Observations report

Name: Jihui. Sheng

ID: 11539324

Course: Cpt_S 427

Deliverable:

An 'observations report'. The report should include the following content:

- Screenshots proving you did perform the tutorial activities (code review exercise checklist)
- Report any bugs, typos, broken links etc
- A brief discussion on the skills you've learned from the tutorial (7 lines maximum)

Screenshots:

National Weather Service/OHD
Science Infusion and Software Engineering Process Group (SISEPG) – C++ Coding Standards and Guidelines Peer Review Checklist

C++ Coding Standards and Guidelines Peer Review Checklist

Last Updated: 25 April 2016

Reviewer's Name:	Lihui Sheng	**Peer Review Date:**	6/25/2021
Project Name:	linked list	**Project ID:** Enter if applicable	
Developer's Name:	JM & AJ	**Project Lead:**	
Review Files & Source code	linked_list.cpp linked_list.h main.cpp		
Code Approved			

The following check list is to be used in the assessment of C++ source code during a peer review. Items which represent the code being reviewed should be checked.

1. General Programming Standards and Guidelines

Refer to the OHD General Programming Standards and Guidelines Peer Review Checklist to assess the adherence to the OHD General Programming Standards and Guidelines.

2. C++ Programming Standards

2.1 Readability and Maintainability

✓ Consistent indentation (3 or 4 spaces)

✗ Consistent use of braces

✗ No tabs used

2.2 File Names

✓ Header files and namespace files use suffixes: `.h`, `.H`, `.hh`, `.hpp`, or `.hxx`

✓ Source files use suffixes: `.C`, `.cc`, `.cpp`, or `.cxx`

✓ `UpperMixedCase` is used for class or namespace file names

✗ `lowerMixedCase` is used for function file names

2.3 File Organization

___ Each file contains only one class declaration or definition except functors and static classes

___ File includes a brief description of the file after the *documentation block*

___ The content of the file is in the following order:

1. The preprocessor directives to prevent multiple inclusions in header files.
2. ___ The *Documentation block* described in the *"OHD General Software Development Standards and Guidelines"*
3. ___ A brief description of the file
4. Include files
5. ___ `#defines` and Macros
6. The '`use`' directives in the source files but not in header files
7. Class or function declaration or definition

2.4 Include Files

___ C++ standard library headers that have no extension are used

___ New prefix `c` is used instead of the old extension `.h` for C standard header files

___ The < > pair for library and system headers is used

___ The " " pair for non-system (user defined) headers is used

___ No absolute or relative paths to point to the header files are used

___ The system header files first in alphabetical order followed by the non system include files (including COTS includes) also in alphabetical order

2.5 Comments

___ The JavaDoc convention format is used for the documentation comment

___ The C++ comment "//" style or the C style (/* ... */) is used for inline comments

2.6 Naming Schemes

___ `namespace`, `class`, `struct`, template argument, and parameter names use uppercase letters as word separators with the first character capitalized

___ Macro and `#defined` constant, `enum`, `union`, class static data member, and global variable names are all capitalized with underscore as separators

___ Class methods and variable names use uppercase letters as word separators with the first character is not capitalized

___ Private class data member names are prepended with the underscore, the rest is

the same as method names

___ `static const` data members are all uppercase

___ `typedef` names reflect the style appropriate to the underlying type

___ Class, struct, variable, and method names that differ by case only are not used

___ C function names follow the *OHD C Programming Standards and Guidelines*

2.7 Class Design

___ Class members are declared in this order: public members, protected members, private members

___ Data members are properly protected (declared as private or protected)

___ Classes (except functors and static classes) implement a default constructor, a virtual destructor, a copy constructor, and an overloaded assignment operator

___ Static classes declare a private default constructor to prevent instantiation

2.8 Safety and Performance

___ Type conversions have been done explicitly. The C++ set of casting operators `static_cast`, `reinterpret_cast`, `const_cast` and `dynamic_cast` have been used instead of C-style casting

___ Global variables are not used except in rare cases and when used include an inline comment describing the reason for use.

___ Dynamically allocated memory is deallocated when no longer needed

___ There is no dangling pointers. Pointers are always tested for `NULL` values before trying to dereference them

___ There is no hardcoded numerical values, const or enum type values are used instead

___ Large objects are created on the heap

___ The arguments specified in a function prototype are associated with variable names

3. C++ Programming Guidelines

3.1 Readability and Maintainability

___ A space is put between the parenthesis and the keywords or the function names

___ A space is put between variables, keywords and operators

___ Pointers are named in some fashion that distinguishes them from other "ordinary" variables

___ Parentheses are used in macros to ensure correct evaluation of the macro

- ✓ The `goto` statement is used very sparingly

3.2 User Defined Types

- ✓ `static const` members are used instead of `#defined` constants
- ✓ Proper `typedefs` are used instead of using templates directly
- ✓ `enum` is used to define a collection of integral constants

3.3 Variables

- ✓ `const` correctness has been practiced
- ✗ All variables are correctly initialized
- ✓ Local variables are declared near their first use.
- ✓ The copy constructor is used to construct an object instead of the assignment operator (=)

3.4 Performance

- ✓ `inline` functions are used instead of Macros
- ✓ The prefix form (`++i` or `--i`) is used instead of postfix form (`i++` or `i--`)
- ✓ Pointer arithmetic has been avoided
- ✓ Repetitive computations are reduced by only doing them once and saving the result in a temporary variable for future access

3.5 Class Design

- ✓ Parts-of relation inheritance has been avoided

Report any bugs, typos, broken links etc:

main.cpp

1.

class UberNode : public int_list

Problem: not a class or struct name.

UberNode *pRoot = new UberNode;

Problem: "UberNode::UberNode()" is inaccessible.

pNew->add_value("COUNT", count);

Problem: function "UberNode::add_value" is inaccessible.

delete pRoot;

Problem: function "UberNode::~UberNode()" is inaccessible.

int count = pRoot->get_count();

Problem: class "UberNode" has no member "get_count".

pLast->append(pNew);

Problem: class "UberNode" has no member "append".

Modify:

```
class UberNode : public int_list_t
{
    public :
        UberNode() {}
        ~UberNode() {}
        void add_value(std::string key, int value)
        {
        m_map[key] = value;
        }
        int value(std::string key) { return m_map[key]; }
    private:
        std::map<std::string, int> m_map;
};
```

2.

double_list_t pDoubleRoot = new double_list_t;

double_list_t *pDoubleLast = pDoubleRoot;

Problem: no suitable constructor exists to convert from "double_list_t *" to "LinkedListNode<double, 75000>".

int count = pDoubleRoot->get_count();

Problem: operator -> or ->* applied to "double_list_t" instead of to a pointer type.

delete pDoubleRoot;

Problem: expression must be a pointer to a complete object type.

Modify:

```
double_list_t * pDoubleRoot = new double_list_t();
double_list_t *pDoubleLast = pDoubleRoot;
```

3.

pDoubleNew->add_value("COUNT", count);

Problem: class "LinkedListNode<double, 75000>" has no member "add_value"

Modify:

Add add_value function into LinkedListNode Class as UberNode Class did.

Linked_list.h

1.

#endif // LINKED_LIST_H

Problem: expected a ';'

Modify:

```
};
```

```
#endif // LINKED_LIST_H
```

Learned from the tutorial:

1. Not coding is better than substandard coding.
2. "Code revies" can help programmers review the code. Reviews can be classified into three categories:
    1) Code Walkthrough
    2) Technical Review
    3) Code Inspection
3. People are imperfect. Engineers are also human, so engineers are not perfect and make mistakes. Attackers will use errors in the code to turn them into vulnerabilities.
4. Dynamic testing is expensive, so before dynamic testing is executed, using static testing (which uses mechanical methods to statically verify the software) can reduce costs, and at the same time can detect and repair defects more effectively.
5. Ignoring false positives leads to a maintenance nightmare.
6. The better development practice steps are:
    1) Careful needs analysis.
    2) Reasonable design practice.
    3) Effective dynamic testing.
    4) Static analysis.
    5) Code review
7. There are four static analysis models:
    1) Syntax and construct analysis.
    2) Class structure and inheritance analysis.
    3) State machine model analysis.
    4) Control and data flow graph analysis.