Programming Language Concepts

Data Types

Janyl Jumadinova
14-16 February, 2017
Arrays

- One of the first composite data types and still used very often.
- Fixed-size collection of values, all of the same type, accessed by one or more indices: sometimes called “subscripts” in analogy to mathematical notation such as $x_0, x_1, x_2, \ldots$ or $a_{0,0}, a_{0,1}, \ldots, a_{5,4}, a_{5,5}$. 

The program `array.cpp` (in the repo):

```cpp
```
Arrays

- One of the first composite data types and still used very often.
- Fixed-size collection of values, all of the same type, accessed by one or more indices:
  sometimes called “subscripts” in analogy to mathematical notation such as \( x_0, x_1, x_2, \ldots \) or \( a_{0,0}, a_{0,1}, \ldots, a_{5,4}, a_{5,5} \).

The program array.cpp (in the repo):

two different ways to represent arrays in C++: the “traditional” C array and the “array class” in C++
Arrays

- **Index ranges**: often zero-based for ease of calculation.
Arrays

- **Index ranges**: often zero-based for ease of calculation.
- Older languages (e.g., older versions of Fortran): 1-based (`a[1],...,a[4]`).
- Some languages allow other ranges (see Fortran 95 program `array2.for` in the repo.)
Arrays

- Usually stored in consecutive memory locations.

```c
int a[10];
a[0] = 99;
a[1] = -12;
a[2] = 42;
```

one int = 4 bytes, so 40 bytes.
Arrays

- Usually stored in consecutive memory locations.
- Example:

```c
int a[10];
a[0] = 99;
a[1] = -12;
a[2] = 42;
```

one int = 4 bytes, so 40 bytes.
To find an int array element, take its base location plus its index times 4:

- $a[2]$ is in location:
  $$10000 + 2 \times (\text{sizeof int}) = 10000 + 2 \times 4 = 10008$$
What about two-dimensional arrays?

```cpp
char c[3][4]
```
What about two-dimensional arrays?

```c
char c[3][4]
```

one char = 1 byte, so 12 bytes.
What about two-dimensional arrays?

```cpp
char c[3][4]
```

one char $= 1$ byte, so 12 bytes.

*We have a choice!*
Arrays

ROW MAJOR ORDER

- char c[3][4]
  c[1][2] = base address
  + (1* (# cols) + 2)*sizeof char
  = 10000 + (1*4+2)*1 = 10006
COLUMNS MAJOR ORDER

- char c[3][4]
  
  c[1][2] = base address
  + (2*(#rows)+1)*sizeof char
  = 10000+(6+1)*1 = 10007
Java, C, and many other languages use row major order. (See program array2.cpp in the repo.)
Arrays

- Java, C, and many other languages use row major order. (See program `array2.cpp` in the repo.)
- Fortran uses column major order. (See program `array.for` in the repo.)
Arrays

- Java, C, and many other languages use row major order. (See program `array2.cpp` in the repo.)
- Fortran uses column major order. (See program `array.for` in the repo.)
- Does it matter?
Efficiency Concerns

In a language that uses row-major order, it is more efficient to access elements by rows because they are in consecutive locations; accessing by columns involves a lot of jumping around in memory.
Efficiency Concerns

In a language that uses row-major order, it is more efficient to access elements by rows because they are in consecutive locations; accessing by columns involves a lot of jumping around in memory.

See programs `byrows.c` and `bycols.c`; `byrows.for` and `bycols.for`; `ByCols.java` and `ByRows.java` in the class repo.

Run experiments - each program 3 runs, record your time results in the Google Form given through Slack.
Efficiency Concerns

Experiments show that in C (which uses row major order), the loop on the left is faster than the loop on the right.

\[
\begin{align*}
\text{for } (r=0; r<1000; r++) \\
\text{for } (c=0; c<1000; c++) \\
\text{x[r][c] = ...;}
\end{align*}
\]

\[
\begin{align*}
\text{for } (c=0; c<1000; c++) \\
\text{for } (r=0; r<1000; r++) \\
\text{x[r][c] = ...;}
\end{align*}
\]

FASTER  SLOWER
Efficiency Concerns

However, in Fortran, the opposite holds:

```
do, r = 1,1000
  do, c = 1,1000
    a(r,c) = ...
  enddo
enddo
```

SLOWER

```
do, c = 1,1000
  do, r = 1,1000
    a(r,c) = ...
  enddo
enddo
```

FASTER
What about `int a[3][4][2][5];`?

Row-major: `a[0][0][0][0][0],
          a[0][0][0][0][1],
          a[0][0][0][0][2], ...

Column-major: `a[0][0][0][0][0],
               a[1][0][0][0][0],
               a[2][0][0][0][0], ...`
Exercise

On the Google Form sent through Slack:

**Row major ordering**

Come up with a formula for the address of \( a[i][j][k][l] \), assuming zero-based indexing, assuming that each array element is 4 bytes and that row-major ordering is used, and, of course, assuming that \( i, j, k, \) and \( l \) are within the array bounds.

**Column major ordering**

Same, but for column-major ordering.
Exercise

On the Google Form sent through Slack:

**Row major ordering**

Come up with a formula for the address of $a[i][j][k][l]$, assuming zero-based indexing, assuming that each array element is 4 bytes and that row-major ordering is used, and, of course, assuming that $i$, $j$, $k$, and $l$ are within the array bounds.

**Column major ordering**

Same, but for column-major ordering.

See exercise.c for one solution.
Other Aspects of Arrays - Bounds

- If arrays are just consecutive locations in memory, how does the computer know whether, e.g., an array is 3 by 4 or 4 by 3 or 2 by 6 or ...?
Other Aspects of Arrays - Bounds

- If arrays are just consecutive locations in memory, how does the computer know whether, e.g., an array is 3 by 4 or 4 by 3 or 2 by 6 or ...?
- It needs to remember this information somewhere—symbol table (static arrays) or “dope vector” (dynamic arrays)—see page 331.
- Checking for out-of-bounds indices at runtime affects security (see sidebar, p. 353).
In C, array size is known at compile time unless we allocate memory from the heap and use pointers to access it.
In C, array size is known at compile time unless we allocate memory from the heap and use pointers to access it.

See program array6.c in the repo for an example using “malloc”.
We can exploit the linear storage of arrays to view the array contents as if it has different “shapes”.

See programs `array3.c` and `array4.c` in the repo.

NOTE: In C we can’t actually change the declared shape, but we can “get around” this using pointers (see section 7.7).
Arrays of Arrays

Consider the C or Java declaration:

```java
int a[3][4];
```

- This can be considered as an “array of arrays”:
- an array `a[3]` whose elements are of type “int [4]”.
- Once we allow this, we can allow more general “arrays of arrays”, e.g., `Array5.java`
Sets

- “Set”-like classes in Java; e.g., program Sets.java shows the HashSet class.
“Set”-like classes in Java; e.g., program Sets.java shows the HashSet class.

Some language have a built-in set data type.

Python:

```python
>>> s = {1,2,4,5,1,2,3,5,3,2,1}
>>> s
set([1, 2, 3, 4, 5])
```
It is relatively easy to implement “small sets” in a programming language (usually max of 32 or 64 elements).

See, e.g., program set.c in the repository.

The idea is to use the bits in an int or long to stand for elements, then use bit operations | and & for union and intersection.
Pointers

- We’ve already seen examples of pointers in C.
- Since C is often used by systems programmers, it is convenient to be able to directly access and manipulate memory addresses.
- However, care must be taken. How many of you have ever seen the words: segmentation fault