逻辑与或非操作

- 与：&&
- 或：||
- 非：！

函数调用过程、形参和实参、传值方式

- 函数调用：

  ```
  <函数名>(<实参>);
  ```

  - 函数调用期间，形参和实参各自拥有独立的存储单元
- 形参和实参：
  - 形参：函数中定义的参数
  - 实参：传递时实际的参数
    - 形式：
      - 形参为简单类型变量，对应的实参可以是：常量， 变量及表达式
      - 形参为数组，对应的实参为数组（名）
      - 形参为结构类型，对应的实参为结构类型变量
- 传值方式
  - 值传递：
    - 形式：形参为普通变量，实参为表达式，实参向形参赋值
    - 特点：参数传递后，实参和形参不再有任何联系
    - 注意：实参是表达式，故形参不可能给实参赋值
    - e.g.

      ```
      void swap(int x, int y) {
              int t = x;
              x = y;
              y = t;
      }
      ```

  - 引用传递（常用）：
    - 形式：形参为引用型变量，实参为变量，实参为引用型形参初始化
    - 特点：参数传递后，形参是实参的别名，彼此关联
    - e.g.

```
void swap(int& x, int& y) {
        int t = x;
        x = y;
        y = t;
}
```

- 指针传递：
  - 形式：形参为指针变量，实参为指针表达式。
  - 特点：参数传递后，形参可读写实参所指的存储空间
  - e.g.

```
void swap(int* x, int* y) {
        int t = *x;
        *x = *y;
        *y = t;
}
```

引用含义、声明方式与使用

函数重载以及影响重载的因素

- 重载：两个或两个以上的函数同名，但形参的类型或形参的个数有所不同
- 仅返回值不同，不能定义为重载函数
- 注意：不能以形参名字或函数返回类型的不同来区分函数

有参宏和无参宏的定义与使用

- 无参宏：
  - `#define PI 3.1415`
  - 只进行字符串代替，不做任何计算，没有分号
  - `#undef PI`：终止宏名作用域
- 有参宏：
  - `#define ADD(a, b) (a)+(b)`
  - 参数列表中不写数据类型，也可看作字符串直接替换
  - 宏名和参数列表之间不能有空格
  - 可以使用\进行换行

指针与二维数组之间关系以及指针运算

new和delete的使用

构造函数和析构函数

- 构造函数：
  - `className(<形参>){}`
  - 与类名同名
  - 无返回值
  - 创造对象时自动调用
  - 可重载
  - 初始化方法： `className(<形参>):<属性>(<值>),<属性>(<值>)...{}`
  - 构造函数调用时，先初始化，再执行函数体
  - const数据成员、const对象成员和从基类继承的数据成员的初始化必须用初始化表
  - 所有参数都带默认值的构造函数也是缺省的构造函数即无参构造
- 析构函数：
  - `~className(){}`
  - 没有参数，没有返回值
  - 无法重载

友元函数与友元类的声明

- 使用友元可在对象外部直接访问对象私有的和保护的数据成员
- 友元函数：
  - `friend <type> FuncName(<args>);`
  - 友元函数不是类的成员，不带this指针，因此必须将对象名或对象的引用作为友元函数的参数，并在函数体中用运算符“.”来访问对象的成员
  - 一个类的成员函数也可作为另一个类的友元函数
- 友元类：
  - `friend class className;`
  - 若声明A类是B类的友元，则A类的所有成员函数都成为B类的友元函数
- 注意事项：
  - 友元关系没有传递性
  - 友元关系不具有交换性，即说明类A为类B的友元时，类B并不一定是类A的友元
  - 友元关系没有继承性

运算符重载为友元和成员函数以及形参数量差异

- 远算符重载： `<type> operator@(<args>){}` （@代表重载的运算符）
- 重载为成员函数（优先）
  - 参数个数为运算符目数减一
  - @函数应说明为公有的，否则外界无法使用
  - 当@函数为类成员时，其第一操作数为当前对象
  - 成员运算符函数的定义方法与普通成员函数相同
  - =、()、[]只能重载为成员函数
- 重载为友元函数
  - `friend <type> operator@(<args>);`
  - 参数个数为运算符目数
  - 无this指针，参数列表中需要列出所有操作数
  - <<、>>只能重载为友元函数

公有继承和私有继承对成员访问控制属性的影响

- 继承：is-a关系，即派生类对象也是一个基类对象
- `class ClassName:<Access> BaseClassName1, <Access> BaseClassName2...`
- 公有继承：public
  - 基类成员在公有派生类中保持原有访问权限
  - 公有派生类内可以访问基类的public和protected部分，类外只能访问基类的public部分
- 私有继承：private
  - 基类中公有成员和保护成员在私有派生类中均变为私有的，在派生类中仍可直接访问，但在派生类之外均不可直接访问
  - 基类中的私有成员在私有派生类中不可直接访问，当然在派生类之外，更不直接访问
     Untitled

抽象类的定义、形成原因、功能

- 抽象类：不能定义对象而只能作为基类派生新类的类
- 举例：
  - 构造函数的访问权限为protected的类
  - 析构函数的访问权限为protected的类
  - 含有纯虚函数的类

冲突和支配形成原因及解决方案

- 冲突：在多重继承的派生类中使用不同基类中的同名成员时出现的二义性

- 冲突的解决：
  - 使各基类中的成员名各不相同
  - –将基类的成员数据的访问权限说明为private，并在相应的基类中提供成员函数访问这些成员数据，但并不实用。原因是，在实际编程时，通常将基类成员数据的访问权限定义为protected，以保障类的封装性和便于派生类访问
  - 用类名限定来指明所访问的成员。格式为：类名::成员名（作用域运算符不能连续使用）
- 支配规则：未用类名限定时，派生类定义的成员优先于基类中的同名成员，并不产生冲突
- C++规定：任一基类在派生类中只能直接继承一次
- 赋值兼容规则：规定派生类的对象与其基类的对象之间互相赋值的规则：
  - 派生类的对象可赋给基类的对象
  - 不允许将基类的对象赋给派生类对象
  - 可将派生类对象的指针赋给基类型的指针变量
  - 派生类对象可以初始化基类型的引用
  - 使用基类的指针或引用时，只能访问从相应基类中继承来的成员，而不允许访问其他基类的成员或在派生类中增加的成员

虚函数和纯虚函数的定义和使用

- 虚基类：
  - `class ClassName:virtual <access> ClassName1{};`
  - 在定义派生类时，在继承的公共基类的类名前加关键字virtual，使得公共基类在派生类中只有一份拷贝
  - 注意：用虚基类进行多重派生时，若虚基类没有缺省的构造函数，则在每一个派生类的构造函数的初始化成员列表中都应有对虚基类构造函数的调用
- 多态性：
  - 编译时的多态性：通过函数重载或运算符重载实现。重载的函数根据调用时给出的实参类型或个数，在程序编译时就可确定调用哪个函数
  - 运行时的多态性：在程序执行前，根据函数名和参数无法确定应该调用哪个函数，必须在程序的执行过程中，根据具体的执行情况来动态确定。它通过类的继承关系和虚函数来实现，主要用来建立实用的类层次体系结构、设计通用程序
- 虚函数：
  - `virtual <type> FuncName(<ArgList>);`
  - 特性：
    - 继承性。若某类有某个虚函数，则在它的派生类中，该虚函数均保持虚函数特性。
    - 可重定义。若某类有某个虚函数，则在它的派生类中还可重定义该虚函数，此时不用virtual修饰，仍保持虚函数特性，但为了提高程序的可读性，通常再用virtual修饰。应强调，在派生类中重定义虚函数时，必须与基类的同名虚函数的参数个数、参数类型及返回值类型完全一致，否则属重载。

    - 通常把析构函数定义为虚函数以便通过运行时多态性，正确释放基类及其派生类申请的动态内存，一般来说，一个类如果定义了虚函数，则应该将析构函数也定义成虚函数。或者，一个类打算作为基类使用，也应该将析构函数定义成虚函数
    - 使用基类类型的指针变量(或基类类型的引用)，使该指针指向派生类的对象(或该引用是派生类的对象的别名)，并通过指针(或引用)调用指针(或引用)所指对象的虚函数才能实现运行时的多态性
    - 若派生类中没有重定义基类的虚函数时，当调用这种派生类对象的虚函数时，则调用其基类中的虚函数
- 虚函数表——多态实现的关键
    - 每一个有虚函数的类（或有虚函数的类的派生类）都有一个虚函数表，该类的任何对象中都放着虚函数表的指针。虚函数表中列出了该类的虚函数地址。
- 纯虚函数：
    - 若类的某些虚函数只能抽象出原型，无法定义其实现，则可定义为纯虚函数，其实现由它的派生类定义
    - `virtual <type> FuncName(<ArgList>)=0;`
    - 含有纯虚函数的类肯定是抽象类，因虚函数没有实现部分，不能创建对象。但可定义抽象类类型的指针(或引用),以便用这种基类类型的指针变量指向其派生类的对象(或用这种基类类型的引用变量关联其派生类的对象)时，调用其派生类重定义的纯虚函数，引发运行时多态性。
    - 纯虚函数的好处：接口与其实现分离，提高了类的抽象层次，便于建立接口统一、功能与时俱进的类体系

标准输入输出

函数原型定义与调用

- 函数原型：
    - 只声明函数名、返回值类型、形参列表
    - e.g. `char toLower(char c);`
    - 可以使得函数先被调用后被定义
- 作用：
    - 编译器正确处理函数的返回值；
    - 编译器检查使用的参数数目是否正确；
    - 编译器检查使用的参数类型是否正确。如果不正确，则转换为正确的类型（如果可能的话）

类的定义、属性成员和函数成员的使用

- 类的定义： `class className{};`
- 属性成员：

    - 对象.属性
    - 对象指针—>属性
- 函数成员：
    - 对象.函数
    - 对象指针—>函数

大小写字母转换

函数形参默认值的使用

- 缺省参数的说明必须出现在函数调用之前
- 参数的缺省值可以是表达式，但应有确定的值
- 函数的缺省参数可有多个，但缺省参数应从参数表的最右边依次向左设定

递归函数的定义和使用

- 递归：
    - 终止条件
    - 调用自身
- e.g.

```
// 求n的阶乘
long test(long n) {
        if(n == 1) {
         return 1;
        }
        else {
                return n*test(n-1);
        }
}
```

提取运算符和插入运算符重载

- 提取运算符

```
//提取运算符，复数类为例
//类中友元函数声明
friend istream& operator>>(istream&, Complex&);
//类外实现
istream& operator>>(ostream& is, Complex& c){
        is >> c.real >> c.imag;
        return is;
}
```

- 插入运算符

```
//插入运算符，复数类为例
//类中友元函数声明
friend ostream& operator<<(ostream&, Complex&);
//类外实现
ostream& operator<<(ostream& os, Complex& c){
        os << c.real << '+' << c.imag << 'i';
        return os;
}
```

自加自减运算符重载

- 自加（++）：
  - 前置：`<type> operator++();`
  - 后置：`<type> operator++(int);`
  - e.g.

```
class Time{
        //构造函数、析构函数省略
public:
        //时间自增
        void AddOne() {
                second++;
                if(second == 60) {
                        second = 0;
                        minute++;
                        if(minute == 60) {
                                minute = 0;
                                hour++;
                                if(hour == 24) {
                                        hour = 0;
                                }
                        }
                }
        }

        //前置自增
        Time& operator++() {
                AddOne();
                return *this;
        }

        //后置自增
        Time& operator++(int) {
                Time t = *this;
                AddOne();
                return t;
        }
private:
        int hour, minute, second;
};
```

- 自减（--）：
  - 前置：`<type> operator--();`
  - 后置：`<type> operator--(int);`
  - 类似自加