C++模板和操作符重载教学题：编写GenericAdder

目标

基于我们给的代码框架，编写MyComplex类和GenericAdder模板方法，实现不同类型的加法

通过这个练习学习：如何使用操作符重载和模板进行设计以简化代码

框架代码概要和要求

- MyComplex类
  - real/imag分别表示实部和虚部
  - 两个MyComplex类实例相加的时候，实部和实部相加，虚部和虚部相加
  - 打印Complex类的时候，使用重载的`operator<<`进行打印
    - 打印结果表示为`(real,imag)`
- GenericAdder模板方法：接受两个模板参数`Ty1`和`Ty2`，返回相加后的结果
  - 这两个参数需要满足可以相加的约束，换句话说，如果a和b不能相加，则模板实例化会失败
  - 可以使用模板特化的技巧，为一些特定的Type设计不同的GenericAdder方法（具体语法自行搜索，后面给出简要介绍）
  - 作业中要求对指定几种Type来实现GenericAdder方法
    - 参考示例：`Ty1=int; Ty2=string`：使用模板特化，将int类型参数转成字符串后，附加在string类型参数前
    - `Ty1=string; Ty2=int`：将int类型参数转成字符串后，附加在string类型参数后
    - `Ty1=int; Ty2=MyComplex`：建立一个新的MyComplex对象，将int类型实参作为real，imag指定为0，将这个新对象和第二个MyComplex类型的参数相加
    - `Ty1=MyComplex*; Ty2=MyComplex*`：求解指针指向的元素之和

测试代码

- 包括main函数和对应的test*()测试用例
- 测试内容为GenericAdder的输出结果

注意事项

- **完成TODO标注的函数**
  - 理想情况下本次作业的代码量应不超过20行，不需要把问题想得太复杂
- 本次作业只用于学习使用，实践中很少需要对函数模板进行特化，而是直接使用函数重载
  - 函数模板特化和函数重载都是基于Type的分发机制
  - 函数模板特化的行为常常和我们预想的不同

    - 类模板特化的适用场景更多

- 提交要求
    - **请不要修改main函数和测试代码！**可能会影响后台用例的判定
    - 不要投机取巧！
        - 助教会人工检查运行行为异常的代码提交，并将本次练习记录为0分

- 测试样例

```
void test() {
  std::cout << GenericAdder(1, 2) << "\n";
  std::cout << GenericAdder(1, std::string("2")) << "\n";
}
//正确输出结果
3
12
```

练习之外（不作为练习，仅供扩展学习）

- C++通过继承来实现动态多态
    - 具体来说应该是绑定动态多态bounded dynamic polymorphism
    - 基于虚表，在程序运行时动态绑定接口的实现
    - 代码体积更小，运行时开销更大
- C++通过template来实现静态多态
    - 具体来说应该是未绑定的静态多态unbounded static polymorphism
    - 接口的实现是在编译期决定的，给相关编译优化提供更多的空间（内联）
    - 学习难度很大，不方便调试
- template相关知识点（仅供参考）
    - https://zhuanlan.zhihu.com/p/454432180

附录：代码框架

```
#include <iostream>

class MyComplex {
public:
    MyComplex(int real, int imag) : real(real), imag(imag) {}
    int real;
    int imag;
    MyComplex operator+(const MyComplex& other) const {
        // TODO: implement this
    }
```

```
    friend std::ostream& operator<<(std::ostream& os, const MyComplex&
mc);
};

std::ostream& operator<<(std::ostream& os, const MyComplex& mc) {
    // TODO: implement this
}

template <typename Ty1, typename Ty2>
auto GenericAdder(Ty1 a, Ty2 b) {
    return a + b;
}
// sample
template<>
auto GenericAdder(int a, std::string b) {
    return std::to_string(a) + b;
}
//TODO: write GenericAdder for string+int

//TODO: write GenericAdder for int+MyComplex

//TODO: write GenericAdder for (Ty1*)+(Ty2*)

void test1() {
    std::cout << GenericAdder(1, 2) << "\n";
    std::cout << GenericAdder(1.0, 2.0) << "\n";
    std::cout << GenericAdder(1, std::string("1")) << "\n";
}

void test2() {
    std::cout << GenericAdder(1, MyComplex(1, 3)) << "\n";
    std::cout << GenericAdder(MyComplex(1, 3), MyComplex(1, 3)) <<
"\n";
}

void test3() {
    MyComplex a{1, 3};
    MyComplex b{2, 6};
    std::cout << GenericAdder(&a, &b) << "\n";
}

void test4() {
  MyComplex a{1, 3};
  MyComplex b{2, 6};
  MyComplex *pa = &a;
  MyComplex *pb = &b;
  std::cout << GenericAdder(&pa, &pb) << "\n";
}
```

```
#define TEST(x) std::cout << "test" << #x << "\n"; test##x();

int main(){
  TEST(1);
  TEST(2);
  TEST(3);
  TEST(4);
  return 0;
}
```