# Датотеки. Справување со грешки

Сите програми кои што ги разгледувавме досега функционираа на принципот на прибирање и печатење на информации на конзола (т.н. стандарден влез/излез). Во ова предавање ќе зборуваме за датотеки и начинот на нивно користење во програмскиот јазик C++.

Во C++, доколку сакаме да читаме податоци од некоја датотека, потребно е да креираме променлива од типот (класата) ifstream. Притоа, името на датотеката го наведуваме при самото креирање на променливата, или подоцна - со повикување на методот open(). За запишување на податоци во одредена датотека потребно е да креираме променлива од типот ofstream. Самото читање и запишување се врши преку вметнување (out << "x") или извлекување податоци (in >> prom) од соодветниот поток кој е дефиниран со променливата - слично како што читавме податоци од стандарден влез (cin) и запишувавме податоци на стандарден излез (cout).

Следната програма креира датотека со име "output.txt" и во неа ги запишува броевите од 1 до 10. Притоа, забележете дека пред да ги користиме типовите ifsteam и ofstream, потребно е да ја вклучиме соодветната датотека во која се тие дефинирани (#include <fstream>).

Програма 17.1

```
#include <fstream>
using namespace std;
int main()
{
ofstream out("output.txt");
for (int i=1; i<=10; i++)
out << i << endl;
return 0;
}
```

Во продолжение е дадена уште една програма која ги запишува броевите од 1 до 10 - овојпат, наместо да го наведеме името на датотеката при самото креирање на променливата out, го повикуваме методот open(), со еден аргумент: името на датотеката во која сакаме да запишуваме податоци:

Програма 17.2

```
#include <fstream>
using namespace std;
int main()
```

```
{
ofstream out;
out.open("output.txt");
for (int i=1; i<=10; i++)
out << i << endl;
return 0;
}
```

Доколку датотеката во која сакаме да запишуваме податоци (на пример, "C:/output.txt") веќе постои, преддефинираното однесување на ofstream е да ги избрише податоците кои се наоѓаат во постоечката датотека и да креира нова датотека со истото име. Доколку сакаме, наместо да ги пребришеме старите податоците, нив да ги зачуваме, и, да ги "залепиме" новите податоци на крајот од таа датотека, потребно е да наведеме "ios::app" како втор аргумент на методот open() или како втор аргумент при креирањето на самата променлива:

Програма 17.3

```
#include <fstream>
using namespace std;
int main()
{
ofstream out("output.txt", ios::app);
for (int i=1; i<=10; i++)
out << i << endl;
return 0;
}
```

# Тековна состојба

Класите ifstream и ofstream нудат неколку методи преку кои може да се провери тековната состојба на потоците:

| метод | објаснување |
|---|---|
| is_open() | враќа true доколку датотеката е отворена |
| bad() | враќа true доколку претходната операција завршила неуспешно (на пример, се обидуваме да запишуваме во датотека која се наоѓа на диск на кој нема повеќе простор) |
| fail() | враќа true доколку претходната операција завршила неуспешно (види bad()), |

| | |
|---|---|
| | или програмата не успеала да прочита податок бидејќи тој не се наоѓа во соодветен формат (на пример, се обидуваме да читаме цел број, но во датотеката има запишано букви) |
| eof() | враќа true доколку во влезната датотека нема повеќе податоци за читање |
| good() | враќа true доколку последната операција завршила успешно |

Следната програма го илустрира начинот на користење на дел од методите наведени погоре:

Програма 17.4

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
ifstream dat("input.txt");
int sum = 0;
if (dat.is_open() == false)
{
cout << "datotekata ne e otvorena!";
//zavrshi so izvrshuvanje na programata
return 0;
}
while (true)
{
int temp;
dat >> temp;
if (dat.eof() == true)
{
break;
}
if (dat.fail() == true)
{
cout << "tekovniot podatok ne e cel broj!" << endl;
} else
{
sum += temp;
}
}
cout << "Zbirot na site celi broevi e: " << sum << "." << endl;
return 0;
}
```

Важно е да се забележи дека ifstream и ofstream нудат метод за затворање на соодветната датотека со која работиме - close(). Интересно, во нашите програми ние никогаш не го повикавме овој метод, а програмите и натаму работеа како што очекувавме. Причината за тоа е што, доколку креираме променливи од тип ifstream и/или ofstream и нив ги користиме за работа со одредена датотека, самата програма е задолжена да ја затвори датотеката кога ќе заврши блокот на наредби (наредби ограничени со '{' и '}') во кој е дефинирана променливата која служи за пристап до податоците од таа датотека. На пример, во примерите кои беа дадени погоре, датотеките ќе бидат затворени кога ќе заврши со извршување функцијата main(). Доколку сакаме експлицитно да затвориме датотека во одреден момент, потребно е да го повикаме методот close() - без аргументи.

# Меѓумеморија

Со цел поефикасно користење на дисковите, податоците кои се вметнуваат во одреден податочен поток може да се сместат во меѓумеморија - да се групираат и чуваат во меморија пред да се запишат на диск. На пример, доколку во една датотека набрзина запишеме десет цели броеви, можно е системот да одлучи дека е поефикасно да почека додека ги дознае сите десет броеви (нив да ги смести во меѓумеморија), а потоа, набрзина, да ги запише овие податоци на диск.

Самата програма е задолжена да одлучи кога и како ќе ја испразни меѓумеморијата и ќе ги запише податоците на диск. Доколку овој факт не ви се допаѓа или се наоѓате во ситуација во која е потребно запишување на одредена група на податоци веднаш, може да го повикате методот flush(). Овој метод служи за празнење на меѓумеморијата и тој гарантира дека сите податоци кои дотогаш биле вметнати во соодветниот податочен поток ќе бидат запишани на диск.

Програма 17.5

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
ofstream dat("output.txt");
dat << "a";
dat.flush();
dat << "b";
dat.flush();
dat << "c";
dat.flush();
```

```
return 0;
}
```

# Стандардни методи за читање/печатење

Важно е да напоменеме дека е возможно читање на цел ред податоци од една датотека на ист начин на кој што читавме цел ред податоци од стандардниот поток cin - со повикување на функцијата getline(stream, string).

cСлично, во датотеките (како кај стандардниот поток cout) може да запишуваме децимални броеви на ист начин како што запишувавме децимални броеви во излезните потоци за текстуални податоци од тип ostringstream - со користење на fixed и precision(X).

Следнава програма ги илустрира сите овие методи - најпрвин таа чита цел ред податоци, потоа проверува дали во тој ред се наоѓа зборчето "PI" (и ништо друго), и потоа, на стандарден излез го печати бројот pi - заокружен на 4 децимали:

Програма 17.6

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
using namespace std;
int main()
{
ifstream dat("input.txt");
string line;
getline(dat, line);
if (line == "PI")
{
double pi = 3.14159265;
cout.precision(4);
cout << fixed << pi << endl; //pechati '3.1416'
}
return 0;
}
```

# Справување со грешки

Предавањето за датотеки е одлично место да се запознаеме со три стандардни начини на справување со грешки: cerr, assert(uslov) и goto.

Како што споменавме во едно од почетните предавања, печатењето на текст во C++ се прави преку "вметнување" на текст (cout << "tekst" << endl;) во стандардниот поток cout. cout го означува излезниот поток на тековната конзола (конзолата во која се извршува нашата програма).

cerr е поток кој е многу сличен на cout - но кој, за разлика од cout, се користи за печатење на пораки кои означуваат грешка. Слично како и cout, и cerr е деклариран во датотеката iostream и, истиот се користи на таков начин што се вметнуваат податоци (со помош на операторот '<<') во соодветниот поток. По правило, овие пораки се печатат на екран (во конзолата во која се извршува програмата), но е можно да се препраќаат и на друга локација (на пример, во датотека).

Извадок 17.1

```
ifstream dat("input.txt");
if (dat.is_open() == false)
cerr << "ne postoi takva datoteka!";
else
{
........
........
}
```

Во Windows, доколку сакаме пораките да се запишуваат во датотека (наместо на екран), можеме да ја извршиме следната наредба (во конзола):

program.exe 2>stderr.txt

Слични наредби можат да се извршат и во останатите оперативни системи.

Вториот начин на пријавување на грешки кој ќе го споменеме е макрото assert(uslov). За разлика од претходниот метод, користењето на assert(uslov) резултира со запирање на извршувањето на програмата. Имено, доколку условот кој ќе го предадеме како аргумент на assert(uslov) е исполнет, програмата продолжува со извршување како ништо да не се случило. Од друга страна пак, доколку условот не е исполнет, програмата печати порака

за грешка (давајќи ја линијата на која се случила истата) и запира со извршување. Макрото assert(uslov) е декларирано во датотеката <cassert>.

Програма 17.7

```
#include <iostream>
#include <cassert>
using namespace std;
int main()
{
int a, b;
cin >> a >> b;
assert(b != 0);
cout << (a/b) << endl;
return 0;
}
```

Во C++, како и во повеќето други модерни јазици, постои начин за прескокнување на одреден број на наредби. На пример, следнава програма користи наредба goto за менување на текот на програмата и контрола на внесените податоци од страна на корисникот:

Програма 17.8

```
#include <iostream>
#include <new>
using namespace std;
int main()
{
start:
cout << "Vnesi eden paren broj: ";
int N;
cin >> N;
if (N%2 != 0)
{
cout << "Vnese neparen broj. ";
cout << "Obidi se povtorno." << endl;
goto start; //skokni do "start:"
}
cout << "Vneseniot paren broj e: " << N << endl;
return 0;
}
```

Програмата дадена погоре ќе го повторува делот за печатење на порака "Vnesi eden paren broj: " и читање на вредност на променливата N се додека корисникот не внесе број делив со 2 (парен број). Кога корисникот ќе внесе парен број, програмата ќе испечати "Vneseniot paren broj e: " и вредноста на променливата N (во еден ист ред).

Многу програмери (вклучувајќи го и авторот на овие предавања) избегнуваат да користат goto наредби. Ваквиот начин на пишување на програми води до код кој е многу тежок за сфаќање и контрола - кога една програма содржи повеќе goto наредби, практично е невозможно да се разбере логиката и текот на истата. Од друга страна пак, кога една програма има повеќе функции и секоја функција има име кое соодветствува на работата која таа функција ја врши, кодот е многу полесен за разбирање и анализа.

За крај, еден совет: скоро секоја програма која може да се напише со goto, може многу полесно да се напише со помош на функции, циклуси и/или наредби за условно извршување. Избегнувајте да користите goto наредби по секоја цена.