Lecture 11 Class and Const Correctness

#CS106L

- **Finishing up the BSTLR**
- **Announcements**
- **Brief Intro to Classes**
- **Everything Const!**

Now, Unix mostly uses `.cc`, and outside Unix mostly uses `.cpp`.

`.h` is technically for C programs, so if mixing C and C++ code, use `.hh` instead.

const pointer:

```
//constant pointer to a non-constant int

        int * const p;

//non-constant pointer to a constant int

        const int* p;

        int const* p;

//constant pointer to a constant int
        const int* const p;

        int const* const p;
```

const iterator:

- Remember that iterators act like pointers
- const `vector<int>::iterator itr` ,however, acts like `int* const itr`
- To make an iterator read only, defifine a new `const_iterator`

recap:

● For the most part, always anything that does not get modified should be marked const

● Pass by const reference is better than pass by value

○ Not true for primitives (bool, int, etc)

● Member functions should have both const and non const iterators

● Read right to left to understand pointers

challenge:

```
const int* const myClassMethod(const int* const & param) const;
```

- parameter:a constant reference to a constant pointer to an integer(zxq:`const int* const & param` may be equivalent to `const int* const param`)
- return:a constant pointer to a constant integer
- functions:a const member function

Q1:std::search?

The `std::search` algorithm is a standard algorithm in C++ that searches for a sequence of elements (known as the "pattern") within another sequence of elements (known as the "haystack"). It returns an iterator to the first element of the first occurrence of the pattern within the haystack, or an iterator to the end of the haystack if the pattern is not found.

Here is an example of how to use `std::search` to find a pattern within a string:

```
std::string haystack = "The quick brown fox jumps over the lazy dog.";

std::string pattern = "fox";
```

```
        auto result = std::search(haystack.begin(), haystack.end(),
pattern.begin(), pattern.end());

        if (result != haystack.end()) {

    std::cout << "Pattern found at index: " << std::distance(haystack.begin(),
result) << std::endl;

        }

        else std::cout << "Pattern not found." << std::endl;
```

In this example, the function takes four arguments:

- the first is the start of the haystack range,the second is the end of the haystack range;
- the third is the start of the pattern range,the fourth is the end of the pattern range.

In the example, the function returns an iterator pointing to the first element of the pattern "fox" within the haystack, which is the letter "f". To get the index of the pattern, we use `std::distance` function to calculate the distance between the iterator returned by `std::search` and the start of the haystack. The output of this example would be "Pattern found at index: 16".

It's also important to note that, `std::search` uses the == operator to compare the elements in the pattern and the haystack, if you want to use a different comparison function, you can pass it as an additional argument to the `std::search` function.

Q2: `.h` and `.cpp`?

n C++, ".h" (header) files and ".cpp" (implementation) files are used to organize and separate the interface and implementation of a program.

A ".h" file, also known as a header file, is used to define the interface of a program. It contains the declarations of types, functions, and variables that will be used in the program. These declarations provide the necessary information for the compiler to check the program for errors, but they do not include the actual implementation (i.e. the code that performs the actions).

The header file should only contain declarations and should not contain any executable code.

A ".cpp" file, also known as an implementation file, is used to define the implementation of a program. It contains the code (i.e. the instructions that perform the actions) that corresponds to the declarations in the header file. The implementation file should only contain code and should not contain any declarations.

When you are writing a program, you should create a header file for each class or module you are going to implement and put the class/module declaration in the header file. Then, you should create a corresponding implementation file for each class/module, where you will put the class/module's function and variable's implementation.

By separating the interface (header file) from the implementation (source file), you can organize your code better and keep the implementation details hidden from the user. This allows you to change the implementation without affecting the rest of the program and it also allows you to use the same implementation in multiple programs.

Q3: `construct` and `destruct`?

A constructor is a member function that is automatically called when an object of a class is created. It is used to initialize the object's data members and perform any other setup that is required. A class can have multiple constructors with different parameters, this feature is called constructor overloading.

A destructor is a member function that is automatically called when the object goes out of scope or explicitly deleted. It is used to release any resources that the object has acquired, such as dynamically allocated memory. A class can have only one destructor.

1. Initializing member variables:

```
class MyClass {

public:

    MyClass(int x, int y) : x(x), y(y) { }

    int x, y;
```

```
};

  MyClass obj(5, 10); // obj.x = 5, obj.y = 10
```

2. Allocating dynamic memory:

```
class MyClass {

public:

    MyClass(int size) {

        data = new int[size];

    }

    ~MyClass() {

        delete[] data;

    }

    int* data;

};

MyClass obj(100);
```

3. Opening a file:

```
class MyFile {

public:

    MyFile(const char* fileName) {

        file = fopen(fileName, "r");

    }

    ~MyFile() {

        fclose(file);
```

```
    }

    FILE* file;

};

MyFile file("myfile.txt");
```

Q4:Why don't we use global variables?

1. Global variables can lead to naming conflicts and naming collisions. As global variables are visible throughout the entire program, it is easy for two different parts of the code to use the same variable name and cause naming conflicts.
2. Global variables are not thread-safe. If multiple threads of execution access a global variable simultaneously, it can lead to race conditions and other concurrency issues.
3. Global variables can lead to tight coupling between different parts of the code. When one part of the code modifies a global variable, it can have unexpected effects on other parts of the code that are also using that variable.
4. Global variables make it harder to reason about the state of the program. As global variables are visible throughout the entire program, it can be difficult to understand how they are being used and what their current value is.
5. Global variables can be changed by any part of the code at any time, making it harder to debug and understand the behavior of the program.
6. Global variables can be hard to test as they can be accessed and modified by any part of the code, making it difficult to control their state during testing.

Instead of using global variables, it's generally recommended to use local variables or member variables, and to use function arguments and return values to pass data between different parts of the code.

Q5:why my code doesn't work?

```
struct Planet {

        int countPopulation() const;

        void deathStar();
```

```
};

void evil(const Planet &p) {

        cout << p.countPopulation() << endl;

        p.deathStar();

}
```

The code you provided has a couple of issues:The member function `deathStar()` is not declared as `const`, but it is being called on a const object `p` in the `evil()` function. Since `deathStar()` is not declared as const, it can't be called on a const object, but it can change the state of the object.

To fix this issue, you can either:

1. Declare the `deathStar()` function as `const`:

```
struct Planet {

    int countPopulation() const;

    void deathStar() const;

};
```

This way, it can be called on a const object, but it can't change the state of the object.

2. Remove the `const` keyword from the object `p` in the `evil()` function:

```
void evil(Planet &p) {

    cout << p.countPopulation() << endl;

    p.deathStar();

}
```

This way, you can call non-const member functions on the object, but you'll lose the ability to call this function on a const object.

3. If you want to keep the object `p` const but still able to call `deathStar()` on it, you should change the implementation of the `deathStar()` function to make sure it doesn't change the state of the object.

In summary, you can't call non-const member functions on a const object, because non-const member functions can change the state of the object and in this way, it will break the constness of the object. The error message is due to the fact that you are trying to call a non-const member function on a const object.

Q5:class declarations?

In C++, classes are the building blocks of object-oriented programming. They provide a way to define custom data types and to encapsulate data and behavior in a single unit. Here are a few examples of how to declare classes in C++:

1. A simple class with data members and member functions:

```
class MyClass {

public:

    MyClass(int x, int y) : x(x), y(y) { }

    int getX() const { return x; }

    int getY() const { return y; }

    void setX(int x) { this->x = x; }

    void setY(int y) { this->y = y; }

    int x, y;

};
```

2. A class with private data members and public member functions:

```
class MyClass {

public:

    MyClass(int x, int y) : x(x), y(y) { }
```

```
    int getX() const { return x; }

    int getY() const { return y; }

    void setX(int x) { this->x = x; }

    void setY(int y) { this->y = y; }

private:

    int x, y;

};
```

In this example, the data members `x` and `y` are declared as private, which means that they can only be accessed from within the class. The member functions `getX()`, `getY()`, `setX()`, `setY()` are declared as public, which means that they can be accessed from anywhere.

3. A class with a constructor and a destructor:

```
class MyClass {

public:

    MyClass() { /* constructor code here */ }

    ~MyClass() { /* destructor code here */ }

};
```

4. A class with a copy constructor and copy assignment operator:

```
class MyClass {

public:

    MyClass(const MyClass& other) { /* copy constructor code here */ }

    MyClass& operator=(const MyClass& other) { /* copy assignment operator code
here */ }

};
```

In this example, the class has a copy constructor and a copy assignment operator. These are special member functions that are used to create a new object that is a copy of an existing object, or to assign the value of an existing object to another object of the same class.

The copy constructor is a constructor that takes another object of the same class as a parameter, and it creates a new object that is a copy of that object. The copy assignment operator is a member function that takes another object of the same class as a parameter and assigns the value of that object to the current object.