must be the same type as their corresponding actual parameters, especially user-defined types.

# Structure Type Equivalence

- *Structure type equivalence* means that two variables have compatible types if their types have identical structures
- More flexible, but harder to implement because the entire structures of the two types must be checked for equivalence.

# Structure Type Equivalence (continued)

- Consider the problem of two structured types:
  - Are two record types equivalent if they are structurally the same but use different field names?
  - 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)

# 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