This lesson introduces the concept of templated functions. These allow you to write a function that takes parameters of an unspecified type, and possibly even returns an unspecified type. For example, the function below will take a value of any data type (character, integer, double, user defined structure, etc), run that data type's addition function, and return the result as that data type.

template <class asd> asd increment(asd in){return in+1;}

An array is declared in the following way: the data type is first, as with declaring any variable, followed by the variable name, and then the number of elements in square brackets. In C++, arrays declared this way must be of a constant size because the compiler has to know how much space to allocate. Therefore, we can't do something like have the user enter a number, and then make an array that big. We will look at how to dynamically allocate memory in C++ later. When an array is declared, it can be filled with values separated by commas between curly brackets ({}) as shown below.

int inta[10]={0,1,2,3,4,5,6,7,8,9}; //make integer array

Now we want to see what is in this array. We happen to know right now, but it's always a good idea to verify that everything is working the way you think it is. Since we can use templated functions, we can write a single display function to handle any array type. We will also write this function for arrays of any size. When an array is passed to a function, the size can be specified, as in FunctionName(arraytype a[10]), or left unspecified, by using the square brackets with nothing inside, as in FunctionName(arraytype a[]). Since we are passing arrays of unspecified size, we will include a second parameter to the function for the size of the array, called n.

In C++, arrays start at an index of 0 (as opposed to MATLAB arrays, which start at 1), so we will have a loop that displays the values of the array from 0 to n-1. The way to access an element of an array, called indexing the array, is with the array name followed by the desired index in square brackets. Therefore array[0] will be the first element of an array, and array[n-1] will be the last element of an array of size n. Given all of this, the array display function is shown below.

//display function for any array
template <class asd> void arraydisplay(asd in[], int n) {for (int ii=0;ii<n;cout <<in[ii]<<", ",ii++); cout <<endl;}

Next we will declare a second array of the same size, and store the squares of the elements of the first array into it:

int intb[10];
for (ii=0;ii<10;intb[ii]=inta[ii]*inta[ii],ii++); //demo use of loop with array

Now that you know how to do basic things with arrays, we will move on to pointers. A pointer is a variable that stores a memory location. Since every variable is stored somewhere in memory, we only need to know the address of that memory to read it. The memory address of a variable can be accessed by &var, where var is the variable name. A pointer is created by first specifying a data type, then a * followed by the variable name. Here we create a variable, “a” with a value of 56, and then create a pointer to an integer, into which we store the address of “a”, &a. Therefore, pa is now a pointer to a.

This memory location can now be displayed by looking at either pa or &a. Similarly, the value stored at that memory location can be accessed by either “a” or “*pa”.

An array is actually a pointer to the memory location where the first element of the array is stored, and all other elements of the array are stored in order after it. Therefore we can treat a pointer as an array with at least one element. Here we can also access the data stored in “a” with “pa[0]”. We can try to access pa[1], which is the data stored immediately after “a” in memory. This may or may not cause the program to crash depending on whether the program has access to that memory or not. It is important to keep careful track of what memory you are using when doing this to make sure you don't try to access an unavailable memory location, or even overwrite another one of your own variables.

Since arrays and pointers are the same thing, we will now point “pa” at the array “inta” we created before. We see that not only are “inta” and “pa” both pointers containing the same memory address, and inta[0], and *pa both access the first element of the array, but pa[0] and *inta also access that element. pa[ii] can now be used to access any element of the array. *(inta+ii) can also be used to get the ii-th element of the array, though you probably shouldn't.

Double pointers also exist: they are the same as pointers in that they contain a memory address, they just happen to contain a memory address that contains another address (a common application will be shown later). Declaring b as a double pointer, as performed in the code above, we can give “b” the memory location where “pa” is stored, &pa. Now *b is a pointer to the same location as pa, and **b is the value of “a”. As before, square brackets can be used instead, only this time they are placed in series. b[0] is pa, and b[0][0] is *pa.

Next we look at 2D arrays (matrices). Declaring a variable with 2 sets of square brackets will create an array with two dimensions. This is similar to, but not exactly a double pointer. For int mat[n][m], mat[ii] is a pointer to every mth element of mat. Since in mathematics, the row index is listed first, this
would mean that the rows are stored contiguously, and sequentially in memory, i.e.

```
row1,col1 | row1,col2 | row1,col3 | row2,col1 | row2,col2 | row2,col3 | row3,col1 | row3,col2 | row3,col3
```

Since most other programming languages, such as FORTRAN store matrices by column, many people choose to handle matrices in C++ backwards, with the column index first, or to just use a one dimensional array and index it with `array[(number of columns)*(column index)+(row index)]` so each column is stored contiguously.

You may have noticed that this file began with

```c
void main(int num, char *vals[])
```

as the declaration for its main function. It was actually given parameters corresponding to a number and a pointer to an array of unspecified size (which is a double pointer). This is how parameters are passed to the program through the command line. We can see them using the following:

```c
for (ii=0;ii<num;cout<<ii<<": "<<vals[ii]<<endl,ii++);
```

Unlike integers and doubles, pointers to characters in C++ are displayed as strings instead of memory addresses. Because each `vals[ii]` is an array of characters, i.e. a string, this loop displays each parameter passed to the program (num contains the number of parameters).

```c
cout << vals[0][5]<<endl;
cout << vals[num-1][1]<<endl;
```

We can access each character in each string by indexing it as shown above. The first line prints the 6th character of the first string, and the second line prints the second character of the numth (last) string (remember that arrays start at 0). To see this work, rather than running the program from here, we must open a command prompt, and navigate to the folder where the executable is stored. The easiest way to do this is to find the executable in windows, copy the path from the address bar, then open a command prompt, type “cd” and then paste that path in and press enter. Now running the executable (mine is named lesson5.exe) with parameters, e.g. “lesson5.exe hello world” will display each of those three words along with “n” and “o”.

```c
if (num==2)
{
    double d=atof(vals[1]);
    cout << d<<endl;
}
```

The last lines of this program are shown above. If only one parameter is passed to the program (making two parameters including the name of the program), it assumes it is a number and converts it from a string to a double using the “atof” command. Try this a few times, e.g. “lesson5 12.5” or “lesson5 -32.04”.

C++ Programming Applied to Robotics, Mark Aull
Lesson 5b: Multithreading

Since some operations, like reading from a serial port, cause the program to wait for something to happen, and we never want a program controlling a robot to hang, we need a way for these functions to run separately. Multithreading is one answer (another for example is parallel processing). To implement it, we write a function, and then instead of calling it normally, we start it in its own thread using the “CreateThread” command.

This command requires the function to have a very specific format:

```cpp
long WINAPI ThreadFunction(some kind of pointer);
```

so we create one, and pass it to “CreateThread”. The CreateThread function returns a “Handle” to the thread, which can be used to monitor it later on. The function I have provided takes an array of doubles, and uses the first to identify itself as thread1, thread2, etc. Next the thread runs a loop counting backward from the second element in the array to zero. There are lines doing this with both printf and cout (the cout line is commented out). Running the printf line shows that the threads alternate counting down to zero. Running the cout line is more interesting: the lines being displayed from the different threads actually interrupt each other. If the printf line is run without the cout<<endl line, each thread actually finishes printing its information before the next thread starts because printf runs faster than cout.

Assignment 2:

Part 1
Modify the code provided to access the digital compass convert the string coming from the serial port to numbers using your understanding of arrays and strings and the atof command. Display each value from -180deg to 180deg (instead of from 0 to 360 deg).

Part 2
Run the code from part 1 in a separate thread in a code that can operate the Galil. I recommend sending the thread an array with two values: the first being a control value so you can exit the loop, and the second being a place where the thread can place each value from the compass so the main program can access it. The main program should start with a prompt asking the user to enter a number:

```cpp
double goal;
cout << "Enter the desired angle: ";
cin >> goal;
```

Then set the Galil to rotate slowly (JG 1000,1000 max) to that heading, using the information from the compass.
Appendix A: code for lesson5 part 1

```cpp
#include <iostream>
using namespace std;

//display function for any array
template <class asd> void arraydisplay(asd in[], int n) {
    for (int ii=0; ii<n; cout << in[ii] <<", ", ii++); cout << endl; }

void main(int num, char *vals[])
{
    int ii, jj;
    int inta[10] = {0,1,2,3,4,5,6,7,8,9}; //make integer array
    int intb[10];
    arraydisplay(inta, 10);
    for (ii=0; ii<10; intb[ii]=inta[ii]*inta[ii], ii++); //demo use of loop with array
    arraydisplay(intb, 10);

    int a=56; //make a regular integer
    int *pa=&a; //point pa at that memory location
    cout << endl << &pa << endl << &a << ' ' << pa << endl << ' ' << *pa << ' ' << pa[0] << endl; //look at memory locations and values associated with this location
    cout << pa[1] << endl;

    pa=inta; //make pointer, pointing to first element of inta
    cout << endl << inta << ' ' << *pa << ' ' << **pa << ' ' << b[0][0] << endl << endl;

    int **b; //double pointer
    b=(int**)&pa;
    cout << endl << b << ' ' << *b << ' ' << **b << ' ' << b[0][0] << endl << endl;

    int c2[3][3]; //2D array
    for (ii=0; ii<3; ii++)
        for (jj=0; jj<3; jj++)
            c2[ii][jj]=3*ii+jj;
    for (jj=0; jj<3; jj++)
        for (ii=0; ii<3; cout << &c2[jj][ii] << ' ', ii++);
    cout << endl;

    for (ii=0; ii<num; cout << ii << " : " << vals[ii] << endl, ii++)
        cout << vals[0][5] << endl;
    cout << vals[num-1][1] << endl;
    if (num==2)
    {
        double d=atof(vals[1]);
        cout << d << endl;
    }
}
```
Appendix B: code for lesson5 part 2

```c
#include <windows.h>
#include <stdio.h>
#include <iostream>
using namespace std;

long WINAPI ThreadFunction(double lParam[]);

void main(void)
{
    HANDLE hThreads[3];
    double data1[2],data2[2],data3[2];
    data1[0]=1; data2[0]=2; data3[0]=3;
    unsigned long dwID;

    hThreads[0] = CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)ThreadFunction,LPVOID(&data1),0,&dwID);
    hThreads[1] = CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)ThreadFunction,LPVOID(&data2),0,&dwID);
    hThreads[2] = CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)ThreadFunction,LPVOID(&data3),0,&dwID);
    WaitForMultipleObjects(3,hThreads, true,1000);
    CloseHandle(hThreads[0]);
    CloseHandle(hThreads[1]);
    CloseHandle(hThreads[2]);
}

long WINAPI ThreadFunction(double lParam[])
{
    for (double ii=lParam[1];ii>0;cout<<"Thread"<<lParam[0]<<": ",printf("Threa%d.0f: %.0f",lParam[0],ii),cout<<endl,ii--);
    return 0;
}
```