Strings, multi-dimensional arrays

Lecture Topics

- Arrays as function parameters
- Strings
- Multi-dimensional arrays

Lecture materials

- Textbook Ch. 16

→ arrays as function parameters.

→ two options:

→ pass the entire array of values from calling function to called function:

int Average (int input_values[N]);

→ this will require copying all elements onto the stack of the invoked function Average.

→ for large N this is both time-consuming and wasteful memory-wise.

→ pass the reference to the array. C naturally passes arrays by reference.

int Average (int input_values[]);

↑ function declaration

↓

↑ indicates to the compiler that the corresponding parameter will be the base address of the array with elements of type int.

→ re-write previous example using a function.

→ call within main()

~~Average~~ average = Average (array);

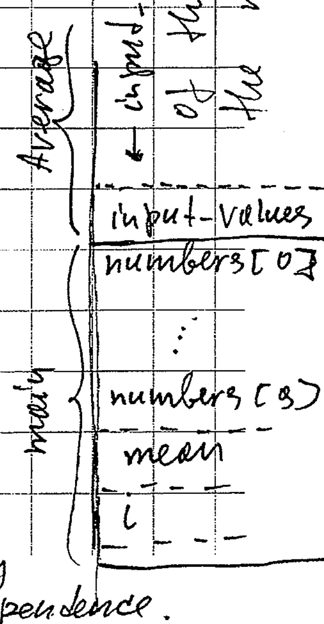
→ we use "array" as an argument to the function. Thus, ~~the~~ only the address of of the array "array" will be pushed onto the stack.

→ within the function, the parameter input_values will be assigned the address of the array.

→ also, in function Average we can use standard array notation [i] to access individual elements.

```c
#include <stdio.h>
#define N 10

int Average (int input_values[]);

int main() {
    int i, mean=0;
    int numbers[N];

    printf("Enter %d numbers.\n", N);
    for (i=0; i<N; i++) {
        printf("Enter number %d:", i);
        scanf("%d", &numbers[i]);
    }

    mean = Average(numbers);

    printf("Average = %d\n", mean);
}
int Average(int input_values[]) {
    int i;
    int sum = 0;

    for (i=0; i<N; i++) {
        sum += input_values[i];
    }
    return sum/N;
}
```

← input_values stores the address of the first element of array numbers in the main's ~~reference for~~ definition record.



→ this code example is not a very good practice because function Average depends on a globally defined value of N. Rewrite it to eliminate this dependence.

strings in C

- strings in C are sequences of chars that represent text; they are declared as arrays of ~~char~~ type char:
- `char word[10];` ← declares an array of elements of type char, 10 such elements.

- string initialization:

①
```
char word[10];
word[0] = 'H';
word[1] = 'e';
word[2] = 'l';
word[3] = 'l';
word[4] = 'o';
word[5] = '\0';
```
→ thus, this array of chars can store up to 9 characters plus end-of-string ('\0') character.

← end-of-string character itself must be stored as an element of the array.

②
```
char word[10] = "Hello";
```
- initialization within the declaration
- note that there is no end-of-string ('\0') char. at the end of "Hello", the compiler will add it automatically.
- also note that the actual text is in double quotes, " ", as opposit for single quotes, ' ', used for individual characters.
- strings in C are <u>null-terminated</u>.

- printing a string:

```
char word[10] = "Hello";
printf("%s", word);
```
↑

Reading a string from the Keyboard.

→ scanf("%s", word); ← will read only the first word in the string, until it finds any space characters.

→ gets (string); ← reads a line from standard input device, e.g., Keyboard, until new line character is input.

→ example:

```
char text[50];
gets (text);
```

← entered string will be stored in the 'text' array of chars, and end-of-string ('\0') character will be added at the end, after the last entered character.

→ example entered text:

|Good␣day.| ← 9 characters

will result in the following:

```
text[0] = 'G';
text[1] = 'o';
text[2] = 'o';
text[3] = 'd';
text[4] = '␣';
text[5] = 'd';
text[6] = 'a';
text[7] = 'y';
text[8] = '.';
text[9] = '\n';
```

} 10 characters.

NOTE that neither scanf nor gets make checks to ensure that there is enough space to enter no more elements than was actually allocated for the string

> Example:

read string of text from the keyboard and count number of occurences of character 'a';

```
#include <stdio.h>
#define MAX_STRING_LEN 50

int main ( )
{
   char text [MAX_STRING_LEN];
   const char testchar = 'a';
   int i, count=0;
   printf ("Enter string: ");
   gets (text);          /* get string from the user */

   for (i=0; i< MAX_STRING_LEN; i++)
   {
      if (text[i] == testchar)
         count++;
      else if (text[i] == '\0').
         break;
   }

   printf ("string %s has %d occurences of
            character %c \n", text, count, testchar);

   return 0;
}
```

Multi-dimensional arrays in C

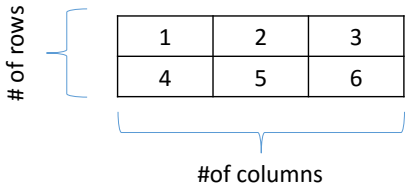
- C supports arrays of multiple dimensions
- 1D array
  - <type> <name> [<dim>];
  - e.g., int A[10];

A[0]									A[9]

- 2D arrays
  - <type> <name> [<dim1>][<dim2>];
  - e.g., int B[2][3];

B[0][0]	B[0][1]	B[0][2]
B[1][0]	B[1][1]	B[1][2]

  - here dim1 is 2 and dim2 is 3
  - array B consists of 2 elements each of which is an array consisting of 3 elements of type int.
- In C, arrays are stored in memory in such a way that rows are stored one after the other. This storage method is caller *row-major order*.
  - In row-major order, rows are identified by the first index of a 2D array and the columns are identified by the second index.
- Example:
  - int B[2][3] = { {1,2,3}, {4,5,6}};



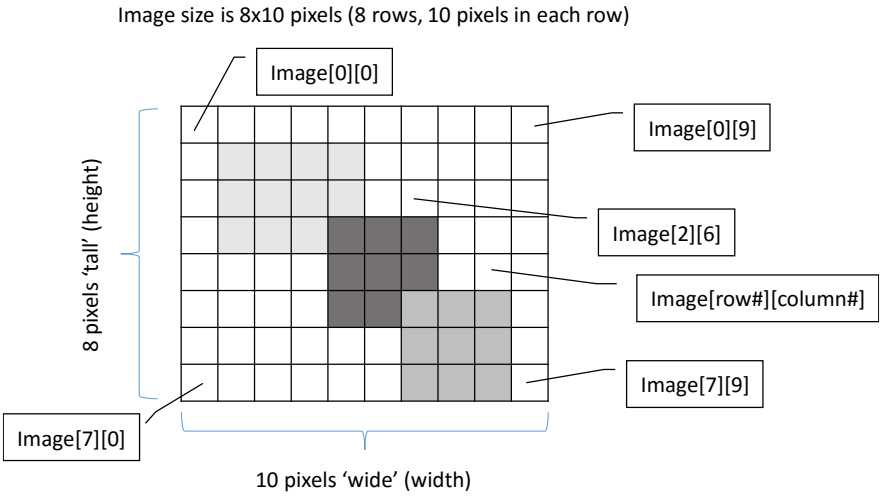
B[0][0] <- 1
B[0][1] <- 2
B[0][2] <- 3
B[1][0] <- 4
B[1][1] <- 5
B[1][2] <- 6

  - The exact offset in memory of an array element at [row][col] can be computed as

```
row x # of columns + col
```

Example: Image storage

- Image consists of individual *pixels*, each representing *brightness* value at a given location. Here we will only consider gray-scale images in which there is only one "color" component – gray.
- Image has *height* and *width*, measured in the number of pixels in each dimension



- Gray-scale image of size *height* x *width* (*# of rows* x *# of columns*) can be represented as a single 2D array of `unsigned char` values:

```
unsigned char Image[8][10];
```

  - How much memory do we need to store it?
    - 8 x 10 x sizeof(unsigned char) bytes
- Color images will consist of 3 separate color channels, red, green, and blue, each of which is a separate 2D array, as above
- As discussed before, 2D arrays (images in our case) are stored as 1D arrays in memory. Given a pointer to such an image, one can access image[row][co] location in 1D array as follows:

```
#define HEIGHT 8
#define WIDTH 10
unsigned char Image[HEIGHT][WIDTH];
unsigned char *imgptr = (unsigned char *)Image;
/* or we can "ask" for address of Image[0][0]
   unsigned char *imgptr = &Image[0][0]; */

/* imgptr[row * WIDTH + col] points to the same pixel in
the image as Image[row][col] */
```

Image analysis example

- Problem statement: Count the number of 16x16 pixel blocks in an image that match the reference block. Here both Image and Block are passed to the function Match as pointers.

```
int Match(unsigned char *imgptr, int width, int height, unsigned char
*blkptr)
{
  int row, col, i, j;
  int match, count = 0;
```

```
  for (row = 0; row < height; row+=16)
  {
    for (col = 0; col < width; col+=16)
    {
      match = 1; /* we initially assume this block will match */
      for (i = 0; i < 16; i++)
      {
        for (j = 0; j < 16; j++)
        {
          if (imgptr[(row+i)*width + (col+j)] != blkptr[i*16+j])
            match = 0;
        }
      }
      if (match)
        count++;
    }
  }
  return count;
}
```