Introduction To C Programming

Lesson 06

Arrays

Arrays

- An array is a collection of variables referenced by the same name.
- All of the variables are the same type
- Each variable, or array member, is accessed by an index
  - The first array member's index is 0
  - The last array member's index is at dimension-1
    - e.g. int size[10] /* first element 0, last element at 9 */

Arrays

- Array members are stored in sequential order in contiguous memory
- Can be one dimensional or multi-dimensional

Arrays

- Single dimension arrays are declared with this format:

```
type name[size];
```

- The size must be a constant expression
- The size is used to determine how much space to allocate
- To define an array of 10 ints:

```
int  score[10];
```

Arrays

- The first element is accessed as:

```
score[0]
```

- The last element is accessed as:

```
score[9]
```

- The number of bytes used by the array:
  - num_bytes = size * sizeof(type)

```
num_bytes = 10 * sizeof(int)
```

  - or, num_bytes = sizeof(score)

```
num_bytes = 40 /* if 4 byte ints */
```

```
/* global, static, auto array init */
#include <stdio.h>
int  val[2];              // global array
int main(void)
{
    int  i, zip[2];
    static int  amt[2];
    for (i = 0; i < 2; ++i)
       printf("val[%d] = %d\n", i, val[i]);
    for (i = 0; i < 2; ++i)
       printf("amt[%d] = %d\n", i, amt[i]);
    for (i = 0; i < 2; ++i)
       printf("zip[%d] = %d\n", i, zip[i]);
}
```

Output

```
val[0] = 0
val[1] = 0
amt[0] = 0
amt[1] = 0
zip[0] = 4206168
zip[1] = 6618628
```

Arrays

- You may initialize an array at declaration time:
- type name[size] = {comma separated list of values}

```
int score[3] = {2, 4, 6};
```

for loops and arrays

- for loops are commonly used to process the elements of an array
- Typical use is:

```
#define SIZE 10
int array[SIZE];
int i;
for (i = 0; i < SIZE; i++) {
   array[i] = i; }
```

Note the pattern of the for statement. Use this pattern when processing arrays

```
#include <stdio.h>
int  val[2] = {10,100};    // global array
int main(void)
{
    int  i, zip[2]={-3, 35};
    static int  amt[2]={75,150};
    for (i = 0; i < 2; ++i)
       printf("val[%d] = %d\n", i, val[i]);
    for (i = 0; i < 2; ++i)
       printf("amt[%d] = %d\n", i, amt[i]);
    for (i = 0; i < 2; ++i)
       printf("zip[%d] = %d\n", i, zip[i]);
}
```

Output

```
val[0] = 10
val[1] = 100
amt[0] = 75
amt[1] = 150
zip[0] = -3
zip[1] = 35
```

Strings

- A string is an array of char terminated with a null character
- It appears as a sequence of characters enclosed in quotes: “Hello”
- Each character is stored in a separate char variable as the integer value for the code from the ANSI character set.
- The null terminator is the null character ‘\0’(byte of all 0s).

Strings

- You could initialize an array of char this way:

```
char greet[6]={‘H’,‘e’,‘l’,‘l’,‘o’,‘\0’};
```

- Notice the array is defined as size 6 to accommodate the null terminator.
- Shorthand notation for this initialization:

```
char greet[6] = “Hello”;
```

- The compiler implicitly adds the null terminator when used this way

Arrays

- What happens when an array is declared larger than the initial values provided?
  - Extra array elements are initialized to 0
- What happens when an array is declared too small for the initial values provided?
  - The compiler issues an error

Arrays

- The compiler can calculate the correct number of elements:
  - Leave the size off and initialize the array:

```
int inputs[]={75, 80, 95, 98, 100, 79};
char greet[] = “Hello”;
```

- To find the number of elements in an array defined this way, use this formula:

```
sizeof(arrayname) / sizeof(arrayname[0])
```

Passing Arrays To Functions

- There is a difference in the way that arrays and individual values (e.g. char, int, float, …) are passed to functions.
  - Individual values are often called scalars
- Scalars are passed “by value”; a copy of the scalar is passed to the function
- Arrays are passed “by reference”; the address of the array is passed to the function

Passing Arrays To Functions

- The function gets access to the actual array, and not a copy of the array
- Changes to the array in the called function are changed in the original array
- The address of an array is the address of the first element

Passing Arrays To Functions

```
float array_ave(int val[], int size)
{
    int i=0; long int sum = 0L;
    for (i=0; i < size; ++i)
    {
        sum += (long)val[i];
    }
    return (float)sum / (float)size;
}
```

Passing Arrays To Functions

- The prototype for the previous function is:
  ```
  float array_ave(int [], int);
  ```
- By leaving the brackets empty, the compiler assumes that the array has had space allocated somewhere else
- The programmer must pass the address of the array to the function using the "address of" operator, the '&'
  ```
  &array_name[0]
  ```

Passing Arrays To Functions

- There is a shorthand for the address of the first element of the array, and that is the name of the array itself:

```
int values[] = {10, 20, 30, 40};
1) array_ave(values, 4);
2) array_ave(values, sizeof(values) /
                     sizeof(values[0])
3) array_ave(&values[0], 4);
```

Fun with arrays

- What is the data type and result of:
  - “012345678”[4];
  - 4[“012345678”]
- int array[2] = { 12, 23 };
  - array[0];
  - array[2];
  - 0[array];
  - sizeof(array);