【C++】Day24

Class	C++
Date	@December 15, 2021
Material	
Series Number	
Summary	

【Ch7】Class

7.3 Additional Class Features

7.3.1 Class Members Revisited

We deifine a `Screen` class:

```
class Screen {
public:
  typedef std::string::size_type pos;
private:
  pos cursor = 0;
  pos height = 0, width = 0;
  std::string contents;
};
```

We defined `pos` in the `public` part of Screen because we want users to use that name.

We can also use type alias:

```
using pos = std::string::size_type;
```

Unlike ordinary members, members that define types must appear before they are used. As a result, type members usually appear at the beginning of the class.

Member Functions of class Screen

Add constructors to our Screen class:

```
class Screen {
public:
  typedef std::string::size_type pos;
  Screen() = default; //default constructor
  Screen(pos ht, pos wd, char c) : height(ht), width(wd), contents(ht * wd, c) {}
  //implicitly inline
  char get() const {
    return contents[cursor];
  }
  //explicitly inline
  inline char get(pos ht, pos wd) const;
  Screen &move(pos r, pos c); //can be made inline later

private:
  pos cursor = 0;
  pos height = 0, width = 0;
  std::string contents;
};
```

Making Members inline

Classes often have small functions that can benefit from being inlined. Member functions defined inside the class are automatically inline.

Screen's constructors and the version of `get()` that returns the character denoted by the cursor are inline by default.

We can explicitly declare a member function as inline as part of its declaration inside the class body. Alternatively, we can specify inline on the function definition that appears outside the class body.

```
//specify inline on the definition(not declared inline in the declaration)
inline Screen &Screen::move(pos r, pos c) {
  pos row = r * width; //compute the row location
  cursor = row + c; //move cursor to the column within that row
  return *this; //return this object as an lvalue
}

//declared as inline in the class(no need to redeclare as inline)
char Screen::get(pos r, pos c) {
  pos row = r * width; //compute the row
```

```
  return contents[row + c]; //return character at the given column
}
```

Although we are not required to do so, it is legal to specify inline on both the declaration and the definition. However, specifying inline only on the definition outside the class can make the class easier to read.

Overloading Member Functions

Member functions may be overloaded so long as the functions differ by the number and/or types of parameters.

For example, our `Screen` class defined two version of get. One version returns the character currently denoted by the cursor; the other returns the character at a given position specified by its row and column.

The compiler uses the number of arguments to determine which version to run:

```
Screen myscreen;
char ch = myscreen.get(); //calls Screen::get()
ch = myscreen.get(0, 0); //calls Screen::get(pos, pos)
```

mutable Data Members

It sometimes happens that a class has a data member that we want to be able to modify, even inside a const member function.

We indicate such members by including the `mutable` keyword in their declaration.

A `mutable` data member is never const, even when it is a member of a const object. Accordingly, a const member function may change a mutable member.

We will give Screen a mutable member named `access_ctr`, which we'll use it to track how often each Screen member function is called:

```
class Screen {
public:
  void some_member() const;
```

```
private:
  mutable size_t access_ctr; //may change even in a cosnt object
};

void Screen::some_member() cosnt {
  ++access_str; //keep a count of the calls to any member function
}
```

Initializers for Data Members of Class Type

We define a Window_mgr class to manage the Screens.

```
class Window_mgr {
private:
  //Screens this Window_mgr is tracking
  //by default, a Window_mgr has one standard sized blank Screen
  std::vector<Screen> screens {Screen(24, 80, ' ') };
} ;
```

Note: when we provide an in-class initializer, we must do so following an = sign or inside braces.

Exercise

Exercises Section 7.3.1

Exercise 7.23: Write your own version of the `Screen` class.

Exercise 7.24: Give your `Screen` class three constructors: a default constructor; a constructor that takes values for height and width and initializes the contents to hold the given number of blanks; and a constructor that takes values for height, width, and a character to use as the contents of the screen.

Screen.h

```
#include <string>
#include <iostream>

using std::cout;
using std::cin;
using std::string;

class Screen {
public:
    using pos = string::size_type;
    Screen() = default;
    Screen(pos ht, pos wd) : height(ht), width(wd), contents(ht * wd, ' ') {}
    Screen(pos ht, pos wd, char c) : height(ht), width(wd), contents(ht * wd, c) {}
    inline char get() const;
    inline char get(pos, pos) const;
    inline Screen &move(pos, pos);
    inline pos getCallCount() const;

private:
    pos cursor = 0;
    pos width = 0, height = 0;
    mutable pos call_count = 0;
    std::string contents;
};

char Screen::get() const {
    ++(this->call_count);
    return contents[cursor];
}

char Screen::get(pos r, pos c) const {
    ++(this->call_count);
    pos row = r * this->width;
    return contents[row + c];
}

Screen &Screen::move(pos r, pos c) {
    ++(this->call_count);
    pos row = r * width;
    this->cursor = row + c;
    return *this;
}

Screen::pos Screen::getCallCount() const {
    return this->call_count;
}
```

ScreenMain.c

```
#include "Screen.h"

int main(int argc, char* argv[]) {
    Screen s = Screen(30, 30);
    s.move(20,20);
    std::cout << s.get() << std::endl;
    std::cout << s.getCallCount() << std::endl;
    return 0;
}
```