COS30031 - Lab 10

102564760 – Ryan Chessum

Part A

- Created a struct with data.

```
struct Test
{
    int anInt;
    char aChar;
    float aFloat;

    void ShowValues()
    {
        cout << anInt << endl;
        cout << aChar << endl;
        cout << aFloat << endl;
    };
};
```

- Created an instance of the struct.

```
int main()
{
    int in = 6;
    char ch = 'a';
    float fl = 2.2;
    Test myData{ in, ch, fl };

    myData.ShowValues();
```

- Wrote code to open a binary file and write to it.

```
    //open in write mode
    fstream fs;
    fs.open("test1.bin", fstream::out | fstream::binary);

    //write data
    fs << myData.anInt;
    fs << myData.aChar;
    fs << myData.aFloat;

    //close file
    fs.close();

    return 0;
}
```

Q: There are different file open modes: What are they?

member constant	stands for	access
in	**in**put	File open for reading: the *internal stream buffer* supports input operations.
out	**out**put	File open for writing: the *internal stream buffer* supports output operations.
binary	**binary**	Operations are performed in binary mode rather than text.
ate	**at e**nd	The *output position* starts at the end of the file.
app	**app**end	All output operations happen at the end of the file, appending to its existing contents.
trunc	**trunc**ate	Any contents that existed in the file before it is open are discarded.

These flags can be combined with the bitwise OR operator (|).

(table from: https://www.cplusplus.com/reference/fstream/fstream/open/)

in -> open file for reading

out -> open file for writing

binary -> performs operations in binary

ate -> output position starts at the end of the file

app -> operations happen at the end of the file

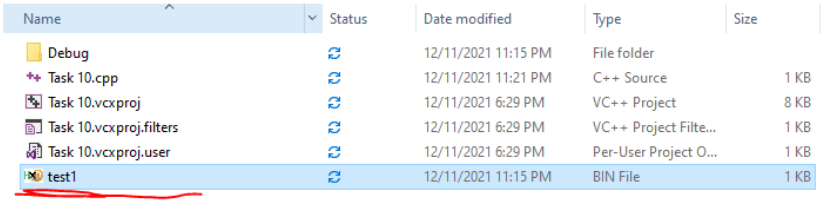
trunc -> discards the content within the file before performing operations

(The difference between ate and app are that ate sets the write position to the end upon opening the file while app sets it to the end before each operation.)

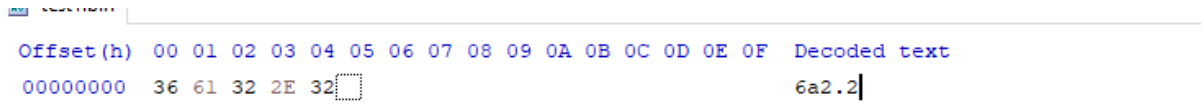
Q: What happens if you don't "close" the file? Is it something we need to worry about?

There is a very, very small chance it could corrupt the file but otherwise the file stream will go out of scope and the file will be closed automatically. So if we forget it we probably don't need to worry too much. Despite it normally being fine, it is best to remember to close it. If you are writing to too many open files at once your program can crash or fail to open more files when you need to.

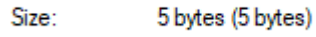
- Code runs and file was created



- Stuff is being written to the file. The hex viewing program I am using even confirms it's what I put in.



Q: How many bytes are in the file? Is this expected based on the size of the variable types?

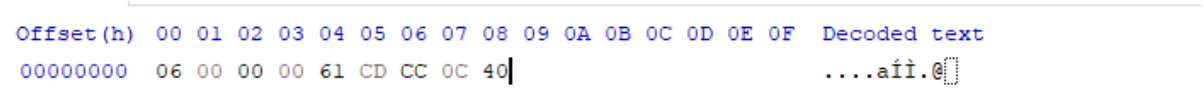


It's 5 bytes. It seems that it saved each value as a char the way I saved to the file initially so I has 1 byte for each character saved in the file.

To specify the type write the code like this.

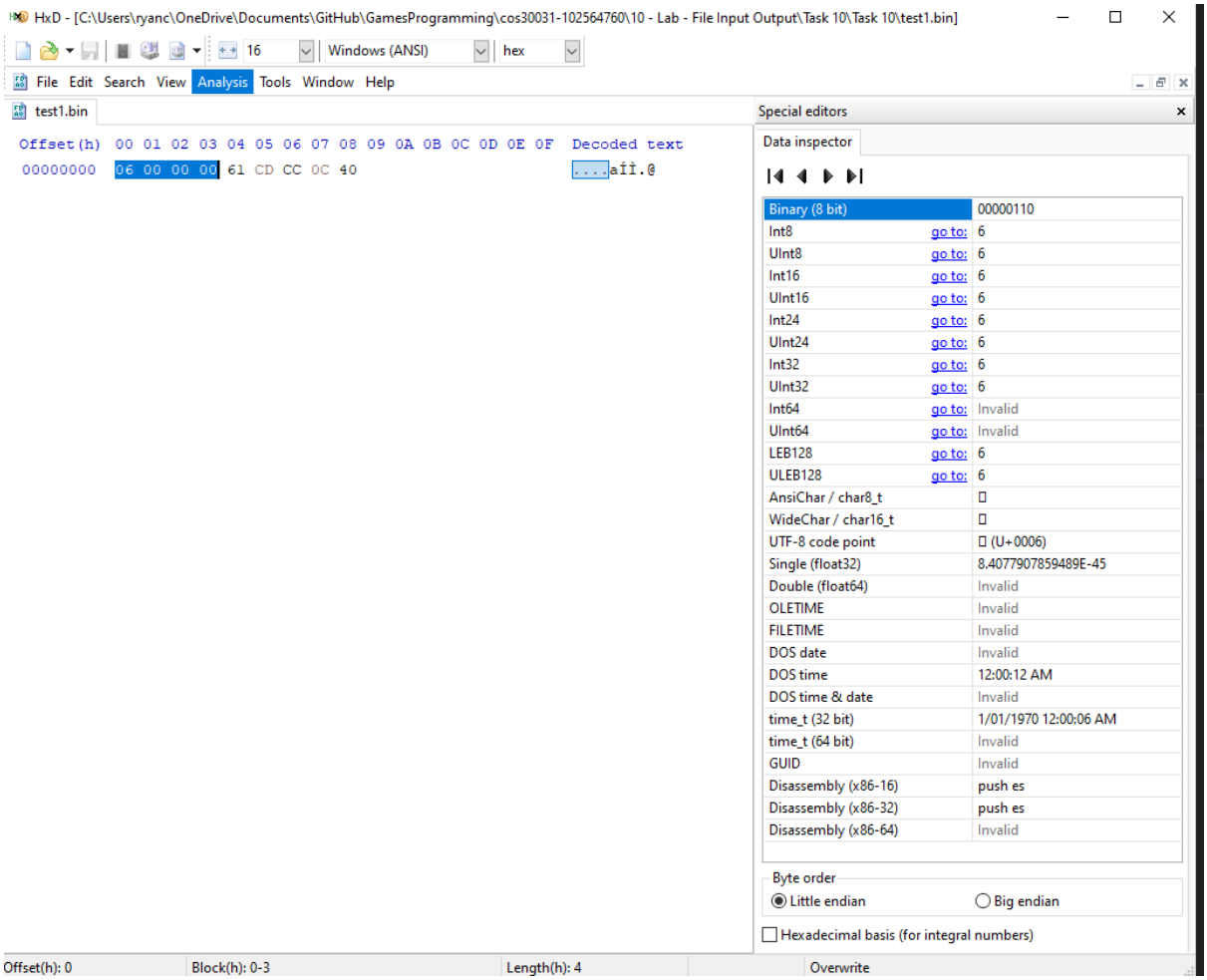
```
//open in write mode
fstream fs;
fs.open("test1.bin", fstream::out | fstream::binary | fstream::trunc);

//write data
fs.write(reinterpret_cast<const char*>(&myData.anInt), sizeof(myData.anInt));
fs.write(reinterpret_cast<const char*>(&myData.aChar), sizeof(myData.aChar));
fs.write(reinterpret_cast<const char*>(&myData.aFloat), sizeof(myData.aFloat));

//close file
fs.close();
```

Opening the file now gives us this.

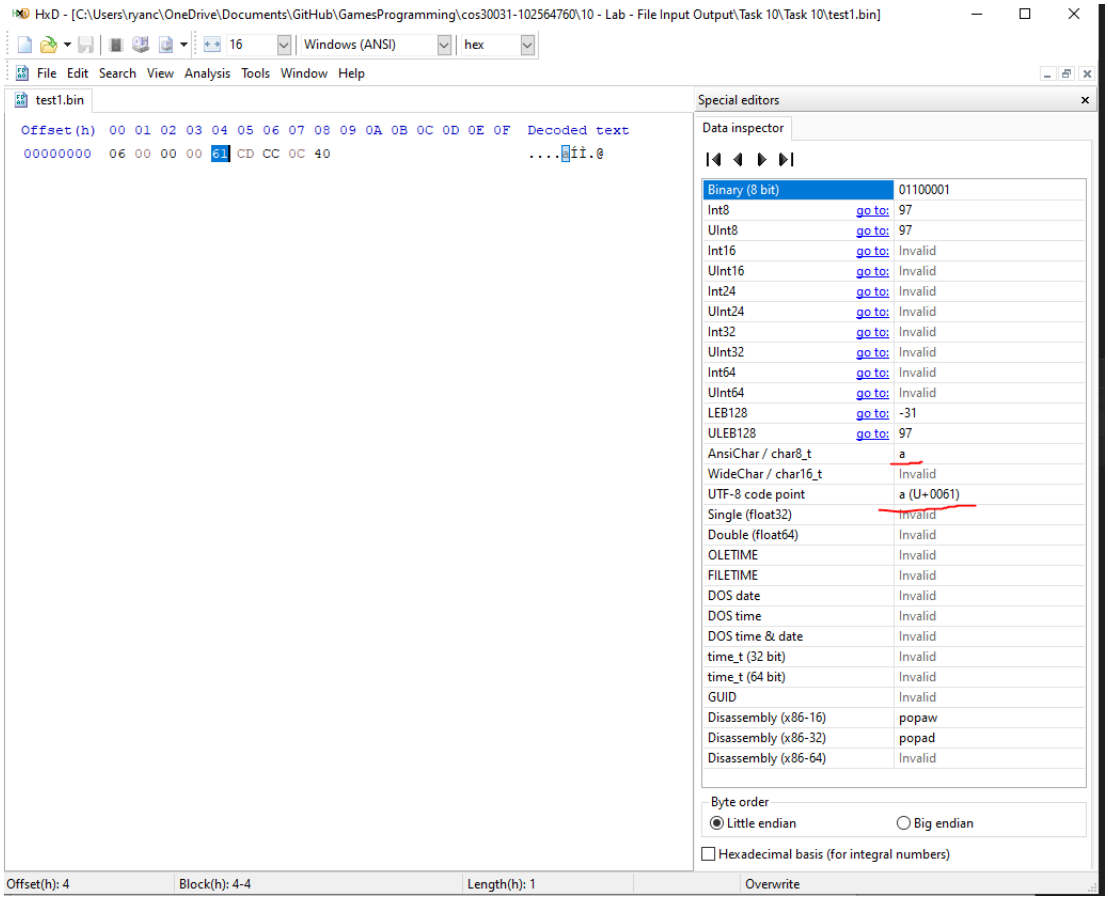


The hex viewer can't really tell us what the text is right away. It got 'a' right though.

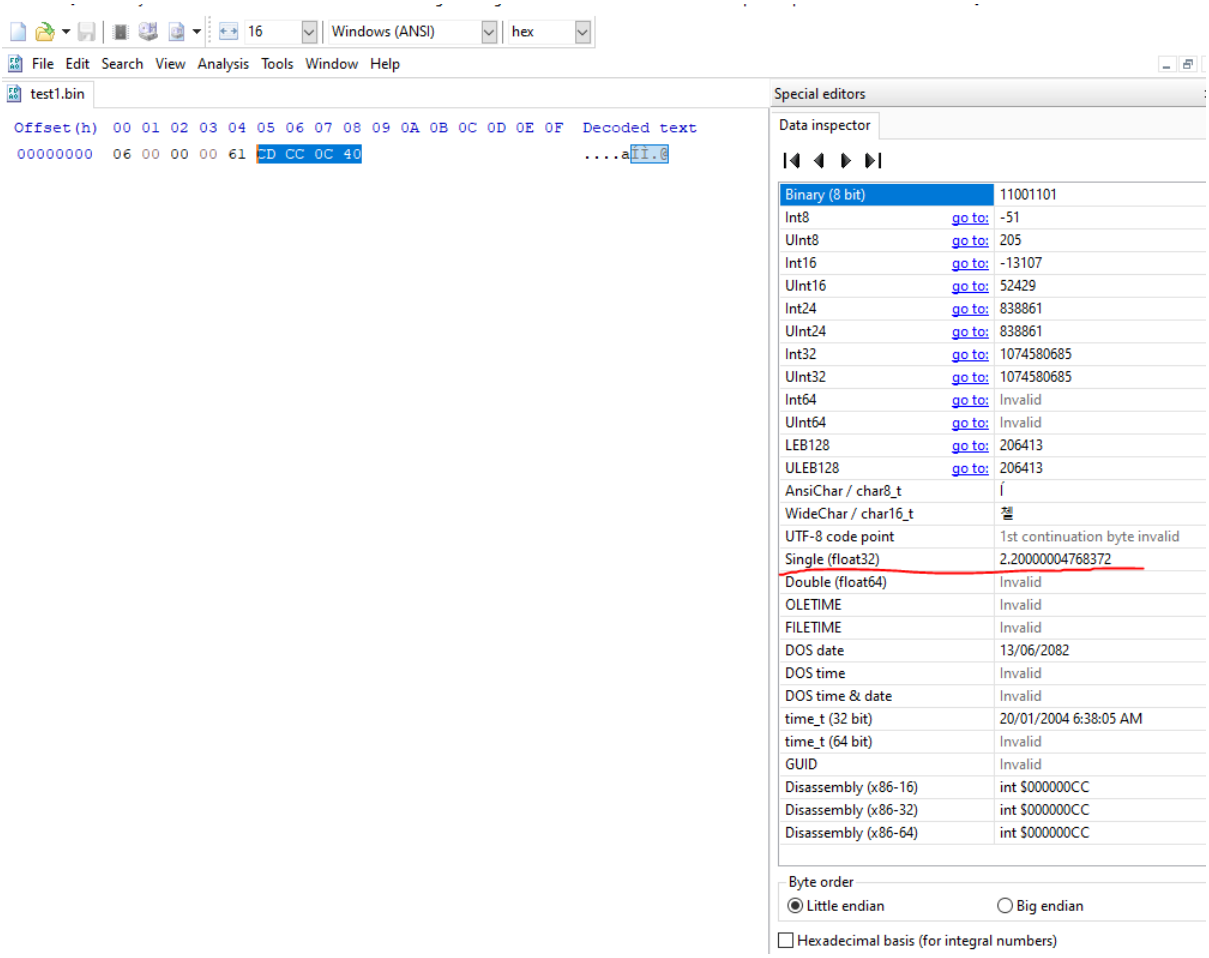
If we check each chunk of data we can see it would be as the appropriate data type.



The int we used must have been a 32 bit integer as it takes up 4 bytes.

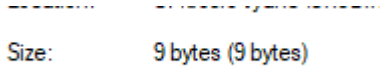


A is still a char so it only takes up 1 byte.



Our float is also now stored as a float. Pretty interesting how you can see floats don't always have 100% precision.

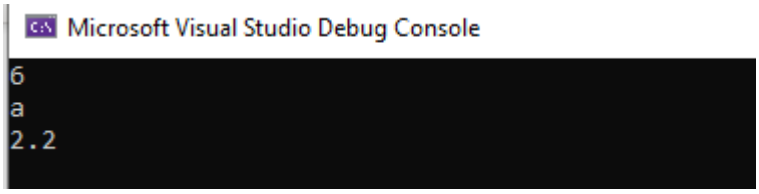
So if we check the size again…



It's now 9 bytes. Still makes sense as floats and larger integers take a bit more space. Still tiny though and makes sense for our small data types.

- Wrote code to read the data stored in the file

```
int in;
char ch;
float fl;

fstream fs;
fs.open("test1.bin", fstream::in | fstream::binary);

fs.read(reinterpret_cast<char*>(&in), 4);
fs.read(reinterpret_cast<char*>(&ch), 1);
fs.read(reinterpret_cast<char*>(&fl), 4);

fs.close();

Test myData{ in, ch, fl };

myData.ShowValues();
```

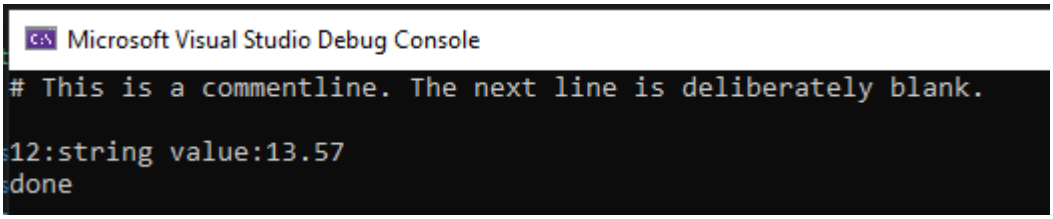


It reads the file successfully!

Part B

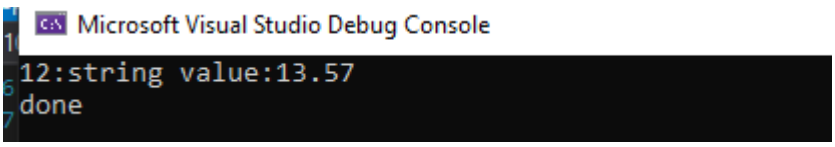
- Wrote code to read each line of the .txt file

```
fstream fs;
string str;

fs.open("test2.txt", fstream::in);

while (getline(fs, str))
{
    cout << str << endl;
}

fs.close();
```



- Edited code to ignore lines if they are blank or commented with a '#'

```
fstream fs;
string str;

fs.open("test2.txt", fstream::in);

while (getline(fs, str))
{
    if (str.size() != 0 && str.at(0) != '#')
    {
        cout << str << endl;
    }
}

fs.close();
```



- Split the string by ':'

```
vector<string> split(string str)
{
    stringstream ss(str);
    vector<string> result;

    while (ss.good())
    {
        string s;
        getline(ss, s, ':');
        result.push_back(s);
    }
    return result;
}
```

```
fstream fs;
string str;

fs.open("test2.txt", fstream::in);

while (getline(fs, str))
{
    if ((str.size() != 0) && (str.at(0) != '#'))
    {
        for (auto s : split(str))
        {
            cout << s << endl;
        }
    }
}

fs.close();
```

Microsoft Visual Studio Debug Console

```
12
string value
13.57
done
```

Part C

- Created Json file
- Downloaded Json library
- Wrote code to read the file using library

```cpp
fstream fs;

fs.open("test3.json", fstream::in);
json jf = json::parse(fs);


cout << "exp: " << jf["exp"] << endl;
cout << "health: " << jf["health"] << endl;
cout << "jsonType: " << jf["jsonType"] << endl;
cout << "level: " << jf["level"] << endl;
cout << "name: " << jf["name"] << endl;
cout << "uuid: " << jf["uuid"] << endl;

fs.close();
```

Microsoft Visual Studio Debug Console

```
exp: 12345
health: 100
jsonType: "player"
level: 42
name: "Fred"
uuid: "123456"

done
```