OOP Review Notes

类 class

- 构造函数
  - 名字和类名一样。
  - 无返回值类型。
  - 可以重载，常用的包括带参、无参、拷贝型。
  - 可以使用初始化列表，格式为 varName (expression)。
  - 调用格式：

```
// class xxx;
// No params required:
x = xxx();
xxx x;
xxx x[100];
xxx x = xxx();

// One or more params required:
x = xxx([params]);
xxx x([params]);
xxx x = param; // only available when 1 param is required.
```

- 析构函数
  - 名字为～ + 类名。
  - 无参数。
  - 无返回值类型。
  - 析构时调用。
- 访问修饰符
  - Public：不限制访问。
  - Protected：仅友元、子类和其成员可访问。
  - Private：仅友元、其成员可访问。
  - 无：默认为private。

- 友元
  - 包括友元函数、友元类，需要在被访问的类内声明（在哪里声明没有关系）。
  - 可以访问友元类的所有成员。
  - 破坏了类的封装性但是提高了程序的执行效率。
  - 实例如在某个类实现 “ << ” 流运算符的重载。
- 静态
  - 被类共有的成员，通过类名而非对象名访问。
  - 静态变量需要在类内声明（没有分配内存），在类外定义和初始化（分配全局内存）。
  - 静态成员函数的目的在于访问类内私有的静态变量，而如果通过非静态成员函数访问静态变量则需要先定义一个实例，影响性能；或者使用类指针对某个数据域进行强制类型转换来访问，影响安全。
- 继承
  - 对父类的成员访问权限由继承方式和原来的访问修饰符共同决定（以限制级别较高者为准）。
  - 子类的构造函数需要用初始化列表显式调用父类构造函数（如果父类没有则不用）。
  - 详见**多态部分**。
- 虚函数
  - 父类声明：**virtual** TYPE FUNCNAME (PARALIST) = 0;
  - 子类实现：**virtual** TYPE NAME (PARALIST) {……}
  - 包含纯虚函数的类为抽象类，它不能实例化，但可以有指针。
  - 可以声明内联（没有语法错误），但是编译器不会这么做。
- 大小计算
  - 空类：大小按1算。
  - static成员：存放在类外，不占空间。
  - 虚函数：vptr指向虚函数表中类内第一个虚函数的结点，指针大小为4或8（只有一个指针）。
  - 其他函数（包括构造和析构）：不占空间。
  - 字节对齐：所有大小单位按照最大的一个计算。比如int为4，有char则从1扩充到4。
  - 继承：父类大小 + 子类大小。但是如果父类为空则退化为0大小（编译器优化）。
  - 复合：按照对齐原则计算累加大小。但是如果包含空类也需要计算一个单位的大小（不是大小为1，而是按照一个size单位）。
- 运算符重载
  - 使用operator关键字
  - 不允许重载的字符：. .* :: sizeof ?: typeid
  - ++前置和后置的区别
    - 前置无参，后置要参数，用来区分。
    - 前置返回加之后的引用，以方便在表达式中连续使用。
    - 后置返回加之前的拷贝，故不能连续使用。

- ++a的重载：

```
class xxx {
  private:
    int num;
  public:
    xxx(int n): num(n) {}
    xxx& operator++() {
        num++;
        return *this;
    }
};
```

- a++的重载：

```
class xxx {
  private:
    int num;
  public:
    xxx(int n): num(n) {}
    xxx operator++(int) {
        xxx tmp = *this;
        this.num++;
        return tmp;
    }
};
```

- <<的重载

```
class xxx {
  private:
    int num;
  public:
    xxx(int n): num(n) {}
    friend ostream & operator<<(ostream &os, const xxx p);
};

ostream & operator<<(ostream &os, const xxx p) {
    os<<p.num;
    return os;
}
```

- 类型转换的重载

```
class xxx {
  private:
    int num;
  public:
    operator int() const {
        return value;
    }
}
```

    - 在cout<<xxx对象时会隐式调用，也可以直接显式调用。

- 常量
    - const成员函数
        - 用于限制函数对变量的修改。
        - 一个成员函数在加上const限定后与原来不是同一个函数，不能重载。
        - const成员函数可以引用const和非const数据成员，但是不能修改它们的值；也不能调用非const成员函数。
        - const成员函数在声明和定义时都需要加const。
        - ***特殊情况：声明mutable的成员可以在const方法内改变。***
    - const对象
        - const实例只能调用const成员函数。
        - 非const实例可以调用const或非const成员函数。
- 其他
    - this指针为类内所有的函数隐含的指针，指向实例本身。
    - const用于限制函数对变量的修改。有const的函数和没有的函数不被认为是同一个函数。const实例只能使用const方法，而非const实例优先使用非const方法（？）。
    - inline内联是编译器决定的。在定义类时实现的编译器默认内联，不过较复杂时不会内联。
    - 引用可以节省空间和提高效率，类似指针。
    - **重要区分**：重载赋值函数以及拷贝构造函数

```
// class A;

// 这是重载赋值，因为a和b都是已经存在的对象
A a, b;
a = b;

// 这是拷贝构造，因为d是未存在的对象
A c;
A d = c; // 这是正常的写法：A d(c);
```

多态 polymorphism

- override重写：子类实现父类特征完全一样的虚方法。
- overload重载：同个类的同名、不同形参的方法。如构造函数的不同重载。
- redefininng重定义：子类重定义父类同名方法。

概念	函数名	形参列表	返回值类型	是否虚	作用范围
override	相同	相同	相同	虚	子对父
overload	相同	不同	无关	无关	相同类内
redefining	相同	无关	无关	非虚	子对父

- 静态多态：重载和重定义，编译时确定。
- 动态多态：重写（虚方法），运行时确定。

模版 template

- 类模版
  - 格式：

```
template <typename / class T>
class xxx {
  public:
    void func();
}

template <typename / class T>
    void xxx<T>::func() {
        // ...definition of func...
    }

int main() {
    xxx<int> yyy;
    yyy.func();
    return 0;
}
```

- 函数模版
  - 格式：

```
template <typename / class T>
    T yyy(T a, T b) {
    // ...definition of yyy...
}
```

  - 有模版函数和正常函数时，优先调用非模版函数。

异常 exception

- 格式：

```
try {
    if(myMoney > 100)
        throw 100;
    else
        throw out_of_range;
} catch (int param) {
    // ...do something if catch an int...
} catch (...) {
    // ...do something if a default exception occurs...
}
```

- 标准异常：定义在exception或者stdexcept文件中，需要在头部包含。
- 自定义异常：需要继承exception基类定义，通过what方法访问信息。

```
# include <exception>
class myException: public exception {
  public:
    myException(): exception("Oh, I don't have enough money!") {}
}

int main() {
    int myMoney = 0;
    try {
        if(myMoney < 100)
            throw myException();
        else
            cout<<myMoney;
    } catch(myException me) {
        cout<<me.what();
    }
}
```