ITP 435 Lab Guide

Spring 2014

By: Sanjay Madhav
University of Southern California

Contents

Spring 2014 Due Dates 2

Basic Setup 3

Lab 1: RLE Compressor 8

Part 1: Compressing Strings 8

Part 2: Single File Compression 12

Part 3: Multi-File Archives 17

Closing Thoughts 20

Lab 2: Passwords and the Concurrency Runtime 21

Part 1: Basic Hashing 22

Part 2: Dictionary Lookup 23

Part 3: Brute Force 25

Closing Thoughts 29

Lab 3: Drawing is Fun!! 30

Part 1: Getting Started w/ GDI+ 31

Part 2: Shapes and Undo/Redo 37

Part 3: Saving/Loading 41

Part 4: Flyweight 43

Lab 4: ZombieSim 45

ZOM Framework 46

Part 1: ZOM Implementation 49

Part 2: Overall Simulation Additions 52

Part 3: Experimentation 53

Lab 5: The Zompiler 54

Part 1: Scanner and Parser 55

Part 2: The AST 57

Part 3: Code Generation 59

Spring 2014 Due Dates

Tuesday, 2/4 @ 11:59PM – Lab 1 Due

Tuesday, 2/25 @ 11:59PM – Lab 2 Due

Tuesday, 3/25 @ 11:59PM – Lab 3 Due

Tuesday, 4/15 @ 11:59PM – Lab 4 Due

Friday, 5/10/2013 @ 11:59PM – Lab 5 Due

Code must be submitted to the SVN server by the specified time. The grader will be syncing to a revision number as close to the due time as possible, with a grace period of no greater than 30 minutes. The code will then be graded on what is submitted to SVN.

Extensions will only be allowed in **documented** emergency situations. “I’m in the hospital” is an emergency situation; “I really want to go to this party” is not.

Basic Setup

Requirements

Software:

1. Windows 7 or higher. If you have a Mac, you'll have to install bootcamp or a VM.
2. Microsoft Visual Studio 2013 Professional (can be downloaded from http://www.dreamspark.com/). Note that Express 2013 will **NOT** work.
3. TortoiseSVN (http://tortoisesvn.net/)

SVN Setup

If you have previously taken ITP 380/485, you should be familiar with TortoiseSVN. You can skip this section and just know that your URL is:

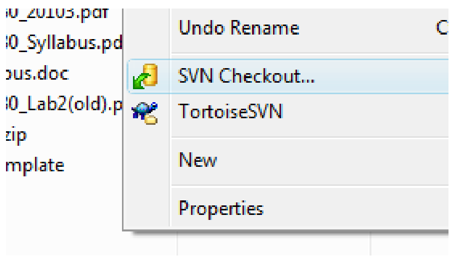
`http://itpsvn.usc.edu/svn/itp/20141/435/`**my_username**

And your username/password is just going to be whatever it was in your last class. If you don't remember it, let me know.

If you haven't used SVN before, don't worry. It's not that complicated! SVN is a source control system which allows you to save backups of your code on a remote server. This makes it easy to work on the same code from multiple locations, provides an easy submission mechanism, and even provides a history of changes which have been committed.

SVN: Checking Out

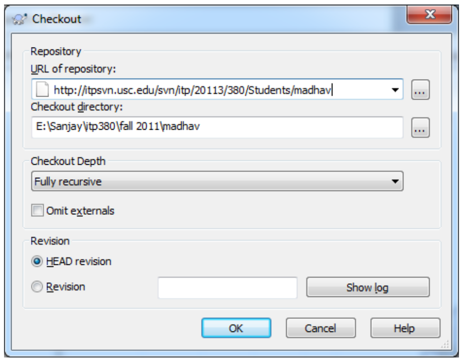
Each student has their own personal sandbox to play in. To get a copy of your code, you must first "check it out" from the SVN server. To do this, first open a destination folder where you would like to store your copy of the code. Right click on an empty spot in the folder window and select "SVN Checkout..." If you don't see this option, that means you haven't installed TortoiseSVN yet!



In the "URL of depository" field of the dialog box, type in:

`http://itpsvn.usc.edu/svn/itp/20141/435/`**my_username**

Where **my_username** is your USC account username. Note that this username is case sensitive. Your username is in all lowercase


Checkout

Repository

URL of repository: http://itpsvn.usc.edu/svn/itp/20113/380/Students/madhav

Checkout directory: E:\Sanjay\itp380\fall 2011\madhav

Checkout Depth: Fully recursive

☐ Omit externals

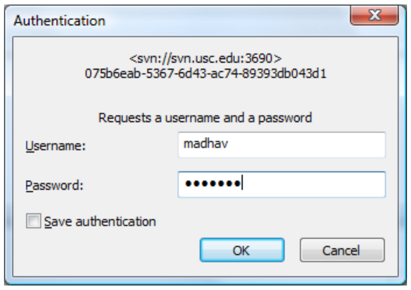
Revision

● HEAD revision

○ Revision

Leave all the other fields at the default options and click "OK."

You will then be asked to enter your username and password. Your username is simply your USC account username, and your temporary password is set to "itp435" (no quotes). Note that if you’re in more than one class with me for this semester, your password might instead be the default for that class. For obvious reasons, I will require you to change your password in class. This password is not encrypted, so please do not use your USC or any other important password.


Authentication

<svn://svn.usc.edu:3690>
075b6eab-5367-6d43-ac74-89393db043d1

Requests a username and a password

Username: madhav

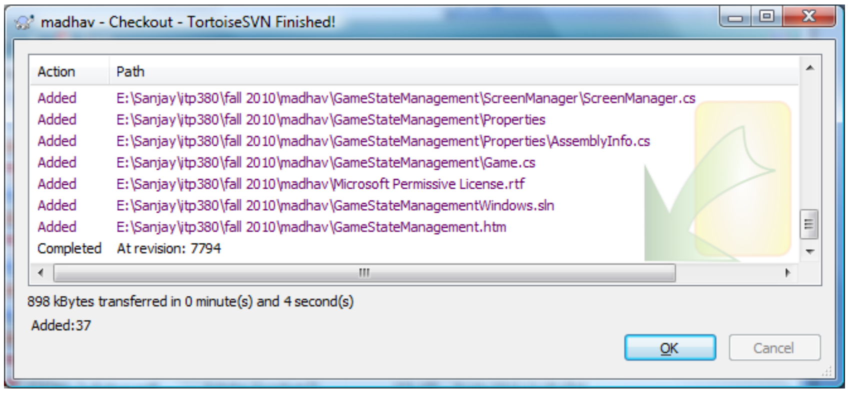
Password: •••••••

☐ Save authentication

If you didn’t get the authentication dialog box, that means someone saved the authentication on your computer with another account (which is bad). Let me know and I’ll show you how to clear saved authentication.

As a side note, you should never check the “Save authentication” box when you’re using a shared computer.

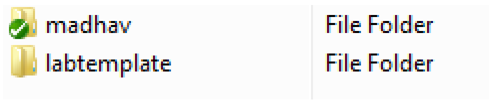
In any event, if everything worked out fine, you should see a dialog box that looks like this:


madhav - Checkout - TortoiseSVN Finished!

Action	Path
Added	E:\Sanjay\itp380\fall 2010\madhav\GameStateManagement\ScreenManager\ScreenManager.cs
Added	E:\Sanjay\itp380\fall 2010\madhav\GameStateManagement\Properties
Added	E:\Sanjay\itp380\fall 2010\madhav\GameStateManagement\Properties\AssemblyInfo.cs
Added	E:\Sanjay\itp380\fall 2010\madhav\GameStateManagement\Game.cs
Added	E:\Sanjay\itp380\fall 2010\madhav\Microsoft Permissive License.rtf
Added	E:\Sanjay\itp380\fall 2010\madhav\GameStateManagementWindows.sln
Added	E:\Sanjay\itp380\fall 2010\madhav\GameStateManagement.htm
Completed	At revision: 7794

898 kBytes transferred in 0 minute(s) and 4 second(s)
Added:37

You should now have a folder called "**my_username**" and it should have a green check mark. This folder will contain all of your source code.

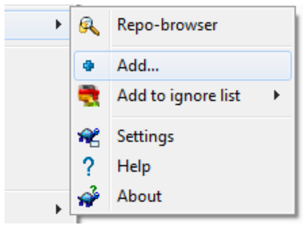


madhav	File Folder
labtemplate	File Folder

SVN: Adding Files

In some of the labs, you will be told to create new .cs files and add them to your project. Whenever you do this, you have to tell the SVN server that you added the files. While it is possible to add the files in the “Commit” screen, it’s a good habit to remember to add a file as soon as you create it so you don’t forget later.

To add a file, simply right click on it and select TortoiseSVN>Add..

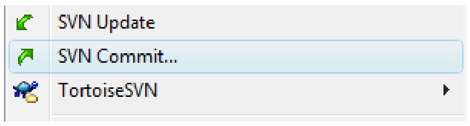


When you do this, it will **not** actually submit it to the server. But basically it tells the server the next time you “commit” your folder, it should also commit the file.

SVN: Commit

Once you are done with your changes for the time being, and would like to save them to the server, you must commit your changes. Go back to the directory you originally checked out to.

Your "**my_username**" folder should now have a red exclamation point (if it doesn't, it probably just hasn't updated). Right click on the folder and select "SVN Commit..."



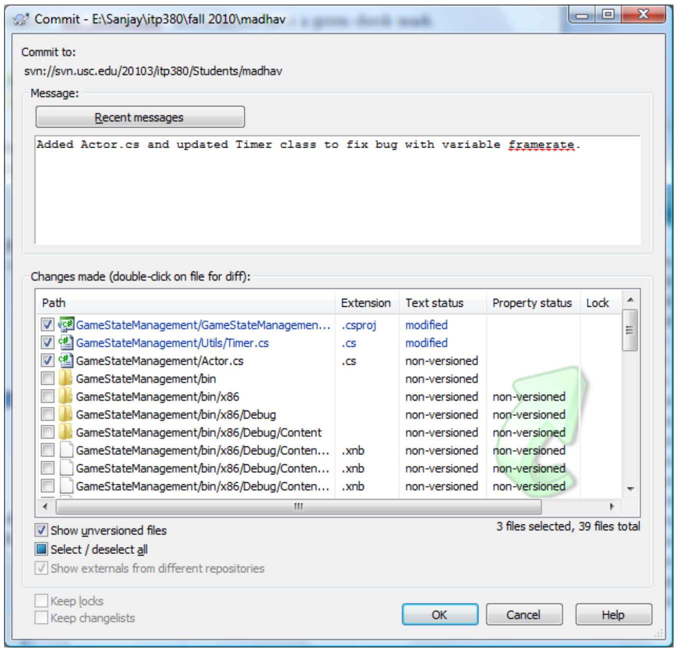
In the commit dialog box, it will automatically select the files you have changed. In addition, it will show any *non-versioned* files, or files which currently do not exist on the server.

If it's a file you created yourself, like Actor.cs, you need to make sure those new files are checked in this box. If you followed the instructions in the Add section, they should be. If the files aren't checked, the server won't know about the files and the files **will not save to the server**.

Make sure you commit your work to the SVN server. If you don't, the files will not be there when it's time to grade.

Note that you will see several non-versioned files which you did not create yourself. These are temporary build files and executables which you don't need or even want to check in to the server, so you can safely ignore these.

Once you are happy with your file selection, be sure to type in a useful note in the "Message" box. This way, if you look at a particular change two weeks from now, you will remember why you made it.



Commit - E:\Sanjay\itp380\fall 2010\madhav

Commit to:
svn://svn.usc.edu/20103/itp380/Students/madhav

Message:

Recent messages

```
Added Actor.cs and updated Timer class to fix bug with variable framerate.
```

Changes made (double-click on file for diff):

	Path	Extension	Text status	Property status	Lock
✓	GameStateManagement/GameStateManagemen...	.csproj	modified		
✓	GameStateManagement/Utils/Timer.cs	.cs	modified		
✓	GameStateManagement/Actor.cs	.cs	non-versioned		
☐	GameStateManagement/bin		non-versioned		
☐	GameStateManagement/bin/x86		non-versioned	non-versioned	
☐	GameStateManagement/bin/x86/Debug		non-versioned	non-versioned	
☐	GameStateManagement/bin/x86/Debug/Content		non-versioned	non-versioned	
☐	GameStateManagement/bin/x86/Debug/Conten...	.xnb	non-versioned	non-versioned	
☐	GameStateManagement/bin/x86/Debug/Conten...	.xnb	non-versioned	non-versioned	
☐	GameStateManagement/bin/x86/Debug/Conten...	.xnb	non-versioned	non-versioned	

3 files selected, 39 files total

[✓] Show unversioned files
[■] Select / deselect all
[✓] Show externals from different repositories
[ ] Keep locks
[ ] Keep changelists

OK Cancel Help

Once you're ready, hit "OK" and you should get a friendly status message.

REMINDER!! Make sure you always submit your changes before logging off an ITP computer, or your changes may be permanently lost.

SVN: Other notes

There are more advanced SVN uses, such as right clicking on a particular file and comparing changes, and so on. However, I will not cover these in this document. If you have any questions, feel free to ask.

And now, we're ready to code!

Lab 1: RLE Compressor

Run-length encoding is one of the most basic forms of compression. In this lab assignment, we will implement a compression program which can compress files using a modified form of RLE. The basic RLE algorithm we'll use was covered in the Week 1 lecture, so you may want to read over the notes and make sure you understand the algorithm before continuing on this project.

For the first part of the assignment, you will be tasked with compressing/decompressing a variety of strings that are hardcoded into the program. After that, you will write code that can both compress and decompress single files. In the final part of the lab, you will implement functionality which allows you to compress an entire directory of files into one compressed archive.

The intent of this lab is to reacquaint you with C++ and especially pointers and memory usage, since those are often stumbling points when returning to an unmanaged language.

The starting code for this project is in the rle subdirectory of your SVN directory. Open up rle.sln to get started.

Part 1: Compressing Strings

The first thing you need to do in the main function (in rle.cpp) is to add a simple text menu. The menu should have three options at the top level. You can name the options however you want, but it should basically look like this:

```
Select an option:
1. Part 1 (Tests)
2. Part 2 (Single Files)
3. Part 3 (Directories)
4. Exit
>
```

The corresponding number input should go to the correct sub-menu, which we'll be adding later. Make sure you bulletproof your menu input system. You don't want people to be able to break the menu by typing in bad values. That's not very professional!

If you don't remember how to I/O in C++, here's a quick reference on that: http://www.cplusplus.com/doc/tutorial/basic_io/. Note that in these samples they always use the `using namespace std;` directive, which I generally don't recommend. If you don't have the using command, you have to refer to the classes/functions with the full scope, like `std::cout` and so on.

Since we want to error correct, you can't just use `std::cin` for your input. You need to use `std::getline` and then process the numbers from that string using string streams.

For now, let's worry about option 1. When the user selects it, all you need to do is run the `Part1Tests` function which is conveniently already in rle.cpp. If you run this right now, you won't get much output because you haven't implemented anything yet! But once you implement the RLE compression/decompression, it will provide output which will help verify that it works.

Compressor

If you open up rle-algo.hpp, you will notice there is a templated `struct` called RLE. It's templated because we want our algorithm to be able to perform RLE on strings of different types. So whether it's an array of characters (C-style string) or an array of shorts, or an array of integers, we want the algorithm to function.

For now, we are going to implement `RLE::Compress`. This function takes in data and compresses it with our RLE algorithm. The function has two arguments: the pointer to the data, and then the size of the data stored at that address. Implementing RLE compression isn't that much code to write, but it may take a while to wrap your head around the pointer arithmetic.

The first thing you want to do in this function is `delete m_Data` and set `m_Size` back to `0`. This is because if `Compress` is called multiple times, we want to make sure all previous data has been wiped.

Next, we need to allocate memory for `m_Data`, which is where we will store the compressed data. Ideally, we'd like to think this will always be smaller than the input data, but this is not guaranteed. So I'd recommend allocating an array (on the heap) that's twice the size of `inSize`.

You also need a variable to track what the maximum run size is for this type. I have provided a `MaxRunSize` function which uses a bit of template trickery to determine what the maximum run size is based on the type. We haven't really covered the advanced aspects of templates yet, so you can just black box `MaxRunSize` for now. But I'd recommend storing the return value of the function in a local variable in `Compress`, rather than calling it repeatedly.

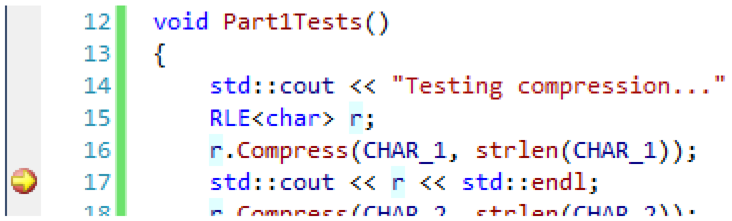
Now we get to the meat of the `Compress` function. You need to move through the data stream and find both regular runs and the so called "negative" runs (which means you have a run of data which does not repeat). There are a few different ways to implement this, but one approach is to make a temporary pointer (called `temp`) which starts out pointing at beginning of the input stream. You can then check if `(temp + 1)` is pointing to data which is the same or different as what temp is pointing to. If it's the same, that means you have a normal run, and can loop through and find out how many times the data repeats. If the values are different, it means you have a negative run, and can search and find the end of the negative run.

Note that if the run is longer than the maximum run size, you need to split it up into multiple runs. Do not try to write a run that's longer than the max size, or it will break!

Once you have the run type and run size, you can write the appropriate data to the `m_Data` array. So for example, if you find a run of 'a' three times in a row, you would first write the number 3, and then the character 'a' into `m_Data`. The easiest way to keep track of where you are in `m_Data` is by incrementing `m_Size` every time you write to `m_Data`. Then you can simply write the next value to `m_Data[m_Size]`.

I'd recommend breaking down the problem into smaller pieces. First just ignore negative runs and try to get regular runs to work properly. There are several constant strings which are defined in rletest.h. These strings are what get sent to `r.Compress` in the `Part1Tests` function.

To test out the algorithm as you write it, I'd recommend using breakpoints after each call to `r.Compress`. You can check and make sure the data inside `r` matches what it should be. For example, you can put a breakpoint here:



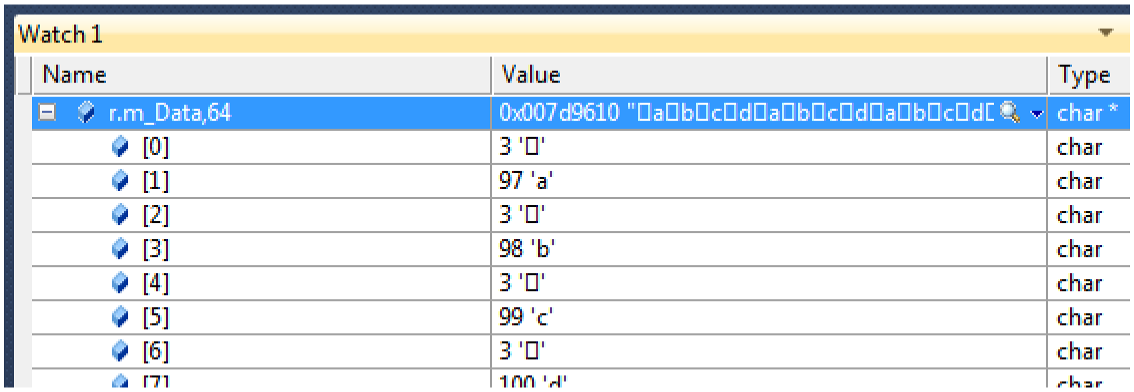
```
void Part1Tests()
{
    std::cout << "Testing compression..."
    RLE<char> r;
    r.Compress(CHAR_1, strlen(CHAR_1));
    std::cout << r << std::endl;
    r.Compress(CHAR_2, strlen(CHAR_2));
```

Then add a watch to look at the data. To add a watch, you first need the watch window open, which you can access from Debug>Window>Watches>Watch 1. Once there, you need to type in the name of the watch, which is as follows:

```
r.m_Data,64
```

The `,64` tells Visual Studio that `r.m_Data` is an array, and you want to view the first 64 elements of that array. You can then inspect this data and make sure the elements line up as they should based on the sample string.

The watch should then display something like this, assuming you correctly implemented compression for regular runs:


Watch 1

Name	Value	Type
r.m_Data,64	0x007d9610 "□a□b□c□d□a□b□c□d□a□b□c□d□	char *
[0]	3 '□'	char
[1]	97 'a'	char
[2]	3 '□'	char
[3]	98 'b'	char
[4]	3 '□'	char
[5]	99 'c'	char
[6]	3 '□'	char
[7]	100 'd'	char

It turns out that `CHAR_1` only has regular runs, and no negative runs. It has `'a'` three times in a row, then `'b'` three times in a row, then `'c'` three times in a row, and so on.

Once you have the regular runs working properly, you can implement negative runs. `CHAR_2` has just one single negative run. `CHAR_3` tests the combination of negative and regular runs, and `CHAR_4` tests and makes sure that if you have a run which is larger than the maximum run size, it's split up into two separate runs.

In any event, once your compression algorithm works properly, your console output should look like this when you select option #1:

```
Testing compression...
♥a♥b♥c♥d♥a♥b♥c♥d♥a♥b♥c♥d♥a♥b♥c♥d♥a♥b♥c♥d♥a♥b♥c♥d♥a♥b♥c♥d♥a♥b♥c♥d
µabcdefghijklmnopqrstuvwxyz
²abc♦d²efg♦h²ijk♦l²mno♦p²qrs♦t²uvw♦x y0z
⌂aIa
```

Please note that the unit tests as provided are only a first pass test, and it's possible to have an incomplete RLE algorithm pass all the provided tests. It is recommended to fully test your implementation with additional test cases to ensure that it works as expected.

The reason some of these symbols look odd is that they are outside of the standard ASCII range, so they can be assigned to whatever symbols the operating system decides to use. In this case, it decided on hearts. Note that if you are using a different character set on your Windows install (such as a different language by default), then you may not get the same characters to show up. In that case, the only way to test it effectively is via the debugger method.

Decompress

Implementing the decompression is a bit easier. You'll notice that `RLE::Decompress` has three parameters instead of two. It still has the input stream and the size of the input stream, but the third parameter is the known size of the decompressed data. This way, `Decompress` can allocate the exact amount of memory necessary for `m_Data`.

The function needs to loop through the input stream and convert it as appropriate. You know that the first element has to be run information, so you can output that run to `m_Data`, and then move onto the next one. You can test this in the debugger in a similar fashion like you did for `Compress`. There's only three test strings for decompression, however. The first one should be 40 `'x'` characters in a row, the second should have 3x `'a'`, 3x `'b'`, 3x `'c'`, and 3x `'d'`, and the last one has a bit of everything.

Once you have this implemented, you can take a look at your console output for decompression, and it should be something like this:

```
Testing decompression...
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
aaabbbcccddd
abcdeaaaaabbbb
```

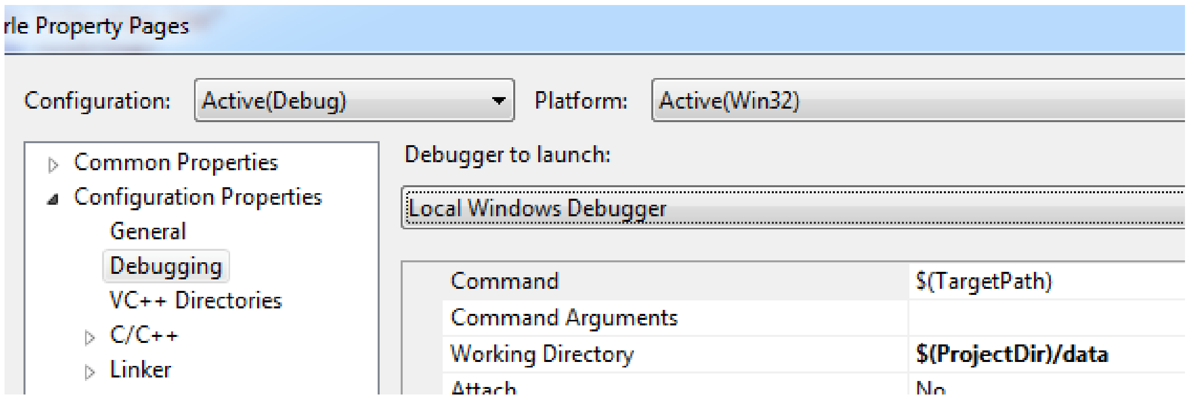
As with compression, these tests may not fully test if your decompression is bullet proof. It's recommended you try some custom tests for decompression, as well.

In any event, once compression/decompression works, you're done with part 1!

Part 2: Single File Compression

Before we start on this part, you should set your working directory for debugging. This is the directory that the program starts in when you're debugging. All of the data files we will use for this part are in a specific directory, so we want to change our working directory to point to this.

To change the working directory, right click on the rle project in the solution explorer. Then go to Configuration Properties>Debugging. Change the Working Directory to **$(ProjectDir)/data**



Command	$(TargetPath)
Command Arguments	
Working Directory	**$(ProjectDir)/data**
Attach	No

Once you setup the working directory, the next step is to add a sub menu for option #2 in the main menu. This sub menu should present three options: create an archive, extract an archive, or return back to the main menu. If the user selects create archive, they should be prompted for the name of the file they wish to compress. Similarly, if they select extract an archive, they should be prompted for the name of the file they want to extract.

The code for creating and extracting the archive will be written in rle-files.cpp. There are two different classes in this file: `RLE_v1`, which is for this part of the lab, and `RLE_v2`, which is for the next part.

CreateArchive

When the user selects "create an archive" and passes in a file name, you should create an instance of `RLE_v1` and call the `CreateArchive` function with the file name as the parameter.

The first thing `CreateArchive` needs to do is open the requested file in binary mode. We can't use the default text reading mode, because there is no guarantee the file will be human readable. It most likely is going to be a random file with random binary data. In binary mode, we can just load all of the contents of the file as one big array. This makes it easy to then compress the data stream using our existing functions.

Here is one way to open a file in binary mode and copy all of its content into a character array:

```
// Requires <fstream>
std::ifstream::pos_type size;
char* memblock;
std::ifstream file (source, std::ios::in|std::ios::binary|std::ios::ate);
if (file.is_open())
{
      size = file.tellg();
      memblock = new char [static_cast<unsigned int>(size)];
      file.seekg (0, std::ios::beg);
      file.read (memblock, size);
      file.close();

      // File data has now been loaded into memblock array

      // Make sure to clean up!
      delete[] memblock;
}
```

After the `file.read` call, all the data will be loaded and ready into `memblock`. Since this is a character array, it can be directly passed to an RLE class for compression. Conveniently, there is already an RLE class which is a member variable in `RLE_v1`! So once you load in the data, you can simply store it in that `m_Data` member variable.

When the file is compressed, you should add a little bit of a status message that tells you what percent compression was achieved. That way you can see how effective the compression was on that particular file. If a file was originally 1000 bytes and is compressed down to 250, the result was 75% compression which is pretty darn good.

Once we have all of the RLE data ready to go, we need to write that data to an output. By default, the file name should just take the initial file name with ".rl1" added on. So if the original file was "rle.bmp", the archive's file name should be "rle.bmp.rl1".

To write data in binary mode, instead of an `ifstream`, you use an `ofstream`. The code to load a file in binary mode is like this:

```
std::ofstream arc(filename, std::ios::out|std::ios::binary|std::ios::trunc);
if (arc.is_open())
{
    // Use arc.write function to write data here
}
```

The `ios::trunc` flag specifies that we want to overwrite the file if it already exists. In a second I'll talk about how to write the binary data. But before we can implement the code to write ou the file, we need to look at the RLE v1 file format. This diagram shows what a sample RL1 file might look like:

Address	0	1	2	3	4	5	6	7
	File Signature				Original File Size			
0x00	‘R’	‘L’	‘E’	0x01				
File Name Size		File Name (Variable Length)						
0x08	7	‘r’	‘l’	‘e’	‘.’	‘b’	‘m’	‘p’
...	RLE Data (Variable Length)							
...								
...								
EOF								

Most file formats have a header which stores some information regarding what's in the file. For this file format, there is a 4 byte file signature which spells out the string "RLE" followed by the hexadecimal number 1. Using a file signature is very common, and most file formats have one.

This makes it relatively easy to determine when opening a file what the intended file format is. Granted, someone could change or corrupt the signature, but it at least gives a starting point.

The next part of the header is the original size of the file before it was compressed. This is so we know what size to make the buffer when it's time to decompress the data. The header then stores the size of the original file name (before it was compressed), and the actual file name string. The size file name string will be vary based on whatever the name is. In our `RLE_v1` class, there is a `m_Header` member variable which has all of the data fields which correspond to those in the header. I'd recommend storing the header information into `m_Header` before actually writing it out to the final file.

After the header, the rest of the file is the compressed RLE data. Generally, unless the original file was extremely small, this will be the bulk of the data in the file.

To write binary data into a file, you need to use the `write` member function of `ostream`. This function takes two parameters: a pointer to the character data that needs to be written, and the number of bytes of data to write. This is easy enough to use if you have a character buffer. For instance, to writing the file signature is just this:

```
arc.write(m_Header.sig, 4);
```

But what about the file size, which is stored as an `int`? If you just dereference it with an `&`, you'll get an `int*`, and it won't work:

```
arc.write(&(m_Header.fileSize), 4); // Compile error :(
```

Since the function expects a `char*`, you must `reinterpret_cast` this `int*` to a `char*`. Then the function will be happy:

```
arc.write(reinterpret_cast<char*>(&(m_Header.fileSize)), 4); // Works!
```

We pass in `4` as the second parameter because the size of an `int` is 4 bytes in Visual Studio.

Also, remember that when writing a std::string to the archive, you will need to convert it to a c-style string by using the `c_str` member function.

It's a bit difficult to test whether or not your RL1 file was written correctly without also writing the `ExtractArchive` function. But you could verify that if you compress rle.bmp, you end up with a file that's approximately 8.77KB.

ExtractArchive

The ExtractArchive function is simply the opposite of the CreateArchive one. It loads in an already compressed archive, and then properly reconstructs the file based on the data. First you should load in the entire file into memory, and try to process the header. This means you should grab the file signature, and copy it to your m_Header structure. You MUST check for the file signature before further processing. If a file which is not contain the RL1 file signature is passed into CreateArchive, it should be rejected with an error message notifying the user that it's not a valid archive.

Grabbing the file signature should be self explanatory, but grabbing the other header variables requires a bit of pointer trickery. You know that the file size element starts at the 5th byte, or memblock[4] (if you used the ifstream code from before). And you know that ints are a total of 4 bytes. The best way to grab those 4 bytes and store them into the integer is like this:

```
m_Header.fileSize = *(reinterpret_cast<int*>(&memblock[4]));
```

What we are doing is getting the pointer to the 5th byte, casting that pointer to an integer pointer, and then dereferencing the integer pointer to get the correct file size value.

At this point you then need to grab the length of the file name, and use that to determine how many characters you need to read in to get the original file name. Once you have the file name, the rest of the data in the file will be RLE compressed data, which you want to send to the Decompress function.

Once you have the decompressed data, you can then open a file for writing in binary mode (with the correct original file name). Then just write out all the decompressed data into this file, and you should have the original file reconstructed.

To test your CreateArchive/ExtractArchive functions, an easy thing to do is to first try compressing rle.bmp. Then delete rle.bmp and try to extract rle.bmp.rl1. If it worked, at the very least you made a reversible compression algorithm.

Next, you should try out the couple of other sample .rl1 files I've included in the data directory. Make sure they all decompress and they all function properly. Both are bitmap files so they should decompress and be viewable. One of the files is even a comic from xkcd. If this works, that means you implemented RL1 archives to spec, and can move on to RL2!

Part 3: Multi-File Archives

The difference between the RL1 and RL2 archives is that the RL2 archive supports compressing an entire directory of files into the archive. Just like with part 2, we want to add a menu for this part. It should again ask if the user wants to create an archive of extract an archive.

For create archive, this time you’ll ask for the name of a directory instead of the name of a specific file. But the extract archive will still be looking for an individual file, of course.

Because we want to support multiple files, we have to adjust the file format a bit:

Address	0	1	2	3	4	5	6	7
	File Signature				Dir Len	Directory Name		
0x00	‘R’	‘L’	‘E’	0x02	7	‘l’	‘e’	‘t’
					File 1: File Size			
0x08	‘t’	‘e’	‘r’	‘s’				
	File 1: Compressed Size				F1: Len	File 1: File Name		
0x10					5	‘a’	‘.’	‘b’
			RLE Data (Variable Length)					
0x18	‘m’	‘p’						
...								
...								
...								
	File 2: File Size				File 2: Compressed Size			
...								
	F2: Len	File 2: File Name					RLE Data	
...	5	‘b’	‘.’	‘b’	‘m’	‘p’		
...								
EOF								

We don’t have to store as much information for the overall archive header. But then for each file in the directory, we have to store a certain amount of information. In the `file_info struct`, we store the uncompressed and compressed data size, the file name information, and then finally the compressed data for the file. The reason we have to store the compressed data size is because we need to know when one file ends and the next one begins.

After each individual file’s bit of header information, we then have all of the RLE data for that file. Once that file’s RLE data is complete, the next element will either be the EOF or the next file’s file size. So the above diagram would have two different files stored in it: a.bmp and b.bmp.

CreateArchive

Now let's look at what we need to do in `RLE_v2::CreateArchive`. Since it receives the name of the directory we want to compress, the first order of business is to generate a list of all the files that are in this directory. There unfortunately is no standardized way in C++ to generate this list of files. There are cross-platform libraries you can use, but for ease we will just use the Win32 API function to search through a directory. The code for this is below.

```
// Find the first file in the directory.
WIN32_FIND_DATA ffd;
std::string path = source + "\\*.*";
HANDLE hFind = FindFirstFile(path.c_str(), &ffd);
if (INVALID_HANDLE_VALUE == hFind)
{
	// No files found, display error message and return!
	return;
}

// Add all files in this directory to a list of strings
std::list<std::string> files;
do
{
	std::string temp = ffd.cFileName;
	if (temp != "." && temp != "..")
	{
		files.push_back(ffd.cFileName);
	}
}
while (FindNextFile(hFind, &ffd) != 0);
```

What this does is uses the `FindFirstFile`/`FindNextFile` functions to iterate through the directory. Notice how I tell it to ignore "." and "..", since those aren't files but rather links to the current or preceding directory, which we don't want. You can test out this code by telling it to search the "letters" directory. After the above code runs, the files list will have a list of "`a.bmp`", "`b.bmp`", "`c.bmp`", and "`d.bmp`".

Once this list of files is generated, we can open the files one by one. Don't forget that when you create an `ifstream` for these files, you need to pass in the path including the directory name. If you just pass in the file name from the files list, you will just get "`a.bmp`" instead of the correct "`letters/a.bmp`".

For each file we open, we want to make a separate `file_info struct` on the heap with `new`. This `struct` stores the compressed data as well as other information relevant to the archive. Once a particular `file_info struct` is created, it should be added to the `m_Files list`. Don't forget that since we are allocating the `file_info struct` on the heap, we need to make sure this data is properly deleted in the destructor for `RLE_v2`.

Once you have everything loaded into memory, you can write out the .rl2 file. First write out the overall header, and then write the information for each individual file. Since `m_Files` is a list of `file_info*`, you can use iterators to loop through the list. Keep in mind since the data in the list is a pointer, you need to do a double dereference: once for the iterator and once for the pointer to the `file_info`. So for example, if your list iterator is called `i`, to access the file size, you need to do this:

```
(*i)->fileSize
```

Once you implement the archive creation code, you should test it out on the letters directory that's provided for you. As with the previous part, you can't really fully test until you also write the extraction function. But if it didn't crash and you got a letters.rl2 file that's approximately 10.2KB, you probably did it correctly.

ExtractArchive

At this point, you are hopefully proficient enough with the file I/O that you can handle extracting the files back from the archive. You're just going to read in the header, then read in a `file_info*` for each file that's in the archive, adding it to the `m_Files` array.

There is one big gotcha that I want to mention, though. When it's time to output the files, the directory you want to output them to may not already exist. If you try to open the file "letters/a.bmp", but the letters directory doesn't exist, it just won't work. So before you actually write out the files, you need to create the directory just in case. To do this in Windows, you call `CreateDirectory`:

```
CreateDirectory(directoryName, NULL);
```

Anyway, once you create the directory, you simply need iterate through your `m_Files` list and output all the files into the directory.

You can test this now by deleting your letters directory locally, and trying to extract the archive. If you implemented it properly all the files should be back. The last thing to check is trying to extract the additional RL2 archive in your data directory. If this also extracts properly, congrats!

Don’t forget to make sure your code does check for errors. You don’t need to catch every single possible error, but make sure at least something like loading an RL1 file as an RL2 file is detected, and check for files not existing, etc. Once you do that, you’re done!

Closing Thoughts

Hopefully this lab gave you an in-depth look at how binary file I/O works in C++. Also, I hope you feel more comfortable using pointers after this lab. If you test out the program on different file types, you’ll notice that often it does not give that great of a compression ratio. Certain bitmap image files compress really well, but other file types like waves actually end up having a negative compression (meaning the compressed file is larger than the original). Actual compression algorithms like those used for ZIP or RAR are a bit more complicated, usually using some form of a dictionary or coding. But since this is not an algorithms class, I did not think it would be a good use of time to implement one of those. And in any event, if you need to implement ZIP style compression you can just use zlib.

One improvement that might be interesting to implement is the idea of multiple data addressing modes. Right now we read everything as a character array and apply RLE on that. But some files may actually compress better if everything was an integer array. And our RLE algorithm as written in rle-algo.hpp can support 1, 2, or 4 byte integral primitives. So it might be interesting to try compressing the file in all three of those modes, and seeing which mode gives a better compression ratio. Then in the header information, it could specify which mode the file was compressed in.

Another thing that would make this program more useful is a command line interface. While the menu system we implemented is okay, if this were going to be a live compression program, we’d want command line arguments so it can be ran with automation.

Lab 2: Passwords and the Concurrency Runtime

Like many things in Computer Science, what you learn can be used for the powers of good or the powers of evil. In this lab, you will demonstrate how easy it is to crack a password that is a word from the dictionary, and that it's pretty hard to crack one that's random and lengthy.

I hope it goes without saying, but if not: **cracking passwords on an actual system is both ethically and legally wrong.** Although some of what you learn on this lab could theoretically be leveraged on an actual system, I trust that you will not do so. This code should only be used on the sample data set I provide for the lab.

With that out of the way, I'll describe this assignment. Most systems store passwords in an encrypted manner. So if you were to find the password file, all the data would be in an unreadable format. However, most passwords are encrypted using a known algorithm. In our case, the sample password data is encrypted using SHA-1, which is a hashing algorithm first published in 1995, based on research at the NSA. It is not considered a terribly secure hashing algorithm anymore, because it is relatively simple to compute. For this reason, no real system should use SHA-1 to encrypt their passwords (though sadly, many of them do).

The premise of a dictionary lookup is pretty simple. You have a list of plain text words which are commonly used as passwords. You encrypt each of these words using the encryption algorithm, then compare the encrypted dictionary words against the encrypted words in the password file. If there's a match, you've now quickly determined the password.

The Concurrency Runtime is a library Microsoft added to Visual Studio 2010. This allows you to do tasks concurrently across multiple threads, to speed them up. A dictionary attack is not very computationally expensive, especially when using the SHA-1 algorithm. You can determine thousands of passwords in a matter of seconds. So it doesn't make much sense to do it concurrently.

However, if the dictionary lookup fails on a particular password, we will then attempt to brute force the password. This can take a decent amount of time, and does make sense to parallelize.

Above a certain size, brute forcing passwords can take an inordinate amount of time. For the constraints of this lab, we will limit non-dictionary passwords to 4 characters which can only be lower case letters or numbers. That's roughly 36^4 + 36^3 + 36^2 + 36 possibilities, which is something like 1.8 million combinations. On my computer, brute forcing a 4 character password fitting that criteria can be done in less than a second, if you use multiple threads.

That's why it's best to use pass phrases, which are sentences with many characters. Easy to remember, and hard to brute force.

In any event, you are provided with two input files for this lab:

pass.txt – This is a list of passwords which have been encrypted using SHA-1.

d8.txt – This is plain text dictionary file which contains a list of commonly used passwords.

When you run the program, you first load the dictionary file and then it will decrypt all the passwords that it can.

Note that although the starting template for this project only has a couple of files, it is expected you will add new h/cpp files as appropriate, so that all of your code isn't simply in password.cpp. You will lose points if you just put everything in that one file. Think a little bit about how you want to structure your files when you get to part 2 and 3.

And don't forget that since you'll be adding in new files, you need to add those files to SVN also!

Part 1: Basic Hashing

As with the last lab, we will have a simple text-based menu user interface. You can implement this in password.cpp, and you should use the same stringstream approach you used in the last lab to ensure that incorrect input does not cause odd program behavior. The menu options should look something like this:

```
Select an option:
1. Basic Hashing
2. Load Dictionary
3. Decrypt
4. Exit
>
```

The first option is relatively simple. You ask the user for a sample password, and then the program will output the SHA-1 hash of that password.

To compute the SHA-1 hashes, we will be using a simple open source library for this purpose. This is declared in sha1.h and implemented in sha1.cpp.

There are only two functions in this library. To calculate the hash, you use `sha1::calc`, which takes three parameters (pointer to data, number of bytes of the data, and the pointer to where you want to store the data). For example, if you wanted to hash the string `"abcd"` you would do:

```
unsigned char hash[20];
sha1::calc("abcd", 4, hash);
```

This will store the 20 unsigned chars into hash. But if we want to display this as a hexadecimal string we can view, you can use the following code:

```
char hex_str[41];
sha1::toHexString(hash, hex_str);
```

This would store the string “81fe8bfe87576c3ecb22426f8e57847382917acf” into `hex_str`.

Note that the size of `hex_str` is 41. That’s because even though the SHA-1 hash is only 40 characters, you need one more byte to store the ‘\0’ to mark the end of a C-style string.

Overall, part 1 should not take very long at all to implement. Here are a couple of sample passwords and what they should output:

```
>abcd
Hashed: 81fe8bfe87576c3ecb22426f8e57847382917acf
>password
Hashed: 5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8
```

In any event, once you are using the hashing function properly, you can move on to part 2.

Part 2: Dictionary Lookup

Now that you know the basics of how to hash, we can add code to load in our dictionary file and convert each word into a corresponding hash. When the user selects option #2 from the main menu, you should ask the user if they want to use the default dictionary file name (d8.txt), or if they want to use their own custom file name.

Once you know which dictionary file to use, you will write code which will load in and hash all the plain text passwords in d8.txt. We will then store these hashed/plain text password pairs into a hash table, so that it will be easy to figure out if any of the passwords in our hashed password file match the plain text words in our dictionary.

Processing the Dictionary

Once the user specifies the dictionary file name, it’s time to load it into memory. We need to read in the dictionary line by line, and then hash that password using the SHA-1 functions from before.

As for storing the dictionary information, you want to have some sort of basic `struct` that can store the information for a particular entry. The important information would be the unencrypted (plain text) word, which you could use a std::string for, and then the necessary arrays for both the hash and hex string form of the hash.

Once you have the `struct`, you can write the actual code to read in every word from the file, create a dictionary entry for it, and hash it.

As for where you should store the entries, I'd suggest using an `std::unordered_map`. This is a new data structure added in C++11. It's basically a hash map, just a different name. Just like a regular std::map, it has a key/value pair. However, unlike a `std::map`, it is not ordered, and instead of being a red/black tree, it uses a bucket style hash table implementation.

Your key in this case should be the hexadecimal string (as a `std::string`) and the value should be a ***pointer*** to your corresponding data structure for that entry. Note that behind the scenes, `std::unordered_map` will hash your string key value into an integer. But it retains the actual string in case there are any hash collisions.

One more thing about `std::unordered_map` in this case. It's going to start out empty, with a very small number of buckets. But as we parse the dictionary, we are going to increase the number of buckets over and over, which copies an unnecessary amount of data over and over. Since we know we are going to have over 100,000 entries, our best bet is to resize the hash table before we add anything to it. To do this, you can call the `rehash` member function with `100000` as the size. This provides a huge performance boost over it dynamically trying to resize itself.

The last thing you want to do is add some timing code, just to see how long the process takes. At the beginning of your process code, add the following:

```
LARGE_INTEGER freq, before, after;
QueryPerformanceFrequency(&freq);
QueryPerformanceCounter(&before);
```

Then, at the end, add this:

```
QueryPerformanceCounter(&after);
float fElapsed = static_cast<float>(after.QuadPart - before.QuadPart) /
                 freq.QuadPart;
```

The `fElapsed` variable will then store the number of seconds (as a float) that it took to load in the dictionary into our hash map. You will want to output this value to the console once the dictionary has been loaded. On my system, loading in the dictionary takes about 4 seconds in Debug, but is much, much, faster in Release.

Once your dictionary is correctly loaded, we can move on to the next part. We're going to try to match the hashed dictionary words with those in the password file.

Decrypting with the Dictionary

Once the user loads in the dictionary, they should be able to return to the main menu and select option #3 to Decrypt passwords. When the user selects the Decrypt option, you should ask them if they wish to use the default password file (pass.txt) or a different file, in which case they can specify the name.

At this point, you should read in the password file. Each line you read in will contain the hexadecimal hash of a particular password. You want to look up this hash in your `std::unordered_map` using the `find` member function, which returns an iterator. This iterator will either be equal to end (which means it was not found), or it will point to the correct entry in your dictionary.

All the solved/unsolved passwords will be output to a separate file. We will just output these by default to pass_solved.txt. If you successfully find a password in the dictionary, you should output the hexadecimal hash, followed by a comma, followed by the plain text password. If you fail to find a particular password in the dictionary, to the right of the comma simply output a couple of question marks.

Anyway, once you run the setup you'll notice the decryption is incredibly fast. You'll discover almost all 5,000 passwords in a split second. The first 10 passwords, and a few other passwords, however, will show up as ?? since you haven't solved them yet. But the 11th password should be:

```
5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8,password
```

The worst possible password in existence!

So it turns out the majority of users using this fictitious system used awful passwords. Sadly, this is actually the case in most real systems, too. But there were a handful of users who had passwords which weren't in the dictionary. So to figure out the passwords, we will try brute forcing them.

Part 3: Brute Force

We will now change option #3 in the main menu so that first the dictionary lookup is attempted on all of the passwords. The handful of passwords which could not be solved with the dictionary lookup will then be brute forced. Note that you should **NOT** run the brute force algorithm on every single password in the password file. Only run it on the passwords which were not in the dictionary, or you'll waste a lot of computation time.

In order for us to use the concurrency runtime effectively, we need to separate the passwords we need to brute force from those we have already solved. And since we need the output order in pass_solved.txt to correspond to the order in pass.txt, we can't just immediately output the text was we did in Part 2.

So you'll want to make a new data structure to store the decrypted information. It needs to store the password entry number, the hexadecimal hash, and the plain text solution. As you attempt to solve passwords using the dictionary lookup, you will separate them out into two containers.

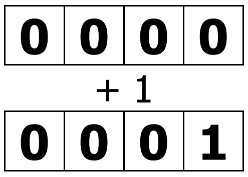
Those you solve with the lookup will be placed in a `std::map` (key should be the entry number, value should be a pointer to your data structure). Those which are not in the dictionary will be placed in a temporary `std::list`. After you're done attempting to dictionary lookup all the passwords, we will then brute force the remaining passwords in the list.

As a reminder, you want to brute force passwords up to and including a length of 4. So you need to first do length of 1, then 2, then 3, and finally 4.

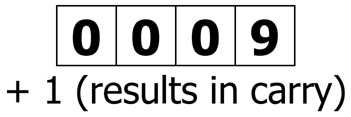
The Algorithm

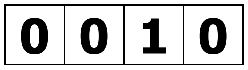
The premise of brute forcing is literally trying every possible combination. In our case, since we're limiting it to lower case letters and numerals, there's 36 possibilities to consider per character (26 letters and 10 numerals).

But for now, let's think of a simpler case. Suppose you have a 4 digit PIN number. Each digit can be a number from 0 to 9. If we wanted to test every number, we would start at 0000, and keep counting up by one:



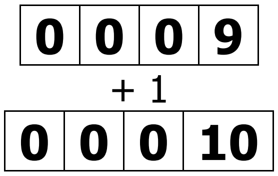
We would continue incrementing the "ones" digit until it gets to 0009. At this point, one more iteration would cause a carry to occur to the "tens" digit. So:



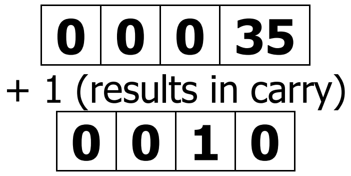


If this seems like elementary math...well, that's because it is. So right now we have effectively a counting machine that increments by one in base 10. But who says we're required to use base 10? Why couldn't we make a counting machine in base 36?

If we were in base 36, each digit can have a value in the range from 0 to 35. So if we go back to the example of 0009, let's see what happens:



Now we would keep going until 0,0,0,35, at which point then finally a carry would occur
In these scenarios, I've found the best way is to abstract away the characters, and instead just think of it as numbers. You are just going from 0 through 35 on each character. Then you can make a helper function that converts the number to the appropriate character.



So what does this all mean? Well, it means we need to implement a counting machine in base 36. Then, we just need code that can convert something like 0,0,0,25 to the string "aaaz".

What I'd suggest doing here is assigning 0-25 to the letters 'a' through 'z', and then 26-35 to the numbers '0' through '9'.

So essentially, create an array of integers, one index for each digit, and keep counting in the "ones" digit until you hit 35, at which point you carry to the "tens" digit. Then separately, make a function which can convert a single digit into a corresponding character. Then it's just a matter of reading in the digits from the array and converting it into a string.

Make sure you wrap your head around the "counting" algorithm before you sit down and try coding the brute force.

Once you have this string, you can hash it with SHA-1 and compare it against the uncracked password. If it's unsuccessful, then you increment by one and try the next password. This way you guarantee that you'll go through every permutation.

I would also strongly suggest that you add some test code to test out the brute force algorithm on one password you hash yourself. That way you can run in debug mode to find any errors in the brute force implementation before you move on to trying it on every password.

Brute Forcing Multiple Passwords

One thing to note is that you'll want to loop through the list of unsolved passwords using `std::for_each` and a lambda expression. Recall that `std::for_each` is in `<algorithm>`. As a reminder, make sure you capture all the variables you need to for the lambda. Especially remember that if you need to use other member functions, you have to capture `this` by value.

You should also take the performance counter timing code that you added in Part 2 and wrap it around the `std::for_each`. That way you can see how long the brute force took with and without the Concurrency runtime.

Once you are brute forcing multiple passwords, you should run in release mode. This is because running in debug mode is painfully slow. That's why you should have tested the brute force algorithm first on one password!

If you look at "pass_solved.txt," you'll notice nearly all the passwords get solved. The only ones which should be unsolved are on lines 4, 25, 28, 32, 38, 44, 46. On Blackboard, I have posted a pass_solved.txt with the reference output that you should get when you are done with this.

Concurrency Runtime

Now it's time to see the awesome (and simple) power of the concurrency runtime. All you need to do is include `<ppl.h>`. Change your `std::for_each` to `Concurrency::parallel_for_each`. That's it. And you can watch your time improve!

Remember that in the `parallel_for_each`, there's no guarantee what order the loop will complete. So make sure you don't have any code that's reliant on it. And make sure any temporary data is declared within the scope of the loop, so you don't have a race condition.

It's worth noting that this algorithm is not the most optimized way to solve this problem. We're recalculating the SHA-1 for every permutation for every password. A better way to do this would be to instead calculate each SHA-1 only once, and check that against every password. But this isn't possible to do with just `parallel_for_each`. You'd have to use a more advanced feature: `parallel_invoke`.

If you implement it in this manner, you can get a fairly significant improvement. As a test, I increased the number of characters in the brute force to 5. It took about 60 seconds on my machine using `parallel_for_each` but only 8.5 using my `parallel_invoke` implementation (in release mode).

For this lab, though, it's okay if you stick with the `parallel_for_each` method. If you have extra time, you can try doing it with `parallel_invoke`. But it's not a requirement by any means.

Expected Output

Nearly all passwords should come up as solved in pass_solved.txt. However, the passwords on the following lines will remain unsolved: 1, 25, 28, 32, 38, 44, 46. They picked slightly more challenging passwords, it seems!

Closing Thoughts

Part of professional coding, especially when you have a user interface, is to make sure your edge cases are handled. In our case, there's a few error cases to handle:

1. Trying to decrypt before loading dictionary.
2. Trying to reload the same dictionary.
3. Invalid dictionary file name.
4. Invalid password file name.

Make sure you handle these cases in your code. Also make sure you aren't leaking memory!

Lab 3: Drawing is Fun!!

WTL (Windows Template Library) is an extension to ATL (Active Template Library), which were developed by Microsoft as an alternative to MFC (Microsoft Foundation Classes). MFC was the first attempt to provide a C++ API for Windows programming. Unfortunately, it resulted in often bloated (and ugly) code.

WTL is an improvement over MFC, but it's not perfect and takes some getting used to. It doesn't help that it's unsupported by Microsoft, and due to that documentation is pretty difficult to find. Thankfully, we will only really need to use a limited subset of WTL to get our user interface working.

For this lab, you are going to need to re-check out for SVN. The new directory you want to check out from is:

`http://itpsvn.usc.edu/svn/itp/20141/435`

When you check out from this URL, you will have two additional directories show up in addition to your personal one. You'll still be editing the code in your personal directory, however.

Inside your personal directory, you should have a "draw" subdirectory which contains the files for this lab. Before you get started on the lab, let's discuss the idea of messages. In Windows, your program receives messages for any number of things which can happen – such as mouse clicks, resizing the window, minimize, etc. In a Windows application, you must specify which of these particular messages you are interested in processing, so that Windows knows your app is ignoring the other ones.

In WTL, you specify which messages you will respond to inside message maps. Open up "drawView.h" and look at the `BEGIN_MSG_MAP` macro. Using this series of macros, each message is registered to a corresponding member function. For example, right now, our program is registered to four messages:

`WM_PAINT` – This message is sent when the window needs to be redrawn.

`WM_LBUTTONDOWN` – This message is sent when you first press down the left mouse button.

`WM_LBUTTONUP` – This message is sent when you release a previously pressed left mouse button.

`WM_MOUSEMOVE` – This message is sent every time the mouse is moved inside the bounds of the window.

Each of these messages then is processed by the corresponding member function which is listed inside the message map. Later in this lab, you will need to process additional messages. When you get to this point, you will need to add new member functions and additional lines to the message map.

Similar to messages, there is an idea of a `COMMAND_ID_HANDLER`. These are application-specified messages that the program can create and process. For example, an Undo command might generate `ID_EDIT_UNDO`. These types of handlers are defined in MainFrm.h.

One other thing I want to note before starting on the lab. There is a special header file in this project called stdafx.h. **You must include stdafx.h at the beginning of any new cpp file you create. If you do not, bad things will happen**.

This is because of the way precompiled headers work in Visual Studio. Precompiled headers is a way that the compiler can avoid recompiling the header file in every cpp it's used in.

Part 1: Getting Started w/ GDI+

In this lab, we are going to use GDI+ to create a shape drawing program that will let us draw things such as lines, circles, rectangles, and so on. Sort of like Microsoft Paint, but with not as much functionality. This is primarily to leverage some of the design patterns we have learned, but also to try to learn a bit about making sure our code isn't inefficient.

GDI+ is a C++ interface implemented by Microsoft for drawing a wide variety of objects. Before we get started, make sure you keep the following reference handy: http://msdn.microsoft.com/en-us/library/ms533798(v=vs.85).aspx. If at any point you get confused with something related to GDI+, make sure you check the extremely detailed reference pages.

The minimum things you need to draw something is a `Graphics` class and a `Pen` class which describes what color, thickness, and style you want to draw in.

In the `drawView` class, there's already a `Graphics` class called `m_GraphicsImage`. This is the class you want to draw all your images to. That's because this particular class is linked to the `Bitmap m_BitmapImage`. The reason we have this link is two-fold: first, it will allow us to load an image file and then draw on top of it. Second, it will allow us to save off whatever we draw into an image file.

An additional benefit of this is that in the `OnPaint` function, rather than having to redraw everything shape by shape, we just have to redraw the bitmap that's cached already. This is a performance increase than unnecessarily redrawing every shape.

However, when we do an undo (as we'll implement later), you will need to clear out `m_GraphicsImage` and redraw every shape in the order they were initially drawn.

Anyway, if you look at drawView.cpp, you'll see right now all that happens is that the constructor draws a line from (0,0) to (200,200) in blue. Go ahead and comment out this code, as we're going to draw functionality which draws lines when you click and drag the mouse.

Click and Draw Lines

So to click and drag to draw lines, we need to implement code in `OnLButtonDown` and `OnLButtonUp`.

When you press down the left mouse button, you should save off the location of the mouse. This will be the start point. Then when you release the mouse button, you can grab the end location and then draw the line to `m_GraphicsImage`.

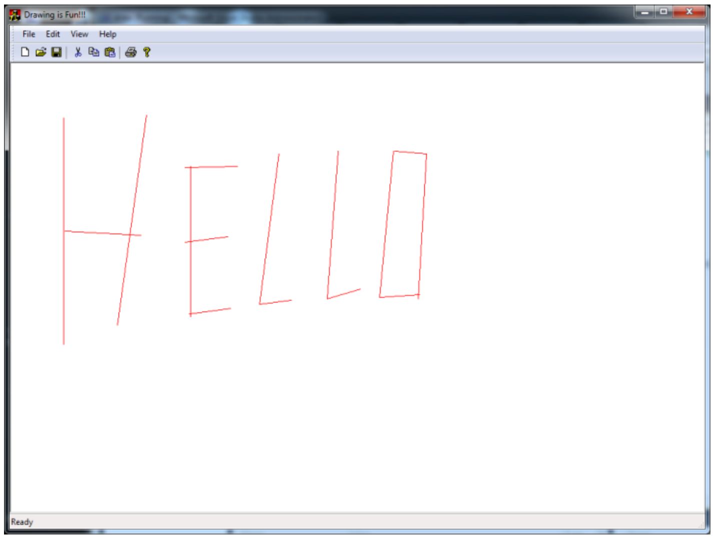
To get the location of the mouse from any of these mouse message functions, you need to grab it from the lParam as such:

```
int xPos = GET_X_LPARAM(lParam);
int yPos = GET_Y_LPARAM(lParam);
```

These values could then be saved into a `Gdiplus::Point` member variable so you can later use them.

Once you have the start and end point of the line, you just need to create a Pen and draw the line (just as was done in the commented out constructor). The order of the colors specified to the Color constructor are (Red, Green, Blue). Make sure after you draw to `m_GraphicsImage`, you also call `RedrawWindow` (or the new line won't be visible).

Anyway, if you implement this properly you should be able to click and draw lines all over the place as below. I chose to just make the lines red so they stick out:



However, there’s a problem with this implementation. While you’re in the process of drawing the line, our program doesn’t show you what the line will look like. You have to release the mouse button and then hope the line looks like how you want it to. We’ll fix this in a bit.

Clearing the Image

It would be nice to be able to delete everything we’ve drawn, in case we want to draw new stuff. So to do this, you need to add code to the `OnFileNew` function that’s already in MainFrm.cpp. In here, simply call the Clear function on `m_GraphicsImage` and pass in the Color white (all 255s).

This way, when you select File>New or click the New button on the toolbar, it will clear out the image.

Changing the Color of the Pen

Instead of just creating the Pen every time we want to draw a line, and just using one set color, we want to be able to change it. So you should add a `Gdiplus::Pen` as a member variable to `CDrawView`. In the constructor’s initialization list, just initialize the `Pen` to black (0,0,0). You also want to change your line drawing code so it uses this new `Pen` object.

In order to do this, we need to add a new Pen menu where the different options can be selected. This involves editing what are known as *resources* in Windows applications. Resources can include things like menus, custom dialog boxes, icons, and images that we want to actually be included inside the EXE file itself. So rather than being in separate files which the end user might be able to easily edit, it’s all hidden in the EXE which raises the barrier a little bit. Though an experienced user can still get to them, of course.

Visual studio has editing functionality that's specifically designed for these resources. To open the Resource View, go to View>Resource View. This will replace the Solution Explorer on the left column with the Resource View. Inside this view, expand draw.rc and select the Menu folder. In here, open up `IDR_MAINFRAME`. This is the standard File/Edit/etc. menu at the top of the window.

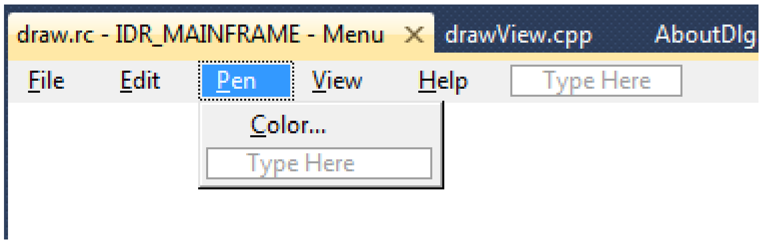
First, you should remove all the menu items related to printing. We aren't going to support that for now. To remove a menu item, simply select it and hit delete.

Then in the Edit menu, right click on "Cut" and select "Properties." This should bring up a new properties window in the bottom right part of the screen which shows various aspects of the menu item. In the properties, change the "Enabled" property to false. That's because for now, we aren't going to support Cut, Copy, and Paste. So go ahead and disable those other two commands, as well.

Now we are going to add a new Pen menu, which will allow us to change the color and thickness of the lines and shapes we draw. To add a new menu, click on the "Type Here" area to the right of the Help menu. In here, you want to type in "&Pen". The ampersand tells window that the P is the shortcut letter, so if you hit Alt+P, it will open up the Pen menu. Drag the menu to the left of the "View" one, since it doesn't make sense for it to be the last menu item.

Then under the Pen menu, add a "&Color..." entry. Look at its properties and make sure the ID is `ID_PEN_COLOR` (which it should be, by default). This corresponds to the command ID we will declare a handler for.

If you've done this correctly, your menu should look something like this:



Once we are done with the resources, it's time to go back to the Solution Explorer. To get the explorer to show up again, go to View>Solution Explorer.

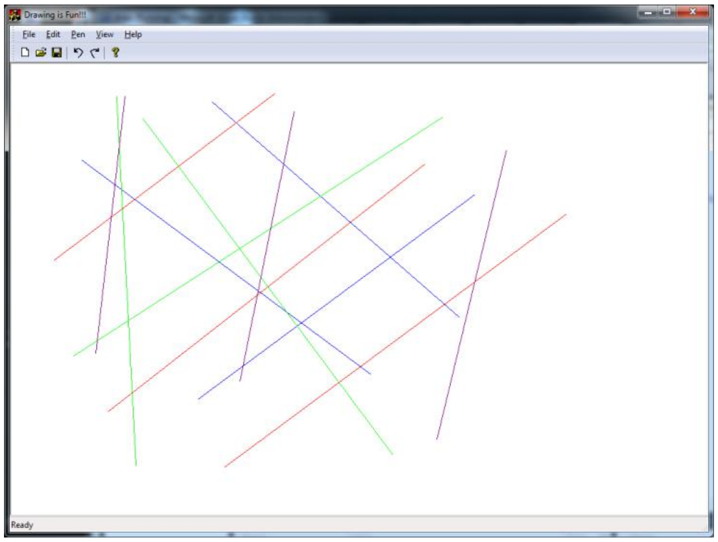
To declare our handler, open up MainForm.h and add a member function with a function signature identical to `OnAppAbout`. Then right under the line that registers the handler for `ID_APP_ABOUT`, register a handler for your `ID_PEN_COLOR`.

We want the Pen>Color menu option to bring up the standard Windows dialog box for selecting a color. This is encapsulated by WTL in the `CColorDialog` class. Here's code of how it roughly works:

```
// Initialize this to current color
COLORREF color = RGB(0,0,0); // FIXME TO BE CURRENT COLOR INSTEAD OF BLACK
// Create dialog box
CColorDialog test(color);
test.DoModal();
// Grab color selected
color = test.GetColor();
Gdiplus::Color newColor(GetRValue(color), GetGValue(color),
     GetBValue(color));
```

Note that this is not the full extent of the code. You need to grab the Pen's current color (using the `GetColor` function) and use that instead of just black, and you will need to set the Pen's new color (using `SetColor`) to be the `newColor` class you created.

Once the pen color changing is implemented, you should be able to draw things like this:



Changing the Width of the Pen

Now that we can change the color, let's change the width of the pen. By default, the width is 1.0. We want to add a submenu to Pen called Width, which lets us set the Pen Width to 0.5, 1.0, 1.5, 2.0, 2.5, and 5.

To make 1.0 checked by default, you want to set the "Checked" property to True.

Now this will end up creating 6 new commands (with IDs by default `ID_WIDTH_0` through `ID_WIDTH_5`). It would be really annoying to have one separate command handler for every single one, since they all have the same functionality other than the fact that the width you want to set is different.

Luckily, you are allowed to map multiple IDs to the same function via the `COMMAND_ID_HANDLER`. So what you want to do is map all 6 of the commands to the same member function. In this function, the way you figure out specifically which `ID_WIDTH` they picked is via the second parameter of a command handler (the `WORD wID` variable).

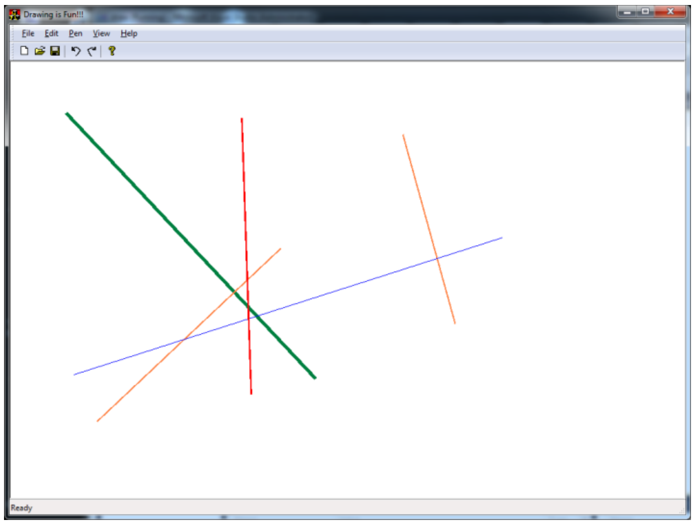
The first thing this function needs to do is remove the checkmark from every single `ID_WIDTH`. Then, you want to add the checkmark to the particular `wID` that was selected.

In order to enable/disable a checkmark, you need code like this:

```
// Remove checkmark from #5 only
m_CmdBar.GetMenu().CheckMenuItem(ID_WIDTH_5, MF_UNCHECKED);
// Add checkmark to wID
m_CmdBar.GetMenu().CheckMenuItem(wID, MF_CHECKED);
```

The next step is to set the Pen's width (via `SetWidth`) to the particular item that was selected.

Once you've done this, you can now not only change the color of the pen, but you can change the widths as well. And you can make designs like this:



That's enough line drawing I'd say. Now that we have the basics of how GDI+ and how adding menu items works, it's time to move on to different shapes and Undo/Redo.

Part 2: Shapes and Undo/Redo

For each shape our program can draw, we'll have a class which stores the information needed to draw the shape (at a minimum this will be the start point, end point, and pen information). These classes will all derive from a base `Shape` class. We'll then have a Factory Method that knows how to construct a particular shape given the current active shape.

Shape and Line Classes

Starting out, the `Shape` class needs the following variables:

1. A `Gdiplus::Point` for the start point.
2. A `Gdiplus::Point` for the end point.
3. A `std::shared_ptr<Gdiplus::Pen>` to store pen information. (include `<memory>`)

It then needs the following functions:

1. A PURE VIRTUAL Draw function, which takes in a `Gdiplus::Graphics` class by reference. This is where all the children classes will overload the data.
2. A way to set the Pen `shared_ptr`
3. A way to set the start point
4. A way to set the end point

You then should create a `Line` class which inherits from your `Shape` class and overloads `Draw` to use the `DrawLine`. Note that `DrawLine` wants a pointer to a `Pen`. In order to get the actual pointer from a `shared_ptr`, you need to use the `get()` member function.

Now you want to add a member variable to `drawView` class: a `shared_ptr` to a `Shape`. This pointer will represent the current shape you're drawing. In `LButtonDown`, you want to create a new `Line` with `make_shared`. You also want to make a new Pen `shared_ptr` in and set that for the Line. The easiest way to do that for now is something like this:

```
std::shared_ptr<Pen> myPen(m_Pen.Clone());
```

Where `m_Pen` is the `Pen` member variable you already had. Use this variable to setup the Pen for the Line you made, and also set its Start point.

Then in `LButtonUp`, you want to set its `End` point and draw the line through the `Shape`. You then can call the `reset()` function on the `shared_ptr` for the `Shape`, which resets it to not pointing at anything. In a bit we will actually add it to a `std::list` before resetting the pointer, but not right now.

Anyway, at this point even though you're using the `Shape` class, the functionality should be identical to what you had previously. Now let's add something new.

Previewing a Line While Drawing

As I mentioned earlier, it would be nice to be able to see the line while we're drawing it. So now let's add this functionality!

The key thing here is to think of `m_GraphicsImage` as the place where finalized shapes are drawn. So once I release the left mouse button, I've finalized the position of the line, and therefore it should be drawn to `m_GraphicsImage`.

However, if an shape is still in the process of being drawn (I'm holding down the left mouse button), it should not be drawn to `m_GraphicsImage`. I don't want those temporary changes to be finalized into the image until I've released the mouse button.

So what we're going to do is add some code into `OnMouseMove`, which is called whenever the mouse is moved (regardless of the button is down or not). First, we only want our code to execute if the `Shape` `shared_ptr` actually is pointing to anything. This is because we only want our special `OnMouseMove` code to execute if there's an active `Shape`, which means we're currently drawing something. Once we call `reset()`, there's no longer an active `Shape` and therefore we don't need our code to execute. Luckily, you can use a `shared_ptr` in a boolean statement just like a regular one.

Then inside the if statement, you need to set the end point of the shape appropriately (use the same macros you used in `LButtonDown`/`Up`). You then want to call `RedrawWindow()` to force an `OnPaint` to occur.

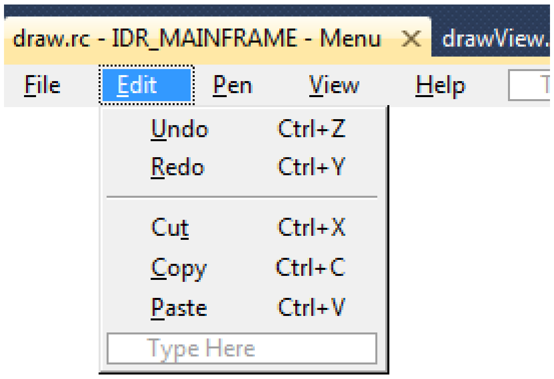
Now take a look at `OnPaint`. What you want to do here is after you draw the `m_BitmapImage`, check if you have a valid `m_Shape`. If you do, then draw that `m_Shape` to the `Graphics` class you created locally here in `OnPaint`.

Unfortunately, you'll definitely notice some flickering when dragging the image. This is because GDI+ is kind of slow. If you can figure out how to make it not flicker, I'm proud of you. In any event, this is a big performance optimization over have to clear and redraw every single shape!

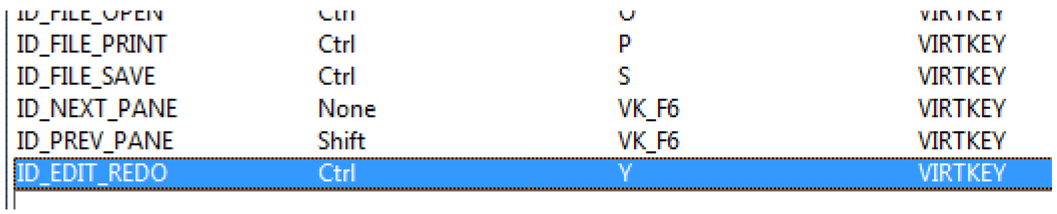
Undo/Redo

Now let's add Undo/Redo functionality so we don't make a rage comic about how we can't undo. In the Edit menu, you want to add a Redo command right under Undo. The redo menu option should be captioned with “&Redo\tCtrl+Y”. Make sure that this redo item is mapped to the ID `ID_EDIT_REDO` in its properties.

Your menu should now look like this:



Now it’s time to add the actual key mapping for Ctrl+Y. To do this, you need to open up the Accelerator folder in the Resource View. You’ll see a list of commands and their corresponding input commands. So you need to add a new one for `ID_EDIT_REDO` that maps to Ctrl and Y. It will look like this:

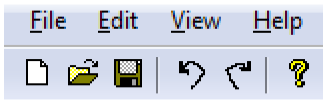


ID_FILE_OPEN	Ctrl	O	VIRTKEY
ID_FILE_PRINT	Ctrl	P	VIRTKEY
ID_FILE_SAVE	Ctrl	S	VIRTKEY
ID_NEXT_PANE	None	VK_F6	VIRTKEY
ID_PREV_PANE	Shift	VK_F6	VIRTKEY
ID_EDIT_REDO	Ctrl	Y	VIRTKEY

The final thing to consider when editing the user interface is the toolbar. To edit this, choose the Toolbar folder in Resource View. From here, to remove a button, simply drag it off of the toolbar. To add a button, click on the empty button space to the right, and set an ID for it. The ID in will correspond to an ID that’s used for one of the regular menus.

So you want to go ahead and remove the buttons for cut, copy, paste, and print. And you want to add buttons for undo and redo that map to `ID_EDIT_UNDO` and `ID_EDIT_REDO`, respectively. As for what the buttons should look like, that’s up to you. It doesn’t have to be fancy at all, and the editor lets you draw then with some simple drawing tools.

I’m not very good at drawing stuff, so my buttons ended up like this:



Now that we have the menus for it, let's implement Undo/Redo. You will need to add two `std::list`s which contain `shared_ptr`s to `Shape`s. One will be your "Undo" linked list, the other will be for "Redo." Whenever you "finalize" a shape in `LButtonUp`, before you call reset on the Shape `shared_ptr`, add the `shared_ptr` to the Undo linked list via `push_back`.

Then if an Undo command is given, what you'll do is grab the `shared_ptr` at the back of the Undo linked list and move it to the Redo linked list. Then, clear out the `m_GraphicsImage` to white (just as you did in File>New) and iterate through the Undo list, redrawing everything.

If a Redo command is given, simply grab the last shape pushed into the Redo linked list and move it back to the Undo list, and just draw that one single shape to `m_GraphicsImage`.

In each case, make sure you call `RedrawWindow()`. Also, make sure that if someone selects New, you clear out both lists. Finally, you need to clear the Redo list when a new shape is drawn.

You will have to add `COMMAND_ID_HANDLER`s for both commands into the `MainFrame`. Once you've implemented this, you should be able to Undo and Redo drawing your Lines.

More Shapes

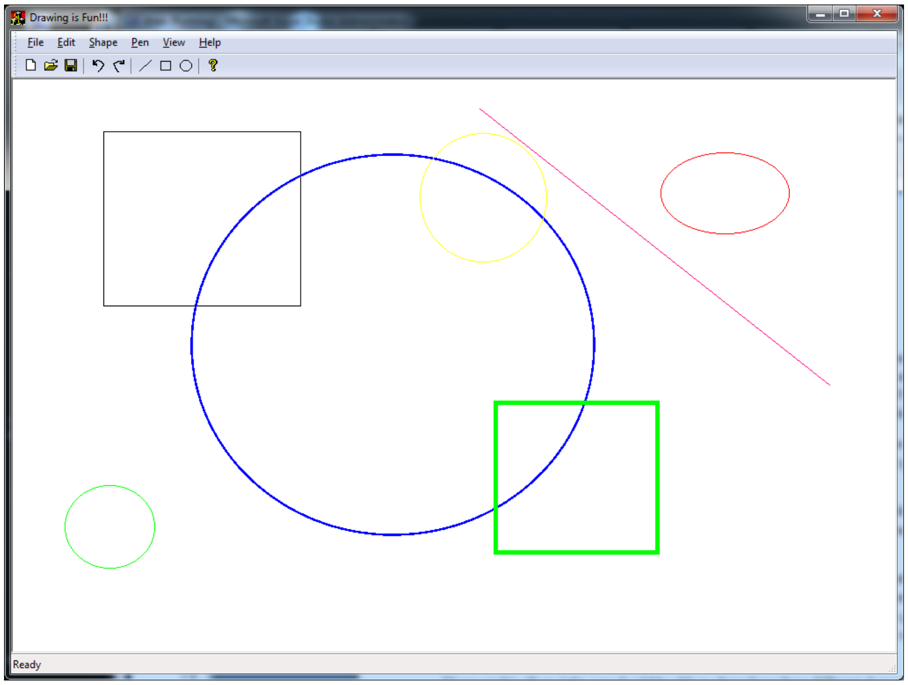
So if you're tired of drawing lines, you're in luck. Now we are going to add support for drawing two more shape types: an ellipse and a rectangle. You'll notice that `DrawEllipse` and `DrawRectangle` expect you to pass in a top left point and then the width/height of the shape. It doesn't accept a start and end point. But that's okay, because you can calculate this.

The X value of the top left point is whichever is smaller from the start and end's X. The Y value of the top left point is whichever is smaller from the start and end's Y. Once you have those, you should be able to calculate the width/height of the shape by figuring out the bottom right point.

You also need to add a new "Shape" menu to the top menu that allows you to pick between a checkbox for Line, Ellipse, and Rectangle. You also want to add corresponding buttons to the toolbar, so that it's easy to change between different shapes on the fly.

Add appropriate command handlers to the main frame, and setup a `enum` or something in the view that keeps track of what shape type currently is drawing. Then make a Factory Method (in a separate class as a static function) which knows how to creates a particular shape given the shape `enum`. When the user then clicks the left mouse button, you'll invoke the Factory method to create the Shape `shared_ptr` you need and the rest should just work.

If you set this all up right, you should be able to draw these three different shape types in a wide variety of colors and widths.



Part 3: Saving/Loading

Saving Your Beautiful Drawing

So you're going to need to add `COMMAND_ID_HANDLER`s for "Save" and "Save as..." into the `MainFrame` class. One thing you need to track is whether you've previously saved the file once (or it was loaded from an existing file). If you haven't, the Save command should cause a Save as... to occur.

Getting a file open dialog box to work in WTL is much, much easier than in the default Win32 API. You can just create a local instance of `CFileDialog` class and with some minor configuration you're good to go.

Here's some examples of how to use the class:

http://www.codeproject.com/Articles/12999/WTL-for-MFC-Programmers-Part-IX-GDI-Classes-Common#usingcfiledialog

So we want to create a `CFileDialog` that does not require the file to already exist, and allows you to save as a .PNG file.

Once you get the string of the file name from the `CFileDialog`, we can write out the PNG file. The code for this is kind of ugly, but it would look something like this:

```
CLSID pngClsid;
CLSIDFromString(L"{557cf406-1a04-11d3-9a73-0000f81ef32e}", &pngClsid);
m_BitmapImage.Save(CA2W("test.png"), &pngClsid); // FIXME Change file name!!
```

Obviously, variables will differ depending on what you named them. Instead of test.png, you'll want to use whatever file name you get through the dialog box.

In any event, this will allow you to save your amazing drawing for future generations.

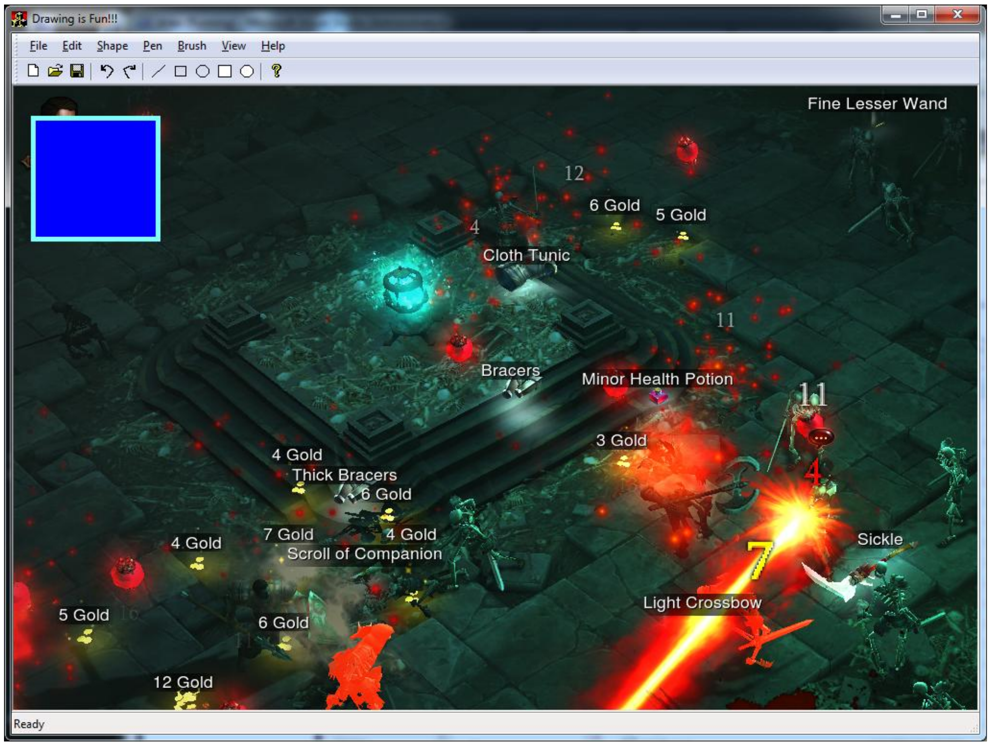
Loading an Existing File

First, you'll have to make a new handler for File>Open. In order to open a file, what we'll do is first load in a new bitmap, and then draw that bitmap onto `GraphicsImage`, as such:

```
Gdiplus::Bitmap myFile(CA2W("file.png")); // CHANGE THIS
m_view.m_GraphicsImage.DrawImage(&myFile, 0, 0);
```

Of course, as before make sure you have the actual file name the user selected in the box instead of "file.png." When you load a file, you should also clear out the Undo/Redo stacks.

This isn't a perfect solution, however. If the image is a size larger than 1024x768, it gets cropped to that size. But that's good enough for our purposes.



There is another issue, as well. If you load an image, draw a shape, and then "Undo," the original image will go away. There is a way you can work around this, but it was a bit of a headache and this lab is long enough as it is!

Part 4: Flyweight

The final thing we're going to do in this lab is optimize the way we use Pens. To do this, we're going to use another design pattern, and some templates too.

Reusing Pens

The way most people will use the drawing program, the same Pen is going to be in many cases reused multiple times to draw several shapes. In our current implementation, we are storing a separate copy of this Pen every single time, which is inefficient. If the same Pen is being used 100 times, it doesn't make sense to store 100 copies of it. If this sounds like a Flyweight, you would be right! And not only will this be a Flyweight, we're also going to make it a Singleton so it can be accessed from any file, if necessary.

I have provided the templated Singleton class as outlined in the Design Pattern (Week 8) lecture notes in singleton.h. In order to use this templated class, there are two things you need to do. Your flyweight class needs to inherit from it, with the templated type being the flyweight class, AND inside the declaration of your templated type, you need to invoke the `DECLARE_SINGLETON` macro with the name of your flyweight class. So for example, if your class is PenWeight, you would do something like this:

```
class PenWeight : public Singleton<PenWeight>
{
    DECLARE_SINGLETON(PenWeight);
};
```

Then, wherever you want to call a function the `PenWeight` Singleton has, you'd use this:

```
PenWeight::get().MyFunction();
```

So once you have your Singleton declared, you need to make a `GetPen` function which returns a `shared_ptr` to a `Pen`. The `GetPen` function should first check if the `Pen` of that color and width already exists; if it does, it should return the `shared_ptr` to that particular `Pen`. If it doesn't already exist, it should create a `shared_ptr` for the new Pen.

To track which Pens do and don't already exist, you could use an `unordered_map` where the key is a `std::string` and the value is a `shared_ptr` to a Pen. In order to use a string as the key, though, you need some way to be able to create a string out of the properties of the Pen. A simple approach would be just to generate a string as such: `"(R,G,B);W"` where R, G, B correspond to the colors and W corresponds to the width of the Pen (note since W is a float, make sure you only output 1 decimal place after the point).

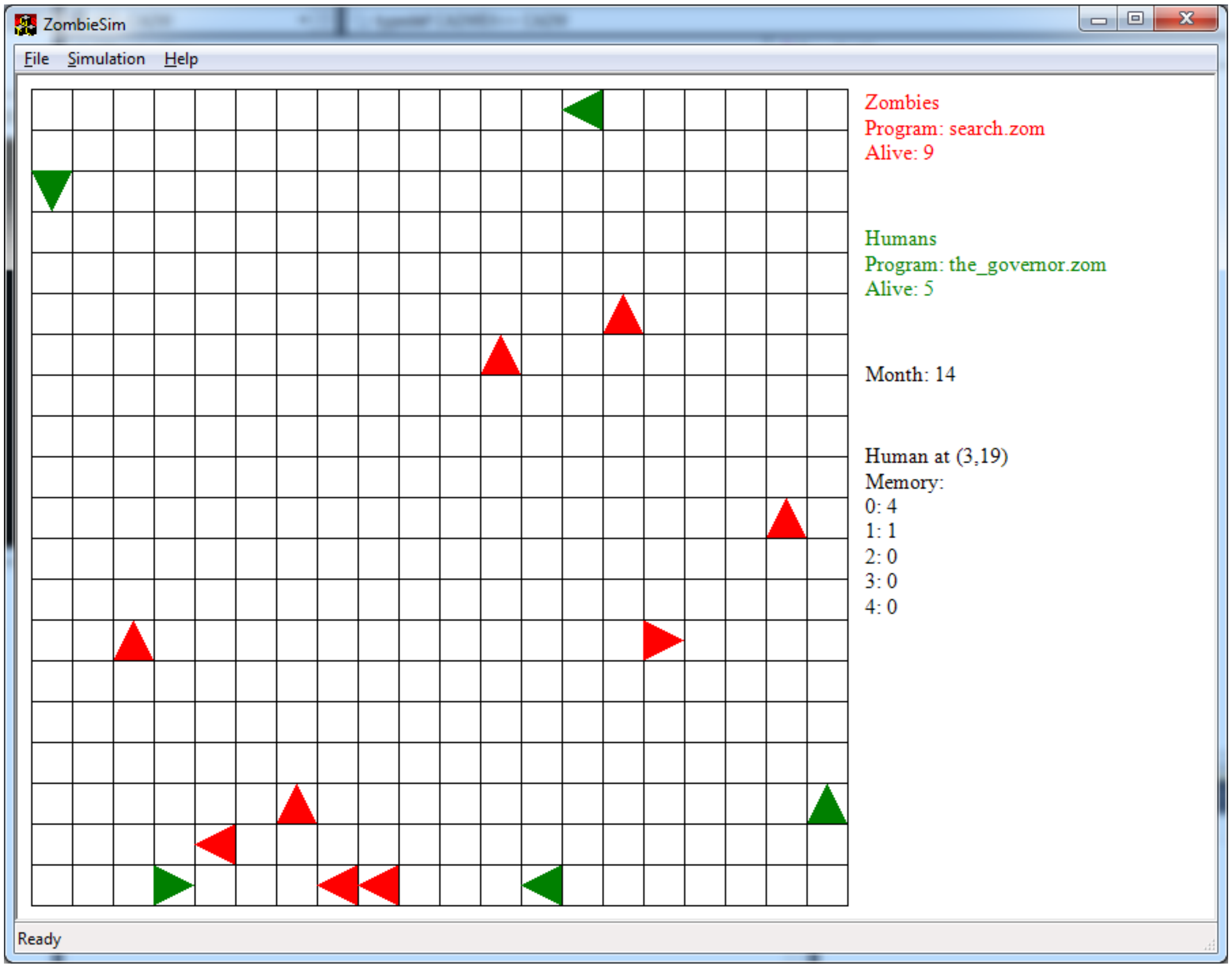
Then you do a lookup by that string. If it's already in the map, that means you already created a Pen of that type, so just return the `shared_ptr` to it. If it isn't already in the map, you will create a new Pen `shared_ptr` via `make_shared`, set the color and weight, and add that to the `unordered_map`. Then next time around, if you try to lookup by the same Pen property string, it should find it in the map.

In any event, once you implement the flyweight, you are done with this lab!

Lab 4: ZombieSim

In this lab we are going to make a zombie simulation program. The idea for this is similar to "Darwin's World" which was developed by Nick Parlante at Stanford. That was definitely the inspiration, and some of the design is similar. But this lab has taken on a mind (and brains) of its own!

The basic idea is that there are zombies and humans fighting for dominance a post-apocalyptic zombie world. Rather than hard-coding the behavior of the zombies and humans, their behavior is off-loaded to ".zom" files. So for instance, "daryl_dixon.zom" is a pretty awesome ranged/melee zombie killer while "michonne.zom" only does melee attacks.



Zombies
Program: search.zom
Alive: 9

Humans
Program: the_governor.zom
Alive: 5

Month: 14

Human at (3,19)
Memory:
0: 4
1: 1
2: 0
3: 0
4: 0

There are three main aspects of this lab. The first is parsing the .zom file format and generating the correct set of instructions from it. Once those files are parsed, then you also need the code that can actually execute those instructions. This is intended to take the majority of the total time of this lab.

Then hand-in-hand with the parsing is maintaining the overall simulation. For this, you will be randomly placing zombie and humans in the world, and then running the virtual machine simulation for every zombie and human until either the zombies or humans win. Once the execution is working for one zombie or human, this part shouldn't be that much more work. Drawing will be done with GDI+, like in the last lab.

The final part is experimentation! You will make a couple of your own custom ZOM files to see how they perform.

Before we start coding, there is one important thing to note. **We are going to use exceptions**. This means you need to throw exceptions for things the user might do (such as try to pass in an invalid .zom file) **and** what a ZOM simulation might do (such as trying to access memory outside its bounds). All these exceptions also need to be caught appropriately. If a ZOM tries to perform an invalid operation, it should just die. The whole simulation shouldn't come crashing down.

Part of your grade is how well you used exceptions. Make sure you create appropriate exceptions in the exceptions.h header file. Don't just use `std::exception` everywhere. You also should use empty throw specifiers for functions that do not throw exceptions. So for example:

```
void DoesntThrowExceptions() throw();
```

Don't try to use throw specifiers where you list out the exceptions that it can throw, because those don't work in Visual Studio. Either use `throw()` if the function can't throw exceptions, or don't if the function can throw exceptions.

ZOM Framework

Before we delve into the specific code files, let's discuss the backbone of ZomSim, which I'm calling the ZOM framework. The idea is this: there are zombies and humans. Humans can perform actions more often than zombies (because they can use vehicles and other tools), and they also have a limited amount of memory, since they are still (mostly) sentient. Zombies can't perform as many actions, and have no memory. Humans also can do a ranged attack.

Zombies have one advantage over humans, though. If a zombie attacks a human, the human becomes a zombie and then the zombie program takes over. However, if a human kills a zombie, the zombie just disappears from the world. So eventually, the zombies have a decent shot of winning. But the humans will try to hold out as long as possible!

There is a grid of 20x20 squares. Each square can either be empty, have a zombie, or have a human. A square cannot be occupied by more than one human or zombie at any one time. But with 400 squares, there's a lot of space!

Zombies/humans face in one of the four cardinal directions. They can only move and attack in the direction they're facing. The rotate action allows the zombie/human to rotate either clockwise or counter-clockwise. The border around the grid is the "wall" so zombies/humans can't move outside the walls.

This is a turn-based simulation. Every turn, a zombie can only perform one "action" per turn. Actions generally are things like attacking, moving, rotating, etc. Humans can perform two actions per turn. Any programming logic such as conditionals, branches, etc., **do not count as actions**. So if a particular ZOM file has a lot of conditions/branches, you can potentially perform a lot of ops in one turn, but only one (or two) actions. If a user tries to do something invalid in the ZOM program, you should throw an exception. That exception can be caught and you can display a nice error message dialog using the `MessageBox` function. That looks something like this:

```
MessageBox("Zombies can't use memory!", "Error", MB_OK);
```

Anyway, here is a small sample ZOM file. It's in stationary.zom:

```
test_human,1; Is this a human?
je,5        ; If so, jump to line 5
rotate,0    ; Rotate clockwise
goto,1      ; Goto line 1
attack      ; Try to eat the human!
goto,1      ; Goto line 1
```

Anything after a semi-colon is a comment, so the parsing code should ignore it. Notice how some ops, like `test_human` have no parameters, whereas others have a parameter, with a comma separating the op and the parameter. This should give you an idea of what the ZOM language looks like. You are only allowed to have one op per line. And line numbers matter, because branches will always specify the line number.

Below are the list of the different actions and other ops that the ZOM language supports.

Action Ops (one action point each!!)

`attack`	Try to kill whatever is right in front of you. If a zombie attacks a human, the human becomes a zombie. If a human attacks a zombie, the zombie dies. If a zombie attacks a zombie, nothing happens. If a human attacks a human

	(bandits!) the human being attacked dies. If you try to attack a wall or there's nothing in front, it just wastes the action.
`ranged_attack`	Attack an two tiles in front of you with a ranged attack. Only humans can use a ranged attack. Whatever a human attacks dies. If there's nothing to attack there, it wastes the action.
`rotate,n`	If n=0, rotate 90° clockwise, otherwise counter-clockwise.
`forward`	Move forward, if possible. If the movement is invalid (wall or otherwise occupied), it consumes an action but nothing else happens.
`endturn`	Automatically ends the turn, even if there are more actions left.

Other Ops

`test_human,n`	Set test flag to true if a human is n tiles in front, where n can be either 1 or 2.
`test_wall`	Set test flag to true if facing a wall.
`test_zombie,n`	Set test flag to true if a zombie is n tiles in front, where n can be either 1 or 2.
`test_random`	Randomly set test flag to true or false.
`test_passable`	Set test flag to true if facing an open tile.
`je,n`	If test flag is true, jump to line number n. Otherwise continue to next line of execution. Throw exception if line n is invalid.
`jne,n`	Exactly like je, but jumps if the flag is false.
`goto,n`	Automatically go to line number n. Throws exception if line n is invalid.
`mem,n`	Set the current memory slot to memory slot n. Throws exception if memory slot n is invalid or attempted by zombie.
`set,n`	Write the value n to the current memory slot. Throws exception if there is no current memory slot or attempted by zombie.
`inc`	Increment the value in the current memory slot by 1. Throws exception if there is no current memory slot or attempted by zombie.
dec	Decrement the value in the current memory slot by 1. Throws exception if there is no current memory slot or attempted by zombie.
`test_mem,n`	Set the test flag to true if the value in the current memory slot equals n. Throws exception if there is no current memory slot or attempted by zombie.
`save_loc`	Saves the coordinates of the current x coordinate into mem slot 0, and current y coordinate into mem slot 1. Throws an exception if attempted by a zombie.

Right now, the only ops which have been implemented for you are `goto` and `rotate`. You'll have to implement everything else. Note that the exception for `goto` has not been implemented, though! So you will need to do that.

Hopefully at this point you understand roughly how the simulation works. Now let's look at the existing source files.

Part 1: ZOM Implementation

There are four main files related to the ZOM virtual machine framework: op.h/cpp, machine.h, and traits.h. I'm going to briefly describe what's in all the files. Read this description and understand it, don't try to implement anything until you get to "What to implement first."

Op.h/cpp defines and implements all the ops. There is an abstract base class called `Op` from which all the different operations derive from. All operations can have up to one parameter. More than one parameter is not supported. There are currently implementations for `OpGoto` and `OpRotate`. You will need to provide implementations for the rest of them. Notice how each `Op` has an `Execute` function, which takes a `MachineState` reference as a parameter. The `Execute` function is what performs the logic of the `Op`.

`MachineState`, in machine.h, represents the state of the current zombie/human. Right now it has several variables which are useful, but **it does not have all the variables you will need for this program**. You will need to add them on your own.

Here are the variables `MachineState` currently has:

`m_ProgramCounter` – Keeps track of the current line number of execution. This will change after every op.

`m_ActionsTaken` – Keeps track of the number of actions taken this turn.

`m_Facing` – Stores the current facing of the zombie/human.

`m_Test` – Boolean used for all the test_* ops.

`m_Memory` – Used for the C-style array of integers in memory. Note that this isn't deleted anywhere right now (memory leak!) so you should fix this.

There will be one `MachineState` in memory for every zombie/human in the world. However, notice how `MachineState` does not store the actual vector of ops. This is intentional, and is what the `Machine` class is for. The `Machine` class will parse in the .zom file, load in all the appropriate ops, and keep track of all that. But it does not store the state information for a particular zombie/human.

`Machine` has some temp code in it right now – the `LoadMachine` function will need to be implemented to parse in ZOM files properly. Right now it just hard codes a few instructions for testing.

Now, it's established that zombies and humans have slightly different characteristics. Rather than sub-classing `Machine`, we are using the Policy/Traits design pattern. These are defined in traits.h, and really are just a few constant values. The `Machine` will query those constant values at the appropriate times, and that will affect the behavior. You could, in theory, add a third type of character, like an alien or something, and have some combination between human/zombie traits.

The `Machine::BindState` function is important. Whenever a `MachineState` is created, it should be bound to a particular `Machine`. That's because it will set the important traits related to that `Machine`. There is no other way to get the traits from a `Machine` inside `MachineState`, so you must remember to bind when creating a human/zombie. Don't forget to change the binding when a human gets infected!

Anyway, let's look at what's working right now.

What's Working So Far

All you can do basically is run the program when you select Simulation>Run. What this does is sets a timer in MainFrm.cpp, so every second the turn timer ticks, and one turn is performed for the zombie. Notice how a console comes up with the app too. I did this so it would be easy to get debug output for the instructions.

If you run it right now, it's just using the hard-coded test instructions in `LoadMachine`. So you'll see it `rotate` and `goto`, and output something to that effect in

What to Implement First

Unlike the previous labs, I'm not going to provide as many step-by-step instructions. You should be ready for prime time now! But so you're not completely lost, here's one way you can approach the implementation.

You can approach this however you want to, but what I'd recommend doing is first implement a class to represent the entire world (using a singleton). You could use a couple of different data structures to track all the zombies/humans and where they are, but I'll leave the choice to you.

The first file you should try to support is basic_movement.zom. Implement the parsing of ZOM files in `Machine::LoadMachine`, and all the ops in basic_movement.zom. You will need some way to query what is in front of you for the `test_wall` function. You also should probably add x/y coordinate information to `MachineState`. Have your single test guy start out at (0,0). Then in the debug output, you can output the current position of the zombie/human as well, because it will really help debugging the execution.

After you think you've implemented the parsing and these first few ops properly, you should track the position of the sample zombie as he goes all the way to the right, before hitting the wall and going down (or in another direction, depending on the randomness). And more or less hugging the wall all the way. This should give you an idea if the ops are implemented properly.

Once you have the test zombie working in the debug console, you should now implement the visualization. Implement the drawing code in `CZombieView::DrawGrid`. This is all using GDI+. The grid is setup so the overall square is 600x600 pixels. So for a 20x20 grid, you want each square to be 30x30 pixels. Anyways. you can use lines for the grid and another shape for the zombie/humans. You can use filled shapes, too, if you want to fill them in. Make sure it's clear which way the zombie/human is facing! I used `FillPolygon` and made triangles, but you can use something else if you so desire.

After you get the single guy moving around in the window, expand to randomly populating the world with some more of the basic_movement.zom zombies on screen. Make sure you don't have any issue with trying to move to an invalid square.

Once you get the visualization of basic_movement.zom working properly, you can then move on to implementing everything other than the memory instructions. Once you think you're done with all of that, try placing a couple of humans and zombies and simulate them. See if it works. You could try using search.zom for the zombies and daryl_dixon.zom for the humans. Daryl is awesome so he will usually win this battle.

Finally, once you have the above working, try implementing the memory instructions. The memory part will be a little bit tricky, and remember that there is a set number of memory slots (either 0 or 2), so don't forget to check to make sure the slot request is valid. Once you think you've implemented these, try switching the human to the_governor.zom. He uses every memory operation other than set and `store_loc`. Basically he walks around and kills both humans and zombies, and keeps a running kill count of each in his memory.

To debug this memory usage, you should add functionality such that when you mouse over a square that contains a human, there should be extra debug information visible that shows what's currently in that human's memory. You need this because this is the only reasonable way to track the memory and make sure it's working properly.

To grab the mouse position when they move it, you need to bind to the `WM_MOUSEMOVE` message in `CZombieView`.

To draw text, you can use the `DrawString` function in GDI+. It will look something like this:

```
Font font(L"Times New Roman", 12);
SolidBrush brushBlack(Color(0,0,0));
m_GraphicsImage.DrawString(CA2W("My text"), -1, &font, PointF(620,270),
      &brushBlack);
```

The last program to test is michonne.zom, which uses the `store_loc` function (though does nothing with it).

Once you think you've implemented all the functionality for this properly, you can move on to the next part.

Part 2: Overall Simulation Additions

If you've gotten this far, quite honestly the hardest part of the lab is over. Now you just need to add a few more features.

First, change the "Start!" button so it instead is Start/Stop. Map it to `VK_SPACE` in the keyboard accelerators. You should make this pause or resume the simulation. The way you stop the timer from firing is by using `KillTimer(1)`. This only works on the ACTIVE WINDOW. So if you have a dialog box up for some reason and then try to `KillTimer`, it won't work properly.

Next, you should Implement Simulation>Clear, which should just nuke any zombies/humans currently in the world. It should also stop the timer from firing.

Simulation>Load Zombie... should bring up a "File Open" dialog box (like implemented in the last lab) that lets you pick the ZOM file for the zombies. Similarly, Load Survivor... should let you select the ZOM file for the humans.

Simulation>Randomize should randomly place 20 zombies and 10 humans in the world. This may seem like an unfair ratio, but it is true to the zombie apocalypse! Plus, the humans generally are stronger than the zombies. This ratio should make it a bit more fair. They should have not only a random position but a random orientation. This option should only be active if a zombie and human file has been loaded.

Finally, you should have some status text drawn on the right part of the screen which displays the following:

1. Number of zombies alive
2. Current zombie program
3. Number of humans alive
4. Current human program
5. Current "month" (how many turns have elapsed)

In the `OnTimer` function in MainFrm.cpp, if you see there are zero zombies or humans left, you should stop running the timer and declare the winner with a message box.

Anyways, if you've implemented everything in this part, you should have a simulation that looks something like the picture at the beginning of this lab.

Part 3: Experimentation

Now that your game simulation works, the lab is basically done! But there is one other thing to do, which is experiment a bit. I want you to write one new zombie program and one new human program. They don't have to be amazing, but they have to be unique enough from the existing ones, and not so simple that they are only like 5-10 lines. Be creative with the behavior.

For the human program, I want you to do something interesting with the memory, because the current ones really don't use it in an exciting way. See if you can come up with a way to get the memory to really help the humans.

If you can come up with a zombie program that is able to defeat Daryl in combat (eg. you spawn 10 Daryl and 20 of your zombie, and Daryl usually loses), **you will get extra credit**. So go out and make a killer zombie!

Lab 5: The Zompiler

In this lab, we will be writing a compiler for ZombieC, a high-level programming language for the zombie control programs we used in Lab 4. Writing the compiler will be roughly broken down into three phases:

1. Get the lexer/parser to correctly read in the tokens and match the grammar of ZombieC.
2. Create the Node classes for the AST.
3. Generate the zombie assembly code given an AST.

Before you can start any of this, though, you need to install cygwin on your system. Cygwin allows you to run flex/bison on a Windows system. Open the cygwin directory from your SVN (you should be grabbing 435 in this lab, as you did in lab 3/4, not your personal directory).

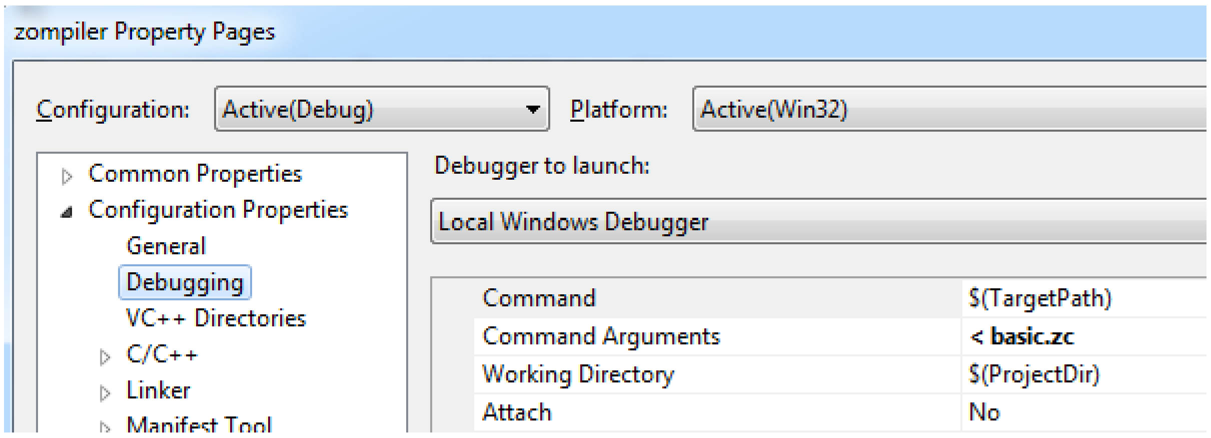
Inside the cygwin directory, double-click on install.bat. This will automatically install all the cygwin files you need to C:\cygwin. If you already have cygwin installed to a different directory, you will have to change directory paths in the project file. Let me know if this is an issue for you.

Once cygwin is properly installed, you will be able to open zompiler.sln, and build the project. If the project doesn't build for you, something isn't working properly.

If you try to run the program as given to you, nothing will happen and it will sit at the console. This is because it is expecting an input file to parse. To setup the input file, right-click on the zompiler project select Configuration Properties>Debugging, and type in the following for Command Arguments:

```
< basic.zc
```

This should look like this in the window:



zompiler Property Pages

Configuration: Active(Debug)   Platform: Active(Win32)

Debugger to launch: Local Windows Debugger

Command	$(TargetPath)
Command Arguments	< basic.zc
Working Directory	$(ProjectDir)
Attach	No

If you run the program now, you should get the following output on your console:

```
Numeric value of 0
Rotate command
Single statement
Main entry point found!
```

If you try any other file, such as two_statements.zc, you will get both lexer errors and syntax errors, because support for all the constructs has not been added yet.

For this lab, it will be very helpful to look at the Week 12, 13, and 14 notes. You also will want to refer to the calc examples on Blackboard.

Part 1: Scanner and Parser

The flex input file is zombie.l, and the bison input file is zombie.y. You will be editing these files exclusively during this part of the lab.

For this lab, I will not be providing you with the BNF for the grammar. That's because as seen in lecture, it's very easy to convert a grammar from BNF to Bison's syntax. That takes away much of the challenge of writing a parser/scanner. So, for the most part you will have to deduce the grammar given the .zc source files included in your project.

two_statements.zc

First, let's try to get two_statements.zc to parse properly. All the words you see in ZombieC files are keywords, and so they should be treated as tokens/terminal symbols. So that's why "rotate" is a token right now in zombie.l/y. To get two_statements.zc to parse properly, the first thing you must do is add a "forward" token. Remember, this means adding both a token declaration in zombie.y and the regular expression pattern in zombie.l. This will look very similar to what is already done for rotate.

Then once you have a token for the forward keyword, you can add an additional type of `statement` in zombie.y. So instead of a `statement` just being `rotate`, as it is right now, you can add `forward` as an option as well. Whenever you add a new grammar rule, I strongly encourage to add a match pattern that outputs some debug text that lets you know a rule was matched.

The next thing you need to do in zombie.l is add a pattern for single-line (C++-style) comments. A single-line comment is `"//"` followed by 0 or more characters of anything, ending with a `\n` to signify the end of the line. Because these comments will not be part of the AST, you don't actually need to declare a token to go with it. Just put the regular expression, and in the code to execute when the match occurs, increment `g_LineNum` (as done for the current `\n` pattern in zombie.l).

The last thing you need to do is add support for multi-statement blocks in zombie.y. A block can be one or more statements, though currently the grammar rule only allows one statement. This is going to be very similar to the left-recursive function "params" example I gave in lecture.

So once you have added support for the forward statement, single-line comments, and multi-statement blocks, you should be able to successfully parse two_statements.zc and have debug output that looks something like this:

```
Numeric value of 0
Rotate command
Single statement
Forward command
Multiple statements
Main entry point found!
```

stationary.zc

Now let's move on to stationary.zc, which is the simplest file that has an if statement. In ZombieC, there are no else if statements, and braces are required. This means it will be one of two potential syntaxes:

```
if (boolean)
{
    block
}
else
{
    block
}
```

OR

```
if (boolean)
{
   block
}
```

Those are the only variations of if/else that are possible in ZombieC. If/else is considered a type of statement, so it should be added as such. This will also require creating a boolean grammar rule, which allows code such as that in stationary.zc. This grammar rule should match the different type of boolean checks including is_zombie(numeric), is_human(numeric), etc. This also means you will need to add any tokens to satisfy the above.

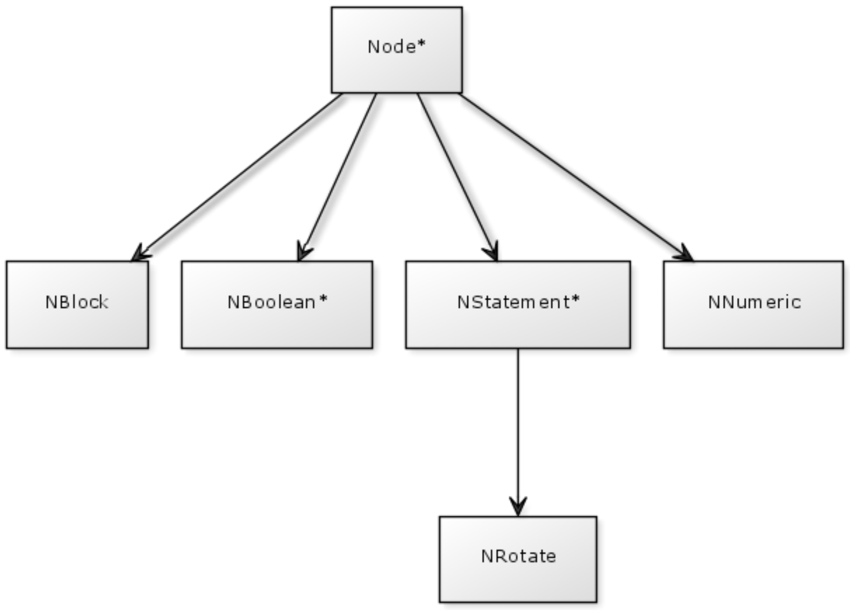
The Rest

Once you have stationary.zc getting parsed properly, you're well on your way to getting the parsing done! I would suggest the following order of getting the remaining files to successfully parse: search.zc, michonne.zc, the_governor.zc, and finally mem_test.zc.

Once all these files parse without syntax error, congrats! You've finished probably the most time-consuming part of the lab.

Part 2: The AST

Now that the ZombieC files are parsing properly, we can generate the Abstract Syntax Tree. If you open up node.h, here are the current classes which are declared. The (*) signifies an abstract class:



You should not need to add any additional abstract classes to successfully build an AST. **All of the other classes you will make for the AST will derive from either `NBoolean` or** `NStatement`. Depending on the node, it may need to take a pointer to another node type in its constructor. Or, if there are no additional parameters to apply to the node, then it may just use the default constructor.

The other thing to note is which types are used in the %union struct in zombie.y. While generally, only abstract classes should be included, I've also intentionally included NBlock* and NNumeric* as options, since they are types which we want strong enforcement for generating the AST. **You should not need to add any more types to the %union**.

Once you have the appropriate node class, you can construct it in the grammar action. For instance, the action for the rotate grammar rule would be:

```
$$ = new NRotate($3);
```

Remember that because CodeGen is a pure virtual function in the abstract classes, any derived class must implement the function before you can create an instance of it. So if you get an error message that a new class, such as NForward, is pure virtual, that means you need to add an empty implementation of CodeGen for now.

The majority of grammar actions will only require setting $$ to a new instance of the node, as with NRotate. But there are a couple of special cases to watch out for.

The main_loop action simply needs to grab the block it has and set the global g_MainBlock pointer to it. This global block should then also have SetMainBlock called on it.

The other special case is the actions for block. When the first statement is matched, you want to construct a new NBlock, and add the statement to this new NBlock's list of statements. But when subsequent statements are matched, rather than creating a new NBlock, you should simply add the new statement to the already existing NBlock.

As with the last part, you should slowly add support for more and more grammar rules and make sure the program runs and builds the AST without crashing. You should follow the same order followed in the last part: basic, two_statements, search, michonne, the_governor, and finally mem_test.

Unfortunately, you can't fully test whether or not the AST built properly without adding the code gen code in part 3. But if the AST builds without crashing, that's a step in the right direction.

By the way, if you are wondering if all these creation of new Node classes leaks memory, you're right, it does. In an actual system we would want to make sure we clean up all the memory. But don't worry about it for this lab, because it's just annoying to have to traverse the AST and delete everything appropriately. And in any event, once the executable exits all the memory will get cleared up for you.

Part 3: Code Generation

The last part to getting the compiler working is code generation. To help test this part, I have included a .out.zom file for every .zc file included in the project. So for example, michonne.out.zom should be what your assembly output looks like if you are correctly generating the code. Make sure you check this for each file as you go along.

In zompiler.cpp, once `zompilerparse` is called in the `main` function, there should be a valid AST if there wasn't a compile error. This is the point where we want to then generate the list of ZOM assembly commands to output.

The way the commands are stored is in a `struct` called `CodeContext`. This `struct` has a vector of strings. Each time a new command is generated, it should be added to the vector with `push_back`. The `CodeGen` function for each node takes this `CodeContext` by reference. So you make a local one in main, and then pass it to the root node. Once the AST is traversed, you'll then have your vector with the list of all the assembly commands.

You only want to code gen in the case that the lexical and syntax analysis was successful. The idea is that `g_MainBlock` will be `nullptr` if compilation failed, so first check to make sure that isn't the case. If it isn't, you can then call `CodeGen` on the main block as such:

```
g_MainBlock->CodeGen(myContext);
```

Right now, this won't do anything because `NBlock::CodeGen` is not implemented. But this is how we will traverse the AST to generate all of the code.

Code Gen of basic.zc

So the first `CodeGen` function you should implement is naturally `NBlock`'s. What it needs to do is loop through all the statements in its member list, and call `CodeGen` on each and every statement. Once the loop is done, you should check if the current block is the main block. If it is, this means that you need to add one final "goto,1" command to complete the main loop.

It turns out that `NRotate::CodeGen` is already written for you. So you don't have to worry about that for basic.zc.

The last thing you need to do is back in the main function. Once the `CodeGen` returns from the main block, you should have a `CodeContext` which has been populated with all of the assembly commands. Each command should be output to an individual line in the output file "out.zom."

If this is implemented correctly, your out.zom should be identical to the basic.out.zom file included in your zompiler directory.

Code Gen of two_statements.zc

The code generation for the second file should only additionally require you to implement `NForward::CodeGen`. This should be easy enough to get working, and you can verify that fact in two_statements.out.zom.

Code Gen of stationary.zc

To do stationarcy.zc, you need to implement `CodeGen` for the most challenging statement in ZombieC: the if/else statement. With the if/else, you want the output code to be something like this:

1. Test flag
2. je to the if block
3. else block code
4. goto past the if block
5. if block code

The problem is that when we are generating code, we simply add the instruction to our vector. But when it's time to output the je, there's no way of knowing how many lines of code the else statement is going to be. Similarly, when it's time to output the `goto`, there's no way of knowing how long the if case code will be.

(Technically, you could also do a jne to the else block, and have the if block code first. This is more like what would be done on an actual compiler, but it works the same as the above).

The way you can solve this is by keeping track of where the instruction for steps #2 and steps #4 are in the vector. Then, once the code for the if and else blocks has been generated, you can go back and fix up their jump locations to point to the correct line numbers.

If this is done properly, your output should match up with stationary.out.zom.

The Other Files

Once you have the if/else working properly, you can go through the other files in the same order as before (search, michonne, the_governor, and mem_test). If their outputs all line up, you're set mostly.

There is one aspect of `CodeGen` that does not currently have a test case, though: an if *without* the else case. This is simpler than the if/else case, though. Just do the test, and jne past the if block.

And a Little Bit of Optimization...

So there is one little problem with the way the code is being generated, specifically with regards to the gotos. Compare michonne.out.zom to michonne.optimized.zom. Notice the difference?

In michonne.out.zom, there are gotos chained together. For instance, line 8 goes to line 10, which goes to line 12, which goes to line 15, which goes to line 1. But each of the statements in the chain should have just been optimized so they immediately jump to line 1.

The problem is that when the `goto` statements are initially added to the m_Ops vector, there is no way of knowing the `goto` is part of a chain that will lead back to the first line. So this is something that needs to be optimized *after* the whole program has been generated.

An easy way to keep track of the gotos is by adding a `std::map<int,int>` of gotos to `CodeContext`. The key is the line number the `goto` statement is on, and the value is the line number the `goto` statement jumps to. Then every time a `goto` is ouput, you should add an entry to the map.

Then, when the `main_block` has been generated, you jump through the map locating any chains, replacing the `goto` chains with a `goto` directly to the line number at the end of the chain. So for example, instead of:

```
8. goto 10 -> 10. goto 12 -> 12. goto 15 -> 15. goto 1
```

The chain can be simplified to:

```
8. goto 1
```

Which is way more efficient!

So it turns out once you put in this goto optimization, the ZombieC compiler will output code which is identical to the hand-written zombie assembly I gave you in lab 4.

In any event, congratulations! You have now made a compiler.

THE END