Chapter 6

Data Type

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

Chapter 6

Data Type

Introduction

- A data type defines a collection of **data objects** and a set of **predefined operations** on those objects.
- Computer programs produce results by manipulating data.
- ALGOL 68 provided a few basic types and a few flexible structure-defining operators that allow a programmer to design a data structure for each need.
- A **descriptor** is the collection of the attributes of a variable.
- In an implementation a descriptor is a collection of memory cells that store variable attributes.
- If the attributes are static, descriptor are required only at compile time.
- They are built by the compiler, usually as a part of the symbol table, and are used during compilation.
- For dynamic attributes, part or all of the descriptor must be maintained during execution.
- Descriptors are used for type checking and by allocation and deallocation operations.

Primitive Data Types

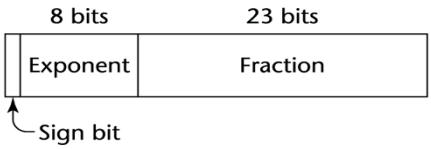
- Those not defined in terms of other data types are called primitive data types.
- The primitive data types of a language, along with one or more type constructors provide structured types.

Numeric Types

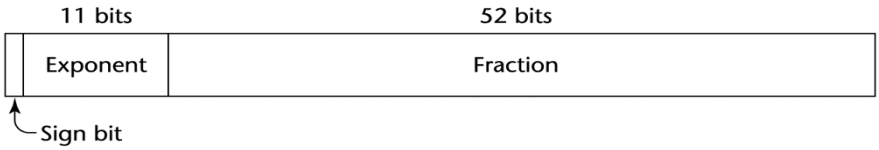
1. **Integer**
   - Almost always an exact reflection of the hardware, so the mapping is trivial.
   - There may be as many as eight different integer types in a language.
   - Java has four: **byte**, **short**, **int**, and **long**.
   - Integer types are supported by the hardware.

2. **Floating-point**
   - Model real numbers, but only as **approximations** for most real values.
   - On most computers, floating-point numbers are stored in binary, which exacerbates the problem.
   - Another problem is the loss of accuracy through arithmetic operations.
   - Languages for scientific use support at least two floating-point types; sometimes more (e.g. **float**, and **double**.)

- The collection of values that can be represented by a floating-point type is defined in terms of precision and range.
- **Precision**: is the accuracy of the fractional part of a value, measured as the number of bits. Figure below shows single and double precision.
- **Range**: is the range of fractions and exponents.



(a)



(b)

3. **Decimal**

- Most larger computers that are designed to support business applications have hardware support for **decimal** data types.
- Decimal types store a fixed number of decimal digits, with the decimal point at a fixed position in the value.
- These are the primary data types for business data processing and are therefore essential to **COBOL**.
- **Advantage**: accuracy of decimal values.
- **Disadvantages**: limited range since no exponents are allowed, and its representation wastes memory.

Boolean Types

- Introduced by ALGOL 60.
- They are used to represent switched and flags in programs.
- The use of Booleans enhances readability.
- One popular exception is C89, in which **numeric** expressions are used as conditionals. In such expressions, all operands with **nonzero** values are considered **true**, and **zero** is considered **false**.

Character Types

- Char types are stored as numeric codings (ASCII / Unicode).
- Traditionally, the most commonly used coding was the **8-bit** code ASCII (American Standard Code for Information Interchange).
- A **16-bit** character set named Unicode has been developed as an alternative.
- **Java** was the first widely used language to use the Unicode character set. Since then, it has found its way into **JavaScript and C#.**

Character String Types

- A character string type is one in which values are sequences of characters.
- **Important Design Issues**:
  1. Is it a primitive type or just a special kind of array?
  2. Is the length of objects static or dynamic?
- C and C++ use **char arrays** to store char strings and provide a collection of string operations through a standard library whose header is `string.h`.
- How is the length of the char string decided?
- The null char which is represented with 0.
- Ex:

```
char *str = “apples”; // char ptr points at the str apples0
```

- In this example, str is a char pointer set to point at the string of characters, apples0, where 0 is the null char.

String Typical Operations:

- Assignment
- Comparison (=, >, etc.)
- Catenation
- Substring reference
- Pattern matching
- Some of the most commonly used library functions for character strings in C and C++ are
  - strcpy: copy strings
  - strcat: catenates on given string onto another
  - strcmp: lexicographically compares (the order of their codes) two strings
  - strlen: returns the number of characters, not counting the null
- In Java, strings are supported as a **primitive** type by **String** class

String Length Options

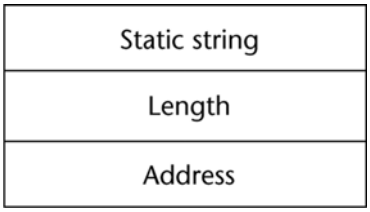
- **Static Length String**: The length can be static and set when the string is created. This is the choice for the **immutable** objects of **Java’s String class** as well as similar classes in the C++ standard class library and the .NET class library available to C#.
- **Limited Dynamic Length Strings**: allow strings to have varying length up to a **declared and fixed maximum** set by the variable’s definition, as exemplified by the strings in **C.**
- **Dynamic Length Strings**: Allows strings various length with no maximum. Requires the overhead of dynamic storage allocation and deallocation but provides flexibility. Ex: **Perl and JavaScript**.

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?

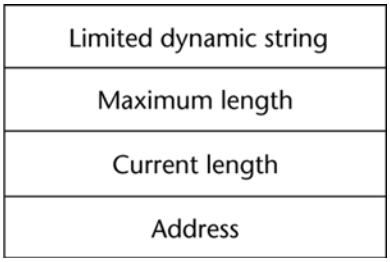
Implementation of Character String Types

- **Static length** - compile-time descriptor has three fields:
    1. Name of the type
    2. Type's length
    3. Address of first char



Compiler-time descriptor for static strings

- **Limited dynamic length Strings** - may need a run-time descriptor for length to store both the fixed maximum length and the current length (but not in C and C++ because the end of a string is marked with the **null** character).



Run-time descriptor for limited dynamic strings

- **Dynamic length Strings**–
    - Need run-time descriptor because **only current** length needs to be stored.
    - Allocation/deallocation is the biggest implementation problem. Storage to which it is bound must grow and shrink dynamically.
    - There are **two** approaches to supporting allocation and deallocation:
    1. Strings can be stored in a **linked list** "Complexity of string operations, pointer chasing"
    2. Store strings in adjacent cells. "What about when a string grows?" Find **a new area** of memory and the old part is moved to this area. Allocation and deallocation is **slower** but using adjacent cells results in faster string operations and requires less storage.

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
- In some languages, users can define 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 colors {red, blue, green, yellow, black};
colors myColor = blue, yourColor = red;
myColor++;                                  // would assign green to myColor
```

  - The enumeration constants are typically implicitly assigned the integer values, 0, 1, …, but can be explicitly assigned any integer literal.
- Java does not include an enumeration type, presumably because they can be represented as data classes. For example,

```
class colors {
    public final int red = 0;
    public final int blue = 1;
}
```

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;
```

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.
- A reference to an array element in a program often includes one or more non-constant subscripts.
- Such references require a run-time calculation to determine the memory location being referenced.

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?

Arrays and Indexes

- Indexing is a mapping from indices to elements.
- The mapping can be shown as:

  map(array_name, index_value_list) $\rightarrow$ an element

- Ex: Ada

  ```
  Sum := Sum + B(I);
  ```

  Because **( )** are used for both subprogram parameters and array subscripts in **Ada**, this results in **reduced readability**.

- C-based languages use [ ] to delimit array indices.
- Two distinct types are involved in an array type:
  - The element type, and
  - The type of the subscripts.
- The type of the subscript is often a sub-range of integers.
- Ada allows other types as subscripts, such as Boolean, char, and enumeration.
- Among contemporary languages, C, C++, Perl, and Fortran **don't** specify range checking of subscripts, but **Java, and C# do**.

Subscript Bindings and Array Categories

- The binding of subscript type to an array variable is usually **static**, but the subscript value ranges are sometimes **dynamically bound**.
- In C-based languages, the lower bound of all index ranges is fixed at **0**; **Fortran** 95, it defaults to **1**.

1. A **static array** is one in which the subscript ranges are statically bound and storage allocation is static (done before run time).
   - Advantages: efficiency "No allocation & deallocation."
   - Ex:
     Arrays declared in C & C++ function that includes the **`static`** modifier are **static**.

2. A **fixed stack-dynamic array** is one in which the subscript ranges are statically bound, but the allocation is done at elaboration time during execution.
   - Advantages: Space efficiency. A large array in one subprogram can use the same space as a large array in different subprograms.
   - Ex:
     Arrays declared in C & C++ function without the **`static`** modifier are **fixed stack-dynamic arrays**.

3. A **stack-dynamic array** is one in which the subscript ranges are dynamically bound, and the storage allocation is dynamic "during execution." Once bound they remain fixed during the lifetime of the variable.
   - Advantages: Flexibility. The size of the array is not known until the array is about to be used.
   - Ex:
     Ada arrays can be **stack dynamic**:

```
Get (List_Len);
declare
    List : array (1..List_Len) of Integer;
    begin
  . . .
    end;
```

     The user inputs the number of desired elements for array **`List`**. The elements are then dynamically allocated when execution reaches the **`declare`** block. When execution reaches the end of the block, the array is deallocated.

4. A **fixed heap-dynamic array** is one in which the subscript ranges are dynamically bound, and the storage allocation is dynamic, but they are both fixed after storage is allocated.
   - The bindings are done when the user program requests them, rather than at elaboration time and the storage is allocated on the heap, rather than the stack.
   - Ex:

     C & C++ also provide **fixed heap-dynamic arrays**. The function *`malloc`* and `free` are used in C. The operations `new` and `delete` are used in C++.

     In Java all arrays are **fixed heap dynamic arrays**. Once created, they keep the same subscript ranges and storage.

5. A **heap-dynamic array** is one in which the subscript ranges are dynamically bound, and the storage allocation is dynamic, and can change any number of times during the array's lifetime.
   - Advantages: Flexibility. Arrays can grow and shrink during program execution as the need for space changes.
   - Ex:

     C# provides **heap-dynamic arrays** using an array class *`ArrayList.`*

     ```
     ArrayList intList = new ArrayList( );
     ```

     Elements are added to this object with the *Add* method, as in
     `intArray.Add(nextOne);`

     Perl and JavaScript also support heap-dynamic arrays.
     For example, in Perl we could create an array of five numbers with

     ```
     @list = {1, 2, 4, 7, 10);
     ```

     Later, the array could be lengthened with the push function, as in

     ```
     push(@list, 13, 17);
     ```

     Now the aary's value is (1, 2, 4, 7, 10, 13, 17).

Array Initialization

- Usually just a list of values that are put in the array in the order in which the array elements are stored in memory.
- Fortran uses the **DATA** statement, or put the values in **/ ... /** on the declaration.

```
Integer List (3)
Data List /0, 5, 5/  // List is initialized to the values
```

- C and C++ - put the values in braces; let the compiler count them.

```
int stuff [] = {2, 4, 6, 8};
```

- The compiler sets the length of the array.
- What if the programmer mistakenly left a value out of the list?
- Character Strings in C & C++ are implemented as arrays of **`char`**.

```
char name [ ] = "Freddie"; //how many elements in array name?
```

- The array will have 8 elements because the null character is implicitly included by the compiler.
- In Java, the syntax to define and initialize an array of references to String objects.

```
String [ ] names = ["Bob", "Jake", "Debbie"];
```

- Ada positions for the values can be specified:

```
List : array (1..5) of Integer := (1, 3, 5, 7, 9);
Bunch : array (1..5) of Integer:= (1 => 3, 3 => 4, others => 0);
     Note: the array value is (3, 0, 4, 0, 0)
```

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-lower_bound) * element_size)

- 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, if the matrix had the values

  3 4 7
  6 2 5
  1 3 8
  - it would be stored in row major order as:

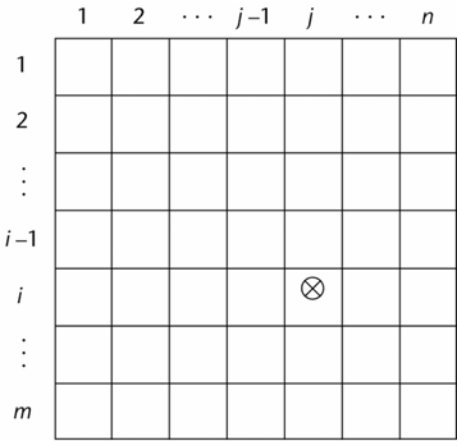
    3, 4, 7, 6, 2, 5, 1, 3, 8

  - If the example matrix above were stored in column major, it would have the following order in memory.

    3, 6, 1, 4, 2, 3, 7, 5, 8

  - In all cases, sequential access to matrix elements will be faster if they are accessed in the order in which they are stored, because that will minimize the **paging**. (Paging is the movement of blocks of information between disk and main memory. The objective of paging is to keep the frequently needed parts of the program in memory and the rest on disk.)
- Locating an Element in a Multi-dimensioned Array (row major)

  Location (a[i,j]) = address of a [1,1] + ((i - 1) * n + (j - 1)) * 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**.
- So each element of an associative array is in fact a pair of entities, a key and a value.
- Associative arrays are supported by the standard class libraries of Java and C++ and Perl.
- Example: In Perl, associative arrays are often called **hashes**. Names begin with %; literals are delimited by parentheses

```
%temps = ("Mon" => 77, "Tue" => 79, “Wed” => 65);
```

  - Subscripting is done using braces and keys

```
$temps{"Wed"} = 83;
```

  - Elements can be removed with delete

```
delete $temps{"Tue"};
```

  - Elements can be emptied by assigning the empty literal

```
@temps = ( );
```

Record Types

- A record is a possibly **heterogeneous** aggregate of data elements in which the individual elements are identified by names.
- In C, C++, and C#, records are supported with the **struct** data type. In C++, structures are a minor variation on classes.
- COBOL uses level numbers to show nested records; others use recursive definition
- Definition of Records in COBOL
    - COBOL uses level numbers to show nested records; others use recursive definition

```
01 EMP-REC.
  02 EMP-NAME.
    05 FIRST PIC X(20).
    05 MID   PIC X(10).
    05 LAST  PIC X(20).
  02 HOURLY-RATE PIC 99V99.
```

- Definition of Records in Ada
    - Record structures are indicated in an orthogonal way

```
type Emp_Rec_Type is record
     First: String (1..20);
     Mid: String (1..10);
     Last: String (1..20);
     Hourly_Rate: Float;
 end record;
 Emp_Rec: Emp_Rec_Type;
```

References to Records

- Most language use dot notation

```
Emp_Rec.Name
```

- Fully qualified references must include all record names
- Elliptical references allow leaving out record names as long as the reference is unambiguous, for example in COBOL

FIRST, FIRST OF EMP-NAME, and FIRST of EMP-REC are elliptical references to the employee's first name

Operations on Records

- Assignment is very common if the types are identical
- Ada allows record comparison
- Ada records can be initialized with aggregate literals
- COBOL provides MOVE CORRESPONDING
    - Copies a field of the source record to the corresponding field in the target record

Unions Types

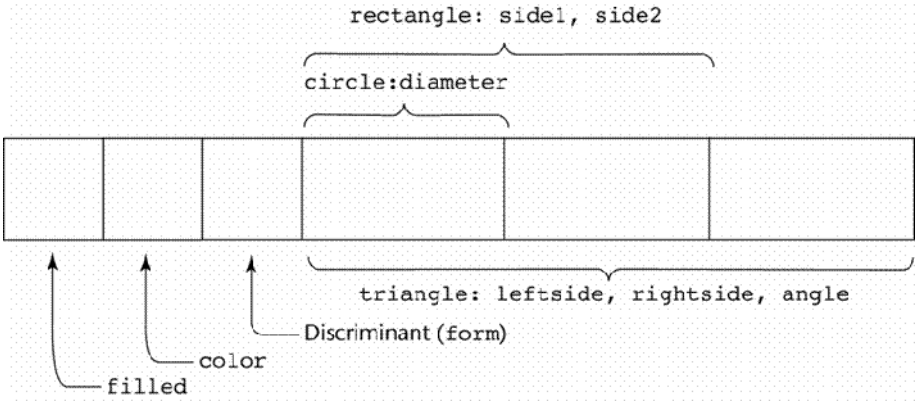
- A union is a type whose variables are allowed to store different type values at different times during execution.

Discriminated vs. Free Unions

- **Fortran, C, and C++** provide union constructs in which there is no language support for type checking; the union in these languages is called *free union*
- Type checking of unions require that each union include a type indicator called a *discriminanted union.*
- Supported by **Ada**

Ada Union Types

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
    Filled: Boolean;
    Color: Colors;
    case Form is
        when Circle => Diameter: Float;
        when Triangle =>
            Leftside, Rightside: Integer;
            Angle: Float;
        when Rectangle => Side1, Side2: Integer;
    end case;
end record;
```



Ada Union Type Illustrated: A discriminated union of three shape variables

Evaluation of Unions

- Potentially **unsafe** construct
- Java and C# **do not** support unions
- Reflective of growing concerns for safety in programming language

Pointers

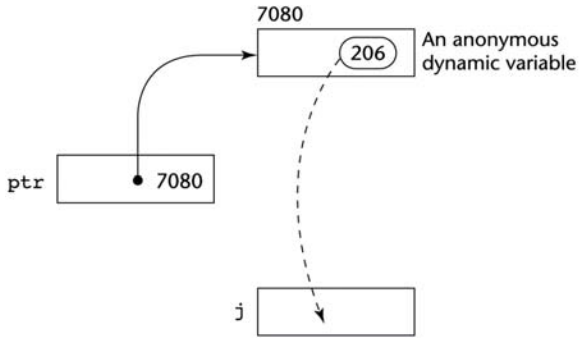
- A pointer type in which the vars have a range of values that consists of **memory addresses** and a special value, nil.
- The value nil is not a valid address and is used to indicate that a pointer cannot currently be used to reference any memory cell.

Pointer Operations

- A pointer type usually includes two fundamental pointer operations, assignment and dereferencing.
- Assignment sets a pointer var's value to some useful address.
- Dereferencing takes a reference through one level of indirection.
- In C++, **dereferencing** is explicitly specified with the (*) as a prefix unary operation.
- If *ptr* is a pointer var with the value 7080, and the cell whose address is 7080 has the value 206, then the assignment

  ```
  j = *ptr
  ```

  sets j to 206.



The assignment operation j = *ptr

Pointer Problems

1. **Dangling pointers** (**dangerous**)
   - A pointer points to a heap-dynamic variable that has been **deallocated**.
   - Dangling pointers are dangerous for the following reasons:
     - The location being pointed to may have been allocated to some new heap-dynamic var. If the new var is not the same type as the old one, type checks of uses of the dangling pointer are invalid.
     - Even if the new one is the same type, its new value will bear no relationship to the old pointer's dereferenced value.
     - If the dangling pointer is used to change the heap-dynamic variable, the value of the heap-dynamic variable will be destroyed.
     - It is possible that the location now is being temporarily used by the storage management system, possibly as a pointer in a chain of available blocks of storage, thereby allowing a change to the location to cause the storage manager to fail.
   - The following sequence of operations creates a dangling pointer in many languages:

     a. Pointer `p1` is set to point at a new heap-dynamic variable.
     b. Set a second pointer p2 to the value of the first pointer p1.
     c. The heap-dynamic variable pointed to by p1 is explicitly deallocated (setting p1 to nil), but p2 is not changed by the operation. P2 is now a dangling pointer.

2. **Lost Heap-Dynamic Variables** (wasteful)

   - A heap-dynamic variable that is no longer referenced by any program pointer "no longer accessible by the user program."
   - Such variables are often called **garbage** because they are not useful for their original purpose, and also they can't be reallocated for some new use by the program.
   - Creating Lost Heap-Dynamic Variables:

     a. Pointer p1 is set to point to a newly created heap-dynamic variable.
     b. p1 is later set to point to another newly created heap-dynamic variable.
     c. The first heap-dynamic variable is now inaccessible, or lost.

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

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 Fortran 95

- Pointers point to heap and non-heap variables
- Implicit dereferencing
- Pointers can only point to variables that have the TARGET attribute
- The TARGET attribute is assigned in the declaration:

```
INTEGER, TARGET :: NODE
```

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 in C and C++ makes their pointers **more interesting** than those of the other programming languages.
- Unlike the pointers of Ada, which can only point into the heap, C and C++ pointers can point at virtually any variable **anywhere** in memory.
- Explicit dereferencing and address-of operators
- In C and C++, the asterisk (*) denotes the **dereferencing** operation, and the ampersand (&) denotes the operator for producing the **address of** a variable. For example, in the code

```
int *ptr;
int count, init;
…
ptr = &init;
count = *ptr
```

- the two assignment statement are equivalent to the single assignement

```
count = init;
```

- Example: Pointer Arithmetic in C and C++

```
int list[10];
int *ptr;
ptr = list;

*(ptr+5)   is equivalent to list[5]   and  ptr[5]
*(ptr+i)   is equivalent to list[i]   and  ptr[i]
```

- 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 in function definition.
- A C++ reference type variable is a **constant** pointer that is always **implicitly** dereferenced.
- Because a C++ reference type variable is a constant, it **must** be **initialized** with the address of some variable in its definition, and after initialization a reference type variable can **never** be set to reference any other variable.
- Reference type variables are specified in definitions by preceding their names with ampersands (&). for example,

```
int result = 0;
int &ref_result = result;
…
ref_result = 100;
```

I this code segment, result and ref_result are **aliases**.

- In **Java**, **reference variables** are extended from their C++ form to one that allow them to replace pointers entirely.
- The fundamental difference between C++ pointers and Java references is that C++ pointers refer to **memory addresses**, whereas Java references refer to **class instances**.
- Because Java class instances are implicitly deallocated (there is no explicit deallocation operator), there **cannot** be a dangling reference.
- C# includes both the references of Java and the pointers of C++. However, the use of pointers is strongly discouraged. In fact, any method that uses pointers must include the **unsafe** modifier.
- Pointers can point at any variable regardless of when it was allocated