CS205 C/C++ Program Design Assignment3

Author: Jimall

Introduction

In assignment 3, we focus on the following topics, which are introduced in our lectures and labs:

- **features of C/C++ functions containing:**
  - function overloading
  - default arguments
  - template function
  - function template specialization
  - inline function
  - call-by-reference
  - call-by-value
  - call-by-pointer
- **recursion**
- **struct of C/C++ containing:**
  - usage of struct
  - usage of struct pointer
- **file I/O**
- **std::string manipulation**
- **header files**
- **build tools of C/C++ containing:**
  - make
  - cmake
- **shared library**

Tasks of this assignment can be quite easy if you master the above knowledge.

Notice: our requirements assume that you are using Linux/wsl. You can use Windows or macOS, so the generated executable file names and generated shared library names may be different, it's ok.

Here are the tasks:

Tasks

Part 1. Split String (30pt in total, 20pt for code, 10pt for `Makefile`)

Part 2.Product and Bigger (30pt in total, 20pt for code, 10pt for `CMakeLists.txt`)

Part 3. Pascal's Triangle (20pt)

Part 4. CSV file (20pt)

Part One - Split String

In this part, you are given the following files in `assignment3/split_string`:

```
split_string
|-- main.cpp
|-- pair.cpp
`-- pair.h
```

Notice: Please do **NOT** modify `main.cpp`

You need to complete these steps to finish this part:

1. Finish the definition of struct `{pair}` in `pair.h`. `{pair}` is a structure that can store two character values, there are multiple ways to define it, choose anyone of them
2. Implement function `{printPair}` in `pair.cpp`, which is defined and documented in `pair.h`. `{printPair}` should be able to print specific `{pair}` value in one line, separated by space and follow a newline character at last. If you call `{printPair}` twice, you will get output of this format:

   ```
   1 1
   2 2
   ```

3. Implement function `{splitPair}` in `pair.cpp`, it is defined and documented in `pair.h`. `{splitPair}` splits a string into pairs and returns a `{pair}` array. String should be split in this way:

   a. If there are not less than 3 characters in the string, put digits in two consecutive even positions (index 0 and index 2) into one `{pair}`;
   b. If there are not less than 4 characters in the string, put digits in two consecutive odd positions (index 1 and index 3) into another `{pair}`;
   c. Discard all the characters which are used to make pairs;

d. Repeat step.a-c until no more {pair} can be produced.

String ababcdcde will generate a {pair} array that has following outputs:

```
a a
b b
c c
d d
```

You may be doubted why we need a parameter {int& length} in {splitPair} function. Because we cannot return an array directly in C/C++ programming language, so if we want to return a array, we return a pointer of the array. In this case, we don't know the length of the returned array since a pointer doesn't have the length information. So we use a parameter {length} to record the length of the returned {pair} array.

After finishing the three steps, you can use main.cpp to test your code

4. Write a Makefile to build your code. After running make, make should generate a executable file main, which executes the main function in main.cpp. Instead of generating the main directly, your makefile should first generate a shared library libpair.so in lib directory, and then, makefile generates main using the shared library.

   If your codes are implemented as required, the execution results of main will be:

```
> ./main
=== testcase 1 ===
1 1
2 2
3 3
4 4
5 5
=== testcase 2 ===
1 1
2 2
3 3
4 4
```

   Here, the step of generating libpair.so is unnecessary in fact, but you can try to imagine a scenario: we provide a application to our user (that is main here). And one day, our user wants us to provide some functions of our application for them to develop application by themselves. We just need to modify the makefile, then generate some shared libraries and share these libraries to them. That's the big power of make, it can handle many scenarios, even if complex scenarios.

Notice: remove all generated files including lib/libpair.so and main before you submit your code

Topics: struct, usage of struct, usage of struct pointer, call-by-reference, make, shared library

Part Two - Product and Bigger

In this part, you are given the following files in `assignment3/product_and_bigger`:

```
product_and_bigger
|-- bigger.h
|-- main_bigger.cpp
|-- main_product.cpp
`-- product.h
```

Notice: do **NOT** modify `main_bigger.cpp` and `main_product.cpp`

Follow the steps to finish this part:

1. Finish **inline** function `{product}` in `product.h` such that you can get the product of anywhere between 2 and 5 integers. You may modify the function prototypes.

   in short, you should write only **one** function to make this work:

   ```
   // test product of int
   cout << "=== test product of int ===" << endl;
   cout << product(1, 2) << endl;
   cout << product(1, 2, 3) << endl;
   cout << product(1, 2, 3 ,4) << endl;
   cout << product(1, 2, 3, 4, 5) << endl;
   ```

(**hint**: check the topics, you may get some ideas from them)

2. Finish one more **inline** function in `product.h` to overload function `{product}` in step 1 to make `{product}` also work for floating number

after finishing step 1 and step 2, you can use `main_product.cpp` to test your code

3. Use function templates to write a **inline** template function `{bigger}`, which returns the bigger one of two integers or two floating numbers
4. In C/C++, integer literal is int type by default, so 1 is int, a floating number literal is double type by default, so 1.1 is double, and a string literal is const char * type by default, so “abc” is const char . *The template function you write can be incompatible with const char* , do some work to make `{bigger}` return the const char * which is longer. Also, do some work to make `{bigger}` returns the longer `std::string`.

after finishing step 3 and step 4, you can use `main_bigger.cpp` to test your code

5. write a `CMakeLists.txt` and use cmake to build your code, after running cmake and make, your code should generate two executable files: `main_product` and `main_bigger`. Run `main_product` to execute main function in `main_prouct.cpp`, run `main_bigger` to execute main function in `main_bigger.cpp`.

If your codes are implemented as required, the execution results of `main_product` and `main_bigger` will be:

```
> ./main_product
=== test product of int ===
2
6
24
120
=== test product of double ===
0.8
2.4
9.6
48

> ./main_bigger
=== test int ===
2
=== test double ===
2.5
=== test const char * ===
ab
=== test string ===
ab
```

Notice: cmake will generate lots of files, remember to **remove them** before you submit your code

Topics: inline function, default arguments, template function, function template specialization, cmake

Part Three - Pascal's Triangle

Here's a part of the Pascal's triangle [Wiki] [Baidu]:

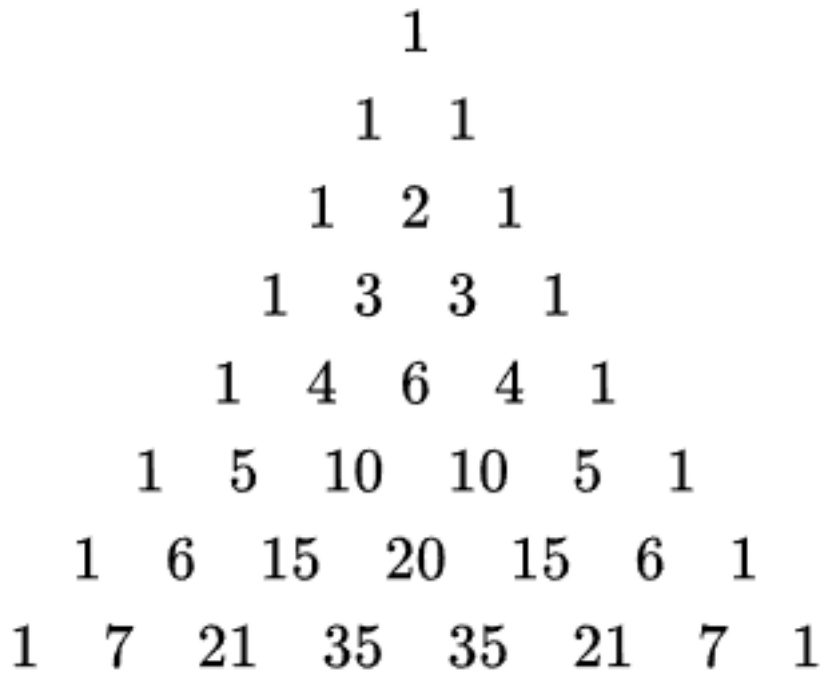

Figure 1: Pascal's Triangle

Most of you should know Pascal's triangle, so it will not be introduced here. If you don't, just read the wiki, this is a simple concept, you will understand it in 2 minutes.

In this part, you will find that Pascal's triangle can be defined easily using recursion.

Firstly we make the triangle look like a 2-dimentional array as below, than we can get a Pascal's triangle by **defining every number in it as the sum of the item above it and the item above and to the left of it**. Use 0 if the item does not exist.

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

Follow the steps to finish this part

1. You should define a function `{int pascal(int row, int column)}` in `pascal_triangle/pascal.h` and implement it in `pascal_triangle/pascal.cpp`, this function returns the value of the item at that position in Pascal's triangle. Rows and columns are zero-indexed; that is, the first row is row 0 instead of 1 and the first column is column 0 instead of column 1.

2. Write some tests in `main.cpp` to test if function `{pascal}` works well
3. Make sure you can run your code using `g++ main.cpp pascal.cpp -o main && ./main` without exception

Here are some test cases:

```
pascal(0, 0) // should return 1
pascal(0, 5) // should return 0
pascal(3, 2) // should return 3
pascal(4, 2) // should return 6
```

Topics: recursion, header files

Part Four - CSV file

You are provided with a text file `assignment3/csv_file/world_cities.csv` that contains this information for many cities in the world (You can add your hometown if you wish). This file contains one line per city with the following information:

- City name
- Province or state (may be absent)
- Country
- Latitude
- Longitude

These values are separated by commas (.csv means "Comma Separated Values", it's a commonly used format for exchanging data). When a piece of data is missing (province or state), you get two commas in succession, for instance (first row in the file): `Aarhus,,Denmark,56.150,10.217`.

You are required to write a program to get information of cities in China from `world_cities.csv` output a file named `china_cities.csv`, which contains information of cities in China and the format should be the same as `world_cities.csv`.

You can use `make`, `cmake`, or directly use `g++ gcc` to build your code of this part, tell us how can we run your code in your report.

Notice: do **NOT** submit `world_cities.csv` and generated files

Topics: file I/O, std::string manipulation

What to submit

1. **your report in pdf format**.

For each part, first you should show your thoughts about the problem solving, then you should give us essential information about your code (for example, tell us how we build your code in part4), at last show us the screenshots of the result.

At the end of the report, you can record difficulties you faced in this assignment.

2. **your source codes** (compress them into frequently-used compressed file format like `.zip` `.7z`), notice that just submit source codes, do not submit generated files like `main` `CMakeCache.txt`, etc.

tips: check your code before submitting

Here are codes given by us:

```
assignment3
|-- csv_file
|   `-- world_cities.csv
|-- pascal_triangle
|-- product_and_bigger
|   |-- bigger.h
|   |-- main_bigger.cpp
|   |-- main_product.cpp
|   `-- product.h
`-- split_string
    |-- main.cpp
    |-- pair.cpp
    `-- pair.h
```

Here are codes your should submit, tagged **[M]** means that file is given by us and should be modified, tagged **[A]** means your should add and write that file yourself, please do **NOT** modify other given files

```
assignment3
|-- csv_file
|   `-- (submit the source codes, no limitation, tell us how we run
   your code in your report)
|-- pascal_triangle
|   |-- [A]main.cpp
|   |-- [A]pascal.cpp
|   `-- [A]pascal.h
|-- product_and_bigger
|   |-- [A]CMakeLists.txt
|   |-- [M]bigger.h
|   |-- main_bigger.cpp
|   |-- main_product.cpp
|   `-- [M]product.h
`-- split_string
    |-- [A]Makefile
    |-- main.cpp
```

```
    |-- [M]pair.cpp
    `-- [M]pair.h
```

There will be some files generated after we build your code, at least these files should be generated, tagged with **[G]**:

```
assignment3
|-- csv_file
|   `-- [G]china_cities.csv
|-- pascal_triangle
|   |-- [G]main
|-- product_and_bigger
|   |-- [G]main_bigger
|   `-- [G]main_product
`-- split_string
    |-- lib
    |   `-- [G]libpair.so
    `-- [G]main
```