第一次进度汇报

一.C/C++薄弱复习

一、sizeof

1. 计算变量或类型的大小
   - long: 4字节 double: 8字节 float: 4字节
   - long和int：windows64位下都是4字节，Linux64位long是8字节
2. 对于**数组**，返回整个数组的大小
3. 对于**指针**，返回指针本身的大小，而不是指向的对象的大小
4. 对于**引用**，返回引用所绑定对象的大小
5. 对于**函数**，返回函数的代码大小，而不是函数调用的结果

二、static关键字

1. **静态全局变量**
   - 限制该变量的作用域仅在定义它的源文件内可见（内部链接）
   - 防止在其他文件中通过extern声明来访问该变量
   - 目的：模块化设计
2. **修饰局部变量**
   - 改变变量的存储位置，从栈区移至全局区
   - 变量的生命周期变为整个程序运行期间，但作用域仍局限于函数内
   - 只在第一次执行到该变量定义时初始化一次
3. **修饰函数**
   - 限制函数的作用域仅在定义它的源文件内可见（内部链接）
   - 防止在其他文件中调用该函数
4. **在C++中修饰类成员**
   - 静态成员变量：不属于对象，而是属于整个类，所有对象共享
     - 如果静态成员变量通过某个函数改变了，所有对象的这个变量都会被改变
     - 类内声明，类外初始化 `类名::变量名 = 值`
   - 静态成员函数：不需要对象即可调用，只能访问静态成员
     - 不需要对象即可调用：可以直接 `类名::函数名` 调用
     - 这个函数只能访问静态成员(静态成员变量和静态成员函数)

三、面向对象

> 熟悉的省略

1. **struct 和 class 区别**
   - 默认访问权限：struct 默认是 public，class 默认是 private
2. **多种初始化方式：**
   - 显示法中不熟的有：

```
Person p2 = Person(10); //有参构造  将匿名对象起了一个名称p2
Person p3 = Person(p2); //创建一个对象，这个对象调用的是拷贝构造
```

- 隐式法中不熟的有：

```
Person p4 = 10; // 相当于写了 Person p4 = Person(10); 有参构造
Person p5 = p4; // 拷贝构造
```

- 匿名对象: `Person(10);`
  - 应用：作为函数参数只用一次、直接调用对象的成员函数 `Person(10).showAge();`
  - 例子1：在一些支持链式调用的API设计中，匿名对象可以使代码更简洁。

```
class Calculator {
public:
    Calculator(int val) : value_(val) {}
    Calculator add(int n) { return Calculator(value_ + n); }
    Calculator subtract(int n) { return Calculator(value_ - n); }
    int getResult() { return value_; }
private:
    int value_;
};

int main() {
    // 匿名对象实现链式调用
    int result = Calculator(5).add(3).subtract(2).getResult(); // result = 6
    return 0;
}
```

  - 例子2：确保资源在使用完毕后立即释放，避免资源泄漏。

```
class FileHandler {
public:
    FileHandler(const string& filename) {
        file = fopen(filename.c_str(), "r");
    }
    ~FileHandler() {
        if (file) fclose(file); // 确保文件被关闭
    }
    void readContent() { /* 读取文件内容 */ }
private:
    FILE* file;
};

int main() {
    // 匿名对象确保文件在读取后立即关闭
    FileHandler("data.txt").readContent();
    return 0;
}
```

3. **拷贝函数调用时机**

```
Person doWork2() // 返回值类型为Person对象
{
    Person p1; // 局部对象
    cout << (int *)&p1 << endl;
    return p1; // 以值的方式返回一个拷贝的对象给外部，拷贝出一个对象p1'与原对象p1不一样,调用拷贝构造函数

    // 程序运行结束，释放原p1，调用析构函数
}

void test03()
{
    Person p = doWork2(); // 这里没有调用拷贝构造函数，直接用p接收拷贝对象p1’,直接在p对象这块内存上构造对象p1
    //以上是编译器优化的结果
    cout << (int *)&p << endl;
    // 程序运行结束，释放拷贝的对象p1',调用析构函数
}
```

4. **构造函数**

   i. 调用规则和编译器提供规则

- 默认情况下，C++编译器至少给一个类添加3个函数。
  - 1.默认构造函数(无参，函数体为空)
  - 2.默认析构函数(无参，函数体为空)
  - 3.默认拷贝构造函数，对属性进行值拷贝
- 构造函数调用规则如下：
  - 1.如果用户定义有参构造函数，C++不再提供默认无参构造，但是会提供默认拷贝构造。
  - 2.如果用户定义拷贝函数，C++不会再提供其他构造函数。

5. **静态成员相关**

- 静态成员变量存储在全局区。非静态成员变量存储在对象内存上。
- **非静态成员函数也不存储在对象内存上，和静态成员函数一样，都存储在代码区。**：目的：节省内存空间
  - 所有对象共享一份函数代码
  - 普通成员函数也可以访问静态成员变量，但是静态成员函数只能访问静态成员变量。

6. **const修饰成员函数**

- 常函数：*本质上是this变为指针常量，指针指向的内容不会被改变，类型 const this;**
  - 成员函数后加const后我们称为这个函数为常函数。
  - 常函数内不可以修改成员属性。
  - 成员属性声明时加关键字mutable后，在常函数中依然可以修改。
- 常对象：
  - 声明对象前加const称该对象为常对象。
  - 常对象不可以修改普通成员属性。
  - 成员属性声明时加关键字mutable后，在常对象中依然可以修改。

7. **友元**

- 友元的目的是让一个函数或者类 访问另一个类中私有成员。
- 友元的关键字为friend。
- 友元的三种实现：
  - 全局函数做友元

```
//goodfriend全局函数是源类好朋友，可以访问源类中私有成员
//在源类的全局区声明：
friend void goodfriend(Buiding* buiding);
```

  - 类做友元

```
//Friend_class类是本类的好朋友，可以访问本类的私有成员
friend class Friend_class;
//在Friend_class类里面可以new出来一个源类对象，则在Friend_class类里面可以访问源类的私有成员
```

- 成员函数做友元

```
//告诉编译器，Friend_class类下的visit成员函数作为本类的好朋友，可以访问私有成员
//在原类的全局区声明：
friend void Friend_class::visit();
```

8. **现代编译器返回值优化：**

> 面临函数创建临时对象，返回临时对象时，并且用临时对象初始化另一个对象时，编译器会进行返回值优化，直接将临时对象的值赋值给另一个对象，而不是创建一个新的临时对象。

- 一些使用场景：
  - 工厂函数：
    a. **封装性**：通过一个专用函数来创建对象，**隐藏了具体的创建逻辑**。调用者只需要调用工厂函数，不需要关心对象是如何被构造的。
    b. **灵活性**：可以根据参数动态决定返回哪种类型的对象（多态场景），或者在创建前后执行额外逻辑（如缓存、日志、验证等）。
    c. **对象创建的集中管理**：集中管理对象的创建过程，当创建逻辑需要修改时，只需修改工厂函数一处。
  - 运算符重载
  - 链式调用中的对象返回

```
int result = Calculator(5).add(3).multiply(2).getValue();
```

9. **继承**

-

特性	public继承	protected继承	private继承
继承关系描述	表示“是一个”(is-a）关系	表示“实现一个”关系	表示“用...实现”关系
基类public成员	在派生类中为public	在派生类中为protected	在派生类中为private
基类protected成员	在派生类中为protected	在派生类中为protected	在派生类中为private
基类private成员	在派生类中不可访问	在派生类中不可访问	在派生类中不可访问
派生类对象对基类成员的访问	可以访问public成员	不可访问任何成员	不可访问任何成员
派生类的派生类	可以访问基类的public和protected成员	可以访问基类的protected成员	无法访问基类的任何成员
最常用场景	一般类继承，保留接口完整性	工具类继承，隐藏部分接口	实现细节继承， 完全封装基类

- **同名成员处理**：
  - 子类对象加作用域可以访问到父类同名成员。
    `子类名.父类名::成员名`
  - 当子类与父类拥有同名的成员函数，子类会隐藏父类中所有同名成员函数(有参、无参)，加作用域才可以访问到父类中同名函数
    `子类名.父类名::成员函数名(参数列表)`
- **同名静态成员处理**：
  - 与以上的同名成员处理一致。
  - 但是静态成员函数和变量可以直接通过 `类名::成员名` 访问。因此，出现同名时，子类会隐藏父类中所有同名成员函数(有参、无参)，加作用域才可以访问到父类中同名函数。
    `Son::Base::func();Son::Base::func(100);`
- **菱形继承**：

- 两个派生类继承同一个基类，又有某个类同时继承这两个派生类，这就是菱形继承。
- 问题：
  - 两个派生类继承同一个基类，会导致数据冗余和二义性。
  - 某个类同时继承这两个派生类，继承基类了两次，其实只需要一份。
- 方法：虚继承
  - 继承之前 加上关键字 virtual 变成虚继承
  - 虚继承后，Animal类称为虚基类，两个子类(B,C)虚继承虚基类(A);
  - 当创建孙子成员(D)时，内存中有全部的BC成员，但是只有一份A成员。
  - 最后的孙子类访问子类中继承自父类的同一个成员时，会访问到虚基类中的成员，这个成员在D中只有一份。
- 如果不这么做：D类中就有多个A成员，会报错。

10. **多态**：
    i. **多态分为两类：**
    - 静态多态：函数重载和运算符重载属于静态多态，复用函数名。静态多态的函数地址早绑定，编译阶段确定函数地址。
    - 动态多态：派生类和虚函数实现运行时多态。动态多态的函数地址晚绑定，运行阶段确定函数地址

    ii. **静态多态和动态多态解释：**
    - 静态多态：

    a. 子类成员函数与父类成员函数相同，子类成员函数会覆盖父类成员函数。

    b. 当调用doSpeak(cat)时，尽管传入的是Cat对象，但由于doSpeak函数参数是Animal&类型，编译器在编译阶段就确定了要调用Animal类的speak()方法（地址早绑定）

    - 动态多态：

    a. 父类函数被声为虚函数，子类重写了该函数。

    b. 当调用doSpeak(cat)或doSpeak(dog)时，编译器不再在编译时确定调用哪个方法，而是在运行时根据传入对象的实际类型来决定（地址晚绑定）
11. **虚函数用于实现多态**
    i. **基础使用**：
    - `virtual void f();`
    - 子类里可以重写父类的虚函数：`void f() override;` 或者 `virtual void func() override {}`
    - 解决了问题：如果定义了一个函数参数是 `(父类 &a)`，函数里调用父类的虚函数，传参是子类，那么则不会调用父类的那一个虚函数而是子类重写的函数
    - 将父类的析构函数声明为虚函数。
    - `virtual 返回值类型 函数名 (参数列表) = 0;`, 纯虚函数：
      - 当一个类中有纯虚函数，这个类就称为**抽象类**。
      - 纯虚函数是在基类中声明的，没有具体的实现，只有声明。
      - **抽象类无法实例化对象**，子类必须重写纯虚函数，否则也会成为抽象类。

    ii. **原理**：
    - 虚函数表：存储该类所有虚函数地址的静态数组。
      - **同一个类的所有对象共享同一份虚函数表**
      - 子类在创建虚函数表时，会先复制父类的虚函数表，**若子类的某个函数被重写，那么子类的相应位置上的函数指针被替换为子类的函数指针，**子类新增的虚函数会被添加到虚函数表的末尾
    - 虚函数指针（vptr）：每个包含虚函数的类的对象在创建时，会在其内存布局的开头自动添加一个虚函数指针，用于指向所属类的虚函数表。这个指针的内存是位于对象里的，而他指向的虚函数表是在静态存储区的。
    - 动态绑定：
      - 父类的指针可以指向子类的地址：`Father* ptr2 = &son;`
      - **也就是用父类指针去解析子类内存。**如果在子类加上一些父类没有的非虚拟成员变量和函数时，用ptr2访问不到新增的。
      - 这样设计目的：掩盖父类函数的复杂实现，人只用从子类直接调父类函数即可
    - 析构函数：
      - 同样 `Father* ptr2 = &son; delete ptr2;` 这时只会析构父类的那一部分，子类新的不会被析构。
      - 因此设计为虚拟析构函数，解决**通过父类指针释放子类对象**。这时 `delete ptr2` 就先调用子类的析构函数，析构新的部分，再由编译器析构父类部分。

- 构造函数
  - 派生类构造函数必须调用父类构造函数，当创建派生类对象时，必须先初始化其基类部分
  - 一般用初始化列表调用父类构造函数：`Son(int a, int b) : Father

四、文件读写

1. **操作文件的三大类**
   - ofstream：写操作
   - ifstream：读操作
   - fstream：读写操作
2. **包含头文件：** `#include <fstream>`
3. **读写txt文本文件：**
   - 代码流程：

```
//写入
ofstream ofs; // 创建流对象

// 3、指定打开方式
ofs.open("D:/VSCODEVS/txt_file_manipulate/test.txt", ios::out); // 当没有指定某盘路径时，创建的文件在该项目文件的路径下

// 4、写内容
ofs << "姓名：张三" << endl;
ofs << "性别：男" << endl;
ofs << "年龄：18" << endl;

// 5、关闭文件
ofs.close();
//读取
ifstream ifs; // 创建流对象

// 3、打开文件，并且判断是否打开成功
ifs.open("D:/VSCODEVS/txt_file_manipulate/test.txt", ios::in);

if (!ifs.is_open())
{
    cout << "文件打开失败" << endl;
    return;
}
char c;
while ((c = ifs.get()) != EOF) // 每一次读取一个字符,字符放入c
{                              // EOF表示读取到文件尾
    cout << c;
}

// 5、关闭文件
ifs.close();
```

4. **读写二进制文件：**

```
    // 1、创建流文件
    ofstream ofs;

    // 2、打开文件
    ofs.open("D:/VSCODEVS/txt_file_manipulate/person.txt", ios::out | ios::binary); // 二进制的方式写文件

    // ofstream  ofs.open("person.txt",ios::out | ios::binary)  也可以两步合成一步

    // 3、写文件
    Person p = {"张三", 18};
    ofs.write((const char *)&p, sizeof(Person)); // 对p取地址，然后强转为常量指针
    // 原型: offstream& write(const char* s, streamsize n);
    //char*是一个字节，又用sizeof计算有多少个字节，这样保证完整输出
    ofs.close();
    // 1、创建流对象
    ifstream ifs;

    // 2、打开文件
    ifs.open("D:/VSCODEVS/txt_file_manipulate/person.txt", ios::in | ios::binary); // 二进制的方式读文件

    if (!ifs.is_open())
    {
        cout << "文件打开失败" << endl;
        return;
    }

    // 3、写文件
    Person p;
    ifs.read((char *)&p, sizeof(Person));

    cout << "姓名: " << p.m_Name << " 年龄: " << p.m_Name << endl;
    ifs.close();
```

五、模板

1. `template <typename T>` 的typename可以替换为class
2. 当函数时模板函数时，调用时如果没有指明类型(没有写< T >)，编译器会报错

```
template<class T>
T myAdd02(T a, T b)
{
    return a + b;
}
//cout << myAdd02(a, c) << endl; //报错，编译器不知道把T转为整型还是字符型
cout << myAdd02<int>(a, c) << endl; //明确告诉编译器T是int类型，不是int类型的，自动转换为int类型
```

1. **既然提供了函数模板，最好就不要提供普通函数，否则容易出现二义性。**
2. 类模板:
    - 类模板没有自动类型推导的使用方式。
    - b. 类模板在模板参数列表中可以有默认参数。 `template<class NameType,class AgeType = int>`
3. 模板可以是自己创建的类，也可以是STL提供的类。
4. 类模板做函数参数:
    - 三种方式:
    - 传入指定类型: `void func(Person<string, int>& p);`

- 参数模板化： `template<class T1, class T2>void prinfunc(Person<T1, T2>& p);`
- 整个类模板化： `template<class T>void printPerson3(T& p);`
- 类模板和函数模板的区别：
  - 类模板没有自动类型推导的使用方式。

5. 类模板与继承

```
template<class T>
class Base
{
    T m;
};

//class Son :public Base // 报错，必须要知道父类中T类型，才能继承给子类，因为编译器不知道给子类多少个内存空间，如果T是int型给1个字节，
class Son:public Base<int>
{

};
//如果想灵活指定父类中T类型，子类也需要变类模板
template<class T1,class T2>
class Son2 :public Base<T2>  //T2给了父类
{
public:
    Son2()
    {
        cout << "T1的类型为： " << typeid(T1).name() << endl;
        cout << "T2的类型为： " << typeid(T2).name() << endl;
    }
    T1 obj; //T1 给了子类
};
```

8. 头文件编译问题：
   - 将类的声明和实现放在一个文件里，**名称后缀是.hpp**，然后在头文件里加上 `#pragma once` ，防止头文件被重复包含。
9. 全局函数做友元实现位置问题：
   - 类内实现：正常实现
   - 类外实现：
     - 全局函数做友元：
       - 声明： `friend void goodfriend<>(Person<T1, T2>& p);` <>表示告诉编译器printPerson2是一个函数模板，这是对之前声明的printPerson2函数模板的引用，再去调用时就能找到实现
       - 实现：在类的具体实现之前要先声明一次模板和类，然后实现这个全局函数
10. 模板局限性：
    - 学习模板并不是为了写模板，而是在STL中能够运用系统提供的模板。
    - 使用了模板就不能在函数里访问类的成员(因为编译器不知道你的模板进来的是什么内容)，函数就无法实现相应功能

六、STL

1. **deque和list：**
   - deque：可以理解为分段存储的vector，可以高效的在头尾插入删除元素，也可以随机访问，但是在中间插入删除元素效率较低，并且会导致迭代器失效。
     - **因此我们用它作为stack, queue, priority_queue的底层容器。只在首尾进行操作**
   - list：支持高效的在任意位置插入和删除元素，但是不支持随机访问，插入和删除只会影响操作元素的迭代器，只有删除才会使迭代器失效
     - 他是一个双向循环链表，最后节点的next指向第一个节点，第一个节点的prev指向最后一个节点。

2. **set容器：**
   - 所有元素都会在插入时自动被排序。默认从小到大
   - 自定义数据结构的排序顺序
     - 如果是普通(Int)，用一个类实现一个函数，在类里面重载一个 `bool operator()(const T& a, const T& b){return a > b;}`
     - 如果是自定义数据结构，用一个类实现一个函数，在类里面重载一个 `bool operator()(const T& a, const T& b){return a.age > b.age;}`
     - 当然，自定义数据结构在声明时也要包含这个函数(或者时某个包含这个函数的类) `set<Person, comparePerson>s1;`
   - multiset容器：
     - 允许容器中有重复的元素。
     - map容器可以根据key值快速找到value值。
3. **map容器:**

   > map容器中的所有元素都是pair。pair中第一个元素为key(键值)，起到索引作用，第二个元素为value(实值)。**所有元素都会根据元素的键值自动排序，默认按key从小到大**

   i. pair两种创建方式：
   - `pair<type,type> p (value1, value2);`
   - `pair<type,type> p = make_pair(value1,value2);`

   ii. 相关接口：

```
// 插入元素
m.insert(pair<int, int>(1, 10));
// 打印元素
for (map<int,int>::iterator it = m.begin();it!=m.end();it++)
{
    cout << "key = " << it->first << " value = " << it->second << endl;
}
// 交换
m1.swap(m2);
// 删除
m.erase(3);  //按key删除
m.erase(m.begin()); //按迭代器删除
```

   iii. 仿函数指定排序规则：同set

七、仿函数

1. 函数对象（仿函数）：
   - 函数对象（仿函数）是一个类，重载了函数调用运算符()。
   - 函数对象可以像函数一样被调用，并且可以保存状态。
   - 函数对象可以作为算法的参数，也可以作为容器的排序规则。

```
class MyAdd
{
public:
    int operator()(int v1, int v2)
    {
        return v1 + v2;
    }
};
class MyPrint
{
public:
    MyPrint()
    {
        this->count = 0;
    }

    void operator()(string test)
    {
        cout << test << endl;
        this->count++;
    }
    int count;  //内部自己状态
};
```

八、谓词

1. 一元谓词：operator()接收一个参数

```
class GreaterFive
{
public:
    bool operator()(int val)
    {
        return val > 5;
    }
};
vector<int> v;
vector<int>::iterator it = find_if(v.begin(), v.end(), GreaterFive());
```

2. 二元谓词：operator()接收两个参数，可以用于sort函数的排序规则

```
class MyCompare
{
public:
    bool operator()(int val1,int val2)
    {
        return val1 > val2;
    }
};
sort(v.begin(), v.end(), MyCompare());
```

九、算法

1. 遍历算法：
    i. for_each遍历算法

```
class print02
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};
vector<int> v;
for_each(v.begin(), v.end(), print02());
//如果是函数，第三个参数只需要传函数名即可
```

ii. transform遍历算法；功能描述：搬运容器到另一个容器中。

```
vector<int>v;
vector<int>vTarget;  //目标容器
vTarget.resize(v.size());  //目标容器，需要提前开辟空间
transform(v.begin(), v.end(), vTarget.begin(), Transform());
class Transform
{
public:
    int operator()(int v)
    {
        return v;
    }
};
```

二、计算机知识

一、内存四个区域

1. **代码区（Text Segment）**
   - 存放机器指令和常量
   - 由操作系统管理
   - 只读，不可修改
2. **全局区（Global/Static Segment）**
   - 存放全局变量、**静态变量(static修饰的变量)**和常量（如字符串常量）
   - 由编译器自动分配和释放
   - 程序启动时分配内存，程序结束时释放
   - 变量的生命周期贯穿整个程序运行期间
3. **栈区（Stack）**
   - 用于存储函数的局部变量、函数参数、返回地址和返回值等
   - 由编译器自动分配和释放，函数结束时自动释放
   - 栈是一种后进先出（LIFO）的数据结构
   - 空间小，容易发生溢出，分配释放速度快
4. **堆区（Heap）**
   - 动态分配的内存，需要人手动申请和释放
   - 通过new/malloc等函数申请，delete/free等函数释放
   - 空间大，但是速度慢
   - 容易发生内存泄漏或空悬指针
5. **代码区常量和全局区常量**

特性	代码区常量	全局区常量
存储内容	主要是字符串字面量、程序指令	全局const变量、静态const变量、static变量
访问方式	通过指针间接访问	可以直接通过变量名访问
内存分配时机	编译时确定并分配	程序启动时分配
修改保护	通常是只读的，不可修改	只读的，不可修改
生命周期	程序运行期间	程序运行期间

二、生命周期和作用域

1. **生命周期**：变量从创建到销毁的时间段
   - 全局变量：程序运行期间一直存在
   - 局部变量：函数执行期间存在，函数结束时销毁
2. **作用域**：变量在程序中的可见范围
   - 全局变量：整个程序
   - 局部变量：当前函数或代码块
3. **static修饰局部变量时的生命周期和作用域**
   - 当static修饰局部变量时，它会同时改变变量的生命周期和存储位置，但不会改变作用域：
     - 生命周期
       - 普通局部变量：生命周期从函数调用开始，到函数返回结束
       - static局部变量：生命周期从程序启动开始，到程序结束结束，与全局变量相同
     - 作用域
       - 普通局部变量：作用域限于定义它的函数或代码块内
       - static局部变量：作用域仍然限于定义它的函数或代码块内
     - 存储位置
       - 普通局部变量：存储在栈区
       - static局部变量：存储在全局/静态存储区
4. **深拷贝和浅拷贝**
   - 浅拷贝：单纯的复制，只复制指针本身，不复制指针指向的内容。
     - 问题：一个对象复制了本对象的一个指针，两个对象同时销毁时同一段内存就会被释放两次，导致程序崩溃。
   - 深拷贝：复制指针指向的内容，而不是指针本身。
     - 问题：如果对象中包含指针，那么深拷贝需要手动分配内存，并且需要考虑释放内存的问题。但是比浅拷贝安全。
5. **类中有类的析构和构造函数调用顺序**
   - B类中有对象A作为成员，A为对象成员，那么当创建B对象时，A与B的构造和析构的顺序是：

   i. 当其他类对象作为本类成员，构造时候先构造其他类对象，在构造自身。

   ii. 当其他类对象作为本类成员，析构的顺序与构造相反，想析构自身，再析构其他类对象。

三、计算机结构：

1. **冯诺伊曼结构**
   - 计算机由运算器、控制器、存储器、输入设备和输出设备组成。
   - 存储器
     - 寄存器（CPU 内部）：你手里拿着的一页纸，上面写着正在处理的最重要信息，取用速度极快，但容量少（几十到几百字节）。
     - L1/L2/L3 缓存（CPU Cache）：书桌抽屉，放着近期需要的书页，速度很快，容量比寄存器大（KB ~ MB）。
     - 内存（RAM）：书架（桌面），放着正在读的整本书，访问速度比缓存慢，但容量大（GB）。
     - 硬盘（HDD/SSD/NVMe）：图书馆的大仓库，存放很多书，但取书要更久（毫秒级）。
     - 长期存储/云盘：更远的仓库，取用延迟更高。

2. **CPU**
   - 作为计算机系统的运算和控制核心，是信息处理、程序运行的最终执行单元
   - CPU 内部同时包含 运算单元（ALU/FPU） 和 控制单元（CU）。所以 CPU 既是运算器，也是控制器。
   - 25%的ALU(运算单元)、有25%的Control(控制单元)、50%的Cache(缓存单元)、
   - 进行少量复杂的计算
3. **GPU**
   - GPU 几乎不负责复杂的控制（流程判断、任务调度等还是 CPU 来做）。所以GPU可以看作运算器。
   - 专门用于图形处理和渲染，90%的ALU(运算单元)，5%的Control(控制单元)、5%的Cache(缓存单元)
   - 进行大量简单的计算
4. **操作系统**
   - 管理和控制计算机硬件与软件资源的系统软件
   - 直接操作硬件资源，对于多个请求能够合理公平的分配，一方面提供了操作硬件资源提供了接口，为应用软件的开发提供方便。
5. **驱动程序**
   - 硬件设备（如显卡、打印机、网卡等）本身只能识别底层的电信号或机器指令，不同设备有不同的控制方式。操作系统通过调用驱动程序来与硬件交互，这样用户和应用软件就不必直接面对复杂的硬件指令。

三、程序实现

1. **库文件**
   - 已经写好的代码集合（别人或你自己提前编译好的）。
   - 分为：
     - 静态库（Windows: .lib，Linux: .a）
       - 在编译/链接时，库的内容被拷贝进最终的 EXE。
       - 优点：运行时不需要额外文件；缺点：EXE 变大。
       - 可移植性：可执行文件是独立的，不需要依赖外部库文件。可以轻易地在不同的系统上运行。
     - 动态库（Windows: .dll，Linux: .so）
       - 程序运行时才加载库，多个程序可共享。
       - 链接时只验证动态库中函数是否存在，运行时，由操作系统的动态链接器去指定路径找到文件并加载到内存中。
       - 优点：节省内存，方便更新；缺点：运行需要 DLL 文件存在。
       - 多进程多程序可以共享一个动态库，只需要加载库一次，节省内存。
2. **编译过程：预处理、编译、汇编、连接**
   - **预处理**：预处理在编译之前，对源代码文件进行文本层面的处理。它处理所有以 # 开头的预处理指令。
     - 头文件包含：将 #include <stdio.h>直接替换为这些头文件的实际内容。这可能会递归地包含很多内容，所以 .i 文件通常会变得非常大。
     - 宏展开：将代码中所有的宏（如 MAX_NUM）替换为其定义的值（100）。所以 if (result < MAX_NUM) 会变成 if (result < 100)。
     - 条件编译：处理 #if, #ifdef, #ifndef, #else, #endif 等指令，根据条件保留或删除特定代码块。
     - 删除注释
   - **编译**：将预处理后的（.i 文件）翻译成特定处理器架构的汇编语言。
     - 工作：检查代码的语法和语义是否正确。进行各种优化（如常量传播、循环优化等）。生成效率最高的汇编代码。
   - **汇编**：将人类可读的汇编代码（.s 文件）翻译成机器可以直接执行的机器码。
     - 工作：将汇编指令助记符（如 movl, callq）转换为对应的二进制操作码（一串 0 和 1），并为变量和函数分配初步的内存地址。
   - **链接**：将一个程序的所有目标文件（如 main.o）以及它们所需要的库文件（如 C 标准库 libc.a 中的 printf 函数）组合在一起，形成一个单一的可执行文件。
     - 工作：解决符号引用（函数和变量的地址），确保所有的函数和变量都能被正确地找到。

四、硬件

一、相机

1. **核心组件：**
   i. **镜头**：相机的眼球，负责收集和聚焦光线
      - 焦距(f)：决定视野范围和放大倍数
      - 光圈(F)：控制进光量和景深
      - 接口类型：C接口、CS接口、F接口等

   ii. **图像传感器**：相机的视网膜，将光信号转为电信号。
      - 传感器类型一般分为CCD和CMOS两种，
        - CCD：前者图像质量好，噪声低，但功耗高、成本高
        - CMOS：功耗低、成本低、集成度高，现代相机主流选择
      - 传感器尺寸
        - 1/4"：小型传感器，常用于消费级产品
        - 1/2.5"：常见于工业相机
        - 1"：较大传感器，提供更好图像质量
        - 全画幅：专业级，最佳图像质量

   iii. **图像处理器**
      - 模数转换 (ADC)，噪声抑制，自动曝光 (AE)，自动白平衡 (AWB)，图像压缩，色彩处理

   iv. **相机接口**
      - USB2.0/3.0/3.1 ： 通用性强，即插即用，USB3.0速度可达5Gbps
      - GigE (千兆网)：传输距离长（可达100米），抗干扰能力强
      - Camera Link ： 高速传输，专业工业应用
      - CoaXPress：超高速度，长距离传输
      - MIPI CSI ： 嵌入式系统常用（如树莓派）

2. **一些概念解释**
   - 传感器捕捉光子之后会产生噪声，影响曝光

   i. **图像噪声**：噪声是图像中不希望出现的、随机的亮度或颜色波动，可以看作是图像的“杂质”或“雪花点”，**光线不足（暗光）的条件下尤为明显。**
      - 读出噪声：在将每个像素的电信号读出并放大过程中，电路本身产生的固有噪声。这是无法完全避免的。
      - 暗电流噪声：即使在没有光的情况下，图像传感器由于热效应也会产生少量电荷，这些电荷会被误认为是光信号。长时间曝光时，这种噪声会非常明显（传感器发热）。
      - 光子散粒噪声：光子组成的，其到达传感器的时间具有量子随机性。到达每个像素的光子数量也存在波动。

   ii. **曝光**：曝光指的是图像传感器在拍摄过程中接收到的总光量。
      - 曝光三角：光圈，快门速度，感光度
      - 光圈越大光线越多，快门速度越快，光线越少，
      - 感光度：传感器对光线的敏感程度。ISO值越高（如ISO 3200），传感器对光越“敏感”，可以在同样光线下获得更亮的图像。但提高ISO的本质是放大电信号，这同时也会放大噪声，因此高ISO通常会导致画质下降。

   iii. **白平衡**：白平衡是为了纠正不同光源下的色偏，确保白色的物体在任何光线下看起来都是白色的。
      - 不同光源的“色温”不同。色温单位是开尔文。
        - 低色温（约2000-3000K）：偏暖色调，如烛光、白炽灯，光线偏黄/橙。
        - 高色温（约6000-10000K）：偏冷色调，如阴天、阴影下，光线偏蓝。
        - *如果在白炽灯下不校正白平衡，拍出的照片就会整体偏黄。*
      - 方法：
        - 自动白平衡：相机算法会分析整个场景，假设场景的平均颜色应该是中性灰（18%灰），从而计算出需要补偿的色温值。
        - 手动/预设白平衡：用户可以告诉相机当前的光源环境（如选择“阴天”、“白炽灯”模式），或者最准确的方法是拿一张纯白色的卡纸在当前光线下让相机对其进行测光，相机以此为标准进行校正。

- 原理：**本质上是通过调整图像中红、绿、蓝三个通道的增益（放大倍数）来实现。如果图像偏蓝，就适当增加红色和绿色的增益，来中和掉多余的蓝色。**

iv. **图像ADC**：连续信号转换为离散信号

- 输入(模拟信号)：图像传感器（CMOS/CCD）的每个像素点在接受光照后，会产生一个强度与光量成正比的电压信号。这个电压是连续变化的，比如从0V到3.3V之间可以有无限个值。
- ADC对这个电压进行“测量”和“分类”。例如，一个8位的ADC会将0-3.3V的电压范围**划分为 $2^8 = 256$ 个等级**。0V对应数字0，3.3V对应数字255。如果一个像素产生的电压是1.65V，ADC就会把它“归类”到第128这个等级。
- 输出（数字信号）：最终，每个像素点的亮度都被转换成了一个数字（例如0到255之间的一个整数）。这一串数字就是最原始的图像数据。

五、刷题笔记

一、算法1-1：高精度模拟：

1. 高精度模板：见高精度模板.md
2. p1007,p1009,p1024,p1098,p1518,p1563,p2670

```cpp
//p1007:
#include <bits/stdc++.h>
using namespace std;

int arr[510][510];
struct cmd_
{
    pair<int, int> pst;
    int r;
    int turn;
};

void trans(int x, int y, int r, int turn)
{
    if (turn == 0)
    { // 顺时针
        vector<int> v;
        for (int i = 0; i < (2 * r) + 1; i++)
        {
            for (int j = 0; j < (2 * r) + 1; j++)
            {
                v.push_back(arr[x - r + i][y - r + j]);
            }
        }
        /*
        for(int i = 0; i < ((2*r)+1)*((2*r)+1); i++){
            cout << v[i] << " ";
        }
        cout << endl;
        */
        for (int i = 0; i < (2 * r) + 1; i++)
        {
            for (int j = 0; j < (2 * r) + 1; j++)
            {
                arr[x - r + j][y + r - i] = v[0 + (i * ((2 * r) + 1)) + j];
            }
        }
    }
    if (turn == 1)
    { // 逆时针
        vector<int> v;
        for (int i = 0; i < (2 * r) + 1; i++)
        {
            for (int j = 0; j < (2 * r) + 1; j++)
            {
                v.push_back(arr[x - r + i][y - r + j]);
            }
        }
        /*
        for(int i = 0; i < ((2*r)+1)*((2*r)+1); i++){
            cout << v[i] << " ";
        }
        cout << endl;
        */
        for (int i = 0; i < (2 * r) + 1; i++)
        {
            for (int j = 0; j < (2 * r) + 1; j++)
```

```
            {
                arr[x + r - j][y - r + i] = v[0 + (i * ((2 * r) + 1)) + j];
            }
        }
    }
}
int main()
{
    int n, m;
    cin >> n >> m;
    int cnt0 = 1;
    cmd_ *cmd = new cmd_[m + 5];
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            arr[i][j] = cnt0;
            cnt0++;
        }
    }

    for (int i = 0; i < m; i++)
    {
        cin >> cmd[i].pst.first >> cmd[i].pst.second >> cmd[i].r >> cmd[i].turn;
    }
    for (int i = 0; i < m; i++)
    {
        trans(cmd[i].pst.first - 1, cmd[i].pst.second - 1, cmd[i].r, cmd[i].turn);
    }
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }

    delete[] cmd;
    return 0;
}
```

```cpp
//p1024
#include <bits/stdc++.h>
using namespace std;
int w_num = 0;
int l_num = 0;
int i = 0;
string game;
void work(int limit)
{

    while (1)
    {
        if (game[i] == 'W')
        {
            w_num++;
        }
        else
        {
            l_num++;
        }
        i++;
        if (i == game.size())
        {
            cout << w_num << ":" << l_num << endl;
            w_num = 0;
            l_num = 0;
            i = 0;
            return;
        }
        if ((w_num == limit || l_num == limit) && (w_num - l_num >= 2 || l_num - w_num >= 2))
        {
            cout << w_num << ":" << l_num << endl;
            w_num = 0;
            l_num = 0;
        }
    }
}

int main()
{

    char temp;
    while (cin >> temp)
    {
        if (temp == 'E')
            break;
        game += temp;
    }

    work(11);
    cout << endl;
    work(21);
    return 0;
}
```

题解见压缩包.....