面向 OP模板

```
//*********************************************************//
//  author：元兔乙                                          //
//  time：202306010                                        //
//*********************************************************//
```

类和对象

显示声明与隐式声明

```
class Circle{
public:
    double d;
    double sum;
    void Get_R()
    {
        double x;
        std::cin>>x;
        d = x;
    }
    void Calculate();
    void Print();
};
void Circle::Calculate()
{
    sum = acos(-1.0) * d * d;
}
void Circle::Print()
{
    std::cout<<std::fixed<<std::setprecision(2)<<sum;
}
```

构造函数

```
class Book
{
private:
    char *name; //书名
    char *author; //作者
    int sale; //销售量
public:
    Book(); //无参构造函数
    Book(char* _name, char* _author, int _sale); //有参构造函数
    Book(const Book &b); //拷贝构造函数
    void print(); //显示数据
    ~Book(); //析构函数
};
```

```
Book::Book()
{
     name = new char[100];
     strcpy (name,"No name");
     author = new char[100];
     strcpy (author,"No author");
     sale=0;
}
Book::Book(char* _name,char* _author,int _sale)
{
    name = new char [strlen(_name) + 1];
    strcpy (name,_name);
    _author = new char [strlen(_author) + 1];
    strcpy (author,_author);
    sale = _sale;
}
Book::Book(const Book &b)
{
    name = new char[strlen(b.name)+1];
    strcpy (name,b.name);
    author = new char[strlen(b.author)+1];
    strcpy (author,b.author);
    sale = b.sale;
}
Book::~Book()
{
    delete[] name;
    delete[] author;
}
```

指针调用函数

```
class Score{
public:
    std::string name;//记录学生姓名
    double s[4];//存储4次成绩，s[0]和s[1]存储2次随堂考试，s[2]存储期中考试，s[3]存储期末考试
    double total;//记录总评成绩
    char grade;  //记录对应的等级
    void Input()
    {
        std::cin>>name;
        for(int i = 0; i < 4; i++)std::cin>>s[i];
        if(s[0] < 0 || s[0] > 50 || s[1] < 0 || s[1] > 50 || s[2] < 0 || s[2] > 100 || s[3] < 0 || s[3] > 100)
        {
            std::cout<<"error"<<std::endl;
            grade = 'W';
        }
    }
    void Evalauate()
    {
        if(grade == 'w')return;
        total = s[3] * 0.5 + s[2] * 0.25 + (s[1] + s[0]) * 0.25;
        if(total >= 90 && total <= 100)grade = 'A';
```

```
        else if(total <= 89 && total >= 80)grade = 'B';
        else if(total <= 79 && total >= 70)grade = 'C';
        else if(total <= 69 && total >= 60)grade = 'D';
        else if(total < 60)grade = 'E';
        else grade = 'W';
    }
    void Output()
    {
        if(grade != 'W')std::cout << "name: " << name << ", total: " << total <<
", grade: " << grade << std::endl;
    }

};

void solve()
{
    Score *s1=new Score;
    s1->Input();
    s1->Evalauate();
    s1->Output();

}
```

拷贝构造函数

```
class Person
{
private:
         int num;
         char *name;
public:
    Person(int n, char *str)
    {
        num = n;
        name = new char[strlen(str)+1];
        strcpy(name, str);
    }
    Person(const Person &x)
    {
        this->num = x.num;
        name = new char [strlen(x.name)+1];
        strcpy(name,x.name);
   }
     ~Person()
    {    cout<<"Destructor:"<<name<<endl;
        delete[] name;
    }
    void show()
    {    cout<<"num="<<num<<"\nname="<<name<<endl;
    }
};
```

静态成员

类内定义，类外声明

通过域作用符访问。

静态数据成员

需要在类外来定义，可以不通过对象直接调用

```
class A{
public:
    static int i;
    const int c = 5;
};
int A::i = 5;
int main()
{
    A a;
    std::cout<<A::i<<"\n";
    std::cout<<a.i<<"\n";
    std::cout<<a.c<<"\n";
    return 0;
}
```

静态成员函数

静态成员函数只能访问静态成员。

非静态成员函数可以访问所有成员，包括静态和非静态。

```
class Box{
    public:
        static int c;
        int a,b;
        Box(int x,int y){
            a=x;b=y;
        }
        static void fun(){
            cout<<c<<endl;
            cout<<"static fun----"<<endl;
        }
};
int Box::c=8;
int main(){
    Box box(2,3);
    Box::fun();//使用作用域运算符直接访问静态成员
    box.fun();//通过对象.引用名
    Box * box2=&box;
    box2->fun(); //通过类Box对象的指针
    return 0;
}
```

const关键字

const关键字修饰的内容为常量，不可修改

const 用于成员变量

const修饰的变量不能被修改，且必须进行初始化

可以使用直接赋值或者使用默认构造函数的初始化列表进行赋值

```
class A{
public:
    const int c;
    const int b = 4;
    A():c(5){};
};

int main()
{
    A a;
    std::cout<<a.c<<"\n";
    std::cout<<a.b<<"\n";
    return 0;
}
```

const修饰成员函数

对于const修饰的成员函数，不可以修改里面的成员变量

```
class A{
public:
    int a;
    A():a(5){};
    void change()const;

};
void A::change()const
{
    this->a = 4;
}
int main()
{
    A a;
    a.change();
    std::cout<<a.a<<"\n";
    return 0;
}
```

如上述代码，结果会报错，如果改成下述内容，则不会

```
class A{
public:
    int a;
    A():a(5){};
    void change();
```

```
};
void A::change()
{
    this->a = 4;
}
int main()
{
    A a;
    a.change();
    std::cout<<a.a<<"\n";
    return 0;
}
```

就像尽可能地将函数形参中的引用和指针声明为const一样，只要成员函数不修改调用对象，就应该将其声明为const，这样既可以避免调用对象被修改，而且普通对象和const对象都可以调用（不会造成读写权限的放大）。

前面提到过，读写权限只能缩小，不能放大，所以，const对象只能调用const成员函数，不可以调用非const成员函数，因为const对象不可以被修改，如果const对象调用非const成员函数，非const成员函数可以修改调用它的const对象，这是权限的放大，会报错。非const对象可以调用非const成员函数，也可以调用const成员函数，因为这是权限的缩小。

const修饰类的成员函数，则该成员函数不能调用类中任何非const成员函数，因为非const成员函数可能会对const对象造成修改，引起权限的放大。

```
class A{
public:
    int a;
    const int b = 6;
    A():a(5){};
    void change()const;
    void print();
};
void A::change()const
{
    std::cout<<a<<"\n";
}
void A::print()
{
    std::cout<<b<<"\n";
}
int main()
{
    A a;
    const A b;
    a.change();
    a.print();
    std::cout<<a.a<<"\n";
    std::cout<<a.b<<"\n";
    return 0;

}
```

const修饰对象

对象内容不可被修改

对象为常对象，则只能调用const成员函数,不能调用非const成员函数

```
#include <bits/stdc++.h>
using namespace std;
class VoiceActor{
public:
    const std::string name;
    const std::string gender;
    VoiceActor(const std::string &_name,const std::string
&_gender):name(_name),gender(_gender){}
    std::string get_name()const {return name;}
    std::string get_gender()const {return gender;}
};
class Cartoon{
public:
    std::string n;
    const VoiceActor a;
    Cartoon(const std::string &s,const VoiceActor &v):n(s),a(v){}
    virtual void print()const
    {
        std::cout<<"Cartoon name: "<<n<<"\n";
        std::cout<<"voice actor: "<< a.get_name() << ", " << a.get_gender();

    }
};
class SeasonsCartoon : public Cartoon
{
public:
    int number;
    SeasonsCartoon(const std::string &s,const VoiceActor &v,int
num):Cartoon(s,v),number(num){}
    virtual void print()const
    {
        std::cout<<"Cartoon name: "<<n<<"\n";
        std::cout<<"voice actor: " << a.get_name()<<", "<<a.get_gender()<<"\n";
        std::cout<<"number of seasons: "<<number<<"\n";
    }
};
int main()
{
    string name_of_voiceActor, gender_of_voiceActor;
    cin >> name_of_voiceActor >> gender_of_voiceActor; // 输入配音演员的姓名和性别
    const VoiceActor va(name_of_voiceActor, gender_of_voiceActor); // 定义一个常配
音演员对象
    cout << "voice actor: " << va.get_name() << ", " << va.get_gender() << '\n';
// 输出配音演员的姓名和性别

    string name_of_cartoon;
    cin >> name_of_cartoon; // 输入动画片的名字
    const Cartoon c(name_of_cartoon, va); // 构造一个常动画片对象

    int number_of_seasons;
```

```
    cin >> number_of_seasons; // 输入季度动画片的季度数（比如第7季）
    const SeasonsCartoon sc(name_of_cartoon, va, number_of_seasons); // 构造一个常季度动画片对象

    const Cartoon* p = &c; // 指针p指向动画片对象c
    p->print(); // 输出动画片的名字，所用的配音演员的名字和性别（参见输出样例）
    cout << '\n';

    p = &sc; // 指针p指向季度动画片对象sc
    p->print(); // 输出季度动画片的名字，所用的配音演员的名字、性别，以及季度数（参见输出样例）

    return 0;
}
```

友元

友元函数

友元函数必须有一个参数是友元类的指针，友元类的对象或友元类的引用。

```
#include <bits/stdc++.h>
class A{
private:
    int a;
public:
    A():a(5){};
    friend void print(const A &);
};
void print(const A &aa)
{
    std::cout<<aa.a<<"\n";
}
int main()
{
    A a;
    print(a);
    return 0;
}
```

使用时直接使用即可

```
class T
{
    public:
        T();
        ~T();
        //不引入类对象
        friend void show_hello_no_param()
        {
            std::cout << "show_hello_no_param() of T : Hello world!\n";
        }
        //引入类对象
```

```
        friend void show_hello(T t)
        {
            std::cout << "show_hello() of T : Hello world!\n";
        }
};

int main(int argc, char* argv[])
{
    T t;
    show_hello_no_param();      //编译不通过
    show_hello(t);              //编译可以通过
    getchar();

    return 0;
}
```

友元类

```
#include <bits/stdc++.h>
class B;
class B{
private:
    int b;
public:
    B():b(10){};
    // friend class A;
};
class A{
private:
    int a;
public:
    A():a(5){};
    friend void print(const A &);
    void print1_mul(const B &b)
    {
        std::cout<<b.b<<"*"<<b.b<<"="<<b.b*b.b<<"\n";
    }
    friend class B;
};

void print(const A &aa)
{
    std::cout<<aa.a<<"\n";
}
int main()
{
    A a;
    B b;
    print(a);
    a.print1_mul(b);
    return 0;
}
```

如上述代码，如果不将A定义成B的友元，则无法在A的成员函数中调用B的成员变量。

```
#include <bits/stdc++.h>
class B;
class B{
private:
    int b;
public:
    B():b(10){};
    friend class A;
};
class A{
private:
    int a;
public:
    A():a(5){};
    friend void print(const A &);
    void print1_mul(const B &b)
    {
        std::cout<<b.b<<"*"<<b.b<<"="<<b.b*b.b<<"\n";
    }
    friend class B;
};

void print(const A &aa)
{
    std::cout<<aa.a<<"\n";
}
int main()
{
    A a;
    B b;
    print(a);
    a.print1_mul(b);
    return 0;
}
```

定义后则可以使用。

与友元类类似的访问方式还有将一种类定义成另一种类的成员。如下

```
#include <bits/stdc++.h>
class A{
private:
    int a;
public:
    A():a(5){};
    void print()
    {
        std::cout<<a<<"\n";
    }
};
class B{
private:
public:
```

```
    A a;
};
int main()
{
    B b;
    b.a.print();
}
```

继承与派生

类型兼容

子类对象可以当作父类对象使用

子类对象可以直接赋值给父类对象

子类对象可以直接初始化父类对象

父类对象不可以直接初始化子类对象

```
class A{
public:
    int a,b;
    A():a(5),b(6){};
};
class B : public A{
public:
    int c;
    B():c(3){};
};
void solve()
{
    B a;
    A c = a;
//  B d = c; 这句不能通过编译，因为父类不能初始化子类
}
```

父类指针可以直接指向子类对象

父类引用可以直接引用子类对象

```
class Animal
{
public:
void sleep()
{
    cout<<"Animal sleep"<<endl;}
};
class Dog:public Animal
{
    void bite(){cout<<"Dog bite"<<endl;}
};
    int main(int argc,char**argv)
{
```

```
//1 子类对象可以当做父类对象使用
    Dog dog;
    dog.sleep();//子类对象可以调用父类方法
//2 子类对象可以直接赋值给父类对象
    Animal a;
    a = dog;
//3子类对象可以直接初始化父类对象
    Animal b=dog;
    //Dog dog1=a; 父类对象不可以直接初始化子类对象
//4父类指针可以直接指向子类对象
    Animal *c =new Dog;
    //Dog *dog3=new Animal; //子类指针不可以指向父类对象
//5 父类引用可以直接引用子类对象
    Animal &d=dog;
    return 0;
}
```

友元函数没有继承

普通继承

对于基类和派生类之间相同的成员名，如果在两个类中均存在，派生类优先调用派生类，基类调用基类;

如果需要在派生类中调用基类中的同名函数，则需要加上域作用符。

```
#include <bits/stdc++.h>
class A{
public:
    int a;
    A():a(5){}
    void print()
    {
        std::cout<<"1\n";
        std::cout<<a<<"\n";
    }
};
class B:public A{
public:
    int b;
    B():b(4){A();};
    void print()
    {
        std::cout<<"2\n";
        std::cout<<b<<"\n";
    }
};
int main()
{
    B b;
    b.print();
    b.A::print();
}
```

单继承中的构造析构函数的调用顺序

从基类构造->子类构造->子类析构->基类析构

```
#include <bits/stdc++.h>
class A{
public:
    int a;
    A(){std::cout<<"1\n";}
    ~A(){std::cout<<"2\n";}
};
class B:public A {
public:
    int b;
    B(){std::cout<<"3\n";}
    ~B(){std::cout<<"4\n";}
};
int main()
{
    B b;
    //输出结果为1 3 4 2
}
```

多重继承

多重继承的初始化如下，子类中初始化父类需要初始化全部参数

```
#include <bits/stdc++.h>
using namespace std;
class VoiceActor{
public:
    const std::string name;
    const std::string gender;
    VoiceActor(const std::string &_name,const std::string
&_gender):name(_name),gender(_gender){}
    std::string get_name()const {return name;}
    std::string get_gender()const {return gender;}
};
class Cartoon{
public:
    std::string n;
    const VoiceActor a;
    Cartoon(const std::string &s,const VoiceActor &v):n(s),a(v){}
    virtual void print()const
    {
        std::cout<<"Cartoon name: "<<n<<"\n";
        std::cout<<"voice actor: "<< a.get_name() << ", " << a.get_gender();

    }
};
class SeasonsCartoon : public Cartoon
{
public:
    int number;
```

```
    SeasonsCartoon(const std::string &s,const VoiceActor &v,int
num):Cartoon(s,v),number(num){}
    virtual void print()const
    {
        std::cout<<"Cartoon name: "<<n<<"\n";
        std::cout<<"voice actor: " << a.get_name()<<", "<<a.get_gender()<<"\n";
        std::cout<<"number of seasons: "<<number<<"\n";
    }
};
int main()
{
    string name_of_voiceActor, gender_of_voiceActor;
    cin >> name_of_voiceActor >> gender_of_voiceActor; // 输入配音演员的姓名和性别
    const VoiceActor va(name_of_voiceActor, gender_of_voiceActor); // 定义一个常配
音演员对象
    cout << "voice actor: " << va.get_name() << ", " << va.get_gender() << '\n';
// 输出配音演员的姓名和性别

    string name_of_cartoon;
    cin >> name_of_cartoon; // 输入动画片的名字
    const Cartoon c(name_of_cartoon, va); // 构造一个常动画片对象

    int number_of_seasons;
    cin >> number_of_seasons; // 输入季度动画片的季度数（比如第7季）
    const SeasonsCartoon sc(name_of_cartoon, va, number_of_seasons); // 构造一个常
季度动画片对象

    const Cartoon* p = &c; // 指针p指向动画片对象c
    p->print(); // 输出动画片的名字，所用的配音演员的名字和性别（参见输出样例）
    cout << '\n';

    p = &sc; // 指针p指向季度动画片对象sc
    p->print(); // 输出季度动画片的名字，所用的配音演员的名字、性别，以及季度数（参见输出样
例）

    return 0;
}
```

*将析构函数最好定义成虚函数

```
class A{
public:
    int a;
    A():a(5){};
};
class B{
public:
    int b;
    B():b(6){};
};
class C:public A,public B
{
public:
    void print()
    {
```

```
        std::cout<<A::a<<" "<<B::b<<"\n";
    }
};
void solve()
{
    C c;
    c.print();
    std::cout<<c.B::b<<" "<<c.A::a<<"\n";
}
```

子类拥有所有父类的成员函数

```
class A{
public:
    int a;
    A():a(6){};
};
class B : public A{
public:
    int b;
    B():b(5){};
};
class C : public B{
public:
    int c;
    C():c(7){};
};
void solve()
{
    C c;
    std::cout<<c.A::a<<" "<<c.B::b<<'\n';
}
```

多继承及其二义性问题

一个子类拥有很多父类，一般指一个类有2个以上父类。

第一类二义性问题

同名二义性：在继承时，基类，基类和派生类之间存在相同名称的成员，可以使用域作用符来限定修饰

```
class A
{
public:
    void show()
    {
        cout<<"This is A\n";
    }
};
class B
{
public:
    void show()
    {
        cout<<"This is B\n";
```

```
    }
};
class Son:public A,public B
{
public:
    void display()
    {
        cout<<"This is son\n";
    }
};
int main()
{
    Son son;
    //son.show(); 错误，因为不知道访问类A中的show()，还是访问类B中的show()
    son.A::show();
    son.B::show();
    son.display();
    cout << "Hello world!" << endl;
    return 0;
}
```

或者直接重写该函数，覆盖掉基类中相关成员

```
class A
{
public:
    void show()
    {
        cout<<"This is A"<<endl;
    }
};
class B
{
public:
    void show()
    {
        cout<<"This is B"<<endl;
    }
};
class Son:public A,public B
{
public:
    void display()
    {
        cout<<"This is son"<<endl;
    }
    void show()
    {
        cout<<"show::This is son"<<endl;
    }
};
int main()
{
    Son son;
    son.show();//此时son.show()将调用son.Son::show()
```

```
    son.display();
    cout << "Hello world!" << endl;
    return 0;
}
```

第二类二义性问题

路径二义性：有最基类A，有A的派生类B、C，又有D同时继承B、C，那么若A中有成员m，那么在派生类B，C中就存在m，又D继承了B，C，那么D中便同时存在B继承A的m和C继承A的m，那么当D的实例调用m的时候就不知道该调用B的m还是C的m，就导致了二义性。

使用域限定符

```
class A
{
public:
    int m;
};
class B:public A {};
class C:public A {};

class D:public B,public C {};

int main()
{
    D d;
    //d.m= 10;错误，因为不知道调用B的m还是C的m
    d.B::m = 10;
    d.C::m = 10;
    cout << "Hello world!" << endl;
    return 0;
}
```

定义同名成员进行覆盖

```
//正确代码举例
#include <iostream>
using namespace std;
class A
{
public:
    int m;
};
class B:public A {};
class C:public A {};
class D:public B,public C
 {
    public:
    int m;
 };
int main()
{
    D d;
```

```
    d.m= 10;//此处相当于访问的D的m
    cout<<d.D::m<<endl;
    cout<<"Hello world!"<<endl;
    return 0;
}
```

使用虚继承

```
class A
{
public:
    int m;
};
class B:virtual public A {};
class C:virtual public A {};
class D:public B,public C
 {
    public:
    int m;
 };
int main()
{
    D d;
    d.m= 10;
    cout<<"Hello world!"<<endl;
    return 0;
}
```

多态与虚函数

虚函数

需要使用虚函数的情况：声明基类指针但是指向子类对象，需要通过指针访问子类成员函数。

如果不使用虚函数，则只能访问到指针所属的类的成员函数。

```
class Person{
public:
    std::string name;
    virtual void input()
    {
        std::cin>>name;
    }
    virtual void display()
    {
        std::cout<<name<<"\n";
    }
};

class Student : public Person{
public:
    std::string number;
    virtual void input()
```

```
    {
        //std::cout<<"zilei\n";
        std::cin>>number>>name;
    }
    virtual void display()
    {
        std::cout<<name<<" "<<number;
    }
};

void solve()
{
    Person * p;
    p = new Person;
    p->input();
    p->display();
    delete p;
    p = new Student;
    p->input();
    p->display();
    delete p;
}
```

抽象类/纯虚函数

抽象类就是含有纯虚函数的类

抽象类不能有实例对象

子类中没有覆盖所有的纯虚函数，则子类还是抽象类

```
class Vehicle{ // 抽象类
public:
    std::string name;
    std::string color;
    Vehicle(const std::string &nn,const std::string &cc):name(nn),color(cc){};
    virtual void display() = 0; // 纯虚函数
};

class Car : public Vehicle{
public:
    int passenger;
    Car(const std::string &nn,const std::string &cc,int n):Vehicle(nn,cc)
    {
        passenger = n;
    }
    void display()
    {
        std::cout<<name<<" car, "<<color<<", passenger volume: "
<<passenger<<"\n";
    }
};

class Truck : public Vehicle{
public:
```

```
    double weight;
    Truck(const std::string &nn,const std::string &cc,double n):Vehicle(nn,cc)
    {
        weight = n;
    }
    void display()//纯虚函数的重写
    {
        std::cout << name << " truck , " << color << ", cargo capacity: " <<
weight<< "(t)" << std::endl;
    }
};
```

友元关系和继承

友元关系不能继承。基类的友元对派生类的成员没有特殊访问权限。

每个类控制自己类的友元函数

静态成员和继承

不管多少个类都共用一个静态成员

```
class A{
public:
    static int a;
};
class B : public A{};
int A::a = 5;
int main()
{
    std::cout<<A::a<<"\n"; // 5
    B b;
    std::cout<<B::a<<"\n"; // 5
    B::a = 6; // 更改a的值，后面也会更改
    std::cout<<b.a<<"\n"; // 6
    std::cout<<A::a<<"\n"; // 6
}
```

重载

函数重载

C++允许在同一作用域中声明几个类似的同名函数，这些同名函数的形参列表（参数个数，类型，顺序）必须不同，常用来处理实现功能类似数据类型不同的问题。

```
// my_max + 参数表
int my_max(int a,int b)
{
    return a > b ? a : b;
}
char my_max(char a,char b)
{
```

```
    return a > b ? a : b;
}
double my_max(double a,double b)
{
    return a > b ? a : b;
}
//每个同名函数的参数表是惟一
int main()
{
    int ix = my_max(12,23);
    double dx = my_max(12.23,34.45);
    char chx = my_max('a','b');
    return 0;
}
```

运算符重载

成员函数的重载比友元函数少一个参数

类作为返回值的重载

两个类进行基础的四则运算

```
class Complex{
public:
    double a;
    double b;
    Complex operator+(const Complex &c)
    {
        Complex a;
        a.a = this->a + c.a;
        a.b = this->b + c.b;
        return a;
    }
    Complex operator-(const Complex &c)
    {
        Complex a;
        a.a = this->a - c.a;
        a.b = this->b - c.b;
        return a;
    }
    Complex operator*(const Complex &c);
    Complex operator/(const Complex &c);
    Complex operator+=(const Complex &c);
    Complex operator-=(const Complex &c);
    Complex operator*=(const Complex &c);
    Complex operator/=(const Complex &c);
};
```

引用作为返回值的重载

赋值运算符的重载

```
class Complex{
public:
    double a;
    double b;
    Complex& operator=(const Complex &c)
    {
        this->a = c.a;
        this->b = c.b;
        return *this;
    }
};
```

其他以引用作为返回值的函数

```
class Complex{
public:
    double a;
    double b;
    Complex& operator=(const Complex &c)
    {
        this->a = c.a;
        this->b = c.b;
        return *this;
    }
    Complex& operator+=(const Complex &c)//也可以返回对象
    {
        this->a += c.a;
        this->b += c.b;
        return *this;
    }
    Complex& operator-=(const Complex &c);
    Complex& operator*=(const Complex &c);
    Complex& operator/=(const Complex &c);
//  std::ostream& operator<<(std::ostream &os,const Complex &c);    输入输出符最好重载为友元函数
//  std::istream& operator>>(std::istream &is,const Complex &c);
};
```

友元函数的重载

双目运算符均可重载为友元函数

```
class Singer{
public:
    std::string name;
    char gender;
    int age;
    double score;
    Singer(std::string name = "",char gender = '?',int age = 0,double score = 0.0);
```

```
    std::string getName();
    friend bool operator>(const Singer &,const Singer &);
    friend bool operator<(const Singer &,const Singer &);
    friend bool operator==(const Singer &,const Singer &);
    friend std::ostream& operator<<(std::ostream &,const Singer &);
    friend std::istream& operator>>(std::istream &,const Singer &);
};
Singer::Singer(std::string name,char gender,int age,double score)
{
    this->name = name;
    this->gender = gender;
    this->age = age;
    this->score = score;
}
std::string Singer::getName()
{
    return name;
}
bool operator>(const Singer &s1,const Singer &s2)
{
    return s1.score > s2.sorce;
}
bool operator>(const Singer &s1,const Singer &s2)
{
    return s1.score > s2.sorce;
}
bool operator==(const Singer &s1,const Singer &s2)
{
    return s1.score == s2.sorce;
}
std::ostream& operator<<(std::ostream &os,const Singer &s)
{
    os << s.name << " " << s.gender << " " << s.age << " " << s.sorce;
    return os;
}
std::istream& operator>>(std::istream &is,const Singer &s)
{
    is >> s.name >> s.gender >> s.age >> s.sorce;
    return os;
}
```

自增自减

如果定义了++c，也要定义c++.

对于++和-而言，后置形式是先返回，然后对象++或者-，返回的是对象的原值。前置形式，对象先++或-，返回当前对象，返回的是新对象。其标准形式为:

调用代码时候，要优先使用前缀形式，除非确实需要后缀形式返回的原值，前缀和后缀形式语义上是等价的，输入工作量也相当，只是效率经常会略高一些，由于前缀形式少创建了一个临时对象。

```
class CheckedPtr{
public:
    CheckedPtr(int *b,int *e):begin(b),end(e),current(b){};
    CheckedPtr& operator++()//++a
    {
```

```
        if(current == end)std::cout<<"end\n";
        else this->current += 1;
        return *this;
    }
    CheckedPtr& operator--();//--a
    {
        if(current == end)std::cout<<"end\n";
        else this->current -= 1;
        return *this;
    }
    CheckedPtr operator++(int)
    {
        CheckedPtr res(*this);
        this->current += 1;
        return res;
    }
    CheckedPtr operator--(int)
    {
        CheckedPtr res(*this);
        this->current -= 1;
        return res;
    }
};
```

```
class Myint {
public:
    Myint(int num)
    {
        this->num = num;
    }
    Myint& operator++()
    {
        this->num = this->num + 1;
        return *this;
    }
    Myint operator++(int)
    {
        Myint temp = *this;
        this->num = this->num + 1;
        return temp;
    }
    int num;
};
```

返回值为其他类型的重载

bool类型的重载

```
//友元声明
    friend bool operator==(const hhdy& a, const hhdy& b);
    friend bool operator!=(const hhdy& a, const hhdy& b);

//非成员函数
    bool operator==(const hhdy& a, const hhdy& b) {
        return a.grades == b.grades && a.item == b.item;
```

```
    }
    bool operator!=(const hhdy& a, const hhdy& b) {
        return !(a == b);
    }
//主函数使用
    std::cout << (h1 == h3) << std::endl;
    std::cout << (h1 != h3) << std::endl;
```

[]的重载

```
class Vector{
public:
    int sum[101];
    Vector()
    {
        for(int i = 1; i < 100; i++)sum[i] = i;
    }
    int& operator[](int n)
    {
        return sum[n];
    }
};
int main()
{
    Vector a;
    std::cout<<a[13]<<"\n"; // 13
    a[13] = 15;
    std::cout<<a[15]<<"\n"; // 15
}
```

成员访问符 -> 和 * 的重载

```
class A{
public:
    string* operator->()
    {
        return & data[curr];
    }
    vector<string> data{"str1","str2","str3"};
    int curr = 0;
};
class B{
public:
    A& operator->()
    {
        return a;
    }

    A a;
};
class C{
public:
```

```
    B& operator->()
    {
        return b;
    }
    B b;
};
int main()
{
    C c;
    cout << c->size() << endl;//顺序的调用，返回的是vector[0]的容器大小
    return 0;
}
```

```
class Date{
public:
    int y;
    int m;
    int d;
};
class DatePtr
{
    Date* ptr;
public:
    //如果创建DataPtr对象时使用参数，那么传进去的指针也就成为了DataPtr的成员ptr的值了
    DatePtr(Date* ptr = NULL):ptr(ptr){}
    //重载了*号
    Date& operator*()
    {
        return *ptr;  //取了ptr的内容后将其返回
    }
    //重载了箭头号
    Date* operator->()
    {
        return ptr;
    }
    //重载了前加加号
    DatePtr& operator++()
    {
        ptr++;
        return *this;
    }
    //重载了中括号
    Date& operator[](int index)
    {
        return *(ptr+index);
    }
};
int main()
{
    Date d = {2012,9,27};
    //DatePtr p(&d);
    //class
    DatePtr p = &d;
    (*p).y = 2013;  //相当于p-> = 2013;，但又能这么写 p.operator*().y = 2013
    p->m = 10;  //p.operator->()->m = 10;
```

```
    cout << d.y << '-' << d.m << '-' << d.d << endl;
    Date ds[3] = {{2012,9,27},{2012,9,30},{2012,12,21}};
    //赋的其实是地址
    DatePtr p2 = ds;
    for (int i = 0; i<3; i++)
    {
    //  cout << p2->y << '-' << p2->m
    //            << '-' << p2->d << endl;
    //            ++p2;
        cout << p2[i].y << '-' << p2[i].m
                  << '-' << p2[i].d << endl;
    }
}
```

() 的重载

```
class cls
{
public:
    void operator() ()  //重载"()"操作符，"()"内无操作数
    {
        printf("HelloWorld!\n");
    }
    void operator() (const char* str) //重载"()"，"()"内的操作数是字符串
    {
        printf("%s", str);
    }
};
int main(void)
{
    cls cc;
    cc(); // HelloWorld!
    cc("Hello Linux\n"); //Hello Linux
    return 0;
}
```

函数模板与类模板

函数模板

```
template <class T>
void Swap(T &a, T &b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

类模板

成员函数都放在类内实现会减少很多麻烦

数组模拟queue类

```
#include <bits/stdc++.h>
using namespace std;
template <class T,int SIZE>
class Queue{
public:
    T a[SIZE];
    int l = 0;
    int r = 0;
    void push(const T &t){a[l] = t; l += 1;}
    T& front(){return a[r];}
    void pop(){r+=1;}
    bool empty(){return abs(r-l) == 0;}
    void clear(){r = SIZE;}
    int size(){return l-r;}
};
int main()
{
    Queue<int, 5> intQueue;   // 可以存放5个int类型元素的队列
    int tmp_i;
    for (int i = 0; i < 5; i++) // 输入5个整数并入队
    {
        cin >> tmp_i;
        intQueue.push(tmp_i);
    }

    for (int i = 0; i < 5; i++)
    { // 依次输出5个整数并出队
        if (i < 4)
            cout << intQueue.front() << ' ';
        else cout << intQueue.front() << '\n';
        intQueue.pop();
    }
    if (intQueue.empty())
        cout << "After 5 pops, intQueue is empty." << endl;

    Queue<string, 3> stringQueue; // 可以存放3个string类型元素的队列
    string tmp_s;
    for (int i = 0; i < 3; i++) // 输入3个字符串并入队
    {
        cin >> tmp_s;
        stringQueue.push(tmp_s);
    }
    cout << stringQueue.front() << '\n'; // 输出队列中的第1个字符串
    cout << "There are " << stringQueue.size() << " elements in stringQueue." <<
'\n';
    stringQueue.clear(); // 清空队列
    if (stringQueue.empty())
        cout << "After clear, there are no elements in stringQueue.";
```

```
}
```

Vector类

```
template <class T>
class Vector{
public:
    T x,y,z;
    Vector()
    {
        x = y = z = 0.0;
    }
    Vector(T a, T b, T c)
    {
        this->x = a;
        this->y = b;
        this->z = c;
    }
    Vector(const Vector &v)
    {
        this->x = v.x;
        this->y = v.y;
        this->z = v.z;
    }
    friend std::istream& operator>>(std::istream &is,Vector<T> &v)
    {
        is >> v.x >> v.y >> v.z;
        return is;
    }
    friend std::ostream& operator<<(std::ostream &os,Vector<T> &v)
    {
        os << v.x << " " << v.y << " " << v.z;
        return os;
    }
    friend Vector<T> operator*(double a,const Vector<T> &v)
    {
        Vector<T> res;
        res.x = a * v.x;
        res.y = a * v.y;
        res.z = a * v.z;
        return res;
    }
    Vector<T> operator+(const Vector<T> &v)
    {
        Vector<T> res;
        res.x = this->x + v.x;
        res.x = this->y + v.y;
        res.z = this->z + v.z;
        return res;
    }
    bool operator==(const Vector<T> &v)
    {
        double op = 1e-6;
```

```
        if(abs(this->x - v.x) > op)return 0;
        if(abs(this->y - v,y) > op)return 0;
        if(abs(this->z - v.z) > op)return 0;
        return 1;
    }

};
```

Myqueue类

```
template <class T> class MyQueue;
template <class T> std::ostream& operator<<(std::ostream &,const MyQueue<T> &);
template <class T>
class QueueItem{
public:
    QueueItem(const T &q):item(q),next(nullptr){}
    T item;
    QueueItem *next;
    friend class MyQueue<T>;
    friend std::ostream& operator<< <T>(std::ostream &os,const MyQueue<T> &mq);
};
template <class T>
class MyQueue{
public:
    MyQueue():head(nullptr),tail(nullptr){}
    MyQueue(const MyQueue &q):head(nullptr),tail(nullptr)
    {
        CopyElements(q);
    }
    ~MyQueue()
    {
        Destroy();
    }
    MyQueue& operator=(const MyQueue &);
    T& Front()
    {
        return head->item;
    }
    const T& Front() const
    {
        return head->item;
    }
    void Push(const T &);
    void Pop();
    bool Empty()const
    {
        return head == nullptr;
    }
    void Display()const;
    QueueItem<T> *head;
    QueueItem<T> *tail;
    void Destroy();
    void CopyElements(const MyQueue &);
    friend std::ostream& operator<< <T>(std::ostream &os,const MyQueue<T> &);
};
```

```
template <class T>
void MyQueue<T>::Destroy()
{
    while(!Empty())Pop();
}

template <class T>
void MyQueue<T>::Pop()
{
    QueueItem<T> *p = head;
    head = head->next;
    delete p;
}

template <class T>
void MyQueue<T>::Push(const T &v)
{
    QueueItem<T> *pt = new QueueItem<T>(v);
    if(!Empty())head = tail = pt;
    else {
        tail->next = pt;
        tail = pt;
    }
}
template <class T>
void MyQueue<T>::CopyElements(const MyQueue<T> &v)
{
    for(QueueItem<T> *pt = v.head;pt;pt = pt->next)Push(pt->next);
}

template <class T>
MyQueue<T>& MyQueue<T>::operator=(const MyQueue<T> &v)
{
    for(QueueItem<T> *pt = v.head;pt;pt = pt->next)Push(pt->next);
        return *this;
}

template <class T>
void MyQueue<T>::Display()const
{
    for(QueueItem<T> *pt = head;pt;pt = pt->next)std::cout<<pt->item<<" ";
}
template <class T>
std::ostream& operator<<(std::ostream &os,const MyQueue<T> &q)
{
    os << " < ";
    for(QueueItem<T> *p = q.head; p; p = p->next)os << p->item<<" ";
    os << ">";
    return os;
}
```

Stack类

```
template <class T,int SIZE = 20>
class Stack{
public:
    T array[SIZE];
    int top;
    Stack();
    void Push(const T &);
    T Pop();
    void Clear();
    const T& Top() const;
    bool Empty() const;
    bool Full() const;
    int Size();
};

template<class T,int SIZE>
Stack<T,SIZE>::Stack():top(-1){};

template<class T,int SIZE>
void Stack<T,SIZE>::Push(const T &t)
{
    if(!Full())array[++top] = t;
}
template<class T,int SIZE>
T Stack<T,SIZE>::Pop()
{
    if(!Empty())return array[top--];
    else exit(1);
}
template<class T,int SIZE>
void Stack<T,SIZE>::Clear(){top = -1;}

template<class T,int SIZE>
const T& Stack<T,SIZE>::Top() const
{
    if(!Empty())return array[top];
    else exit(1);
}
template<class T,int SIZE>
bool Stack<T,SIZE>::Empty() const
{
    return top == -1;
}
template<class T,int SIZE>
bool Stack<T,SIZE>::Full() const
{
    return top == SIZE-1;
}
template<class T,int SIZE>
int Stack<T,SIZE>::Size()
{
    return top + 1;
}
```

文件读写与流

大胆猜一手，文件和流不考。

字符串流

```
unsigned termFrequency(string content, map<string, unsigned>& msu) {
    int L = content.length();
    for (int i = 0; i < L; i++) {
        if (content[i] == '.' || content[i] == ',' || content[i] == '"')
            content[i] = ' ';
    }
    string word;
    stringstream ss(content);
    while (ss >> word)
    {
        int flag = 0;
        for (int i = 0; i < word.size(); i++)
        {
            if (word[i] < '0' || word[i]>'9') {
                flag = 1;
                break;
            }
        }
        if (flag == 1)
        {
            transform(word.begin(), word.end(), word.begin(), ::tolower);
            map<string, unsigned>::iterator ite = msu.find(word);
            if (msu.count(word) == 0)
                msu.insert(map<string, unsigned>::value_type(word, 1));
            else
                ite->second++;
        }
    }
    return msu.size();
}
void alphabetSortedFrequency(map<string, unsigned> msu) {
    map<string, unsigned>::iterator it;
    for (it = msu.begin(); it != msu.end(); it++) {
        cout << it->first << ":" << it->second << endl;
    }
}
int main()
{
    // 从标准输入获取文本串
    std::string content;
    std::getline(std::cin, content, '\n');
    map<string, unsigned> msu;
    // 要求termFrequency实现分词，去掉标点
    // 获取单词存放在map中，记录词频（出现次数）
    // 最后返回不重复的单词数量
    unsigned nWords = termFrequency(content, msu);
    // 按首字母A-Z排序一行一词输出词频
```

```
    alphabetSortedFrequency(msu);
    return 0;
}
```

文件读写流

```
void processPoints()
{
    double points_x, points_y, points_z, point_r, point_g, point_b;
    double sum_x = 0, sum_y = 0, sum_z = 0;
    int pr = 0, ps = 0, pb = 0;
    double X[20], Y[20];
    double x[] = {
244407.100,244407.097,244407.080,244407.124,244407.120,244407.125,244407.090,2444
07.072,244407.119,244407.079,244407.096,244407.089,244407.066,244407.112,244407.0
89,244407.067,244407.103,244407.057,244407.127,244407.074 };
    double y[] = {
6010942.604,6010942.547,6010942.541,6010942.599,6010942.553,6010942.565,6010942.5
70,6010942.575,6010942.576,6010942.575,6010942.581,6010942.604,6010942.598,601094
2.599,6010942.598,6010942.569,6010942.524,6010942.512,6010942.525,6010942.524 };
    double z[] = {
19.256,19.244,19.242,19.253,19.240,19.241,19.249,19.253,19.246,19.248,19.250,19.2
55,19.253,19.252,19.255,19.247,19.238,19.240,19.235,19.241 };
    int R[] = {
140,142,144,144,140,144,142,145,140,161,142,140,144,137,138,149,147,161,144,154
};
    int G[] = {
131,131,135,131,130,133,131,135,130,151,133,131,135,128,130,142,137,153,135,142
};
    int B[] = {
124,126,128,126,124,128,126,126,124,147,126,124,128,121,124,133,130,144,128,135
};
    fstream myfile;
    myfile.open("D:\\points.csv", ios::out | ios::in | ios::trunc);
    if (!myfile) {
        exit(0);
    }
    myfile << "X, Y, Z, R, G, B\n";
    for (int i = 0; i < 20; i++)
    {
        myfile << fixed << setprecision(3) << x[i] << "," << y[i] << "," << z[i]
<< "," << R[i] << "," << G[i] << "," << B[i] << endl;
    }
    for (int i = 0; i < 20; i++)
    {
        sum_x += x[i];
        sum_y += y[i];
        sum_z += z[i];
        pr += R[i];
        ps += G[i];
        pb += B[i];
    }
    points_x = sum_x / 20;
```

```
    points_y = sum_y / 20;
    points_z = sum_z / 20;
    point_r = pr / 20;
    point_g = ps / 20;
    point_b = pb / 20;
    myfile << fixed << setprecision(3) << points_x << "," << points_y << "," <<
points_z << "," << point_r << "," << point_g << "," << point_b;
    myfile.close();
    myfile.open("D:\\points_offset.csv", ios::trunc | ios::app);
    for (int i = 0; i < 20; i++)
    {
        X[i] = x[i] + 100;
        Y[i] = y[i] - 50;
    }
    myfile << "X, Y, Z, R, G, B\n";
    for (int i = 0; i < 20; i++)
    {
        myfile << fixed << setprecision(3) << X[i] << "," << Y[i] << "," << z[i]
<< "," << R[i] << "," << G[i] << "," << B[i] << endl;
    }
    myfile.close();
}
int main() {
    std::cout << "Point cloud in processing..." << endl;
    processPoints();
    return 0;
}
```

STL

```
#include <bits/stdc++.h>
using namespace std;
class Contestant{
public:
    std::string name;
    int num;
    int time;
    int ax;
    int all_time;
    Contestant():name(""),num(0),time(0),ax(0),all_time(0){};
    friend std::istream& operator>>(std::istream &is,Contestant &c);
    friend std::ostream& operator<<(std::ostream &os,const Contestant &c);
    void add_penalty()
    {
        all_time = time;
        all_time = all_time + (ax * 20);

    }
};
bool compare(const Contestant a,const Contestant b)
{
    if(a.num > b.num)return 1;
    else if(a.num < b.num)return 0;
    else {
        return a.all_time < b.all_time;
```

```
    }
}
std::istream& operator>>(std::istream &is,Contestant &c)
{
    is >> c.name >> c.num >> c.time >> c.ax;
    return is;
}
std::ostream& operator<<(std::ostream &os,const Contestant &c)
{
    os << c.name << " " << c.num << " " << c.time << " " << c.ax << " " <<
c.all_time;
    return os;
}
int main()
{
    int n;
    cin >> n;
    Contestant c;
    vector<Contestant> v;
    for (int i = 0; i < n; i++)
    {
        cin >> c;          //输入选手名、解题数、解题用时、错误提交次数
        c.add_penalty(); //根据错误提交次数计算罚时，并加到总用时上
        v.push_back(c);
    }

    sort(v.begin(), v.end(), compare);

    for (int i = 0; i < v.size(); i++)
        cout << v[i] << '\n'; //输出选手名、解题数、解题用时、错误提交次数、加上罚时的总用
时

    return 0;

}
```

STL 从广义上分为: 容器(container) 算法(algorithm) 迭代器(iterator)。

使用时需要加上对应的头文件

string 类

```
    // 基本操作
    string s1;              // 构造空的string类对象s1
    string s2("giturtle");  // 用C格式字符串构造string类对象s2
    string s3(10, 'a');     // 用10个字符'a'构造string类对象s3
    string s4(s2);          // 拷贝构造s4
    string s5(s3, 5);       // 用s3中第5个字符起、字符串结尾止的字符串构造string对象s5

    // 注意：string类对象支持直接用cin和cout进行输入和输出
    string s("giturtle");
    cout << s.length(); // 返回字符串的长度
    cout << s.size() << endl; // 返回字符串的大小
    cout << s.capacity() << endl; //返回空间总大小
    cout << s << endl;
```

```
    // 将s中的字符串清空，注意清空时只是将size清0，不改变底层空间的大小
    s.clear();
    cout << s.size() << endl;
    cout << s.capacity() << endl;

    // 将s中有效字符个数增加到10个，多出位置用'a'进行填充
    // “aaaaaaaaaa”
    s.resize(10, 'a');
    cout << s.size() << endl;
    cout << s.capacity() << endl;

    // 将s中有效字符个数增加到15个，多出位置用缺省值'\0'进行填充
    // "aaaaaaaaaa\0\0\0\0\0"
    // 注意此时s中有效字符个数已经增加到15个
    s.resize(15);
    cout << s.size() << endl;
    cout << s.capacity() << endl;
    cout << s << endl;

    // 将s中有效字符个数缩小到5个
    s.resize(5);
    cout << s.size() << endl;
    cout << s.capacity() << endl;
    cout << s << endl;

    string s1("giturtle");
    const string s2("giturtle");
    cout << s1 << " " << s2 << endl;
    cout << s1[0] << " " << s2[0] << endl;

    s1[0] = 'H';
    cout << s1 << endl;

    for (size_t i = 0; i < s1.size(); ++i){
        cout << s1[i] << endl;
    }

    // s2[0] = 'h'; 代码编译失败，因为const类型对象不能修改

    //修改操作
    void push_back(char c)  //在字符串后尾插字符c
    string& append (const char* s); //在字符串后追加一个字符串
    string& operator+=(const string& str)   //在字符串后追加字符串str
    string& operator+=(const char* s)   //在字符串后追加C个数字符串
    string& operator+=(char c)  //在字符串后追加字符c
    const char* c_str() const   //返回C格式字符串
    size_t find (char c, size_t pos = 0)const//从字符串pos位置开始往后找字符c，返回该字符在字符串中的位置 没找到返回-1

    string substr(size_t pos = 0, size_t n = npos)const //在str中从pos位置开始，截取n个字符，然后将其返回

    // 非成员函数
    bool operator<(const std::string &s1,const std::string &s2)
    bool operator>(const std::string &s1,const std::string &s2)
```

```
    bool operator==(const std::string &s1,const std::string &s2)
    getline();
```

容器

此类型下的容器均可以通过迭代器访问

vector 向量

```
//初始化
    vector<int>vec;   //初始化为空
    vector<int>vec(v1);  //用另一个vector来初始化，即构造一个副本
    vector<int>vec(n, i); //大小为n，并全部初始化为元素i （常用）
    vector<int>(n); //构造大小为n的容器，没有初始化里面的元素
    vector<int>{1,2,3,4}; //构造大小为4，并初始化里面的各个元素

    //成员函数
    c.assign(beg,end)  //将[beg; end)区间中的数据赋值给c。
    c.assign(n,elem)    //将n个elem的拷贝赋值给c。
    c.at(idx)   //传回索引idx所指的数据，如果idx越界，抛出out_of_range。    或者通过下标访问
    c.back()    //传回最后一个数据，不检查这个数据是否存在。
    v.begin()   //返回第一个元素的迭代器，相当于一个地址
    v.capacity()    //返回容器中数据个数
    v.clear()   //移除容器中所有元素，大小不变
    v.empty()   //判断容器是否为空
    v.end()     //与begin类似
    v.erase(pos)    //删除pos位置的元素，返回下一个数据的位置
    v.erase(begin,end) //删除[beg,end)区间的数据，传回下一个数据的位置。
    v.front()       //返回第一个元素
    v.push_back()   //在尾部加入一个元素
    v.pop_back()    //删除最后一个元素
    v.size()        //返回容器中实际的元素个数
    operator[];     //返回指定位置的一个引用
```

list 列表

list采用链式存储方法，所以list不能随机存取，但是在list插入和删除元素高效

双向的

list的排序使用成员中的sort的函数

```
#include <bits/stdc++.h>
bool cmp(int a,int b)
{
    return a > b;
}
int main()
{
    std::list<int>l;
    for(int i = 1; i <= 10; i++)l.push_back(i);
    l.sort(cmp);
    for(auto x:l)std::cout<<x<<" ";
}
```

```
#include <list>
int main()
{
    list<int> l;    //创建一个没有任何元素的空 list 容器：
    list<int> l(10)     //创建一个包含 n 个元素的 list 容器：
    list<int> l(10,5)   //创建一个包含 n 个元素的 list 容器，并为每个元素指定初始值。如此就创建了一个包含 10 个元素并且值都为 5 个 values 容器。
    list<int> l1(l2)    //拷贝构造

    int a[] = { 1,2,3,4,5 };
    std::list<int> values(a, a+5);  //拷贝普通数组，创建list容器

    //拷贝其它类型的容器，创建 list 容器
    std::array<int, 5>arr{ 11,12,13,14,15 };
    std::list<int>values(arr.begin()+2, arr.end());//拷贝arr容器中的{13,14,15}

    // 成员函数
    l.begin()   //  返回指向容器中第一个元素的双向迭代器。
    l.end()     //  返回指向容器中最后一个元素所在位置的下一个位置的双向迭代器。
    l.rbegin()  //  返回指向最后一个元素的反向双向迭代器。
    l.rend()    //  返回指向第一个元素所在位置前一个位置的反向双向迭代器。

    l.cbegin()  // 与上述类似，但是返回值是const属性，不能进行修改
    l.cend()
    l.crbegin()
    l.cend()

    l.empty()   //  判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。
    l.size()    //  返回当前容器实际包含的元素个数。
    l.front()   //  返回第一个元素的引用
    l.back()    //  返回最后一个元素的引用
    l.emplace_front()   //在容器头部生成一个元素。该函数和 push_front() 的功能相同，但效率更高。
    l.emplace_back()    //  在容器尾部直接生成一个元素。该函数和 push_back() 的功能相同，但效率更高。
    l.push_back()   //  在容器头部插入一个元素。
    l.pop_front()   //  删除容器头部的第一个元素
    l.emplace()     //  在容器中的指定位置插入元素。该函数和 insert() 功能相同，但效率更高。
    l.insert()      // 在容器中的指定位置插入元素。
    l.insert(begin,val) //
    l.erase()   //  删除容器中一个或某区域内的元素
    l.clear()   //  将一个 list 容器中的元素插入到另一个容器的指定位置。
    l.remove(val)   //  删除容器中所有等于 val 的元素。
    l.unique()  //删除容器中相邻的重复元素，只保留一个。
    l.reverse() //反转容器中元素的顺序。
}
```

set 集合

set 翻译为集合，是一个**内部自动有序**且**不含重复元素**的容器。

```
#include <set>
int main()
{
    //初始化
    set<string> s;  //调用默认构造函数，创建空的 set 容器
    set<typename> s{};  //  可以使用{}进行初始化操作
    //由于set会自动进行升序去重的操作
    set<typename,greater<typename> >s;  //此时，s自动进行降序排列

    //常用的成员函数
    s.insert()  // 插入函数
    s.find(val) //  查找val,成功找到就返回所在位置的迭代器，否则返回end;
    s.erase()   //删除单个元素
    s.size()    // 返回set中的元素个数
    s.clear()   // 返回clear
    s.lower_bound(x)    //返回一个指向当前 set 容器中第一个大于或等于 val 的元素的双向迭代器。如果 set 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
    s.upper_bound(x)    //返回一个指向当前 set 容器中第一个大于 val 的元素的迭代器。如果 set 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
    s.empty()   // 若容器为空，则返回 true；否则 false。
    s.count(x)  //在当前 set 容器中，查找值为 val 的元素的个数，并返回。注意，由于 set 容器中各元素的值是唯一的，因此该函数的返回值最大为 1。

    // 访问
//  该容器只能通过迭代器来进行访问，修改时，为了不破坏容器中的有序性，需要先删除，再插入。
    set<typename>::iterator it;
}
```

map 映射

map是STL的一个关联容器，它提供一对一的hash。

```
#include <bits/stdc++.h>
int main()
{
    //初始化
    // 内部本质是 pair存储的键值对
    map<int,int>m;  //构建了一个空的容器
    map<int,int>m{};    //使用{}进行初始化
    map<int,int>m(m1);  //拷贝构造函数

    //map 的插入方式
    // 定义一个map对象
    map<int, string> mapStudent;
    // 第一种 用insert函数插入pair
    mapStudent.insert(pair<int, string>(000, "student_zero"));
    // 第二种 用insert函数插入value_type数据
    mapStudent.insert(map<int, string>::value_type(001, "student_one"));
    // 第三种 用"array"方式插入
    mapStudent[123] = "student_first";
    mapStudent[456] = "student_second";
```

```
    // c++ 11 新特性
    mapStudent.insert({1,"aaa"});
    mapStudent.insert(pair<int,string>{1,"aaaa"});

    //成员函数
    m.begin()   //这些方法与vector类似，故不再整理
    m.find(key) //查找相同的key值，找到返回迭代器，否则返回end迭代器
    m.lower_bound(key)  // 返回一个指向当前 map 容器中第一个大于或等于 key 的键值对的双向迭代器。
    m.upper_bound(key)  //返回一个指向当前 map 容器中第一个大于 key 的键值对的迭代器。
    m.empty()   // 若容器为空，则返回 true；否则 false。
    m.size()    //  返回当前 map 容器中存有键值对的个数。
    m.clear()   //清空 map 容器中所有的键值对，即使 map 容器的 size() 为 0。
    m.count(key)    //  在当前 map 容器中，查找键为 key 的键值对的个数并返回。
}
```

deque 双端队列

```
#include <deque>
int main()
{
    // 初始化
    deuqe<int> d;   // 创建一个没有任何元素的空 deque 容器：
    deque<int> d(10); // 创建一个具有 n 个元素的 deque 容器，其中每个元素都采用对应类型的默认值：
    deque<int> d(10,5)  //创建一个具有 n 个元素的 deque 容器，并为每个元素都指定初始值
    deque<int> d1(d2);  //拷贝构造
    //拷贝普通数组，创建deque容器
    int a[] = { 1,2,3,4,5 };
    std::deque<int>d(a, a + 5);
    //适用于所有类型的容器
    std::array<int, 5>arr{ 11,12,13,14,15 };
    std::deque<int>d(arr.begin()+2, arr.end());//拷贝arr容器中的{13,14,15}

    // 成员函数
    d.begin();  //  返回首个元素的迭代器
    d.end();    // 返回末尾元素下一个位置的迭代器
    d.rbegin(); //  返回最后由一个元素的迭代器
    d.rend();   // 返回第一个元素的迭代器
    d.cbegin(); //返回第一个元素的迭代器，但是是const类型，不能进行修改
    d.crend();  // 返回第一个元素的迭代器，但是是const属性，不能进行修改元素
    d.size();   // 返回实际元素的个数
    d.empty();  //判断容器是否为空
    d.shrink_to_fit();  //将内存减少到当前元素实际所使用的大小
    d.at()      //使用通过边界检查的索引访问元素
    d.front()   // 返回第一个元素的引用
    d.back()    //返回最后一个元素的引用
    d.push_back();  // 在末尾添加一个元素
    d.push_front(); // 在序列的头部添加一个元素
    d.pop_back();   // 移除容器尾部的元素
    d.pop_front();  // 移除头部元素
    d.insert(pos,val);  //在指定位置添加一个
    d.inset(pos,n,elem);    // 在指定位置添加n个elem数据
```

```
    d.insert(pos,beg,end);  //在pos位置插入【beg,end)区间的数据
    d.erase(it) // 删除一个或一段元素
    d.clear()   // 清除所有数据
    d.emplace_front();  // 在容器头部生成一个元素。和 push_front() 的区别是，该函数直接在容器头部构造元素，省去了复制移动元素的过程。
    d.emplace_back();   //在容器尾部生成一个元素。和 push_back() 的区别是，该函数直接在容器尾部构造元素，省去了复制移动元素的过程。

    // 访问
//  通过迭代器访问
}
```

unordered_set/unordered_multiset 哈希表

应该不考

multimap/multiset

其余用法与map/set相同，区别在于这两种容器内允许存在相同的值

算法

主要定义在：algorithm，functional，numeric中

sort

常常需要compare数组的辅助才能满足需求，默认是从小到大排序

```
#include <algorithm>
bool cmp(T a,T b)// 降序
{
    return a > b;
}
bool cmp(T a,T b)//升序
{
    return a < b;
}
int main()
{
    std::vector<typename T>v;
    T a[101]
    std::sort(v.begin(),v.end(),cmp);// 基于迭代器的排序
    std::sort(a,a+n,cmp)    // 基于数组的排序
}
```

is_sorted

判断元素是否已经有序

```
#include <bits/stdc++.h>
bool cmp(int a,int b)// 自定义排序函数
{
    return a > b;
}
int main()
```

```
{
    std::vector<int>a,b;
    for(int i = 1; i <= 5; i++)a.push_back(i);
    for(int i = 5; i >= 1; i--)b.push_back(i);
    std::cout<<is_sorted(a.begin(),a.end())<<"\n";  // 输出 1
    // 判断是否为降序排列
    std::cout<<is_sorted(b.begin(),b.end(),cmp)<<"\n";  // 输出 1
}
```

find

find函数的返回值是一个迭代器或者地址

将返回的地址减去数组或者容器的首地址就能得到元素在容器中的位置

```
#include <bits/stdc++.h>
int main()
{
    std::vector<int>v;
    for(int i = 1; i <= 5; i++)v.push_back(i);
    int a[10];
    for(int i = 1; i <= 9; i++)a[i] = i;

    //find 函数的实例
    int cnt = find(v.begin(),v.end(), 3)-v.begin();
    std::vector<int>::iterator it = find(v.begin(),v.end(), 3);
    std::cout<<cnt<<"\n";   // 输出 2
    std::cout<<*it<<"\n";   // 输出 3

    int res = std::find(a+1,a+10,5)-a;
    std::cout<<res<<"\n";   // 输出 5
}
```

count

用于查找容器或数组中某一元素出现的次数，返回值为整形

```
#include <bits/stdc++.h>
int main()
{
    std::vector<int>v;
    for(int i = 1; i <= 5; i++)v.push_back(1);
    int a[10];
    for(int i = 1; i <= 9; i++)a[i] = i % 2;

    //count 函数的实例
    int cnt = count(v.begin(),v.end(),1);
    std::cout<<cnt<<"\n";   // 输出5

    int res = std::count(a+1,a+10,0),ans = std::count(a+1,a+10,1);
    std::cout<<res<<"\n"<<ans<<'\n';    // 输出4 5
}
```

merge

merge() 函数用于将 2 个有序序列合并为 1 个有序序列，前提是这 2 个有序序列的排序规则相同（要么都是升序，要么都是降序）。并且最终借助该函数获得的新有序序列，其排序规则也和这 2 个有序序列相同。

```
#include <iostream>
#include <algorithm>
#include <vector>
int main()
{
    std::vector<int>v;
    std::vector<int>result(14); // 需要提前给目标容器限定位置
    for (int i = 1; i <= 5; i++)v.push_back(i);
    int a[10];
    for (int i = 1; i <= 9; i++)a[i] = i;

    // merge函数实例
    std::merge(v.begin(), v.end(), a + 1, a + 10, result.begin());
    for (auto x : result)std::cout << x << " ";//输出 1 1 2 2 3 3 4 4 5 5 6 7 8
9;
}
```

最大公约数

__gcd函数，返回两个参数的最大公约数

```
#include <bits/stdc++.h>
int main()
{
    std::cout<<std::__gcd(4,10)<<"\n"; // 输出 2
}
```

最大最小值

```
#include <bits/stdc++.h>
int main()
{
    std::cout<<std::max(4,5)<<"\n";
    std::cout<<std::min(4,5)<<'\n';
}
```

二分查找函数

lower_bound

返回第一个大于等于的数字的地址

```
#include <bits/stdc++.h>
int main()
{
    std::vector<int>v;
    for(int i = 1; i <= 5; i++)v.push_back(i);
    auto it = lower_bound(v.begin(),v.end(),3)-v.begin();
    std::cout<<it<<'\n';    // 输出2
}
```

upper_bound

返回第一个大于的数字的地址

```
#include <bits/stdc++.h>
int main()
{
    std::vector<int>v;
    for(int i = 1; i <= 5; i++)v.push_back(i);
    auto it = upper_bound(v.begin(),v.end(),3)-v.begin();
    std::cout<<it<<'\n';    // 输出3
}
```

binary_search

用于查找数字是否出现

找到返回1，没找到返回0

```
#include <bits/stdc++.h>
int main()
{
    std::vector<int>v;
    for(int i = 1; i <= 5; i++)v.push_back(i);
    auto b = binary_search(v.begin(),v.end(),3);
    std::cout<<b<<'\n';
}
```

reverse

反转函数：将容器中元素顺序反转

```
#include <bits/stdc++.h>
int main()
{
    std::vector<int>v;
    for(int i = 1; i <= 5; i++)v.push_back(i);
    for(auto x:v)std::cout<<x<<" ";
    std::cout<<"\n";
    reverse(v.begin(),v.end());
    for(auto x:v)std::cout<<x<<" ";
    //输出
//  1 2 3 4 5
//  5 4 3 2 1
}
```

全排列函数

```
#include <bits/stdc++.h>
int main()
{
    int a[4];
    for(int i = 1; i <= 3; i++)a[i] = i;
    do
    {
        for(int i = 1; i <= 3; i++)std::cout<<a[i]<<" \n"[i == 3];
    }while(std::next_permutation(a+1,a+4));
//  输出
//  1 2 3
//  1 3 2
//  2 1 3
//  2 3 1
//  3 1 2
//  3 2 1
}
```

memset

初始化函数，用于将数字赋值为0或-1

```
#include <bits/stdc++.h>
int main()
{
    int a[5];
    for(int i = 0; i < 5; i++)std::cout<<a[i]<<" \n"[i == 4];
//  输出: 8 0 10 0 14882528
    memset(a,0,sizeof(a));
    for(int i = 0; i < 5; i++)std::cout<<a[i]<<" \n"[i == 4];
//  输出: 0 0 0 0 0
}
```

unique

将不重复的元素放到最前面

返回值是不重复元素的下一个元素，配合erase可以完全删除

```
#include <bits/stdc++.h>
int main()
{
    std::vector<int>v;
    for(int i = 1; i <= 5; i++){
        for(int j = 1; j <= i; j++){
            v.push_back(i);
        }
    }
    for(auto x:v)std::cout<<x<<" ";
    std::cout<<"\n";
    std::unique(v.begin(),v.end());
```

```
    for(auto x:v)std::cout<<x<<" ";
//  输出
//  1 2 2 3 3 3 4 4 4 4 5 5 5 5 5
//  1 2 3 4 5 3 4 4 4 4 5 5 5 5 5
}
```

transform

将某操作应用于指定范围的每个元素

```
#include <bits/stdc++.h>
int main()
{
    std::string s = "abcdefg";
    transform(s.begin(),s.end(),s.begin(),::toupper);//转大写
    std::cout<<s<<"\n";
    transform(s.begin(),s.end(),s.begin(),::tolower);//转小写
    std::cout<<s<<"\n";
}
```

strcmp/strcpy

字符串比较函数和字符串复制函数

strcmp:返回-1，a<b;返回0，a=b;返回1，a > b

strcpy（a,b）：将b拷贝到a

```
#include <bits/stdc++.h>
int main()
{
    char a[101];
    char b[101];
    std::cin>>a>>b;
    std::cout<<strcmp(a,b)<<"\n";
    char c[101];
    strcpy(c,a);
    std::cout<<strcmp(a,c)<<"\n";
}
```

copy

与strcpy类似，但用于容器

```
    //declaring & initializing an int array
    int arr[] = { 10, 20, 30, 40, 50 };

    //向量声明
    vector<int> v1(5);

    //复制数组元素到向量
    copy(arr, arr + 5, v1.begin());

    Output:
    //如果我们打印值
    arr: 10 20 30 40 50
    v1: 10 20 30 40 50
```

for_each

与transform函数类似

可与lambda表达式结合

```
#include <iostream>
#include <algorithm>
#include <vector>
int main()
{
    std::vector<int> vec;
    size_t count = 10;
    for (size_t i = 0; i < count; i++)
    {
        vec.emplace_back(i + 1);
    }
    std::for_each(vec.begin(), vec.end(), [&](int value)
                  { std::cout << value << std::endl; });
    return 0;
}
```

迭代器

迭代器（iterator）是一种可以遍历容器元素的数据类型。迭代器是一个变量，相当于容器和操纵容器的算法之间的中介。C++更趋向于使用迭代器而不是数组下标操作，因为标准库为每一种标准容器（如vector、map和list等）定义了一种迭代器类型，而只有少数容器（如vector）支持数组下标操作访问容器元素。可以通过迭代器指向你想访问容器的元素地址，通过*x打印出元素值。这和我们所熟知的指针极其类似。

```
std::vector<int> ::iterator it;     //it能读写vector<int>的元素
std::vector<int>::const_iterator it;//it只能读vector<int>的元素，不可以修改
vector<int>中的元素
std::vector<int>::reverse_iterator it = v.rbegin() // 逆序迭代
```

使用方法均类似

仿函数

仿函数（Functor）又称为函数对象（Function Object）是一个能行使函数功能的类。

通过重载（）进而行使某些功能

```
class StringAppend {
public:
    explicit StringAppend(const string& str) : ss(str){}
    void operator() (const string& str) const {
         cout << str << ' ' << ss << endl;
    }
private:
    const string ss;
};

int main() {
    StringAppend myFunctor2("and world!");
    myFunctor2("Hello");
}
```

适配器

priority_queue 优先队列

本质是容器的适配器，因此没有迭代器！！！

```
#include <queue>
int main()
{
    // 初始化
    priority_queue<int>q;    // 默认从大到小排列的一个vector容器
    //本质上是
    priority_queue<int,vector<int>,less<int>>q;
    // 从小到大排列可以重载 < 符，或者使用
    priority_queue<int,vector<int>,greater<int>>q;

    //使用普通数组
    int values[]{4,1,3,2};
    std::priority_queue<int>copy_values(values,values+4);//{4,2,3,1}
    //使用序列式容器
    std::array<int,4>values{ 4,1,3,2 };
    std::priority_queue<int>copy_values(values.begin(),values.end());//{4,2,3,1}

    // 成员函数
    q.empty()   //  如果 priority_queue 为空的话，返回 true；反之，返回 false。
    q.size()    //  返回 priority_queue 中存储元素的个数。
    q.top()     //  返回 priority_queue 中第一个元素的引用形式。
    q.push()    //  插入一个元素，会自动进行排序
    q.pop()     //  移除第一个元素
}
```

stack 栈

本质上是一种容器适配器，因此没有迭代器！！！

具有后进先出的特性，适合进行某些模拟

```
#include <stack>
int main()
{
    // 初始化
    stack<int>s;    // 空的stack,底层通过deque来实现
    stack<int,vector<int>>s1;   // 空的stack,底层通过vector实现
    // 拷贝构造
    vector<int>v;
    stack<int,vector<int>>s(v);     // 通过拷贝构造，给s赋值成v
    // 适配器初始化另一个适配器
    std::list<int> values{ 1, 2, 3 };
    std::stack<int, std::list<int>> my_stack1(values);
    std::stack<int, std::list<int>> my_stack=my_stack1;

    //成员函数
    s.empty();      //  当 stack 栈中没有元素时，该成员函数返回 true；反之，返回 false。
    s.size();   //返回 stack 栈中存储元素的个数。
    s.top();    //  返回一个栈顶元素的引用，类型为 T&。如果栈为空，程序会报错。
    s.push();   // 先复制 val，再将 val 副本压入栈顶。这是通过调用底层容器的 push_back() 函数完成的。
    s.pop();    //  弹出栈顶元素。
}
```

queue 队列

本质上是一种容器适配器，因此没有迭代器！！！

具有先进先出的特性，适合进行某些模拟

```
#include <queue>
int main()
{
    //初始化
    queue<int>q;    // 创建一个空的queue，底层默认是deque
    queue<int,vector<int>>q;    // 创建一个底层是vector的空queue
    //初始化
    std::deque<int> values{1,2,3};
    std::queue<int> my_queue(values);
    // 初始化
    std::deque<int> values{1,2,3};
    std::queue<int> my_queue1(values);
    std::queue<int> my_queue(my_queue1);

    //成员函数
    q.empty();  //  如果 queue 中没有元素的话，返回 true。
    q.size();   //  返回 queue 中元素的个数。
    q.front();  //  返回 queue 中第一个元素的引用。如果 queue 是常量，就返回一个常引用；如果 queue 为空，返回值是未定义的。
```

```
    q.back();    //  返回 queue 中最后一个元素的引用。如果 queue 是常量，就返回一个常引用；如果 queue 为空，返回值是未定义的。
    q.push(T);  //在 queue 的尾部添加一个元素的副本。这是通过调用底层容器的成员函数 push_back() 来完成的。
    q.pop();     //  删除 queue 中的第一个元素。


}
```

空间配置器

c++11 起的新标准

可以在类的内部进行初始化操作

下述代码现在可以编译

```
class A{
public:
    int a = 5;
    int b = 4;
    double c = 5.0;
};
int main()
{
    A a;
    std::cout<<1<<"\n";
    std::cout<<a.a<<"\n";
}
```

使用{}对容器、数组、结构体进行初始化或者赋值

```
// 对容器map进行插入值
map.insert({T,T});

//对结构体进行赋值
struct Node{
    int a,b,c;
}node[10];
    node[1] = {1,2,3};
```

lambda表达式

匿名函数

```
//  使用方法一，替代sort排序函数中cmp函数
    std::vector<int>v;
    for(int i = 1; i <= 5; i++)v.push_back(i);
    // v 中元素从小到大排列
    std::sort(v.begin(),v.end(),[](int i,int j){
        return i > j;
    });
    for(auto x:v)std::cout<<x<<" "; // 现在是从大到小排列
```

```
//  使用方法二
#include <bits/stdc++.h>
int main()
{
    int a = 5,b = 6;
    auto change = [](int &a,int &b){a = 10,b = 11;};
    std::cout<<a<<" "<<b<<"\n"; //  输出 5 6
    change(a,b);
    std::cout<<a<<" "<<b<<"\n"; // 输出 10 11
}
```

自动类型推断

auto关键字，c++ 11 起加入的新特性，自动推断变量类型

```
auto a = 5; //a 为 int
auto p = &a;  // p 为 int* 类型
int a = 10;       // a为 int类型
auto b = a;       // b由a推理得到 int类型
auto c = 'a';     // c为 char类型
auto it = v.begin();     // it 为迭代器类型
auto change = [](int &a,int &b){a = 10,b = 11;};  // 与lambda表达式使用，可理解为函数调用方式为：change(a,b)

vector<int>v
for(auto x:v)   // 与范围for连用，  x 为容器类的数据类型,此时为int型
```

Range-based for循环

范围for，自动遍历容器内部的数据

```
// vector 为例
    std::vector<int>v(10);
    for(auto &x:v)std::cin>>x;
    for(auto x:v)std::cout<<x;
// map 为例
    std::map<int,int>m;
    for(int i = 1; i <= 10; i++)m[i] = 1;
    for(auto x:m)std::cout<<x.first<<" "<<x.second<<"\n";
    // 在c++17 中可以使用如下的语法（学校OJ仅支持到c++14，不能使用）
/*
    for(auto [x,y]:m)std::cout<<x<<" "<<y<<"\n";
*/
```