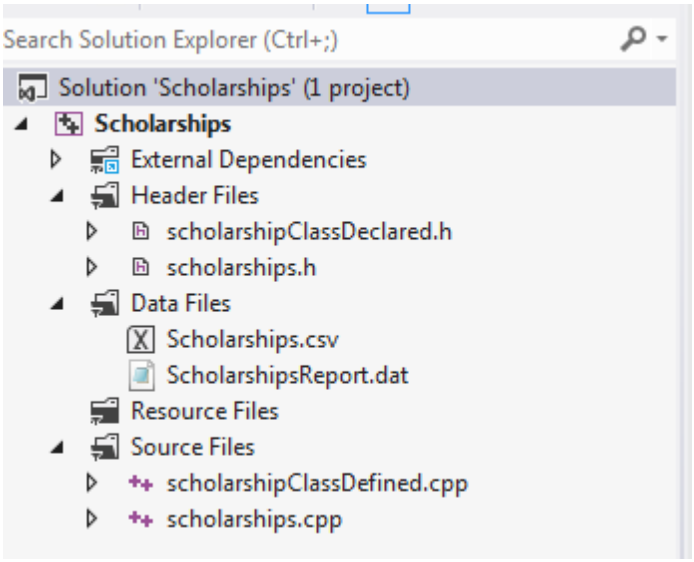
Pointers Tutorial Part 1 – Scholarships

This tutorial demonstrates how to use pointers. It uses the Scholarship vector array of objects that you have seen in the past tutorials. DO NOT USE a previous project. Create a NEW one and use the NEW files provided in the download. DELETE the OLD project if you have to before creating the new one below. This new project includes the BubbleSort solution presented at the end of the last tutorial.

NOTE: If you find errors when working this tutorial, do NOT start adding or deleting your *own* code to fix it. This tutorial *does* work when done correctly.

1. Create an empty Visual Studio ***Win32 Console Application*** on the **DESKTOP** named ***Scholarships***.
2. Download Search and Sort Tutorial ZIP file from Blackboard>Lesson 6 and save to your computer.
3. Using Windows Explorer, COPY all of the of the files from the ZIP file into the new project's folder located at **…Desktop\Scholarship\Scholarships**
4. With the new project open in *Visual Studio*, add the files to the different filters of the ***Solution Explorer*** panel so that your project looks like the following screen shot. You will need to add the ***Data Files*** filter to add the scholarships.csv file to the project in the correct location as shown.



The Address Operator (&)

5. You should have read about the Address Operator in your chapter reading assignment. It simply returns the Address in RAM that the variable is located. This can be used to show what a pointer is actually pointing to. Placing the & in front of any variable gives you the address of its location in memory. This value is what is "stored" in a pointer variable. Pointer variables must be the same type as the variable they are pointing at. **Add** the highlighted code below to your main() function to see how this works. Place it between the call to the summary report and the return at the end.

```
//  file handle to the function
createReportSummary(scholars, fout);

  cout << "Address of the scholars array in hexidecimal is:  x"
       <<  &scholars << endl << endl;


return 0;
```

6. **Run** the program and you should see a display of the address where the *scholars* array begins. It is displayed in *hexadecimal* numbers (base 16), which is normally what memory addresses are displayed as. *Note: If you are not sure what hexadecimal numbers are, be sure to Google it later. Understanding this numbering system is VITAL to your programming career.*
7. Now, **add** the code below just under the statement you just typed. You will notice you do not use the & with the reference to "fout." This is because "fout" is a pointer and only holds the "address" of where the output file handler begins.

```
cout << "Address of the output file in hexidecimal is:     x"
   <<  fout << endl << endl;
```

8. **Run** the program and note the address returned.

Since "fout" is not the file handler itself, but simply the "pointer" to the file hander, let's see exactly where the "pointer" variable is located. **Add** the following code just below the last statement you typed. Note the & added to "fout."

```
cout << "Address of the pointer variable that is pointing " << endl
     << "to the output file handler in hexidecimal is:     x"
     <<  &fout << endl << endl;
```

9. **Run** the program and notice that the address of the "fout" pointer variable is different from the actual file handler. The value of the "fout" pointer is simply the address. "fout" itself simply holds an integer value, which is the address of the handler. This is all a pointer is in any situation. A pointer holds the address (a long integer) of what it is pointing to.

10. **Your turn**: Write another line of code, as you just did, that will return the address of the *sArraySize* variable to the console for display.
11. **Run** the program and compare the addresses returned by YOUR computer with that of the person next to you. Discuss between the two of you why they may be different from yours.

Pointer Variables

12. **Add** the following code below the statement you just typed.

```
int *ptr;     // Pointer variable, can point to an int
ptr = &sArraySize;     // Store the address of sArraySize in ptr
cout << "The value in sArraySize is: " << sArraySize << endl;
cout << "The address of sArraySize in hex is:              x"
       << ptr << endl;
```

13. **Run** the program and compare the address returned previously for *sArraySize* and the address given by the *ptr* variable. Here, you created a pointer variable named *ptr* and gave it the address of *sArraySize*. The variable *ptr* now contains the address of sArraySize. This is like giving a variable 2 names or handles.
14. You can gain access to the variable that the pointer is pointing to by using the Indirection Operator (*) with the pointer name. Add the following code below and run the program to see how this works.

```
cout << "The value in sArraySize is: " << *ptr << endl;
*ptr = 100;
cout << "The value in sArraySize is: " << *ptr << endl;
*ptr += 1;
cout << "The value in sArraySize is: " << *ptr << endl;
```

Pointers and Arrays of Objects

15. Let's take it up a notch by making a pointer to an object. **Add** the following code below the statement you just typed.

```
Scholarship *sPtr;   // Pointer variable, can point to a Scholarship oject
sPtr = &scholars[0]; // Store the address of scholars' 1st element in sPtr
cout << endl << "The value in scholars's 1st record is: "
       << scholars[0].getID() << endl;
cout << "The address of scholars's 1st record in hex is:   x"
       << sPtr << endl;
```

16. **Run** the program and compare the address returned previously for the *scholars* array at the top and the address of its 1st element at the bottom of your output. Since the first element of an array is at the beginning, you though think these two would be the same. For a "normal" array, they would be. However, since *scholars* is a "*vector*" class container, it is *not* true for this instance. *Vectors* create arrays dynamically

"on the fly," so to speak, and therefore have an array *inside* the *vector*. *sPtr* is pointing to the *vector* "container" and NOT pointing to the actual *array* that it holds.

17. **Run** the program several times. Notice the different addresses that are displayed with each run. This is because we never know where a program will be launched by the operating system into memory. Pointers become valuable when working with dynamically allocated variables (using the new keyword). We will discuss this in the next tutorial.
18. **Add** the following code below where your last statement typed.

```
// make a non-vector array and copy 10 scholars into it
Scholarship sArray[10];
for(int i = 0; i < 10; i++)
      sArray[i] = scholars[i];
cout << endl << "The value of sArrays's 1st record in hex is: "
       << sArray[0].getID() << endl;
cout << "The address of sArray's beginning in hex is:      x"
       << sArray << endl;
cout << "The address of sArray's 1st record in hex is:     x"
       << &sArray[0] << endl;
```

19. **Run** the program and note that the *name* of the non-vector array holds the address of the array's first element as shown by *&sArray[0]*. Therefore, the name of a non-vector array is a pointer to its first element.
20. **Add** the following code below where your last statement typed.

```
// Point the sPtr to the new array by reasigning it
sPtr = sArray;  // pointers can be reasigned as needed
cout << "The address of sArray's 1st record in hex is:     x"
       << sPtr << endl;
```

21. **Run** the program and note that the *sPtr* variable now contains the address of the *sArray* beginning, which is the address of its first element. Compare the last three addresses in your output. If you run the program several times, these addresses change since we do not know where the program will be loaded by the operating system in memory.
22. **Add** the following code below where your last statement typed.

```
// Use the indirection operator to gain access to the
//   contents of sArray
cout << endl << "The value of sArrays's 1st record in hex is: "
       << sPtr[0].getID() << endl;
```

23. **Run** the program and note the output. You can use the pointer just like the *sArray's* name because it points to the same location that *sArray* does.

24. You can also use the following "pointer" notation for the first element. Add the code below and run the program.

```
cout << endl << "The value of sArrays's 1st record in hex is: "
     << sPtr->getID() << endl;
```

Pointers and Functions

You have already been using a *type* of pointer for functions on occasion, that being the *reference* variable. When the overloaded & is used to pass a variable into a function, such as in the function below

```
void bubbleSort(vector<Scholarship> &s)
```

"s" is in reality a pointer that does not need the (*) indirection symbol to access the original array. This is the best way to pass *vectors* into functions. Reference variable cannot be reassigned, however, like pointer variables can.But what if you wanted to pass a pointer to something else into a function and needed the functionality of a pointer?

25. **Copy** the *bubbleSort*() function to the end of your program file and make the modifications below to make a new overloaded the function.

```
void bubbleSort(Scholarship *s, int iSize)
{
   Scholarship temp;    // Holds Scholarship object
   bool swap;

   do
   {  swap = false;
      for (int count = 0; count < iSize-1; count++)
      {
         if (s[count].getLname() < s[count + 1].getLname())
         {
            temp = s[count];
            s[count] = s[count + 1];
            s[count + 1] = temp;
            swap = true;
         }
      }
   } while (swap);
}// End bubbleSort
```

26. **Add** the code below to the end of your main() function, but before the return 0; line.

```
bubbleSort(sArray, 10); // Call the overloaded bubbleSort using a pointer
fout.close();
fout.open("sArrayReport.dat");
if(!fout)
{
      cout << "Output file did not open. Program will exit." << endl;
      exit(0);
}
createReportHeadings(fout);
for(int i = 0; i < 10; i++)
      writeFile(sArray[i], fout);         // Write a line to the output file
```

27. **Run** the program, then add the sArrayReport.dat file to your project and open it. Consider the results.