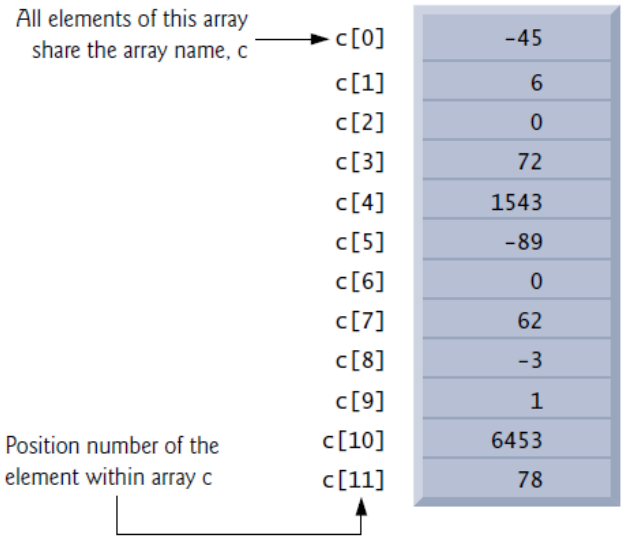
COMP2006 Computer Organization
Arrays

An array is a group of contiguous memory locations that all have the same type. To refer to a particular location or element in the array, we specify the array's name and the position number of the particular element in the array.



Defining Arrays

Arrays occupy space in memory. We specify the type of each element and the number of elements each array requires so that the computer may reserve the appropriate amount of memory.

C

```
int c[12];
```

Processing and Java

```
int[] c = new int[12];
```

In C programming language, the elements of an array will not be initialized to the default value. The elements store some old data (the memory may be used for other purposes previously). We can use the following ways to initialize arrays:

```
int c[5] = {32, 27, 64, 18, 95};

int d[10] = {0};

int n[] = {1, 2, 3, 4, 5};
```

Specifying an Array's Size with a Symbolic Constant

Consider the following example code:

```
#include <stdio.h>
# define SIZE 5


int main() {
   int i;
   int s[SIZE];

   for (i=0; i< SIZE; i++)
      s[i] = 2 + 2 * i;

   printf("%s%13s\n", "Element", "Value");

   for (i=0; i< SIZE; i++)
      printf("%7d%13d\n", i, s[i]);

   return 0;
}
```

The **#define preprocessor directive** is introduced in this program. It defines a symbolic constant **SIZE** whose value is *5*. A symbolic constant is an identifier that's replaced with replacement text by the C preprocessor before the program is compiled. When the program is preprocessed, all occurrences of the symbolic constant **SIZE** are replaced with the replacement text *5*.

Summing the Elements of an Array

The following example program sums the values contained in the 12-element integer array a. The for loop does the totaling.

```
#include <stdio.h>
# define SIZE 12

int main() {
   int a[SIZE] = {1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45};
   int total = 0, i = 0;

   for(i=0; i<SIZE; i++)
      total += a[i];

   printf("Total of array element values is %d\n", total);
   return 0;
}
```

The output of the program above is:

```
Total of array element values is 383
```

Initializing a Character Array with a String

Character arrays have several unique features. A character array can be initialized using a string literal. For example,

```
char string1[] = "Hello";
```

The string "Hello" contains *5* characters plus a special string-termination character called the **null character** ('\0'). Therefore, the actual size of array string1 is *6*.

A character arrays also can be initialized with individual character constants in an initializer list, but this can be tedious. The preceding definition is equivalent to

```
char string1[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

We also can input a string directly into a character array from the keyboard using the **scanf** function and the conversion specifier %s. For example,

```
char string2[10];
```

creates a character array capable of storing a string of at most 19 characters and a terminating null character.

```
scanf("%9s", string2);
```

Passing Array to Function

Do you know what the following function does?

```
void func(char text[]) {
   int i=0;
   while(text[i] != '\0') {
      if (text[i] >= 'a' && text[i] <= 'z')
         text[i] = text[i] + 'A' - 'a';
      i++;
   }
}
```

The function above accepts a character array (**char []**). Remember that C language does not have **String** data type and we use a character array to store a string. Therefore, we can use this method to define a function that accepts a string.

In the while loop, we read each character from the array until it is a null character (\0). If the character is in the range from 'a' to 'z', we change it to the upper case. The results are put back to the array.

The following is the **main** function that executes the function above:

```
int main() {
   char string[] = "Hello World!";
   func(string);

   printf("%s\n", string);
}
```

The name of an array, in fact, is a variable that stores the starting memory address of the array. If we call a function with the name of the array, we pass the reference of the array to the function. In the declaration of the function, we do not need to add an asterisk (*) sign to the beginning of the parameter. We only need to define the parameter in the corresponding array type.

```
#include <stdio.h>
# define SIZE 4

int main() {
   int i;
   int array[SIZE];

   printf("array = %p\n", array);  // %p --> hexadecimal number

   for (i=0; i< SIZE; i++)
      printf("&array[%d] = %p\n", i, &array[i]);

   return 0;
}
```

The example program above demonstrates that “the value of an array name” is the starting memory address of the array, also the address of the first element. You can also find the address of each element.

The following is the sample output. Note that you may get different results if you re-run the program after a period.

```
array = 0xffffcbe0
&array[0] = 0xffffcbe0
&array[1] = 0xffffcbe4
&array[2] = 0xffffcbe8
&array[3] = 0xffffcbec
```

Size of an Array

To check the size of an array, you can use the **sizeof()** function as follows:

```
sizeof(array)/sizeof(data_type)
```

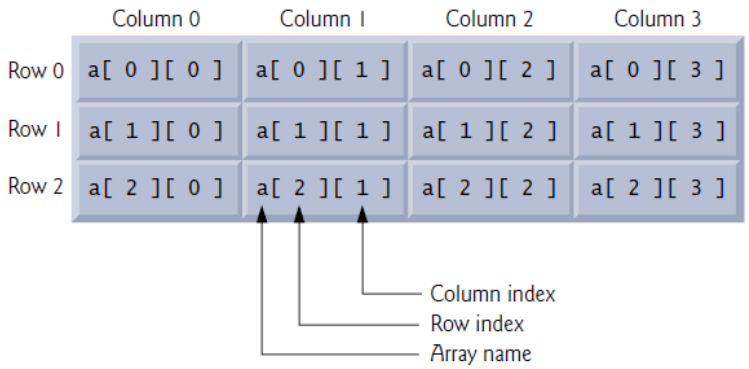
But, the **sizeof** function cannot be used to determine the size of an array parameter of a function because the parameter is a pointer to the array and the **sizeof** function returns the size of the pointer only.

```
#include <stdio.h>

int getSize(int a[]) {
   return sizeof(a)/sizeof(int);
}

int main() {
   int array[100];

   printf("The size of the array is %d\n", sizeof(array)/sizeof(int));
   printf("The size returned from the getSize function is %d\n", getSize(array));

   return 0;
}
```

Multidimensional Array

A multidimensional array can be initialized when it's defined. For example, a two-dimensional array **int a[3][4]** could be defined and initialized with:

```
int a[3][4] = {{1, 2}, {3, 4}, ... };
```



The following example program shows the memory addresses of each element of the two-dimensional array.

```c
#include <stdio.h>

int main() {
   int w, h, i, j;
   int array[3][4];

   w = sizeof(array[0])/sizeof(int);
   h = sizeof(array)/sizeof(int)/w;

   printf("%d x %d array\n", h, w);
   printf("array = %p\n", array);  // %p --> hexadecimal number

   for (i=0; i< h; i++)
      for(j=0; j <w; j++)
         printf("&array[%d][%d] = %p\n", i, j, &array[i][j]);

   return 0;
}
```

These two statements demonstrate how to find the size of a two-dimensional array.

The following is the sample output of the example program:

```
3 x 4 array
array = 0xffffcbc0
&array[0][0] = 0xffffcbc0
&array[0][1] = 0xffffcbc4
&array[0][2] = 0xffffcbc8
&array[0][3] = 0xffffcbcc
&array[1][0] = 0xffffcbd0
&array[1][1] = 0xffffcbd4
&array[1][2] = 0xffffcbd8
&array[1][3] = 0xffffcbdc
&array[2][0] = 0xffffcbe0
&array[2][1] = 0xffffcbe4
&array[2][2] = 0xffffcbe8
&array[2][3] = 0xffffcbec
```

Dynamic Array – pointer & malloc

In the previous examples, we define an array with the size. But we may not be able to determine the size of the array in advance. For that situation, we declare a pointer and use **malloc** to define an array. We need to include *stdlib.h* for **malloc**.

Consider the following example:

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *a;
    int n;
    int status;

    while (1)
    {
        printf("Please input an integer (1 to 5): ");
        status = scanf("%d", &n);

        fflush(stdin);

        if (status == 1 && n > 0 && n < 6)
            break;
    }

    a = malloc(n * sizeof(int));

    for (int i = 0; i < n; i++)
        a[i] = i * 10;

    printf("We have ");

    for (int i=0; i< n; i++)
        printf("%d ", a[i]);

    printf("\n");
}
```

In the example code above, we declare an integer pointer. Then, we create an array with the *n * sizeof(int)*. Where *n* represents the number of elements in the array. the **malloc** function allocates the memory space in bytes for the array and returns the reference address of the memory space.

The ***fflush(stdin)*** statement flushes the cache of standard input (the console by default). It ensures that the previous input will not affect the next input.

Exercise 1

Given an integer array that stores some integers in the range *1* to *9*. Finish the **mean** function of the following C program to find and print the **mean** of the integers stored in the given array.

```c
#include <stdio.h>

float mean(int a[], int size);

int main() {

   int array[] = {
      6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
      7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
      6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
      7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
      6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
      7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
      5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
      7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
      7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
      4, 5, 6, 1, 6, 5, 7, 8, 7
   };

   float m = mean(array, sizeof(array)/sizeof(int));

   printf("the mean is: %.2f\n", m);

   return 0;
}

float mean(int a[], int size) {
   // your code here
}
```

Save your code and name the file **mean.c**. Submit your file to Moodle.

Exercise 2

Given an integer array that stores some integers in the range *1* to *9*. Finish the **plotGraph** function of the following C program to plot a graph with the asterisk (*) sign using the data stored in the array.

```c
#include <stdio.h>

void plotGraph(int a[], int size);

int main() {

   int array[] = {
      6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
      7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
      6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
      7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
      6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
      7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
      5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
      7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
      7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
      4, 5, 6, 1, 6, 5, 7, 8, 7
   };

   plotGraph(array, sizeof(array)/sizeof(int));

   return 0;
}

void plotGraph(int a[], int size) {
   // your code here
}
```

The following is the expected output:

```
1: *
2: ***
3: ****
4: *****
5: ********
6: *********
7: ***********************
8: ***************************
9: ******************
```

Save your code and name your file **graph.c**. Submit your file to Moodle.

Exercise 3

Write a C program that does the followings:

1. Ask the user to input a size *N*.
2. Repeat step 1 if the user inputs a non-integer value.
3. Create an integer array of size *N*.
4. Ask the user to input *N* integers and store the integers in the integer array.
5. After input, the program prints the minimum, maximum, and the average of the integers.

Save your code and name the file *dynamic1.c*. Submit your file to Moodle.