directory

review

- 构造函数、析构函数
- 全局和局部对象的构造与析构时机
- 引用
- 运算符重载：
    - 流运算符(<<,>>)
    - 函数运算符()
    - 前缀运算符(++,--)
    - 下标运算符[]

友元

- 被声明为友元的函数或类，具有对出现友元声明的类的private及protected成员的访问权限，即可以访问该类的一切成员。
- 友元的声明只能在类内进行
- 有时需要允许某些函数访问对象的私有成员，可以通过声明该函数为类的“友元”来实现

```
#include<iostream>
using namespace std;
class A {
    private:
        int data = 1;
        friend void foo(A &a);
        //函数foo 是 类A的朋友
        //函数foo 不是 类A的成员函数
        //在foo内可以访问A的私有成员和保护成员

};
void foo(A &a) {
    cout << a.data;
}
int main() {
    A a;
    foo(a);
    return 0;
}
```

```
#include <iostream>
using namespace std;

class Test {
    int id;
public:
    Test(int i) : id(i) { cout << "obj_" << id << " created\n"; }

    friend istream& operator>> (istream& in, Test& dst);
    friend ostream& operator<< (ostream& out, const Test& src);
};
istream& operator>> (istream& in, Test& dst) {
    in >> dst.id;
    return in;
}
ostream& operator<< (ostream& out, const Test& src) {
    out << src.id;
    return out;
}   // 以上类中声明了Test类的两个友元函数 —— 全局流运算符重载函数，
    // 使这两个函数在实现时可以访问对象的私有成员（如int id）.
int main() {
    Test test(0);
    cin >> test;
    cout << test << endl;
}
```

- 被友元声明的函数一定不是当前类的成员函数，即使该函数的定义写在当前类内
- 当前类的成员函数也不需要友元修饰
- 可以声明别的类的成员函数，为当前类的友元
    - 其中，构造函数、析构函数也可以是友元。
    - 则在X的成员函数内可直接访问/修改Y的私有成员

```
class Y {
    int data;
    friend void X::foo(Y);
    friend X::X(Y), X::~X();
};
```

- 友元的声明与当前所在域是否为private或public无关
- 一个普通函数可以是多个类的友元函数

```
#include<iostream>
using namespace std;
class Y;
class X {
  private:
      int data;
  public:
```

```
      X(int i) : data(i) {}
      friend void show(X &x, Y &y);

};
class Y {
  private:
    int data;
    friend void show(X &x, Y &y);
  public:
    Y(int i) : data(i) {}
};

void show(X &x, Y &y) { //全局函数，可以访问X，Y的私有数据
  cout << x.data << " " << y.data << endl;
}
int main() {
  X x(1);
  Y y(2);
  show(x,y);
}
```

友元类

- 可对class/struct/union进行友元声明，代表该类的所有成员函数均为友元函数
- 对基础类型的友元声明会被忽略（因为没有实际价值）。编译器可能会发出警告，但不会认为是错误

```
class Y {}; // 定义类Y，且Y能访问A的所有成员
class A {
    int data;  // 私有数据成员
    enum { a = 100 };  // 私有枚举项
    friend class X;  // 友元类前置声明（详细类型指定符）
    friend Y;  // 友元类声明（简单类型指定符）（C++11起）
};
class X {}; // 定义类X，X能访问A的所有成员
```

- 注意:
  - **非对称关系**：类A中声明B是A的友元类，则B可以访问A的私有成员，但A不能访问B的私有成员。
  - **友元不传递**: 朋友的朋友不是你的朋友
  - **友元不继承**（继承为后续内容）: 朋友的孩子不是你的朋友
  - 友元声明不能定义新的class，如

```
class X
{
    friend class Y {}; □
}
```

5.2 静态成员与常量成员

回顾：C中的静态变量/函数

- 静态变量：使用static修饰的变量
  - 定义示例：static int i = 1;
  - 初始化：初次定义时需要初始化，且只能初始化一次。
  - 静态局部变量存储在静态存储区，生命周期将持续到整个程序结束
  - 静态全局变量是内部可链接的，作用域仅限其声明的文件，**不能被其他文件所用**，可以避免和其他文件中的同名变量冲突
- 静态函数：使用static修饰的函数
  - 定义示例：static int func() {...}
  - 静态函数是**内部可链接的**，作用域仅限其声明的文件，不能被其他文件所用，可以避免和其他文件中的同名函数冲突
- 区别：静态全局变量/静态函数和非静态全局变量/非静态全局函数
  - 静态全局变量/静态函数是内部可链接的，作用域仅限其声明的文件，不能被其他文件所用
  - 非静态全局变量/非静态全局函数是**外部可链接**的，可以被其他文件所用

静态数据成员

- 静态数据成员：使用static修饰的数据成员，是隶属于类的，称为类的静态数据成员，也称"类变量

  - 静态数据成员被该类的所有对象共享（即所有对象中的这个数据域处在同一内存位置）
  - 类的静态成员（数据、函数）既可以通过对象来访问，也可以通过类名来访问，如ClassName::static_var或者a.static_var（a为ClassName类的对象）
  - 类的静态数据成员要在实现文件中赋初值，格式为： Type ClassName::static_var = Value;
  - 和全局变量一样，**类的静态数据成员在程序开始前初始化**

- 静态数据成员的多文件编译

  - 静态数据成员应该在.h文件中声明，在.cpp文件中定义。
  - 如果在.h文件中同时完成声明和定义，会出现问题。
    - 包含了该头文件的所有源文件中都定义了这些静态成员变量，即该头文件被包含了多少次，这些变量就定义了多少次。
    - 同一个变量被定义多次，会导致链接无法进行，程序编译失败。

静态成员函数

- 静态成员函数：在返回值前面添加static修饰的成员函数，称为类的静态成员函数
  - 和静态数据成员类似，类的静态成员函数既可以通过对象来访问，也可以通过类名来访问，如`ClassName::static_function`或者`a.static_function`(a为ClassName类的对象)
- 静态成员函数不能访问非静态成员
  - 静态成员函数属于整个类，在类实例化对象之前已经分配了内存空间。
  - 类的非静态成员必须在类实例化对象后才分配内存空间。
  - 如果使用静态成员函数访问非静态成员，相当于没有定义一个变量却要使用它。

```
// 错误
    #include <iostream>
    using namespace std;

class A {
```

```
    int data;
public:
    static void output() {
        cout << data << endl; // 编译错误
    }
};
int main()
{
    A a;
    a.output();
    return 0;
}
```

```
#include <iostream>
using namespace std;

class Test {
public:
  static int count; //声明静态数据成员
  float value;
  Test(int v): value(v) { count ++; }
  ~Test() { count --; }
  static int how_many() { return count; }
  //静态成员函数how_many仅能访问count，无法访问value
};

int Test::count = 0;//定义静态数据成员

int main() {
  Test t1(2);
  cout << “Test#: ” << Test::how_many() << endl;
  cout << “Test.value: ” << t1.value << endl;
  return 0;
}
```

- 注：在静态成员函数中，不能使用this指针来引用对象的成员变量或成员函数
  - 静态成员函数是与类关联的函数，它们不属于任何一个对象，因此不存在this指针
  - 在静态成员函数中，只能使用类名和作用域解析符（::）来访问静态成员变量或函数，而不能使用this指针

```
class A {
public:
    static int s_value; // 静态成员变量

    static void s_func() { // 静态成员函数
        s_value = 10; // 可以访问静态成员变量
        // this->value = 10; // 错误，不能使用this指针
    }
};
```

```
int A::s_value = 0; // 静态成员变量定义和初始化

int main() {
    A::s_func(); // 通过类名调用静态成员函数
    return 0;
}
```

回顾const

- 常量关键字const常用于修饰变量、引用/指针、函数返回值等
    - 修饰变量时（如const int n = 1;），必须就地初始化，该变量的值在其生命周期内都不会发生变化
    - 修饰引用/指针时（如int a=1; const int& b=a;），不能通过该引用/指针修改相应变量的值，常用于函数参数以保证函数体中无法修改参数的值
    - 修饰函数返回值时（如const int* func() {...}），函数返回值的内容（或其指向的内容）不能被修改
- 常量数据成员：使用const修饰的数据成员，称为类的常量数据成员，在对象的整个生命周期里不可更改
- 常量数据成员可以在
    - 构造函数的初始化列表中被初始化
    - 就地初始化
    - 不允许在构造函数的函数体中通过赋值来设置

```
#include <iostream>
using namespace std;

class Test {
  const int ID; //常量数据成员
  const int age = 9; // 就地初始化
public:
  Test(int id) : ID(id) {} // 通过初始化列表初始化常量数据成员
  int Next() {
    ID++; //该处会出现编译错误，因为常量数据成员不可更改
    return ID;
  }
};

int main() {
  Test obj1(20151145);
  cout << "ID = " << obj1.Next() << endl;
  return 0;
}
```

常量成员函数

- 常量成员函数：成员函数也能用const来修饰，称为常量成员函数。
- 常量成员函数的访问权限：实现语句不能修改类的数据成员，即不能改变对象状态（内容）
    - ReturnType Func(...) const {...}
    - 注意区别：const ReturnType Func(...) {...}

- 若对象被定义为常量(const ClassName a;)，则它只**能调用以const修饰的成员函数**
    - 常量对象：对象中的"数据"不能变

```
#include <iostream>
using namespace std;

class Test {
  int ID;
public:
  Test(int id) : ID(id) {}
  int MyID() const { return ID; } //常量成员函数
  // int Next() const { ID++; return ID; } 编译错误，常量成员函数不能修改数据成员
  int Next()  { ID++; return ID; } // 编译通过
  // int Who() { return ID; } 编译错误，常量对象不能调用非常量成员函数
  int Who() const { return ID; }
};

int main()
{
  Test obj1(20151145);
  cout << "ID_1 = " << obj1.MyID() << endl;
  cout << "ID_2 = " << obj1.Who() << endl;

  const Test obj2(20160301);
  cout << "id_1 : " << obj2.MyID() << endl;
  cout << "id_2 : " << obj2.Who() << endl;
  return 0;
}
```

常量静态变量

- 当然，我们可以定义既是常量也是静态的变量
    - 作为类的常量变量
- 常量静态变量需要在类外进行定义
    - 和静态变量一样
    - 但有两个例外：int和enum类型可以就地初始化
- 常量静态变量和静态变量一样，满足访问权限的任意函数均可访问，但都不能修改
- **注意**：不存在常量静态函数
    - 静态函数隶属于类，可以不实例化而直接通过类名访问
    - 常量/非常量函数的访问权限需要通过实例化后的对象是否为常量对象来决定。常量修饰函数必须绑定在对象上
    - 因此，静态函数和常量函数互相冲突

```
#include<iostream>
using namespace std;
class foo {
    static const char* cs; // 不可就地初始化
    static const int i = 3; // 可以就地初始化
```

```
    static const int j; // 也可以在类外定义
public:
    const char* return_cs(foo &f) { return f.cs;}
    int return_i(foo &f) { return f.i;}
    int return_j(foo &f) { return f.j;}
};

const char* foo::cs = "foo C string";
const int foo::j = 4;

int main() {
    foo f;
    cout << f.return_cs(f) << endl;
    cout << f.return_i(f) << endl;
    cout << f.return_j(f) << endl;
}
```

5.3 常量/静态/参数对象的构造与析构时机

- 常量全局/局部对象的构造与析构时机和普通全局/局部对象相同
    - **常量全局对象**：在main()函数调用之前进行初始化，在main()函数执行完return，程序结束时，对象被析构
    - **常量局部对象**：在程序执行到该局部对象的代码时被初始化。在局部对象生命周期结束、即所在作用域结束后被析构
- 静态全局对象的构造与析构时机和普通全局对象相同
- 函数中静态对象：在函数内部定义的静态局部对象
    - 在程序执行到该静态局部对象的代码时被初始化。
    - 离开作用域不析构。
    - 第二次执行到该对象代码时，不再初始化，直接使用上一次的对象
    - 在main()函数结束后被析构。

```
void fun(int i, int n) {
    if (i >= n)
        static A static_obj("static");
}
```

静态对象的构造与析构

- 类静态对象：类A的对象a作为类B的静态变量
    - a的构造与析构表现和全局对象类似，即在main()函数调用之前进行初始化，在main()函数执行完return，程序结束时，对象被析构
    - 和B是否实例化无关

```
class A {};
```

```
class B {
    static A a; //
};
```

```
#include <iostream>
using namespace std;

class A {
    const char* s;
public:
    A(const char* str):s(str) {
        cout << s << " A constructing" << endl;
    }
    ~A() {
        cout << s << " A destructing" << endl;
    }
};

class B {
    static A a1;
    const A a2;
public:
    B(const char* str):a2(str) { }
    ~B() { }
};

void fun() {
    static A static_obj("static");
}

const A c_a("const c_a");
static A s_a("static s_a");
A B::a1("static B::a1");
int main() {
    cout << "main starts" << endl;
    static B main_b("static main_b");
    for (int i = 0; i < 4; i++) {
        fun();
    }
    cout << "main ends" << endl;
    return 0;
}
/*运行结果:
const c_a A constructing
static s_a A constructing
static B::a1 A constructing
main starts
static main_b A constructing
static A constructing
main ends
static A destructing
```

```
static main_b A destructing
static B::a1 A destructing
static s_a A destructing
const c_a A destructing
*/
```

如果传递的是形参

```
void fun(A b) {
    cout << "In fun: b.s=" << b.s << endl;
}
fun(a);
```

- 在函数被调用时，b被构造，调用拷贝构造函数（以后内容）进行初始化。默认情况下，对象b的属性值和a一致。
- 在函数结束时，调用析构函数，b被析构。

```
#include <iostream>
using namespace std;

class A {
public:
    const char* s;
    A(const char* str):s(str) {
        cout << s << " A constructing" << endl;
    }
    ~A() { cout << s << " A destructing" << endl; }
};

void fun(A b) {
    cout << "In fun: b.s=" << b.s << endl;
}

int main() {
    A a("a");
    fun(a);
    return 0;
}
/*
运行结果:
a A constructing
In fun: b.s=a
a A destructing
a A destructing
*/
```

- 构造一次，析构两次.

如果参数是类对象的引用

```
void fun(A &b) {
    cout << "In fun: b.s=" << b.s << endl;
}
fun(a);
/*
运行结果:
a A constructing
In fun: b.s=a
a A destructing
*/
```

- 在函数被调用时，b不需要初始化，因为b是a的引用。
- 在函数结束时，也不需要调用析构函数，因为b只是一个引用，而不是A的对象。

如果一个类含有指针成员?

```
#include <iostream>
using namespace std;

class A {
public:
    int *data; // 注意这是一个指针
    A(int d) {data = new int(d);}
    ~A() {delete data;} // 注意这里，释放之前申请的内存
};

void fun(A a) {
    cout << *(a.data) << endl;
}

int main() {
    A object_a(3);
    fun(object_a);
    return 0; // 在程序结束时会出错
}
```

- 对象a和对象object_a的data成员一样（地址一样），所以delete的时候释放的是同一块内存地址。
- 对象a析构时不会出错。但对象object_a析构时，因为试图释放一块已经释放过的内存，所以会出错。
    - 析构两次，两个对象的指针是同一个内存地址，但是被删除两次
- 尽量使用对象引用作为参数，这样做还可以减少时间开销

```
void fun(A &a) {
    cout << *(a.data) << endl;
```

```
}
```

对象的new和delete

- new
  - 生成一个类对象，并返回**地址**（构造函数会被调用）
  - `A *pA = new A(some parameters);`
- delete
  - 删除该类对象，释放内存资源（析构函数会被调用）
  - `delete pA;`
- e.g.生成一个类对象的数组 `A *pA = new A[3];`
  - ☐ 标准库函数来分配足够大的原始未类型化的内存。
    - 注意要多出4个字节来存放数组的大小。
  - ②在刚分配的内存上运行构造函数对新建的对象进行初始化构造。
  - ③返回指向新分配并构造好的对象数组的指针。
- e.g.删除该对象数组及数组中的每个元素（释放内存资源）
  - ①对数组中各个对象运行析构函数，数组的维数保存在pA前4个字节里。
  - ②调用operator delete[ ]标准库函数释放申请的空间。 不仅仅释放对象数组所占的空间，还有上面的4个字节。
- new和delete要配套使用
  - new 和 delete
  - new[] 和 delete[]
  - 如果同时使用new[]和delete
    - delete命令做了两件事：
      - 调用一次 pA 指向的对象的析构函数。
      - 释放pA地址的内存。
    - 后果如下：
      - 只调用一次析构函数。如果类对象中有大量申请内存的操作，那么因为没有调用析构函数，这些内存无法被释放，造成内存泄漏。
      - 直接释放pA指向的内存空间，这个会造成严重的段错误，程序必然会崩溃。因为分配空间的起始地址是pA-4byte。（delete[] pA的释放地址自动转换为pA-4byte）
- 例题：

```
假定A是一个类的名字，下面四个语句总共会引发类A构造函数的调用多少次
1. A *p = new A; // 调用一次构造函数，创建一个A类型的对象，并将指针p指向该对象
2. A p2[10]; // 创建一个A类型的数组，调用10次构造函数，创建10个A类型的对象
3. A p3; // 调用一次构造函数，创建一个A类型的对象
4. A *p4[10];  // 创建一个A类型的指针数组，不会调用构造函数
// 数组中的每个元素都是指针类型，它们只是指向某些内存位置的地址，并不会创建任何对象
```