指针

类和对象

类的定义

访问权限

- private
  - 私有，无法被对象访问
  - 需要通过public方法操作
- public
- protected

数据成员

this指针

- this 指针**指向当前对象的地址本身**
- 只有**非静态成员函数**有隐式 this 指针
- 参数名与成员变量名冲突时需要显式使用 this

- const 成员函数：this 是 `const MyClass*`
- `*this` 是“当前对象本身”的引用
  - `*this` 的类型是 `MyClass&`
  - 是当前对象的**引用**，不是副本
  - 智能指针也是。

成员函数

> “
> 隐式传递 this 指针
>
> - 成员函数 `void func(int x)` 在编译器被处理为 `void func(MyClass* this, int x)`
>
> 先构造 后析构

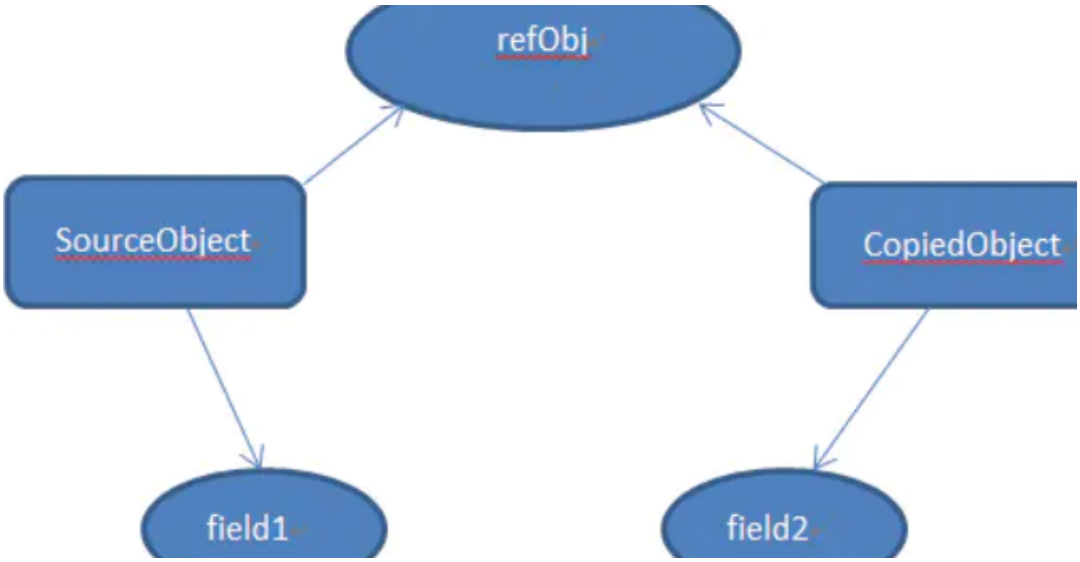
构造函数

- `Date(){// ...}`
- `Date(int a, int b) : x(a), y(b) {// ...}`
  - 最好用这个
  - **初始化列表**内进行**初始化**
  - 代码块内执行其他逻辑/初始化代码(**赋值**)
- 构造函数的参数
  - 缺省
    - 从左往右给出
  - 默认

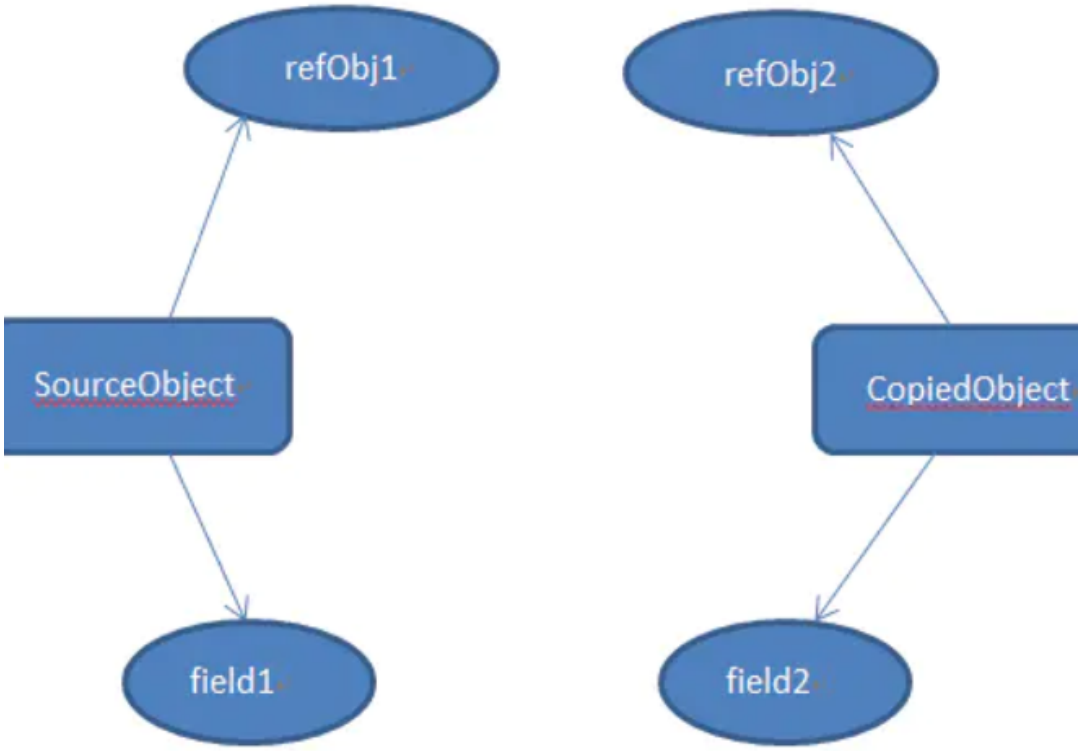
- 析构函数

- `~Date(){// ...}`

- 拷贝构造

- 用已经存在的对象初始化新对象
  - `Date d2 = d1;`
- `Date(const Date &t){// ...}`
- **浅拷贝**
  - `int b = a;`
  - `Base obj2 = obj1;`
  - **简单赋值**，只复制指向某个对象的指针，而不复制对象本身，新旧对象还是共享同一块内存
  - 如果属性是基本类型，拷贝的就是基本类型的值；
  - **如果属性是内存地址（引用类型），拷贝的就是内存地址**
  - 

- **深拷贝**
  - 会另外创造一个一模一样的对象，新对象跟原对象不共享内存

- **动态分配的内存、指向其他数据的指针**等，必须**显式地定义拷贝构造函数**，以完整地拷贝对象的所有数据。
- 


```
Array::Array(const Array &arr){  //拷贝构造函数
    this->m_len = arr.m_len;
    this->m_p = (int*)calloc( this->m_len, sizeof(int) );
    memcpy( this->m_p, arr.m_p, m_len * sizeof(int) );
}
Array a();
//  ...
Array b(a);
```

- 普通函数

- 成员函数

对象/实例

- 实例
- 实例数组
- 实例指针
  - `Date *p = &d;`
  - `p->func()`
  - `(*p).func()`
- 引用
  - `Date &r = d;`
  - 别名
- 动态分配
  - `Date *p = new Date()`
  - `delete p`

对象成员

- 一个类的对象作为一个类的成员

静态成员

> “
> 被该**类的所有对象共享**，小范围全局变量
>
> 编译时初始化

- 静态数据变量
  - `static int cnt;`
- 静态成员函数
  - 可以通过类名访问
  - 静态函数只能访问静态成员
- 类外初始化
  - `int Date::cnt = 0`

常量

- 常成员函数
  - `returnType func() const {// ...}`
- 常数据成员
- 常对象
  - `const Date d;`
  - **只能调用常成员函数**

友元

> “
> 访问私有

友元函数

- 在类内声明为友元的全局函数
- 声明后，此函数可以访问类的私有成员

```
class A{friend ostream& operator<<(ostream& q, const A& p);}
// 重载运算符的全局函数可以访问A类内的私有成员
```

友元类

- 在**类内声明为友元的类**
- 声明后，友元类的成员函数可以访问类的私有成员

```
class A{friend class B};
class B{};
// 其中B为友元类，B的成员函数可以访问A
```

友元成员函数

- 在类内声明为友元的**其他类成员函数**
- 声明后，此成员函数可以访问类的私有成员

继承、派生

> 实现代码复用，子类有父类的属性方法

单继承

```
class Stud : ctrlWay Person {}
```

构造 && 析构

- `class Child : public Parent {Child(int a, int b) : Parent(a) {// ...}}`
  - 子类的构造函数**需要先初始化父类部分**,必须在初始化列表部分调用父类构造函数
  - 如果父类有默认构造函数，子类可以省略显式调用
- 父类构造 -> 自己构造 -> 自己析构 -> 父类析构

继承方式

- public
- private
- protected

- **访问权限**

  - 基类中私有变量不可以被子类访问
  - 基类权限和继承方式决定子类对父类的访问权限
  - 取最低即可

- **成员函数**

  - 访问基类成员函数需要指定作用域
  - ?

多继承

`class Stu : public Person, public Oth {};`

虚继承

> 菱形继承

```
class A{};
class B : virtual public A{};
class C : virtual public A{};
class D : public B, public C{};
```

虚函数、多态

虚函数

> 实现多态，动态绑定

- `virtual void func() {}`
- 虚函数最好为虚构实现
  - 确保通过基类指针删除派生类对象时正确调用派生类的析构函数。
- ~~虚函数表~~
  - 虚函数表（vtable）是编译器为每个包含虚函数的类生成的一个表，存储该类所有虚函数的函数指针。
  - 如果派生类重写了基类的虚函数，vtable 中的对应条目会指向派生类的实现。
  - 每个包含虚函数的对象内部都会有一个隐藏的指针，**称为虚表指针（vptr）**，指向该对象**所属类的 vtable**。
  - 当通过基类指针或引用调用虚函数时，程序使用 this 指针访问对象的 vptr，vptr 指向类的 vtable，**vtable 中存储的函数指针**决定**调用哪个虚函数**实现。

多态

- 虚函数

- 赋值兼容性，子集
  - 派生类对象可以赋值给基类对象、基类指针或基类引用
  - is-a 关系
- **指针或引用以调用虚函数**
  - 运行时根据对象的实际类型（而非指针/引用的类型）决定调用哪个版本的函数。
- 虚函数需要通过指针或引用调用才能实现动态绑定。如果直接用对象调用（如 Base obj; obj.func();），则调用的是静态绑定
  - `Base* ptr = new Derived();  ptr->func();`

纯虚函数、抽象类

- `virtual void func() = 0`
  - 含有这句话即为抽象类
- 纯虚函数可以有实现，派生类仍需显式重写，但可以通过作用域解析
  - ```
    class Base {
    public:
        virtual void func() = 0;
    };
    void Base::func() { // 纯虚函数的实现
        std::cout << "Base::func()" << std::endl;
    }
    class Derived : public Base {
    public:
        void func() override {
            Base::func(); // 调用基类的实现
            std::cout << "Derived::func()" << std::endl;
    ```
    ```

```
    }
};
```

- 抽象类不可以创建实例
- 继承抽象类的派生类要重写(@Override)抽象类的所有接口，否则仍是抽象类类
    - `void func() override {}`

运算符重载

函数重载

在同一个作用域内，可以声明几个功能类似的同名函数，但是这些同名函数的形式参数（指参数的个数、类型或者顺序）必须不同。

```
class printData
{
   public:
      void print(int i) {
        cout << "整数为: " << i << endl;
      }
      void print(double  f) {
        cout << "浮点数为: " << f << endl;
      }
      void print(char c[]) {
        cout << "字符串为: " << c << endl;
      }
};
```

不能重载的运算符

- 成员访问运算符 `.`
- 域操作运算符 `::`
- 空间运算符 `sizeof`
- 三目条件运算符 `?:`
- 成员指针运算符 `*`

隐式转换

成员函数重载

> 隐式调用对象， 参数列表只需包含其他操作数
>
> 左侧操作数必须是当前类的对象，**无法支持非类类型**

```
class Student{
    // 比较大小
    bool operator> (const Student& s) {
        if(score > s.s.score) {
            return true;
        }else {
            return false;
        }
    }
    // 实现++Score 返回引用以链式调用
```

```
    Student& operator++ () {
        score++;
        return *this;
    }
    // 实现Score++ 不支持链式调用
    Student operator++ (int) {
        Student tmp = *this;
        score++;
        return tmp;
    }

    friend ostream& operator<< (ostream& qq, const Student&
s);

    Student& operator= (const Student& t) {
        x = t.x;
        y = t.y;
    }
};
```

```
class A{
    int a;
    public:
    A():a(0){} // 无参构造
    A(int n):a(n){} // 有参构造
    A operator+(const A& obj){
        return a + obj.a;
        // 注意这里 完成加法后隐式调用A(int)
    }
    /*
    A operator+(const A& obj) {
        return A(a + obj.a);  // 显式构造更易懂
    }
    */
    A operator+(const int b){
        return A(a+b);
```

```
    }
    friend A operator+(const int b, A obj);
};
A operator+(const int b, A obj){
    return obj + b;
    // 调用成员函数A operator+(const int b)
}
```

友元函数重载

> 把运算符重载函数声明为类的**友元**函数，不用创建对象而直接调用函数。
>
> **需要显式传递所有操作数**，代表左侧和右侧操作数
>
> 可以支持任意类型的左侧操作数（如`int + MyClass`），只需定义相应的友元函数

```
class Distance
{
   private:
      int feet;
      int inches;
   public:
      friend ostream &operator<<( ostream &output, const
Distance &D ){
         output << "F : " << D.feet << " I : " << D.inches;
         return output;
      }
      friend istream &operator>>( istream  &input, Distance
&D ){
```

```
        input >> D.feet >> D.inches;
        return input;
    }
};
```

模板

函数模板

```
template <typename T>
const T& Max(const T& a, const T& b){return a < b ? b:a; }
```

类模板

```
template <class T>
class Stack {
  private:
    vector<T> elems;     // 元素
  public:
    void push(T const&);  // 入栈
    void pop();               // 出栈
    T top() const;            // 返回栈顶元素
    bool empty() const{       // 如果为空则返回真。
        return elems.empty();
    }
```

```
};
template <class T>
void Stack<T>::push (T const& elem)
{
    // 追加传入元素的副本
    elems.push_back(elem);
}
template <class T>
void Stack<T>::pop ()
{
    if (elems.empty()) {
        throw out_of_range("Stack<>::pop(): empty stack");
    }
    // 删除最后一个元素
    elems.pop_back();
}
template <class T>
T Stack<T>::top () const
{
    if (elems.empty()) {
        throw out_of_range("Stack<>::top(): empty stack");
    }
    // 返回最后一个元素的副本
    return elems.back();
}
```

IO

格式控制

文件操作

- `ifstream`
- `ofstream`