Chapter 6 Data Types

6.1 Introduction

- A data type defines a collection of data values and a set of predefined operations on those values.
  - In pre-90 Fortrans, linked lists and binary trees were implemented with arrays
  - ALGOL 68, provides a few basic types and a few flexible structure-defining operators that allow a programmer to design a data structure for each need.

6.1 Introduction (Cont'd)

– Abstract data type supported by most programming languages designed since the mid-1980s.

6.1 Introduction (Cont'd)

- Uses of type system in PL
  - Error detection
  - Program modularization
  - Document
- The type system defines how a type is associated with each expression and includes its rule for type equivalence and type compatibility

6.1 Introduction (Cont'd)

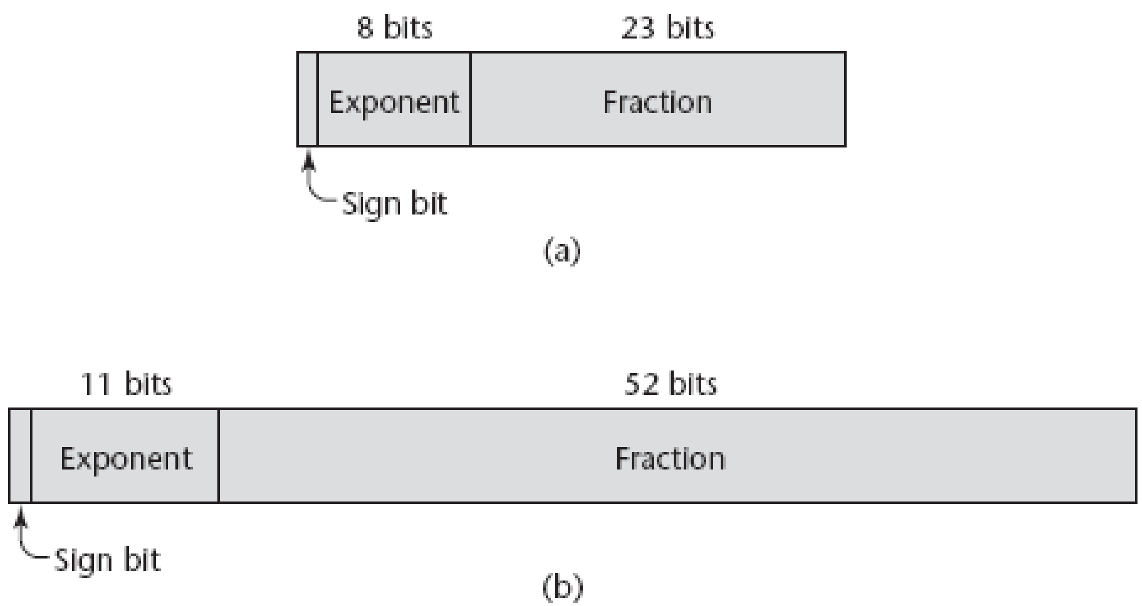
- Think of variables in terms of descriptors
  - A descriptor is the collection of attributes of a variable
  - Static descriptor & dynamic descriptor

6.2 Primitive Data Types

- Numeric Types
  - Integer
  - Floating-point
    - IEEE Floating-Point Standard 754 format
    - See next page
  - Complex
    - Fortran and Python

Figure 6.1

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



6.2 Primitive Data Types (Cont'd)

- Decimal
  - To support business systems applications
    - COBOL, C#, F#
  - To precisely store decimal number
  - binary coded decimal (BCD)
- Boolean Types
  - Introduced in ALGOL 60

6.2 Primitive Data Types (Cont'd)

- Character Types
  - Traditionally, 8-bit code ASCII
    - 0 to 127
  - EASCII
    - Extended ASCII, ISO 8859-1
    - Allow 256 different characters
      - See next slice.

128	Ç	144	É	160	á	176	░	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	^{n}
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	^{2}
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

Source: www.LookupTables.com

6.2 Primitive Data Types (Cont'd)

- UCS-2
  - 16-bit character set
  - Called Unicode
  - Java was the first widely used language to use the Unicode
    - » JavaScript, Python, Perl, C#, F#

📦 UCS-2 Encoding Example

UCS-2 always uses **2 bytes** per character. It only supports the **Basic Multilingual Plane (BMP)**.

```text
"A"    => U+0041 => [00 41]
"中"    => U+4E2D => [4E 2D]
"😊"   => U+1F60A => ❌ Cannot be encoded (outside BMP)
```

6.2 Primitive Data Types (Cont'd)

- UTF-16
  - An extension of UCS-2
    - » 2 or 4 bytes

Feature	UCS-2	UTF-16
Encoding length	Fixed: 2 bytes per character	Variable: 2 or 4 bytes
Unicode range	`U+0000` to `U+FFFF` (BMP only)	`U+0000` to `U+10FFFF` (includes Supplementary Planes)
Supports emoji	❌ No	✅ Yes (via surrogate pairs)
Supports ancient scripts	❌ No	✅ Yes
Uses surrogate pairs	❌ No	✅ Yes
Status	Legacy / obsolete	Widely used (Windows, Java, etc.)

> 在 UTF-16 中，超過 16 位元（即超過 U+FFFF） 的碼點會用「代理對（surrogate pair）」來表示，那麼它和普通的 16-bit 字元（BMP 範圍）混在一起時，解碼器怎麼知道什麼是單個字元、什麼是兩個合起來的字元，不會搞混？

答案是：**UTF-16 設計時就考慮到這點，它使用「特殊保留區」來避免混淆。**

🔑 關鍵設計：使用「代理區（surrogate range）」

UTF-16 的代理對編碼方式如下：

名稱	範圍（16-bit 編碼）	說明
高位代理（High Surrogate）	`D800` – `DBFF`	第一個 16-bit 單元
低位代理（Low Surrogate）	`DC00` – `DFFF`	第二個 16-bit 單元

這兩段 **2048 個值**（共佔 `D800` ~ `DFFF` 的範圍）**專門保留給代理對使用，BMP 中不會出現這些值作為單獨字元**。

✅ 解碼時怎麼判斷？

當解碼器看到一個 16-bit 單元時：

1. **如果它不在 `D800` ~ `DFFF` 之間** →
   ➤ 它就是一個獨立的 BMP 字元（如：`0041` 是 A）。
2. **如果它在 `D800` ~ `DBFF` 之間** →
   ➤ 它是代理對的「高位代理」，**解碼器會預期下一個是低位代理（`DC00` ~ `DFFF`）**。
3. **如果搭配正確的低位代理，就能組成一個字元**（如 Emoji）。
4. **如果格式錯誤**（例如低位代理不跟在高位後）↓ ;碼器會報錯或用替代字元。

6.3 Character String Types

- A character string type is one in which the values consist of sequences of characters
- Design issues:
  - Should strings be simply a special kind of character array or a primitive type?
  - Should strings have static or dynamic length?

6.3.2 Strings and Their Operations

- Most common string operations
  - Assignment, catenation, substring reference, comparison, and pattern matching
- If strings are not defined as a primitive type
  - Stored in arrays of single characters
  - Taken by C and C++
  - `str` is an array of **`char`** elements, specifically `apple0`, where `0` is the null charater.

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

6.3.2 Strings and Their Operations (Cont'd)

- The string manipulation functions of the C standard library, also available in C++ are inherently unsafe
- Consider the following situations, |`dest`|=20, and |`src`|=50:

```
strcopy (dest, src);
```

- C++ also supports `string` class.

6.3.2 Strings and Their Operations (Cont'd)

- In Java,
  - `String` class
    - Constant string
    - For each assignment to a `String` object, a new object should be created (instantiated).

```
String S1 = "abc";

     For(int I = 0 ; I < 10000 ; I ++)          {
            S1 + = "def";
            S1 = "abc";
     }
```

6.3.2 Strings and Their Operations (Cont'd)

- In Java,
  - `StringBuffer` class
    - Changeable

```
StringBuffer Sb = new StringBuilder("This is only
a").append("simple").append("test");
```

6.3.2 Strings and Their Operations (Cont'd)

- Building-in pattern-matching operations of strings
  - Perl, JavaScript, Ruby, PHP…
  - Regular expression
    - E.g.

```
/[A-Za-z][A-za-z\d]+/     (name form in PL)
/\d+\.?\d* | \.\d+/       (numeric literal)
```

- Included in class libraries of pattern-matching operations of strings
  - C++, Java, Python, C#, F#

Regular Expressions (補充教材)

- Tokens are built from symbols of a finite vocabulary.
- We use regular expressions to define structures of tokens.

Regular Expressions

- The sets of strings defined by regular expressions are termed ***regular sets***
- Definition of regular expressions
  - ∅ is a regular expression denoting the empty set
  - λ is a regular expression denoting the set that contains only the empty string
  - A string **s** is a regular expression denoting a set containing only **s**
  - if A and B are regular expressions, so are
    - A | B (alternation)
    - AB (concatenation)
    - A* (Kleene closure)

Regular Expressions (Cont'd)

some notational convenience

P^+ == PP^*

Not(A) == V - A

Not(S) == V^* - S

A^K == AA …A (k copies)

Regular Expressions (Cont'd)

- Some examples

Let $D = (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid \ldots \mid 9)$

$L = (A \mid B \mid \ldots \mid Z)$

comment = -- not(EOL)* EOL

decimal = $D^+ \cdot D^+$

ident = $L\ (L \mid D)^*\ (\_\ (L \mid D)^+)^*$

comments = $\#\#((\# \mid \lambda)\,\text{not}(\#))^*\ \#\#$

Regular Expressions (Cont'd)

- Is regular expression as power as CFG?

$\{ [^i]^i \mid i \geq 1 \}$

6.3.3 String Length Options

- Static length string
  - Strings of Python, Java's String class, C++, Ruby's built-in `String` class, .NET class library
- Limited dynamic length strings
  - Allow strings to have varying length up to a declared and fixed maximum set
    - Strings in C

6.3.3 String Length Options

- Dynamic length string
  - Have varying length with no maximum
    - JavaScript, Perl, Java's `StringBuffer`

6.3.4 Evaluation

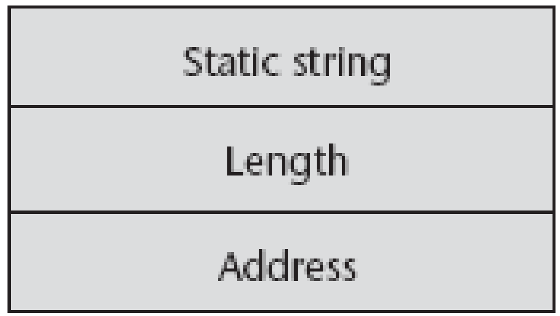
- The addition of strings as a primitive type to a language is not costly in terms of either language or compiler complexity.
- Providing strings through a standard library is nearly as convenient as having them as a primitive type.

6.3.5 Implementation of Character String Types

- A descriptor for a static character string type, which is required only during compilation, has three fields.

Figure 6.2

Compile-time descriptor for static strings

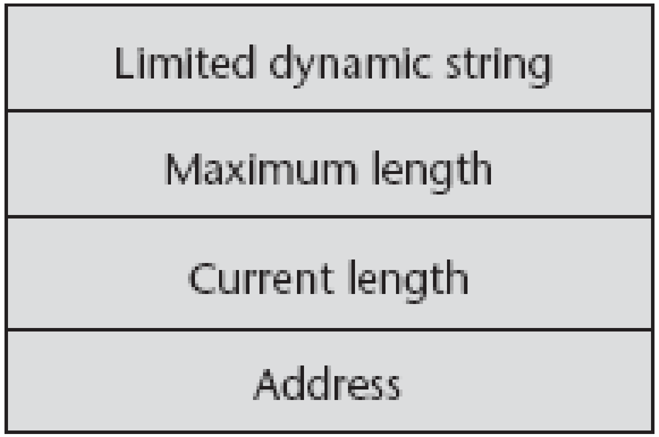


6.3.5 Implementation of Character String Types (Cont'd)

- Limited dynamic strings require a run-time descriptor to store both the fixed maximum length and the current length.

Figure 6.3

Run-time descriptor for limited dynamic strings



6.3.5 Implementation of Character String Types (Cont'd)

- The limited dynamic strings of C does not require run-time descriptor
  - End of a string is marked with the null character.

6.3.5 Implementation of Character String Types (Cont’d)

- Dynamic length strings require more complex storage management
  - (1) Strings can be stored in a linked list
  - (2) Store strings as arrays of pointers to individual characters allocated in the heap
  - (3) Store complete strings in adjacent storage cells
    - How to deal when a string grows

6.4 Enumeration Types

- An enumeration type is one in which all of the possible values, which are named constants, are provided, or enumerated, in the definition.
  - Provides a way of defining and grouping collections of named constants
    - **Enumeration constants**

6.4 Enumeration Types (Cont'd)

- C example

```
#include <stdio.h>
enum week{ sunday, monday, tuesday, wednesday,
thursday, friday, saturday};
int main(){
    enum week today;
    today=wednesday;
    printf("%d day\n",today+1);
    return 0;
   }
```

Output: 4 day

6.4.1 Design issues

- 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?

6.4.2 Designs

- Why enumeration type?
  - We can simulate them with integer values

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

  - The problem is we have not defined a type for our colors.
    - No type checking when they are used.

6.4.2 Designs (Cont’d)

- Why enumeration type?
  – We can simulate them with integer values

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

  – The problem is we have not defined a type for our colors.
    - No type checking when they are used.

6.4.2 Designs (Cont'd)

- C and Pascal were the first widely used languages to include an enumeration data type.
  - C++ includes C's enumeration types
    - `myColor++;` ⇒ legal
    - `myColor=4;` ⇒ illegal
    - `int i=myColor;` ⇒ legal (called coerce)

6.4.2 Designs (Cont'd)

- Java
  - Enumeration type was added to Java in Java 5.0
  - No expression of any other type can be assigned to an enumeration variable
  - An enumeration variable is never coerced to any other type. (See next slice)
- C#
  - Like those of C++, except that they are never coerced to any other type.

```java
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}

public class EnumTest {
    Day day;

    public EnumTest(Day day) {
        this.day = day;
    }

    public void tellItLikeItIs() {
        switch (day) {
            case MONDAY:
                System.out.println("Mondays are bad.");
                break;

            case FRIDAY:
                System.out.println("Fridays are better.");
                break;

            case SATURDAY: case SUNDAY:
                System.out.println("Weekends are best.");
                break;

            default:
                System.out.println("Midweek days are so-so.");
                break;
        }
    }
```

```java
    public static void main(String[] args) {
        EnumTest firstDay = new EnumTest(Day.MONDAY);
        firstDay.tellItLikeItIs();
        EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
        thirdDay.tellItLikeItIs();
        EnumTest fifthDay = new EnumTest(Day.FRIDAY);
        fifthDay.tellItLikeItIs();
        EnumTest sixthDay = new EnumTest(Day.SATURDAY);
        sixthDay.tellItLikeItIs();
        EnumTest seventhDay = new EnumTest(Day.SUNDAY);
        seventhDay.tellItLikeItIs();
    }
}
```

The output is:

```
Mondays are bad.
Midweek days are so-so.
Fridays are better.
Weekends are best.
Weekends are best.
```

6.4.3 Evaluation

- Enumeration types can provide advantages in both
  - Readability and
  - Reliability
    - In Ada, C#, F#, and Java 5.0
      - No arithmetic operations are legal
      - No enumeration variable can be assigned a value outside its defined range (See footnote)

6.5 Array Types

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

6.5.1 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?
- Are ragged or rectangular multidimensional arrays allowed, or both?
- What is the maximum number of subscripts?
- Can array objects be initialized?
- Are any kind of slices supported?

6.5.2 Arrays and Indices

- Specific elements of an array are referenced by means of a two-level syntactic mechanism, where the first part is the aggregate name, and the second part is a possible dynamic selector consisting of one or more items known as **subscripts** or **indices**

6.5.2 Arrays and Indices (Cont'd)

- The syntax of array references is fairly universal
  - The array name is followed by the list of subscripts which is surrounded by either parentheses or brackets
- In Ada

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

- Most languages other then Fortran and Ada use brackets to delimit their array indices

6.5.2 Arrays and Indices (Cont'd)

- Two distinct types are involved in an array type
  - The element type
  - The type of subscript

```
Type Week_Day_Type is (Mon, Tue, Wed, Thu, Fri);
Type Sales is array (Week_Day_Type) of Float;
```

6.5.2 Arrays and Indices (Cont'd)

- Early programming languages did not specify that subscript ranges must be implicitly checked
  - Range errors in subscripts are common
  - Unreliable
  - Java, ML, and C# do
    - Java may generate `java.lang.ArrayIndexOutOfBoundsException`

6.5.3 Subscript Bindings and Array Categories

- The binding of the subscript type to an array variable is usually static, but the subscript value ranges are sometimes dynamically bound
- In some language, the lower bound of the subscript range is implicit
  - C-based languages, fixed at 0

6.5.3 Subscript Bindings and Array Categories (Cont’d)

- The are five categories of arrays, based on the binding to script ranges, the binding to storage, and from where the storage is allocated
  - Static array
  - Fixed stack-dynamic array
  - Stack-dynamic array
  - Fixed heap-dynamic array
  - Heap-dynamic array

6.5.3 Subscript Bindings and Array Categories (Cont'd)

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

6.5.4 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"};
```

6.5.4 Array Initialization

- Ada
  - ```
    List1 : array (1..5) of Integer :=
       (1, 2, 3, 4, 5);
    ```
  - ```
    List2 : array (1..5) of Integer :=
       (1 => 17, 3 => 34, others => 0);
    ```

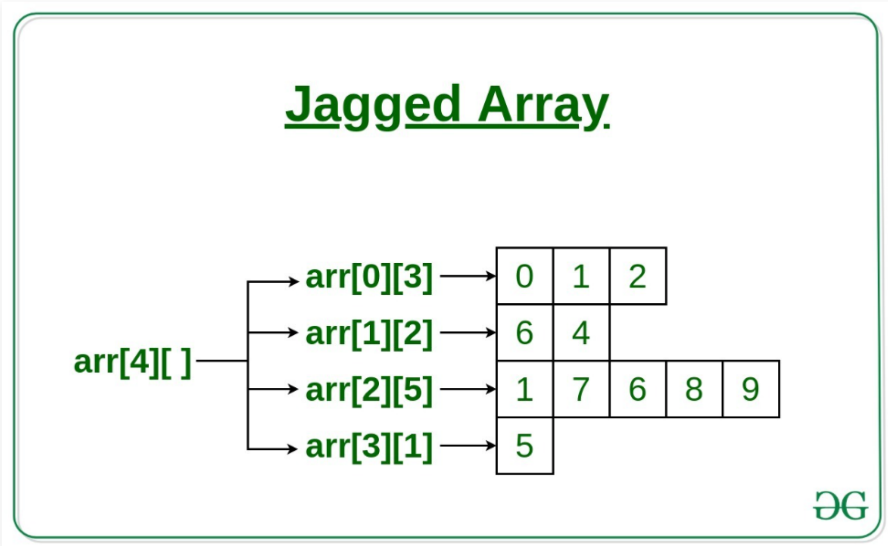
6.5.6 Rectangular and Jagged Arrays

- A rectangular array is a multidimensional array in which all of the rows have the same number of elements and all of the columns have the same number of elements
  - `int arr[100][50];`
- A jagged array is one in which the lengths of the rows need not be the same.
  - C does not support jagged array
    - To implement a jagged array, programmers must manually use an array of pointers along with malloc/free.

6.5.6 Rectangular and Jagged Arrays

Example:

```
arr[][] = { {0, 1, 2},
            {6, 4},
            {1, 7, 6, 8, 9},
            {5}
          };
```



```
// C program to show the
// implementation of Jagged Arrays

#include <stdio.h>
#include <stdlib.h>

int main()
{

    int row0[4] = { 1, 2, 3, 4 };
    int row1[2] = { 5, 6 };

    int* jagged[2] = { row0, row1 };

    // Array to hold the size of each row
    int Size[2] = { 4, 2 }, k = 0;

    // To display elements of Jagged array
    for (int i = 0; i < 2; i++) {

        // pointer to hold the address of the row
        int* ptr = jagged[i];

        for (int j = 0; j < Size[k]; j++) {
            printf("%d ", *ptr);

            // move the pointer to the
            // next element in the row
            ptr++;
        }

        printf("\n");
        k++;

        // move the pointer to the next row
        jagged[i]++;
    }

    return 0;
}
```

6.5.6 Rectangular and Jagged Arrays

- In Java you cannot specify array sizes in the variable declaration itself- so the following statement is illegal
  - `int b[10];`
  - `int a[10][100];`
- Java multi-dimensional arrays are really arrays of arrays, not one contiguous block of memory
  - `int[] b = new int[10];`
  - `int[][] a = new int[10][100];`
- Java multi-dimensional arrays are really arrays of arrays, not one contiguous block of memory. Even when you write new int[10][100], the JVM allocates one int[] of length 10, then for each of those 10 slots it allocates a separate int[100].

6.5.6 Rectangular and Jagged Arrays

- The jagged-array model ties directly into how the garbage collector works. Consider the following Java codes:
  - `int[][] a = new int[10][100];`
- The JVM actually allocates 11 separate objects:
  - One `int[][]` of length 10
  - Ten `int[]` sub-arrays of length 100 each
  - All 11 live on the heap and are individually tracked by the GC.
- If you later do
  - `a[5] = null;`
  - then that one `int[100]` becomes unreachable and can be collected, while the rest of a remains intact.
  - A true contiguous "rectangular" block would have been a single object, so you couldn't free part of it without freeing the whole.
  - Java's GC is designed to manage objects, not arbitrary chunks of memory.

6.6 Associative Arrays

- An associative array is an unordered collection of data elements that are indexed by an equal number of values call **keys**

6.6.1 Structure and Operations

- In Perl, associative arrays are called hashes
  - Names begin with **`%`**; literals are delimited by parentheses

```
%hi_temps = ("Mon" => 77, "Tue" => 79, "Wed"
  => 65, …);
```

- Subscripting is done using braces and keys

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

  - Elements can be removed with **`delete`**

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

6.6.2 Implementing Associative Arrays

- A 32-bit hash value is computed for each entry and is stored with the entry

6.7 Record Types

- A **record** is an aggregate of data elements in which the individual elements are identified by names and accessed through offsets from the beginning of the structure
- In C, C++, and C#, records are supported with the `struct` data type

Name
age
address

```
struct Student_PersonalData {
    char name[4];
    int age;
    char address[30];
} SP_Data;
```

```
#include <stdio.h>
#include <string.h>
void main() {
    struct Student_Personal_Data {
        char name[10];
        int age;
        char address[50];
    }  stu;
    strcpy(stu.name,“My name");
    stu.age = 35;
    strcpy(stu.address, “Dept. CSIE, NTNU");
    printf("The student's name is: %s\n", stu.name);
    printf("The student's age is: %d\n", stu.age);
    printf("The student's address is: %s\n", stu.address);
}
```

6.7 Record Types

- Design issues:
  - What is the syntactic form of references to the field?
  - Are elliptical references allowed

6.7.1 Definitions of Records

- Record **elements**, or **fields**, are not referenced by indices.
  - Fields are named with identifiers, and references to the fields are made using these identifiers

6.7.1 Definitions of Records

- 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.
```

6.7.1 Definitions of Records

- Record structures are indicated in an orthogonal way

```
type Employee_Name_Type is record
        First: String (1..20);
        Mid: String (1..10);
        Last: String (1..20);
end record;
type Employee_Record_Type is record
        Employee_Name: Employee_Name_Type ;
        Hourly_Rate: Float;
End record;
Employee_Record: Employee_Record_Type;
```

6.7.2 References to Record Fields

- COBOL field references have the form:

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

- Most of the other languages use **dot notation**

  `record_name_1.record_name_2. ... record_name_n.field_name`

6.7.2 References to Record Fields

- A **fully qualified reference** to a record field is one in which all intermediate record names, from the largest enclosing record to the specific field, are named in the reference.
- Elliptical reference (Pascal as an example)

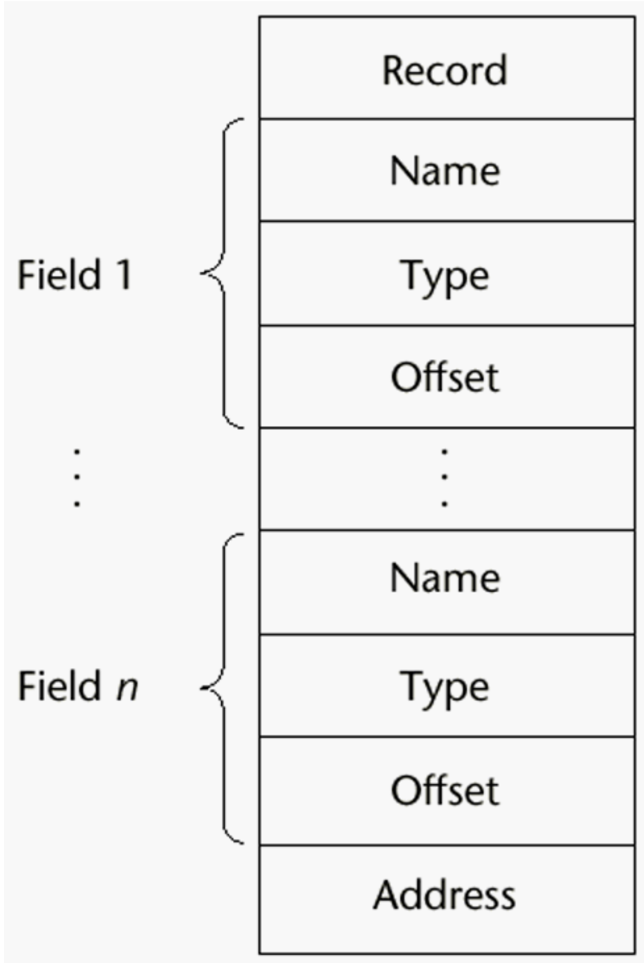
```
employee.name="bob";
employee.age=42;

with employee do
  begin
    name="Bob";
    age=42;
  end
```

6.7.3 Evaluation

- Field names are like literal, or constant, subscripts
  - Because they are static, they provide very efficient access to the fields
  - Dynamic subscripts could be used to access record fields, but it would disallow type checking and would also be slower

6.7.4 Implementation of Record Types

- The fields of records are stored in adjacent memory locations
  - Offset address, relative to the beginning of the record, is associated with each field
  - Field accesses are all handled using these field



6.8 Tuple Types

- A tuple is a data type that is similar to a record, except that the elements are not named
- Used in Python, ML, and F# to allow functions to return multiple values
  - Python
    - Closely related to its lists, but immutable
    - Create with a tuple literal

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

      Referenced with subscripts (begin at `1`)

    Catenation with + and deleted with **`del`**

6.8 Tuple Types (Cont'd)

- ML

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

- Access as follows:

`#1(myTuple)` is the first element

- A new tuple type can be defined

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

- F#

```
let tup = (3, 5, 7)
```

`let a, b, c = tup` This assigns a tuple to a tuple pattern `(a, b, c)`

6.9 List Types

- 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"]
```

6.9 List Types (Cont'd)

- Python Lists (continued)
  - List elements are referenced with subscripting, with indices beginning at zero

    `x = myList[1]` Sets `x` to `5.8`
  - List elements can be deleted with `del`

    ```
    del myList[1]
    ```
  - List Comprehensions – derived from set notation

    ```
    [x * x for x in range(6) if x % 3 == 0]
    ```

    `range(6)` creates `[0, 1, 2, 3, 4, 5, 6]`

    Constructed list: `[0, 9, 36]`

6.10 Union Types

- A **union** is a type whose variables may store different type values at different times during program execution

```
union customer
{
   char person[30];
   char company[30];
 };
 union customer c1;

 struct Data {
        union customer myCustomer;
        char address[50];
 };
```

6.10.1 Design Issues

- The problem of type checking union types

6.10.2 Discriminated Versus Free Unions

- C and C++ provide union constructs in which there is no language support for type checking
  - Free union

```
union flexType {
  int intE1;
  float floatE1; };
union flexType el1;
float x;
…
el1.intE1=27;
x=el1.floatE1;   //non-sense
```

6.10.2 Discriminated Versus Free Unions

- Type checking of unions requires that each union construct include a type indicator
  - Tag, discriminant
  - Discriminated union
    - ALGOL 68, Ada, ML, Haskell, F#

6.10.3 Unions in F#

- Defined with a type statement using OR (|)

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

`intReal` is the new type

`IntValue` and `RealValue` are constructors

To create a value of type `intReal`:

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

6.10.3 Unions in F# (Cont'd)

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

`match` pattern **`with`**

| expression_list_1 -> expression_1

| …

| expression_list_n -> expression_n

- Pattern can be any data type
- The expression list can have wild cards (_)

6.10.3 Unions in F# (Cont'd)

Example:

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

6.10.3 Unions in F# (Cont'd)

To display the type of the `intReal` union:

```
let printType value =
    match value with
        | IntVale value -> printfn "int"
        | RealValue value -> printfn "float";;
```

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

`printType ir1` returns `int`

`printType ir2` returns `float`

6.10.5 Evaluation

- Unions are potentially unsafe constructs in some languages
  - Thus, C and C++ are not strongly typed
- Neither Java nor C# includes unions

6.11 Pointer and Reference Types

- A **pointer** type is one in which the variables have a range of values that consists of memory addresses and a special value, **nil**.
- Two distinct kinds of uses:
  - Indirect addressing
  - Manage dynamic storage
    - Heap
      - Dynamic variables
      - Anonymous variables

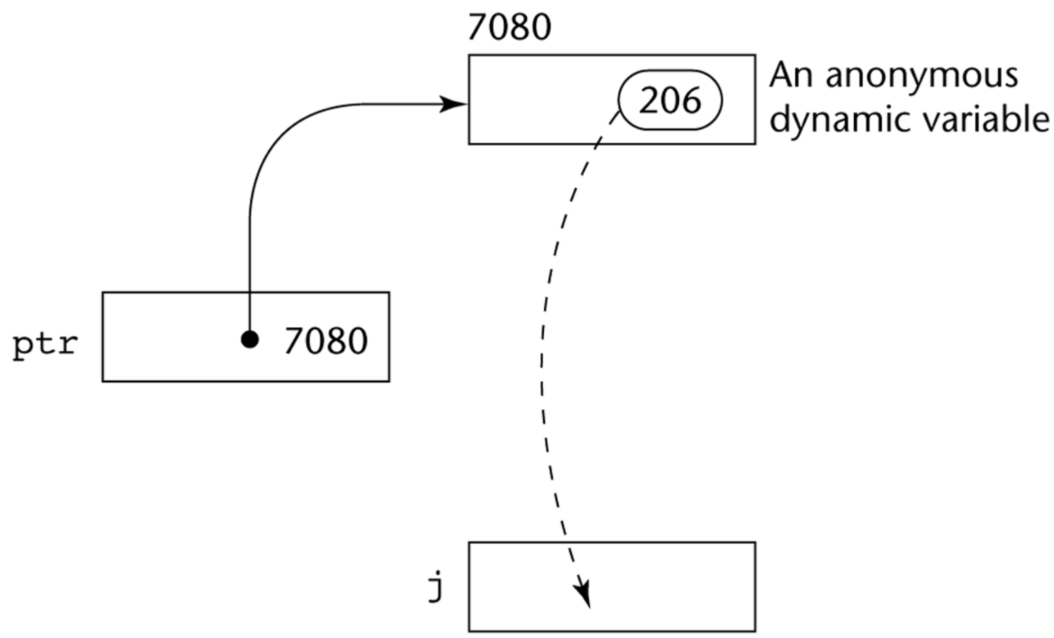
6.11.1 Design Issues

- 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

- Two fundamental pointer operations
  - Assignment
  - Dereferencing
    - Takes a reference through one level of indirection
    - Can be either explicit or implicit

      `j=*ptr`



6.11.2 Pointer Operations

- In C and C++, there are two ways a pointer to a record can be used to reference a field in that record

```
i=(*p).age;
i=P->age;
```

6.11.2 Pointer Operations

- Management of heap must include an explicit allocation operation
  - In C,
    - `malloc`, `free`
  - In C++,
    - `new`, `delete`

6.11.3 Pointer Problems

- The use of pointer could lead to several kinds of programming errors
  - Some recent languages, such as Java, have replaced pointers completely with reference types
    - Implicit deallocation (Automatic garbage collections)
    - A pointer with restricted operations

6.11.3.1 Dangling Pointers

- A dangling pointer, or dangling reference, is a pointer that contains the address of a heap-dynamic variable that has been deallocation

6.11.3.1 Dangling Pointers (Cont'd)

- The following sequence of operation creates a dangling pointer
  - (1) A new heap-dynamic variable is created and pointer `p1` is set to point at it
  - (2) Pointer `p2` is assigned `p1`'s value
  - (3) Variable pointed by `p1` is explicitly deallocated

  ⇒`p2` is now a dangling pointer

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

6.11.3.2 Lost Heap-Dynamic Variables

- A lost heap-dynamic variable is an allocated heap-dynamic variable that is no longer accessible to the user program
  - Also called **garbage**

6.11.3.2 Lost Heap-Dynamic Variables

- The following sequence of operation creates a lost heap-dynamic variables
  - (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

```
int * p1;
p1=new int[100];
p1=new int[200];
```

6.11.4 Pointers in C and C++

- The design offers no solutions to the dangling pointer or lost heap-dynamic variable problems
- Pointers in C and C++ can point to functions

```
int addInt(int n, int m) {
    return n+m;
}

main()
{
...

int (*functionPtr)(int,int);

functionPtr = &addInt;
int sum = (*functionPtr)(2, 3);
...
```

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.

6.11.5 Reference Types

- C++ includes a special kind of reference type that is used primarily for the formal parameters in function definitions.

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

6.11.5 Reference Types

```
Swap(int *a, int *b)   //using pointer
{ int t;
  t=*a; *a=*b;*b=t;}

Swap (int &x, int &y) //using reference
{ int t;
  t=x;x=y;y=t;}
```

6.11.5 Reference Types

- Pointer as a parameters require explicit dereferencing, making the code less readable and less safe.
- Reference parameters are referenced in the called function exactly as are other parameters.

6.11.5 Reference Types

- The designers of Java removed C++ style pointers altogether.
  - All Java class in stances are referenced by reference variables
    - The only use of reference variables in Java

```
String str1;
…
str1="This is a book";
```

6.11.5 Reference Types

- Because Java class instance are implicitly deallocated, there cannot be dangling references in Java.

6.11.6 Evaluatin

- Pointers have been compared with the "**goto**."
- Pointers are essential in some kinds of programming applications
  - Writing device driver

6.11.7 Implementation of Pointer and Reference Types

- In most languages, pointers are used in heap management
  - The same is true for Java and C# reference,
  - As well as variables in Smalltalk and Ruby

6.11.8.1 Representations of Pointers and References

- Pointers and References are single values stored in memory cells.

6.11.8.2 Solutions to Dangling-Pointer Problem

- There have been several proposed solutions to dangling-pointer problem
- Tombstones
  - Actual pointer variable pointers only at tombstones
  - When a heap-dynamic variable is deallocated the tombstone remains but is set to nil.

6.11.8.2 Solutions to Dangling-Pointer Problem

- Locks-and-keys approach
  - Used in UW-Pascal
  - Pointer values => (key, address)
  - When a heap-dynamic variable is allocated, a lock value is created and placed both in the lock cell of the variable and in the key cell of the pointer
  - Every access to the dereferenced pointer compares the key value and the lock value

6.12 Optional Type

- Some newer languages provide types that can have a normal value or a special value to indicate that their variable have no value.
  - C#: type name with a question mark (?)
    - int? x;
    - It can be tested against `null`
  - Swift: `nil`

6.13 Type Checking

- For the discussion of type checking, the concept of operands and operators is generalized to include subprograms and assignment statements.
- **Type checking** is the activity of ensuring that the operands of an operator are of compatible types

6.13 Type Checking (Cont'd)

- A **compatible type** is one that either
  - is legal for the operator, or
  - is allowed under language rules to be implicitly converted by compiler-generated code to a legal type
- **Coercion**
  - Automatic conversion

6.13 Type Checking (Cont'd)

- A **type error** is the application of an operator to an operand of an inappropriate type
- Static and dynamic type checking

6.14 Strong Typing

- One of the ideas in language design that became prominent in the so-called structured-programming revolution of the 1970s was
  - **strong typing**
    - A highly valuable language characterstics
    - Only loosely defined

6.14 Strong Typing (Cont'd)

- A programming language is **strongly typed** if type errors are always detected
  - Static time or run time detection
  - C and C++ are not strongly typed because "`union`" types.

6.14 Strong Typing (Cont’d)

- Java and C#,
  - Types can be explicitly cast, which could result in a type error
  - See next slice

```
//X is a supper class of Y and Z which are sibblings.
public class RunTimeCastDemo {
  public static void main(String args[]) {
    X x = new X();
    Y y = new Y();
    Z z = new Z();
    X xy = new Y(); // compiles ok (up the hierarchy)
    X xz = new Z(); // compiles ok (up the hierarchy)
    // Y yz = new Z();   incompatible type (siblings)
    // Y y1 = new X();   X is not a Y
    // Z z1 = new X();   X is not a Z
    X x1 = y; // compiles ok (y is subclass of X)
    X x2 = z; // compiles ok (z is subclass of X)
    Y y1 = (Y) x; // compiles ok but produces runtime error
    Z z1 = (Z) x; // compiles ok but produces runtime error
    Y y2 = (Y) x1; // compiles and runs ok (x1 is type Y)
    Z z2 = (Z) x2; // compiles and runs ok (x2 is type Z)

    Object o = z;
    Object o1 = (Y) o; //compiles ok but produces runtime error }
}
```

Type Equivalence

- Type compatibility
  - The type of an operand can be implicitly converted by the compiler or run-time system to make it acceptable to the operator
- Structure types are complex to make type compatible
  - Coercion is rare
  - The issue is not type compatibility, but type equivalence

Type Equivalence (Cont'd)

- Two types are *equivalent* if an operand of one type in an expression is substituted from one of the other type without coercion
  - Without coercion
- There are two approaches to defining type equivalence
  - Name type equivalence
  - Structure type equivalence

Type Equivalence (Cont’d)

- Name type equivalence is easy to implement but is more restrictive.
  - Defined either in the same declaration or in declarations that use the same type name

```
typedef int fahrenheit;
typedef int celsius;
fahrenheit f;
celsius c;
c = f; // type error in C++
```

    - A variable whose type is a subrange of the integers would not be equivalent to an integer type variable.

Type Equivalence (Cont’d)

- Structure type equivalence is more flexible than name type equivalence
  - Difficult to implement
  - Entire structures of two types must be compared
  - Disallow differentiating between types with the same structure

```
type Celsius = Float;
     Fahrenheit = Float;
```

Type Equivalence (Cont’d)

- Ada uses a restrictive form of name type equivalence but provides two type constructs for avoiding the problems associated with name type equivalence,
  - Subtypes and derived type
  - A derived type is a new type which it is not equivalent, although it may have identical structure

```
type Celsius is new Float;
     Fahrenheit is new Float;
```

Type Equivalence (Cont'd)

- An Ada subtype is a possibly range-constrained version of an existing type
  - A subtype is type equivalent with its parent type

```
// Compatible
subtype Small_type is Integer range 0..99

// Not compatible
type Derived_Small_type is Integer range 0..99
```

Type Equivalence (Cont'd)

- For variable of an Ada *unconstrained* array type, structure type equivalence is used

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

Type Equivalence (Cont'd)

- For constrained anonymous types, Ada uses a highly restrictive form of name type equivalence.

```
// A and B would be of anonymous but distinct
and not equivalent types
A : array (1:10) of Integer;
B : array (1:10) of Integer;

// C and D would be of anonymous but distinct and
// not equivalent types
C, D : array (1:10) of Integer;

// F and G would be equivalent
type list_10 is array (1:10) of Integer;
F, G: List_10;
```

Type Equivalence (Cont'd)

- C uses both name and structure type equivalence
  - Every `struct`, `enum`, and `union` declaration creates a new type that is not equivalent to any other type
  - Other nonscalar types use structure type equivalence
    - Array
  - Any type defined with `typedef` is type equivalent to its parent type

Type Equivalence (Cont’d)

- Object-oriented languages such as Java and C++ bring another kind of type compatibility issue with them