Intervals

Submission deadline:	**2019-04-07 23:59:59**
Late submission with malus:	**2019-06-30 23:59:59** (Late submission malus: 100.0000 %)
Evaluation:	**5.7600**
Max. assessment:	**5.0000** (Without bonus points)
Submissions:	27 / 25 Free retries + 20 Penalized retries (-2 % penalty each retry)
Advices:	2 / 2 Advices for free + 2 Advices with a penalty (-10 % penalty each advice)

The task is to develop a C++ classes `CRange` and `CRangeList`. The classes will represent an interval of integers and a list of integer intervals, respectively.

Assume an interval of integers. The interval may be either just 1 number or several numbers. Intervals of length 1 number will be denoted as a single integer, e.g., `42`. Longer intervals will be denoted as two points (the boundaries) in angle braces. For example, interval `<1..5>` represents integers `1,2,3,4,5`. The intervals are implemented by the `CRange` class. The integer/boundaries in `CRange` will be stored as `long long` values (negative values are supported). The interface of `CRange` is:

constructor (lo, hi)
: initializes the interval from `lo` to `hi`. The constructor checks the boundaries, i.e., `lo ≤ hi`. If the condition fails, the constructor throws `InvalidRangeException` (the exception class is implemented in the testing environment).

further
: further interface may be needed, study the attached source and add methods to support the desired behavior. You are free to add your own methods that are needed for your implementation. The only limit is the size of the instance, two `long long` integers maximum.

Class `CRangeList` implements a list of intervals. The class must be efficient when storing the intervals, i.e. overlapping intervals will be merged. The interface of `CRangeList` offers methods to add / remove intervals, compare instances, and print to an output stream. Moreover, there is an extended interface (in the bonus test) that adds more functionality. The required interface is:

default constructor
: initializes an empty list of intervals,

destructor, copy constructor, op=
: implemented if your representation requires them. In general, we recommend to use an appropriate STL container and let the compiler implement the copying/destruction.

`Includes(long long) / Includes (CRange)`
: the methods test whether the list of intervals includes the given value / interval. The methods return `true` (contains the value/interval) or `false` (does not contain).

`+=`
: this operator adds interval(s) to the list, indeed it is a set union operation. The operator is responsible for the compact representation, i.e., it must merge the overlapping intervals. Since the intervals represent integers, the merging is a bit unusual. Naturally, the following intervals are merged:

  - `<10..20>` and `<15..30>`,
  - `<10..30>` and `<15..18>`, and
  - `<10..20>` and `<20..30>`,

  Surprisingly, intervals `<10..20>` and `<21..30>` are merged although they do not overlap or touch. Nevertheless, they together cover all integers from in the range, thus represent interval `<10..30>`. On the other hand, intervals `<10..19>` and `<21..30>` are not merged, number 20 is not included.

`-=`
: this operator is complementary to operator +=. The interval(s) on the right-hand side are removed from the interval list. The operation may be seen as a set difference. Once again, the representation must be efficient, thus unneeded intervals must be removed.

`=`
: this operator may be used to replace the contents with the intervals on the right-hand side.

`==, !=`
: the operators compare two instances of `CRangeList`. Two instances are identical if the list of intervals is the same.

`<<`

the operator prints out the contents to the given output stream. The resulting list will be enclosed in curly braces, individual intervals are separated by a comma. The list is printed in an ascending order. Intervals of length 1 are printed as a single integers, longer intervals are printed in angle braces. The output format is clear from the attached examples.

The task offers a bonus test for extra points, the bonus requires additional interface that simplifies some operations. You are free to either implement the extra interface, or not. However:

- if you decide not to implement the extra interface, keep the preprocessor directive `EXTENDED_SYNTAX` in a comment. The additional construct will not be compiled, bonus test will be skipped, and bonus points will not be awarded,
- if you decide to implement the extra interface, define preprocessor directive `EXTENDED_SYNTAX` (i.e., remove the comment). In this case, the testing environment will include the extra test in compilation and will execute the bonus test.
- Caution. The compilation fails if you uncomment the `EXTENDED_SYNTAX` directive without the implementation of the extra interface.

The extra interface required for the bonus test:

constructor to initialize `CRangeList` from a list of values
: the list of integers is interpreted as the intervals included in the new instance,

range for loop
: range for loops must be able to handle `CRangeList` instances. The intervals will be iterated in an ascending order,

output operator must always display the output in radix 10, however, it must preserve the former settings of the `ostream` object.

Submit a source file with your implementation of `CRange` and `CRangeList` classes. The submitted file shall not contain any #include directives nor `main` function. If your `main` function or #include remains in the file, please place them into a conditional compile block.

This task does not provide the required class interface. Instead, you are expected to develop the interface yourself. Use the description above, the attached examples, and your knowledge of overloaded operators.

There are bonus tests included in this homework, these tests try the following:

- The first bonus test checks whether your implementation supports the extra syntax, or not (see above).
- The second bonus tests the efficiency of `Includes`. Faster than linear solution is required.
- Finally, the third bonus tests the efficiency of operators +, +=, -, and -=. Faster than linear solution is required. The third bonus is difficult to achieve, it reauires rather long code. This is the reason why the reference solutions is longer than usually. A program that passes all tests except the last bonus is about half of the length of the reference.

Advice

- The testing environment uses output operator (<<) to examine your instance. If your overloaded operator << does not work properly, the tests will be negative.
- Implement the output operator properly -- do not blindly send the data to `cout`. Instead, send the data to the output stream passed as the parameter. Do not add any extra whitespace/newline characters.
- If your program does not compile (and especially if it compiles locally, however it does not compile in Progtest), there might be some problem in your interface design. Check your operator overloads, pay special attention to the `const` qualifiers.
- Operators +=, -=, and = are tricky, a list of intervals must be accepted on the right hand side. Do not start your implementation until you are sure how to handle the list.
- STL classes `std::vector`, `std::list`, and `std::string` are available. However, the rest of STL is not.
- Interval boundaries are `long long` integers. Borderline tests include `LLONG_MIN` and `LLONG_MAX` values. A special care must be paid to the comparisons.
- Do not use 128 bit data types. An attempt to do so results in a compile error.

A correct solution of this homework may be used for code review. A solution is considered correct if it passes all mandatory tests for 100%. Your solution does not have to pass the bonus tests to be eligible for code review.

Sample data: Download

☐ **Reference**

27 **2019-04-03 11:49:39** Download