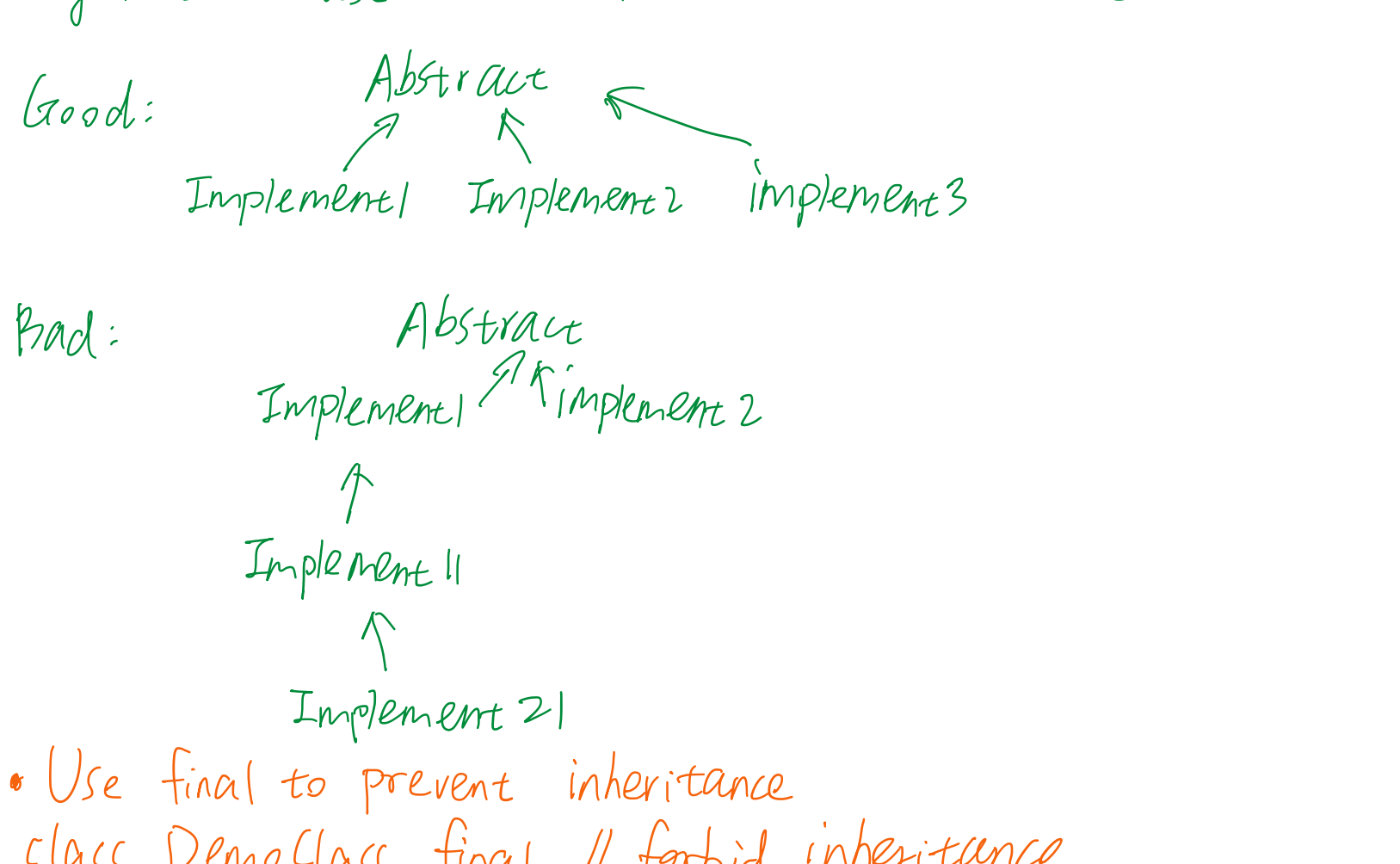
Morden cpp note — class

Thursday, December 3, 2020 10:11 AM

- Abstraction
- Encapsulation

Try not to use virtual function and inheritance.



- Use final to prevent inheritance

```
class DemoClass final // forbid inheritance
{
   ...
};
```

When it's necessary to use inheritance, only use public, but not use virtual, protected. When we reach the end of inheritance, also use final.

```
class Interface        // can be inherited
{ ...
};

class Implement final :  // disallow inheritance
      public Interface  // only use public for inheritance.
{
   ...
};
```

公有继承(public)：当一个类派生自公有基类时，基类的公有成员也是派生类的公有成员，基类的保护成员也是派生类的保护成员，基类的私有成员不能直接被派生类访问。

实例：

```
#include <iostream>

using namespace std;

//基类
class Shape
{
  public:
     void setWidth(int w)
     {
        width = w;
     }
     void setHeight(int h)
     {
        height = h;
     }
  protected:
     int width;
     int height;
};

//派生类
class Rectangle : public Shape
{
   public:
      int getArea()
      {
        reture (width * height);
      }
};

int main(void)
{
  Rectangle Rect;
  Rect.setWidth(5);
  Rect.setHeight(7);

  //输出对象的面积
  cout << "Total area: " << Rect.getArea() << endl;

  return 0;
}
```

- Use default and delete in C++11:

for the constructor and destructor, we should tell the compiler that we need those:

```
class DemoClass final
{
public:
     DemoClass() = default;  // Use default constructor
    ~DemoClass() = default;  // Use default destructor.
};
```

delete can be used to disable functions:

```
class DemoClass final
{
public:
   DemoClass(const DemoClass &) = delete; // 禁止拷贝构造.
   DemoClass & operator=(const DemoClass &) = delete; // 禁止拷贝赋值.
};
```

因为C++有隐式构造和隐式转型的规则，如果类里有单参数的构造函数，或者转型操作符函数，为了防止意外的类型转换，就要使用explicit标记为显式。

```
class DemoClass final
{
public:
   explicit DemoClass(const string_type& str)
   { ...     // 显式单参构造函数.
   }

   explicit operator bool()
   { .... // 显式转型为bool.
   }
};
```

From GeeksforGeeks:
Use of explicit keyword in C++:

```
#include <iostream>

using namespace std;

class Complex
{
private:
     double real;
     double imag;

public:
   // Default Constructor
   Complex(double r = 0.0, double i = 0.0):
        real(r), imag(i) {}

   // A method to compare two Complex numbers
   bool operator ==(Complex rhs) {
      return (real == rhs.real && imag == rhs.imag) ? true: false;
   }
};

int main()
{
   // a Complex object
   Complex com1(3.0, 0.0);

   if (com1 == 3.0)
      cout << "Same";
   else
      cout << "Not Same";

   return 0;
}
```

The program compiles fine and output: Same.
If a class has a constructor which can be called with a single argument, then this constructor becomes conversion constructor, because such a constructor allows conversion of a single argument to the class.
We can avoid such implicit conversions as these may lead to unexpected results.
We can use the explicit keyword to make the constructor explicit, eg. the following program gives a compilation error.

```
#include <iostream>

using namespace std;

class Complex
{
  private:
     double real;
     double imag;

  public:
     // default constructor
     explicit Complex(double r = 0.0, double i = 0.0):
        real(r), imag(i) {}

     // A method to compare two Complex numbers
     bool operator == (Complex rhs) {
        return (real == rhs.real && imag == rhs.imag);
     }
};

int main()
{
    // a Complex object
    Complex com1(3.0, 0.0);

    if (com1 == 3.0)
       cout << "Same";
    else
       cout << "Not Same";

    return 0;
}
```

Output: Compilation Error:
no match for 'operator ==' in 'com1 == 3.0e+0'.
We can still typecast the double values to Complex, but now we have to explicitly typecast it,

```
#include <iostream>

using namespace std;

class Complex
{
private:
    double real;
    double imag;

public:
    // default constructor
    explicit Complex(double r = 0.0, double i = 0.0):
       real(r), imag(i)  {}

    // A method to compare two Complex numbers
    bool operator==(Complex rhs){
       return (real == rhs.real && imag == rhs.imag) ? true: false;
    }
};

int main()
{
   // a Complex object
   Complex com1(3.0, 0.0);

   if (com1 == (Complex) 3.0)
      cout << "Same";
   else
      cout << "Not same";

   return 0;
}
```

This program compiles fine and output: Same.

- delegating constructor

如果类有不同形式的构造函数，为了初始化成员肯定会有大量的重复代码。为了避免重复，常见的做法是把公共的部分取出，放在 init() 函数里，然后构造函数再去调用。
这种方法效率和可读性差，因为init不是真正的构造函数。
在C++11里，可以使用委托构造的新特性，一个构造函数直接调用另一个构造函数，把构造工作委托出去。

```
class DemoDelegating final
{
 private:
    int a;    // 成员变量

public:
    DemoDelegating(int x): a(x) {}  // 基本的构造函数

    DemoDelegating():         // 无参数的构造函数.
      DemoDelegating(0)       // 给出默认值, 委托给
    {}                        // 第一个构造函数.

    DemoDelegating(const string& s):   // 字符串参数构造函数
      DemoDelegating(stoi(s))          // 转换成整数, 再委托给第一个构造函数.
    {}
};
```

- 成员变量初始化 In-class member initializer.

在C++11里，可以在类里声明变量的同时赋值.

```
class DemoInit final
{
 private:
    int    a = 0;           // 整数成员, 赋值初始化
    string  s = "hello";    // 字符串成员, 赋值初始化
    vector<int>  v{1,2,3};  // 容器成员, 使用花括号的初始化列表

public:
    DemoInit() = default;  // 默认构造函数
   ~DemoInit() = default;  // 默认析构函数.

public:
    DemoInit(int x) : a(x) {}  // 可以单独初始化成员,
                               // 其他用默认值.
};
```

- Type Alias

```
   using uint_t = unsigned int;    // using 别名
   typedef unsigned int uint_t;    // 等价的 typedef.
```

写类的时候，我们经常会用到许多外部类型，比如标准库里的string, vector，还有第三方库和自定义类型，这些名字通常都很长，写起来不方便，这时我们可以用using简化。

```
class DemoClass final
{
public:
    using this_type = DemoClass;  // 给自己起个别名.
    using kafka_conf_type = KafkaConfig; // 外部类别名.

public:
    using string_type = std::string; // 字符串类型别名.
    using uint32_type = uint32_t;    // 整数类型别名.

    using set_type = std::set<int>;  // 集合类型别名.
    using vector_type = std::vector<std::string>; // 容器类型别名.

private:
   string_type m_name = "tom"; // 使用类型别名声明变量.
   uint32_type m_age = 23;
   set_type    m_books;

private:
    kafka_config_type  m_conf;
};
```

类型别名不仅能够让代码规范整齐，而且因为引入了"语法层面的宏定义"，将来在维护时还可以随意改换成其他类型。比如把字符串改成string_view (C++17里的字符串只读视图)，把集合类型改成unordered_set，只要变动别名定义就行了。原代码不需要任何改动。

Tips

- 面向对象编程要点是抽象和封装，"继承"和"多态"是衍生出的特性，不完全符合现实
- 在C++里应少用继承和虚函数，降低对象的成本。
- 使用 final 可以禁止类被继承，简化类的层次关系。
- 类有六大基本函数，对于重要的构造/析构函数，可以使用"= default"来显式要求编译器用默认实现。
- 委托构造和成员变量初始化特性可以让创建对象的工作更加轻松。
- 使用 using 或 typedef 可以为类型起别名，可以适应未来的变化。
- 传统的类编写方式是"*.h + *.cpp"，声明与实现分离，但在 .hpp里实现类的全部功能更加现代，很多开源的现代C++项目全面采用了hpp的方式，eg. Boost.
- 在类里可以多次书写public, private，粉组不同的逻辑段落，增强了可读性，借鉴了Java.