Wczytywanie z pliku - pojedyncze dane

Przy wczytywaniu danych z pliku istotna jest kontrola poprawności wykonania operacji. Często pojawiającym się błędem jest przyjęcie założenia, że jeśli nie został osiągnięty koniec pliku, to kolejna operacja na pewno zakończy się powodzeniem, na przykład:

```
while(!file.eof())
{
   file >> val;
  // tutaj wykorzystanie danych wczytanych do zmiennej val
  ...
}
```

Jeśli po ostatnio wczytanej liczbie następuje koniec pliku, to pokazany wyżej kod będzie działał poprawnie (nie będzie wczytywana kolejna liczba, ponieważ *file.eof()* zwróci wartość *true* i pętla *while* zakończy się). Jeśli jednak po wczytanej liczbie jest przejście do następnej linii (LF) i dopiero potem koniec pliku (brak kolejnych danych), to instrukcja:

```
file >> val;
```

zakończy się niepowodzeniem a w zmiennej *val* pozostanie dotychczasowa wartość. Brak informacji o poprawności wczytania doprowadzi do niekontrolowanego powielenia ostatnio wczytanej liczby, co z pewnością nie jest intencją programisty.

Poprawne podejście to kontrola **po** wykonaniu operacji wczytania ze strumienia plikowego. Ilustruje to poniższy fragment programu, w którym z pliku wczytywana jest pierwsza liczba reprezentująca ilość danych a następnie kolejne liczby (tutaj wpisywane do tablicy):

```
if(file.is_open())
{
  file >> size;

  if(file.fail())
    cout << "File error - READ SIZE" << endl;
  else
    for(int i = 0; i < size; i++)
    {
      file >> val;

      if(file.fail())
      {
        cout << "File error - READ DATA" << endl;
        break;
      }
      else
        tab[i] = val;
    }
    file.close();
}
else
  cout << "File error - OPEN" << endl;
```

Poniżej przedstawiono opis funkcji *fail()* w kontekście innych, używanych do sprawdzania operacji wykonywanych na strumieniu I/O. Warto zwrócić uwagę, że funkcja *fail()* nie jest negacją *good()*. Przy wczytaniu (poprawnym) ostatniej liczby, po której plik kończy się (bez przejścia do nowej linii) otrzymamy:

good() = false *eof() = true* *fail() = false* *bad() = false*

Opis ze strony: www.cplusplus.com http://www.cplusplus.com/reference/ios/ios/fail/

std::ios::fail

```
bool fail() const;
```

Check whether either failbit or badbit is set

Returns `true` if either (or both) the `failbit` or the `badbit` *error state flags* is set for the stream.

At least one of these flags is set when an error occurs during an input operation.

`failbit` is generally set by an operation when the error is related to the internal logic of the operation itself; further operations on the stream may be possible. While `badbit` is generally set when the error involves the loss of integrity of the stream, which is likely to persist even if a different operation is attempted on the stream. `badbit` can be checked independently by calling member function `bad`:

`iostate` value (member constants)	indicates	functions to check state flags				
		`good()`	`eof()`	`fail()`	`bad()`	`rdstate()`
goodbit	No errors (zero value `iostate`)	true	false	false	false	goodbit
eofbit	End-of-File reached on input operation	false	true	false	false	eofbit
failbit	Logical error on i/o operation	false	false	true	false	failbit
badbit	Read/writing error on i/o operation	false	false	true	true	badbit

`eofbit`, `failbit` and `badbit` are member constants with implementation-defined values that can be combined (as if with the bitwise OR operator).

`goodbit` is zero, indicating that none of the other bits is set.

Reaching the *End-of-File* sets the `eofbit`. But note that operations that reach the *End-of-File* may also set the `failbit` if this makes them fail (thus setting both `eofbit` and `failbit`).

Wczytywanie z pliku - grupy danych w osobnych liniach

Dane liczbowe w pliku mogą być niekiedy połączone w grupy, umieszczone w osobnych liniach. Mechanizm wczytywania danych powinien wtedy zapewnić kontrolę struktur danych, aby nie dopuścić do ich błędnej interpretacji (na przykład wtedy, gdy w jednej z grup będzie zbyt mało danych). Zilustrowane zostanie to przykładem wczytywania z pliku opisu grafu według następującego formatu:

- pierwsza linia pliku zawiera cztery liczby opisujące graf (liczba krawędzi, liczba wierzchołków, wierzchołek początkowy, wierzchołek końcowy),
- w kolejnych liniach podane są opisy krawędzi w postaci trzech liczb (wierzchołek początkowy, wierzchołek końcowy, waga krawędzi).

Poniżej przedstawiony jest wariant wczytywania z użyciem funkcji *getline()* i strumienia napisowego (*stringstream* - należy dołączyć plik nagłówkowy *<sstream>).* Wczytanie jednej linii z pliku do stringu *s* wygląda następująco:

```
getline(file, s);
```

Poprawność wykonania operacji można sprawdzić (podobnie jak przy wczytywaniu pojedynczej liczby) przy pomocy funkcji *file.fail()*, ewentualnie sprawdzając dodatkowo, czy wczytany string nie jest pusty:

```
if(file.fail() || s.empty())
  // wczytanie linii nie powiodło się
```

Następnie należy utworzyć obiekt typu strumień napisowy (w tym przypadku *istringstream* a więc strumień tylko do odczytu), inicjując go zawartością stringu wczytanego wcześniej przez funkcję getline():

```
istringstream in_ss(s);
```

Operacje pobierania danych ze strumienia wyglądają podobnie jak dla konsoli lub strumienia plikowego, na przykład:

```
in_ss >> val;
```

W analogiczny sposób można również kontrolować poprawność wczytywania danych:

```
if(in_ss.fail())
  // wczytanie danej nie powiodło się
```

Poniżej przedstawiony jest prawie kompletny przykład wczytywania opisu grafu z pliku według podanego wcześniej formatu danych. Pominięte zostały szczegóły związane z wykorzystaniem wczytanych danych (są one tylko zasygnalizowane przez zapis do zmiennych graph_ i edge_, deklaracje tych zmiennych są pominięte).

Ponieważ sprawdzanie poprawności wczytania ze strumienia napisowego po każdej danej jest dość kłopotliwe, zastosowano pomocniczą funkcję *file_read_line()*, która wczytuje z jednej linii strumienia plikowego (*file*) do tablicy (*tab*) określoną liczbę danych (*size*). Można rozszerzyć kontrolę poprawności tak, aby sygnalizowana była również nadmiarowa liczba danych w linii (w pokazanym przykładzie takie dane są po prostu ignorowane):

```
//-------------------------------------------------------------------------
bool file_read_line(ifstream &file, int tab[], int size)
{
string  s;

  getline(file, s);

  if(file.fail() || s.empty())
    return(false);

  istringstream in_ss(s);

  for(int i = 0; i < size; i++)
  {
    in_ss >> tab[i];
    if(in_ss.fail())
      return(false);
  }

  return(true);
}
```

```
//------------------------------------------------------------------------
void file_read_graph(string file_name)
{
ifstream    file;
int         tab[4];

  file.open(file_name.c_str());

  if(file.is_open())
  {
    if(file_read_line(file, tab, 4))
    {
      graph_edges    = tab[0];
      graph_vertices = tab[1];
      graph_start    = tab[2];
      graph_end      = tab[3];

      for(int i = 0; i < graph_edges; i++)
        if(file_read_line(file, tab, 3))
        {
          edge_start  = tab[0];
          edge_end    = tab[1];
          edge_weight = tab[2];
        }
        else
        {
          cout << "File error - READ EDGE" << endl;
          break;
        }
    }
    else
      cout << "File error - READ INFO" << endl;

    file.close();
  }
  else
    cout << "File error - OPEN" << endl;
}
```

Opis ze strony: www.cplusplus.com
http://www.cplusplus.com/reference/string/string/getline/

std::getline (string)

```
(1)  istream& getline (istream& is, string& str, char delim);
(2)  istream& getline (istream& is, string& str);
```

```
(1)  istream& getline (istream&  is, string& str, char delim);
     istream& getline (istream&& is, string& str, char delim);
(2)  istream& getline (istream&  is, string& str);
     istream& getline (istream&& is, string& str);
```

Get line from stream into string
Extracts characters from *is* and stores them into *str* until the delimitation character *delim* is found (or the newline character, `'\n'`, for *(2)*).

The extraction also stops if the end of file is reached in *is* or if some other error occurs during the input operation.

If the delimiter is found, it is extracted and discarded (i.e. it is not stored and the next input operation will begin after it).

Note that any content in *str* before the call is replaced by the newly extracted sequence.

Each extracted character is appended to the string as if its member push_back was called.

Parameters

is
: istream object from which characters are extracted.

str
: string object where the extracted line is stored.
The contents in the string before the call (if any) are discarded and replaced by the extracted line.

Return Value

The same as parameter *is*.

A call to this function may set any of the internal state flags of *is* if:

flag	error
`eofbit`	The end of the source of characters is reached during its operations.
`failbit`	The input obtained could not be interpreted as a valid textual representation of an object of this type. In this case, *distr* preserves the parameters and internal data it had before the call. Notice that some `eofbit` cases will also set `failbit`.
`badbit`	An error other than the above happened.

(see ios_base::iostate for more info on these)

Additionally, in any of these cases, if the appropriate flag has been set with *is*'s member function ios::exceptions, an exception of type ios_base::failure is thrown.