计算机学院　高级语言程序设计　课程实验报告

实验题目：多态扩展		学号：202200400053
日期：2024-05-09	班级：　2202	姓名：　王宇涵
Email：1941497679@qq.com		

实验目的：

1. 掌握多态的扩展知识，能分析、编程。
2. 大整数运算挑战

实验软件和硬件环境：

软件环境 ：VSCODE + DEV-C++

硬件环境 ：Legion Y7000P

实验步骤与内容：

扩展 1. 赋值运算符重载

系统对类提供的默认赋值运算符(=)只能完成浅拷贝(仅复制对象的成员)，对于含指针成员的类，需重载=运算符完成深拷贝(即指针成员所指内容的复制)。

针对之前的习题 ex6-20，设计重载=运算符完成深拷贝，并且能够完成型如 X=Y=2;的连续赋值运算。

```
//使用=运算符重载函数实现深拷贝
SimpleCircle & SimpleCircle::operator = (const SimpleCircle & rhs)
{
    if (this == &rhs)
    {
        return *this;
    }
    delete itsRadius;
    itsRadius = new int(rhs.getRadius());
    return *this;
}
```

测试输入

```
  SimpleCircle c1, c2;
  c1 = c2 = 3;
  cout << c1.getRadius() << " " << c2.getRadius() << endl;
  return 0;
```

输出

```
PS D:\Baidu
3 3
```

扩展 2.两个特殊的类成员函数：转换构造函数，类型转换函数（类型转换运算符）

（1）转换构造函数：（转入：外部类型对象转换为本类对象）

Complex(double r){ //转换构造函数。也可用 Complex(double r=0, double i=0)默认参数值形式替换。将 double 类型的值 转为 Complex 类型对象

real=r;imag=0;

}

（2）类型转换函数(也称类型转换运算符)（转出：本类对象转换为外部类型对象）。

例：解决 double d=Comlex(1,2)+Complex(3,4); //Complex 向 double 类型转换

operator double(){ //类型转换函数(运算符）与普通函数不同，无返回类型，但函数名就是返回类型，无参数；

return real;

}

//如果转换构造、类型转换及双目运算符重载同时存在，则应用中可能产生二义性问题：

如表达式：1.0+Complex(2.0,3.0) //1.0 是向 Complex 转，还是 Complex 向 double 转？

答 ：解决二义性问题: 使用显示转换

例如 double result = 1.0 + static_cast<double>(Complex(2.0, 3.0));

扩展 3. 显式类型转换 cast

请阅读运行附件 cast.zip 中的例子，分析其中 p3.cpp 的运行结果。

```
PS D:\BaiduSyncdisk\CLASS
unsafe dynamic_cast_1
```

答 ：输出结果为 dynamic_cast_1，说明

pd = dynamic_cast<Derived*> (&b); //父类对象地址转为子类地址指针（子指针指向父对象）

这个转化是不安全的, 因为 b 不含有 derived 的类型.

扩展 4. 抽象类的作用（深刻体会向上抽象思想的应用）

设计一个用户界面类 UI，界面中可以包含不同的形状并能显示其信息。

分析附件 8 中两种 UI 的设计，如果没有设计抽象类 Shape,当增加新的圆形类 Circle 时，程序的易修改性、模块的独立、互不影响性。

当设计了抽象类 `Shape` 时：

1. 易修改性：

- 当需要添加新的圆形类 `Circle` 时，只需继承 `Shape` 类并实现相应的方法即可。由于所有形状类都继承自 `Shape`，因此在新增 `Circle` 类时，不需要对已有的代码做太多修改。
- 如果后续需要修改 `Shape` 类的一些公共方法，也可以更轻松地进行修改，因为这些修改会自动应用到所有继承自 `Shape` 的形状类中。

2. 模块的独立性：

- 每个形状类都是基于相同的抽象类 `Shape`，因此它们之间具有相似的接口和行为。这种一致性使得模块更加独立，易于理解和维护。
- 新增 `Circle` 类时，它与其他形状类的实现是相互独立的，不会影响其他形状类的行为或接口。

3. 互不影响性：

- 如果在某个形状类的实现中出现了问题，修复该问题不会影响其他形状类的行为，因为它们是相互独立的。
- 当新增 `Circle` 类时，它的实现与其他形状类的实现无关，因此不会对其他形状类产生任何影响。

当没有设计抽象类 `Shape` 时：

1. 易修改性：

- 新增 `Circle` 类时，需要在不包含抽象类 `Shape` 的 UI 中添加新的类，并确保其方法与其他形状类的方法保持一致。这可能需要更多的修改，因为没有一个公共的接口来统一不同形状类的行为。

- 如果后续需要修改形状类的一些公共方法，可能需要逐个查找并修改每个形状类的实现，这增加了修改的复杂性和风险。

2. 模块的独立性：

- 没有抽象类 `Shape`，每个形状类都是独立的实现，它们之间可能没有共享的接口或行为，这降低了模块的独立性。新增 `Circle` 类可能需要重新设计整个类结构，以确保它与其他形状类的实现方式相符合。

3. 互不影响性：

- 如果修改了某个形状类的实现，可能会影响其他形状类的行为，因为它们之间可能存在一些依赖或相互作用。新增 `Circle` 类时，可能需要调整其他形状类的实现以适应新的类结构，这可能导致意想不到的影响。

创新与挑战：

第 8 章 PPT，P27，创新与挑战：创建大整数类型

（1）创建一个更大的无符号长整型（精度可达 1000 位十进制）

（2）创建该类型，都需要设计哪些内容？（存储表示，数据范围，运算，进位、输出）

设计完成 100 位大整数类的加、减、乘、除运算符重载。

（3）如何提高它的性能？（空间、时间）你能超过下面视频中的例子吗？

参考：https://www.bilibili.com/video/BV1Ap4y1h75e 如果用笨方法计算 2 的 100，000 次幂会怎样

https://www.bilibili.com/video/BV1Xb4y1R7hH 计算 3 的 1，000，000 次幂

完成程序设计如下:

类

```
class BigInteger {
private:
    std::vector<int> digits; // 存储每一位的数字
    std::vector<int> R;//余数
public:
    // 构造函数
    BigInteger(const std::string& number);//构造函数
    BigInteger() {};//默认构造函数
    BigInteger(const BigInteger &rhs) {//拷贝构造函数
        digits = rhs.digits;
    }
    // 加法运算符重载
    BigInteger operator+(const BigInteger& other) const;
    // 减法运算符重载
    BigInteger operator-(const BigInteger& other) const;
    // 乘法运算符重载
    BigInteger operator*(const BigInteger& other) const;
    // 除法运算符重载
    BigInteger operator/(const BigInteger& other) const;
    // 重载比较运算符
    bool operator<(const BigInteger& other) const;
    // 输出函数
    friend std::ostream& operator<<(std::ostream& os, const BigInteger& num);
};
```

加法:

```
std::vector<int> add(std::vector<int>&A, std::vector<int>&B)
{
    std::vector<int> C;
    int  t = 0;
    for (int i = 0; i < A.size() || i < B.size(); i++) {
        if (i < A.size()) t += A[i];
        if (i < B.size()) t += B[i];
        C.push_back(t % 10);
        t /= 10;
    }
    if (t) C.push_back(1);
    return C;
}
```

减法 ：

```cpp
std::vector<int> subtract(std::vector<int>&A, std::vector<int>&B)
{
    std::vector<int> C;
    int t = 0;
    for (int i = 0; i < A.size(); i++) {
        t = A[i] - t;
        if (i < B.size()) t -= B[i];
        C.push_back((t + 10) % 10);
        if (t < 0) t = 1;
        else t = 0;
    }
    while (C.size() > 1 && C.back() == 0) C.pop_back();
    return C;
}
```

乘法

```cpp
std::vector<int> multi(std::vector<int>&A, std::vector<int>&B)
{
    std::vector<int> C(A.size() + B.size() + 7, 0); // 初始化为 0，C的size可以大一点

    for (int i = 0; i < A.size(); i++)
        for (int j = 0; j < B.size(); j++)
            C[i + j] += A[i] * B[j];

    int t = 0;
    for (int i = 0; i < C.size(); i++) { // i = C.size() - 1时 t 一定小于 10
        t += C[i];
        C[i] = t % 10;
        t /= 10;
    }

    while (C.size() > 1 && C.back() == 0) C.pop_back(); // 必须要去前导 0，因为最高位很可能是 0
    return C;
}
```

除法

```cpp
//除法
std::pair<std::vector<int>, std::vector<int>> divide(std::vector<int>&A, std::vector<int>&B)
{
    std:: vector<int> C, R;
    int n = A.size(), m = B.size(), d = n - m;
    C.resize(d + 1, 0);
    // 枚举补 0 的个数
    for (int len = d; len >= 0; len--) {
        std::vector<int> Bp(len, 0);
        for (int x : B) Bp.push_back(x);

        // A >= Bp
        while (!cmp(A, Bp)) {
            C[len] += 1;
            A = subtract(A, Bp);
        }
    }
    while (C.size() > 1 && C.back() == 0) C.pop_back();
    R = A;
    return make_pair(C, R);
}
```

注意: 除法中‘.’后的数字代表余数

测试输入 1

```cpp
BigInteger num1("500000000000000000000000000000000000000000000000000000000000000");
BigInteger num2("200000000000000000000000000000000000000000000000000000000000001");
```

测试输出 1

```
Sum: 500020000000000000000000000000000000000000000000000000001
Difference: 499979999999999999999999999999999999999999999999999999999
Product: 100000000000000000000000000000000000000000000000000050000000000000000000000000000000000000000000000000000000000000
Quotient: 24999.199999999999999999999999999999999999999999975001
```

测试输入 2

```cpp
BigInteger num1("500000000000000000000000000000000000000000000000000000000000000");
BigInteger num2("200000000000000000000000000000000000000000000000000000000000000");
```

测试输出 2

```
Sum: 500020000000000000000000000000000000000000000000000000000
Difference: 499980000000000000000000000000000000000000000000000000000
Product: 100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
Quotient: 25000.0
```

优化算法: 以求解 2 的 100000 次方为例

使用快速幂算法:

模板如下, 其中 a = 2, k = 100000, 因为结果很大, 因此 p 为模数

```
LL qmi(LL a,int k,int p)
{

    LL res=1;
    while(k)
    {
        if(k&1)res=res *a%p;
        k>>=1;
```

```
        a=a*a%p;
    }
    return res;
}
```

结论分析与体会：

本次实验我掌握了多态的扩展知识，能分析代码并进行独立编程. 同时, 我也认真思考, 主动调试, 成功完成了大整数运算挑战, 实现高精度加法, 减法, 乘法, 除法的运算(并得到余数), 提高了自己的编程能力, 并为以后的学习奠定了良好的基础.

就实验过程中遇到的问题及解决处理方法，自拟 1－3 道问答题：

1. 实现大整数运算的主体思路是什么？

答 ：将整数的数字存入数组中, 数组的长度代表数字的长度, 对数组的数字进行运算操作, 这样就可以计算出大整数.

2. 使用虚函数的优点是什么？

答 ：实现运行时多态, 实现了一致接口, 简化了代码结构, 提高了可读性和可维护性, 支持动态绑定等等.