整数流

通过重载 `++`（包含前置和后置）和 `*` 运算符实现一个**惰性**求值的 `int` 流。

> 本题灵感来自《计算机程序的结构和解释》第 3.5 节“流”

可通过如下三种方式创建一个 `int` 流：

- 仅指定起始值。表示从该值开始、每次加一的流，例如 `IntStream(1)` 能够生成序列 1、2、3、…
- 指定起始值和结束值。表示从指定的起始值开始、每次加一、不包含指定的结束值在内的流。例如，`IntStream(1, 11)` 能够生成序列 1、2、…、**10**
- 指定起始值、结束值和步长。表示从指定的起始值开始、每次增加指定的步长、不包含指定的结束值在内的流。例如，`IntStream(1, 11, 3)` 能够生成序列 1、4、7、10，`IntStream(1, 14, 3)` 能够生成序列 1、4、7、10、13

其中：

- 前置 `++` 和后置 `++` 用于推进 `int` 流
- `*` 用于获取当前 `int` 流最前面的值
- `operator bool` 用于将当前 `int` 流转换为 `bool` 值

输入

本题不需要处理输入。

输入格式参见代码框架的 `main` 函数。

输出

本题不需要处理输出。

输出格式参见代码框架的 `test_*` 系列函数。

提示

算术运算可能存在溢出问题！

代码框架

```
#include <functional>
#include <iostream>
#include <limits>
#include <string>
```

```cpp
#include <unordered_map>

class IntStream {
 public:
  explicit IntStream(int first);
  IntStream(int first, int last);
  IntStream(int first, int last, int stride);

  IntStream &operator++();
  IntStream operator++(int);
  int operator*() const;

  operator bool() const;

  // TODO: your code
};

void print_answer(const IntStream &s, int expect) {
  std::cout << std::boolalpha;
  if (s) {
    std::cout << (*s == expect) << ' ' << *s << std::endl;
  } else {
    std::cout << false << std::endl;
  }
}

/**
 * @brief 测试 IntStream(int)
 */
void test_1() {
  IntStream s(0);
  for (size_t i = 0; i < 10; i++) {
    ++s;
  }
  print_answer(s, 10);
}

/**
 * @brief 测试 IntStream(int, int)
 */
void test_2() {
  IntStream s(0, 10);
  for (size_t i = 0; i < 9; i++) {
    s++;
  }
  print_answer(s, 9);
}

/**
```

```
 * @brief 测试 IntStream(int, int, int) - 不考虑溢出
 */
void test_3() {
  IntStream s(0, 10, 2);
  for (size_t i = 0; i < 4; i++) {
    ++s;
  }
  print_answer(s, 8);
}

/**
 * @brief 测试 IntStream(int, int, int) - 步长为负数
 */
void test_4() {
  IntStream s(10, 0, -1);
  for (size_t i = 0; i < 10; i++) {
    s++;
  }
  print_answer(s, 0);
}

/**
 * @brief 测试 IntStream(int, int, int) - 考虑溢出，大于最大值
 */
void test_5() {
  IntStream s(std::numeric_limits<int>::max() - 10000,
              std::numeric_limits<int>::max(), 123);
  for (size_t i = 0; i < 50; i++) {
    ++s;
  }
  print_answer(s, 2147479797);
}

/**
 * @brief 测试 IntStream(int, int, int) - 考虑溢出，小于最小值
 */
void test_6() {
  IntStream s(std::numeric_limits<int>::min() + 10000,
              std::numeric_limits<int>::min(), -123);
  for (size_t i = 0; i < 50; i++) {
    s++;
  }
  print_answer(s, -2147479798);
}

/**
 * @brief 测试步长为 0 的情况
 */
void test_7() {
```

```
  IntStream s(std::numeric_limits<int>::min(),
std::numeric_limits<int>::max(),
              0);
  for (size_t i = 0; i < 10000; i++) {
    s++;
  }
  print_answer(s, std::numeric_limits<int>::min());
}

/**
 * @brief 测试范围 [first, last) 非常大的情况
 */
void test_8() {
  IntStream s(std::numeric_limits<int>::min(),
std::numeric_limits<int>::max());
  for (size_t i = 0; i < 10000; i++) {
    s++;
  }
  print_answer(s, std::numeric_limits<int>::min() + 10000);
}

int main() {
  std::unordered_map<std::string, std::function<void()>>
test_cases_by_name = {
      {"test_1", test_1}, {"test_2", test_2}, {"test_3", test_3},
      {"test_4", test_4}, {"test_5", test_5}, {"test_6", test_6},
      {"test_7", test_7}, {"test_8", test_8},
  };
  std::string tname;
  std::cin >> tname;
  auto it = test_cases_by_name.find(tname);
  if (it == test_cases_by_name.end()) {
    std::cout << "输入只能是 test_<N>, 其中 <N> 可取整数 1 到 8." <<
std::endl;
    return 1;
  }
  (it->second)();
}
```

有理数

我们知道，当前计算机中常用的浮点数表示方式是离散的，本题需要你实现一个以分数形式表示的有理数。

你需要实现如下的功能：

- 用分子（numerator）和分母（denominator）构造一个有理数
  - 如果分母为 0，抛出 `std::logic_error` 异常，异常信息为 `denominator must be != 0`
- 拷贝构造函数
- 重载的 `operator=`
- 有理数的加、减、乘、除
  - 保证不会出现相减结果为负的情况
  - 保证不会出现除数为 0 的情况
- 获取有理数的分子和分母
- 向 `std::ostream` 对象输出一个有理数，即实现 `operator<<` 的重载
  - 如果分子为 0，则不论分母的值为多少，都输出 `0`
  - 将分子记为 `N`、分母记为 `D`，分子和分母的最大公约数（GCD）为 `G`，记 `N / G` 的值为 `M`、`D / G` 的值为 `E`，则输出为 `M/E`，注意**不包含**空格！
  - 如果约分后的分母为 1，则不输出分母
  - 示例
    - `std::cout << Rational<int>(3, 4)`，得到 `3/4`
    - `std::cout << Rational<int>(2, 4)`，得到 `1/2`
    - `std::cout << Rational<int>(4, 2)`，得到 `2`
- 实现 `_r` 后缀以支持通过 `"3/4"_r` 的形式构建一个有理数
  - 该特性称为“User-Defined Literals”，从 C++11 引入
  - 参见官方文档、这篇文章和这篇文章

输入、输出描述

本题不需要处理输入输出。

输入格式参见代码框架的 `main` 函数，输出格式参见代码框架的 `test_*` 系列函数。

示例

见代码框架中的 `test_*` 系列函数。

扩展

- 假设我们用 `int` 作为 `Rational` 的模板参数。如果想要获取 `double` 类型的分子或分母，可以通过先获取到 `int` 值然后进行强制类型转换。然而，还有另一种方式：通过模板参数指

定返回值的类型。探索此种实现方式，了解 type traits 的基本概念和标准库提供的 `is_convertible` traits

```
// 示例：通过模板参数指定返回值类型
Rational<int> r(3, 4);
double n = r.numerator<double>();
```

- 如何确保模板参数 `T` 对应的类型一定是可以进行算术运算的？了解相关的 type traits

代码框架

```
#include <cctype>
#include <cstring>
#include <functional>
#include <iostream>
#include <stdexcept>
#include <string>
#include <unordered_map>

template <typename T>
class Rational {
 public:
  Rational(const T &n, const T &d);
  Rational(const Rational<T> &rhs);

  T numerator() const;
  T denominator() const;

  Rational<T> operator+(const Rational<T> &rhs) const;
  Rational<T> operator-(const Rational<T> &rhs) const;
  Rational<T> operator*(const Rational<T> &rhs) const;
  Rational<T> operator/(const Rational<T> &rhs) const;

  Rational<T> &operator=(const Rational<T> &rhs);

  friend std::ostream &operator<<(std::ostream &out, const Rational<T>
&r);

  // TODO: your code
};

Rational<int> operator""_r(const char *str, size_t len);

void test_1() {
  Rational<int> r(3, 4);
```

```
  std::cout << r << std::endl;
}

void test_2() {
  bool exception_thrown = false;
  bool expected_message = false;
  try {
    Rational<int> r = Rational<int>(1, 0);
  } catch (std::logic_error &exn) {
    exception_thrown = true;
    if (!strncmp(exn.what(), "denominator must be != 0", 24)) {
      expected_message = true;
    }
  } catch (...) {
  }
  if (exception_thrown) {
    std::cout << "std::logic_error thrown!" << std::endl;
    if (expected_message) {
      std::cout << "the message is as expected." << std::endl;
    }
  } else {
    std::cout << "Oops!" << std::endl;
  }
}

void test_3() {
  Rational<int> r(3, 4);
  std::cout << r.numerator() << ' ' << r.denominator() << std::endl;
}

void test_4() {
  Rational<int> lhs(1, 6), rhs(1, 3);
  std::cout << (lhs + rhs) << std::endl;
}

void test_5() {
  Rational<int> lhs(1, 2), rhs(1, 6);
  std::cout << (lhs - rhs) << std::endl;
}

void test_6() {
  Rational<int> lhs(2, 4), rhs(4, 6);
  std::cout << (lhs * rhs) << std::endl;
}

void test_7() {
  Rational<int> lhs(2, 4), rhs(4, 6);
  std::cout << (lhs / rhs) << std::endl;
}
```

```
void test_8() {
  Rational<int> r(3, 4);
  std::cout << r << std::endl;
  Rational<int> rhs(101, 203);
  r = rhs;
  std::cout << r << ' ' << rhs << std::endl;
}

void test_9() {
  auto r = "3/4"_r;
  std::cout << r << std::endl;
}

void test_10() { std::cout << Rational<int>(4, 2) << std::endl; }

void test_11() {
  std::cout << (Rational<int>(1, 2) - Rational<int>(2, 4)) <<
std::endl;
}

void test_12() { std::cout << Rational<int>(3, 6) << std::endl; }

int main() {
  std::unordered_map<std::string, std::function<void()>>
test_cases_by_name = {
      {"test_1", test_1},   {"test_2", test_2},   {"test_3", test_3},
      {"test_4", test_4},   {"test_5", test_5},   {"test_6", test_6},
      {"test_7", test_7},   {"test_8", test_8},   {"test_9", test_9},
      {"test_10", test_10}, {"test_11", test_11}, {"test_12", test_12},
  };
  std::string tname;
  std::cin >> tname;
  auto it = test_cases_by_name.find(tname);
  if (it == test_cases_by_name.end()) {
    std::cout << "输入只能是 test_<N>, 其中 <N> 可取整数 1 到 12." <<
std::endl;
    return 1;
  }
  (it->second)();
}
```

C++操作符重载教学题：编写bitvector

目标

基于我们给的代码框架，编写一个容器MyBitVector，用该容器维护一个0/1序列

通过这个练习学习：如何使用操作符重载进行设计以简化代码

框架代码概要和要求

- 需要设计两个私有成员变量
  - size 表示存储和打印的0/1序列长度
  - data 表示存储的0/1序列
- 构造函数
  - `MyBitVector(int size)` 存储一个长度为size，内容为0的序列
  - `MyBitVector(const string &s)` 参数为0/1字符串序列，将其转为对应的0/1序列进行存储
- 成员函数
  - `set(int index)` 将index位置的bit设置为1
  - `clear(int index)` 将index位置的bit设置为0
  - `get(int index)` 获取index位置的bit信息
  - `print` 打印一行长度为size的0/1序列，中间没有空格
- 操作符重载
  - 二元操作符：
    - `MyBitVector operator&(MyBitVector &other)` 按位与操作
    - `MyBitVector operator|(MyBitVector &other)` 按位或操作
    - `MyBitVector operator^(MyBitVector &other)` 按位异或操作
  - 单元操作符
    - `MyBitVector operator~()` 按位取反
    - `MyBitVector &operator&=(MyBitVector &other)` 原地按位与操作
    - `MyBitVector &operator|=(MyBitVector &other)` 原地按位或操作
    - `MyBitVector &operator^=(MyBitVector &other)` 原地按位异或操作

测试代码

- 包括main函数和对应的test*()测试用例
- 测试内容为`print`的打印结果，即当前MyBitVector实例中存储的0/1序列

注意事项

- **完成TODO标注的函数**
  - 注意操作符重载的函数签名

        - 有的是返回新的MyBitVector
        - 有的是返回已有MyBitVector的引用
    - 建议在实现get/set/clear时加上运行时检查，防止产生非法越界操作

- 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(){
  MyBitVector bv1("00000000");
  MyBitVector bv2("11110011");
  MyBitVector bv3 = bv1 | bv2;
  bv3.print();
}
//正确输出结果
11110011
```

练习之外（不作为练习，仅供扩展学习）

- bitvector有很多良好的特性，在实践中的应用非常广泛
    - 数据存储得非常紧密，能充分利用缓存，减少耗时的内存store/load操作
    - 有利于实现bit-level的并行计算
    - 可以用在需要大量求解交并集的场景中
- 但是bitvector很容易造成大量的空间浪费
    - 一般会根据具体场景设计或使用特定的bitvector实现，可以搜索`sparse bit vector`
- 一个设计良好的bitvector需要考虑很多因素
    - 如何实现高效的扩容
    - 如何对长度不同的bitvector进行位操作
    - 如何减少空间浪费

附录：代码框架

```
#include <iostream>
using namespace std;

class MyBitVector {
private:
    // TODO: Add your private member variables here
```

```
public:
    explicit MyBitVector(int size) {
        // TODO: Add your constructor here, and initialize your
bitvector with 0s
    }

    explicit MyBitVector(const string &s) {
        // TODO: Add your constructor here, and initialize your
bitvector with the given string of 0s and 1s
    }

    void set(int index) {
        // TODO: Set the bit at index to 1
    }

    void clear(int index) {
        // TODO: Set the bit at index to 0
    }

    bool get(int index) {
        // TODO: Return the value of the bit at index
        return false;
    }

    void print() {
        // TODO: Print the bitvector to continuous 0s and 1s of given
length
    }

    // TODO: operator overloads to generate new bitvectors from
existing ones
    MyBitVector operator&(MyBitVector &other) {
        // TODO: bv = bv1 & bv2
    }

    MyBitVector operator|(MyBitVector &other) {
        // TODO: bv = bv1 | bv2
    }

    MyBitVector operator^(MyBitVector &other) {
        // TODO: bv = bv1 ^ bv2
    }

    MyBitVector operator~() {
        // TODO: bv1 = ~bv2
        // question: what if we want to modify the original bitvector
without creating a new one?
    }
```

```
    // TODO: operator overloads to modify existing bitvectors in place
    MyBitVector &operator&=(MyBitVector &other) {
        // TODO: bv1 &= bv2
    }

    MyBitVector &operator|=(MyBitVector &other) {
        // TODO: bv1 |= bv2
    }

    MyBitVector &operator^=(MyBitVector &other) {
        // TODO: bv1 ^= bv2
    }

};

void test1() {
    MyBitVector bv1("01010100");
    MyBitVector bv2("11101111");
    MyBitVector bv3 = bv1 & bv2;
    bv1 &= bv2;
    bv1.print();
    bv2.print();
    bv3.print();
}

void test2() {
    MyBitVector bv1("00001000");
    MyBitVector bv2("11010011");
    MyBitVector bv3 = bv1 | bv2;
    bv1 |= bv2;
    bv1.print();
    bv2.print();
    bv3.print();
}

void test3() {
    MyBitVector bv1("00010010");
    MyBitVector bv2("10111001");
    MyBitVector bv3 = bv1 ^ bv2;
    bv1 ^= bv2;
    bv1.print();
    bv2.print();
    bv3.print();
}

void test4() {
    MyBitVector bv1("00100100");
    MyBitVector bv2 = ~bv1;
    bv1 = ~bv1;
```

```
    bv1.print();
    bv2.print();
}

void test5() {
    MyBitVector bv1("00000001");
    bv1.set(2);
    bv1.print();
}

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

int main() {
    TEST(1);
    TEST(2);
    TEST(3);
    TEST(4);
    TEST(5);
    return 0;
}
```

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;
}
```

斐波那契数列

> —— 通过模板元编程实现编译期计算

斐波那契数列的定义如下：

- 如果 `N == 0`，`Fib(N)` 为 0
- 如果 `N == 1`，`Fib(N)` 为 1
- 对于任意 `N >= 2`，`Fib(N)` 为 `Fib(N - 1) + Fib(N - 2)`

参考相关教程（例如，*A gentle introduction to Template Metaprogramming with C++*），在给定代码框架的基础上实现通过模板元编程在编译期计算斐波那契数列，不需要考虑算术运算溢出问题。

输入

本题不需要处理输入。

输入格式参见代码框架的 `main` 函数。

输出

本题不需要处理输出。

输出格式参见代码框架的 `test_*` 系列函数。

提示

- 你只需要实现斐波那契数列递归定义的两个基本情况

扩展

- 了解 `constexpr` 关键字及其与 `const` 的不同
- 完成代码后，在 C++ Insights 上查看模板实例化的结果
- 探索使用模板元编程计算斐波那契数列是否有上限（同样可以借助 C++ Insights 实现）。如果有，上限是多少？能够调整这个上限？

代码框架

```cpp
#include <cstdint>
#include <functional>
#include <iostream>
#include <unordered_map>
```

```cpp
template <uintmax_t N>
struct Fib {
  static constexpr uintmax_t value = Fib<N - 1>::value + Fib<N -
2>::value;
};

// TODO: your code

void test_1() { std::cout << Fib<0>::value << std::endl; }

void test_2() { std::cout << Fib<1>::value << std::endl; }

void test_3() { std::cout << Fib<2>::value << std::endl; }

void test_4() { std::cout << Fib<5>::value << std::endl; }

void test_5() { std::cout << Fib<20>::value << std::endl; }

int main() {
  std::unordered_map<std::string, std::function<void()>>
test_cases_by_name = {
      {"test_1", test_1}, {"test_2", test_2}, {"test_3", test_3},
      {"test_4", test_4}, {"test_5", test_5},
  };
  std::string tname;
  std::cin >> tname;
  auto it = test_cases_by_name.find(tname);
  if (it == test_cases_by_name.end()) {
    std::cout << "输入只能是 test_<N>, 其中 <N> 可取整数 1 到 5." <<
std::endl;
    return 1;
  }
  (it->second)();
}
```