Lecture 13.1

Topics

1. Data Storage and Initialization – Automatic, Global, Static
2. Array and Pointer Notations
3. Arrays and Function Returns – Examples

---

1. Data Storage and Initialization – Automatic, Global, Static

The initialization of an array and variable depends on how the array/variable was stored. There are three kinds of storage in C in which a variable's value can be stored under.

1.1 Automatic (Local) Storage

An automatic or local storage will be given to variables and arrays that were defined inside functions (including the formal arguments). These variables and arrays will be available only within the functions and they will no longer exist as soon as the functions are done with their execution.

Example

```
int main() {
  int iA1[ 3 ];
  int iA2[ 2 ] = { 1, 2 };

  return 0;
}
```

1.2 External (Global) Storage

External (global) storage is given to variables and arrays that were defined outside of any functions. Global variables and arrays are available everywhere after they are defined. They last through out the program execution. Global variables and arrays are initialized with **`zero(s)`** unless other specific values were given with the declarations.

Example

```
int common;
const int MAXSIZE = 5;
int iExtArray[ 3 ] = { 1, 2, 3 };

int main() {
  int iA1[ 3 ];
  int iA2[ 2 ] = { 1, 2 };

  return 0;
}
```

1.3 Static Storage

A declaration that was modified with the **`static`** keyword will force the system to provide a **static storage**. This static storage reserves a common block of memory for a static variable and its value will be shared by any invocation that would require access to this static member.

Static variables are defined inside a method and will have the values initialized to **zero(s)** unless other values were given.

Example

```
int test() {
  static int iCommonValue;
  int iATemp[ 2 ] = { 1, 2 };
  return 0;
}
```

2. Array & Pointer Notations

Recall the following interpretations of any array name:

(a) Array name is the address of the first byte of the allocated memory for the array.

(b) Array name is a constant variable.

That means

- `aryName` is the same as `&aryName[ 0 ]`
- `aryOne = aryTwo` is an error.

And, C arrays and pointers will have the following relationship,

`ary[ index ]` is equivalent to **`( *( ary + index ) )`**

This means that an array subscript **`ary[ index ]`** is the same as dereferencing **`ary`** pointer an offset by **`index`** from the beginning of the array. This relationship holds for any array data type.

This equivalent relationship comes handy to decipher pointer expressions. In any expression that contains **`ary[ 0 ]`**, one may substitute with **`*ary`**. Likewise, **`buf + index`** is equivalent to **`&buf[ index ]`** as follows,

```
ary[ 0 ] = *( ary + 0 ) = *ary
&buf[ index ] = &*( buf + index ) = buf + index
```

The following 2 programs are equivalent.

Fragment #1:

```
#define MAX_ARY_SIZE 15

int main( void ) {
  int i, j;
  int iTemp;
  int iSize;
  int iAry[ MAX_ARY_SIZE ] = { 89, 72,  3, 15, 21,
                               57, 61, 44, 19, 98,
                               5, 77, 39, 59, 61 };


  iSize = MAX_ARY_SIZE;
  for ( i = 0; i < iSize; i++ ) {
    for ( j = i + 1; j < iSize; j++ ) {
      if ( iAry[ i ] < iAry[ j ] ) {
        iTemp = iAry[ i ];
        iAry[ i ] = iAry[ j ];
        iAry[ j ] = iTemp;
      }
    }
  }

  return 0;
```

```
      }
```

Fragment #2:

```
      #define MAX_ARY_SIZE 15

      int main (void) {
        int i, j;
        int iTemp;
        int iSize;
        int iAry[ MAX_ARY_SIZE ] = { 89, 72,  3, 15, 21,
                                     57, 61, 44, 19, 98,
                                     5, 77, 39, 59, 61 };

        iSize = MAX_ARY_SIZE;
        for ( i = 0; i < iSize; i++ ) {
          for ( j = i + 1; j < iSize; j++ ) {
            if ( *( iAry + i ) < *( iAry + j ) ) {
              iTemp = *( iAry + i );
              *( iAry + i ) = *( iAry + j );
              *( iAry + j ) = iTemp;
            }
          }
        }

        return 0;
      }
```

3. Arrays and Function Returns – Examples

What would be wrong with the following functions?

Example

```
      int[] foo1( ) {
        int ary[ 2 ];

        ary[ 0 ] = 5;
        ary[ 1 ] = -9;

        return ary;
      }

      int* foo2( ) {
        int ary[ 2 ];

        ary[ 0 ] = 5;
        ary[ 1 ] = -9;

        return ary;
      }
```