前言

Testlib.h 是由 CodeForces 的创办者 Mike Mirzayanov 打造的一份专注于数据层面的 C++ 功能库，其中包含了 generator，validator，interactor 和 checker 四个模块，基本上涵盖了出题流程中制造数据的流程。

这个项目存放在 GitHub 上。你可以从 这里 查看这个项目，或者直接从 Release 页面 下载整个库和示例文件。

模块介绍

Testlib.h 为四种环境提供帮助：

- **Generator**，也就是数据生成器。你可以使用内置的 `rnd` 随机库以及灵活的命令行参数操作生成一些输入数据。
- **Validator**，也就是数据校验器。在 **严格模式** 下，你可以将用户提供的数据识别精度达到每一个字符。
- **Interactor**，也就是交互库。你可以使用它实现交互题的库。
- **Checker**，也就是自定义判分器。如果你的题目需要细化得分需求，可以使用它实现。

注意事项

在 Testlib.h 中，你需要注意以下几个要求：

1. 不要使用 `srand()`，`rand()` 和 `random_shuffle()`。请使用 `rnd` 对象和 `shuffle()` 函数。
2. 在函数交互题目中，请告诉选手需要使用 `mt19937` 等方式随机生成一些数字。
3. 在函数交互题目中，请告诉选手不要使用和 `inf`，`ouf`，`ans` 等名字冲突的变量，特别是 `ans`。
4. 如果一些预处理的工作需要比较长的时间的话，可以放在答案文件中，Checker 只需要读入使用即可。
5. 在 Checker 中，即使输出文件有不必要信息，在返回 OK 的时候仍然需要读取干净，否则会判为错误。
6. Testlib.h 会利用命令行参数作为随机种子的生成依据，所以多次跑一个 Generator 并且不加命令行参数的话理论上来说会出现完全一样的结果。
7. 选手的答案可能比答案更加优秀，此时应该返回 Fail 并且让选手和出题人 / 出题组联系。

同时，下载的文件中包含了非常常用的 Checker 示例，如果题目只需要简单的判断（例如浮点数的判断），只需要直接把里面符合要求的代码放到 checker.cpp 里面就行了。

认识 Testlib.h 的输入输出重定向

由于 Testlib.h 使用自己的类型管理输入输出（`InStream`），而且在交互题中会涉及到 Checker、Interactor 和用户程序三者的交互，所以在一开始弄清楚所有的输入输出关系很重要。

对于 Generator 而言，它不会重定向输入输出。你可以直接和生成器进行交互，生成器也会直接将数据输出到控制台。接下来我将会展示三种编写方案：

1. 通过文件以及控制台的输入重定向实现输入。此时你真正的生成器（这里是因为 Generator 每次只会产生一个测试点）就需要把参数写在一个文件内，然后使用控制台重定向，例如 `gen.exe < opt.in` 或者是 `./gen < opt.in`。

```cpp
#include "testlib.h"
#include <iostream>

using namespace std;

```

```cpp
int main(int argc, char* argv[])
{
    registerGen(argc, argv, 1);

    int length; cin >> length;
    cout << rnd.next("[a-zA-Z0-9]{%d}", length) << endl;

    return 0;
}
```

这份代码支持读入一个长度，并且输出在这个长度下的随机字符串，其中每一个字符都是字母或者数字。

2. 通过命令参数传入。实际上，上面的 `argc` 和 `argv` 其实就代表了命令行输入的时候附加的参数。比如说，`a.exe 1 2 3` 的时候 `argc` 就是 4，而 `argv` 为一个字符串数组，内部存储的是 `["a.exe", "1", "2", "3"]`。所以可以通过命令行传入参数的方式实现和生成器的交互。你可以使用 `gen.exe 100 > 1.in` 创造一个数据点。

```cpp
#include "testlib.h"
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
    registerGen(argc, argv, 1);

    int length = atoi(argv[1]);
    cout << rnd.next("[a-zA-Z0-9]{%d}", length) << endl;

    return 0;
}
```

这份代码的功能和上面的代码一致。

3. 你可以使用 `startTest()` 等函数实现多数据点管理。这些函数将会在之后介绍。

```cpp
int main(int argc, char* argv[]){
    registerGen(argc, argv, 1);
    int a, b, c;
    for(int i = 1; i <= 10; i ++){
        startTest(i);
        a = rnd.next(100);
        b = rnd.next(100);
        c = rnd.next(100);
        genData(a, b, c);
    }
    return 0;
}
```

对于 Validator 来说，初始化完毕之后，出于检测需要，Testlib.h 提供了一个读入的对象—— `inf`。为了保证严格检验，你应当 **只使用** 这个类读入。检测完毕或者出现错误的时候，将会在控制台打印信息并且退出。

```cpp
/**
 * Validates that input contains the only integer between 1 and 100, inclusive.
 * Also validates that file ends with EOLN and EOF.
 */

#include "testlib.h"

using namespace std;

int main(int argc, char* argv[])
{
    registerValidation(argc, argv);

    inf.readInt(1, 100, "n");
    inf.readEoln();
    inf.readEof();

    return 0;
}
```

这个例子来自 `/validators/ival.cpp`，功能为检测输入文件是否只包含一行一个在 1 到 100 范围内的整数，并且检测是否有恰好一个回车且没有其余字符。

在 Checker 中，Testlib.h 也会对输入进行重定向。其中会提供以下对象：

- `inf`：输入文件。
- `ouf`：选手的输出。
- `ans`：答案文件。

三者均为读入流。你应当使用对象内专门的函数读取信息。

```cpp
#include "testlib.h"
#include <string>

using namespace std;

const string YES = "YES";
const string NO = "NO";

int main(int argc, char * argv[])
{
    setName("%s", (YES + " or " + NO + " (case insensetive)").c_str());
    registerTestlibCmd(argc, argv);
```

```
13
14      std::string ja = upperCase(ans.readWord());
15      std::string pa = upperCase(ouf.readWord());
16
17      if (ja != YES && ja != NO)
18          quitf(_fail, "%s or %s expected in answer, but %s found", YES.c_str(),
    NO.c_str(), compress(ja).c_str());
19
20      if (pa != YES && pa != NO)
21          quitf(_pe, "%s or %s expected, but %s found", YES.c_str(), NO.c_str(),
    compress(pa).c_str());
22
23      if (ja != pa)
24          quitf(_wa, "expected %s, found %s", compress(ja).c_str(), compress(pa).c_str());
25
26      quitf(_ok, "answer is %s", ja.c_str());
27  }
```

这个例子来自 `./checkers/yesno.cpp`，用处是判断单组数据中大小写不敏感的 YES 和 NO 的正确性。

洛谷在评测交互题的时候，评测机首先会将用户程序和 Interactive Library 两个文件拼接在一起编译（如果只是想用 Checker 或者直接弄 IO 交互的话那就把这个文件留空就好了，这样就相当于只编译选手文件），然后使用这个编译出来的可执行文件和 Checker 进行交互。因此，如果是函数式交互，那么里面除了 `registerInteraction` 之外应该和 IO 交互的写法一致。

由于这种编译方案，洛谷的交互库编写思路会和 CodeForces 完全不一样。CodeForces 的思路是先调用交互库进行 IO 交互，将交互的信息通过 `tout` 流输出到一个文件中，然后运行 Checker 并提供这个文件的内容作为选手输出，从而实现整个交互流程。而洛谷选择的是直接让用户的程序和 Checker 交互，稍显鲁莽。

对于 IO 交互时的选手程序来说，输入流接受 Checker 输出的内容，而输出流将内容输出给 Checker。

对于函数交互时的交互库来说，输入流和 `inf` **同时接受** 来自 Checker 输入的内容，而输出流将内容输出给 Checker。

对于 Checker 来说，`inf` 用于读入输入文件的信息，然后可以使用输出流向选手程序或者交互库提供信息。对方输出的信息可以通过输入流或者 `ouf` 读入。`ans` 可以读入答案文件。

虽然这种方法看似是把函数交互换成了 IO 交互，其实不然。函数交互时只需要让 Checker 向交互库发送有用信息，然后交互库自行测试并将结果输出给 Checker 就行。

你可以直接从 洛谷的交互题功能说明 中看到例子。不过需要说明的是，使用交互库而不使用 Special Judge 似乎会带来一些问题，所以建议两个都使用。

目前除了实现 IO 交流层之外，没有比较好的测试交互题的方法。

以下介绍基于 Testlib v0.9.12。我将会将 Testlib.h 的内容拆成若干个部件进行介绍。

本地测试的命令行参数

这一个部分主要讲述的是本地测试时命令行需要提供的参数。本地编译的时候，请加上 `-std=c++11 -O2` 开关。

对于 Generator 而言，Testlib 并不会对其参数进行验证，只会用于生成随机种子。所以你可以填入任意内容。你可以使用上面提到的方式，将参数作为输入控制数据生成。

对于 Validator 而言，需要按照下面的方式编写参数：

```
val.exe [--testset testset] [--group group] [--testOverviewLogFileName fileName]
```

其中 `testset` 和 `group` 可以设置这一个测试点所属的子任务编号和组编号，`testOverviewLogFileName` 是 Validator 的信息需要被输出的地方。具体的 `Validator` 操作我们将会在之后谈到。

对于 Checker 而言，需要按照下面的方式编写参数：

```
check.exe <Input_File> <Output_File> <Answer_File> [<Result_File> [-appes]]
```

其中前三个文件分别是输入文件、选手输出和答案文件，为必选参数。后面的则为可选参数，其中 `Result_File` 指定了错误信息需要输出的文件，而 `-appes` 是在此基础上将输出改成 xml 可以识别的格式。例如：

```
---------------------------- Without -appes -----------------------------
answer is NO
----------------------------- With -appes ------------------------------
<?xml version="1.0" encoding="windows-1251"?><result outcome = "accepted">answer is
NO</result>
```

对于 Interactor 而言，需要按照下面的方式编写参数：

```
interactor.exe <Input_File> <Output_File> [<Answer_File> [<Result_File> [-appes]]]
```

其中 `Input_File` 表示输入文件（在洛谷为 Checker 提供的信息），`Output_File` 为需要传输给 Checker 的信息（这个文件被链接到了一个叫做 `tout` 的输出流，用法和 `cout` 相同）。`Answer_File` 为答案文件，剩余参数意义和 Checker 一致。

在运行完毕之后，可以通过返回值确定错误信息。

代码	返回值	含义
_ok	0	正确
_wa	1	错误
_pe	2	格式错误
_fail	3	运行失败，程序出错
_dirt	4	输出文件含有多余信息
_points	5	部分分数
_unexpected_eof	8	文件读完时仍然尝试读入
_partially	16+分数	部分正确

`_partially` 和 `_points` 不同的是，前者只支持 `0` 到 `100` 之内的整数作为分数，并且返回值需要累加 16，作为 `quitf` 等函数的第一个参数；而 `_points` 支持浮点数分数，同时有专门的返回函数。

Testlib.h 同时内置了一个命令行参数解析函数。比如说，在调用如下命令时：

```
gen.exe -n10 -m 200000 -t=a -increment
```

这相当于：设置 `n, m, t` 分别为 `100, 200000, a`，同时启用了 `increment` 开关。

那么在代码中，你可以使用 `opt<T>(str)` 获取参数。其中 `T` 是变量类型，而 `str` 表示变量的名称。在调用后，Testlib.h 将会自行解析命令行参数，并且转换类型后返回。例如：

```
int n = opt<int>("n");
long long m = opt<long long>("m");
string t = opt("t");
bool increment = opt<bool>("increment");
```

在 `opt` 函数的加成下，代码的命令行参数会变得更加便于读入。

基础函数

对于每一份 Testlib.h 的代码，我们都需要引用其头文件：

```
#include "testlib.h"
```

并且将 testlib.h 和你的代码文件放在同一个文件夹下，这样就能够被编译器识别。如果你比较熟悉头文件引用规则，也可以直接放在编译器的 include 文件夹中，这样就可以和标准库一样调用。

在程序的 `main` 函数中需要加入一些参数，从而获取从命令行传入的内容：

```
int main(int argc, char* argv[]) { ... }
```

其中 `argc` 表示参数（包括这个程序的名字）的个数，`argv` 是从 0 开始编号的参数列表。

在 `main` 函数中，第一句话需要让 Testlib 获取并且绑定你的参数。具体如下：

- Generator：`registerGen(argc, argv, 1)`，其中最后一个数字代表随机生成器的版本号，写 1 即可。
- Validator：`registerValidation(argc, argv)`。
- Checker：`registerTestlibCmd(argc, argv)`。同时，你也可以使用 `registerTestlib(3, "in.txt", "out.txt", "ans.txt")`，这样就不需要在命令行再输入参数，方便本地调试。不过提交到 OJ 上的时候，只能使用 `registerTestlibCmd`。
- Interactor：`registerInteraction(argc, argv)`。

整个程序应该以判分语句结尾。以下为判分语句：

- `void quitf(TResult result, const char* msg, ...)` 表示确定了选手代码的错误类型或者是判为正确，并且发送相关信息。其中 `result` 是上面的程序返回值表格中第一列的变量，直接使用即可，而 `msg` 就是提示信息。你可以继续传参对消息串进行格式化，例如 `quitf(_ok, "%d lines", n);`。
- `void quitp(T points, const char* format, ...)` 表示为选手提供部分分。其中 `points` 为一个浮点数，其中储存一个在 $[0,1]$ 范围的数字，表示得分比例。而后面的就是提示信息，和上面一样。
- `void quitif(bool condition, TResult result, const char* format, ...)` 表示在 `condition` 为真的时候确定结果，每一个变量的功能和 `quitf` 大致相同。

程序在确认得分情况之后会直接终止并且返回结果。

你可以在代码中使用 `void ensure(bool condition)` 函数，在 `condition` 为假的时候将会直接判作 `FAIL` 并且返回该条件的信息。你也可以使用 `void ensuref(bool cond, const char* format, ...)` 自定义错误消息。

需要注意的是：`ensure() / ensuref()` 函数在全局以及 `InStream`（就是 `inf, ouf, ans` 的类型）中都有定义。如果调用前者将会直接判为 `FAIL`，否则将会检查当前 `InStream` 连向的是否为选手的输出（也就是 Checker 中 `ouf.ensure()` 或者 `ouf.ensuref()`），如果是的话将会判为 `WA`，否则仍然判定为 `FAIL`。

随机类 random_t

在 Testlib.h 中，提供了一个随机类型，同时定义了 rnd 方便调用类型内部的函数。这个类型使用伪随机。下面是其中 `nextBits(int bits)` 的实现，含义是获取“随机”的 `bits` 位整数。大家可以作为题目中伪随机数构造方式的参考。

```
long long nextBits(int bits)
{
    if (bits <= 48)
    {
        seed = (seed * multiplier + addend) & mask;
        return (long long)(seed >> (48 - bits));
    }
    else
    {
        if (bits > 63)
            __testlib_fail("random_t::nextBits(int bits): n must be less than 64");

        int lowerBitCount = (random_t::version == 0 ? 31 : 32);

        long long left = (nextBits(31) << 32);
        long long right = nextBits(lowerBitCount);

        return left ^ right;
    }
}
```

以下是默认的参数：

```
random_t()
    : seed(3905348978240129619LL)
{
}
// ...
const unsigned long long random_t::multiplier = 0x5DEECE66DLL;
const unsigned long long random_t::addend = 0xBLL;
const unsigned long long random_t::mask = (1LL << 48) - 1;
```

`random_t` 中支持两种 `setSeed()` 形式：通过 `argc, argv` 设置，以及直接传入一个数字设置。在每次运行 Testlib 程序的时候，必然会调用一次 `rnd.setSeed(argc, argv)` 获取种子。

```
/* Sets seed by command line. */
void setSeed(int argc, char* argv[])
{
    random_t p;

    seed = 3905348978240129619LL;
    for (int i = 1; i < argc; i++)
    {
        std::size_t le = std::strlen(argv[i]);
        for (std::size_t j = 0; j < le; j++)
            seed = seed * multiplier + (unsigned int)(argv[i][j]) + addend;
        seed += multiplier / addend;
    }

    seed = seed & mask;
}

/* Sets seed by given value. */
void setSeed(long long _seed)
{
    _seed = (_seed ^ multiplier) & mask;
    seed = _seed;
}
```

然后就可以调用接下来的函数了：

- `T next(T n)` 函数，将会返回在 $[0, n)$ 内和 n 同类型的数。`T` 可以是 `int, unsigned int, long long, unsigned long long, double`。
- `T next()` 函数，等价于 `next(1.0)`。
- `T next(T from, T to)` 函数，在 `T` 是整数类型时返回 $[from, to]$ 范围内的任意一个整数，否则返回 $[from, to)$ 中的任意一个浮点数。`T` 的定义范围和上面一样。
- `std::string next(std::string ptrn)` 函数，将会把 `ptrn` 视为一个正则表达式类 `pattern`，并且返回匹配正则表达式的一个随机字符串。我们将会马上介绍 `pattern` 类。
- `std::string next(const char* format, ...)` 函数，会先使用其余参数格式化字符串，然后和上面的步骤一致。例如：`next("[0123]{1,%d}", 100)` 等价于 `next("[0123]{1,100}")`。
- `typename Container::value_type any(const Container& c)` 函数，随机返回容器内的一个数值。
- `typename Iter::value_type any(const Iter& begin, const Iter& end)` 函数，随机返回 $[begin, end)$ 中一个迭代器的值。
- `T wnext(T n, int type)` 函数为带权随机。`type` 决定了返回的值在分布列中最大值的偏移程度，`type` 越大，返回数字就越有可能是大数。具体的实现分以下几种类型：
  - 如果 `type` 等于 0，函数功能和 `next(n)` 一致。
  - 如果 `type` 大于 0，函数将会随机出 $type + 1$ 个在 $[0, n)$ 范围内的数字，并且取 **最大值** 返回。

- 如果 `type` 小于 0，函数将会随机出 $-type+1$ 个在 $[0,n)$ 范围内的数字，并且取 **最小值** 返回。
- 如果 $|type|$ 不小于 `random_t::lim`（在实现中是 25），将会使用其他方式构造：

```
// int wnext(int n, int type)
double p;

if (type > 0)
    p = std::pow(next() + 0.0, 1.0 / (type + 1));
else
    p = 1 - std::pow(next() + 0.0, 1.0 / (-type + 1));

return int(n * p);
```

`T` 的定义范围和上面一样。同时，前面的 `next` 在追加上 `type` 之后也可以作为 `wnext` 的参数，范围跟随一开始的 `next` 函数，生成方式一致。

- `typename Container::value_type wany(const Container& c, int type)` 函数，等价于 `*(c.begin() + wnext(c.size(), type))`。
- `typename Iter::value_type wany(const Iter& begin, const Iter& end, int type)` 函数，等价于 `*(begin + wnext(end - begin, type))`。

正则表达式类 pattern

在 Testlib.h 中，你可以在一个精简版的正则表达式规则的基础上创建正则表达式类。接下来我们将会简单介绍一下这个精简后的正则表达式规则。它支持：

- **字符集**：由 `[]` 包裹起来的块，里面可以填入一系列的字符，可以匹配这些字符的任意一个。你可以使用 `-` 填充两个字符之间的所有字符。例如：`[A-Za-z0-9]` 匹配一个字母或者一位数字，`[^A-Z]` 匹配除了大写字母外的所有字符。如果需要匹配 `^`，需要转义为 `\\^`。在 Testlib 中，字符集中一个字符的个数也会决定其被选中的可能性。
- **范围**：由 `{}` 包裹起来的数字或者数据对，分别表示匹配的数量和匹配数量的范围。例如：`[A-Z]{100}` 匹配 100 个大写字母组成的字符串，`[1-9][0-9]{0-99}` 匹配一个最多 100 位的正整数字符串。
- **或** 运算符：表示匹配两者中的任意一个。比如：`0|[1-9][0-9]{0-99}` 匹配一个最多 100 位的自然数字符串。
- **可选** 运算符：表示匹配最多一个。比如：`0|(-?[1-9][0-9]{0-99})` 因为加入了负号的匹配，可以匹配一个最多 100 位的整数字符串。
- **闭包** 运算符：表示匹配若干个。其中有 `*`（可以不匹配）和 `+`（至少匹配一个）两种。例如：`[A-Z]+` 匹配不为空的大写字母字符串，而 `[1-9][0-9]*` 匹配任意一个正整数字符串。

由于基于贪婪匹配机制，`1` 在匹配 `[0-9]?1` 的时候将会直接和 `[0-9]?` 匹配，而不是 `1`。

你可以使用如下方式生成一个正则表达式类：

```
pattern ptrn("[1-9][0-9]*");
```

然后可以调用以下函数：

- `std::string next(random_t& rnd)` 函数，使用 `rnd` 内的种子信息生成一个匹配该正则表达式的字符串。`rnd.next(str::string ptrn)` 基于这个函数实现。

- `bool matches(const std::string& s)` 函数，检测字符串是否匹配正则表达式。
- `std::string src()` 函数，获取正则表达式的字符串形式。

读入类 InStream

这是 Testlib.h 中最重要，也是功能最丰富的类。同时，这一部分的介绍也会比较多。

`InStream` 类包含三个子类：

- `StringInputStreamReader`：基于一个字符串实现的读入流。
- `FileInputStreamReader`：基于一个 `FILE*` 对象实现的读入流，使用在和标准输入输出流直接连接的对象上。
- `BufferedFileInputStreamReader`：基于一个 `FILE*` 对象实现的带缓存读入流，使用在和文件直接连接的对象上。

  缓存的机制为：定义缓存长度 2000000 和最大回退字符数量 1000000。在最开始的时候先将缓存读满，如果读取完毕，为了保证可以回退字符，并且回退字符是需要钦定字符内容的，所以我们可以考虑将指针定在缓存右数最大回退字符数量个字节，从此开始读入直到抵达缓存尾端。

每个 `InStream` 类型会附带一些参数：

- `stdfile`：这个对象是否和标准输入输出流（`stdin`，`stdout`，`stderr`）连接。比如说，Validator 模式下的 `inf` 变量连接了 `stdin`，这个参数就是 `true`，而 Checker 模式下 `inf` 变量连接了输入文件，这个参数就是 `false`。
- `strict`：这个对象对读入的处理是否严格。在严格模式下，读入数字、字符串等将不会自动跳过空白字符，需要使用 `readSpace()`、`readEoln()`、`readEof()` 等函数去除空格并检查多余字符。只有 Validator 的 `inf` 变量会采用严格模式。
- `mode`：这个对象表示这个对象连接的类型，为“读入文件”、“选手输出”、“答案文件”之一。

我们注意到 `StringInputStreamReader` 并没有使用。在平常对字符串的处理中，我们可以使用 `stringstream` 进行读入，但是当我们希望使用 Testlib 的读入特性处理字符串时，就可以使用这个类型了。比如说，我想使用 `inf` 的参数作为基础建立一个字符串读取流，就可以使 `InStream str_reader(inf, "123 456")` 的方式创建。

```cpp
#include "testlib.h"
#include <string>
#include <iostream>

using namespace std;

int main(int argc, char * argv[]){
    registerTestlibCmd(argc, argv);
    InStream str_reader(inf, "123 456");
    int x = str_reader.readInt();
    cout << x << endl;
    quitf(_ok, "");
}
```

以上代码将会输出 123。

接下来我将会展示所有可用的函数。其中一些函数具有某些特性，我将会函数的前面标注：

- 范围可控性（^）：对于数值读入，可以在参数后面追加 `minv, maxv` 两个参数，以检查获取的数字是否在 $[minv, maxv]$ 之内。对于其余读入会添加说明。
- 变量名自定（!）：在添加值域范围的前提下，可以在参数的后面再追加字符串，或者待格式化字符串及其参数，代表这个变量的名字， 在变量超出范围的时候就可以在错误的信息中告诉选手哪个变量超出边界了。**对于读入序列的函数，不允许使用待格式化字符串。**

  你可以对 已经设置变量名称 的 序列读取函数 后面继续追加参数 `indexBase`。`indexBase` 主要用于在读入发生错误向选手提供信息的时候附加上错误位置的下标。

- `void skipBlanks()` 函数，不断跳过空白字符直到遇见非空白字符或者到达末尾。
- `bool curChar()` 函数，获取当前指向的字符。
- `void skipChar()` 函数，跳过当前指向的字符。
- `char nextChar()` 函数，读入并返回当前指向的字符。
- [^] `char readChar()` 函数，效果和 `nextChar()` 一样，但是此时范围可控性并不是限制在一个区间内，而是一个字符，例如：`readChar('a')`。
- `char readSpace()` 函数，等价于 `readChar(' ')`，主要用于 Validator 中跳过空格。
- `void unreadChar(char c)` 函数，在输入流的最前面插入 `c` 这个字符。此时所有字符并挪到了 `c` 后面，当前字符变为 `c`。
- `bool eof() / bool seekEof()` 函数，检测当前字符是否为 EOF。`seekEof()` 函数会跳过空白字符后再判断。
- `bool eoln() / bool seekEoln()` 函数，检测当前字符是否为回车符。注意：在严格模式下，可以判断 `CRLF (#13#10)` 和 `LF (#10)` 两种回车方式。如果是回车符，将会自行读入这个回车符来到下一行第一个字符。`seekEoln()` 函数会跳过空格和 Tab 后再判断。
- `void nextLine()` 函数，等同于 `readLine()`，但是舍去了返回值。
- `readWord() / readToken()` 系列。
  - [^!] `std::string readWord() / std::string readToken()` 读取一个字符串，其内部不包含空白字符。你可以添加 `pattern` 类或者正则表达式字符串作为第一个参数，在读入完成后进行正则匹配。建议使用后者。
  - [^!] `std::vector<std::string> readTokens(int size, const std::string& ptrn);` 函数，读入一个字符串数组并返回结果。你也可以使用 `readWords()`。其中正则匹配测试是必须的，同时也可以替换成 `pattern` 类。
  - [^!] `void readWordTo(std::string& result) / void readTokenTo(std::string& result)` 函数，读入一个字符串并且储存在 `result` 中。你可以添加 `pattern` 类或者正则表达式字符串作为限制。
- 数值读入函数组合。
  - [^!] `T read???()` 函数，读入一个数字并且返回。
  - [^!] `std::vector<T> read???s(int size, T minv, T maxv)` 函数，读取一个数组序列并且返回。其中数值范围限制是必须的。
  - [^!] `double readStrictReal(double minv, double maxv, int minAfterPointDigitCount, int maxAfterPointDigitCount)` 函数使用严格的浮点数读入方法读入浮点数。你也可以使用 `readStrictDouble()`。其中不仅限制了数值范围，同时限制了小数点后位数的范围，并且不允许出现科学计数法和多余的小数点。

- [^!] `std::vector<double> readStrictReals(int size, double minv, double maxv, int minAfterPointDigitCount, int maxAfterPointDigitCount);` 函数使用严格的浮点数读入方式读入浮点数数组。你也可以使用 `readStrictDoubles()`。其中数值范围限制是必须的。

  对于前两个函数，每一个类型对应的 `???` 如下表：

类型	???
int	Int / Integer
long long	Long
unsigned long long	UnsignedLong
double	Real / Double

- `readString() / readLine()` 系列。除了读入变成一整行字符串并且自动读取回车符之外，和 `readWord() / readToken()` 系列的函数支持大体相同，限于篇幅不展开叙述。你可以查看 `testlib.h` 第 1831 到 1870 行找到这些函数。

  不过在这里，读入字符串序列的时候，可以直接使用 [!] `std::vector<std::string> readLines(int size, int indexBase = 1)` 读入，也就是不需要检测字符串的格式。
- `readEoln()` 函数，读入一个回车符或者失败。建议使用在 Validator 中检查格式。
- `readEof()` 函数，检查是否到达文件末尾。建议使用在 Validator 中检查格式。
- `close()`，`xmlSafeWrite()`，`init()`，`reset()` 等函数，由于用不上所以不会展开。

数据检验时传输特殊信息

在之前的命令行参数部分我们就提到了 Validator 具有和其他三者完全不同的参数。实际上，在 Testlib 中有一个专门的变量处理这件事情—— `validator`。这个部分主要就是讲解一下 `Validator` 类型的构造，以及其日志文件的组成部分。

首先，`validator` 会解析传入的参数，并且获取这个测试点的特殊信息：

- `testset`：测试部分信息，也就是常说的子任务。
- `group`：测试点特殊信息，也就是常说的一些数据点满足的信息。
- `testOverviewLogFileName`：测试点满足的所有信息的总览。

在代码中，你可以使用 `validator.testset()`，`validator.group()` 和 `validator.testOverviewLogFileName()` 分别获取这三个信息。信息的返回类型均为字符串。代码中需要添加这些部分的原因是：在 Polygon 中，每个测试点将会被分配一个数据组和特殊类型，随后就需要使用这个校验器确认数据是否符合特殊信息。你可以在 Polygon 上直接查看到日志文件。

```
/**
 * Validates that input depenging on testset and group.
 */

#include "testlib.h"
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
```

```
{
    registerValidation(argc, argv);

    int n, m;

    if (validator.testset() == "pretests")
    {
        n = inf.readInt(1, 10, "n");
        inf.readSpace();
        m = inf.readInt(1, 10, "m");
    }
    else
    {
        n = inf.readInt(1, 100, "n");
        inf.readSpace();
        m = inf.readInt(1, 100, "m");
    }

    if (validator.group() == "even-n-and-m")
    {
        ensure(n % 2 == 0);
        ensure(m % 2 == 0);
    }

    addFeature("n equals m");
    if(n == m)
        feature("n equals m");

    inf.readEoln();
    inf.readEof();
}
```

以上代码更改自 `./validators/validate-using-testset-and-group.cpp`，里面涉及到了若干个需求：数据分为两个部分，其中 `pretest` 部分要求 $n,m \leq 10$，其余要求 $n,m \leq 100$；有一部分数据需要满足 $2|n \wedge 2|m$。其中数据点还有一个特殊性质—— $n=m$。此时我们尝试以 `pretest` 作为所属测试集合，并且强制要求特殊数据性质，那么我们需要输入：

```
val.exe --testset main --group normal --testOverviewLogFileName val.txt
```

在输入完毕后，你将会在 val.txt 中看到结果。如果我们输入 `2 10\n^Z`，此时日志内容如下：

```
"m": max-value-hit
"n":
feature "n equals m":
```

前半部分为变量范围检测部分，其中每个变量按照其字典序进行排序，并且显示了在该数据点的限制下，等于其范围的最大值或是等于最小值。所有的范围在读入后自动加入，也可以通过函数加入。**注意，为了防止数组元素过多导致变量数量膨胀，所有包含数字的变量都不会统计在内。**

后半部分为特殊性质检测部分，其中我们定义了一个性质为 `n 等于 m`，如果这个数据点满足这个性质，那么就会在该 `feature` 后面出现一个 `hit`。

正如前面所说，你可以使用类似于 `validator.addBoundsHit("k", ValidatorBoundsHit(false, true))` 的方法手动添加一个范围检测。其表示 `k` 变量等于其范围内的最大值。

其他函数

在 Testlib.h 中也提供了一些实用函数，方便在代码中直接调用。

函数	功能
`doubleCompare(expected, result, MAX_DOUBLE_ERROR)`	判断期望值和实际值在精度范围下是否相同，相同则返回 `false`
`doubleDelta(expected, result)`	计算期望值和实际值的相对误差和绝对误差的最小值
`isEof(c) / isEoln(c) / isBlanks(c)`	判断字符是否为 EOF / 换行符 / 空白字符
`disableFinalizeGuard()`	在程序判断代码得分的时候，取消对输出文件多余内容的检查
`vtos(x) / toString(x)`	将任意一个可以被输出的变量转化为字符串并返回
`shuffle(first, last)`	随机排列迭代器区间内的元素（不能使用 `random_shuffle`）
`startTest(id)`	用于生成多个数据。将数字 id **直接作为**当前数据的储存文件名称 (*)
`upperCase(s) / lowerCase(s)`	将字符串所有字母转换成大写 / 小写并返回
`compress(s)`	通过保留前后部分将字符串长度压缩至 64 个字符内并返回
`englishEnding(x)`	返回英文中 x 的序数缩写
`trim(s)`	去除字符串前后的空白字符并返回
`join(first, last, seperator)`	将 $[first, last)$ 之间的值通过 seperator（默认为空格）连接成一个字符串并返回
`split(s, separator)`	将字符串通过 seperator 拆分成字符串列表并返回。其中 seperator 可以为字符或者字符串
`tokenize(s, seperator)`	和 split 功能相同，但是保证列表中没有空字符串

(*) ：如果你想要加 `.in` 后缀等，请自行前往 `testlib.h` 中第 3967 行进行修改。