Arrays

Chapter 8

Simple structured data
- An array is an ordered collection of variables, each of the same type, referenced by one name and a subscript
- Subscripts or indexes range from 0..size-1
- C does not check array bounds
- Array definitions include the data type and array size
- Bracketed values are dimensional sizes
- Each dimension is individually bracketed
- Each dimension must be compile-time constant
- Name of the array, without a subscript, is a constant address (the address of the first element)

One-dimensional Array

```c
int test[10]; /* space for 10 ints */
```

Two-dimensional Array

```c
float test_score[10][4]; /* space for 40 floats */
```

Three-dimensional Array

```c
double class_score[5][5][5]; /* space for 125 doubles */
```

Array Syntax

- Arrays are often used with for-loops
- One loop per dimension
- Index goes from 0 to size - 1 (i.e., < size)
- Examples
  ```c
  for (i = 0; i < 10; i++)
  cout << test[i] << endl;
  ```
  ```c
  for (j = 0; j < 4; j++)
  for (i = 0; i < 10; i++)
  cout << test_score[i][j] << endl;
  ```
**Static Array Initialization**

Compile-time initialization

```c
int month_length[12] = 
{ 31, 28, 31, 30, 
31, 30, 31, 31, 
30, 31, 30, 31 
};
```

```c
int test_scores[5] = 
{ 95, 98, 97, 96 }, 
{ 79, 89, 79, 85 }, 
{ 99, 98, 99, 99 }, 
{ 90, 89, 83, 86 }, 
{ 75, 72, 79, 69 };
```

**Dynamic Array Sizing**

Compiler counts the number of elements

- **General format**
  - `type name[i] = {...};` /* do static initialization */
  - `int number = sizeof(name) / sizeof(type);`
    - `sizeof(name)` is the total number of bytes occupied by the array
    - `sizeof(type)` is the number of bytes in type
    - the quotient is the number of elements in the array

**Example**

```c
int month_length[] = 
{ 31, 28, 31, 30, 
31, 30, 31, 31, 
30, 31, 30, 31 
};
```

```c
int number = sizeof(month_length)/sizeof(int);
```

**Example -- Matrix Addition**

Arrays on either side of the assignment operator

\[
A = \begin{pmatrix}
  a_{0,0} & a_{0,1} \\
  a_{1,0} & a_{1,1} \\
  \vdots & \vdots \\
  a_{m-1,0} & a_{m-1,1}
\end{pmatrix}
B = \begin{pmatrix}
  b_{0,0} & b_{0,1} & \cdots & b_{0,n-1} \\
  b_{1,0} & b_{1,1} & \cdots & b_{1,n-1} \\
  \vdots & \vdots & \ddots & \vdots \\
  b_{m-1,0} & b_{m-1,1} & \cdots & b_{m-1,n-1}
\end{pmatrix}
\]

\[
C = A + B \rightarrow c_{ij} = a_{ij} + b_{ij}
\]

```c
double a[m][n], b[m][n], c[m][n];
for (i = 0; i < m; i++)
  for (j = 0; j < n; j++)
    c[i][j] = a[i][j] + b[i][j];
```

**Example -- Matrix Multiplication**

Arrays and loops: frequent companions

\[
C = AB \rightarrow c_{ij} = \sum_{k=0}^{n-1} a_{ik}b_{kj}
\]

```c
double a[m][n], b[p][n], c[m][n];
for (i = 0; i < m; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < p; k++)
      c[i][j] += a[i][k] * b[k][j];
```

**Arrays and Memory Mapping**

Storing arrays in memory

- Multidimensional arrays are stored as single dimensional arrays in memory
- Two ways to store:
  - row major: stored by rows
  - column major: stored by columns
- C used row major
- `RMaddr = i * ncols + j`

```c
H is at RM address 7
```

**Variables: Address vs Content**

Address is intrinsic, address vs content

- A left value (aka l-value)
  - is a value that can be on the left hand side of =
  - is the address of a variable
- A right value (aka r-value)
  - is a value that can be on the right hand side of =
  - can be any valid expression (i.e., constant, variable, function call, or any of these joined with operators and grouped with parentheses)

```c
i = 10; // i is an l-value, 10 is an r-value
j = i;  // j is an l-value, i is an r-value
```

- r-value of i is 10 (i.e., the content of i)
- l-value of i is 0xffff000 (i.e., the address of i)
**Pointer and Address Operators**

- Pointers are variables that contain memory addresses
  - Pointer-type specifier (variable definitions & declarations)
  - `*` - same as below `*`
  - `&` - address of operator returns the address of variables
  - `*` dereference or indirection operator (expressions)
- Operator relationships
  - `p` is assignment compatible with `t` (i.e., `t = *p`)
  - `p` is assignment compatible with `p` (i.e., `p = &p`)
  - `*(&variable)` - variable
- Special pointers (that point to nothing)
  - `NULL`
  - `0` (zero)

**Address Operators**

- Memory content vs memory address
  - `int i;`  
    - `i` - `0x0a000010`  
    - `i` is undefined
  - `int* p = &i;`  
    - `p` - `0x0a000014`  
    - `p` is `0x0a000010`
  - `i = 10;`  
    - `i` - `0x0a000010`  
    - `i` is `10`
  - `*p` is now `10`  
    - `p` - `0x0a000010`  
    - `*p` is `0x0a000010`

**Arrays As Function Arguments**

- Arrays are always passed by address
- Arrays are the exception: always passed by address
- Function definitions
  - `void func1(int* student) { ... } /* most common */`
  - `void func1(int student[]) { ... }`  
  - `void func1(int student[10]) { ... } /* least common */`
  - `void func2(float test[]) { ... }`
  - `void func3(double test[4][4][5]) { ... }
- Function calls
  - `func1(test);`
  - `func2(test_score);`
  - `func3(class_score);`

**Pass By Address Illustrated**

- Efficient function calls
- `int test[10];`
- `func1(test);`
- `void func1(int* student) {
  student[6] = 95;
}
- `student is an alias for test`
- `func1(test);`

**Returning Arrays**

- Returning pointers (a common interview test)
  - `/* technique 1 - WRONG */`
  - `char* GetData() { char line[80];
    return line; }
  `
  - `/* technique 3 - okay */`
  - `char* GetData(char* line) {
    ... return line;
  }
  `
  - `/* technique 2 - okay */`
  - `char* GetData() { char* line = new char[80];
    ... return line; }
  `
  - `/* technique 4 - okay */`
  - `char* GetData() { static char* line[80];
    ... return line; }
  `

**Command-Line Arguments**

- Arrays of pointers
  - `int main(int argc, char* argv[])`  
    - `argdemo hello word from CS2250`
  - `argv`  
    - `5`
  - `for (i = 1; i < argc; i++)
    printf("%s\n", argv[i]);`
  - `argv[0]`  
    - `0`  
    - `a r g d e m o b`
  - `argv[1]`  
    - `1`  
    - `h e l l o 1 0`
  - `argv[2]`  
    - `2`  
    - `w o r l d 0`
  - `argv[3]`  
    - `3`  
    - `f r o m 0 b`
  - `argv[4]`  
    - `4`  
    - `C S 2 2 5 0 9`