Обектно-ориентирано програмиране
(Информатика, Информационни системи)
2021/ 22 ФМИ, СУ

Файлове
До сега… данни се съхраняват в паметта, с която работи една програма. Тази памет е ограничена, достъпът до нея е бърз, но тази памет се изчиства (освобождава) веднага след като програмата приключи. Къде и как може да се съхраняват по-големи обеми от данни и съхранението да бъде за постоянно?

Като механизъм за съхраняване и извличане на информация могат да се използват файловете.

Файл. Представяне.
Файлът е именувана последователност от данни (байтове), които са подредени последователно. Може да считаме, че номерацията им започва от 0 (подобно на масивите) и завършва с маркер за край на файл.

Типове файлове.
Файловете могат да се класифицират по няколко признака.

Според **операциите**, които могат да бъдат извършвани върху файловете, различаваме:

- файлове само за *писане*;
- файлове само за *четене;*
- файлове *и за четене, и за писане*.

Според **начина на достъп** до данните различаваме:

- файлове с последователен достъп;
- файлове с пряк достъп.

Езикът не налага никакви изисквания относно структурата на файловете. По тази причина, файловете се структурират според нуждите на приложението, което се разработва. Трябва да се вземат предвид например изисквания като: какви данни трябва да се съхраняват, от какъв тип, в каква последователност, какви действия ще се прилагат върху тях и др. Важно е следното. *Начинът, по който данните са съхранени (записани) във файла, определя начина, по който тези данни в последствие ще бъдат прочетени (извлечени) от файла.*

Например, нека е необходимо данни за студентите от ФМИ да бъдат съхранени във файл. За всеки студент трябва да се поддържа информация за факултетен номер (цяло число), име (символен низ), среден успех (число с плаваща точка). Данните за един студент могат да се разглеждат като един запис във файла.

Какво означава файл с *последователен* достъп?

Данните във файловете с последователен достъп се обработват последователно от началото към края, в реда, в който са записани. Данните обикновено са с различен размер.

Пример за запис

123 Ivan Ivanov 5.78 (разделител между полетата в записа е табулацията)

За да се прочете средния успех, трябва да се изчете всичко преди него. Няма начин да се достъпи директно, защото не може да се дефинира някакво универсално правило за достигане до тази част

от записа. Ако за конкретния запис трябва да бъдат прескочени 16 символа, дали това ще важи и за останалите записи във файла? Също така няма възможност да се върнем назад и да получим достъп до вече прочетени данни.

Какво означава файл с *пряк* достъп?

Когато се спомене пряк достъп, първата асоциация са масивите. Съществува пряк достъп до всеки от елементите в даден масив. Когато трябва да се достъпи елемент на определена позиция, всички елементи преди него се прескачат без да се прочитат.

Защо е възможно това? Защото масивът съдържа елементи от един и същи тип, елементи с еднакъв размер.

Аналогично, за да се даде възможност за пряк достъп във файл, записите в него трябва да бъдат с *еднакъв* размер, да се обработват като едно цяло, а не като отделни полета (характеристики).

Как се постига това? Малко по-късно.

Според **типа на данните** различаваме:

- текстови файлове;
- двоични файлове.

Двоичните файлове често са копия от паметта, еквивалентни или близки до представянето на данните на програмата в паметта. За разлика от тях, *текстовите* са близки до представянето на данните на екрана или принтера.

Текстовите файлове са последователност от символи, в тях има форматиране. Например всеки запис може да е на отделен ред, полетата от записа могат да бъдат подравнени (вляво, вдясно), да бъдат изведени с определена дължина, да бъдат разделени с някакъв разделител (табулация, запетая или др.). Поради това, тестовите файлове са по-лесни за четене от нас (потребителите), но по-трудни за обработка програмно. Обикновено достъпът до данните в тях е последователен.

В двоичните файлове няма форматиране. В тях са записани суровите (чистите) данни като последователност от байтове. Тези файлове са по-лесни за обработка програмно, но не са четими за потребителя. В зависимост от начина, по който са организирани данните в тях, достъпът често е пряк.

Сравнение. *Основна разлика между текстови и двоични файлове с прост пример.*

Числото 123 се записва като 3 различни символа в текстов файл, но в паметта може да заема 1, 2, 4 байта, в зависимост от типа на променливата, в която е съхранено.

Изобщо, ако една целочислена променлива от тип `int` е записана в текстов файл, тя може да бъде записана с различен брой символи (от 1 до 11 символа), докато в двоичен файл или в оперативната памет, променливата винаги заема 4 байта.

Входно-изходни операции от гледна точка на нашата програма.

Когато данни се съхраняват програмно във файл, те се пренасят от оперативната памет към файла. Това се нарича още *изходна операция* за една програма.

Когато данни от файл се четат програмно, те се пренасят от файла към оперативната памет. Това се нарича още *входна операция* за една програма.

В C++ съществува понятието *поток*. Потокът е абстракция, с която описваме връзката между програмата и даден файл или устройство. За потока може да се мисли като за канал за комуникация, канал, по който данните протичат в определена посока (от паметта към файла или обратно).

Потоците, които са разглеждани в курса по УП са входният поток `cin`, свързан със стандартния вход, и изходният поток `cout`, свързан със стандартния изход. Техните дефиниции са достъпни чрез заглавния файл `<iostream>`.

Например:

```
int var;
cin >> var;
```

Операцията за вход >> показва в каква посока протичат данните – от стандартния вход (клавиатурата) към паметта, с която работи програмата. Самите данни могат да се възприемат като поредица от символи.

Ако от клавиатурата се въведе 765a, това се разглежда като последователност от символите ‘7’, ‘6’, ‘5’ и ‘a’. Тъй като трябва да се прочете цяло число, операцията за вход започва да извлича последователно символите, които могат да бъдат част от едно цяло число. Четенето ще спре върху символа ‘a’, който не може да бъде част от такова. Символите, които се прочитат, се премахват от потока.

Как да се направи връзката между поток и файл? От какъв тип трябва да бъде потокът?

Файлови потоци

В езика C++ има три вида файлови потоци, които могат да се използват след включването на заглавния файл `<fstream>` (`съкратено от file stream`):

- `ifstream` (`съкратено от input file stream`) предназначен за връзка с файлове, от които се четат (извличат) данни;
- `ofstream` (`съкратено от output file stream`) предназначен за връзка с файлове, в които се записват данни;
- `fstream` (`съкратено от file stream`) предназначен за връзка с файлове, с които ще се извършват както входни, така и изходни операции, т.е. файлове за четене и писане.

За разлика от `cin` и `cout`, които са предварително дефинирани, свързани със съответните устройства и готови за използване, файловите потоци трябва да бъдат свързани с файл явно.

Как се отваря файл за записване?

При записване на информация във файл, потокът на данните е от оперативната памет към файла, т.е. операцията е *изходна* за програмата.

Необходим е *изходен файлов поток*.

```
// заглавният файл, в който са декларирани файловите потоци
#include <fstream>
...
// създаване на изходен файлов поток
std::ofstream outputFile;
// потокът трябва да бъде свързан с файла, в който ще се записва,
```

```
// като се извика функцията open и се укаже пътят до него
outputFile.open("hello.txt");
// или директно след името на потока
std::ofstream outputFile("hello.txt");
```

Файлът може да се укаже с *относителен* или *абсолютен* път. Пътищата са специфични за операционната система. Стандартът на езика не ги дефинира.

Ако е използван *относителен* път до файла, то пътят се търси спрямо папката, която е текуща за приложението – най-често от където то е стартирано.

Пример за относителен и пълен път (Windows):

Нека пътят до изпълнимия файл на приложението е:

```
c:\folder\myprogram.exe
```

А пътят до текстовия файл, в който ще се записва, е:

```
c:\myfile.txt
```

За да се подаде правилният път до файла `myfile.txt` във функцията `open` може да се използва следният *относителен* път:

```
"../myfile.txt" или "..\\myfile.txt"
```

или *пълният* път

```
"c:/myfile.txt" или "c:\\myfile.txt"
```

Пътят се подава като символен низ, ограден с кавички. В този случай символът \ е escape символ. Трябва да се използват две обратно наклонени черти \\ или една /.

Ако пътят е описан по следния начин

```
“./files/myfile.txt”,
```

то в текущата за приложението директория, се търси папка `files`, а в нея файлът `myfile.txt`.

Изходният поток, който се създава и свързва с файла, може да носи произволно, валидно за езика име, подобно на всяка една променлива, която се създава.

Как се отваря файл за четене?

При извличане (четене) на информация от файл, потокът на данните е от файла към оперативната памет, т.е. операцията е *входна* за програмата.

Необходим е *входен файлов поток*.

```
std::ifstream inputFile;
inputFile.open(“hello.txt”);
```

Променливата за входния поток, който се създава и свързва с файла, може да носи произволно, валидно за езика име.

Проверка дали файл е отворен успешно

Особено, когато файлът се използва за четене, е важно да се провери дали функцията `open` е завършила успешно. Един начин да се направи това е като се използва булевата функцията

`is_open()`, други ще бъдат описани по-долу след като се разгледат възможните състояния на един поток.

```
if (!inputFile.is_open())
{
	std::cout << "The file cannot be opened for reading!\n";
	// други действия…
}
```

Какво може да се обърка?

Някои често срещани проблеми са следните. Възможно е файлът или пътят към него да не съществува, т.е. да не може да бъде намерен на указаното място. Възможно е за нашата програма да не е разрешен достъпът за четене или писане до този файл

Режими за достъп до файл.

Какво ще се случи, ако във файлът, в който искаме да пишем, вече има записани някакви данни?

Нека достъпът до файл за вмъкване (запис) е осъществен както в примера по-горе. Ако приложението се стартира няколко последователни пъти, се забелязва, че ако преди стартирането във файла има записани някакви данни, то след стартиране на програмата (след отваряне на файла) тези вече съществуващи данни се изтриват. Има ли начин файлът да бъде отворен, но старото му съдържание да се запази, т.е. файлът да се отвори и новите данни да бъдат добавени в края, вместо на мястото на старите данни?

Функцията `open` има още един аргумент, с който можем да се укаже режимът, в който се отваря даденият файл. Допустими са следните режими за достъп до файл:

Режим	Значение
`ios::in`	За извличане. Подразбира се за `ifstream`.
`ios::out`	За вмъкване. Може да се пише на произволни места във файла. Ако файлът съществува, съдържанието му се изтрива. Подразбира се за `ofstream`.
`ios::app`	За вмъкване, като се пише само в края на файла.
`ios::ate`	Този режим запазва всички останали, като допълнително премества маркера в края на файла.
`ios::binary`	Файлът се отваря в двоичен режим, не в текстов.
`ios::trunc`	Ако файлът съществува, съдържанието му се изтрива. Това поведение се подразбира за режима `ios::out`.

За да е успешна операцията `open` с режим `ios::in`, файлът задължително трябва да съществува.

Ако файлът е отворен за вмъкване (запис), но файл с указаното име не съществува, той може да бъде създаден, стига по пътя до файла да няма несъществуващи директории.

В горния пример, ако пътят до файла е подаден като “`./files/myfile.txt`” и папката `files` не съществува, файлът няма да бъде създаден и операцията `open` ще пропадне. Ако папката съществува, но самият файл не, то ще бъде създаден нов празен файл с указаното име.

Режимите могат да бъдат комбинирани с побитовата операция or (`|`).

Например, за да се отвори файл за четене (извличане), но не в текстов режим, комбинацията е:

```
inputFile.open("hello.dat", ios::in | ios::binary);
```

Подразбиращият се режим за поток `fstream` е `ios::in | ios::out`. Файлът се отваря и за четене, и за писане. Няма да бъде създаден, ако не съществува, а отварянето му ще предизвика грешка.

```
std::fstream file("helloRW.txt", std::ios::in | std::ios::out);
```

Затваряне на файл

Файлът е *външен* ресурс. Това предполага, че достъпът до него трябва да се контролира правилно. След като ресурсът е заявен с функцията `open()`, след приключване на работа с него, ресурсът трябва да бъде освободен. Използва се функцията `close()`.

```
outputFile.close();

inputFile.close();
```

Как пишем в текстов файл с последователен достъп?

За записване (въвеждане) на данни в текстов файл, с помощта на изходен файлов поток, могат да се използват същите операции, които са валидни и за извеждане на стандартния изход.

За форматиран изход в потока може да бъде използван операторът за изход <<. В допълнение, могат да бъдат приложени и различни манипулатори на изхода (boolalpha, showbase, showpos, uppercase и др.)

```
outputFile
    << "Here is a formatted number: "
    << std::setiosflags(std::ios::fixed) << std::setprecision(2)
    << 12.34567 << std::endl;
```

Може да бъде използвана и функцията `put()`, която ще запише в потока един символ.

```
outputFile.put('Y');
```

Как четем от текстов файл с последователен достъп?

За четене (извеждане) на данни от текстов файл, с помощта на входен файлов поток, могат да се използват същите операции, които са валидни и за четене от стандартния вход.

`operator >>`

Операторът за вход >> обработва последователно символите от потока, докато те могат да бъдат интерпретира като валидни за посочения тип. Например, ако се чете цяло число, четенето продължава, докато от потока се извличат цифри. За разделители се използват празни символи (интервал, табулация, нов ред).

```
int number;

inputFile >> number;
```

Други особености на оператора за вход:

- Операторът за вход прескача белите полета (интервали, табулации и нови редове) в началото на потока.
- След валидна последователност от символи, спира да чете при срещането на бяло поле.
- Прочетените символи се премахват от потока.

- Ако трябва да бъде прочетен символен низ, не се прави проверка за препълване на паметта, в която се записват символите.
- С оператора за вход не може да бъде прочетен низ, съдържащ бели полета.

int get();

Прочита се един символ, не се пропуска нищо и връща цялото число, съответстващо на символа в ASCII таблицата. Ако се достигне края на файла, функцията връща специална стойност, дефинирана с макроса EOF. Тази форма на `get()` не е особено удобна, заради начинът, по който се проверява за достигнат край на файл.

```
int c = inputFile.get();
```

istream& get(char&);

Прочита се един символ, не се пропуска нищо, а символът се записва в аргумента на функцията. Резултатът от функцията е самият поток и може да се провери състоянието му.

```
char c;
inputFile.get(c);
```

istream& get (char* s, streamsize n, char delim);

Прочита редица от символи от потока и ги съхранява в символния низ `s`, до прочитането на (n – 1) символа или до достигането на разделителя `delim`. По подразбиране за разделител се използва символът за нов ред '\n'. Разделителят НЕ СЕ премахва от входния поток. Следващото четене ще започне от него. Терминиращата нула '\0' се добавя автоматично в края на низа `s`. Резултатът от функцията е самият поток и може да се провери състоянието му. Ако не бъдат прочетени никакви символи, операцията пропада (състоянието на потока не е добро).

```
char str[64];
inputFile.get(str, 64);
```

istream& getline(char * s, streamsize n, char delim);

Прочита редица от символи от потока и ги съхранява в символния низ `s`, до прочитането на (n – 1) символа или до достигането на разделителя `delim`. По подразбиране за разделител се използва символът за нов ред '\n'. Разделителят СЕ премахва от входния поток. Следващото четене ще започне от него. Терминиращата нула '\0' се добавя автоматично в края на низа `s`. Резултатът от функцията е самият поток и може да се провери състоянието му. Ако символният низ `s` се напълни преди достигане до разделителя `delim`, операцията пропада (състоянието на потока не е добро).

```
char str[64];
inputFile.getline(str, 64);
```

Състояния на потока

В зависимост от това дали последната операция над потока е била успешна, поток може да бъде в 4 основни състояния, които се описват с отделни битове.

good

Потокът е в добро състояние, ако последната операция е била успешна. Това е най-общият флаг. За да е вдигнат, никой от останалите флагове не трябва да е вдигнат. Състоянието може да се провери с булевата функцията `good()`, приложена върху потока.

```
fileStream.good();
```

bad

Потокът е в състояние bad, ако последната операция не е успешна. Възникнала е критична грешка по време на работата с потока, от която потокът не може да се възстанови. Например при хардуерен проблем. Състоянието може да се провери с булевата функцията `bad()`, приложена върху потока. Когато потокът е в състояние bad, той не е в състояние good, но състоянието bad не е противоположно на състоянието good.

```
fileStream.bad();
```

fail

Потокът е в състояние `fail`, ако последната операция не е успешна. Когато потокът е в състояние `fail`, той не е в състояние good. Пример за това е грешен формат на данните. Например очакваме да се прочете число, но от текущата позиция не може да се разпознае такова. С определени действия, състоянието на потока може да бъде възстановено и работата с него да продължи. Състоянието може да се провери с булевата функцията `fail()`, приложена върху потока. Обикновено този флаг се вдига когато е вдигнат флага bad.

```
fileStream.fail();
```

eof

Флагът за край на файл `eof` (end of file) се вдига, ако е направен опит за четене *след последния* символ, а не при прочитането на последния символ. Заедно с него се вдига и флага `fail`, защото се търсят някакви данни, но такива не се откриват и операцията пропада. Състоянието може да се провери с булевата функцията `eof()`, приложена върху потока.

```
fileStream.eof();
```

Важно!

- Ако състоянието на потока не е добро, то остава такова, докато не бъде възстановено. За да се възстанови обратно в състояние good, може да се използва функцията `clear()`.

  ```
  fileStream.clear();
  ```

  Функцията сваля всички флагове за грешка и възстановява състояние good.

- Ако състоянието на потока не е добро, (почти) всички операции, които се прилагат върху него ще бъдат неуспешни. Програмата ще премине през указаните операции, но няма да произведе никакъв резултат.

- При обхождане на даден файл, най-добре е да не се използва проверката за достигане на край на файл като условие в цикъл. Така няма да се провери за състояния bad или `fail`, получено поради друга причина. Най-добре е в условието на цикъла да се проверява дали състоянието на потока е good. Също при получаване на състояние `fail`, ако за програмата има значение по каква причина то е възникнало, трябва да се изследват останалите състояния.

- Затварянето на файл *не изчиства* състоянието на потока. Ако файлът е изчетен докрай и трябва да се изчете повторно със същия поток, преди това трябва да се изчисти неговото състояние.

Други начини да се провери състоянието на потока

Потокът може да бъде преобразуван неявно до булева стойност. Стойност `true` е съответствие на добро състояние на потока.

Например следните проверки разчитат на подобно неявно преобразуване:

```
if (inputFile) ...
```

Проверява се дали потокът е в добро състояние. Еквивалентно на

```
if (inputFile.good()) ...
```

```
if (!inputFile) ... еквивалентно на if (!inputFile.good())
```

```
while (inputFile >> number)
```

Четенето от потока продължава, докато потокът е в добро състояние. Ако при приключване на цикъла е необходимо да се потвърди, че всички данни са изчетени, то трябва да се провери дали е вдигнат флагът за край на файл.