Project 0: hexdump

Assigned:	1/17
Final Version Due:	1/27 by 11:59 pm

Note: there is only one due date for Project 0.

GitHub Invitation Link (https://classroom.github.com/assignment-invitations/146591f20af0cbfadfafebd9363da62b)

The goal of this project is to get used to the course workflow and be sure you understand how to submit code for grading. You will write a simple command line application called hexd that will read a file and write its contents in a special format as described below.

This assignment requires only basic computer organization knowledge and very little C++ experience beyond basic functions and IO operations. If you struggle with this assignment you are woefully ill-prepared for this course. This means you should devote additional time to it and seek advice from the TAs and myself, or as a last resort drop.

Description

Recall that all files are just sequences of bytes. To view a file you have to assume some form of format to the file. For example a plain-text, ASCII encoded file is one where the bytes are interpreted as representing ASCII characters, while an image file is one where the bytes represent the color and arrangement of pixels.

A hex viewer is a program that shows the raw byte sequence of a file formatted as hexadecimal (base 16) numbers. Each byte of the file is represented a a two-digit hexadecimal number. The sequence is organized into rows representing 16 bytes at a time. The first column is a 7-digit zero-based index, written in hex, of the first byte on that line, followed by a `:` character. The next column is the next two bytes grouped together written in hex. These columns continue seven more times until a total of 16 bytes (8 columns of two bytes each) are shown. The last column shows the ASCII representation of the byte sequence for that row, printing a `.` character for non-printable bytes (those without an ASCII character representation). It is separated by the previous column by two spaces.

Since our program will view the file by simply printing it in the above format to standard output, i.e. dumping it to the screen, we will call our program `hexd`, short for "hex dump".

Lets look at an example. Suppose you write the following program:

```
#include <fstream>

int main(){

  std::ofstream outs("temp.txt");
  outs << "This is a line of text.\nThis is another.\n";
  outs.close();

  return 0;
}
```

When your run it (assuming it has write permissions), it would produce a plain-text, ASCII file `temp.txt` with the contents (as it would look in an editor):

```
This is a line of text.
This is another.
```

On a UNIX system, the hex dump of that file would be:

```
0000000: 5468 6973 2069 7320 6120 6c69 6e65 206f  This is a line o
0000010: 6620 7465 7874 2e0a 5468 6973 2069 7320  f text..This is
0000020: 616e 6f74 6865 722e 0a                   another..
```

From this we can see the first byte of the file is hex 54 (binary 01010100, decimal 84, ASCII letter T). The byte at index 16 (7 digit hex 0000010) is hex 66 (binary 01100110, decimal 102, ASCII letter f). The byte at index 23 is hex 0a (binary 00001010, decimal 10, ASCII Line Feed). Etc. Notice that if a row does not have 16 entries, the rest are padded with spaces. Study the dump to be sure you understand the format.

On a Windows system that same file (by all appearances in the editor) would have the hex dump

```
0000000: 5468 6973 2069 7320 6120 6c69 6e65 206f  This is a line o
0000010: 6620 7465 7874 2e0d 0a54 6869 7320 6973  f text...This is
0000020: 2061 6e6f 7468 6572 2e0d 0a               another...
```

Notice that the byte at index 23 is now hex 0d followed by hex 0a. That is because text files on windows use two characters (ASCII Carriage Return and Line Feed) to represent the end of a line.

These are the kind of subtle differences that can be discovered using a hex dump and why it is an invaluable tool when debugging low-level issues and reverse engineering. Of course there are already tools you can get that do this, but now you will have written your own!

Some hints. Recall that to read a file byte-by-byte in C++ it should be opened in binary mode. Reading and writing from/to a binary file is different than when using text files. First, you have to open the file in binary mode, e.g.

```
std::ifstream instream("input_binary.file", std::ios::binary);
```

To read a binary value you use the read method, which takes a pointer to a memory location as a `std::fstream::char_type *` (a pointer to the type fstreams use to hold characters) and the number of bytes to read. The pointer must be cast using a reinterpret cast. For example to read an unsigned 32 bit integer one would do

```
uint32_t value;
instream.read(reinterpret_cast<std::fstream::char_type*>(&value), sizeof value);
```

The fixed-width integer types (int8_t, uint8_t, int16_t, uint16_t, int32_t, uint32_t, etc.) are defined in the header cstdint (http://en.cppreference.com/w/cpp/types/integer). You should always use them instead of the types `int`, `long`, etc. when you need to depend on the memory size of the type.

Setup

Accept the GitHub invitation above and then clone the git repository to your local machine. Implement your program in a source file named `hexd.cpp`. You should use git to commit versions of your program source (only) as you go along, demonstrating good incremental programming technique.

Correctness

The file to read is specified as command line argument (whatever the user types on the command line after the executable). If you need a primer on how to use command line arguments, see below. The file should be printed in the hex format above (exactly) to standard output. So for example, assume that a file named `text` with the contents above was in the current directory along with the executable on a Windows system. Then this would be how the program would be run to dump the file to hex (assume > is the prompt):

```
> .\hexd.exe text
```

which would print out to the terminal/console:

```
0000000: 5468 6973 2069 7320 6120 6c69 6e65 206f  This is a line o
0000010: 6620 7465 7874 2e0d 0a54 6869 7320 6973  f text...This is
0000020: 2061 6e6f 7468 6572 2e0d 0a               another...
```

If no file is specified, or the specified file cannot be opened, the program should print an appropriate error message to standard error and return `EXIT_FAILURE`. Otherwise it should return `EXIT_SUCCESS`.

Testing

The initial git repository has a sub-directory called test which has several examples of input files (ending in .bin) and corresponding expected output (ending in .hex). You can test your code in the reference grading environment by starting a VM (with the supplied Vagrantfile as in meeting 3), compiling the program and ensuring it writes the same output as in the .hex files when run on the .bin inputs. When invoking the compiler be sure to use the flags "-std=c++11 -Wall -Wextra`` to enable better static analysis. Here is an example transcript of a correct invocation:

```
ubuntu@ubuntu-xenial:~$ g++ -std=c++11 -Wall -Wextra /vagrant/hexd.cpp
ubuntu@ubuntu-xenial:~$
```

Submission

To submit your assignment:

1. Tag the git commit that you wish to be considered for grading as "final".

   ```
   git tag final
   ```

2. Push this change to GitHub

   ```
   git push origin final
   ```

Be sure you have committed all the changes you intend to. **Failure to complete these steps by the due date will result in no credit being assigned.**

Grading

There are 30 points allocated to this assignment.

Correctness	24
Code Quality	6

Correctness means that the program reads the filename from the command line and write the output per the specification. Code quality will be assessed in this assignment by ensuring your code compiles cleanly, with no warnings, using the g++ flag `-Wall -Wextra` in the reference environment, as specified above. That is in the VM typing `g++ -std=c++11 -Wall -Wextra /vagrant/hexd.cpp` should produce no errors **or warnings**. You should also have made more than 2 commits to your repository before submission.

Grading Script: Since P0 does not have two due dates we are going to give you the grading script we will use to evaluate your submission. To use it:

1. Copy the file `grade.py` from this zip file (grade.zip) to your repository (I have to distribute it as a zip to get around a server issue).
2. In your Vagrantfile, add the string "cmake" at the end of line 4 (starts with apt-get)
3. Start the VM and login.
4. Change to your source directory (`cd /vagrant`)
5. Type `python3 grade.py`

You should now have the files `build.log` and `feedback.log` added to your source directory. Do not commit these to your repository.

A Primer on Command Line Arguments

Recall the primary ways to get user input into a program, standard input (`std::cin`) and reading from files. Another very convenient one is to use command line arguments. When your program is run, usually from another program (the operating system or a shell program), it starts executing in the function `main` . This is called the *entry point*.

```
int main()
{
    //code here

    return 0;
}
```

There is another form of this entry point with two arguments. The first (traditionally named `argc`) is the number of string arguments to the program, the second (traditionally named `argv`) is an array of ``argc'' C-style strings (pointers to null-terminated memory) with the arguments themselves.

```
int main(int argc, char*argv[])
{
    //code here

    return 0;
}
```

You can specify the command line arguments when you run the program from a text shell (e.g. powershell or bash), from a graphical shell (like when you click on an icon), or from a script.

It is easy to convert the more C-style arguments to a more modern C++ style as follows

```
#include <vector>
#include <string>

int main(int argc, char*argv[])
{
    std::vector<std::string>  arguments;
    for(int i = 0; i < argc; ++i) arguments.push_back(argv[i]);

    // code can use arguments as a C++ vector of C++ strings

    return 0;
}
```

The most common use of command line arguments is to be able to run a program easily with different inputs, often filenames to use. A common followup question is why is this better than just prompting for input with standard output and standard input (''std::cout'' followed by ''std::cin'')?

Than answer is that such input is difficult to automate. Scripts are one of the best ways to combine programs and often need to run unattended. A prompt blocks the program, waiting on standard input, usually requiring a user to be present and attending to the task. Although standard input can be "piped" into a program, it is much easier to use command line arguments. A script can also use the output of one program as the command line argument to another, allowing one to mix and match smaller programs into a single task. This is one of the core philosophies in computing.

A related question to the previous is why does main return an int anyway? Well, when your program is run to completion it can indicate to the running process if it succeeded or if an error occurred by returning this int. Success is defined as zero, which is why most simple examples returns that. The following are defined in the header ''cstdlib'': ''EXIT_SUCCESS'' and ''EXIT_FAILURE'', and can be used when returning from main.

Note you do not have to call return in main (this is a special case). In such cases the compiler inserts a success return for you.
