Object Oriented Programming in C++

CS 247

University of Waterloo

cs247@uwaterloo.ca

May 12, 2017
Overview

1. Constructors
2. Assignment Operators
3. Initialization Lists
4. Const
A Shallow Copy copies the object and the object’s pointers’ addresses. This means the original and copied pointer will refer to the same object. A Deep Copy copies the object and copies what its pointers point to. This means the original copied pointers will point to different objects.
The “Rule of Five” is that if a class defines a copy constructor, move constructor, copy assignment operator, move assignment operator or a destructor it should define all five. The reasoning behind this is that if a deep copy is required then the destructor will need to deallocate memory used by the deep copy.
Copy Constructors

The copy constructor for a class is a constructor that takes a reference to an object of the same class as its only parameter. If you do not include a copy constructor a default one will be provided that performs a shallow copy.

Node(const Node &);
`Node::Node(const Node & other):
data{other.data},
next{other.next ? new Node{*(other.next)} : nullptr}{}
There are three places where copy constructors are called:

- When creating a new object and initializing it with values from another object of the same type.
- When passing an object by value.
- When an object is returned from a function by value.
The copy assignment operator (operator = ) is used to copy one object into another existing object. If you do not declare a copy assignment operator a basic one which performs a shallow copy will be provided.
The copy and swap idiom is a way of avoiding code duplication and easily made errors. Assignment operators may be implemented by creating a new object of the same type with the copy constructor then swapping the old values of the object being assigned to with the values in the newly created object.
std::swap

void swap(T& a, T& b);
Copy and Swap Implementation

void Node::swap(Node & other) {
    using std::swap;
    swap(data, other.data);
    swap(next, other.next);
}

// copy and swap idiom
Node & Node::operator=(const Node & other) {
    Node tmp{other};
    swap(tmp);
    swap(tmp);
    return *this;
}
Initialization lists can be used as part of a constructor to set the values of variables within an object.
It may seem that setting values in an initialization list is no different than doing so in the body of the constructor, however, using the initialization list can be necessary for a few reasons:

- References and const members cannot be set outside the initialization list.

- Any object not initialized in the initialization list will have its default constructor called which may cause unnecessary computation or unwanted side effects.
One issue to be noted is that the values set in the initialization list are set in the order that they are declared in the class definition, not in the order they appear in the list. This can cause problems when attempting to set values based on the value of other variables being initialized.
class C{
    int x;
    int y;
public:
    C();
};

C::C(): y{1}, x{y} {} // illegal
The C++ keyword const is a promise not to change the value stored by variables.

- Const Variables
- Const Functions
- Const References
We cannot leave const variables uninitialized.
The const keyword should be applied to the type to the left of it unless it’s at the leftmost position.
const int* x
int const * x

x is a pointer to a constant integer. *x cannot be changed, but x can.

int* const y

y is a constant pointer to an integer. y cannot be changed, but *y can.

const int* const z

z is a constant pointer to a constant integer. Neither z nor *z can be changed.
Scope of Constants

Even if defined outside of a pair of scope braces the scope of a constant will be limited to the file where it is defined by default. To avoid this the keyword extern is needed. In the file providing the constant:

```c
extern const double pi = 3.1415926535;
```

In the file using the constant:

```c
extern const double pi;
```
A function declared as:

<return type> MyClass::f() const;

promises not to change any values in the MyClass object it is called on. It may be understood as having the “this” pointer being a pointer to a constant object.

- Standalone functions cannot be const.
- Const objects can only have const functions called on them.
Often variables should be passed to and returned from functions as const references. Passing by const reference provides:

- Increased efficiency when passing objects since the object does not need to be copied since it is a reference.
- A promise that the original object will not be changed despite being a reference since it is constant.
- The ability to pass literals and temporary values because the reference is constant.
#include <iostream>
using namespace std;

void print(const string& word) {
    cout << "Printing: " << word << endl;
}

int main() {
    string word;
    getline(cin, word);
    print(word);
    print("Hello World!");
}

Suggestions for A1

- Class relation
- What’s different
- Json, Boost Documentation
- Provided files
The End