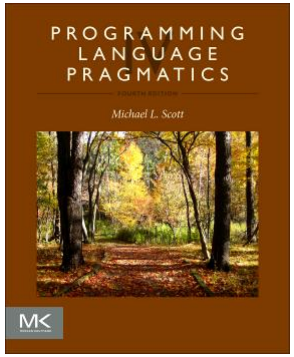
Chapter 8 :: Composite Types

Programming Language Pragmatics, Fourth Edition

Michael L. Scott



Copyright © 2016 Elsevier

ELSEVIER

Composite Types

- Formed by joining together one or more simpler types using a *type constructor*
- Some composite types:
  - Records
  - Arrays
  - Strings
  - Sets
  - Pointers
  - Lists
  - Files

ELSEVIER

Records (Structures)

- Allow related data of heterogeneous types to be stored and manipulated together
- Originally introduced by Cobol
- Also appeared in Algol 68, which called them *structures*, and introduced the keyword *struct*
- Many modern languages, including C and its descendants, employ the Algol terminology
- C++ - *structures*
- Java – *class*

ELSEVIER

Records (Structures)

- ```
  struct element {
  char name[2];
  int atomic_number;
  double atomic_weight;
  bool metallic;
  }
  ```

ELSEVIER
  ```

Variant Records (Unions)

- Programming languages of the 1960s and 1970s were designed in an era of severe memory constraints
- Many allowed the programmer to specify that certain variables (presumably ones that would never be used at the same time) should be allocated “on top of” one another, sharing the same bytes in memory

ELSEVIER

Variant Records (Unions)

- A variant record provides two or more alternative fields or collections of fields, only one of which is valid at any given time.

- ```
  union {
      int i;
      double d;
      _Bool b;
  };
  ```

ELSEVIER
  ```

Variant Records (Unions)

```
type tag = (is_int, is_real, is_bool);
var test: record
 case which: tag of
    is_int : (i:integer);
    is_real: (r:real);
    is_bool: (b:Boolean);
end;
----
test.which : = is_real;
test.r := 3.0;
writeln(test.r);

Output: 3.0
```

ELSEVIER

Variant Records (Unions)

```
type tag = (is_int, is_real, is_bool);
var test: record
 case which: tag of
    is_int : (i:integer);
    is_real: (r:real);
    is_bool: (b:Boolean);
end;
----
test.which : = is_real;
test.r := 3.0;
writeln(test.i);

Dynamic semantic error!
```

ELSEVIER

Variant Records (Unions)

```
type tag = (is_int, is_real, is_bool);
var test: record
 case which: tag of
    is_int : (i:integer);
    is_real: (r:real);
    is_bool: (b:Boolean);
end;
----
test.which : = is_real;
test.r := 3.0;
test.which:= is_int;
writeln(test.i);
```

Not an error, but the output will be junk!

ELSEVIER

Variant Records (Unions)

```
union Test {
    int i;
    float r;
    bool b;
};

int main() {
    Test t;
    t.r = 3.0;
    t.i = 5;
    printf("%f", t.r);
    return 0;
}
```

Not an error, but the output will be junk!

ELSEVIER

Arrays

- Arrays are the most common and important composite data types
- Unlike records, which group related fields of disparate types, arrays are usually homogeneous
- Semantically, they can be thought of as a mapping from an *index type* to a *component* or *element type*
- A *slice* or *section* is a rectangular portion of an array (See figure 8.5)

ELSEVIER

Arrays

- Array slices in Python:
  - [start:end] or [start:end:step]
  - `arr = np.array ([0, 1, 2, 3, 4, 5, 6, 7])`
    - `print(arr[0:6])` - `[0, 1, 2, 3, 4, 5]`
    - `print(arr[4:])` - `[4, 5, 6, 7]`
    - `print(arr[:3])` - `[0, 1, 2]`
    - `print(arr[0:7:2])` - `[0, 2, 4, 6]`

ELSEVIER

Arrays

```
my_array = [[1, 2], [3, 4]]

print(my_array[0][0])     # 1
print(my_array[0][1])     # 2
print(my_array[1][0])     # 3
print(my_array[1][1])     # 4
```

ELSEVIER

Arrays

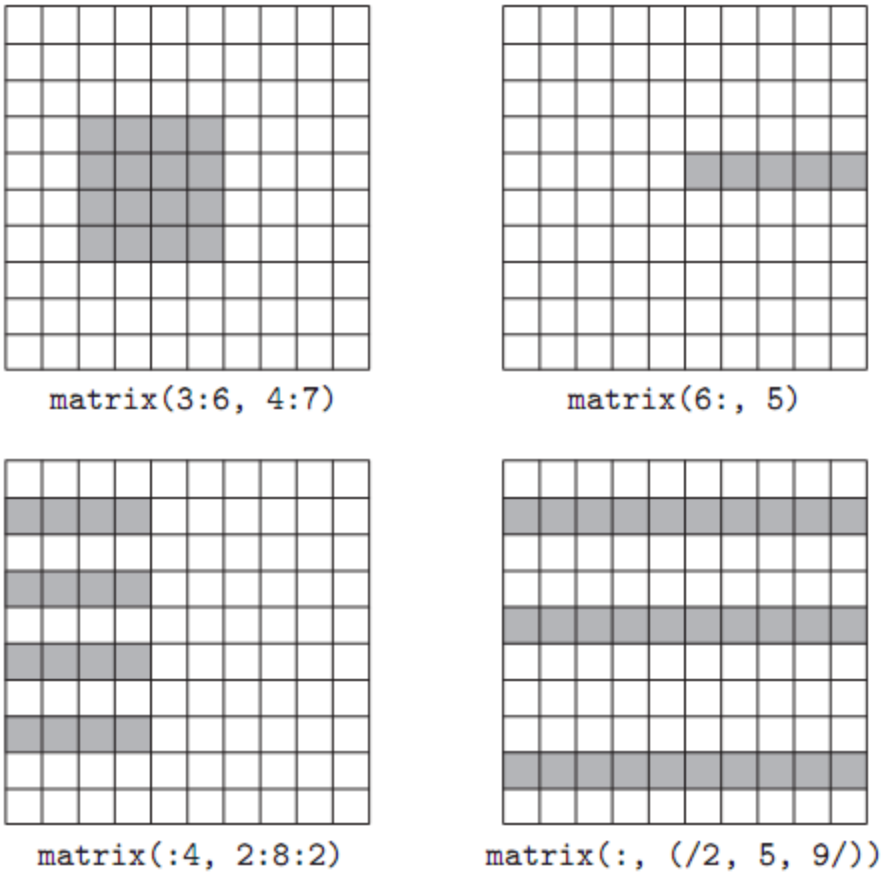


Figure 8.5 Array slices(sections) in Fortran90. Much like the values in the header of an enumeration-controlled loop (Section6.5.1), a: b: c in a subscript indicates positions a, a+c, a+2c, ...through b. If a or b is omitted, the corresponding bound of the array is assumed. If c is omitted, 1 is assumed. It is even possible to use negative values of c in order to select positions in reverse order. The slashes in the second subscript of the lower right example delimit an explicit list of positions.

ELSEVIER

Arrays

- Array slices(sections) in Fortran90:
  - matrix(3:6, 4:7) columns 3-6, rows 4-7
  - matrix(6:, 5) columns 6-end, row 5
  - matrix(:4, 2:8:2) columns 1-4, every other row from 2-8
  - matrix(:, /2, 5, 9/) all columns, rows 2, 5, and 9

ELSEVIER

Arrays

- Contiguous elements (see Figure 8.8)
  - column major - only in Fortran
  - row major
    - used by everybody else
    - makes array [a..b, c..d] the same as array [a..b] of array [c..d]

ELSEVIER

Arrays

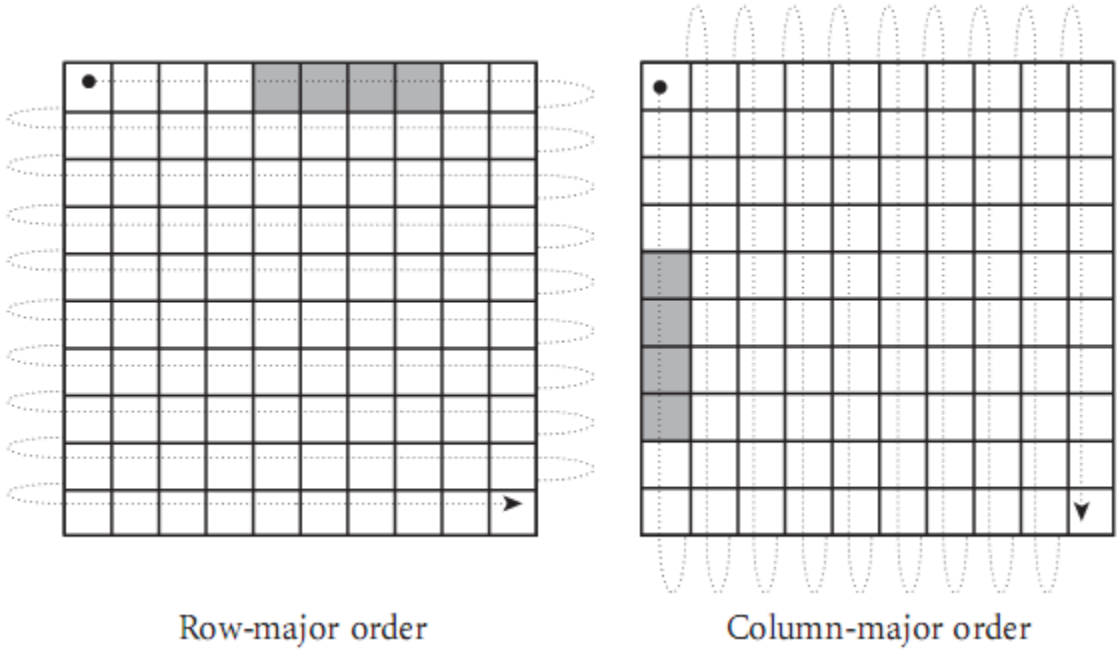


Figure 8.8 Row- and column-major memory layout for two-dimensional arrays. In row-major order, the elements of a row are contiguous in memory; in column-major order, the elements of a column are contiguous. The second cache line of each array is shaded, on the assumption that each element is an eight-byte floating-point number, that cache lines are 32 bytes long (a common size), and that the array begins at a cache line boundary. If the array is indexed from A[0,0] to A[9,9], then in the row-major case elements A[0,4] through A[0,7] share a cache line; in the column-major case elements A[4,0] through A[7,0] share a cache line.

ELSEVIER

Arrays

- Two layout strategies for arrays (Figure 8.9):
  - Contiguous elements
  - Row pointers

ELSEVIER

Arrays

```
char days[][10] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

```
char *days[] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

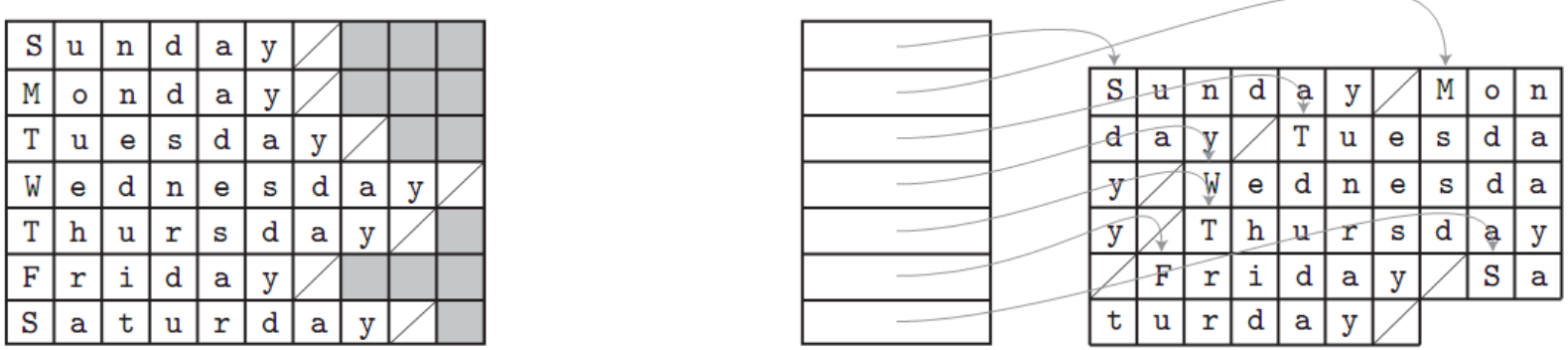


Figure 8.9 Contiguous array allocation v. row pointers in C. The declaration on the left is a true two-dimensional array. The slashed boxes are NUL bytes; the shaded areas are holes. The declaration on the right is a ragged array of pointers to arrays of character s. In both cases, we have omitted bounds in the declaration that can be deduced from the size of the initializer (aggregate). Both data structures permit individual characters to be accessed using double subscripts, but the memory layout (and corresponding address arithmetic) is quite different.

ELSEVIER

Strings

- In some languages, strings are really just arrays of characters
- In others, they are often special-cased, to give them flexibility (like dynamic sizing) that is not available for arrays in general

ELSEVIER

Strings

```
my_string = "hello"
print("id of my_string = ", id(my_string))
my_string += " world"
print(my_string)
print("id of my_string = ", id(my_string))
```

ELSEVIER

Strings

```
my_string_1 = "hello"
print("id of my_string_1 = ", id(my_string_1))
my_string_2 = "hello"
print("id of my_string_2 = ", id(my_string_2))
```

ELSEVIER