C Basics 11: Multidimensional and String Arrays

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
8. Errors and Error Handling
9. Functions and Parameter Passing
10. Dynamic and Static Memory

C Basics 11: Multidimensional and String Arrays

11. **Multidimensional and String Arrays**
    - **multidimensional array definitions and access**
    - **arrays of arrays**
    - **linearization: row major order**
    - **arrays and pointers**
    - **subarrays and function parameters**
    - **string arrays: multidimensional vs. char* arrays**
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. Miscellaneous

Multidimensional Arrays

Arrays can be **multidimensional**:

```
int array_2d[10][20];
```

`array_2d` is a 10×20 array (10 *"rows"* with 20 *"columns"* each).

An multidimensional array element can be accessed as:

```
array_2d[1][19] = 20;
```

In C, "multidimensional arrays" are technically *arrays of arrays*.

`array_2d` is a *10-element array*, each element of which is a *20-element array* of `int`.

Multidimensional Arrays (contd.)

Multidimensional arrays are not limited to two dimensions:

```
int array_3d[10][20][30];
```

We would normally call this a $10 \times 20 \times 30$ array.

In C though, it is technically an 10-element array, each element of which is a 20-element array, each element of which is a 30-element array of `int`.

Linearization: Row Major Order

As with one-dimensional arrays, a multidimensional array is stored in a *block of sequential bytes*.

Array elements are *linearized* using **row major order**:

```
int a2d[3][5] = {{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15}};
```

defines an array with *three rows* each with *five columns*.

The fifteen array elements are stored linearly in memory as:

`a2d[0][0], a2d[0][1], a2d[0][2], a2d[0][3], a2d[0][4],`
`a2d[1][0], a2d[1][1], ..., a2d[1][4], ..., a2d[2][4]`

(Note: rightmost index/subscript changes fastest.)

Arrays and Pointers

As noted previously, there is a close connection between arrays and pointers in C.

An array identifier is effectively converted to a pointer to the first byte of the array memory block.

The C99 standard says:

- "`E1[E2]` is identical to `(*((E1)+(E2)))`."
- "Successive subscript operators designate an element of a multidimensional array object."
- "If `E` is an n-dimensional array...with dimensions $i \times j \times ... \times k$, then `E`...is converted to a pointer to an $(n-1)$-dimensional array with dimensions $j \times ... \times k$."

Arrays and Pointers (contd.)

Because of the array-pointer connection, there are numerous ways to get the value of element `i-j` in `a2d[3][5]`:

```
a2d[i][j]
*(a2d[i] + j)
*(&a2d[0][0] + 5*i + j)
(*(a + i))[j]
*((*(a + i))+j)
```

Arrays and Pointers (contd.)

The connection between multidimensional arrays and pointers can become fairly complicated to understand.

For example, the C99 standard has the following:
"Consider the array object defined by the declaration

```
int x[3][5];
```

Here `x` is a 3x5 array of ints; more precisely, `x` is an array of three element objects, each of which is an array of five ints. In the expression `x[i]`, which is equivalent to `(*((x)+(i)))`, `x` is first converted to a pointer to the initial array of five ints. Then `i` is adjusted according to the type of `x`, which conceptually entails multiplying `i` by the size of the object to which the pointer points, namely an array of five int objects. The results are added and indirection is applied to yield an array of five ints. When used in the expression `x[i][j]`, that array is in turn converted to a pointer to the first of the ints, so `x[i][j]` yields an int."

Subarrays

Because multidimensional arrays are truly arrays of arrays, it is possible to refer to the *subarrays* using partial subscripts.

Consider again the array `int a2d[3][5]`.

The *rows* in `a2d` can be denoted as:

`a2d[0]`, `a2d[1]`, `a2d[2]`

Each row is of type: 5-element array of `int`.

Arrays as Function Parameters

When using multidimensional arrays as *function parameters*, the *leftmost dimension* need *not* be specified in the definition:

```
void process_2d(int iarr_2d[][20])
void process_3d(int iarr_3d[][20][30])
```

Even without knowing that the size of the leftmost dimension, the appropriate offset can be computed for any array element:

`iarr_2d[i][j]`: `*(&iarr_2d[0][0] + 20*i + j)`

`iarr_3d[i][j][k]`: `*(&iarr_3d[0][0][0] + 20*30*i + 30*j + k)`

(Of course without knowing all dimensions it is not possible to tell whether an element is "out of bounds," but C does not check this anyway!)

Arrays of Strings

It is common to want to manipulate a set of related strings, and a typical way to store these strings is as an *array of strings*.

Since a string is an array of `char` in C, an array of strings is really an array, each of whose elements is an array of `char`:

```
char strarray[][]
```

An array of 10 strings of 20 characters each can be declared:

```
char strarr[10][21];
```

Such an array can be initialized using special notation:

```
char strarr[10][21] = {"first string", "second string",...};
```

It is possible to use a *single array index* to access each (sub)string:

```
strcpy(strarr[1],"test string");
```

Arrays of Strings (contd.)

Because a character array is effectively a pointer, another way to declare an array of strings is:

```
char *strarr[10];
```

An important difference between this declaration and the one above is that this one *does not allocate any space to actually store the strings*.

That would have to be done via something like:

```
for (int i=0; i<10; i++)
  strarray[i] = malloc(21);
```

This would be slower than using arrays.

On the other hand, it would allow the array to hold strings with very different lengths without having to allocate space for the longest possible string length for every string element.

Arrays of Strings (contd.)

We have already seen an array of strings with `main`, where `argv` is an array of the arguments as strings (array of `char*`):

```
main(int argc, char *argv[])
```

Because of the array-pointer connection, we can also do:

```
main(int argc, char **argv)
```

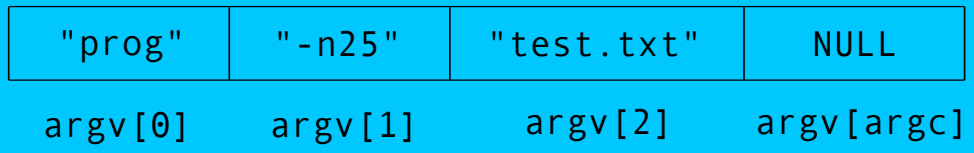
(`char**` is read *right to left*: `argv` is a pointer to `char*`, or array of `char*`, and `char*` is a pointer to `char`, or array of `char`.)

Using pointer arithmetic to print command line arguments:

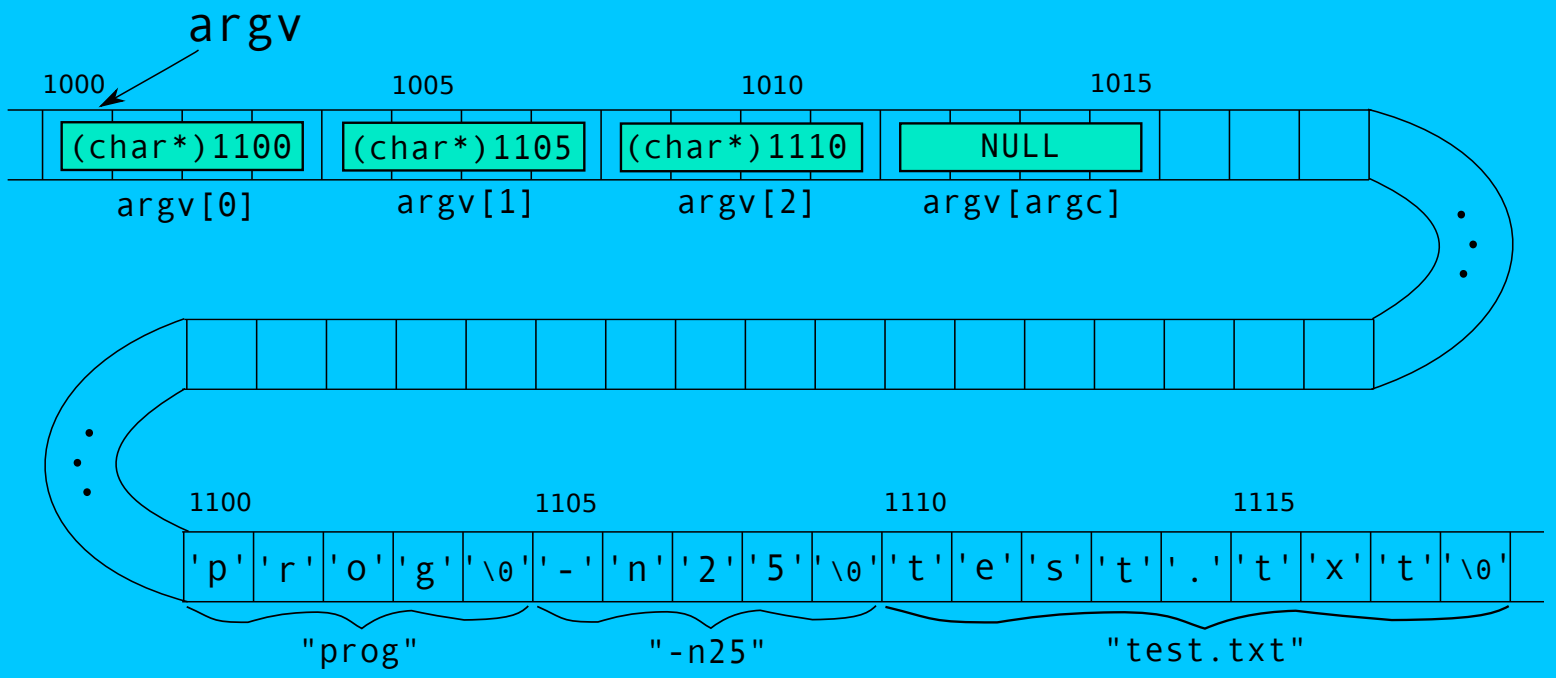
```
int main(int argc, char **argv)
{
  char **args = argv;
  while(*++args != NULL)
    printf("%s\n",*args);

  return EXIT_SUCCESS;
}
```

Arrays of Strings (contd.)

Earlier, we showed `argv` for the command "`prog -n25 test.txt`" as an array of strings:



But `argv` is actually an array of arrays of `char`:



Initializing Arrays of Strings

An array of strings can be initialized as:

```
const char *strarr[] = {
  "First entry",
  "Second entry",
  "Third entry",
};
```

Here `strarr[0]` is a pointer to the *string constant* `"First entry"`.

Note that while e.g. `strarr[0]` can be changed to point to a different `char*`, modifying the elements (`char`'s) of the string constants can cause problems, hence the `const` declaration (so compiler will catch).