New Class Features

Special member functions

In C++11, there a total of 6 special member functions:

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

Under certain circumstances they are defined by the compiler even if not defined by the user. The following table provides good summaries as to when implicit member functions will be defined by compiler. Check out the C++11 ISO standard for rigid legal descriptions.

Member function	implicitly defined	default definition
Default constructor	if no other constructors	does nothing
Destructor	if no destructor	does nothing
Copy constructor	if no move constructor and no move assignment	copies all members
Copy assignment	if no move constructor and no move assignment	copies all members
Move constructor	if no destructor, no copy constructor and no copy nor move assignment	moves all members
Move assignment	if no destructor, no copy nor move constructor, and no copy assignment	moves all members

The rule of three/five/zero

- Rule of three (before C++11): if a class requires a user-defined destructor, a user-defined copy constructor, or a user-defined copy assignment operator, it almost certainly requires all three.
- Rule of five (after C++11): Because the presence of a user-defined destructor, copy-constructor, or copy-assignment operator prevents implicit definition of the move constructor and the move assignment operator, any class for which move semantics are desirable, has to declare all five special member functions:
- Rule of Zero (after C++11): Classes that have custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership (which follows from the Single Responsibility Principle). Other classes should not have custom destructors, copy/move constructors or copy/move assignment operators.

Rule of Zero basically means that one should never use a raw pointer to manage a resource. Therefore no destructor, copy constructor, copy assignment operator, move constructor and move assignment operator has to be implemented. In the following example, the management of char pointers are delegated to std::string, does the class in business logic is not required to have any custom destructors, copy/move constructors or copy/move assignment.

```
class rule_of_zero
{
    std::string cppstring;
 public:
    rule_of_zero(const std::string& arg) : cppstring(arg) {}
};
```

Defaulted and Deleted Methods

Suppose that you wish to use a defaulted function that, due to circumstances(such as the exceptions listed above), isn’t created automatically, you can use the keyword default to explicitly declare the defaulted versions of these methods:

```
class Someclass {
public:
Someclass(Someclass &&);
Someclass() = default; // use compiler-generated default constructor Someclass(const Someclass &) = defaul
Someclass & operator=(const Someclass &) = default;
...
};
```

The delete keyword, on the other hand, can be used to prevent the compiler from using a particular method.

```
class Someclass {
public:
Someclass() = default; // use compiler-generated default constructor
// disable copy constructor and copy assignment operator:
Someclass(const Someclass &) = delete;
Someclass & operator=(const Someclass &) = delete;
// use compiler-generated move constructor and move assignment operator:
Someclass(Someclass &&) = default;
Someclass & operator=(Someclass &&) = default;
Someclass & operator+(const Someclass &) const;
... };
```

Only the six special member functions can be defaulted, but you can use delete with any member function. Now suppose the Someclass definition is modified thusly:

```
class Someclass {
public:
...
void redo(double);
void redo(int) = delete; ...
};
```

In this case, the method call `sc.redo(5)` matches the redo(int) prototype. The compiler will detect that fact and also detect that redo(int) is deleted, and it will then flag the call as a compile-time error.

Managing Virtual Methods: override and final

Suppose the base class declares a particular virtual method, and you decide to provide a different version for a derived class. This is called overriding the old version. But a redefined method in the derived class doesn't just override the base class declaration with the same function signature. Instead, it hides all base-class methods of the same name, regardless of the argument signatures.

```
struct base {
    virtual void say(char c) {
        cout << "base::parameter is 'char c':"<< endl;
    }
};
struct deriv : base {
    virtual void say(char * c) {
        cout << "deriv::parameter is 'char * c': "<< endl;
    }
};
int main()
{
    char c='c';
    deriv deriv_obj;
    deriv_obj.say(c); // won't compile!
}
```

With C++11, you can use the virtual specifier override to indicate that you intend to override a virtual function. Place it after the parameter list. If your declaration does not match a base method, the compiler objects.

```
struct deriv : base {
    virtual void say(char * c) override {
        cout << "deriv::parameter is 'char * c': "<< endl;
    }
};
```

Using the Apple compiler, I got the following message: `main.cpp:11:18: 'say' marked 'override' but does not override any member functions`

The specifier final addresses a different issue.You may find that you want to prohibit derived classes from overriding a particular virtual method.

```
struct base {
    virtual void say(char c) final {
        cout << "base::parameter is 'char c':"<< endl;
    }
};
```

In the derived class, we cannot define another "virtual void say(char c)" method. However, we can define "virtual void say(char * c)".