OBJECT ORIENTED PROGRAMMING AND MACHINE LEARNING

STUDENT NAME: MEENAKSHI SRIDHARAN SUNDARAM

CWID : A20592581

PROFESSOR NAME : DR.WON JAE YI

SUBJECT CODE : ECE 590

SUBMISSION DUE : 21ST OCTOBER 2024, MONDAY

[1] AIM:

To create a report that includes all answers to the questions below, including codes and actual outputs vs. desired outputs.

[2] PLATFORM USED:

The codes were modified and run in VScode after adding, installing, and configuring the necessary C++ environment and saving the file names with an extension of .cpp. The same code could also be run in TurboC but VScode is recommended due to its integration with a Virtual Machine.

[3] BODY:

I. **QUESTION:** Consider the following function access_element_by_index that returns the iterator to the element at the index i and the list l.

```
typedef std::list int_list;
int_list::iterator access_element_by_index(size_t i, int_list &l)
{
assert( ? );
... }
```

Assume the first element of the list is at the index 0, the second at the index 1, and so on. If the function is required to return an iterator that can be dereferenced, determine the precondition of the function, and write an assertion to validate it. (You don't need to implement the function)

ANSWER: According to the given problem statement the index i should be between 0 and l.size() - 1 (inclusive) because:

- i must be less than l.size() to avoid accessing out-of-bounds.
- i must be non-negative, but since it is an unsigned size_t, this is inherently enforced.

Hence the assertion statement of

```
assert(i < l.size());
```

can be added which ensures that the index i is valid for the list l. This assertion guarantees that the function will not try to access an out-of-bounds index, preventing runtime errors.

II. **QUESTION:** Consider a container of type std::map. A C++ function find_or_throw will search for a value from one such container given a key. It will return the value if the key exists and throw std::runtime_error otherwise. Design the function interface (parameters and return type) and implement the function body.

ANSWER: the find_or_throw function, searches for a key in std::map<std::string, int>. If the key is found, the function returns the corresponding value; otherwise, it shows a std::runtime_error. The function returns an int (the value corresponding to the key). Figures 2.1 and 2.2 shows the code snippet for find or throw function along with it's respective outputs shown in figure 2.3.

hw3.cpp

home > ece449 > hw3.cpp > ...

```cpp
#include <iostream>
#include <map>
#include <stdexcept>
#include <string>

// Function to find a key in the map and return its value.
// If the key is not found, throws a std::runtime_error.
int find_or_throw(const std::map<std::string, int>& m, const std::string& key) {
    // Search for the key in the map
    auto it = m.find(key);

    // If the key is not found, throw an exception
    if (it == m.end()) {
        throw std::runtime_error("Key not found: " + key);
    }

    // If the key is found, return the corresponding value
    return it->second;
}

int main() {
    // Example map with string keys and integer values
    std::map<std::string, int> example_map = {
        {"apple", 1},
        {"banana", 2},
        {"cherry", 3}
    };

    try {
        // Case where the key exists
        int value = find_or_throw(example_map, "banana");
        std::cout << "Value found: " << value << std::endl;

        // Case where the key does not exist, should throw an exception
        value = find_or_throw(example_map, "orange");
```

Figure 2.1: code for find and throw (a)

```cpp
        std::cout << "This will not be printed!" << std::endl;
    }
    catch (const std::runtime_error& e) {
        // Catch the exception and print the error message
        std::cerr << "Error: " << e.what() << std::endl;
    }

    return 0;
}
```

Figure 2.2: code for find and throw (b)

```
PROBLEMS  DEBUG CONSOLE  TERMINAL  OUTPUT  PORTS

Value found: 2
Error: Key not found: orange
[1] + Done                "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft
-MIEngine-In-ulfzqeof.12c" 1>"/tmp/Microsoft-MIEngine-Out-yaddfpjw.eeq"
ece449@ece449:~$
```

Figure 2.3: output of a key to a found value and a value not found

III. **QUESTION:** Assume there is no compiling or linking error. Review the following pieces of code and briefly explain potential issues.

A.
```cpp
std::string &get_hello() {
    std::string s = "hello";
    return s;
}
```

B.
```cpp
class time {
    int hour, min, sec;
public:
    bool set(int h, int m, int s);
    int get_hour() const;
    int get_min() const;
    int get_sec() const;
};
```

C.
```cpp
class collection {
};
class my_array : public collection {
    std::vector<int> vec_;
};
void test() {
    collection *p = new my_array;
    delete p;
}
```

ANSWER: When the above-mentioned code was run in VScode, the errors shown in Figure 3.1 were reported however more conceptual errors would not be shown as an error but result in undesired output.

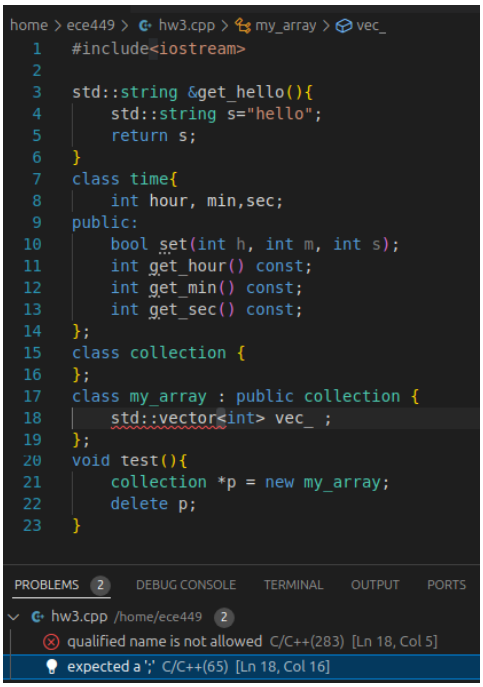

Figure 3.1: code and the output showing errors

Firstly, with the function get_hello() returning a reference to a local string variable s. Since s is a local variable when the function returns, it will be destroyed which leaves the reference to point to a non-existing object. Instead, we can make the function return by value instead of a reference.
For example: **std::string get_hello() {**

return "hello";

}

The next error could be with the class time where the set() function is declared but never implemented in the function. The set function should be added with a validation to ensure that the values are within their valid ranges which are hours in the range of 0 to 24, minutes and seconds in the range of 0 to 60.

For example: **bool set(int h, int m, int s)**

{ if (h < 0 || h > 23 || m < 0 || m > 59 || s < 0 || s > 59) {

return false; // Invalid time

} hour = h;

min = m;

sec = s;

return true; }

the next problem resides with class collection and my_array, where the base class (collection) lacks a virtual destructor, which will result in my_array not getting called properly when an attempt to delete a my_array object through a pointer to the collection is made. This can cause resource leaks. To

rectify this error making the destructor of the collection virtual will ensure that the destructor of my_array is called correctly when the object is deleted through a pointer to the collection.

For example,

class collection {

public:

virtual ~collection() {}

};

IV. Consider the classes base and derived as follows.

```
class base {
protected:
    virtual void step_one() {std::cout << "base::step_one" << std::endl;}
    virtual void step_two() {std::cout << "base::step_two" << std::endl;}
public:
    void run() {
        std::cout << "enter base::run" << std::endl;
        step_one();
        step_two();
        std::cout << "exit base::run" << std::endl;
    }
};

class derived : public base {
protected:
    void step_one() {std::cout << "derived::step_one" << std::endl;}
    void step_two() {std::cout << "derived::step_two" << std::endl;}
public:
    void run() {

            std::cout << "enter derived::run" << std::endl;
            step_one();
            step_two();
            std::cout << "exit derived::run" << std::endl;
        }
     };
```

A. What's the output of the following function test1?

```
void test1() {
    derived d;
    derived *p = &d;
    p->run();
}
```

B. What's the output of the following function test2?

```
void test2() {
    derived d;
    base *p = &d;
    p->run();
}
```

ANSWER: the above code was run on VScode environment with relevant header files and output testing methods as given in figure 4.1 and 4.2

```cpp
#include <iostream>

class base {
protected:
    virtual void step_one() {
        std::cout << "base::step_one" << std::endl;
    }

    virtual void step_two() {
        std::cout << "base::step_two" << std::endl;
    }

public:
    void run() {
        std::cout << "enter base::run" << std::endl;
        step_one();  // Virtual, can be overridden
        step_two();  // Virtual, can be overridden
        std::cout << "exit base::run" << std::endl;
    }
};

class derived : public base {
protected:
    void step_one() override {
        std::cout << "derived::step_one" << std::endl;
    }

    void step_two() override {
        std::cout << "derived::step_two" << std::endl;
    }
```

Figure 4.1: code (a)

```cpp
public:
    void run() {
        std::cout << "enter derived::run" << std::endl;
        step_one();  // Calls derived's version
        step_two();  // Calls derived's version
        std::cout << "exit derived::run" << std::endl;
    }
};

// A. Output of test1
void test1() {
    derived d;
    derived *p = &d;
    p->run();
}

// B. Output of test2
void test2() {
    derived d;
    base *p = &d;  // Base pointer to derived object
    p->run();      // Calls base::run, but step_one/step_two are overridden
}

int main() {
    std::cout << "Running test1:" << std::endl;
    test1();
    std::cout << std::endl;

    std::cout << "Running test2:" << std::endl;
    test2();
    return 0;
}
```

Figure 4.2:code(b)

The following outputs shown in figure 4.3 were obtained when the codes were debugged and run. The object d of type derived is first created where the pointer p points to the address of d. Since the pointer points to a derived object and the call to p-> run() is resolved

using the method from the derived class, the function derived::run() will be executed first which in turn will call derived::step_one() and derived::step_two().

For the output of test2(), an object d is created the same as test1() but this time the pointer p is of type base* but points to d which is the derived object. Since run() is not virtual in base, the call to p-> run() will invoke the base::run() method, and not the derived run() method.

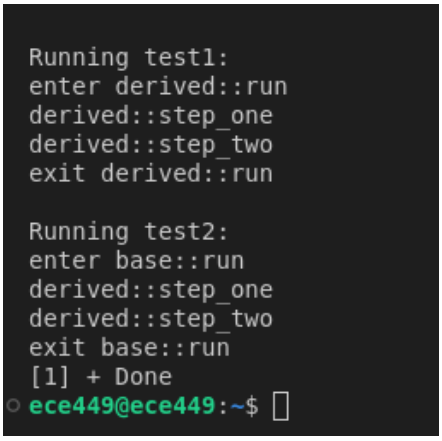

Figure 4.3: output of test1 and test2