Chapter 4
Arrays and Pointers
Tip of the Day

- Read small programs given in the book.
- Then close the book
- Write down on a piece of paper what the program was doing.
- Re-write it in C using your own imagination.
- Compare with the program given in the book
Topics

- Arrays
  - One Dimensional Arrays
  - Initialization of Arrays
  - Arrays as Parameters
- Pointers
  - What is a pointer
  - Pointer Arithmetic
  - Dereferencing a Pointer Value
- Arrays and Pointers
  - An Array as a Pointer
  - Passing Parameters to a Function
  - Searching and Sorting Arrays
  - Dynamic Memory Allocation
One Dimensional Arrays

• An array is a table of data of the same type in C.
• An array is a single C Identifier that can be associated with multiple values.
• Example:
• `int MyArray[15]` creates an 15 integers associated with the identifier MyArray
One Dimensional Arrays

- A dimension of an array is an indicator of the coordinates needed to find a piece of data inside of the array.
- For this chapter, only one dimension is being used.
- Each one dimensional array will be one column by $n$ rows.
One Dimensional Arrays

- The location of a value in an array is called an **index**.
- Given that some number \( n \) defines how many rows are in the table, the **index** states which row is currently being used.
One Dimensional Arrays

• The memory where a value of an array is called a **cell**
• Each entry in the table created by the array can hold one value of that data type.
• Each cell can be accessed by using the identifier with an index.

Examples:
- MyArray[12] = 25;
- x=MyArray[12];
- MyArray[5]=y;
- printf("%d\n",MyArray[2]);
One Dimensional Arrays

- Arrays are used in many different languages.
- Each system has different symbols and rules.
- In ANSI C, to either create an array or to index a cell of an array \([\text{ and }]\) are used.
- In ANSI C, the index of an array starts at 0.
One Dimensional Arrays

- A “One dimensional” array can be shown as one column of $n$ rows of data.
- Declaration of an array
  ```
  int ArrayOfSamples[8];
  ```

ArrayOfSamples:

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>?</td>
</tr>
<tr>
<td>4</td>
<td>?</td>
</tr>
<tr>
<td>5</td>
<td>?</td>
</tr>
<tr>
<td>6</td>
<td>?</td>
</tr>
<tr>
<td>7</td>
<td>?</td>
</tr>
</tbody>
</table>
One Dimensional Arrays

• Like other variables, the cells of the array contain the left-over information in the system memory.

• An array needs to be initialized for every cell of its structure.
One Dimensional Arrays

• A for loop is a good choice to initialize an array.

```c
for (LoopControl=0; LoopControl<8; LoopControl++)
    ArrayOfSamples[LoopControl] = LoopControl + 100;
```
### One Dimensional Arrays

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>?</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>?</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>?</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>?</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>?</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>?</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>
One Dimensional Arrays

• To set the value of a single cell, use an pair of [] with the index and the identifier of the array.

ArrayOfSamples[4]=10;

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>
One Dimensional Arrays

- To get the value stored in a cell of the array, use the same [] notation with the index on the right side of an expression.

\[
\begin{align*}
\text{int Value} & = 0; \\
\text{Value} & = \text{ArrayOfSamples}[4]; \\
\text{printf(“Value is %d\n”, Value);}
\end{align*}
\]

Value is 10

ArrayOfSamples

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>
Other Initializations

- Using Braces {} to initialize the array.

```cpp
char name[7]={'A','l','b','e','r','t','\0'};
```
• Each cell of the array can be named individually

```c
char name[7];
name[0]=’A’;
name[1]=’l’;
name[2]=’b’;
name[3]=’e’;
name[4]=’r’;
name[5]=’t’;
name[6]=’\0’;
```
Other Initializations

- The size of the array could be left unspecified, and take its properties from the initialization.

```c
float Fahrenheit[] = {32.0, 98.6, 212.0};
```
#include <stdio.h>
#define NUMCHARS 10
#define NUMFLOATS 8

int main(void)
{

char chararray[NUMCHARS];
int intarray[] = {2,1,3,5,4,8,3,7};
double dblarray[NUMFLOATS] = {1.2, 3.4, -2.3, 1.4, 4.5};
int index;

for (index = 0; index < NUMCHARS; index++)
    chararray[index] = 127 - index;
```c
#include <stdio.h>
#define NUMCHARS 10
#define NUMOFFLOATS 8

int main(void)
{
    char chararray[NUMCHARS];
    int intarray[] = {2, 1, 3, 5, 4, 8, 3, 7};
    double dblarray[NUMOFFLOATS] = {1.2, 3.4, -2.3, 1.4, 4.5};
    int index;

    for (index = 0; index < NUMCHARS; index++)
        chararray[index] = 127 - index;
}
printf("chararray occupies %d bytes.\n", sizeof(chararray));

printf("intarray occupies %d bytes.\n", sizeof(intarray));

printf("dblarray occupies %d bytes.\n", sizeof(dblarray));
printf("The element in chararray with index 3 is \%
" , chararray[3]);

printf("The element in intarray with index 3 is \%
" , intarray[3]);

printf("The element in dblarray with index 3 is \%
" , dblarray[3]);
for (index = 0; index < 8; index++)
printf("The element in intarray with index %d is \n\"%d\".\n", index, intarray[index]);
for (index = 0; index < 8; index++)
printf("The element in intarray with index %d is ‘%d’.\n," index, intarray[index]);
for (index = 0; index <= 8; index++)
printf("The element in intarray with index %d is \\
'\%d'.\n", index, intarray[index]);
for (index = 0; index < 8; index++)
printf("The element in intarray with index %d is 'd'.\n", index, intarray[index]);
Arrays as Parameters to Functions

• An Array can also be passed as a parameter to a function.
• The Array is a memory area.
• The Function
• The function definition appears as:
  ```c
  void ShowArray(int ArrayOfInt[], int size)
  ```
• The function call will look like:
  ```c
  { int ArrayOfInt[10];
    int LoopControl=0;
    for (LoopControl=0; LoopControl<10; LoopControl++)
      ArrayOfInt[LoopControl]=LoopControl*2;
    ShowArray(ArrayOfInt, 11);
  }
  ```
void ShowArray(int ArrayOfInt[], int size) {
    int LoopControl = 0;
    for (LoopControl = 0; LoopControl < size; LoopControl++)
        printf("[%d][%d]\n", LoopControl, Array[LoopControl]);
}
Arrays as Parameters

- One item about arrays is that an array is a table.
- When a parameter is passed to a function, a copy of that parameter is sent.
Arrays as Parameters

- When an array is sent as a parameter, it is the location of the table of values.
- If a value is sent as a parameter, it is copied.
- If the location of a table of values is sent as a parameter, are the values copied?
Computer Memory

• One aspect of Array Variables and Scalar Variables are the fact they use computer memory
Computer Memory

- The identifier associated with the variable is a label saying it is in some memory area.
Memory

- What is shown to the human is a location that has been mapped to a value that is more people friendly.
- A single 16-bit integer is simply out there in a pool of memory that is at some location.
- That location in Memory is called the **Address**
Pointers

- A pointer is an address in memory that is the beginning of a piece of data.
Pointers

• The data can be of any size or type, be it a float, char, int, or an array.
Pointers

• An idea of a pointer would be the handle on a suitcase. You grip the case by the handle, but the suitcase can be of different sizes.
Pointers

• But the handle is always of the same size.
Pointers

The handle (pointer) is the same size to two different storage areas.
Pointers

- All variables in C are stored in memory.
- Each location in memory is called an address.
- Each location in memory has a type associated with it.
- Pointers are declared with a type, followed by a *, followed by the identifier.
- Pointers, like other variables, start out with a nonsense value. The zero value for pointers is called “NULL”.

```c
int *IntPointer=NULL;
float *FloatPointer=NULL;
char *CharPointer=NULL;
```
Pointers

- Now they are declared, they can be attached to values.
- Assignment to a non-pointer value is done with an &

```cpp
IntPointer = &IntValue;
FloatPointer = &FloatValue;
CharPointer = &CharValue;
```
Pointer Arithmetic

• Pointers are always whole numbers, or integers.

• **Pointers do not behave the same way as numbers.**

• Pointers behave as increments of their type size in memory.

• The value of a pointer will be the number added multiplied by the `sizeof()` the data type.
Print the respective addresses:

```c
printf("The int is at %p\n", IntPointer);
printf("The float is at %p\n", FloatPointer);
printf("The char is at %p\n", CharPointer);
printf("The next int is at %p\n", IntPointer+1);
printf("The next float is at %p\n", FloatPointer+1);
printf("The next char is at %p\n", CharPointer+1);
```

The int is at 0012FE68
The float is at 0012FE5C
The char is at 0012FE53
The next int is at 0012FE6C
The next float is at 0012FE60
The next char is at 0012FE54
Dereferencing a Pointer Value

• A pointer, being an address of the memory of a variable, is said to reference the variable.

• In order to see the value of the data of the pointer, the pointer has to be dereferenced.

• The symbol to dereference a pointer is the asterick (star) *
Dereferencing a Pointer Value

printf("The value at %p is %g\n", FloatPointer, *FloatPointer);
printf("The value at %p is %d\n", IntPointer, *IntPointer);
printf("The value at %p is %c\n", CharPointer, *CharPointer)

The value at 0012FE5C is 10
The value at 0012FE68 is 15
The value at 0012FE53 is 5
Arrays and Pointers

• Both an Array and a Pointer can see an area of memory.
• An array operates on an index, and the pointer operates on the size of the data type.
• Given that the pointer is handling the same memory as the array.
• Given that the pointer is limited to the size of the array.
• The Array can be referenced as a pointer.
Arrays and Pointers

```c
for (LoopControl=0; LoopControl<10; LoopControl++)
{
    DataArray[LoopControl]=2*LoopControl;
}
DataPointer=DataArray;
for (LoopControl=0; LoopControl<10; LoopControl++)
{
    printf("%p=%d\n", DataPointer+
            LoopControl, *(DataPointer+LoopControl));
}
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>12FE68</td>
<td>0</td>
</tr>
<tr>
<td>12FE6C</td>
<td>2</td>
</tr>
<tr>
<td>12FE70</td>
<td>4</td>
</tr>
<tr>
<td>12FE74</td>
<td>6</td>
</tr>
<tr>
<td>12FE78</td>
<td>8</td>
</tr>
<tr>
<td>12FE7C</td>
<td>10</td>
</tr>
<tr>
<td>12FE80</td>
<td>12</td>
</tr>
<tr>
<td>12FE84</td>
<td>14</td>
</tr>
<tr>
<td>12FE88</td>
<td>16</td>
</tr>
<tr>
<td>12FE8C</td>
<td>18</td>
</tr>
</tbody>
</table>
Passing Pointers to Function

• It is possible to pass pointers to a function by including the pointer notation in the declaration

```c
void ChangeMemory(int *PointerVariable)
```

• By doing so, the handle of the memory area is sent off, and can be changed by the function.
• The return value can remain void.
Passing Pointers to Function

- This is called *pass by reference*
- Previous method of passing variables as parameters is called *pass by value*
- Pass by value means the value of the parameter is copied and sent to be processed
- Pass by reference means the address of the value is sent to be processed
Searching and Sorting arrays

- One method of searching an array is a **linear search**
- In a linear search, each entry of the array is read one at a time, and compared to the target value.
Searching and Sorting Arrays

- One method of sorting is a selection sort.
- The routine searches for the highest value of the data in the array, and swaps it with the last valid location.

```plaintext
[6][8][9][1][5]
[6][8][9][1][5]
[6][8][5][1][9]
[6][8][5][1][9]
[6][1][5][8][9]
[6][1][5][8][9]
[5][1][6][8][9]
[5][1][6][8][9]
[1][5][6][8][9]
[1][5][6][8][9]
```
Intro to Dynamic Memory

- `sizeof()` – returns the size of a variable
- `sizeof(double);`
- `malloc()` – gives a memory area of a known size
- `intpointer=(int *) malloc (sizeof(int));`
free() – releases the assigned memory
free((void *)intpointer)
So, to create an array of memory it could be done as:
intarray=(int *)malloc(n*sizeof(int));