Class 3 Handout

Review

Last time we talked about **if statements**, **while loops**, and more about **class objects**. We created a Car that would drive forward and backward once turned on and shifted into gear.

If Statements take the form:

```
if ( condition )
{
}
```

You can also have *else if* and *else*cases tagged on to the end.

While Loops take the form:

```
while ( condition )
{
}
```

And Class declarations look like:

```
class Name
{
    public:
    // Anything outside the clas scan view these items

    private:
    // Only the class can view these items
};
```

The class **declaration** should go in a header file (Name.h), while the function **definitions** for functions belonging to this class go in a source file (Name.cpp).

Look at the handouts for Class 1 and Class 2 for more about classes.

Loops and Arrays

Loops are also very useful if we have an **array** of data. An array is essentially a list of data, where everything is the same data type. For example:

```
Kid myKids[7];
```

With this statement, somebody has written a class called **Kid**, and we're declaring that we have an array of 7 kids.

This way, we don't have to store them all as separate variables, which would get unruly very fast:

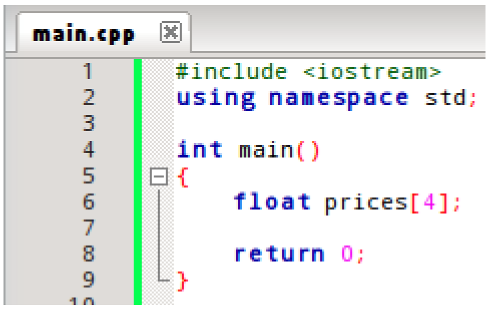
```
Kid sarah;
Kid shannon;
Kid angela;
Kid luke;
Kid josh;
Kid adam;
Kid rachel;

sarah.DrawToScreen();
shannon.DrawToScreen();
angela.DrawToScreen();
... etc ...
```

If we store the kids in an array (don't worry, it's humane), we can loop through them all when we're doing the same actions.

```
for each kid in "myKids" array:
        kid.EatFoodInFridge();
        kid.DrawToScreen();
```

So, what would this look like in code?

Let's start by storing **floats**, rather than having to write a Kid class.
This time, we're going to be storing an array of prices:



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

int main()
{
    float prices[4];

    return 0;
}
```

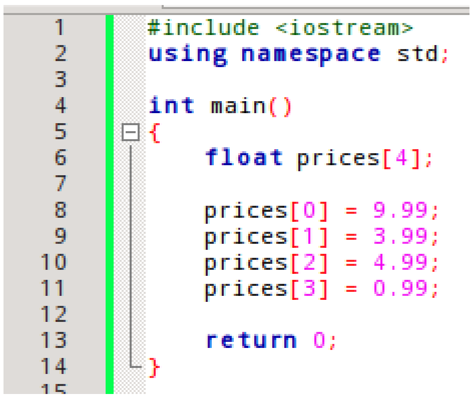
Creating an array is similar to creating a variable.

If we were just storing one price, we would use

```
float price;
```

but with an array, we add square-brackets, and the size of the array inside.

Now, we can initialize what these prices are by using a **subscript**.



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

int main()
{
    float prices[4];

    prices[0] = 9.99;
    prices[1] = 3.99;
    prices[2] = 4.99;
    prices[3] = 0.99;

    return 0;
}
```

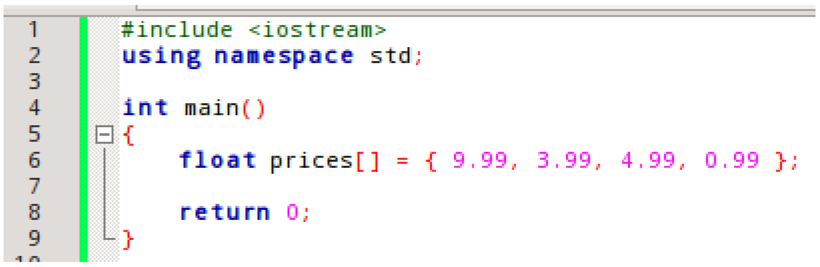
Again, we use square brackets, and we use the **index** of the item we want.

In C++, arrays begin at the index of 0, rather than 1 like would happen in math.

Therefore, if we have exactly four items, our indices go from 0 to 3, rather than 1 to 4.

This is important, and can be confusing, so keep it in mind!

Additionally, if we wanted to set up all the values of the array when declaring it, we could do it this way:

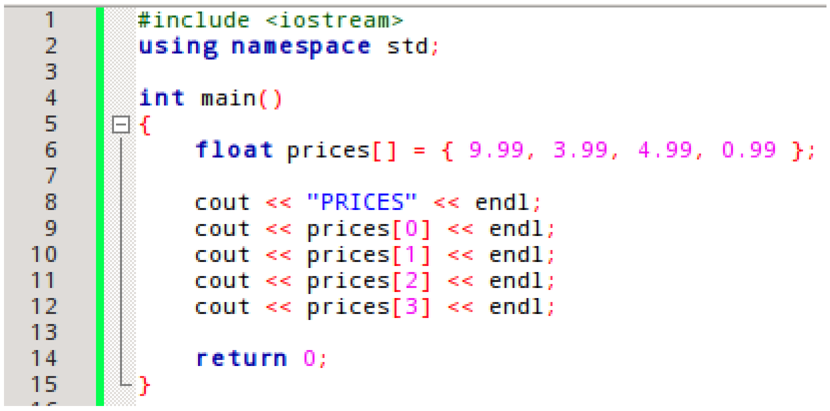


```
#include <iostream>
using namespace std;

int main()
{
    float prices[] = { 9.99, 3.99, 4.99, 0.99 };

    return 0;
}
```

Compare this to the previous example- when we declared the array, we specifically said that the array's size was [4]. We could do that here as well, but we could also leave it empty since we are immediately putting data in the array.

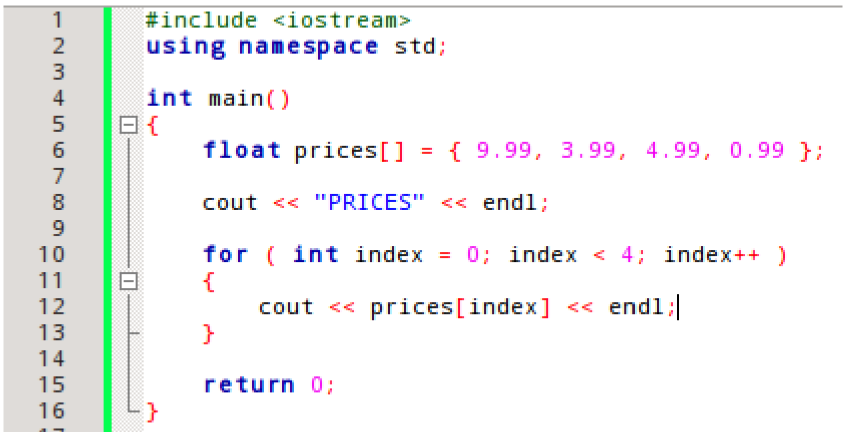
When we're initializing the array like this, use curly-braces, and separate items with commas.

Now, if we want to tell the user all of the prices, we could do it the long way...

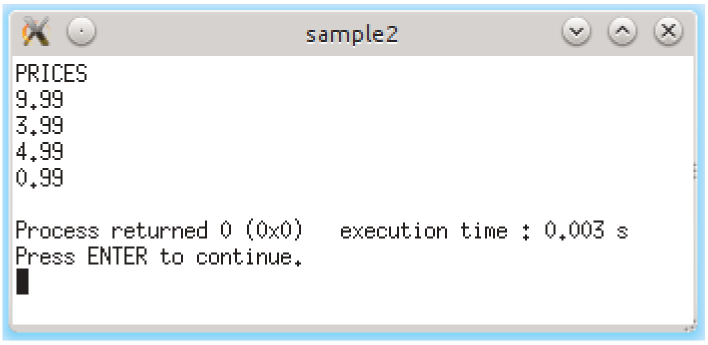


```cpp
#include <iostream>
using namespace std;

int main()
{
    float prices[] = { 9.99, 3.99, 4.99, 0.99 };

    cout << "PRICES" << endl;
    cout << prices[0] << endl;
    cout << prices[1] << endl;
    cout << prices[2] << endl;
    cout << prices[3] << endl;

    return 0;
}
```

Or we could use a **for loop**...



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

int main()
{
    float prices[] = { 9.99, 3.99, 4.99, 0.99 };

    cout << "PRICES" << endl;

    for ( int index = 0; index < 4; index++ )
    {
        cout << prices[index] << endl;
    }

    return 0;
}
```



```
PRICES
9.99
3.99
4.99
0.99

Process returned 0 (0x0)   execution time : 0.003 s
Press ENTER to continue.
```

A for loop is structured like:

```
for ( variable start value ; variable end value ; variable increment amount )
{
	// looping code
}
```

So what we have above translates to:

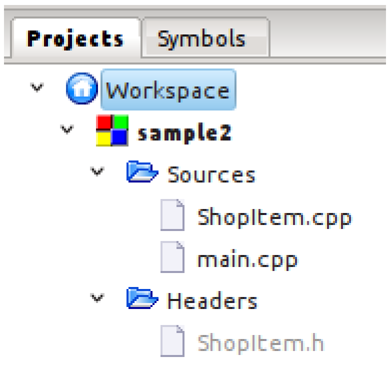
```
for ( int index = 0; index < 4; index++ )
{
    cout << prices[index] << endl;
}
```

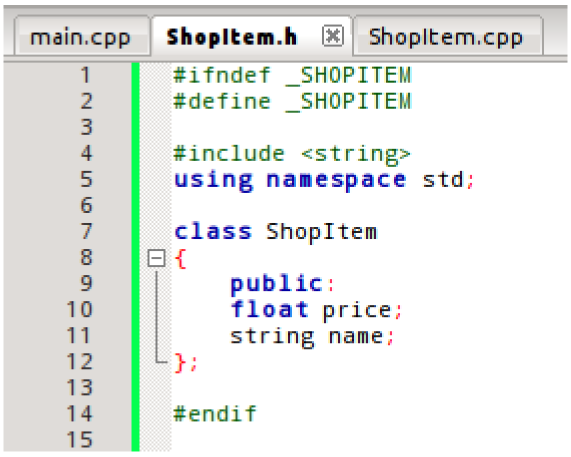
1	`int index = 0;`	Create variable "index", which is an int. Set it to 0.
2	`index < 4;`	Keep looping while the index is less than 4.
3	`index++`	Every loop, increment the index variable by 1.

With argument 3, we could add, subtract, multiply, or divide if we wanted to.

`index += 4;`
`index -= 4;`
`index *= 4;`
`index /= 4;`
	Add by 4
	Subtract by 4
	Multiply by 4
	Divide by 4

Now, let's put this into some use and create a simple "ShopItem" class. Create a corresponding .h and .cpp file for this class.



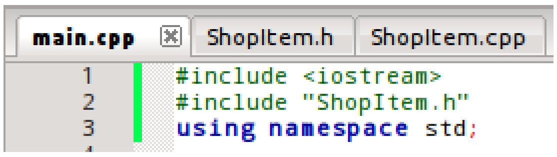


```
#ifndef _SHOPITEM
#define _SHOPITEM

#include <string>
using namespace std;

class ShopItem
{
    public:
    float price;
    string name;
};

#endif
```

In our class, we are storing **price** and **name**.
Notice on line 4 that we have to include the standard **string** library, but since we're not inputting or outputting to the console in this file, we're leaving out **iostream**.

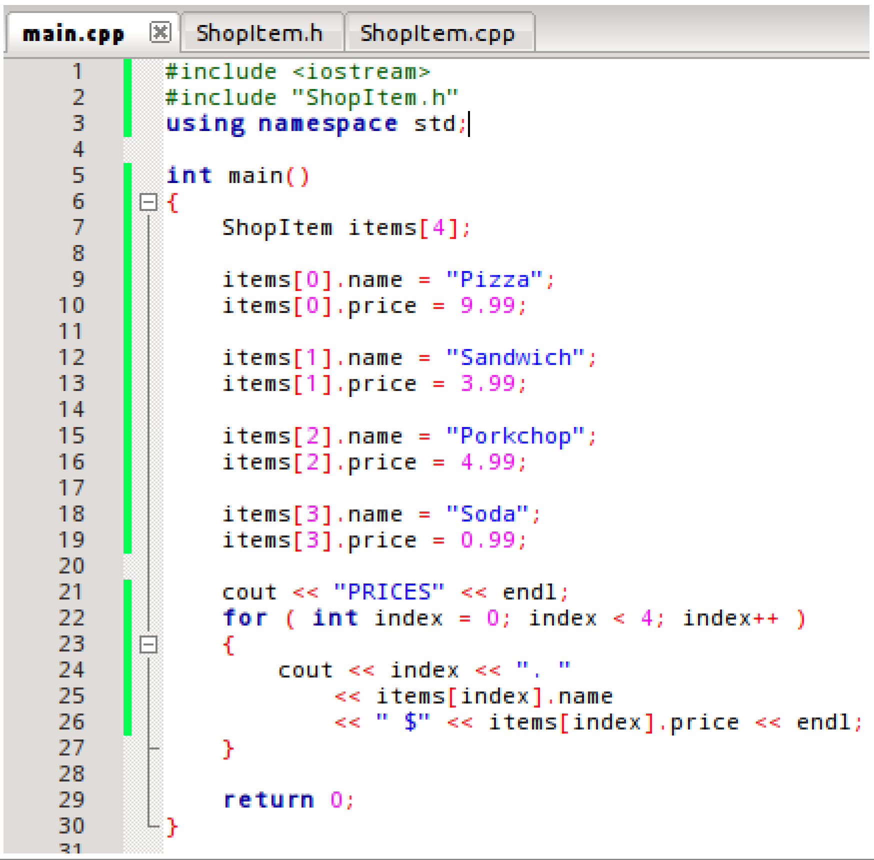
If we want to add functions later, we can create the **declaration** in the header file (.h), and we can put the function **definition** in the source file (.cpp).

Back to main!



```
#include <iostream>
#include "ShopItem.h"
using namespace std;
```

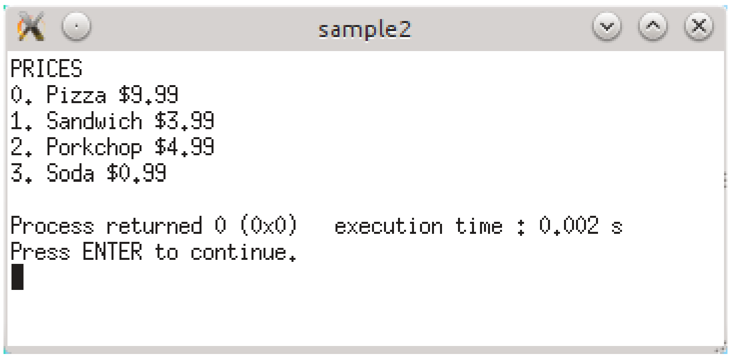
Include the ShopItem.h file, and then we can replace our current "price" array:


main.cpp ShopItem.h ShopItem.cpp

```
#include <iostream>
#include "ShopItem.h"
using namespace std;

int main()
{
    ShopItem items[4];

    items[0].name = "Pizza";
    items[0].price = 9.99;

    items[1].name = "Sandwich";
    items[1].price = 3.99;

    items[2].name = "Porkchop";
    items[2].price = 4.99;

    items[3].name = "Soda";
    items[3].price = 0.99;

    cout << "PRICES" << endl;
    for ( int index = 0; index < 4; index++ )
    {
        cout << index << ". "
            << items[index].name
            << " $" << items[index].price << endl;
    }

    return 0;
}
```

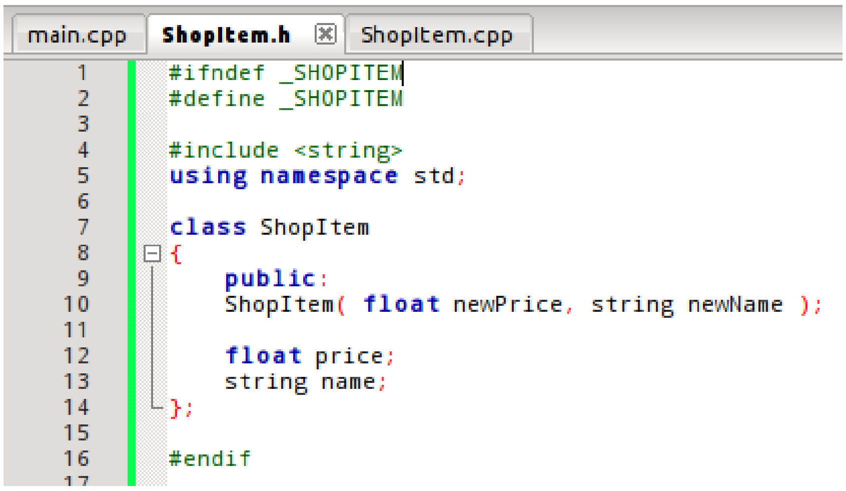
Now what's bloating our code is having to initialize the name and price of each item, which takes two lines per item.

If you run the program, we will at least get a nice list:


sample2

```
PRICES
0. Pizza $9.99
1. Sandwich $3.99
2. Porkchop $4.99
3. Soda $0.99

Process returned 0 (0x0)   execution time : 0.002 s
Press ENTER to continue.
```

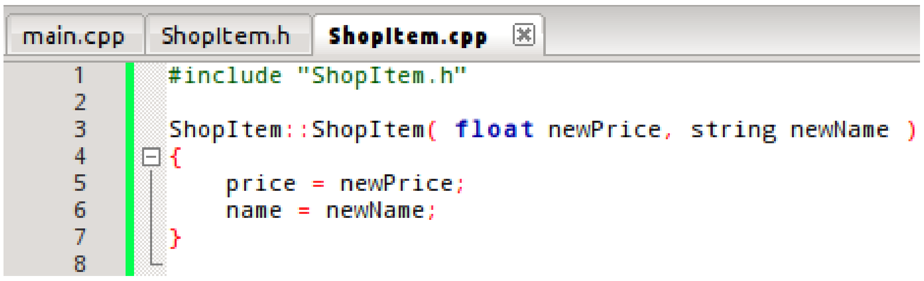
We can add a **Constructor** to our class. Constructors are similar to functions, except have no return-type and the name must match the name of the class. Usually constructors are used for initializing variables to a default value, but we can also pass in parameters (just like a function), to set the internal variables.



```
1   #ifndef _SHOPITEM
2   #define _SHOPITEM
3
4   #include <string>
5   using namespace std;
6
7   class ShopItem
8   {
9       public:
10      ShopItem( float newPrice, string newName );
11
12      float price;
13      string name;
14  };
15
16  #endif
17
```

On line 10 we add our constructor **declaration**. We have essentially declared the name and parameters (parameters being the variables listed within the parenthesis).

When we go to ShopItem.cpp, we will add the constructor's **definition**, or the function body.

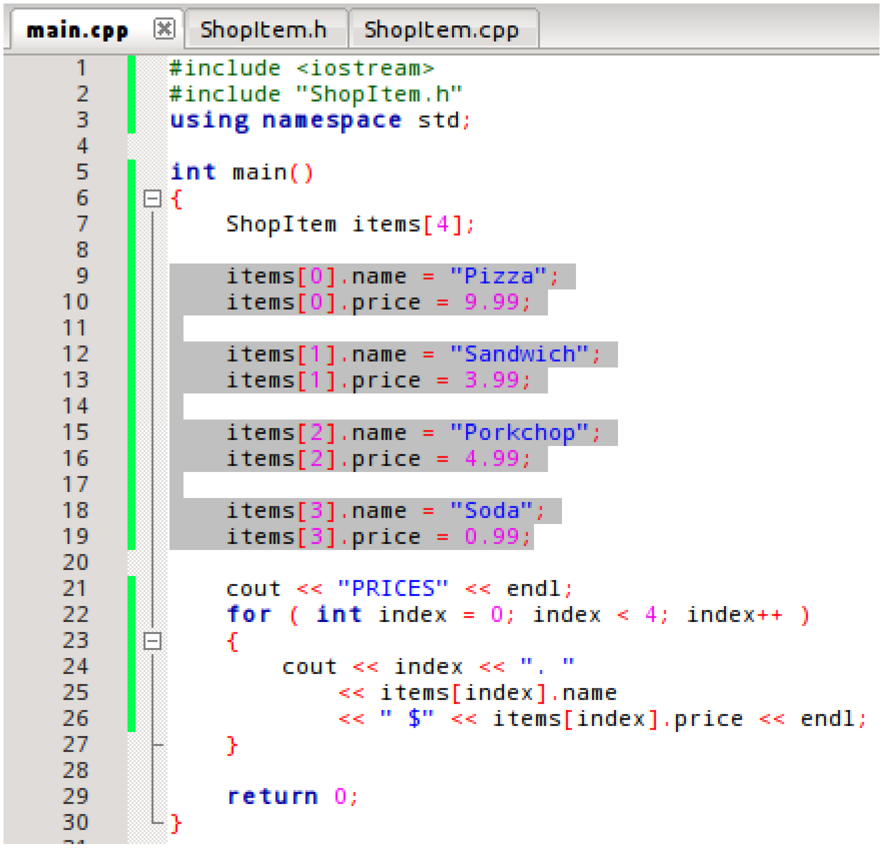


```
#include "ShopItem.h"

ShopItem::ShopItem( float newPrice, string newName )
{
    price = newPrice;
    name = newName;
}

```

Now we can initialize the internal **price** and **name** variables, simply by assigning them.

Then we can clean up main.cpp:

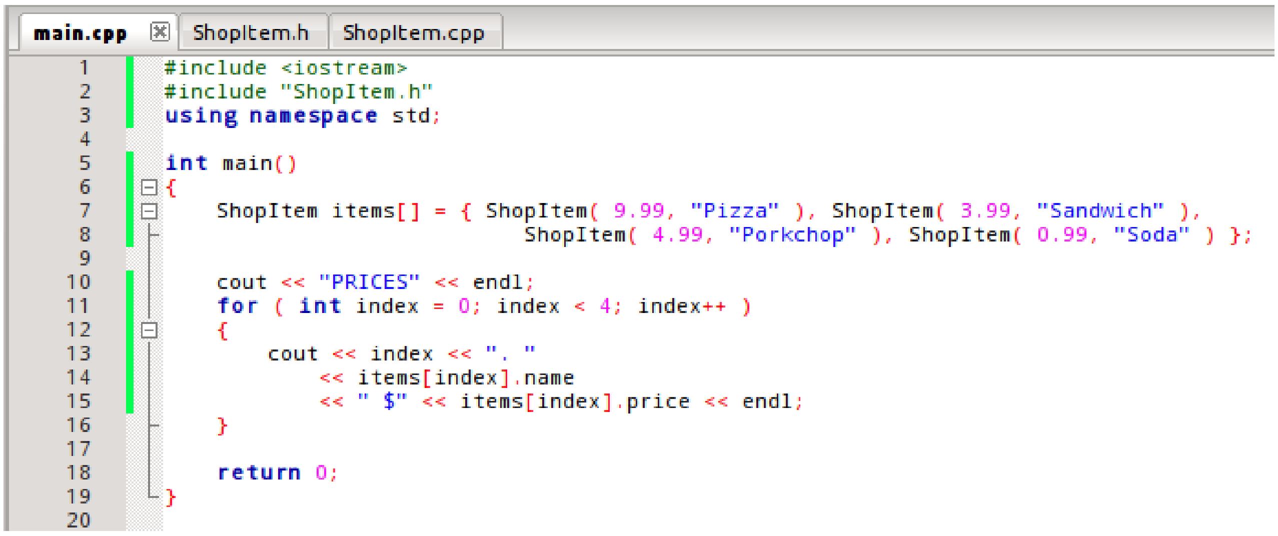
Old Version:



main.cpp | ShopItem.h | ShopItem.cpp

```
#include <iostream>
#include "ShopItem.h"
using namespace std;

int main()
{
    ShopItem items[4];

    items[0].name = "Pizza";
    items[0].price = 9.99;

    items[1].name = "Sandwich";
    items[1].price = 3.99;

    items[2].name = "Porkchop";
    items[2].price = 4.99;

    items[3].name = "Soda";
    items[3].price = 0.99;

    cout << "PRICES" << endl;
    for ( int index = 0; index < 4; index++ )
    {
        cout << index << ". "
            << items[index].name
            << " $" << items[index].price << endl;
    }

    return 0;
}
```

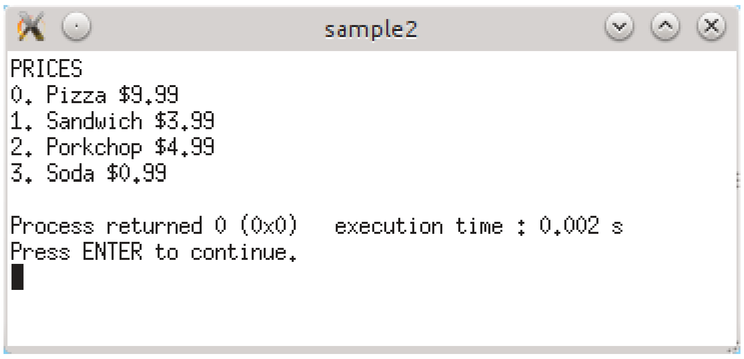
New Version:



main.cpp | ShopItem.h | ShopItem.cpp

```
#include <iostream>
#include "ShopItem.h"
using namespace std;

int main()
{
    ShopItem items[] = { ShopItem( 9.99, "Pizza" ), ShopItem( 3.99, "Sandwich" ),
                        ShopItem( 4.99, "Porkchop" ), ShopItem( 0.99, "Soda" ) };

    cout << "PRICES" << endl;
    for ( int index = 0; index < 4; index++ )
    {
        cout << index << ". "
            << items[index].name
            << " $" << items[index].price << endl;
    }

    return 0;
}
```

What we just did was **refactor** the program.
If we run the program now, we get:



```
PRICES
0. Pizza $9.99
1. Sandwich $3.99
2. Porkchop $4.99
3. Soda $0.99

Process returned 0 (0x0)   execution time : 0.002 s
Press ENTER to continue.
```

The output hasn't changed at all. When you **refactor**, you clean up your code with **no change to the way the program runs**. To the user, v3.0 of our "Prices" program is no different than v2.0, but now that our program is cleaner, we can expand it better, with less chance to get tied in a knot of spaghetti code.

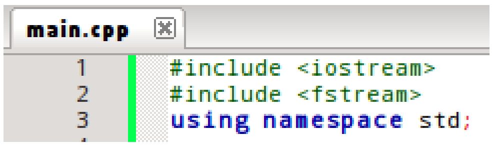
Make sure to check out the homework for a list of ways to expand this program.

Writing to Files

The C++ Standard Library contains console input and output, which we have been using this entire time (through iostream), and a library for handling strings, but it also has a library for opening and reading files.

We can save plain text, CSV files (comma separated values- these can be opened in Excel), or even something like an HTML file with working markup. After all, HTML files are essentailly just plaintext, but with rules for what a web browser recognizes.

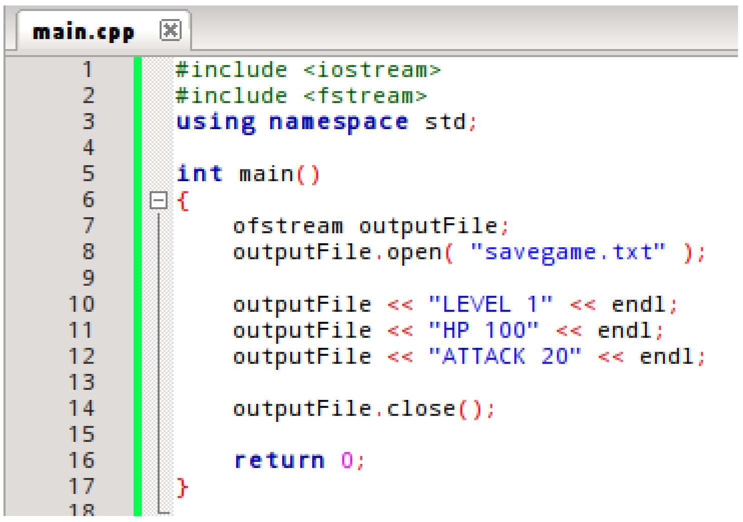
If we create a new project and add in **main.cpp**, we will start by including the appropriate library:



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

fstream is the library to use File input and output.

First, let's write a file.



main.cpp

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

int main()
{
    ofstream outputFile;
    outputFile.open( "savegame.txt" );

    outputFile << "LEVEL 1" << endl;
    outputFile << "HP 100" << endl;
    outputFile << "ATTACK 20" << endl;

    outputFile.close();

    return 0;
}
```

We have to create a new variable of type **ofstream**. I've named my "outputFile", because it's the file we're outputting text to.

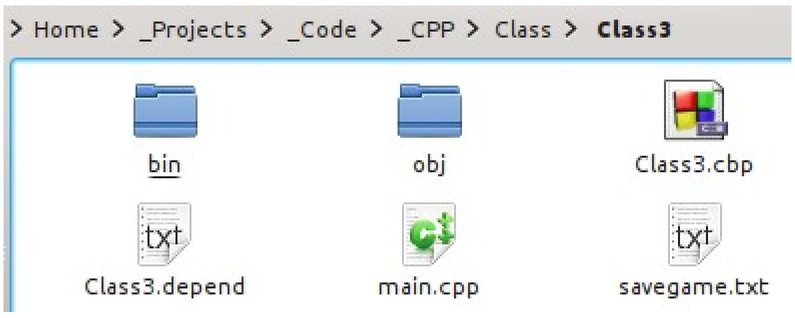
Secondly, we have to **open** a file. We can use any filename we want- you could even name it .sav instead of .txt. If it's a standard text format, C++ can read it either way.

Then, since this is a **file stream**, we can use it similarly to the **console output stream** we have been using this entire time.

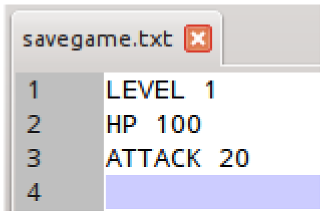
Use the variable name we created, then the double less-thans << to output strings. We can add **endl** for end-lines, and we can use **n** and **t** inside of double-quotes for new line and tab characters, respectively.

Finally, at the end of the program, we **must close the output file.** It's good practice to close a file as soon as we're done with it to prevent data loss.

If you run the program, nothing happens in the console, but if you open up your project folder, you'll see a text file:



If you oepn "savegame.txt", it will have the data we wrote to the file:



savegame.txt

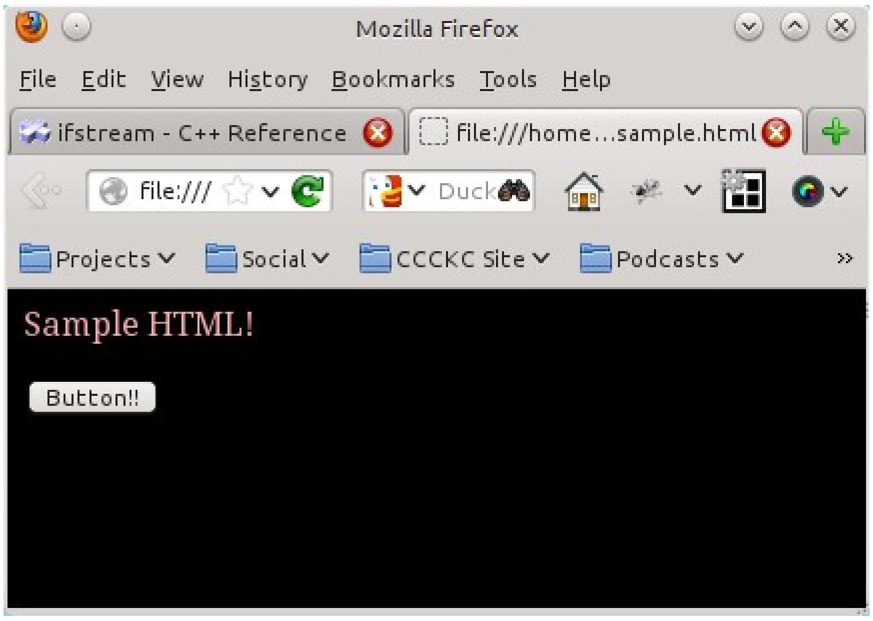
```
LEVEL 1
HP 100
ATTACK 20
```

So here we're saving a mock save-game file for a video game. Later, we can load this file to restore the player to the same stats they were at.

Let's output some basic HTML:

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

int main()
{
    ofstream outputFile;
    outputFile.open( "sample.html" );

    outputFile << "<body style='background: #000000; color: #ffaaaa;'>" << endl;
    outputFile << "<p>Sample HTML!</p>" << endl;
    outputFile << "<input type='button' value='Button!!'/>" << endl;
    outputFile << "</body>" << endl;

    outputFile.close();

    return 0;
}
```

Run the program, then open the "sample.html" file in the program directory:



Sample HTML!

Button!!

Beautiful and ugly HTML!

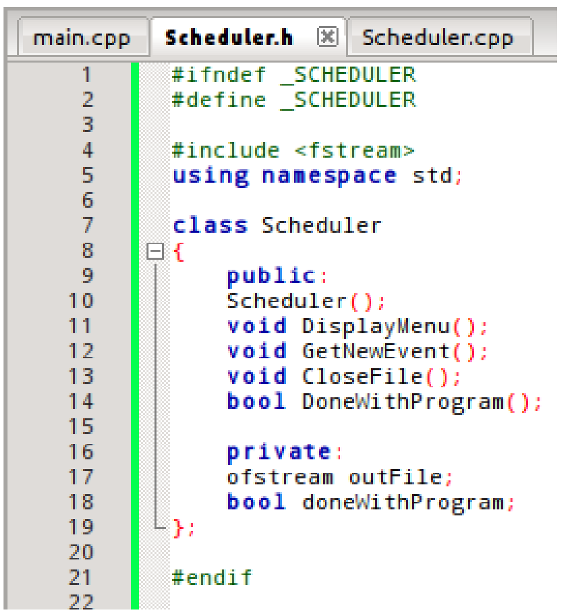
Now, usually you're not going to be outputting pure HTML in your files. You CAN, but usually you should (a) Use a pre-existing C++ library to write HTML for you, or (b) Write classes to handle outputting HTML.

Usually it's easier to let class objects handle the rules and making sure everything is formatted better. It's the same for *reading* in files- XML files, HTML files, etc. These types of special files usually have special rules and you need to ***parse*** them in order to get any useful data out of it.

Scheduler Project

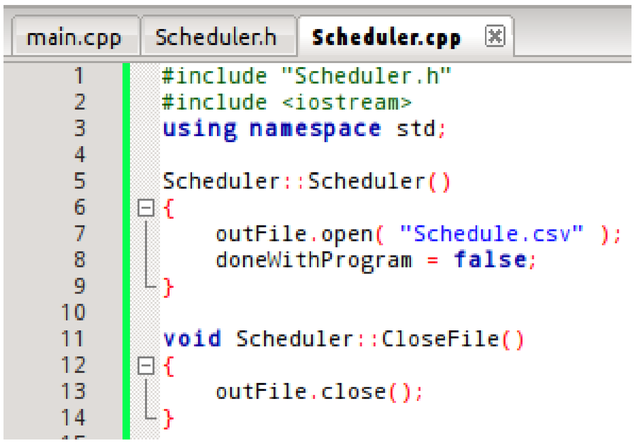
Let's create a program where we get information from the user, and output the data to a CSV file. We will keep getting schedule information from the user until they quit.

Let's create a Scheduler class. This class will have its constructor, a function to Display a Menu, to Get a New Event from the user, and to Close the file once we're done.



```cpp
#ifndef _SCHEDULER
#define _SCHEDULER

#include <fstream>
using namespace std;

class Scheduler
{
    public:
    Scheduler();
    void DisplayMenu();
    void GetNewEvent();
    void CloseFile();
    bool DoneWithProgram();

    private:
    ofstream outFile;
    bool doneWithProgram;
};

#endif
```

Then we'll begin filling out the function bodies in Scheduler.cpp:



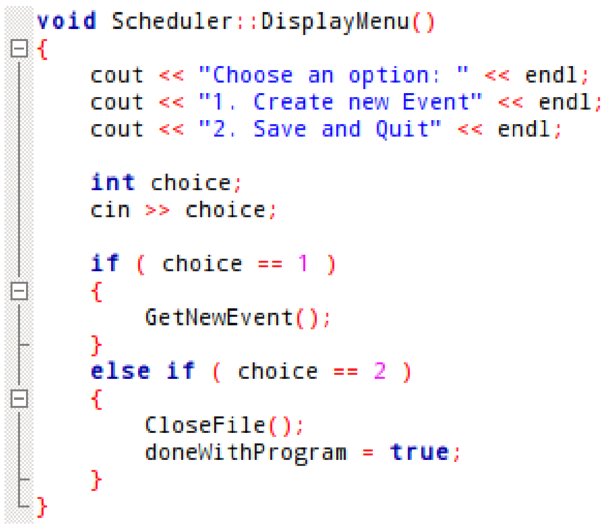
```cpp
#include "Scheduler.h"
#include <iostream>
using namespace std;

Scheduler::Scheduler()
{
    outFile.open( "Schedule.csv" );
    doneWithProgram = false;
}

void Scheduler::CloseFile()
{
    outFile.close();
}
```

With writing our output file, we'll want to open it before we do anything else. I'm going to open it in the constructor.

After we go through the program and the user decides to quit, we will use CloseFile to close our output file before the program ends.

For an event, we will get three things from the user:

- Event name
- Event date
- Event time

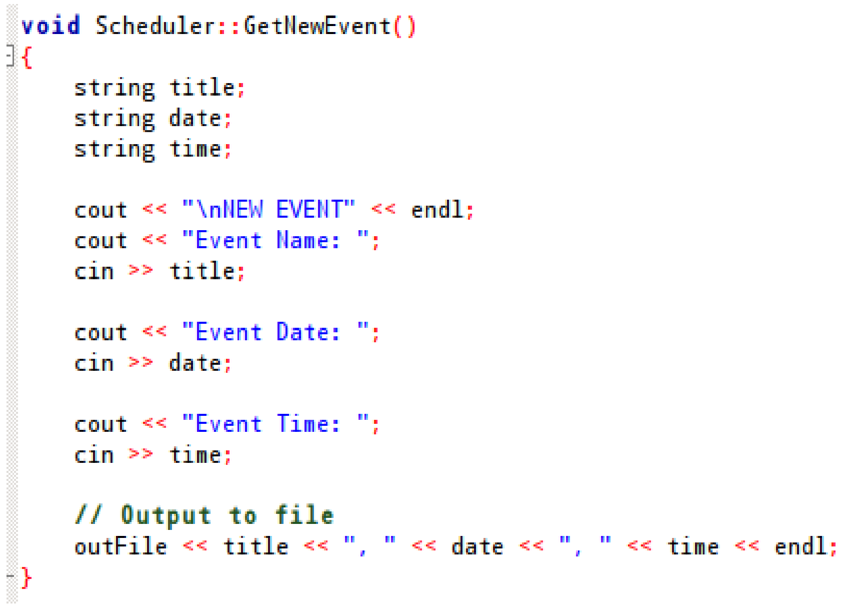


```cpp
void Scheduler::DisplayMenu()
{
    cout << "Choose an option: " << endl;
    cout << "1. Create new Event" << endl;
    cout << "2. Save and Quit" << endl;

    int choice;
    cin >> choice;

    if ( choice == 1 )
    {
        GetNewEvent();
    }
    else if ( choice == 2 )
    {
        CloseFile();
        doneWithProgram = true;
    }
}
```

In the DisplayMenu function, we can output some basic choices. If the user chooses "Create new Event", then we call another function within the Scheduler – GetNewEvent().

Otherwise, we'll set the doneWithProgram flag to **true**, so the program will quit. We also close the file so everything will be wrapped up and good to go.

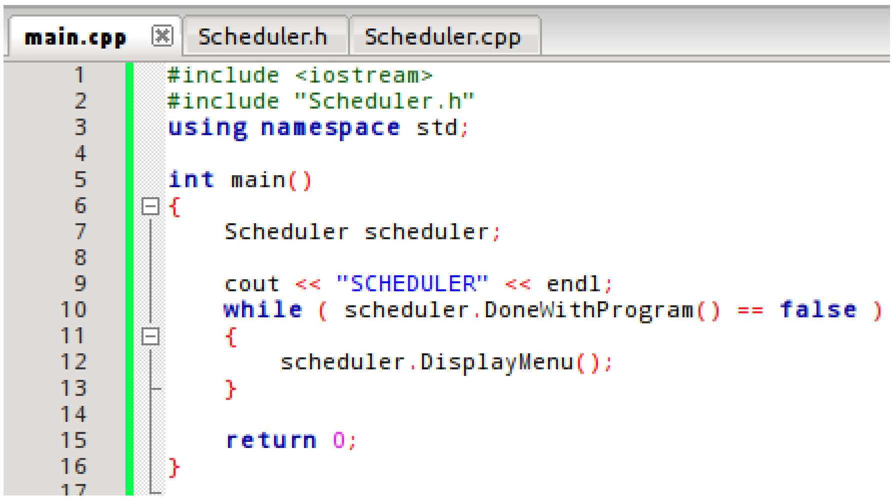
Then we can get the new event data and output that to the file afterwards.



```
void Scheduler::GetNewEvent()
{
    string title;
    string date;
    string time;

    cout << "\nNEW EVENT" << endl;
    cout << "Event Name: ";
    cin >> title;

    cout << "Event Date: ";
    cin >> date;

    cout << "Event Time: ";
    cin >> time;

    // Output to file
    outFile << title << ", " << date << ", " << time << endl;
}
```

We are separating our variables with commas, so when we open it in Excel or LibreOffice, it will separate items into their own columns.

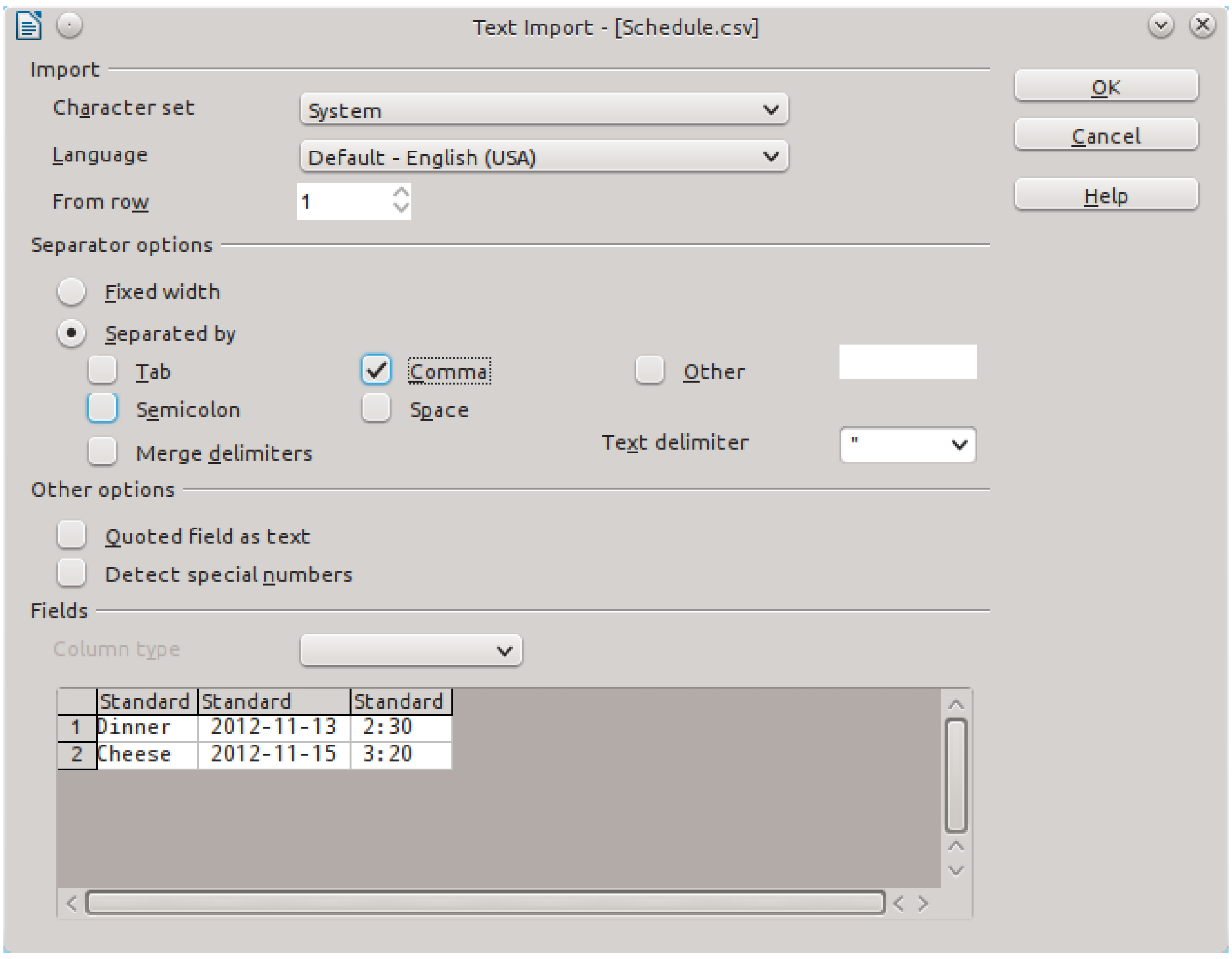
Now, finally, in main we have to actually get these parts working:



```
#include <iostream>
#include "Scheduler.h"
using namespace std;

int main()
{
    Scheduler scheduler;

    cout << "SCHEDULER" << endl;
    while ( scheduler.DoneWithProgram() == false )
    {
        scheduler.DisplayMenu();
    }

    return 0;
}
```

Now, if you run the program and enter some schedule information, then go to the output folder, you will see our csv file.

Open the file with Excel or Libre Office Calc. You will get a little pop up like this:


Text Import - [Schedule.csv]

Import
Character set: System
Language: Default - English (USA)
From row: 1

Separator options
Fixed width
[✓ Separated by]
Tab [✓ Comma] Other
Semicolon Space
Merge delimiters
Text delimiter: "

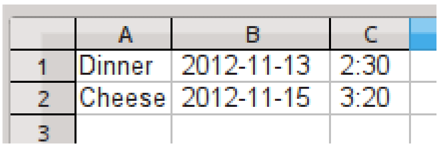
Other options
Quoted field as text
Detect special numbers

Fields
Column type

	Standard	Standard	Standard
1	Dinner	2012-11-13	2:30
2	Cheese	2012-11-15	3:20

Under the "Separated by" option, choose "Comma" (since we used Commas).

Then, when everything is done, our schedule will be in an excel file:



	A	B	C
1	Dinner	2012-11-13	2:30
2	Cheese	2012-11-15	3:20
3			

So that's a few examples of what we could do with file output in C++. We will cover file input (reading files) next time.

Check out the homework for some homework!