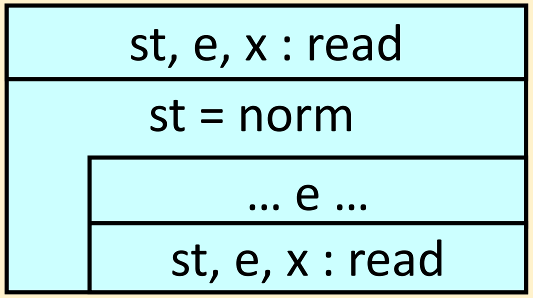
# Enumeration of a sequential input file

# Enumeration of a sequential input file

- Items of a sequential input file x:infile(E) (can be considered as a sequence) can be enumerated via operation st,e,x:read (e:E, st:Status={abnorm, norm} ).
- Operations of the enumeration:

| | | |
|---|---|---|
| first() | ~ | st, e, x : read |
| next() | ~ | st, e, x : read |
| current() | ~ | e |
| end() | ~ | st=abnorm |



```
st, e, x : read
st = norm
    … e …
    st, e, x : read
```

- Enumeration is based on pre-reading strategy: first reading, then examining if the reading was successful and if it was, then processing the item.
- In the specification, the enumeration might be denoted by e∊x.

# Processing files

- In practice, there are a lot of problems where sequences have to be generated (from sequences). If sequences are e.g. in files (or are reachable in console), then it is worthy to handle them as sequential input and output files.
- Most common tasks:
  - copy and elementwise process (e.g. creating a report)
  - multiple item selection
  - partitioning
  - union of sorted sequences
- The common is that all of them are based on summation, and except the union (which needs a custom enumerator), all of them use a sequential input file enumerator.

# Summation in file processing

**General summation**

*A* : t:enor(E), s:H

*Pre* : $t = t_0$

*Post*: $s = \Sigma_{e \in t_0} f(e)$

$f : E \rightarrow H$
$+ : H \times H \rightarrow H$
$0 \in H$

**Special summation for files**

*A* : x:infile(E), y:outfile(F)

*Pre* : $x = x_0$

*Post*: $y = \bigoplus_{e \in x_0} f(e)$

$f : E \rightarrow F^*$
$\oplus : F^* \times F^* \rightarrow F^*$
$<> \in F^*$

Summation:

| | | |
|---|---|---|
| t:enor(E) | ~ | x:infile(E) |
| | | st,e,x : read |
| H,+,0 | ~ | F*, ⊕, <> |

```
s := 0
t.first()
¬t.end()
    s := s+f(t.current())
    t.next()
```

```
y := <>
st,e,x : read
st=norm
    y : write(f(e))
    st,e,x : read
```

$y := y \oplus f(e)$

## 1st task

Transform a text: change every accented letter to unaccented in a sequential input file!

*A* : x:infile(Char) , y:outfile(Char)

*Pre* : $x = x_0$

*Post*: $y = \bigoplus_{ch \in x_0} \langle transform(ch) \rangle$

where transform : Char $\to$ Char and transform(ch) = …

<u>Summation</u>:

| | | |
|---|---|---|
| t:enor(E) | ~ | x:infile(Char) |
| | | st,ch,x : read |
| e | ~ | ch |
| f(e) | ~ | <transform(ch)> |
| H,+,0 | ~ | Char*, $\oplus$, <> |

```
y := < >
st, ch, x : read
st = norm
    y : write(transform(ch))
    st, ch, x : read
```

# Grey box texting

- In case of summation we have to check
  - the enumerator (like in other patterns)
    - length: 0, 1, 2, and more items in the enumerator
    - "sides" of the enumerator: if there are at least 2 different items in the enumerator, then it is checkable
  - the loading, but the size of the output file equals to the size of the input file. It is not necessary.
- The conversion has to be verified, too.

length-based: input of 0, 1, 2, and more characters (copy)

| | | | |
|---|---|---|---|
| conversion-based: | x = <áéíöőúüű> | → | y = <aeioouuu> |
| | x = <aeioouuu> | → | y = <aeioouuu> |
| | x = <bsmnz> | → | y = <bsmnz> |
| | x = <Ferenc Puskás …> | | |

## C++

- Language C++ uses pre-reading strategy for processing a file.
- Implementations of reading a character (st, ch, x : read):
    - x >> ch
        - Does not read white spaces except if this automatism is switched off (x.unsetf(ios::skipws)).
    - x.get(ch)
        - Reads every character, even white spaces, too.
- In C++, operation st==norm is implemented as !x.eof(). Many times, using !x.fail() is more secure, because it indictes not just the end of file, but every type of unsuccessful reading, like the file is not correctly filled up.

# C++ program

```
int main()
{
    ifstream x( "input.txt" );
    if ( x.fail() ){ … }
    ofstream y( "output.txt" );
    if ( y.fail() ){ … }

    char ch;
    while(x.get(ch)){
        y << transform(ch);
    }

    return 0;
}
```

st, ch, x : read

st==norm

```
x.get(ch);
while(!x.fail()){
    y << transform(ch);
    x.get(ch));
}
```

y : write(transform(ch))

## C++ program

```
char transform(char ch)
{
    char new_ch;
    switch (ch) {
        case 'á' :                                  new_ch = 'a'; break;
        case 'é' :                                  new_ch = 'e'; break;
        case 'í' :                                  new_ch = 'i'; break;
        case 'ó' : case 'ö' : case 'ő' :            new_ch = 'o'; break;
        case 'ú' : case 'ü' : case 'ű' :            new_ch = 'u'; break;
        case 'Á' :                                  new_ch = 'A'; break;
        case 'É' :                                  new_ch = 'E'; break;
        case 'Í' :                                  new_ch = 'I'; break;
        case 'Ó' : case 'Ö' : case 'Ő' :            new_ch = 'O'; break;
        case 'Ú' : case 'Ü' : case 'Ű' :            new_ch = 'U'; break;
        default  :                                  new_ch = ch;
    }
    return new_ch;
}
```

## 2nd task

Assort the even numbers from a sequential input file containing integers.

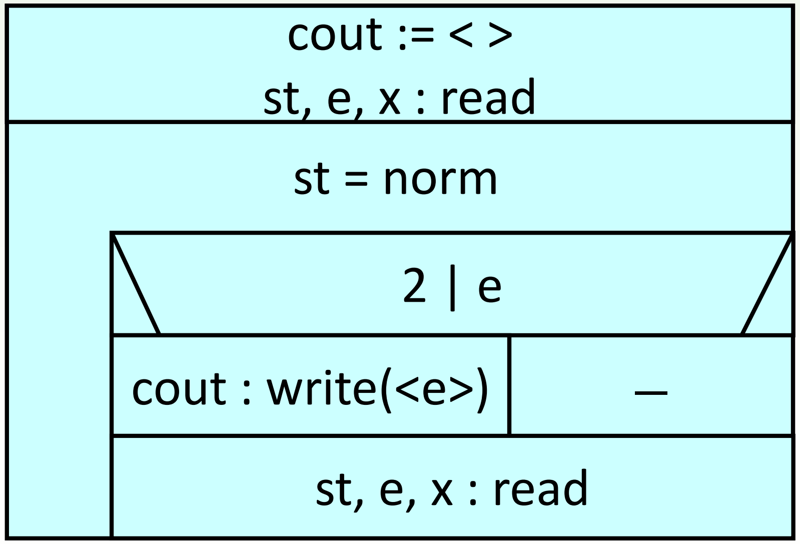
*A* : x:infile($\mathbb{Z}$) , cout:outfile($\mathbb{Z}$)

*Pre* : $x = x_0$

*Post*: $cout = \bigoplus_{\substack{e \in x_0 \\ 2 \mid e}} \langle e \rangle$

<u>Conditional summation:</u>

| | | |
|---|---|---|
| t:enor(E) | ~ | x:infile($\mathbb{Z}$) |
| | | st,e,x : read |
| f(e) | ~ | <e> |
| cond(e) | ~ | 2\|e |
| y | ~ | cout |
| H,+,0 | ~ | $\mathbb{Z}^*$, ⊕, <> |



```
cout := < >
st, e, x : read
st = norm
    2 | e
        cout : write(<e>)
    else
        –
    st, e, x : read
```

## Grey box testing

- We have to check
  - the enumerator
    - length: 0, 1, 2, and more items
    - “sides” of the enumerator: at least 2 different elements
  - loading is not necessary
  - condition of the assortment

| | |
|---|---|
| length-based: | input of 0, 1, 2, and more even numbers (copy) |
| condition-based: | x = <-100, -55, -2, -1, 0, 1, 2, 55, 100> |
| | → cout = <-100, -2, 0, 2, 100 > |

# C++ program

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ifstream x;
    bool error = true;
    do{
        string fname;
        cout << "file name: ";
        cin >> fname;
        x.open(fname.c_str());
        if( (error=x.fail()) ) {
            cout << "Wrong file name!\n";
            x.clear();
        }
    }while(error);

    cout << "Selected even numbers: ";
    int e;
    while(x >> e) {
        if(0==e%2) cout << e << " ";
    }
    return 0;
}
```

After skipping the white spaces, it reads the data of type of *e*.

st, e, x : read

st==norm

```
x >> e;
while(!x.fail()){
    if(0==e%2) cout << e;
    x >> e;
}
```

cout : write(<e>)

## 3rd Task

From the registry of a library, assort books of count 0 and those that were published before 2000.

*A* : x:infile(Book) , y:outfile(Book2) , z:outfile(Book2)

Book = rec( ID : $\mathbb{N}$ , author: String, title : String, publisher : String, year : String, count : $\mathbb{N}$, isbn : String)

Book2 = rec( ID : $\mathbb{N}$ , author : String, title : String)

*Pre* : $x = x_0$

*Post* : $y = \bigoplus_{\substack{dx \in x_0 \\ dx.count=0}} <(dx.ID, dx.author, dx.title)> \wedge$

$z = \bigoplus_{\substack{dx \in x_0 \\ dx.year < "2000"}} <(dx.ID, dx.author, dx.title)>$

# Algorithm

<u>Conditional summation</u>:

| | |
|---|---|
| t:enor(E) | ~ x:infile(Book), sx,dx,x : read |
| e | ~ dx |
| $f_1$(e) | ~ <(dx.ID, dx.author, dx.title)>, $cond_1$(i) ~ dx.count = 0 |
| $f_2$(e) | ~ <(dx.ID, dx.author, dx.title)>, $cond_2$(i) ~ dx.year < ”2000” |
| H,+,0 | ~ Book2*, ⊕, <> |

```
y, z := < >
sx, dx, x : read
sx = norm
    dx.count = 0
        y : write(<(dx.ID, dx.author, dx.title)>)  |  —
    dx.year < ”2000”
        z : write(<(dx.ID, dx.author, dx.title)>)  |  —
    sx, dx, x : read
```

# Grey box testing

- We have to check
  - the enumerator
    - length: 0, 1, 2, and more items
    - “sides” of the enumerator: at least 2 different elements
  - loading is not necessary
  - conditions of the assortment

| | |
|---|---|
| <u>length</u>-based: | 0, 1, 2, and more books that satisfy the condition (copy) |
| <u>condition</u>-based: | books of count zero and non zero and |
| | books published before 2000 and after 1999 |

# Implementation with operations read and write

```
struct Book{
    int id;
    string author;
    string title;
    string publisher;
    string year;
    int count;
    string isbn;
};

enum Status{abnorm, norm};
```

```
bool read(Status &sx, Book &dx, ifstream &x);
void write(ofstream &x, const Book &dx);

int main()
{
    ifstream x("inp.txt");
    if (x.fail() ) { … }
    ofstream y("out1.txt");
    if (y.fail() ) { … }
    ofstream z("out2.txt");
    if (z.fail() ) { … }

    Book dx;
    Status sx;
    while(read(sx,dx,x)) {
        if (0==dx.count)    write(y,dx);
        if (dx.year<"2000") write(z,dx);
    }
    return 0;
}
```

```
read(sx,dx,x);
while(norm==sx){
    if (0==dx.count)    write(y,dx);
    if (dx.year<"2000") write(z,dx);
    read(sx,dx,x);
}
```

# Operations *read* and *write*

made up of lines,
strictly positioned input file

```
12 J. K. Rowling   Harry Potter II.     Animus         2000   0 963 8386 94 O
15 A. A. Milne     Winnie the Pooh      Móra           1936  10 963 11 1547 X
```

```
    bool read(Status &sx, Book &dx, ifstream &x){
        string line;
        getline(x,line);
        if (!x.fail()) {
            sx = norm;
            dx.id        = atoi(line.substr( 0, 4).c_str());
            dx.author    =      line.substr( 5,14);
            dx.title     =      line.substr(21,19);
            dx.publisher =      line.substr(42,14);
            dx.year      =      line.substr(58, 4);
            dx.count     = atoi(line.substr(63, 3).c_str());
            dx.isbn      =      line.substr(67,14);
        }
        else sx=abnorm;

        return norm==sx;
    }
```

reads a line

transforms a character chain to integer

substring

creates a C-style string

returns a logical value

```
    void write(ofstream &x, const Book &dx){
        x << setw(4)  << dx.id     << ' '
          << setw(14) << dx.author << ' '
          << setw(19) << dx.title  << endl;
    }
```

positioned writing
#include <iomanip>

# Implementation with classes

```
struct Book{
      int id;
      std::string author;
      std:: string title;
      std:: string publisher;
      std:: string year;
      int count;
      std:: string isbn;
};
```

```
f.open(fname.c_str());
if(f.fail()) throw FILE_ERROR;
```

```
int main()
{
    try{
        Stock  x("input.txt");
        Result y("output1.txt");
        Result z("output2.txt");

        Book dx;
        Status sx;
        while(x.read(dx,sx)){
            if (0==dx.count)    y.write(dx);
            if (dx.year<"2000") z.write(dx);
        }
    }catch(Stock::Errors e){
        if(Stock::FILE_ERROR==e)  cout << …
    }catch(Result::Errors e){
        if(Result::FILE_ERROR==e) cout << …
    }
    return 0;
}
```

```
enum Status{abnorm, norm};
class Stock{
public:
    enum Errors{FILE_ERROR};
    Stock(std::string fname);
    bool read(Book &dx, Status &sx);
private:
    std::ifstream x;
};
```

its body is unchanged

```
class Result{
public:
    enum Errors{FILE_ERROR};
    Result(std::string fname);
    void write(const Book &dx);
private:
    std::ofstream x;
};
```

its body is unchanged

```
f.open(fname.c_str());
if(f.fail()) throw FILE_ERROR;
```

## 4th Task

In a textfile, results of tests of students are stored. Results of one student are in one line. In one line, divided by whitespaces or tabs, data is given in the following order:

- neptun-code (6 characters),
- sequence of characters "+" and "-" without white space (non empty string)
- Results of one assignment and 4 tests (all of them between 0 and 5)

Give the final mark of those students who do not fail the course!

```
AA11XX   ++++-+++++ 5 5 5 5 5
CC33ZZ   ++++--++-- 2 1 0 5 5
BB22YY   --+---+++- 2 2 3 3 5
```

# Plan of the solution

*A* : x : infile(Student) , y : outfile(Evaluation)
Student = rec(neptun : String, pm : String, marks : $\{0..5\}^5$)
Evaluation = rec(neptun : String, mark : {2..5})

*Pre* : $x = x_0$

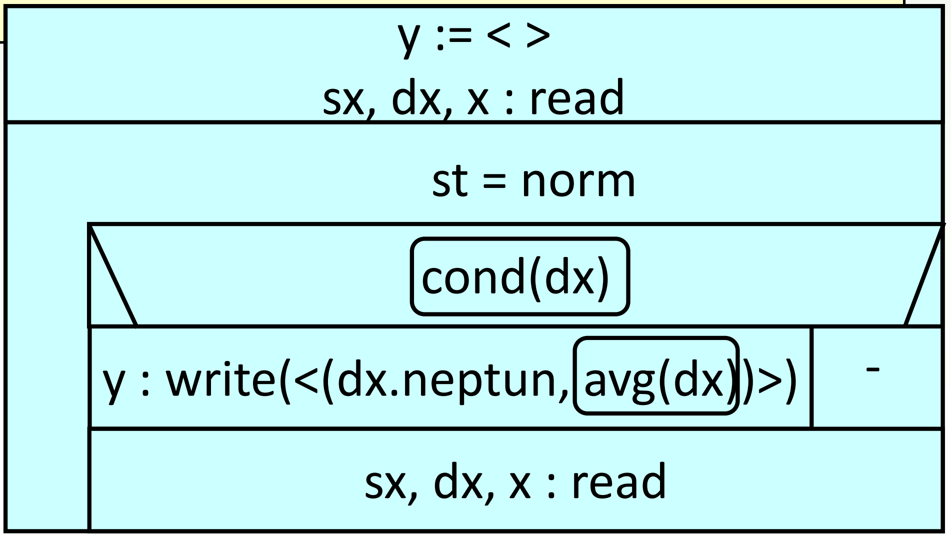
*Post* : $y = \bigoplus_{\substack{dx \in x_0 \\ cond(dx)}} \langle dx.neptun,\ avg(dx) \rangle$

$$cond(dx) = \forall\mathbf{SEARCH}_{i=1}^{5}\ (dx.marks[i] > 1)\ \wedge\ \Big( \sum_{\substack{i=1 \\ dx.pm[i]='-'}}^{|dx.pm|} 1 \ \le \sum_{\substack{i=1 \\ dx.pm[i]='+'}}^{|dx.pm|} 1 \Big)$$

$$avg(dx) = \Big( \sum_{i=1}^{5} dx.marks[i] \Big) / 5$$

<u>Conditional summation:</u>

| | | |
|---|---|---|
| t:enor(E) | ~ | x:infile(Student) |
| | | sx,dx,x : read |
| e | ~ | dx |
| f(e) | ~ | <(dx.neptun, avg(dx))> |
| H,+,0 | ~ | Evaluation*, ⊕, <> |



```
y := < >
sx, dx, x : read
st = norm
    cond(dx)
        y : write(<(dx.neptun, avg(dx))>)  |  -
    sx, dx, x : read
```

# Subprograms

$$l := \left( \overset{5}{\underset{i=1}{\forall\mathbf{SEARCH}}}\ dx.marks[i]>1 \right)$$

**Opt. linear search:**

| | | |
|---|---|---|
| t:enor(E) | ~ | i = 1 .. 5 |
| e | ~ | i |
| cond(e) | ~ | dx.marks[i]>1 |

$$p, m := \sum_{\substack{i=1\\ dx.pm[i]='+'}}^{|dx.pm|} 1\ ,\ \sum_{\substack{i=1\\ dx.pm[i]='-'}}^{|dx.pm|} 1$$

**Two countings:**

| | | |
|---|---|---|
| t:enor(E) | ~ | i = 1 .. \|dx.pm\| |
| e | ~ | i |
| cond1(e) | ~ | dx.pm[i] = ’+’ |
| cond2(e) | ~ | dx.pm[i] = ’-’ |

$$s := \left( \sum_{i=1}^{5} dx.marks[i] \right) / 5$$

**Summation:**

| | | |
|---|---|---|
| t:enor(E) | ~ | i = 1 .. 5 |
| e | ~ | i |
| f(e) | ~ | dx.marks[i] |
| H,+,0 | ~ | ℕ, +, 0 |

```
l := cond(dx)

l, i := true, 1
l ∧ i ≤ 5
    l := dx.marks[i] > 1
    i := i + 1
p, m := 0, 0
i = 1 .. |dx.pm|
    dx.pm[i] = ’+’
        p := p + 1      | —
    dx.pm[i] = ’-’
        m := m + 1      | —
l := l ∧ p ≥ m
```

```
a := avg(dx.marks)

s := 0
i = 1 .. 5
    s := s + dx.marks[i]
a := s / 5
```

# Grey box testing

Outer conditional summation:

| | |
|---|---|
| length-based: | 0, 1, 2, and more students who pass the course |
| “sides” of the enumerator: | done by the above |
| loading: | not needed |
| cond() and f() : | see below |

Counting pluses and minuses:

| | |
|---|---|
| length-based: | 0, 1, 2, and more, only ’+’ |
| “sides” of the enumerator: | enumerations of 2 items, with ’+’ and ’–’ (4 cases) |
| result-based: | 0, 1, and more ’–’ with some ’+’es |

There is no failed test (optimistic linsearch) :

| | |
|---|---|
| length-based: | not needed (length is 5) |
| “sides” of the enumerator: | only the first test is failed, only the last one is failed |
| result-based: | only 1s, there is 1, all of them at least 2 |

Sum of the marks:

| | |
|---|---|
| length-based: | not needed (length is 5) |
| “sides” of the enumerator: | different marks at the beginning and at the end |
| loading: | not needed |

# C++ program

```
bool cond(const vector<int> &marks, const string &pm );
double  avg(const vector<int> &marks);

int  main(){
    try{
        InpFile x("input.txt");
        OutFile y("output.txt");
        Student dx;
        Status sx;
        while(x.read(dx,sx)) {
            if (cond(dx.marks, dx.pm)) {
                Evaluation dy(dx.neptun, avg(dx.marks));
                y.write(dy);
            }
        }
    }catch( InpFile::Errors er ) {
        if( er==InpFile::FILE_ERROR ) cout << … ;
    }catch( OutFile::Errors er ) {
        if( er==OutFile::FILE_ERROR ) cout << … ;
    }
    return 0;
}
```

## C++ functions

```
bool cond(const vector<int> &marks, const string &pm ) {
     bool l = true;
     for(unsigned int i=0; l && i<marks.size(); ++i){
          l=marks[i]>1;
     }
     int p, m; p = m = 0;
     for(unsigned int i = 0; i<pm.size(); ++i){
          if(pm[i]=='+') ++p;
          if(pm[i]=='-') ++m;
     }
     return l && m<=p;
}
```

```
double avg(const vector<int> &marks) {
     double s = 0.0;
     for(unsigned int i = 0; i< marks.size(); ++i){
          s += marks[i];
     }
     return (0 == marks.size() ? 0 : s / marks.size());
}
```

# Sequential input file

```
bool InpFile::read(Student &dx, Status &sx)
{
    string line;
    getline(x, line);
    if (!x.fail() && line!="") {
        sx=norm;
        istringstream in(line);
        in >> dx.neptun;
        in >> dx.pm;
        dx.marks.clear();
        int mark;
        while( in >> mark )
            dx.marks.push_back(mark);
    } else sx=abnorm;
    return norm==sx;
}
```

```
struct Student {
    std::string neptun;
    std::string pm;
    std::vector<int> marks;
};
enum Status {abnorm, norm};

class InpFile{
public:
    enum Errors{FILE_ERROR};
    InpFile(std::string fname){
        x.open(fname.c_str());
        if(x.fail()) throw FILE_ERROR;
    }
    bool read( Student &dx, Status &sx);
private:
    std::ifstream x;
};
```

# Sequential output file

```
struct Evaluation {
    std::string neptun;
    double mark;
    Evaluation(std::string str, double j) : neptun(str), mark(j) {}
};

class OutFile{
public:
    enum Errors{FILE_ERROR};
    OutFile(std::string fname){
        x.open(fname.c_str());
        if(x.fail()) throw FILE_ERROR;
    }
    void write(const Evaluation &dx) {
        x.setf(std::ios::fixed);
        x.precision(2);
        x << dx.neptun << std::setw(7) << dx.mark << std::endl;
    }
private:
    std::ofstream x;
};
```

#include <iomanip>

## Task and program modification

In the textfile, lines begin with the name of the students which consists of optional number of (but at least one) parts (separators in between).

```
Muhammad Ali                           AA11XX ++++++++++ 5 5 5 5 5
Cher                                   CC33ZZ +++++-++++ 2 1 0 5 1
Cristiano Ronaldo dos Santos Aveiro    BB22YY ---+++---- 2 4 4 0 0
```

```
int  main(){
    try{
        InpFile x("input.txt");
        OutFile y("output.txt");
        Student dx;
        Status sx;
        while(x.read(dx,sx)) {
            if (dx.has) {
                Evaluation dy(dx.neptun, dx.result);
                y.write(dy);
            }
        }
    }
    …
}
```

```
struct Student {
    std::string name;
    std::string neptun;
    bool has;
    double result;
};
```

# Reading varying number of data

```
bool InpFile::read(Student &dx, Status &sx)
{
    string line, str;
    getline(f, line);
    if (!f.fail() && line!="") {
        sx=norm;

        istringstream in(line);
        in >> dx.name;
        in >> dx.neptun;
        in >> str;
        while( !('+'== str[0] || '-'== str[0]) ){
            dx.name += " " + dx.neptun;
            dx.neptun = str;
            in >> str;
        }
        vector<int> marks;
        int mark;
        while( in >> mark ) marks.push_back(mark);
        dx.has = cond(marks, str);
        dx.result = avg(marks);
    } else sx=abnorm;
    return norm==sx;
}
```

filling dx based on variable *line*

for reading data from the line (#include <sstream>)

If str does not start with + or –, then it is part of the name, or it is the neptun code

we thought it was a neptun code, but it is part of the name

str is considered to be a neptun code

private methods of class InpFile

# Reading from input files

| x : infile(E) | st, data, x : read | st = abnorm |
|---|---|---|
| E ≡ char<br>// characters without separation | x.get(data);<br>x >> data; //x.unsetf(ios::skipws) | x.eof()<br>x.fail() |
| E ≡ <basic type><br>// basic values divided by separators | x >> data; | x.fail() |
| E ≡ struct(s1 : <basic type>,<br>s2 : <basic type>,<br>sn : <basic type>$^n$,<br>… )<br>// record of fixed number of basic types<br>// divided by separators | x >> data.s1 >> data.s2;<br>for(int i=0; i<n; ++i) {<br>x >>data.sn[i];<br>} | x.fail() |
| E ≡ line<br>// line-buffered data<br>// number of data varies | string data;<br>getline(x, data);<br>istringstream is(data);<br>is >> … | x.fail() |