MECH215
Lecture Notes 6

Arrays

Suppose we want to find the maximum value of 5 integers stored in variables called value0, value1, value2, value3, value4. The following portion of code will achieve the desired result:

```
int maximum  = value0;
if (value1 > maximum) { maximum = value1; }
if (value2 > maximum) { maximum = value2; }
if (value3 > maximum) { maximum = value3; }
if (value4 > maximum) { maximum = value4; }
```

Note how 4 separate if statements were required and a total of 5 separate integer variables were needed to hold the values being compared. One can easily see how such an approach would not be practical to find the maximum of 1000 integer values.

C++ allows for the array data type to simplify such situations in which one is working with a set of similar data items. An array is simply a collection of similar items referred to by a single common name in which every individual item or element in the collection can be individually referenced.

The following program defines and initializes an array of 10 integers:

```
// Author: Ted Obuchowicz and John Peach
// array.cpp
// example program illustrating use of an array

#include <iostream>
#include <string>

using namespace std;

int main() {

    // declare and initialize an array of 10 integers
    int myArray[10] = {12, 234, 23, 1, -7, 55, 18, 67, 99, 100};

    // sequentially access elements of the array
    for(int i = 0; i < 10; i++) {
        cout << "myArray[" << i << "] = " << myArray[i] << endl;
    }

    return 0;
}
```

The program produces the following output:

```
myArray[0] = 12
myArray[1] = 234
myArray[2] = 23
myArray[3] = 1
myArray[4] = -7
myArray[5] = 55
myArray[6] = 18
myArray[7] = 67
myArray[8] = 99
myArray[9] = 100
```

In general arrays are declared in the following manner:

baseType arrayName[size]

where
baseType: is the data type of each array element such as char, int, float, double, etc... (it is even possible to have an array of structs)

arrayName: is an identifier which specifies the name of the array

size: is a constant or a constant expression which identifies the SIZE of the array. The expression must be capable of being evaluated to integer value at COMPILE time.

Some examples of legal array declarations are:

```
float  someRealNumber[25];   // an array of 25 floats
char   twoLetter[2];         // an array which can hold two chars
const  int m = 5;
double constSize[m];         // Defining the size using a constant
```

Some illegal array definitions are:

```
float someRealNumber[25.5];  // an array of 25 floats
const float m=4;
const float n=5;
int anArray[m*n];             // m*n = 20 but this is a float
```

The above are illegal since non-integer types are used to specify the array sizes. The compiler will give error messages like the following:

size of array 'someRealNumber' has non-integer type
size of array 'anArray' has non-integer type

Let us explore this issue of the array size specifier being a CONSTANT integer expression:

Here is a C++ program which attempts to use an integer variable y to specify the size of an integer array called x:

```
#include <stdio.h>

int main() {
    int y;
    int x[y];

    return 0;
}
```

This program does NOT compile on some compilers. They will give an error message like:

```
integral constant expression expected
```

Some other compiler's, like g++, allows this to compile but it is dangerous to do so. Let us add a little more functionality to our program and try to compile it with g++ and run it:

```
// Author: Ted Obuchowicz and John Peach
// arrayRunTime.cpp
// example program illustrating the sizing of
// an array at run-time.  This may not always work

#include <iostream>

using namespace std;

int main() {

    int size;               // Size of the array

    // Get the size from the user
    do {
        cout << "Enter the array size: ";
        cin >> size;
    } while (size < 1);

    int x[size];            // Allocate room for the array.
                            // This does not work on all compilers

    // Enter values into the array
    for(int i = 0; i < size; i++) {
```

```
        cout << "Enter an integer: ";
        cin >> x[i];
    }

    // Write out the array
    for(int i = 0; i < size; i++) {
        cout << x[i] << endl;
    }

    return 0;

}
```

It compiles fine... it even runs fine:

```
Enter the array size: 3
Enter an integer: 3
Enter an integer: 5
Enter an integer: 2
3
5
2
```

Let us change the program slightly, so that we input the value of size AFTER, we declare the array x:

```
// Author: Ted Obuchowicz and John Peach
// arrayRunTimeNotInitialized.cpp
// example program illustrating the sizing of
// an array at run-time.  This may not always work

#include <iostream>

using namespace std;

int main() {

    int size;               // Size of the array
    int x[size];            // Allocate room for the array.
                            // This does not work on all compilers
                            // Note: size has not been initialized

    // Get the size from the user
    do {
        cout << "Enter the array size: ";
        cin >> size;
    } while (size < 1);
```

```
    // Enter values into the array
    for(int i = 0; i < size; i++) {
        cout << "Enter an integer: ";
        cin >> x[i];
    }

    // Write out the array
    for(int i = 0; i < size; i++) {
        cout << x[i] << endl;
    }

    return 0;

}
```

This program may compile fine with the g++ compiler. It depends on the default value of size. Recall that size will have a value even if we do not initialize it. If the value happens to be negative during run time we will get a segmentation fault. Or, if the user enters a value bigger then the junk value that is in size before we initialized it, then we will corrupt the memory.

The issue of the array size specifier being an integer constant is compiler-dependent. The Sun C compiler is the most strict, there may be some command-line options with g++ which may tell the compiler to be more strict about these issues. Actually, the ISO and ANSI standard for C++ does not allow use to do this. To make your code portable from one machine to another, it is best to always code using the ISO standards. g++ can be forced to use these standard if we were to compile with the following options:

```
#> g++ arrayRunTimeNotInitialized.cpp -Wall -pedantic
arrayRunTimeNotInitialized.cpp: In function ‘int main()’:
arrayRunTimeNotInitialized.cpp:13:15: warning: ISO C++ forbids variable length
array ‘x’ [-Wvla]
```

You can get a long list of options for the g++ compiler by using the "man" command in unix. This gives you access to the official manuals for unix commands. To access the manual for g++ use the command

```
#> man g++
```

If we scroll down we will see this entry for the -pedantic option

> -pedantic
> Issue all the warnings demanded by strict ISO C and ISO C++; reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++. For ISO C, follows the version of the ISO C standard specified by any -std option used.
>
> Valid ISO C and ISO C++ programs should compile properly with or without this option (though a rare few will require -ansi or a -std option specifying the required version of ISO C).

However, without this option, certain GNU extensions and traditional C and C++ features are supported as well. With this option, they are rejected.

These small examples help to prove that one does not know what will occur with a program until you actually compile and run it (using DIFFERENT) compilers. It should be pointed out that there is not a single C++ ISO standard. Over the years the language has evolved and therefore different standards have been added. Also, each compiler has "extensions" added to the language which makes it behave differently. If you always code to the ISO standard you will not have an issue as all compilers are support all the ISO standards.

In C++, array element number ALWAYS BEGINS with 0 for the first array element, 1 for the second array element, 2 for the third array element, etc. The programmer has no choice in this numbering sequence (unlike the Pascal, FORTRAN and BASIC programming languages).

The elements of an array ARE STORED IN CONTIGUOUS MEMORY LOCATIONS. This following program prints out the 10 array elements as well as the address in main memory where each element is stored:

```
// Author: Ted Obuchowicz and John Peach
// arrayAddress.cpp
// example program illustrating use of an array

#include <iostream>
#include <string>
#include <iomanip>

using namespace std;

int main() {

    // declare and initialize an array of 10 integers
    int myArray[10] = {12, 24, 23, 1, -7, 55, 18, 67, 99, 10};

    // Sequentially access the elements of the array
    for(int i = 0; i <=9; i++){
        cout <<   "myArray[" << i << "] = " << myArray[i] <<
        "\tstored at address " << (unsigned long)&myArray[i] << endl;
    }

    return 0;
}
```

Note the use of the & operator in front of the &myArray[i]. This operator is used to return the main memory address which is used to store the variable. Note how we cast this value into an unsigned long integer to avoid having addresses given in their default hexadecimal value. All memory addresses are

unsigned long integers. The * operator is the opposite of & in that is gets the value at the memory address. For example *myArray gives 12 as it is the value at the memory address stored in myArray.

The output is:

```
myArray[0] = 12     stored at address 140734606839088
myArray[1] = 24     stored at address 140734606839092
myArray[2] = 23     stored at address 140734606839096
myArray[3] = 1      stored at address 140734606839100
myArray[4] = -7     stored at address 140734606839104
myArray[5] = 55     stored at address 140734606839108
myArray[6] = 18     stored at address 140734606839112
myArray[7] = 67     stored at address 140734606839116
myArray[8] = 99     stored at address 140734606839120
myArray[9] = 10     stored at address 140734606839124
```

Note how an integer occupies 4 successive bytes in main memory. Recall that this may vary from machine to machine and can be determined by using the sizeof(int) function.

If we had not cast the address as an unsigned long integer, the addresses wold be given in hexadecimal as:

```
myArray[0] = 12     stored at address 0x7fff87b98e50
myArray[1] = 24     stored at address 0x7fff87b98e54
myArray[2] = 23     stored at address 0x7fff87b98e58
myArray[3] = 1      stored at address 0x7fff87b98e5c
myArray[4] = -7     stored at address 0x7fff87b98e60
myArray[5] = 55     stored at address 0x7fff87b98e64
myArray[6] = 18     stored at address 0x7fff87b98e68
myArray[7] = 67     stored at address 0x7fff87b98e6c
myArray[8] = 99     stored at address 0x7fff87b98e70
myArray[9] = 10     stored at address 0x7fff87b98e74
```

Array Bounds

C++ does not prevent the programmer from exceeding the boundaries of an array. This may cause strange program behaviour leading to incorrect results. It has been the source of many security issues in C++ programs and has been one of the most common criticisms of the language. Proper programming techniques and attention to detail can prevent these errors.

```
// Author: Ted Obuchowicz and John Peach
// arrayBounds.cpp
// example program illustrating exceeding an arrays bounds

#include <iostream>
#include <string>

using namespace std;
```

```
int main() {

    // define and initialize an array of 14 characters
    char name[14] = {'K', 'E', 'I', 'T', 'H', ' ', 'R', 'I', 'C', 'H', 'A', 'R',
                     'D', 'S' };
    // Another array that will probably be located after "name" in memory
    char extra[5] = {'o', 't', 'h', 'e', 'r'};

    // use a for loop to sequentially access the individual elements of the array
    // and intentionally go beyond the end of the array

    for(int i = 0; i < 20; i++){
        cout << "name[" << i << "] = " << name[i] << endl;
    }

    return 0;

}
```

The output is:

```
name[0] = K
name[1] = E
name[2] = I
name[3] = T
name[4] = H
name[5] =
name[6] = R
name[7] = I
name[8] = C
name[9] = H
name[10] = A
name[11] = R
name[12] = D
name[13] = S
name[14] =
name[15] =
name[16] = o
name[17] = t
name[18] = h
name[19] = e
```

Note how the values for name[14] through to name[19] are simply attempt to interpret the bit patterns found in these main memory locations as characters. The creation of the array extra right after creating name resulted in extra taking the memory following the memory allocated to name. The compiler does not have to do this, but it often does. It can be seen from the output that once the last element in name, name[13] was reached there were a few blank memory addresses and then the values in extra started to get printed. This is not something that we want to do but C++ lets us do it. The way that C++ access elements in the array is to perform a simple calculation. For example, if we wanted to access name[17] where name is an array of integers the formula would be:

```
&name[17] = name + 17 * sizeof(int)
```

&name[17] is the memory address of the 18th (recall, 0 indexed) element. If the name array started at address 140733886790864 and sizeof(int) returned a 4 then &name[17] would be 140733886790932. C++ will not do any checking to see if that is within the 14 * 4 values that we have reserved for the array. While this can be problematic, and it is, it actually allows the code to execute much faster.

C++ even lets a programmer inadvertently supply a negative value as an array index (and go to a memory location which PRECEDES the first array element) :

```
// Author: Ted Obuchowicz and John Peach
// arrayBoundsBefore.cpp
// example program illustrating exceeding an arrays bounds

#include <iostream>
#include <string>

using namespace std;

int main() {

    // define and initialize an array of 14 characters
    char name[14] = {'K', 'E', 'I', 'T', 'H', ' ', 'R', 'I', 'C', 'H', 'A', 'R',
                     'D', 'S' };
    // Create another array that will probably be put right after "name" in memory
    char extra[5] = {'o', 't', 'h', 'e', 'r'};

    // use a for loop to sequentially access the individual elements of the array
    // and intentionally go beyond the end of the array

    for(int i = -10; i < 5; i++){
        cout << "extra[" << i << "] = " << extra[i] << "\t" <<
        (unsigned long) extra + i * sizeof(char) << "\t" <<
        (unsigned long) &extra[i] << endl;
    }

    return 0;

}
```

The output is:

```
extra[-10] = R 140734930672582     140734930672582
extra[-9] = I  140734930672583     140734930672583
extra[-8] = C  140734930672584     140734930672584
extra[-7] = H  140734930672585     140734930672585
extra[-6] = A  140734930672586     140734930672586
extra[-5] = R  140734930672587     140734930672587
extra[-4] = D  140734930672588     140734930672588
extra[-3] = S  140734930672589     140734930672589
```

```
extra[-2] =    140734930672590     140734930672590
extra[-1] =    140734930672591     140734930672591
extra[0] = o   140734930672592     140734930672592
extra[1] = t   140734930672593     140734930672593
extra[2] = h   140734930672594     140734930672594
extra[3] = e   140734930672595     140734930672595
extra[4] = r   140734930672596     140734930672596
```

Notice that it accesses memory before extra[0] which is the first element in the array extra. The second column of numbers is the calculated memory address give the formula in the above section. This uses the memory address of extra (140734930672592) and then adjusts for the offset taking the sizeof the data type, in this case char, into account. The third column is the address determined by getting C++ to tell us the address of extra[i]. This is done with the &, reference operator, with the command &extra[i].

Now that we have covered some of the basics, let us rewrite our program which finds the maximum of a list of items making use of array notation:

```
// Author: Ted Obuchowicz and John Peach
// max.cpp
// example program illustrating use of an array to store
// 10 numbers and find the maximum value

#include <iostream>
#include <string>

using namespace std;

int main() {

    const int size = 10;    // Size of the array
    int data[size];         // Allocate room for the array.
    int max;                // max value in data

    // Enter values into the array
    for(int i = 0; i < size; i++) {
        cout << "Enter an integer: ";
        cin >> data[i];
    }

    // Find the max value
    max = data[0];          // set the max value to the first element
    for(int i = 1; i < size; i++) {
        if ( data[i] > max ) {
            // We found a bigger value, store it
            max = data[i];
        }
```

```
    }

    // Report the largest value
    cout << "Max: " << max << endl;

    return 0;

}
```

The output is:

```
Enter an integer: 5
Enter an integer: 6
Enter an integer: 4
Enter an integer: 5
Enter an integer: 3
Enter an integer: 67
Enter an integer: 3
Enter an integer: 7
Enter an integer: 3
Enter an integer: 6
Max: 67
```

Array Initialization

C++ supports various methods of specifying the initial values of array elements.

```
// Author: Ted Obuchowicz and John Peach
// arrayInitialize.cpp
// example program illustrating
// different methods of array initialization

#include <iostream>
#include <string>

using namespace std;

int main() {

    // declare and initialize an array of 10 integers
    // by giving the values of all 10 elements in a
    // comma spearated list enclosed in curly parentheses
    int array1[10] = {12, 234, 23, 1, -7, 55, 18, 67, 99, 100};

    //  declare and initialize an array of 5 integers by
```

```
    //  explicitly initializing the first element to 5,
    //  the remaining unspecified elements will be set to 0
    //  also by the compiler
    int array2[5] = {5};

    // use a for loop to sequentially access the individual
    // elements of the array
    for(int i = 0; i < 10; i++) {
        cout << "array1[" << i << "] = " << array1[i] << endl;
    }

    for(int i = 0; i < 5; i++) {
        cout << "array2[" << i << "] = " << array2[i] << endl;
    }

    return 0;
}
```

The output is:

```
array1[0] = 12
array1[1] = 234
array1[2] = 23
array1[3] = 1
array1[4] = -7
array1[5] = 55
array1[6] = 18
array1[7] = 67
array1[8] = 99
array1[9] = 100
array2[0] = 5
array2[1] = 0
array2[2] = 0
array2[3] = 0
array2[4] = 0
```

Array Initialization Without Specifying An Explicit Array Size

It is possible to initialize an array without specifying it size. The compiler will determine the array size by counting the number of expressions encountered in the initialization list. A few examples illustrate this:

```
int numbers[] = {0,1,2,3,4,5,6,7,8,9};
char lowerCaseVowels[] = {'a','e','i','o','u'}
```

The size of 'numbers' will be set to 10 and the size of the lowerCaseVowels array will be set to 5.

Character Array String Initialization

C++ allows for an alternate initialization of character arrays which uses a character string (enclosed in double quotes) to specify the individual initial values of the array elements:

For example:

```
char name[] = "Ted";
```

will create and initialize an array of FOUR elements (not three):

```
name[0] = 'T'
name[1] = 'e'
name[2] = 'd'
name[3] = '\0'
```

The last element of the my_name array is initialized to the NULL CHARACTER ('\0'). The null character is used in C++ to indicate the end of a string. This type of array is called a "c-string" as it was the way that strings were stored in C. It has been a source of many bugs and when C++ was developed it was replace by a String Class. We will learn about them in the future. For now we will use c-strings as they are still commonly used.

Character arrays which have been initialized in such a fashion may either be printed out element by element or by simply sending the array directly to the cout stream with the << manipulator:

```
// Author: Ted Obuchowicz and John Peach
// outputCString.cpp
// example program illustrating use of c-strings

#include <iostream>
#include <cstring>

using namespace std;

int main() {

    char name[] = "Ted";

    // print out the array element by element
    for(int i = 0; i < strlen(name); i++) {
        cout << name[i];    // print a single character
    }
    cout << endl;

    // alternate method of printing out character array strings
```

```
    cout << name << endl;   // print all the characters in the array
                            // but not including the null character

    return 0;
}
```

The output is the same in both cases:

```
Ted
Ted
```

We used the standard library function strlen, which is part of cstring. It returns the number of characters in the string name. It does this by counting the number of characters up to but excluding the \0 character. While \0 is not printed if we did send \0 to cout it is ignored.

Here are some more examples of different ways of initializing array to the same initial value and different ways of printing out the identical results:

```
// Author: Ted Obuchowicz and John Peach
// cStrings.cpp
// example program illustrating use of c-strings

#include <iostream>
#include <cstring>

using namespace std;

int main() {

    //  four ways to initialize an array to the same values
    char name1[]   = "Ted";                 // \0 will be added automatically
    char name2[10] = {'T','e','d','\0'};    // more space then needed
    char name3[]   = {'T','e','d','\0'};    // compiler figures out the space
    char name4[10] = "Ted";                 // more space then needed

    // print out the array element by element
    for(int i = 0; i < strlen(name1); i++) {
        cout << name1[i];  // print a single character
    }
    cout << endl;

    // print out the array element by element
    // Not all 10 characters are printed
    for(int i = 0; i < strlen(name2); i++) {
        cout << name2[i];  // print a single character
    }
    cout << endl;

    // print out the array element by element
    for(int i = 0; i < strlen(name3); i++) {
        cout << name3[i];  // print a single character
    }
```

```
    cout << endl;

    // print out the array element by element
    // Not all 10 characters are printed
    for(int i = 0; i < strlen(name4); i++) {
        cout << name4[i];  // print a single character
    }
    cout << endl;

    // print all the arrays as arrays, not char by char
    cout << name1 << endl << name2 << endl << name3 << endl <<  name4 << endl;

    // print a c-string array by passing the ADDRESS of the first element
    // This is actually doing the same thing as the command above
    cout << &name1[0] << endl << &name2[0] << endl << &name3[0] << endl
         << &name4[0] << endl;

    // this will print out the letter 'T'
    cout << name1[0] << endl;

    return 0;
}
```

The output is:

```
Ted
Ted
Ted
Ted
Ted
Ted
Ted
Ted
Ted
Ted
Ted
Ted
T
```

Reading In Character Arrays From The Keyboard

Character arrays may be read in directly from the keyboard using a simple

```
cin >> nameOfArray;
```

statement. Any other type of array (array of ints, array of floats, array of doubles) cannot be read in entirely in a similar fashion, they have to be entered element by element within a loop as in :

```
int intArray[5];

for(int i = 0; i < 5 : i++) {
    cin >> intArray[i];
}
```

When a character array is read in directly from the keyboard, the end-of-string character ('\0' is added after the last character is entered by pressing the enter key of the keyboard). You have to be careful that the user does not enter more data then the length of the array or it will corrupt the data.

Here is a simple program which illustrates these concepts. It declares a character array of up to 80 chars which is used to store a name which the user enters from the keyboard. Pressing the Enter key terminates the input (and stores the '\0' character into the array). The program then prints out the array in reverse element by first locating the array index which corresponds to where the '\0' is stored, then using a for loop from this index value backwards to value 0 to print out each individual array element:

```
// Author: Ted Obuchowicz and John Peach
// readInWord.cpp

#include <iostream>
#include <string>

using namespace std;

int main() {

    const int size = 15;    // The size of the buffer
    char word[size];        // reserve up to size characters
    int nullIndex = 0;      // index where the null character is

    // Ask the user to enter a word
    cout << "Enter a word (" << size - 1 << " characters max or bad things may
happen): ";
    cin >> word;   // read in the entire string from the keyboard directly

    // Print out the word
    cout << word << endl;   // print it out .. this will print only up to the '\0'

    // use a for loop to print out the ENTIRE 80 characters...
    for (int i = 0; i < size; i++) {
        cout << "word[" << i <<"] = " <<  word[i] << "\t" << (short)word[i]
             << endl;
    }

    // find where the '\0' is stored which indicated the end of the entered word
    // Use the fact that \0 is really false
    while (word[nullIndex]) {
        nullIndex++;
    }

    // Tell the user where \0 was found
    cout << "\\0 is at index: " << nullIndex << endl;
```

```
    // print the string backwards
    for (int i = nullIndex - 1; i > -1; i--){
        cout << word[i];
    }
    cout << endl;

    return 0;
}
```

The output produced by the program is:

```
Enter a word (14 characters max or bad things may happen): hello
mech215
hello
word[0] = h    104
word[1] = e    101
word[2] = l    108
word[3] = l    108
word[4] = o    111
word[5] =      0
word[6] =      0
word[7] =      0
word[8] =      0
word[9] =
               12
word[10] = @   64
word[11] =     0
word[12] =     0
word[13] =     0
word[14] =     0
\0 is at index: 5
olleh
```

There are a couple of things to note from this output. The user entered "hello mech215" but only "hello" was taken as cin will parse the input on white space and take the first word. However, it does not read the input until the user presses the Enter key. At index 5 a null character (\0) was inserted by cin. However, there are still values in index 6 to 14 that have values. Some of these just happen to be zero. However, at index 9 and 10 there were two none zero values that were left over from the memories previous usage. The last column is the ASCII code of the character. h is represented by 104. At index 9 the code is 12 which is the code for form feed. Meaning, advance the output by one line and that is why there is a vertical space in the output. At index 64 is the @ symbol because its ASCII code is 64. The output you get will vary as it is based on what junk was left in the memory the last time it was used.

Passing Arrays To Functions

Consider the following program which passes an array of 5 elements to to a function. The function simple sets the value of all the array elements to 0 and returns back to the calling program:

```
// Author: Ted Obuchowicz and John Peach
// zero.cpp
// example program illustrating how arrays are
// passed to functions

#include <iostream>
#include <string>

using namespace std;

void howAreArraysPassed(int param[], int size_of_array);

int main() {

    // declare and initialize an array of 5 integers
    int size    = 5;        // size of the array
    int array[] = {100, 100, 100, 100, 100};

    // Write out the array
    cout << "Before function: " << endl;
    for(int i = 0; i < size; i++) {
        cout << "array[" << i << "] = " << array[i] << endl;
    }

    // pass the array to the function for mutation
    howAreArraysPassed(array, size);

    // Write out the mutated array
    cout << "After function: " << endl;
    for(int i = 0; i < size; i++) {
        cout << "array[" << i << "] = " << array[i] << endl;
    }

    return 0;
}

void howAreArraysPassed(int param[], int size) {
    for (int i = 0; i < size; i++) {
        param[i] = 0; // set all the elements to 0
    }
}
```

The program output may be surprising to a few readers:

```
Before function:
array[0] = 100
array[1] = 100
array[2] = 100
array[3] = 100
array[4] = 100
After function:
array[0] = 0
array[1] = 0
array[2] = 0
array[3] = 0
array[4] = 0
```

AHHHH!!!! the values of the array elements in the main program were actually changed by the function! We have already seen that C++ always passes by value and that if we change the value in a function that value is not changed in the calling function. So, what happened here? C++ is passing the memory address of the array to the function. The function can then directly access the elements in the array and change them. If we changed the value of param in howAreArraysPassed it would not affect the value of array.

It would be inefficient to pass an array using thr call-by value mechanism (imagine copying a 1 000 000 element array into a function's stack frame every time to function is invoked, this would have dramatic effects on a program's run time and memory requirements). In C++ (and in good old C) ARRAYS ARE ALWAYS PASSED AS REFERENCE PARAMETERS. NO & IS NEITHER NEEDED NOR EXPECTED IN THE ARRAY PARAMETER DEFINITION.

A few astute readers may have noticed that in the definition of the array parameter to the function no array size was specified. C++ allows the size to be left out so that the function may work with array arguments of different size. Of course, a second integer parameter is passed to the function idnicating the actual size of the array which the function will be working with during the course of its execution.

Also note how the fuction was invoked:

```
howAreArraysPassed(array, size);
```

We simply put the variable name "array" as a parameter. However, in the definition of the function there was a different notation:

```
void howAreArraysPassed(int param[], int size) {
```

Observe the [] after the parameter name. That lets C++ know that an array is being passed. Another popular notation is:

```
void howAreArraysPassed(int* param, int size) {
```

note that the [] has been removed and an * has been put after int. That tells C++ that param is a pointer to a memory address. Since an array is simply a pointer to a memory address then they are the same

thing. Later we will learn that pointers are much more then arrays.

Searching A Non-Sorted Array (LINEAR Search)

Suppose we wish to search an unsorted array to see if it contains a particular element (the key). If the element is in the array, we want to (first) index of where it occurs, otherwise a message saying the element is not contained in the array is to be displayed. We can perform a LINEAR SEARCH on the array starting from the first array element and comparing it with the key, we keep on comparing successive array elements with the key until we either find a match or reach the end of the array. This is known as a LINEAR SEARCH of an array.

Here is an implementation of the linear search in C++:

```
// Author: Ted Obuchowicz and John Peach
// linearSearch.cpp
// example program illustrating use of
// linear search on an array to find a key

#include <iostream>
#include <string>

using namespace std;

int main() {

    // declare and initialize an array of 10 integers
    int size   = 10;        // number of elements in data
    int data[] = {12, 234, 23, 1, -7, 55, 18, 67, 99, 100};
    int keyValue;           // the value to search for
    int keyIndex = 0;       // index of the key or search position
    bool keyFound = false;  // Has the key been found

    // Ask the user for the search value
    cout << "Enter the key value to search for: ";
    cin >> keyValue;

    // Find the keyValue
    while (!keyFound && keyIndex < size) {
        if (data[keyIndex] == keyValue) {
            // The key was found
            keyFound = true;
        } else {
            keyIndex++; // increment the search postition
        }

    }
```

```
    // check to see if the keyValue was found
    if (keyFound) {
        cout << "Key found in position " << keyIndex << endl;
    } else {
        cout << "Key is not in the array." << endl;
    }

    return 0;
}
```

The output is:

```
Enter the key value to search for: 55
Key found in position 5

Enter the key value to search for: -99
Key is not in the array.
```

We can employ recursion to perform a linear search of an unordered array. We simply start from the end of the array, if the last element of the array matches the key, the index of the key is returned. If there is no match we make a recursive call with the remaining elements of the array. That is, we act as if the array is one element smaller. Eventually, all the elements will be compared one-by-one with the key resulting in either a match or the key not being found. If all elements in the array have been checked then return -1 for the index. This sentianal value can be used as it is not a valid index.

```
// Author: Ted Obuchowicz and John Peach
// linearSearchRecursive.cpp
// example program illustrating use of linear search
// using recusion

#include <iostream>
#include <string>

using namespace std;

int linearSearch(int* array, int key, int size);

int main() {

    // declare and initialize an array of 10 integers
    int size   = 10;            // size of data
    int data[] = {-56, 0, 5, -234, 1, 12, -3, 678, 34, 100 };
    int keyValue;               // key to search for
    int keyIndex;               // index of the key or -1 not found
```

```
    // Get the key to search for
    cout << "Enter the value to be searched for:  ";
    cin >> keyValue;

    // Search for the key and report
    keyIndex = linearSearch(data, keyValue, size);
    if ( keyIndex < 0 ) {
        cout << "Key not found" << endl;
    } else {
        cout << "Key found at position " << keyIndex << endl;
    }

    return 0;
}

int linearSearch(int* array, int key, int size){
    if (!size) {
        // base case
        return -1;
    } else {
        if ( array[size-1] == key ) {
            return size-1;
        } else {
            return linearSearch(array, key, size-1);
        }
    }
}
```

The output of the program is:

```
Enter the value to be searched for:  12
Key found at position 5

Enter the value to be searched for:  -99
Key not found
```

Sorting An Array Using Bubble Sort

Sorting a set of data into some numeric order (increasing order or decreasing order) is a commonly required task in data processing. Here is a C++ program which performs a LINEAR SORT on an array of 10 elements:

```
// Author: Ted Obuchowicz and John Peach
// bubbleSort.cpp
// example program illustrating use of linear sort on
// an array to sort it into ascending order
```

```
#include <iostream>

using namespace std;

void printArray(int data[], int size);
void bubbleSort(int* data, int size);
void swap(int* first, int* second);

int main() {

    // declare and initialize an array of 10 integers
    const int size = 10;                // size of data
    int data[size] = {12, 234, 23, 1, -7, 55, 18, 67, 99, 100};

    // Print the unsorted data
    cout << "The unsorted array is: " << endl;
    printArray(data, size);

    // sort
    bubbleSort(data, size);

    // Print the sorted data
    cout << "The sorted array is: " << endl;
    printArray(data, size);

    return 0;
}

void swap(int* first, int* second) {
    int temp;       // Used in swapping values
    temp    = *second;
    *second = *first;
    *first  = temp;
}

void bubbleSort(int* data, int size){

    // now sort the array in place
    for(int i = 0; i < size; i++) {

        for(int j = i + 1; j < size; j++) {
            // bubble the value up the array
            if (data[j] < data[i]) {
                swap(&data[i], &data[j]);
            }
```

```
        }
    }
}

void printArray(int data[], int size){
    for(int i = 0; i < size; i++) {
        cout << data[i] << "\t";
    }
    cout << endl;
}
```

The program output is:

```
The unsorted array is:
12   234  23   1    -7   55   18   67   99   100
The sorted array is:
-7   1    12   18   23   55   67   99   100  234
```

The swap function takes two memory addresses and then swaps the values at those memory addresses. The second line:

```
temp    = *second;
```

Uses the dereference operator, *, to get the value of the integer stored in the memory location given by second. The value is then stored in temp.

The third line:

```
*second = *first;
```

is interesting as we actually perform an operation on the left side of the equals sign. Using the * operator on the right side, we get the value at the memory location given by first. The * on the left hand side of the equality operator says, store the value in the memory address given by second. Normally the left hand side of the equality operator cannot have an operation. However, the reference operator, &, and the dereference operator, *, are allowed.