CCNU-UOW
CSCI851 Advanced Programming
Autumn 2020

Prof. Zhifeng Wang

Laboratory Exercise 4 (Week 4)

1 Warm–up exercises

1. File handling exercise: `Warmup-file.cpp`. This isn't debugging and the program should compile using CC on Banshee without problems.
   (a) How are the input elements separated?
   (b) Enter a few valid lines of data, end the program, and look at the content of the data file. You can use `cat` to view the file.
   (c) Run the program again, entering some more data before terminating the program. What has happened to the data file?
   (d) Run the following:
   ```
   $ mv Students.txt Data.txt               This is renaming the file.
   $ ./a.out < Data.txt                     This is directing input from a file.
   ```
   (e) Modify `Warmup-file.cpp` to open the file in append mode instead of the default write mode. Careful with the scope.
   (f) Re-run the program. What happens to the data file now?

2. Vector handling : `Warmup-vector.cpp` is similar to the `Warmup-file.cpp` but goes through the syntax for adding elements to vectors.

3. Two files to fix: `Debug-B1.cpp` and `Debug-B2.cpp`. How do these differ? Fix them, compile them and check how the executables differ.

4. Memory leak:
   ```
   int *p;
   p = new int(5);
   ```
   Use `bcheck` to confirm some memory is leaking. Fix it and use `bcheck` again to confirm it's fixed. You can run bcheck as follows:
   ```
   $ bcheck ./a.out
   ```

5. Demonstrate the use of cerr to report errors. You can direct the error output to somewhere using this type of instruction from the command line.
   ```
   $ ./a.out 2> Errors.txt
   ```
   Standard error will be redirected to the file `Errors.txt`.

6. What does the following code do, with C++11 compilation?

   ```
   int array[10]={3, 5, 7, 10, 3, 5, 4, 4, 9, 3};

   for (auto &element : array)
           cout << element * element << endl;
   ```

   What about using this?

   ```
    int *array = new int[10]{3, 5, 7, 10, 3, 5, 4, 4, 9, 3};
   ```

   You should be able to write an ordinary for loop to make sure the values are in there, but the `for` from above won't work because it's dynamic memory. The error message indicate `begin` and `end` functions used to work out the range don't work! More on those later in the subject.

2 Task Two: File handling

1. Make a copy of the file `/usr/dict/words` in your directory. Write a program to read words from the words file, and output them together with the line number of each word. Include appropriate checks on the opening of the files. Think about the order you do the checking in.
2. Test each of the open modes for files. See what effect each has on a text file you generate.
3. Write a program to write some text to a file, and then read the file back in binary mode.
4. Write a program to read in a file full of digits and produce a digit distribution. You can use the file `digits.txt` as sample data. Output the digit distribution in a sensible format to a file `Distribution.txt`.
5. * How could you find the size of the output buffer?
6. * How about the size of the input buffer?

3 Task Three: Investigating random

1. The file `Old-Random.cpp` uses the old style randomness.
   (a) Run it to see what the output is like.
   (b) Run it several times immediately after each other. Using the following might help:

   ```
   $ ./a.out ;./a.out
   ```

   What is happening? Why?
2. Copy `Old-Random.cpp` to a new file `Old-Random-Count.cpp` and modify this new file so the program produces N integers in the range 0 to 9 inclusive, so effectively digits, with N being a parameter given on the command line. Set up an integer array to count how many of each digit occurs. Output a sensible description of the distribution.
3. Write a program in `New-Random.cpp` that uses the C++11 code in the lecture notes to that uses the randomness engine to produce integers in the range 0 to 9 inclusive from a uniform distribution. Use similar code to the above to output the distribution. Again the number of random digits produced should be based on a command line argument.
4. Write a new program in `Random-normal.cpp` that is similar to the above but runs as

```
./Random-normal N mean stdev
```

where N is the number of digits to be generated, mean is the mean of the normal distribution, and stdev is the standard deviation of the distribution. The output for this version should be rounded to the nearest integer.

5. * Run your `Random-normal.cpp` program and produce test data for the three parameters being `10000 1000 100`. Count the different number of outcomes in the range 500 to 1500.

6. * Mersenne Twister: This is a family of pseudo–random number generators, the best known member being Mersenne Twister 19937.

```
std::random_device randev;
std::mt19937 mt_eng(randev());
```

4 Task Four: Basic struct construction

Carry out the debugging exercise first, `Debug-struct.cpp`.

Create a piece of code `Dog.cpp`. This code should contain a struct `Dog` with the following properties:

- Data fields for name, breed and age.
- A constant static field for the license fee, which is $17.55.
- Functions to set and display the data. You are not to use explicit constructors.
- A `main()` function that demonstrates the struct operates correctly.

```
Dog: Flash is a Spotty thing.
The dog's age is 4
License fee: $17.55
```

5 * Task Five: Memory ...

How much memory can you allocate on Banshee using `new`?