

Custom enumerators

Types of enumerators

- ❑ Until now, only **standard enumerations** of well-known collections were shown, where creating an enumerator class is not necessary.
- ❑ **Custom enumerator** is that
 - does not traverse the items of the collection in a usual way, or
 - uses more items of the collection in one step, or
 - merges more collections
- ❑ **An enumerator may be considered as custom, if a standard enumerator**
 - does not start with operation `first()` because it is already in state “in operation”, or
 - ends before reaching the last item of the collection, or
 - there is no real collection behind the enumerator

1st task

Given two function $f : \mathbb{N} \rightarrow \mathbb{R}$ and $g : \mathbb{N} \rightarrow \mathbb{R}$, and a number $e \in \mathbb{R}$. There exist two arguments, i and j , for which $f(i)+g(j)=e$ holds. Give such i and j !

$A : e:\mathbb{R}, i:\mathbb{N}, j:\mathbb{N}$

$Pre : e = e_0 \wedge \exists i, j \in \mathbb{N} : f(i)+g(j)=e$

$Post: Pre \wedge f(i)+g(j)=e$

Unfortunately, there is no clue of the algorithmic pattern in this specification.

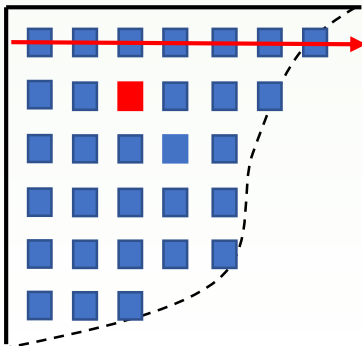
Idea

Arrange the values of $f(i)+g(j)$ in an infinite **matrix**, where indexes start from 0 and value $f(i)+g(j)$ is located in the i^{th} row and j^{th} column in the matrix.

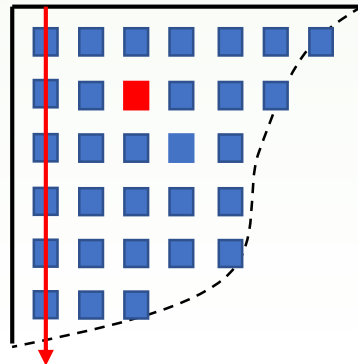
Define an enumerator that traverses the indexes of this matrix.

Standard enumeration (row- and column major order) is not good, since the rows and columns are infinite long. It would not traverse the second and higher rows/columns.

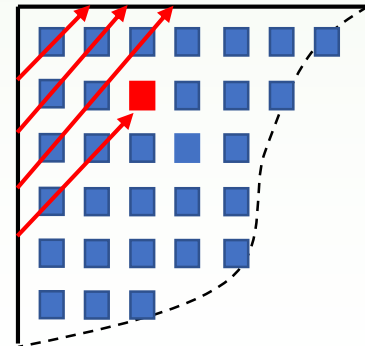
row-major order



column-major order



antidiagonal order



Plan

There is no collection behind this enumerator,
just a theoretical matrix traversed in an unusual way.

$A : t:\text{enor}(\mathbb{N} \times \mathbb{N}), e:\mathbb{R}, i:\mathbb{N}, j:\mathbb{N}$

$Pre : t = t_0 \wedge e = e_0 \wedge \exists i, j \in \mathbb{N}: f(i) + g(j) = e$

$Post : e = e_0 \wedge (i, j) = \text{SELECT}_{(i, j) \in t_0} (f(i) + g(j) = e)$

Selection:

$t:\text{enor}(E) \sim t:\text{enor}(\mathbb{N} \times \mathbb{N})$

$e \sim (i, j)$

$\text{cond}(e) \sim f(i) + g(j) = e$

$t.\text{first}()$

$\neg \text{cond}(t.\text{current}())$

$t.\text{next}()$

$i, j := t.\text{current}()$

Enumerator of the pair of indexes

The enumerator pretends to give the next row- and column indexes of an existing matrix.

enor($\mathbb{N} \times \mathbb{N}$)	Sequence of index pairs.		
$(\mathbb{N} \times \mathbb{N})^*$	first()	next()	current()
$i, j: \mathbb{N}$	$i, j := 0, 0$	if $i > 0$ then $i, j := i - 1, j + 1$ elsif $i = 0$ then $i := j + 1 ; j := 0$	(i, j)

operation end() is not needed

Plan

There is no collection behind this enumerator,
just a theoretical matrix traversed in an unusual way.

$A : t:\text{enor}(\mathbb{N} \times \mathbb{N}), e:\mathbb{R}, i:\mathbb{N}, j:\mathbb{N}$

$Pre : t = t_0 \wedge e = e_0 \wedge \exists i, j \in \mathbb{N}: f(i) + g(j) = e$

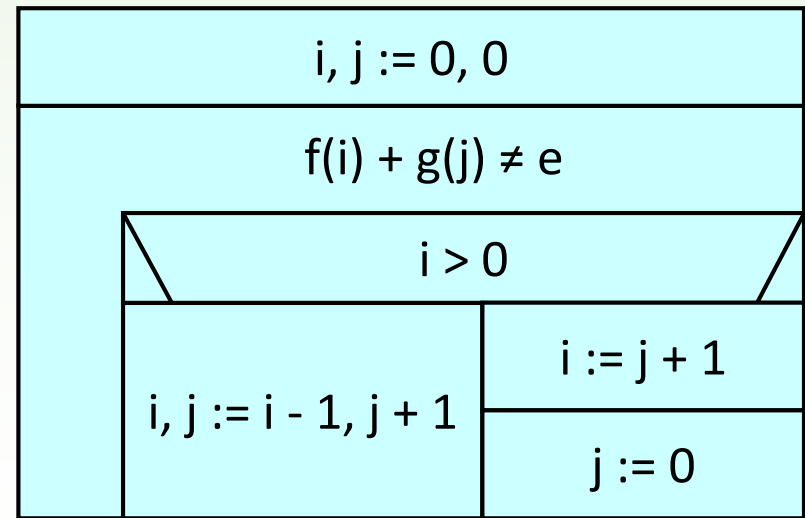
$Post : e = e_0 \wedge (i, j) = \text{SELECT}_{(i, j) \in t_0} (f(i) + g(j) = e)$

Selection:

$t:\text{enor}(E) \sim t:\text{enor}(\mathbb{N} \times \mathbb{N})$

$e \sim (i, j)$

$\text{cond}(e) \sim f(i) + g(j) = e$



Testing the selection

□ Testing of selection means testing the enumerator.

- **Length-based**: the enumerator is infinite, but it can be tested that the searched item is the first, second, or umpteenth item. Or the searched item is in the first, second, or umpteenth diagonal.
- It can be considered that the searched item is **at the beginning, in the middle, or at the end of a diagonal**.
- **Beginning of the enumeration**: $f(0)+g(0)=e$

Program

```
int main()
{
    bool all(double r) { return true; }

    double e = read<double>("Give a real number: ",
                           "This is not a real number!", all);

    int i, j;
    i = j = 0;
    while( f(i)+g(j) != e ){
        if(i>0) { --i; ++j; }
        else    { i = j+1; j = 0; }
    }

    cout << "The given number is equal to the sum f("
         << i << ")+g(" << j << ")\n";

    return 0;
}
```

function template

Template for reading

```
template <typename Item>
Item read( const std::string &msg, const std::string &err, bool valid(Item))
{
    Item n;
    bool wrong;
    do{
        std::cout << msg;
        std::cin >> n;
        if((wrong = std::cin.fail())) std::cin.clear();
        std::string tmp = "";
        getline(std::cin, tmp);
        wrong = wrong || tmp.size() != 0 || !valid(n);
        if(wrong) std::cout << err << std::endl;
    }while(wrong);
    return n;
}
```

operator>> has to be defined
for type Item

2nd task

On a trip, somebody's smartwatch measured the altitude in every minute, and stored it in a sequential input file. What percent of the trip went uphill?

$A : f:\text{infile}(\mathbb{R}), v:\mathbb{R}$

$Pre : f = f_0 \wedge |f_0| \geq 2$

$Post : v = \left(\sum_{\substack{i=2 \dots |f_0| \\ f_0[i] > f_0[i-1]}} 1 \right) / \left(\sum_{i=2 \dots |f_0|} 1 \right)$

Two successive items are referred, but reading of a sequential input file does not support it.

$A : t:\text{enor}(\mathbb{R} \times \mathbb{R}), v:\mathbb{R}$

$Pre : t = t_0 \wedge |t_0| > 0$

$Post : v = \left(\sum_{\substack{(first, second) \in t_0 \\ first < second}} 1 \right) / \left(\sum_{(first, second) \in t_0} 1 \right)$

Instead of a sequential input file, a custom enumerator would be useful that enumerates the successive pairs of the file.

Buffered enumerator

The enumerator pretends to read the next two successive items of the file.

enor($\mathbb{R} \times \mathbb{R}$)

A sequence of successive pairs from the original file.

$(\mathbb{R} \times \mathbb{R})^*$	first()	next()	current()	end()
f: infile(\mathbb{R}) first, second : \mathbb{R} st : Status	st, first, f : read st, second, f: read	first:= second st, second, f : read	(first, second)	st =abnorm

Plan

$A : t: \text{enor}(\mathbb{R} \times \mathbb{R}), v: \mathbb{R}$

$Pre : t = t_0 \wedge |t_0| > 0$

$Post : v = (100 \cdot \sum_{\substack{(first, second) \in t_0 \\ first < second}} 1) / (\sum_{(first, second) \in t_0} 1)$

Counting and summation
are in the same loop.

Counting:

$t: \text{enor}(E) \sim t: \text{enor}(\mathbb{R} \times \mathbb{R})$
 $e \sim (first, second)$
 $\text{cond}(e) \sim second > first$

Summation:

$t: \text{enor}(E) \sim t: \text{enor}(\mathbb{R} \times \mathbb{R})$
 $e \sim (first, second)$
 $f(e) \sim 1$
 $H, +, 0 \sim \mathbb{N}, +, 0$

st, first, f : read
st, second, f : read

c, d := 0, 0

st = norm

first < second

c := c + 1

—

d := d + 1

first := second
st, second, f : read

v := 100 · c / d

Testing counting and summation

- ❑ Counting and summation use the same enumerator. These cases do not have to be tested twice:
 - length-based: one, two, or more (always uphill)
 - beginning of the enumeration: uphill just at the beginning
 - end of the enumeration : uphill just at the end
- ❑ Result-based (counting):
 - there is no uphill
 - there is only one minute of uphill
 - there are more uphills
- ❑ Testing the memory loading at the summation is not necessary.

Program

```
int main()
{
    ifstream f("input.txt");
    if(f.fail()){
        cout << "Wrong file name!\n";
        exit(1);
    }

    int first, second;
    int c = 0; int d = 0;
    for( f >> first >> second; !f.fail(); first = second, f >> second){
        if( first < second ) ++c;
        ++d;
    }

    cout.setf(ios::fixed);
    cout.precision(2);
    cout << "Rate of the uphill part of the trip: "
        << (100.0*c)/d << "%" << endl;

    return 0;
}
```

Diagram illustrating the code structure and annotations:

- An arrow points from the `for` loop header in the main code to a callout box containing the loop body logic:

```
f >> first >> second;
while( !f.fail() )
    if( first < second ) ++c;
    ++d;
    first = second;
    f >> second;
}
```

- An arrow points from the expression `(100.0*c)/d` in the `cout` statement to a callout box containing the text:

real division, since the dividend is double

3rd task

On a trip, somebody's smartwatch measured the altitude in every minute, and stored it in a sequential input file. How long was the longest consecutive uphill?

A : $f:\text{infile}(\mathbb{R}), \text{max}:\mathbb{N}$

Pre : $f = f_0 \wedge$

$\exists i \in [1 .. |f|]: f[i] > f[i-1]$

Post : ?

It should be a maximum search in the length of the consecutive uphill, but they cannot be read from the file. It is difficult to specify the task precisely.

Idea



- ❑ It would be nice to enumerate the length of the consecutive uphill. Then it would be easy to find the maximum.
- ❑ For that, it should be seen in which minute slopes the road upwards and downwards. Based on the original file, it is an easy task.

t:

2

1

3

2

b:

0

1

1

0

1

0

1

1

1

0

0

1

1

f:

450 432 444 448 437 445 439 448 456 463 458 450 457 464

Plan

Suppose to have an enumerator that enumerates the length of the consecutive uphill.

$A : t:\text{enor}(\mathbb{N}), \text{max}:\mathbb{N}$

$Pre : t = t_0 \wedge |t_0| > 0$

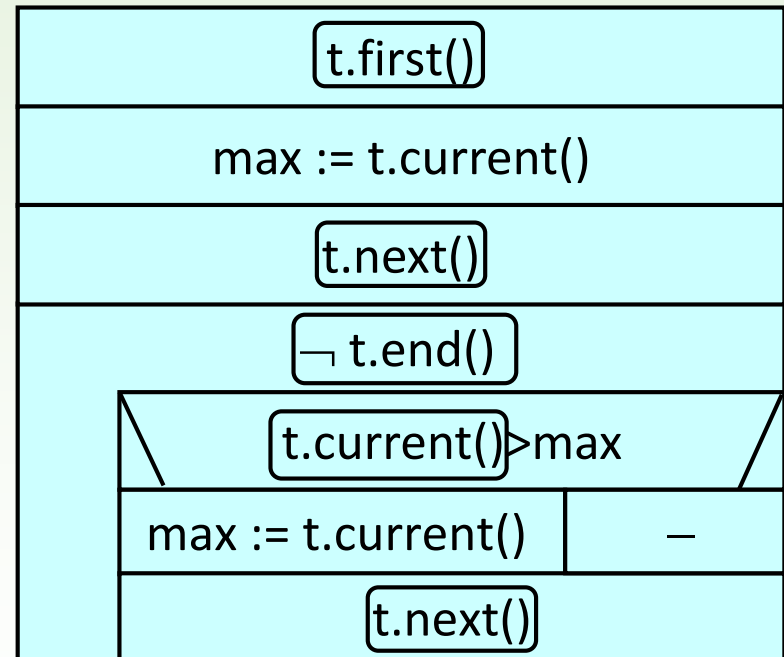
$Post : \text{max} = \mathbf{MAX}_{e \in t_0} e$

Maximum search:

$t:\text{enor}(E) \sim t:\text{enor}(\mathbb{N})$

$f(e) \sim e$

$H, > \sim \mathbb{N}, >$



Testing the maximum search

❑ Enumerator

- length-based: one, two, or more
- beginning of the enumerator: first is the maximum
- end of the enumerator: last is the maximum

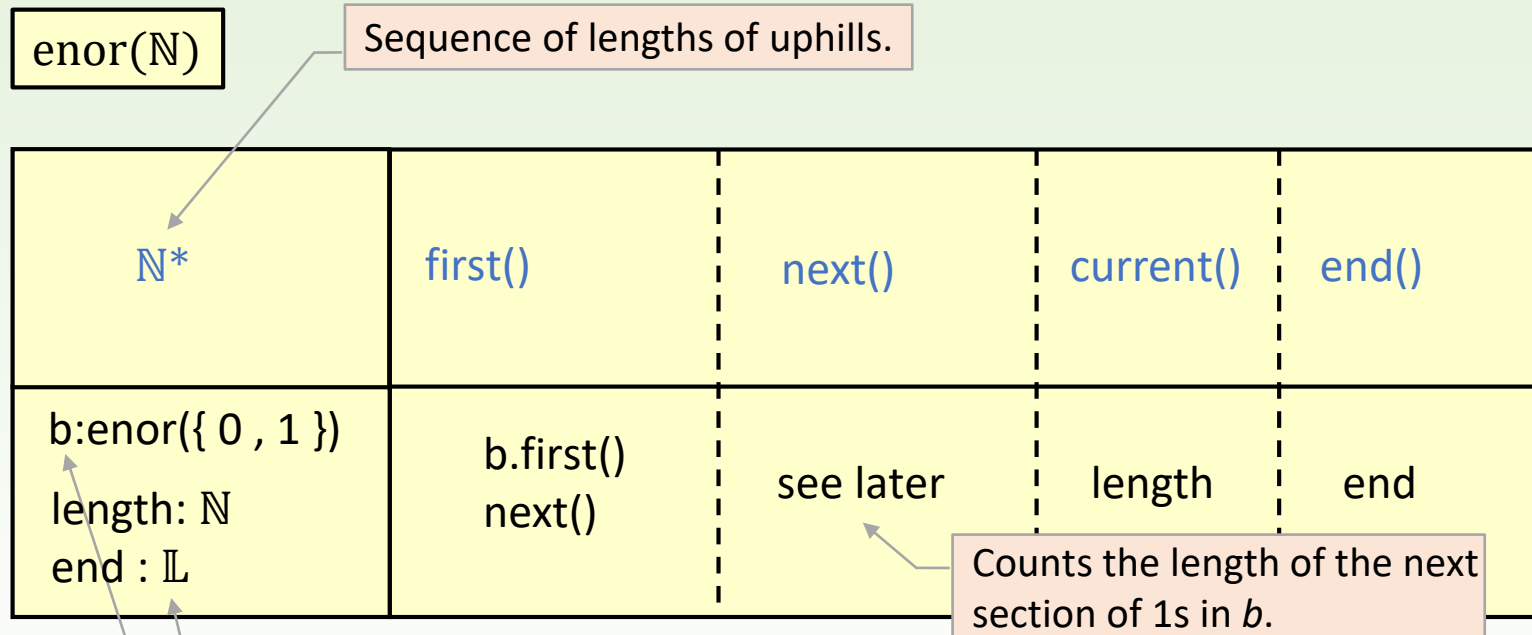
❑ Result-based:

- One maximum
- More maxima

Main program

```
int main()
{
    LengthEnumerator t("input.txt");
    t.first();
    int max = t.current();
    for( t.next(); !t.end(); t.next() ){
        if( max < t.current() ) max = t.current();
    }
    cout << "The length of the longest uphill: " << max << endl;
    return 0;
}
```

Abstract enumerator of lengths



current() and end() give the value of variables length and end, respectively, the value of which is calculated by methods first() and next().

Suppose to have an enumerator that indicates the changing of the slope with enumeration of 0s and 1s.

Operation next()

Counts the length of the next section of 1s, if there is any.

$A : b:\text{enor}(\{0, 1\}), \text{length}:\mathbb{N}, \text{end}:\mathbb{L}$

$Pre : b = b' \wedge b' \text{ is „in process” } (\neg b.\text{end}())$

$Post : (e'', b'') = \text{SELECT}_{e \in (b'.\text{current}(), b')} (b'.\text{end}() \vee e=1) \wedge$

$\wedge \text{end} = b''.\text{end}()$

$\wedge (\neg \text{end} \rightarrow (\text{length}, b = \sum_{e \in (e'', b'')}^{e=1} 1))$

- b' – initial state of b
- b'' – state of b after jumping over the 0s
- b – final state of the enumerator

Looks for the beginning of the next section of 1s or the end of the enumerator in b' .

This notation (instead of $e \in b'$) means that $b'.\text{current}()$ is accessible without $b'.\text{first}()$ since b' is already “in process”.

Length of the next section of 1s is given by a summation which holds as long as $e = 1$ (here $e = b.\text{current}()$). It can stop before reaching the end of b . For that, the previously processed enumeration is used, where $e'' = b''.\text{current}()$.

If there is another section of 1s, then $e'' = b''.\text{current}() = 1$ and $\neg b''.\text{end}()$.

Notation for specification

sx, dx, x : read enumerates (sx, dx) pairs, but instead of $(sx, dx) \in x_0$, $dx \in x_0$ is easier.

Enumeration until the end:

t:enor(E) Σ, MAX	h:set(E)	i:[m .. n]	x:infile(E)
$r = \boxtimes_{e \in t_0} f(e)$	$r = \boxtimes_{e \in h_0} f(e)$	$r = \boxtimes_{i=m..n} f(i)$	$r = \boxtimes_{dx \in x_0} f(dx)$
$r, t = \boxtimes_{e \in t_0} \text{cond}(e)$	$r, h = \boxtimes_{e \in h_0} \text{cond}(e)$	$r, i = \boxtimes_{i=m..n} \text{cond}(i)$	$r, (sx, dx, x) = \boxtimes_{dx \in x_0} \text{cond}(dx)$

SEARCH, SELECT

sx, dx, x (like h or i) indicates the actual and the final state of the enumerator.

Enumeration as long as a condition holds:

t:enor(E)	h:set(E)	i:[m .. n]	x:infile(E)
$\dots = \boxtimes_{e \in t_0}^{\text{cond}(e)} f(e)$	$\dots = \boxtimes_{e \in h_0}^{\text{cond}(e)} f(e)$	$\dots = \boxtimes_{i=m..n}^{\text{cond}(i)} f(i)$	$\dots = \boxtimes_{dx \in x_0}^{\text{cond}(dx)} f(dx)$
$\neg t.\text{end}() \wedge \text{cond}(e)$	$h \neq \emptyset \wedge \text{cond}(e)$	$i \leq n \wedge \text{cond}(i)$	$sx = \text{norm} \wedge \text{cond}(e)$

Continue an enumerator previously stopped:

t:enor(E)	h:set(E)	i:[m .. n]	x:infile(E)
$\dots = \boxtimes_{e \in (t'.\text{current}(), t')} f(e)$	$\dots = \boxtimes_{e \in h'} f(e)$	$\dots = \boxtimes_{i=i'+1..n} f(i)$	$\dots = \boxtimes_{dx \in (dx', x')} f(dx)$

Operation next()

$$e'', b'' = \text{SELECT}_{e \in (b'.\text{current}(), b')} (b.\text{end}() \vee e=1) \wedge \text{end} = b''.\text{end}()$$

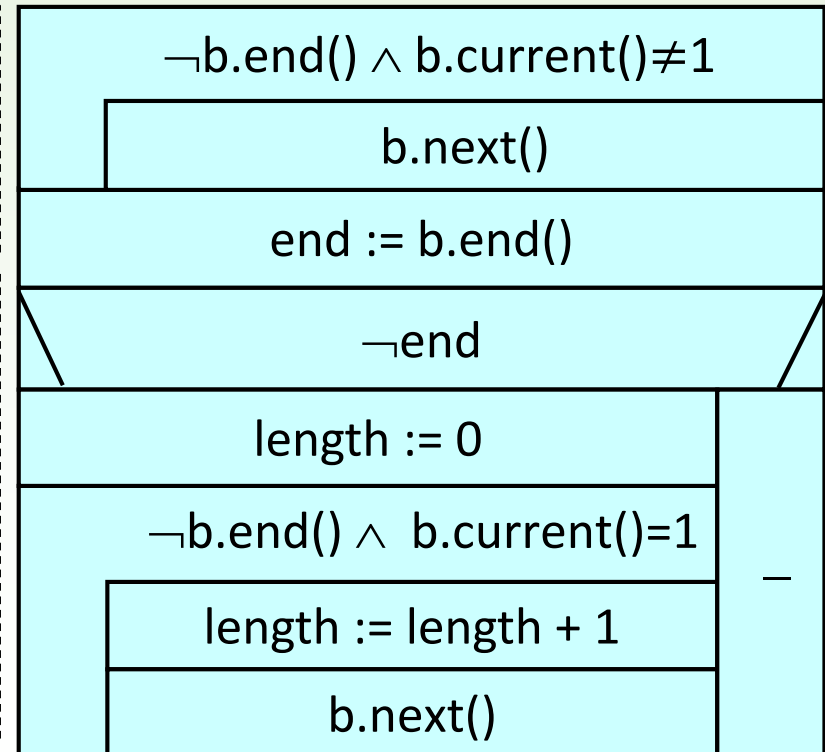
$$\wedge (\neg \text{end} \rightarrow (\text{length}, b = \sum_{e \in (e'', b'')} 1))$$

Selection with enumerator "in process"

t:enor(E) ~ b:enor(\mathbb{L})
without first()
cond(e) ~ b.end() \vee b.current()=1

Summation as long as a condition holds with enumerator "in process"

t:enor(E) ~ b:enor(\mathbb{L})
without first()
while b.current()=1
s ~ length
f(e) ~ 1
H,+,0 ~ $\mathbb{N},+,0$



Grey box testing of next()

❑ Selection with enumerator “in process”:

- length-based: zero, one, or more slope without uphill
- beginning of the enumerator: uphill right at the beginning
- end of the enumerator: uphill just at the end
- Condition of selection: there is or there is no uphill

❑ Summation as long as a condition holds with enumerator “in process”:

- length-based: zero, one, or longer uphill at the beginning
- beginning of the enumerator: uphill just at the beginning
- end of the enumerator: uphill just at the end
- Loading: not necessary

Enumerator class of lengths


```
class LengthEnumerator{  
public:  
    LengthEnumerator(const std::string &fname) : _b(fname){}  
    void first() { _b.first(); next(); }  
    int current() const { return _length; }  
    bool end() const { return _end; }  
    void next();  
private:  
    BitEnumerator _b;  
    int _length;  
    bool _end;  
};
```

```
void LengthEnumerator::next()  
{  
    for( ; !_b.end() && !_b.current(); _b.next() );  
    if ( (_end = _b.end()) ) return;  
    for( _length = 0 ; !_b.end() && _b.current(); _b.next() ) ++_length;  
}
```

Abstract enumerator of slope changing

enor({ 0 , 1 })

Sequence of 0s and 1s the length of which is size of the original file – 1 (processes two successive elements in the file). If the slope goes upwards, the actual item of the enumerator is 1 (0 otherwise).

<div>$\{ 0 , 1 \}^*$</div>	<div>first()</div>	<div>next()</div>	<div>current()</div>	<div>end()</div>				
<div>f: infile(\mathbb{R}) first, second:\mathbb{R} st: Status</div>	<div>st, first, f: read st, second, f: read</div>	<div>first:= second st, second, f : read</div>	<div><table><tr><td colspan="2"><div>\backslashfirst < second$/$</div></td></tr><tr><td>1</td><td>0</td></tr></table></div>	<div>\backslashfirst < second$/$</div>		1	0	<div>st = abnorm</div>
<div>\backslashfirst < second$/$</div>								
1	0							

Enumerator class of slope changing

```
class BitEnumerator{
public:
    enum Errors { FILEERROR };
    BitEnumerator(const std::string &fname){
        _f.open(fname);
        if(!_f.fail()) throw FILEERROR;
    }
    void first() { _f >> _first >> _second; }
    void next() { _first = _second; _f >> _second; }
    int current() const { return (_first < _second ? 1 : 0); }
    bool end() const { return _f.fail(); }
private:
    std::ifstream _f;
    int _first, _second;
};
```