- No references to references.
- No pointers to references. (References to pointer is OK)
- No arrays of references.

```
vector<int> v;
vector<int>::iterator it = v.begin();
```

- Default arguments Cannot write in the def part, but the 函数声明！ part.

```
string p1("Fred");
const string* p = &p1; // p1 cannot be changed through p.
string const* p = &p1; // like the first one.
p = &p2; // p can be changed to point to p2.
string *const p = &p1; // p cannot be changed.only point to p1.
```

- 静态成员函数没有 this, 不能调用非静态成员变量，也不能访问非静态函数
- 可以在没有创建类的对象的时候就能调用静态成员函数
- 一个静态的全局变量，必须在头文件对应的.cpp中定义和初始化静态成员变量(此时不能再加static)： `int Sales_data::cnt = 0;`

```
void render(Shape* p) {
    p->render(); // calls correct render function
} // for given Shape! void func() {
Ellipse ell(10, 20);
ell.render(); // static -- Ellipse::render();
Circle circ(40);
circ.render(); // static -- Circle::render();
render(&ell); // dynamic -- Ellipse::render();
render(&circ); // dynamic -- Circle::render()
```

What happens if ?

```
Ellipse ell(10, 20);
Circle circ(40);
elly = circ;
```

- Area of circ is **sliced off**!
- Only the part of the object that is of the type of the variable is copied.
- vptr remans to be `elly` 's vptr.

```
a = b;
p = &a;
p -> f();
```

- `A::f()`
- `p->f()` will call the parent class's function.

What about?

```
Ellipse *elly = new Ellipse(20F,40F);
Circle *circ = new Circle(60F);
elly = circ;
```

- The original Ellipse for `elly` is lost.
- `elly ->render();//Circle::render()`

What about ?

```
void func(Ellipse &elly){
  elly.render()
}
Circle circ(60F);
func(circ);
```

- 一个类中有一个或多个纯虚函数，这个类就是抽象类

```
virtual void render() = 0;
```

```
class class_c{
public:
    int max;
    int min;
    int middle;
    class c(int my_max){
        max = my_max>0?my_max:10;
    }
    class_c(int my_max, int my_min):class_c(my_max){
        min = my_min>0&&my_min<max?my_min:1;
    }
    class_c(int my_max, int my_min, int my_middle):class_c(my_max, my_min){
        middle = my_middle>min&&my_middle<max?my_middle:(min+max)/2;
    }
};
int main(){
    class_c c(10, 5, 7);
}
```

Overloading

+ - * / % ^ & | ~

```
const T operator+(const T& left, const T& right);
```

! && || == != < > <= >= :

```
bool operator==(const T& left, const T& right);
```

++ and -- (prefix and postfix)

```
const T& operator++(); // prefix
const T operator++(int); // postfix
```

stream extractor/inserter

```
ostream& operator<<(ostream& os, const T& t);
istream& operator>>(istream& is, T& t);
```

```
ostream& operator <<(ostream & os, const A &a)
{
    cout << a.size() << endl;
    return os;
}
```

Assignment

```
MyClass& operator=(const MyClass& other) {
    if (this != &other) { // 检查是否自赋值
        delete data; // 释放原有内存
        data = new int(*other.data); // 深拷贝数据
    }
    return *this; // 返回引用
    }


template<class T>
T& vector<T>::operator[](int i){ //Note this <T>!!!
    return m_elements[i];
}
```

- 模板的所有东西都需要在头文件中定义，因为编译器需要看到模板的定义才能生成对应的函数。

`void abc():throw (MathErr);`

- 如果有这个声明，那么这个函数里面就不能抛出其他异常，只能抛出 MathErr 异常。抛出了其他异常，编译器会报错，终止程序。

```
Printer::print(Document&) : throw(PrinterOffLine, BadDocument) { ... }
PrintManager::print(Document&) : throw (BadDocument) { ... }// raises or doesn'
void goodguy() : throw () { }// handles all exceptions
void average() { } // no spec, no checking,
```

- 1.表示会抛 PrinterOffLine, BadDocument 异常。（不一定抛，但可能）
- 3.表示不会抛任何异常，这样调用的时候不需要 try catch.
- 4.可能会抛异常，但是编译器不会进行检查。

Note:

- a. in C++ , exception is not a routine and not for good design.
- b. Exception in constructor : 先分配内存，再执行构造

```
f() {
    A *p = new A();
    ...
    delete p;
}
```

- 如果构造的时候出异常, p 无法得到分配的地址，但是内存却没有被析构。内存泄漏!
- Solve: 1. Never New? 2. Catch error and `delete this` :必须是一个局部对象，不能是一个全局对象。
  - 如果是 new error类,需要记得同时delete error类. `c++ Error *p = new Error(); ... catch (Error e) { delete e; delete this; }`
  - 如果是全局的error类，等堆栈清空的时候，会自动析构。 3. Altimate: 两阶段构造:构造函数不做任何事情(打开网络，读取文件等)，只是分配内存，然后用explicit的init函数（第二阶段构：需要主动调用）来初始化
- 也就是说构造函数不允许抛异常，只有init函数可以抛异常。

Complement : stream

```
int get()
while((ch=cin.get())!=EOF) {
    cout.put(ch);
}
```

```
class Error {};

class MyClass {
public:
    MyClass() {
        // 在构造函数中可能会创建 Error 对象
        try {
            Error* p = new Error(); // 这将会抛出异常
        }
        catch (const Error& e) { // 通过引用捕获异常
            // 处理异常
            std::cout << "Caught Error exception" << std::endl;
            delete this; // 删除当前对象
        }
    }
};
```