Overview

1. Constructors
2. Assignment Operators
3. Initialization Lists
4. Const
A **shallow copy** simply copies the member of the object by value. This means a pointer member and the pointer in the copy will refer to the same object.

A **deep copy** copies the object and create copies for what its member pointers point to. This implies that the pointer in the object and it’s copy will point to different objects.
The **Rule of Five** states that if a class defines any of

- copy constructor
- move constructor
- copy assignment operator
- move assignment operator
- destructor

it usually should define all five.

If one of the above methods are defined, it is usually the case that we would require special handling of some resources that is acquired and released explicitly (e.g. heap memory might require a deep copy); in this case, all of those methods should be taken care of.

**Question**: Think about a case where we have some resources in the class definition as described above, but Rule of Five does not apply.
The copy constructor for a class is a constructor that takes a const 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 shallow copy.

Node(const Node &);
Copy Constructors

Assume we have a class Node.

class Node{
    int data,
    Node *next;
public:
    // Assume the class definition is complete
    .......
};

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. \(^1\)

\(^1\)Not exactly the case. See *copy (or move) elision*
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.
Copy Constructors

For the swapping we will just use the provided std::swap

// defined in <utility>
namespace std{
    template<class T>
    void swap( T& a, T& b );
}

Copy and Swap Idiom Implementation

```cpp
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;
}
```
Move Semantics

Move constructor and move assignment operator are similar to copy constructor and copy assignment operator; however

- the argument is a rvalue (in C++14), and
- the resource in the argument is stolen to the object created or assigned.
Move Semantics

Node::Node(Node &&other): data{other.data}, next{other.next}{
    other.next = nullptr;
}

Node &Node::operator=(Node &&other){
    delete next;
    data = other.data;
    next = other.next;
    other.next = nullptr;
}

**Exercise:** Write another version of move assignment operator that takes advantage on the assumption that the right hand side will be destroyed soon.
Initialization lists can be used as part of a constructor to set the values of variables within an object.
Initialization Lists

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 initialized 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.

Question: What if the object member does not have a default constructor?
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:
    // the value of x is undefined
    C():y{1}, x{y}{};
};
The C++ keyword const provides a promise not to change the logical value of the variable.

- Const Variables
- Const Methods
- Const References
We cannot leave const variables uninitializd.

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

In the file providing the constant:

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

In the file using the constant:

```c
extern const double pi;
```

Note that we need to have extern keyword for telling the compiler that this variable is defined elsewhere.

Note: Avoid using global (constant) objects, since the order of initialization of different global objects, if defined in different files, are undefined.
A method declared as:

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

promises not to change the logical value of the object it is called on. It may be understood as this pointer having type const MyClass *.

- Standalone functions cannot be const.
- Const objects can only have const methods 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!");

}
The End