Comments on Homework 4

Design

1. You *must* check for the success or failure of dynamic memory allocation. Until we learn to work with exceptions, use either of the following.

```
#include <cassert>

type* aPtr = NULL;
assert( aPtr = new(nothrow) amount);
```

or

```
if ( ( aPtr = new(nothrow) amount))
{
    continue
}
exit;
```

   Using nothrow ensures that a failed allocation will return NULL. Omitting nothrow permits a failed allocation to throw an exception. If an exception is not handled, the flow of control will call the terminate function and the program will end immediately. This is not good. For that matter, neither is the assert, however, at least you are in control of that.

2. When are using dynamic allocation to increase the size of a container, perform the allocation first. Assign the pointer to the newly allocated memory to a temporary pointer. If that allocation succeeds, then and only then, delete the memory pointed to by the existing pointer.
   If the allocation fails, and you have deleted the existing memory, you cannot recover.

3. Until you learn exceptions, do not dynamically allocate memory in a constructor. You have no way to tell the outside world if that allocation fails and you will return a partially constructed object. We'll learn about a better way to do this shortly.

4. Always ensure that any containers that you use are large enough to hold what you are putting in them. Java and C# add the extra overhead to catch this for you. C and C++ are leaner (smaller memory footprint, faster, etc.) languages and so, it is up to the designer to manage this.
   The C style string function strlen() *does not* count the terminating NULL on the end of such a string. Thus, such a string must be put into a container of length strlen() + 1. To hold the terminating NULL. If you do not do this, like the first debugging problem we had, your program will most likely compile and run…however, you are using memory that you do not own and may fail in unpredictable ways as you alter the program or add more code.

5. When your program prompts the user for information…bring it in. If there is an error in what was entered, under no circumstances try to guess, then correct, what has been entered. Flag this and return an error message. If you try to correct, you may 'correct' to a value that can create a dangerous and/or potentially unsafe situation.

6. A declaration brings the name of a variable into namespace…it makes the name visible to the translation unit(s) that wish to use it.

   For example,

```
extern int myInt;      // is a declaration and brings the name myInt into the
                       // translation unit where it might be used

struct MyStruct        // is also a declaration
{
    Struct body;
};
```

   A variable can be *declared* in a C or C++ program multiple times.

   A definition allocates memory (and potentially initializes) to hold a variable of the specified type.
   A variable can only be *defined* in a C or C++ program one time.

   The following are definitions.

```
MyStruct aStruct;      // is a definition of an instance of the struct MyStruct

int myInt = 0;         // is a definition and initialization. If it is the first time that the
                       // name appeared, it is also a declaration.
```

   myInt is being *initialized* to the value 0. A variable can only be *initialized* one time.

```
myInt = 8;             // is an assignment
```

   myInt is being *assigned* the value 0. A variable can be assigned to many times.

   Never include a *definition*, code that allocates memory, in a header file.

7. Change your if statements
   From:

```
if(aVar == aVal)
```

   To:

```
if(aVal==aVar)
```

   this way the compiler will catch it if you drop an =

The value aVal is an *rvalue*. That is, it can only appear on the right hand side of an assignment statement.

The variable aVar is an *lvalue*. That is, it can appear on the left or right hand side of an assignment statement.

In the first form of the if clause, if you drop one of the'=', evaluating the clause will assign aVal to aVar each time the clause is evaluated. Further, the if will never evaluate to not true.

8. Use a case or switch statement rather than a if-else clause if you are making decisions on different values of the same variable. This will yield a smaller and faster program.

Program Structure

1. Align your comments and code.

2. Separate major functional blocks and minor segments with white space. Whitespace adds nothing to the final size of the compiled program and proper inclusion makes the program much easier to read and follow and ultimately can lead to fewer mistakes if it ever needs to be modified.

3. Use indentation to visually structure the layout of your program.

4. Limit the number of characters on a single line to 80 or fewer. Never wrap or truncate lines. Exceeding this number or wrapping lines make your code very difficult to read and follow…particularly if it is printed.

5. Use the preprocessor to control how a program is built and to conditionally include code.

   For example,

```
#define DEBUG0
#define DEBUG1

●●●●
#ifdef DEBUG1              //  if DEBUG1 is defined (it is), include the following
                           //  code in the program
    Conditionally included code Code block 0
#endif

Always included code

#ifdef DEBUG0              //  if DEBUG0 is defined (it is), include the following
                           //  code in the program
    Conditionally included code Code block 1
#endif
```

```
Always included code

#ifdef DEBUG1
    Conditionally included code Code block 2
#endif

etc.
```

You can use as many such blocks as you wish and can also use as many different keys as you wish to specify which code blocks are included.

To not include a code block you can either comment out the definition of the key or follow the definition with the line

```
#undef DEBUG0
```

This line can go anywhere in your program. Following the line, DEBUG0 is no longer defined and the conditional code will not be included in the final build of the program.

6. Now that we have moved beyond the basics, your programs should be build as a multiple file project. These files decompose into the following:

MainHeader.h
Mainfile.cpp
ImplementationFile0Header0.h
ImplementationFile0.cpp
●●●●
ImplementationFileN-1HeaderN-1.h
ImplementationFileN-1.cpp

Do not #include any of your .cpp files into another file….these should always be linked in.

For each header file, put it together as follows, for example…

```
#ifndef  HEADER0
#define HEADER0

    Header declarations etc.

#endif
```

When a C++ program is compiled, each file is read by the preprocessor and is used to build a temporary file called a *translation unit.* The translation unit is what the compiler ultimately compiles to produce an object file. To start, the preprocessor locates each header file and places a copy into the translation unit, replacing the *#include* directive for that header file. The preprocessor then processes any other directives such as *#ifndef, #define*, etc. After the translation unit is compiled to create the object file, the translation unit is deleted – it's no longer necessary. Each translation unit is self-contained. It cannot use variables or functions that are part of another translation unit. Constructing the header as shown above ensures that, at the end of the day, we only have a single declaration for each variable or function name.

Putting the definition of a variable or function in multiple header files will cause multiple pieces of memory to be set aside for the same variable or function body. The system linker, which is building the executable for the program, does not like this.

7. Use a multiple file architecture to support your tests and test code. Use a structure such as the following:

```
#includes
Global declarations and/or definitions
int main(void)
{
    Local declarations and/or definitions
    Local code
    #define TEST

    #ifdef TEST
        testFunctions(<args>)

    #endif

    Other local code

    return aValue;

}

testFunctions.cpp                    //  this file should be linked in as part of your project.
```

8. When you design your program, once again, keep things simple. Don't make something unnecessarily complex. Learn and use the library functions that are available rather than writing your own unless your version offers some benefit.

9. When you turn in your design, saving the results as a .png file looses a lot of information during the compression. The resulting files are very difficult to read.