Chapter 6

Data Types

Chapter 6

Data Types

6.1 Introduction 236

- A data type defines a collection of data **values** and a set of predefined **operations** on those values.
- 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.
- These descriptors 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.

6.2 Primitive Data Types 238

- Those not defined in terms of other data types are called **primitive data types**.
- Almost all programming languages provide a set of primitive data types.
- Some primitive data types are merely reflections of the hardware – for example, most integer types.
- The primitive data types of a language are used, along with one or more type constructors.

6.2.1 Numeric Types

- Integer
  - The most common primitive numeric data type is integer.
  - The hardware of many computers supports several sizes of integers.
  - These sizes of integers, and often a few others, are supported by some programming languages.
  - Java includes four signed integer sizes: **byte**, **short**, **int**, and **long**.
  - C++ and C#, include **unsigned** integer types, which are types for integer values without sings.

- 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.

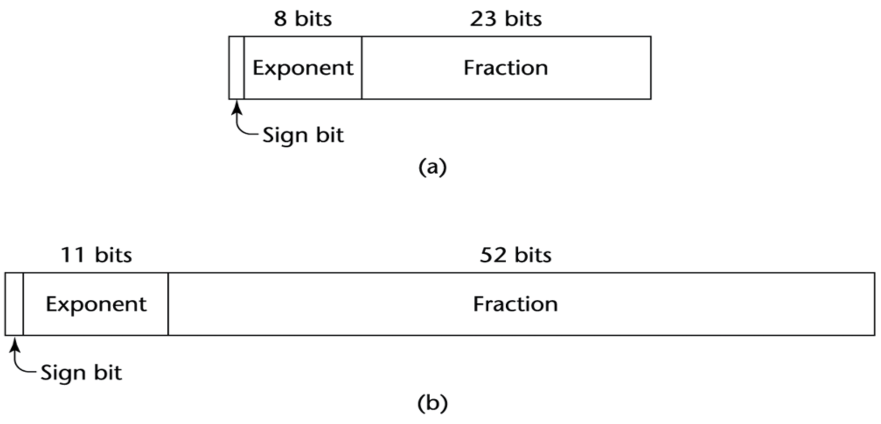


Figure 6.1 IEEE floating-point formats: (a) single precision, (b) double precision

- 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
  - Literal form (in Python):

```
(7 + 3j)
where 7 is the real part and 3 is the imaginary part
```

- 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

6.2.2 Boolean Types

- Boolean Types
  - Introduced by ALGOL 60
  - They are used to represent switched and flags in programs
  - The use of Booleans enhances readability
  - Range of values: two elements, one for “true” and one for “false”
  - 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**
  - A Boolean value could be represented by a single **bit**, but often statured in the smallest efficiently addressable cell of memory, typically a **byte**

6.2.3 Character Types

- 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)
  - An alternative, **16-bit** coding: Unicode (UCS-2)
  - **Java** was the first widely used language to use the Unicode character set. Since then, it has found its way into **JavaScript, Python, Perl, C#** and **F#**
  - After 1991, the Unicode Consortium, in cooperation with the International Standards Organization (ISO), developed a **4-byte** character code named UCS-4 or UTF-32, which is described in the ISO/IEC 10646 Standard, published in 2000

6.3 Character String Types 242

- A character string type is one in which values are sequences of characters

6.3.1 Design Issues

- The two most important design issues:
  - Is it a primitive type or just a special kind of array?
  - Is the length of objects static or dynamic?

6.3.2 String and Their Operations

- Typical operations:
  - Assignment
  - Comparison (=, >, etc.)
  - Catenation
  - Substring reference
  - Pattern matching
- 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
- Character string are terminated with a special character, **null**, with is represented with zero
- How is the length of the char string decided?
  - The null char which is represented with 0
  - Ex:

    ```
    char str[] = "apples";
    ```

    - In this example, str is an array of char elements, specifically **apples0**, where 0 is the null character
- 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 by **String** class, whose value are constant string, and the **StringBuffer** class whose value are changeable and are more like arrays of single characters
- C# and Ruby include string classes that are similar to those of Java
- Python strings are **immutable**, similar to the String class objects of Java

6.3.3 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#** and **F#**
- **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. This option requires the overhead of dynamic storage allocation and deallocation but provides flexibility. Ex: **Perl and JavaScript**

6.3.4 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?

6.3.5 Implementation of Character String Types

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

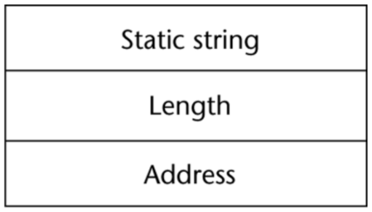


Figure 6.2 Compile-time descriptor for static strings

- **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)

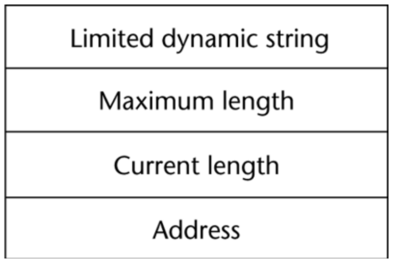


Figure 6.3 Run-time descriptor for limited dynamic strings

- **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 **three** approaches to supporting allocation and deallocation:
    1. Strings can be stored in a **linked list**, so that when a string grows, the newly required cells can come from anywhere in the heap
       - The drawbacks to this method are the extra storage occupied by the links in the list representation and necessary complexity of string operations0
       - String operations are **slowed** by the required pointer chasing
    2. Store strings as **arrays** of pointer to individual character allocated in the heap
       - This method still uses extra memory, but string processing can be **faster** that with the linked-list approach
    3. Store strings in **adjacent** storage 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. This approach is the one **typically** used

6.4 Enumeration Types 247

- All possible values, which are named constants, are provided, or enumerated, in the definition
- Enumeration types provide a way of defining and grouping collections of named constants, which are called **enumeration constants**
- C# example

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

  – The enumeration constants are typically implicitly assigned the **integer** values, 0, 1, …, but can explicitly assigned any integer literal in the type's definition

6.4.1 Design issues

- The designed issues for enumeration types are as follows:
  – 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?

6.4.2 Designs

- In languages that do not have enumeration types, programmers usually simulate them with integer values
- For example, **C** did not have an enumeration type. We might use 0 to represent `blue`, 1 to represent `red`, and so forth. These values could be defined as follows:

```
int red = 0, blue = 1;
```

- In C++, we could have the following:

```
enum colors {red, blue, green, yellow, black};
colors myColor = blue, yourColor = red;
```

  – The `colors` type uses the default internal values for the enumeration constants, 0, 1, …, although the constants could have been assigned any **integer** literal.

- In 2004, an enumeration type was added to Java in Java 5.0. All enumeration types in Java are implicitly subclasses of the predefined class **`Enum`**
- Interestingly, **none** of the relatively recent scripting languages include enumeration types.
  – These included **Perl, JavaScript, PHP, Python, Ruby,** and **Lua**
  – Even Java was a decade old before enumeration types ware added

6.4.3 Evaluation

- Enumeration type can provide advantages in **both** readability and reliability.
- Aid readability
  - e.g., no need to code a color as a number
- Aid to reliability, e.g., compiler can check:
  - **No** arithmetic operations are legal on enumeration types
    - e.g., don’t allow colors to be added
  - No enumeration variable can be assigned a value outside its defined range
    - e.g., If the colors enumeration type has 10 enumeration constants and uses 0..9 as its internal values, no number greater than 9 can be assigned to a colors type variable.
  - 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

6.5 Array Types 250

- 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.
- The individual data elements of an array are of the **same** type.
- References to individual array elements are specified using subscription expressions.
- If any of the subscript expressions in a reference include variables, then the reference will require an addition run-time calculation to determine the address of the memory location being referenced.

6.5.1 Design Issues

- The primary design issues specific to arrays are the following:
  - 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?
  - Can arrays be initialized when they have their storage allocated?
  - What kinds of slices are allowed, if any?

6.5.2 Arrays and Indices

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

```
array_name (index_value_list) → an element
```

- Ex, Ada assignment statement:

```
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.
- Among contemporary languages, C, C++, Perl, and Fortran **don't** specify range checking of subscripts, but Java, ML, and C# **do**.
- In Perl, all arrays begin with at sign (@), because array elements are always scalars and the names of scalars always being with dollar signs ($), references to array elements use dollar signs rather that at signs in their names. For example, for the **@list**, the second element is referenced with **$list[1]**.

6.5.3 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**.

- There are **four** categories of arrays, based on the binding to subscript ranges, the binding to storage, and rom where the storage is allocated.

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 **fixed heap-dynamic array** is similar to fixed stack-dynamic in which the subscript ranges are dynamically bound, and the storage allocation is dynamic, but they are both fixed after storage is allocated (i.e., binding is done when requested and storage is allocated from heap, not stack)
   - The bindings are done when the user program requests them during execution, 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 non-generic arrays are **fixed heap dynamic arrays**. Once created, they keep the same subscript ranges and storage.

4. 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:
     - Objects of the C# `List` class are generic **heap-dynamic arrays**. These array object are created without any elements, as in

```
List<String> stringList = new List<Stirng>( );
```

Elements are added to this object with the `Add` method, as in

```
stringList.Add("Michael");
```

- Java includes a generic class similar to C#'s List, named **ArrayList**.
- Perl, JavaScript, Python, and Ruby support heap-dynamic arrays.
  - A Perl array and JavaScript also support heap-dynamic array to grow with the `push` (puts one or more new elements on the end of the array) and `unshift` (puts one or more new elements on the beginning of the array)
  - 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 array's value is (1, 2, 4, 7, 10, 13, 17).

6.5.4 Array Initialization

- Some language allow initialization at the time of storage allocation.
- Usually just a list of values that are put in the array in the order in which the array elements are stored in memory.
- C, C++, Java, and C# allow initialization of their arrays. Consider the following C declaration:

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

  – The compiler sets the length of the array.
- Character Strings in C & C++ are implemented as arrays of **char**. These arrays can be initialized to string constants, as in

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

  – The array will have 8 elements because all strings are terminated with a **null** character(zero), which is implicitly supplied by system for string constants.
- Arrays of strings in C and C++ can also be initialized with string literals. For example,

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

- In Java, similar syntax is used to define and initialize an array of references to `String` objects. For example,

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

6.5.5 Array Operations

- The most common array operations are assignment, catenation, comparison for equality and inequality, and slices.
- The C-based languages do **not** provide any array operations, except thought methods of Java, C++, and C#.
- Perl supports array assignments but does not support comparisons.
- Python's arrays are called **lists**, although they have all the characteristics of dynamic arrays. Because the objects can be of any types, these arrays are **heterogeneous**. Python's array assignments, but they are only reference changes. Python also supports array catenation and element membership operations
- Ruby also provides array catenation
- APL provides the most powerful array processing operations for vectors and matrixes as well as unary operators (for example, to reverse column elements)

6.5.6 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. A jagged matrix has rows with varying number of elements.
  - For example, a jagged matrix may consist of three rows, one with 5 elements, one with 7 elements, and one with 12 elements.
  - Possible when multi-dimensioned arrays actually appear as arrays of arrays
- C, C++, and Java support jagged arrays but **not** rectangular arrays. In those languages, a reference to an element of a multidimensioned array uses a **separate** pair of brackets for each dimension. For examples,

```
myArray[3][7];
```

- C# and F# support rectangular arrays and jagged arrays. For rectangular arrays, all subscript expressions in references to elements are placed in a **single** pair of brackets. For example,

```
myArray[3, 7]
```

6.5.7 Slices

- A slice of an array is some **substructure** of an array.
- It is a mechanism for referencing part of an array as a unit.
- If arrays cannot be manipulated as units in a language, that has no use for slices. 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]]
```

  - `vector (3:6)` is a three-element array, which is `[8, 10, 12]`
  - `mat[0][0:2]` is the first and second element of the first row of mat, which is `[1, 2]`
- Ruby supports slices with the `slice` method

```
list = [2, 4, 6, 8, 10]
```

  - `list.slice(2, 2)` return returns the third and fourth elements of list: `[6, 8]`
  - `list.slice(1..3)` return `[4, 6, 8]`

6.5.8 Evaluation

- Arrays have been included in virtually all programming languages.
- The primary advances since their introduction in Fortran I have been slices and dynamic arrays.
- The latest advances in arrays have been in associate arrays.

6.5.9 Implementation of Arrays

- Access function maps subscript expressions to an address in the array
- A single-dimensioned array is implemented as a list of **adjacent** memory cells.
  - The address to be accessed by a reference such as: list[k]
  - Suppose the array list is defined to have a subscript range lower bound of 0. The access function for list is often of the form

    address(list[k]) = address (list[0]) + k * element_size

- Access function for single-dimensioned arrays:

    address(list[k]) = address (list[lower_bound]) + ((k-lower_bound) * element_size)

- The compile-time descriptor for single-dimensioned arrays can have the form show in Figure 6.4.

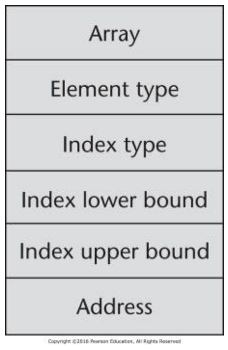


Figure 6.4 Compile-time descriptor for single-dimensioned arrays

- Accessing Multi-dimensioned Arrays.
- Two common ways:
  - Row major order (by rows) – used in **most** languages
  - Column major order (by columns) – used in Fortran
- 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 [0,0] + ((i * n) + j)) * element_size

- In Figure 6.5, we assume that subscript lower bounds are all zero.

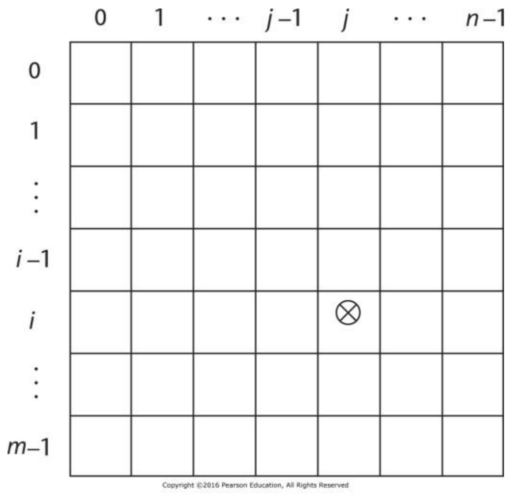
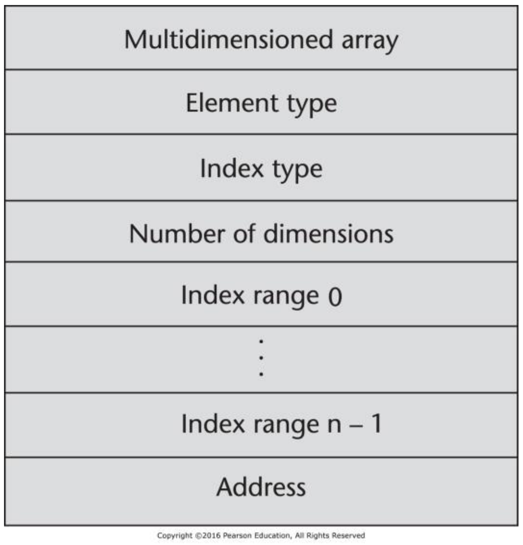


Figure 6.5 The location of the [i,j] element in a matrix

- The compile-time descriptor for a multidimensional array is show in Figure 6.6.



Multidimensioned array
Element type
Index type
Number of dimensions
Index range 0
⋮
Index range n – 1
Address

Copyright ©2016 Pearson Education, All Rights Reserved

Figure 6.6 A compile-time descriptor for a multidimensional array

6.6 Associative Arrays 261

- 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, C++, C#, and F#.
- Example: In Perl, associative arrays are often called **hashes**. Names begin with %; literals are delimited by parentheses. Hashes can be set to literal values with assignment statement, as in

```
%salaries = (“Gary” => 75000, “Perry” => 57000,
             “Mary” => 55750, “Cedric” => 47850);
```

  – Subscripting is done using braces and keys. So an assignment of `58850` to the element of `%salaries` with the key `“Perry”` would appear as follows:

```
$salaries{“Perry”} = 58850;
```

  – Elements can be removed with **delete** operator, as in the following:

```
delete $salaries{“Gary”};
```

  – Elements can be emptied by assigning the empty literal, as in the following:

```
@salaries = ();
```

- Python’s associative arrays, which are called **dictionaries**, are similar to those of Perl, except the values are all reference to objects.
- PHP’s arrays are both normal arrays and associative array.
- A Lua table is an associate array in which both the keys and the values can by any type.
- C# and F# support associative arrays through a .NET class.

6.7 Record Types 263

- 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.

6.7.1 Definitions of Records

- The fundamental difference between a record and an array is that record elements, or **fields**, are **not** referenced by indices. Instead, the fields are named with identifier, and references to the fields are made using these identifiers.
- The COBOL form of a record declaration, which is part of the data division of a COBOL program, is illustrated in the following example:

```
01 EMPLOYEE-RECORD.
   02 EMPLOYEE-NAME.
      05 FIRST     PICTURE X(20).
      05 Middle    PICTURE X(10).
      05 LAST      PICTURE X(20).
   02 HOURLY-RATE PICTURE 99V99.
```

  – COBOL uses level **numbers** to show nested records; others use recursive definition
  – The numbers `01`, `02`, and `05` that begin the lines of the record declaration are level numbers, which indicate by their relative values the hierarchical structure of the record.
  – The `PICTURE` clauses show the formats of the field storage locations, with `X(20)` specifying 20 alphanumeric characters and `99V99` specifying four decimal digits with decimal point in the middle.

6.7.2 References to Records

- References to the individual fields of records are syntactically specified by seral different methods, two of which name the desired field and its enclosing records. COBOL field references have the form

  field_name `OF` record_name_1 `OF . . . OF` record_name_n

- For example, the `Middle` field in the COBOL record example above can be reference with

```
Middle OF EMPLOYEE-NAME OF EMPLOYEE-RECORD
```

- Most language use dot notation

```
Employee_Record.Employee_Name.Middle
```

- 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 EMPLOYEE-NAME`, and `FIRST OF EMPLOYEE-RECORD` are elliptical references to the employee's first name

6.7.3 Evaluation

- Records and arrays are closely related structural forms.
- Arrays are used when all the data values have the **same** type and/or are processed in the same way.
- Records are used when the collection data values is **heterogeneous** and the different fields are not processed in the same way. Also, the fields of a record often need not be processed in a particular order.
- Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)

6.7.4 Implementation of Record Types

- The fields of records are stored in adjacent memory locations.
- Field accesses are all handled using these **offsets**. Offset address relative to the beginning of the records is associated with each field
- The compile-time descriptors for a record has the general form show in Figure 6.7.

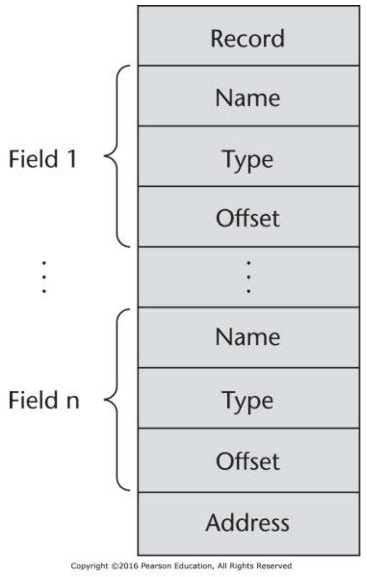


Figure 6.7 A compile-time descriptor for a record

6.8 Tuple Types 266

- A tuple is a data type that is similar to a record, except that the elements are **not** named
- Python
  - Closely related to its **lists**, but tuples are **immutable**
  - If a tuple needs to be changed, it can be converted to an array with the list function
  - Create with a tuple literal

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

  - Note that the elements of a tuple need **not** be of the same type
  - The elements of a tuple can be referenced with indexing in brackets, as in the following:

```
myTuple[1]
```

This references the first element of the tuple, because tuple indexing begins at **1**

  - Tuple can be catenated with the plus (+) operator
  - They can be deleted with **del** statement

- ML
  - Create with a tuple

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

  - Access as follows:

```
#1(myTuple);
```

This reference the first element

  - A new tuple type can be defined in ML with a type declaration, such as the following

```
type intReal = int * real;
```

- F#

```
let tup = (3, 5, 7);;
let a, b, c = tup;;
```

This assign `3` to `a`, `5` to `b`, and `7` to `c`

- Type are used in Python, ML, and F# to allow functions to return multiple values

6.9 List Types 268

- Lists were first supported in the first **functional** programming language.
- Lists in Common Lisp and Scheme are delimited by parentheses and the elements are not separated by any punctuation (no commas). For example

  ```
  (A B C D)
  ```
  Nested lists have the same form, so we could have

  ```
  (A (B C) D)
  ```
  In this list, `(B C)` is a list nested inside the outer list

- Data and code have the same form
  - As data, `(A B C)` is literally what it is
  - As code, `(A B C)` is the function A applied to the parameters `B` and `C`
- The interpreter needs to know which a list is, so if it is data, we quote it with an apostrophe

  ```
  `(A B C)                  is data
  ```

- List operations in Scheme
  - CAR returns the first element of its list parameter

    ```
    (CAR `(A B C))            returns A
    ```

  - CDR returns the remainder of its list parameter after the first element has been removed

    ```
    (CDR `(A B C))            returns (B C)
    ```

  - CONS puts its first parameter into its second parameter, a list, to make a new list

    ```
    (CONS `A `(B C))          returns (A B C)
    ```

  - LIST returns a new list of its parameters

    ```
    (LIST `A `B `(C D))       returns (A B (C D))
    ```

- List operations in ML
  - Lists are written in brackets and the elements are separated by commas, as in the following list of integers:

    ```
    [5, 7, 9]
    ```

  - List elements must be of the same type, so the following list would be illegal:

    ```
    [5, 7.3, 9]               illegal
    ```

  - The Scheme CONS function is implemented as a binary infix operator in ML, represented as ::, For example,

    ```
    3 :: [5, 7, 9]            return new list [3, 5, 7, 9]
    ```

- ML has functions that correspond to Scheme's CAR and CDR functions are named `hd` (head) and `tl` (tail), respectively

```
hd [5, 7, 9]                is 5
tl [5, 7, 9]                is [7, 9]
```

- F# Lists
  - Like those of ML, except elements are separated by semicolons and `hd` and `tl` are methods of the `List` class

```
List.hd [1; 3; 5; 7]      return 1
```

- 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]               Assign 5.8 to x
```

  - List elements can be deleted with `del`

```
del myList[1]
```

  - **List Comprehensions** – A list comprehension is an idea derived from **set** notation.

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

    - The `range(12)` function creates `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]`
    - The conditional filter out all numbers in the array that are not evenly divisible by `3`. This list comprehension returns the following array: `[0, 9, 36, 81]`

- Haskell's List Comprehensions

```
[n * n | n <- [1..10]]
```

    - This define a list of the squares of the numbers from 1 to 10

- F#'s List Comprehensions

```
let myArray = [|for i in 1 .. 5 -> [i * i) |]
```

    - This statement creates the array `[1; 4; 9; 16; 25]` and names it `myArray`

- Both C# and Java supports lists through their generic heap-dynamic collection classes, `List` and `ArrayList`, respectively. These structures are actually **lists**.

6.10 Union Types 270

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

6.10.1 Design Issues

- The major design issue: Should type checking be required?

6.10.2 Discriminated vs. Free Unions

- 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.**

```
union flexType {
   int    intEl;
   float  floatEl;
};
union flexType el1;
float x;
. . .
el1.intE1 = 27;
x = el1.floatE1;              // assign 27 to float variable x
```

  - This last assignment is not type checked, because the system **cannot** determine the current type of the current value of el1, so it assigns the bit string representation of `27` to `float` variable `x`, which of is nonsense.

- Type checking of unions requires that each union include a type indicator called a **discriminated union**. The firs language to provide discriminated union was ALGOL 68. They are supported by ML, Haskell, and F#.

6.10.3 Unions in F#

- A union is declared in F# with a type statement using OR operators (|)

```
type intReal =
   | IntValue of int
   | RealValue of float;;
```

  - `intReal` is the new type
  - `IntValue` and `RealValue` are constructors

- Values of type `intReal` can be created using the constructors as if they were a function, as in the following examples:

```
let ir1 = IntValue 17;;
let ir2 = RealValue 3.4;;
```

- Accessing the value of a union is done with pattern-matching structure

```
match pattern with
  | expression_list₁ -> expression₁
  | …
  | expression_listₙ -> expressionₙ
```

  – Pattern can be any data type
  – The expression list can have wild cards (_) or be solely a wild card character.

- Example:

```
let a = 7;;
let b = "grape";;
let x = match (a, b) with
        | 4, "apple" -> apple
        | _, "grape" -> grape
        | _ -> fruit;;
```

- To display the type of the `intReal` union:

```
let printType value =
     match value with
         | IntVale value -> printfn "It is an integer"
         | RealValue value -> printfn "It is a float";;
```

  – If `ir1` and `ir2` are defined as previously,

```
printType ir1;;            output: It is an integer
printType ir2;;            output: It is a float
```

6.10.4 Evaluation

- Potentially **unsafe** construct
  – They are one of the reasons why C and C++ are not strongly typed
- Java and C# do **not** support unions
  – Reflective of growing concerns for safety in programming language

6.10.5 Implementation of Union Types

- Unions are by implemented by simply using the **same** address for every possible variant. Sufficient storage for the largest variant is allocated.

6.11 Pointer and Reference Types 273

- A pointer type in which the variables 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.
- Pointers are designed for two distinct kinds of uses:
  - 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**)

6.11.1 Design Issues

- The primary design issues particular to pointers are the following:
  - 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?

6.11.2 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. This process is show in Figure 6.8.
```

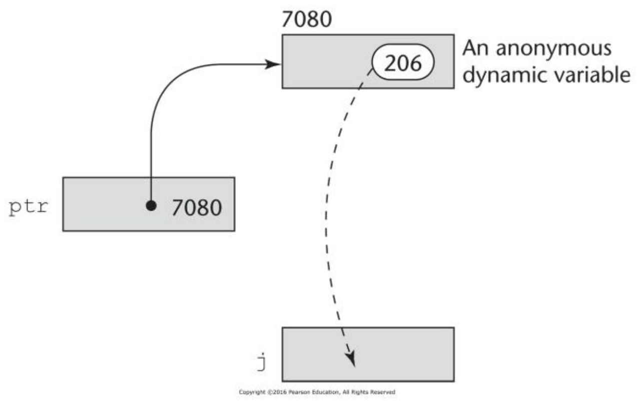

7080
206
An anonymous dynamic variable
ptr
7080
j
Copyright ©2016 Pearson Education, All Rights Reserved

Figure 6.8 The assignment operation j = *ptr

- In C and C++, there are **two** ways a pointer to a record can be used to reference a field in that record.
  - If a pointer variable `p` points to a record with a field name age, `(*p).age` can be used to refer to that field.
  - The operator `->`, when use between a pointer to a `struct` and a field of that `struct`, combines dereferencing and field reference.
  - For example, the expression

    `p -> age` is equivalent to `(*p).age`

- Languages that provide pointers for the management of a **heap** must include an explicit allocation operation.
  - Allocation is sometimes specified with a subprogram, such as `malloc` in C.
  - In a language that support object-oriented programming, allocation of heap objects is often specified with `new` operation. C++, which does **not** provide implicit deallocation, used `delete` as its deallocation operator.

6.11.3 Pointer Problems

- **Dangling Pointers** (**dangerous**)
  - A pointer points to a heap-dynamic variable that has been **deallocated**.
  - Dangling pointers are dangerous for the following reasons:
    1. The location being pointed to may have been allocated to some new heap-dynamic variable. If the new variable is not the same type as the old one, type checks of uses of the dangling pointer are invalid.
    2. Even if the new one is the same type, its new value will bear no relationship to the old pointer's dereferenced value.
    3. If the dangling pointer is used to change the heap-dynamic variable, the value of the heap-dynamic variable will be destroyed.
    4. 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:
    1. A new heap-dynamic variable is created and pointer `p1` is set to point at a new heap-dynamic variable.
    2. Pointer `p2` is assigned `p1`'s value.
    3. The heap-dynamic variable pointed to by `p1` is explicitly deallocated (possibly setting `p1` to `nil`), but `p2` is not changed by the operation. `P2` is now a dangling pointer. If the deallocation operation did not change `p1`, both `p1` and `p2` would dangling. (Of course, this is a problem of aliasing – `p1` and `p2` are aliases.)

- For example, in C++ we could have the following:

```
int * arrayPtr1;
int * arrayPtr2 = new int[100];
arrayPtr1 = arrayPtr2;
delete []   arrayPtr2;
// Now, arrayPtr1 is dangling, because the help storage
// to which it was pointing has been deallocated.
```

  - In C++, **both** `arrayPtr1` and `arrayPtr2` are now dangling pointers, because the C++ delete operator has **no** effect on the value of its operand pointer.

- **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 cannot be reallocated for some new use in the program.
  - Lost heap-dynamic variables are often created by the following sequence of operations:

    1. Pointer `p1` is set to point to a newly created heap-dynamic variable.
    2. `p1` is later set to point to another newly created heap-dynamic variable.
  - The first heap-dynamic variable is now inaccessible, or lost.
  - The process of losing heap-dynamic variables is called **memory leakage**.

6.11.4 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.
- C and C++ pointers can point at any variable, regales of where it is allocated. In fact, they can point **anywhere** in memory, whether there is a variable there or not, which is one of the dangers of such pointers.
- 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;              // variable ptr sets to the address of init
count = *ptr;             // dereference ptr to produce the value at init,
                          //   then assign to count
```

  – the two assignment statement are equivalent to the single assignment

```
count = init;
```

- Example: Pointer Arithmetic in C and C++

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

  Now consider the assignment

```
ptr = list;
```

  – This assigns the address of `list[0]` to `ptr`. Given this assignment, the following are true:
    - `*(ptr + 1)` is equivalent to `list[1]`
    - `*(ptr + index)` is equivalent to `list[index]`
    - `ptr[index]` is equivalent to `list[index]`
      - ex. `*(ptr+5)` is equivalent to `list[5]` and `ptr[5]`

- C and C++ include pointers of type `void *`, which can point at values of **any** type. In effect they are generic pointers.
  – Type checking is not a problem with `void *` pointers, because these languages disallow dereferencing them.
  – One common use of `void *` pointers is as the types of parameters of functions that operate on memory.

6.11.5 Reference Types

- A reference type variable is similar to a pointer, with one important and fundamental difference: A pointer refers to an **address** in memory, while a reference refers to an object or a **value** in memory.
- 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;
```

  – In this code segment, result and ref_result are **aliases**.

- In their quest for increased safety over C++, the designers of Java removed C++ style pointer altogether.
- 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 reference variables 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.
- All variables in the object-oriented languages Smalltalk, Python, Ruby, and Lua are references. They are always **implicitly** dereferenced.

6.11.6 Evaluation

- **Dangling** pointers and **garbage** are problems as is heap management
- Pointers are like goto's
  - The goto statement widens the range of statements that can be executed next
  - Pointer variables widen the range of memory cells that can be referenced by a variable
- Pointers are essential in some kinds of programming applications.
  - For example, pointers are necessary to write device drivers, in which specific absolute addresses must be accessed
- The references of Java and C# provide some of the flexibility and the capabilities of pointers, without the hazards.
- It remains to be seen whether the programmers will be willing to trade the **full** power of C and C++ pointers for the greater safety of references.

6.12 Optional Types 285

- Optional types are useful when there is a need for a variable to indicate that it currently has **no value**
- C#, F#, and Swift, among others, have optional types
- C# has two categories of variables, value and reference types.
  - Reference types in C# are already optional types (use **null** for no value)
  - Value types in C# (struct types) can be declared to be optional by attaching a question mark (?) to the type name in their declaration

```
int? x;
```

  - The no-value is **null**, which can be assigned to x and x can be tested for it

```
int? x;
. . .
if(x == null)
   Console.WriteLine("x has no value");
else
   Console.WriteLine("The value of x is: {0}", x);
```

- In Swift, **nil** is used instead of null

```
var Int? x;
. . .
if x == nil
   print("x has no value")
else
   print("The value of x is: \(x)")
```

6.13 Type Checking 286

- 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**.
  - Ex: an `int` variable and a `float` variable are added in Java, the value of the `int` variable is coerced to `float` and a floating-point is performed.
- A **type error** is the application of an operator to an operand of an inappropriate type.
  - Ex: in C, if an `int` value was passed to a function that expected a `float` value, a type error would occur (compilers did **not** check the types of parameters)
- If all type bindings are static, nearly all type checking can be static.
- If type bindings are dynamic, type checking must be dynamic and done at **run-time**.
- Some languages, such as JavaScript and PHP, because of their type binding, allow only dynamic type checking.
- It is better to detect errors at compile time **than** at run time, because the earlier correction is usually less costly.
  - The penalty for static checking is reduced programmer **flexibility**.
- Type checking is **complicated** when a language allows a memory cell to store values of different types at different time during execution.

6.14 Strong Typing 287

- A programming language is strongly typed if type errors are **always** detected. This requires that the types of all operands can be determined, either at compile time or run time.
- Advantage of strong typing: allows the detection of the **misuses** of variables that result in type errors.
- C and C++ are **not** strongly typed language because both include **union** type, which are not type checked.
- **Java and C#** are strongly typed. Types can be explicitly cast, which would result in type error. However, there are no implicit ways type errors can go undetected.
- The coercion rules of a language have an important effect on the value of type checking.
- Coercion results in a loss of part of the reason of strong typing – error detection.
  - Ex:

```
int a, b;
float d;
a + d;    // the programmer meant a + b, but mistakenly type a + d
// The compiler would not detect this error. Variable a would be coerced to float
```

- So, the value of strong typing is **weakened** by coercion
  - Languages with a great deal of coercion, like C and C++ are **less** reliable than those with no coercion, such as ML and F#.
  - Java and C# have **half** as many assignment type coercions as C++, so their error detection is better than that of C++, but still not nearly as effective as that of ML and F#.

6.15 Type Equivalence 288

- Two types are equivalent if an operand of one type in an expression is substituted for one of the other type, **without** coercion.
- There are **two** approaches to defining type equivalence: name type equivalence and structure type equivalence.
- **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
  - Ex, Ada

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

    - The type of the variables `count` and `index` would **not** be equivalent; ; `count` could not be assigned to `index` or vice versa.

- **Structure type equivalence** means that two variables have equivalent types if their types have identical structures
  - More flexible, but harder to implement

    ```
    type Vector is array  (Integer range <>) of Integer;
    Vector_1:  Vector (1..10);
    Vector_2:  Vector (11..20);
    ```

    - The types of these two objects are equivalent, even though they have different names and different subscript rang, because of unconstrained array types.

- C uses both name and structure type equivalence.
  - Name type equivalence is used for structure, enumeration, and union types.
  - Other nonscalar types use structure type equivalence. Array type are equivalence if they the same type components. Also, if an array type has a constant size, it is equivalent either to other arrays with the same constant size or to with those without a constant size.
- In languages that do not allow users to define and name types, such as Fortran and COBOL, names equivalence obviously cannot be used.

6.16 Theory and Data Types 292

- Type theory is a broad area of study in mathematics, logic, computer science, and philosophy
- Two branches of type theory in computer science:
  - Practical – The practical branch concerned with data types in commercial programming languages
  - Abstract – The abstract branch primarily focuses on typed lambda calculus, an area of extensive research by theoretical computer scientist over the past half century
- A data type defines a set of values and a collection of operations on those values
- A type system is a set of types and the rules that govern their use in programs

Summary 294

- A data type defines a collection of data **values** and a set of predefined **operations** on those values
- 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
- 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
- An **associative array** is an unordered collection of data elements that are indexed by an equal number of values called keys.
  - Each element of an associative array is in fact a pair of entities, a **key** and a **value**
- A **record** is a possibly **heterogeneous** aggregate of data elements in which the individual elements are identified by names
- There are **four** categories of arrays, based on the binding to subscript ranges, the binding to storage, and rom where the storage is allocated.
  - **Static array**: as in C++ array whose definition includes the static specifier
  - **Fixed stack-dynamic array**: as in C function (without the static specifier)
  - **Fixed heap-dynamic array**: as with Java's objects
  - **Heap dynamic array**: as in Java's ArrayList and C#'s List
- **Tuples** are similar to records, but do **not** have names for their constituent parts. They are part of Python, ML, and F#. Python's tuples are closed to lists, but **immutable**
- **Lists** are staples of the functional programming languages, but are now also included in Python and C#. Python's lists are **mutable**
- **Unions** are locations that can store **different** type values at different times
- A **pointer** type in which the variables have a range of values that consists of **memory addresses** and a special value, nil.
  - Pointers are used for addressing flexibility and to control dynamic storage management
  - Pointers have some inherent dangers: **dangling pointers** are difficult to avoid, and **memory leakage** can occur
- **Reference** types, such as those in Java and C#, provide heap management **without** the danger of pointers
- Strong typing is the concept of requiring that **all** type errors be detected
- The type equivalence rules of a language determine what **operations** are legal among the structured types of a language