Обектно-ориентирано програмиране (записки)

- Илиан Запрянов

Тема 02. Потоци и текстови файлове

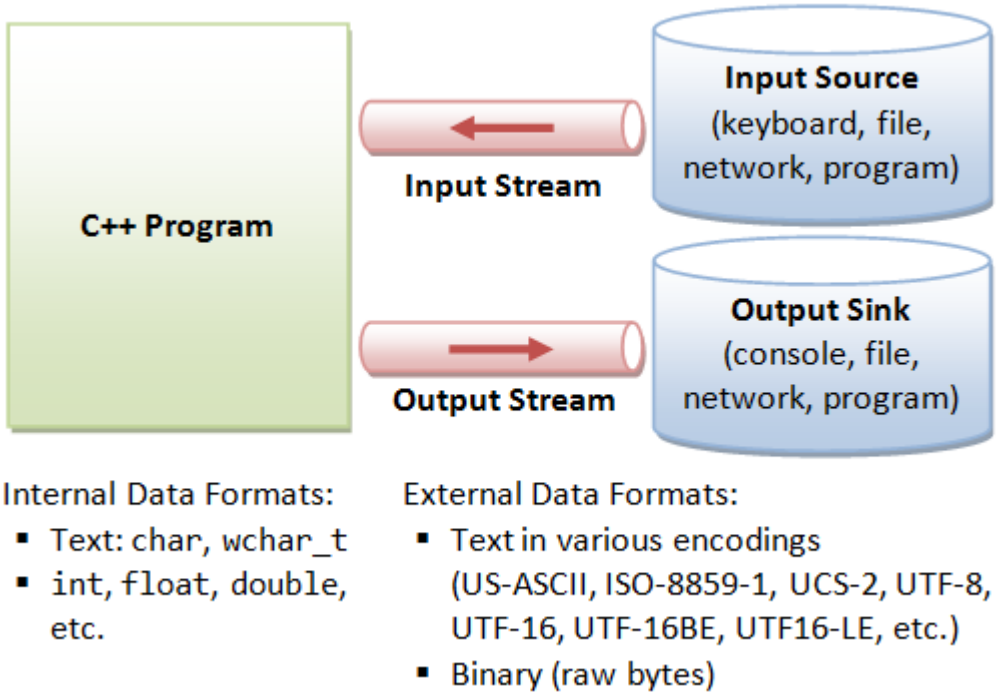
def.| Поток - последователност от байтове “насочени” в определена посока

- При операциите за вход, байтовете идват от източник за вход (клавиатура, файл, мрежа или друга програма)

- При операциите за изход, байтовете данни излизат от програмата и се "вливат" във външно "устройство" (конзола, файл, мрежа или друга програма)

- Потоците служат като посредници между програмите и самите IO устройства по начин, който освобождава програмиста от боравене с тях.

- Потокът дефинира интерфейс с операции върху него, които не зависят от избора на IO устройство

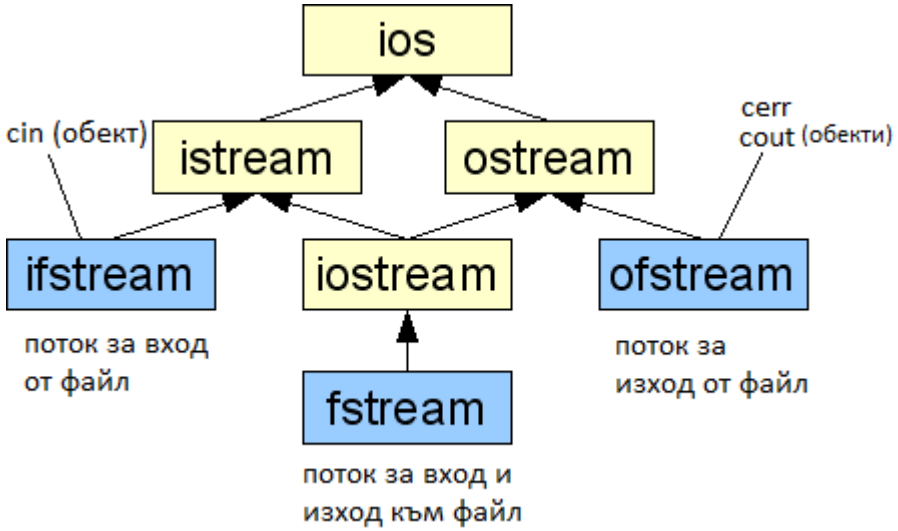


За да извърши вход или изход, една C++ програма:

- Създава поток;
- Свързва потока с IO устройството (напр. конзола, клавиатура, файл, мрежа или друга програма);
- Извършва операции за вход/изход върху потока;
- Прекъсва връзка с потока;
- Освобождава потока;

Видове потоци:

- Потоци за вход
- Потоци за изход




```cpp
#include <iostream>
#include <fstream>

int main()
{
    //отваряне на поток за вход към файл

    std::ifstream fileIn("test.txt"); //важно е да отбележим, че вътрешно (скришно)
    //    ^        ^       ^                        се извиква .open()
    //    |        |       |
    // ifstream   име    файлът, към който
    // (input)           отваряме потока


    //аналогично е отварянето на поток за изход към файл

    std::ofstream fileOut("test.txt"); //важно е да отбележим, че вътрешно (скришно)
    //    ^        ^       ^                        се извиква .open()
    //    |        |       |
    // ofstream   име    файлът, към който
    // (output)           отваряме потока

    fileIn.close();
    fileOut.close();  //<------------------------------
                      //                              |
    return 0;         //                              |
                      //                              |
} // в края на съответния scope, в който е отворен потока скришно се извиква .close(),
  // но е добра практика да го пишем сами
```

Интерфейс

- "интерфейс" обикновено се отнася до набор от функции или методи, предоставени от езика за програмиране или библиотеката, които позволяват на програмиста да извършва операции върху файлове (в случая). Тези операции могат да включват отваряне и затваряне на файлове, четене от файлове, писане във файлове, манипулиране на позицията на четене/писане във файл и други подобни действия.

01. Потоци за изход

- можем да си го представим все едно отваряме един **БЕЗКРАЕН МАСИВ**

При потоците за изход имаме така наречения **put указател, който се мести до първата свободна позиция**

-> форматиран изход - отнася се до процеса на конвертиране на данните в четим за човека формат преди тяхното извеждане. **(<<)**

```
fileOut << <обект> << 37;
//             ^         ^
//             |         |
// символ по символ    всеки обект се интерпретира
// "поставя обекта"    по различен начин, 37 не е един символ
```

-> неформатиран изход - отнася до директното записване на данни в изходен поток без никаква промяна на формата им. Това означава, че данните се предават точно в същия вид, в който са представени в паметта, без да се конвертират в четим за човека формат **(.put() .write())**

```
char ch = 'a';
fileOut.put(ch); //използва се за записването САМО на един
символ

char str[4] = "abc";
fileOut.write((const char*) str, sizeof(str)); //ползва се при
//                                              двоични файлове
//                   ^                ^
//                   |                |
//приема като аргумент          броят на байтовете, които
//                                  искаме да запишем
//константен char пойнтър
//(терминиращата нула не ни трябва)
```

-> синхронизация - тоест правим промени в буфера и след някакъв интервал от време ще се изсипе във файла

.flush() - принудително изпращане на съдържанието на буфера към крайния изход, гарантирайки, че всички междинно съхранени данни са били обработени

```
fileOut << 3;
fileOut << 3;
fileOut << 3;
fileOut << 3;
fileOut.flush(); //изсипваме и запазваме новите промени във
                                                    файла
```

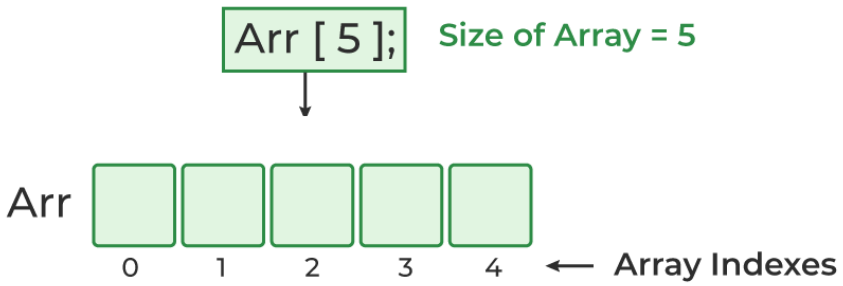
-> позициониране - способността да управляваме текущата позиция във файл

```
fileOut.tellp(); //връща на коя позиция е пойнтърът в момента

//има два начина за използване на seekp

fileOut.seekp(3); //мести пойнтъра 3 позиции напред ОТ НАЧАЛОТО =>

std::cout << fileOut.tellp(); // 3
fileOut.seekp(3);
std::cout << fileOut.tellp(); // 3

-----------------------------------------------------------------

feleOut.seekp(3, std::ios::cur);  //мести пойнтъра 3 позиции напред
                                  //от текущата позиция

fileOut.seekp(0, std::ios::beg);  //мести пойнтъра 0 позиции напред
                                  //от началото

fileOut.seekp(-3, std::ios::end); //мести пойнтъра 3 позиции назад
                                  //от края

// [NOTE] end ни праща след последния символ във файла
```

02. Потоци за вход

При протоците за вход имаме така наречения **get указател, който се намира на текущата позиция за четене**

-можем да си го представим все едно отваряме един **КРАЕН МАСИВ**



-> форматиран вход- се отнася до процеса на четене на данни от входен поток, където данните се интерпретират и конвертират в определени типове на данни **(>>)**

```
fileIn >> <обект>;
//             ^
//             |
//        обект, който
//      знаем как да четем
```

```
int n;
fileIn >> n; //вижда, че е int и чете като int
```

-> неформатиран вход - чете данните точно както са представени, без да променя съдържанието или да пропуска символи **(.get() .getline())**

```
fileIn.get() //връща 1 символ и мести
             //указателя с една позиция

fileIn.get(char* buff, <брой символи>); //запазва до n-1 на брой
                                        //символи в buff и мести указателя с до n-1 позиции
                                        //заради терминиращата 0

file.get(char* buff, <брой символи>, <разделител>) //запазва в buff
                                                   //или докато не стигне броя символи,
                                                   //или разделителя (eof по default)
                                                   //и също мести указателя напред с
                                                   //[0;n-1] позиции


file.getline(char* buff, <брой символи>, <разделител>) //запазва в buff
                                                       //или докато не надмине размера
                                                       //или докато не достигне разделителя
                                                       //('\n' по default)
                                                       //мести указателя до след разделителя
```

```cpp
//[!] Важно е да разграничаваме get() и getline()
//getline() прескача разделителя, който сме задали ('\n' по default)
//get() НЕ прескача разделителя, който сме задали (EOF по default)

is.getline(<buff>, <size>, 'X');
is.get(<buff>, <size>, 'X');
//-------------------------------------------------------------------------
//Нека нашият ред е ABCX123
//getline() ще напълни buff с ABC\0, защото среща нашия разделител 'X',
//прескача го и на следващата входна операция X няма да бъде отразен, т.е

char strOne[10] = {};
std::cin.getline(strOne, 4, 'X');

char strTwo[10] = {};
std::cin.getline(strTwo, 10); //помним, че по default разделителя е нов ред

std::cout << "[STR ONE]: " << strOne << std::endl; //ABC
std::cout << "[STR TWO]: " << strTwo << std::endl; //123

//get() ще напълни buff с ABC\0, защото среща нашия разделител 'X',
//НЕ го и на следващата входна операция X ЩЕ бъде отразен, т.е

char strOne[10] = {};
std::cin.get(strOne, 4, 'X');

char strTwo[10] = {};
std::cin.get(strTwo, 10); //помним, че по default разделителя е EOF
                          //и cin е поток за вход насочен някъде (абстракция)

std::cout << "[STR ONE]: " << strOne << std::endl; //ABC
std::cout << "[STR TWO]: " << strTwo << std::endl; //X123
```

Като извод можем да кажем, че:
getline = get + ignore -> тоест местим пойнтъра до разделителя с помощта на get, и игнорираме разделителя с ignore, за да го прескочим
(както прави getline)

get = getline + seekg(-1, ios::curr) -> тоест местим пойнтъра до СЛЕД разделителя с помощта на getline и се връщаме назад с една позиция с помощта на seekg, за да се върнем на разделителя (където отиваме с get)

-> позициониране - способността да управляваме текущата позиция във файл

```
fileIn.tellg(); //връща на коя позиция е пойнтърът в момента

//има два начина за използване на seekp

fileIn.seekg(3); //мести пойнтъра 3 позиции напред ОТ НАЧАЛОТО =>

std::cout << fileIn.tellg(); // 3
fileIn.seekg(3);
std::cout << fileIn.tellg(); // 3

-----------------------------------------------------------------

fileIn.seekg(3, std::ios::cur);  //мести пойнтъра 3 позиции напред
                                  //от текущата позиция

fileIn.seekg(0, std::ios::beg);  //мести пойнтъра 0 позиции напред
                                  //от началото

fileIn.seekg(-3, std::ios::end); //мести пойнтъра 3 позиции назад
                                  //от края

// [NOTE] end ни праща след последния символ във файла

-----------------------------------------------------------------

//може да преместим пойнтъра, без да връщаме резултат
fileIn.ignore(<брой символи, които искаме да пропуснем>, <разделител>)
//                              ^                                ^
//                              |                                |
//                       по default 1                   по default eof
//                  (unsigned -> не можем да
//                                връщаме назад;
//
//           Напр.: не можем да игнорираме -1 символа
```

```
//форматирания вход прескача ' ', '\n', '/t'
//или с други думи ги смята за излишни
//но има една особеност
int a;
std::cin >> a; //когато std::cin прочете всичко, което му е казано приключва, тоест
               //ако отидем на нов ред след std::cin, новият ред няма да се прескочи

char buff[1024];
std::cin.getline(buff, <размер>); //разделителят е '\n', но след въвеждането на [a],
                          //имаме нов ред, който не сме отразили => buff ще бъде празен,
                                  //тъй като веднага среща разделителя

------------------------------------------------------------------------
//в такива ситуации използваме ignore()

int a;
std::cin >> a;
std::cin.ignore(); //прескачаме новия ред

char buff[1024];
std::cin.getline(buff, <размер>); //записваме в buff
```

```
//това няма да бъде проблем, ако
//1. разделителят ни не е '\n'
//2. не слагаме разделител между [a] и [buff]

//1.
int a;
std::cin >> a;


char buff[1024];
std::cin.getline(buff, <размер>, 'X');

//ако входът ни е 123\nabcX
std::cout<< a << " "; //123 -> чете числото

std::cout<< buff;  //\nabc -> тъй като разделителят не е
                   //'\n', '\n' просто се записва в buff

//2.
int a;
std::cin >> a;


char buff[1024];
std::cin.getline(buff, <размер>);

//ако входът ни е 123abc, тоест без разделител между тях
std::cout<< a << " "; //123
std::cout<< buff; //abc

//очевиден е проблемът, че ако искаме [a] да е 12, а [buff] - 3abc,
//това е невъзможно без да разделим числото и стринга, тъй като по този начин
//записваме в [a], докато срещнем символ, който не е цифра
```

```
//за потоци имаме и командата .peek(),
//която връща текущия символ и не мести указателя

//Например, ако имаме файл със съдържание (32abc)

fileIn.get(); //прескачаме "3"
std::cout << (fileIn.peek()); //намираме се на "2" и го отпечатваме

//разликата между .peek() и .tellg() е
//1. .peek() ни казва на кой символ сме и не мести указателя нататък
//2. .tellg() ни казва на кой индекс сме във файла и не мести указателя нататък

std::ofstream fileOut("test.txt");
fileOut << "123";
fileOut.close();

std::ifstream fileIn("test.txt");

std::cout << fileIn.tellg() << " "; //0 -> намираме се на нулевия индекс

fileIn.get(); // прескачаме "1"

std::cout << fileIn.peek() - '0' << " "; //2 -> намираме се на "2"

std::cout << fileIn.tellg() << " "; //1 -> намираме се на първи индекс
std::cout << fileIn.tellg() << " "; // 1 -> tellg() не мести указателя и все още
сме на първи индекс

fileIn.get();
std::cout << fileIn.tellg() << " "; //2 -> намираме се на втори индекс, защото
                                    //get() мести указателя в случая с 1 позиция
```

Режими на работа

Режимите на работа представляват 8-битово число. Всеки бит има стойност 1, ако е вдигнат и 0, ако не е. Всеки бит отговаря за различно нещо. Например ако последният бит е вдигнат, това означава, че файлът е отворен за вход


```
//ofstream ofs(<име>, <число>)


std::ofstream fileOut("text.txt, 10);

//числото 10 в двоичен вид е 0000 1010 => ще бъде отворен за
конкатенация и изход
```

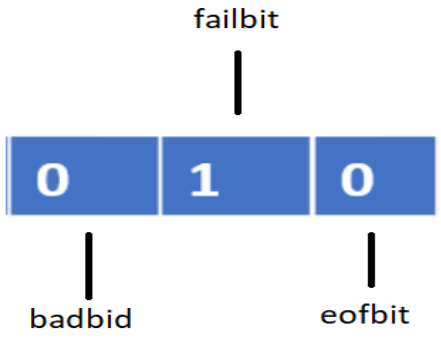
```cpp
std::ofstream fileOutContain("text.txt");
fileOutContain << 1;
fileOutContain.close();

//ако имаме файл със съдържание (1) и отворил файлът за
конкатенация, то
std::ofstream fileOutConc("text.txt", 10); //конкатенация и изход

fileOutConc << 2;
fileOutConc.close();

//във файлът ще се запише съдържанието (12), тъй като чрез
конкатенацията
//сме отишли накрая и пишем след това, а не пишем отгоре на
съдържанието на файла,
//тоест не заменяме 1 с 2, както сме свикнали, а ги слепваме и
става 12
```

```cpp
//с цел да не помним числа наизуст, за да вдигнем даден бит, за
//удобство са създадени специални флагове, които го правят
//ако искаме да вдигнем няколко бита, можем да ги изредим с побитовата
операция или ("|")
std::ofstream fileOut("text.txt", std::ios::app | std::ios::out);

fileOut.close();
```

ios	Ефект:	
ios::in	Отваря файл за извличане.	1
ios::out	Отваряне на файл за вмъкване. Допуска се вмъкване на произволни места във файла. Ако файлът съществува, съдържанието се изтрива.	2
ios::ate	Отваря за вмъкване и установява указателя put в края на файла. Допуска вмъкване на произволни места.	4
ios::app	Отваря за вмъкване и установява указателя put в края на файла	8
ios::trunc	Ако файлът съществува, съдържанието се изтрива.	16
ios::binary	Превключва режима от текстов в двоичен	32
ios::_Nocreate	Отваря за вмъкване, само ако файлът с указаното име съществува.	64
ios::_Noreplace	Отваря за вмъкване само ако файлът с указаното име не съществува.	128

Състояние на потока

Подобно на битовете за режим на работа, съществуват битове за състояние на потока, които също са число.



съществува и **goodbit**, който не е част от това число и е равен на 0, ако има операция, която не се е извършила успешно, това се проследява чрез другите флагове.

Флагове на състоянията на потока

Флаг:	Значение:
bad()	Има загуба на информация. Някоя операция за четене и писане не е изпълнена.
fail()	Последната входно/изходна операция е невалидна.
good()	Всички операции са изпълнени успешно.
clear()	Изчиства състоянието на потока (Вече good() ще върне истина).
eof()	Достигнат е края на файла.

Разликата между ***fail()*** и ***bad()*** е, че когато е вдигнат ***badbit-a*** има проблем с потока за вход или този за изход. Това означава, че сме загубили информация и води до невъзможно ползване на следващи операции. ***Failbit-a*** се вдига когато имаме проблем с логиката, например искаме да прочетем някакво число, но ни подават буква. Тоест не губим информация и са възможни следващи операции.

iostate value (member constant)	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

Когато е вдигнат ***badbit-a***, то задължително е вдигнат и ***failbit-a***. Това означава, че не можем директно да проверим дали ***failbit-a*** е вдигнат поради логически проблем. За да проверим това ни е нужна проверката

```
if(file.fail() && !file.bad())
{
     std::cout<<"Fail";
}
```

```
//Имаме командата .clear(), която изчиства състоянието на потока,
//fail, bad, eof стават 0

file.clear();
```

Stringstream

Stringstream представлява нещо като **“фалшив поток”** към даден **стринг**

```
std::stringstream ss("33"); //отваряме поток към стринг,
                            // ЧИЕТО СЪДЪРЖАНИЕ е "33"

//главната му полза е за по-бързо и ефективно прехвърляне
//на int v char arr и обратно
int a = 0;
ss >> a; //присвояваме стойност на [a], имплицитно превръща "33" в int
std::cout << a << std::endl; // 33
```

```cpp
std::stringstream ssTwo("33Test"); //отваряме поток към стринг, ЧИЕТО СЪДЪРЖАНИЕ е "33Test"

int b = 0;
ssTwo >> b; //присвояваме стойност на [b], имплицитно превръща "33" в int,
            //33, защото докато чете стига до символ, който не е цифра и спира

std::cout << b << " "; //33

char strTwo[10];
ssTwo >> strTwo; //в sstTwo ни остана "Test", и го записваме в strTwo
std::cout << strTwo << std::endl; //Test
```

```cpp
//Забележете, че използването на stringstream е в доста специфични случай, напр.:

std::stringstream ssThree("Test33"); //отваряме поток към стринг, ЧИЕТО СЪДЪРЖАНИЕ е "Test33"

int c = 0;
ssThree >> c; //присвояваме стойност на [c], но още първият символ не е цифра
              //=> няма да му запазим нищо => ще се вдигне failbit-а, заради
              //грешка, с която не се губи информация => трябва изчистим състоянието на потока

ssThree.clear(); //изчистваме потока

std::cout << c << " "; //0 (default-натата стойност)

char strThree[10];
ssThree >> strThree; //в ssThree имаме "Test33", и го записваме в strThree
std::cout << strThree; //Test33
```

```
//можем да създаваме фалшиви потоци без нищо в тях
//и съответно в някакъв етап от нашата програма да запишем в тях
std::stringstream ss;

//пишем 5 в ss
ss << 5;

//тъй като ss е "поток", макар и фалшив,
//можем да ползваме интерфейса на потоците
ss.seekp(1, std::ios::beg);
ss.seekg(1, std::ios::beg);

ss.tellg();
ss.get();

char str[10];
ss.getline(str, 10);
```

Примери, които трябва да знаем:

https://github.com/Angeld55/Object-oriented_programming_FMI/tree/master/Week%2002