Лекция 6 (8. Ноември, 2011)

Връзки
http://www.cplusplus.com/doc/tutorial/files/
http://www.cplusplus.com/reference/clibrary/cstdio/fprintf/
http://www.cplusplus.com/reference/clibrary/cstdio/fscanf/
http://www.cplusplus.com/reference/clibrary/cstdio/freopen/
http://download.oracle.com/javase/1.5.0/docs/api/java/util/Scanner.html
http://download.oracle.com/javase/1.4.2/docs/api/java/io/BufferedReader.html
http://download.oracle.com/javase/1.4.2/docs/api/java/io/BufferedWriter.html

Първи Час
Понякога, когато пишем софтуер се налага да гоним скорост и да оптимизираме бързодействието на нашата програма с честни и нечестни методи. От друга страна често се налага да боравим с файлове (които могат да бъдат доста различни от текстов файл, песен, филмче или някой от другите масови видове файлове). Затова този час ще разгледаме как можем да оптимизираме четенето и писането от/във файлове (съответно от/в конзолата, която, всъщност, също е файл).
Интересното в случая е, че ако четенето или писането е критичната част от кода (bottle neck-а), подобряването може да доведе до ускорение в ПЪТИ на изпълнението на програмата. Ето едно примерно сравнение, в което ще изпечатаме 1 милион рандом double числа чрез C-style писане и после чрез C++ (потоково) писане.

```
#define NUM_DOUBLES 1048576
double array[NUM_DOUBLES];
void writingReadingTest()
{
	unsigned sTime;
	for (int i = 0; i < NUM_DOUBLES; i++)
		array[i] = (double)rand() / (double)RAND_MAX;

	sTime = clock();
	FILE *out1 = fopen("test1.out", "wt");
	for (int i = 0; i < NUM_DOUBLES; i++)
		fprintf(out1, "%lf\n", array[i]);
	fclose(out1);
	fprintf(stdout, "Execution time fprintf: %.3lf seconds.\n",
		(double)(clock() - sTime) / (double)CLOCKS_PER_SEC);

	sTime = clock();
	ofstream out2;
	out2.open("test2.out");
	for (int i = 0; i < NUM_DOUBLES; i++)
		out2 << array[i] << endl;
	out2.close();
	fprintf(stdout, "Execution time streams: %.3lf seconds.\n",
		(double)(clock() - sTime) / (double)CLOCKS_PER_SEC);
	system("pause");
}
```

Резултатът от изпълнението е:
Execution time fprintf: 1.406 seconds.
Execution time streams: 14.688 seconds.

Малко над 10 пъти! Интересно е, че ако печатаме char-ове, потоковото писане е малко по-бързо! Като цяло скоростта е доста непредвидима (но почти може да сте сигурни, че потоковото ще е по-бавно).
Аналогична програма на Java изкара времена, малко по-бавни от тези на C-style писането (1.64s срещу 2.58s, ако и в C варианта печатаме с 16 цифри след десетичната запетая).

```
public static int NUM_DOUBLES = 1048576;
public static void main(String[] args) throws IOException
{
	File file = new File("test.out");
	FileOutputStream fileOutputStream = new FileOutputStream(file);
	Writer out = new OutputStreamWriter(fileOutputStream);

	long sTime = System.currentTimeMillis();
	Random rand = new Random();
	for (int i = 0; i < NUM_DOUBLES; i++)
		out.write(rand.nextDouble() + "\n");
	long eTime = System.currentTimeMillis();
	out.close();
	System.out.println("Execution time in java: " +
		(eTime - sTime) / 1000.0);
}
```

Провеждайки същия тест, само че с четене (отново 1 милион doubles) даде следните резултати:

C: 2.359s
C++: 7.485s
Java (Scanner): 10.109s

Отново доминация на C, но този път най-зле представилият се е Java (Scanner е МНОГО бавен, ползвайте го само ако входът е относително малък).

Какво можем да направим за да подобрим тези резултати? Можем да използваме буфериране. Буферите по принцип са лесно-достъпна (и бърза) памет (примерно RAM или някакви хардуерни кешове), в която можем да записваме информацията, докато съберем достатъчно за да я пренесем на по-бавния носител (харддиск, CD, flash-ка, iPod и всъщност каквото ви хрумне) наведнъж. Например вместо да записваме на диска число по число 1 милион пъти, можем да записваме по 100 числа наведнъж. Така бихме направили само 10000 писания, вместо един милион. Но няма ли да е по-добре да са 1000 числа наведнъж, вместо 100? Или пък направо всичките 1 милион? Зависи. Като цяло най-добрият начин да се менажират буферите е не на брой елементи а на количество данни.

Както можете да се досетите 1 милион char-а са различни по обем от 1 милион double-а. Затова буферите е хубаво да се оразмеряват с някакъв брой байтове (примерно 10K или някакво друго число, което е избрано да е приблизително оптимално за целта (според хардуера или други съображения)). От друга страна пък ако потребителят иска да прочете/запише само няколко числа, а ние чакаме буферът да се напълни, то може програмата да „забие" в чакане на още данни. Затова повечето буфери са реализирани с таймер и ограничение по размер. Така ако пишем голямо количество данни и буферът се напълни, той трябва да се изпразни автоматично и да бъде готов да поеме следващите данни. Ако пък сме в другата крайност и записваната информация е сравнително малко, той трябва да се изпразва през някакво време независимо дали е пълен или не (примерно няколко стотин милисекунди). Също е хубаво буферите да се изпразват при по-специфични явления (примерно затваряне на файла, в който пишем или пък излизане от програмата, която го е ползвала).
Съвременните компютри ползват буфери на много места без това да се натрапва на потребителя. Примерно един нов харддиск има между 8 и 64 мегабайта кеш, който се ползва именно за буфериране. Без него операциите биха били дори по-бавни отколкото са сега (и то с много!). Същото важи за други записващи устройства като CD-ROM-а. Операционната система също се стреми да ползва буфери за каквото може, като своевременно забързва допълнително процеса, ползвайки ефективно RAM-а за буфериране (имаше една песен на колегите от пловдивското ФМИ "2 гигабайта, 2 гигабайта...").
Нека например разгледаме какво можем да постигнем, ако буферираме четенето в горния пример на Java.

```
Execution time in java (Scanner): 9.875s
Execution time in java (BufferedReader): 1.203s
```

Хоп! Постигнахме осем пъти подобрение. По-интересното е, че сега бием и C-style четенето. Но преди феновете на Java да са се въодушевили твърде много, трябва да споменем, че то е небуферирано (поне не от нас) - ако буферираме и него ще постигнем сходни или по-добри резултати.

Ето и Java варианта на четенето чрез Scanner и BufferedReader:

```
private void readingTest() throws NumberFormatException, IOException
{
	File file = new File("test.out");
	FileInputStream fileInputStream = new FileInputStream(file);
	Scanner in = new Scanner(fileInputStream);

	long sTime, eTime;
	double array[] = new double[NUM_DOUBLES];
	sTime = System.currentTimeMillis();
	for (int i = 0; i < NUM_DOUBLES; i++)
		array[i] = in.nextDouble();
	eTime = System.currentTimeMillis();
	in.close();
	System.out.println("Execution time in java (Scanner): " +
		(eTime - sTime) / 1000.0);
```

```
            sTime = System.currentTimeMillis();
            FileReader fileReader = new FileReader(file);
            BufferedReader br = new BufferedReader(fileReader);
            for (int i = 0; i < NUM_DOUBLES; i++)
                  array[i] = Double.parseDouble(br.readLine());
            eTime = System.currentTimeMillis();
            br.close();
            System.out.println("Execution time in java (BufferedReader): " +
                  (eTime - sTime) / 1000.0);
      }
```

Забележете, че при хубаво четене или писане на Java постигаме резултати малко по-бързи или малко по-бавни от C-style четенето и писането в C. Интересен парадокс тук е, че създателите на C++ са искали да направят потоците (iostream, fstream) по-бърз и по-удобен вариант на досегашния начин за четене и писане. И докато за по-удобен може и да са успели, то за по-бърз със сигурност са се провалили.

C-style четене и писане

Тъй като това не се преподава във ФМИ (или аз съм го изпуснал) ще напиша малко за него. Първо, то е архаично (съответно има някои странности). Второ, то е по-сложно от потоковото четене и писане. Трето, то е бързо.

Откъде да започнем? Първо, всичко относно C четенето и писането можете да намерите в хедъра cstdio (stdio.h). Кратко предупреждение: поради липса на референции в C, за подаване на аргументи се използват pointer-и. Това не бива да ви плаши, тъй като това вас, като потребители на функциите, почти не ви касае.

При всяко четене и писане (било то fscanf, fprintf, fgetc, fgets, fputc, fputs или техните алтернативи без f в началото, които го правят скрито) се ползва указател към структура FILE, която указва къде се пише или откъде се чете. Не бива да ви интересува какво точно се пази в тази структура – нека тя бъде просто една черна кутия, държаща достатъчно информация за даден файл. Има няколко default-но отворени такива файла: stdin, stdout, stderr – съответно стандартният вход (от конзолата), стандартният изход, и изходът за грешка. Те се отварят като се стартира програмата. Както казах на лекцията, можете спокойно да не ползвате printf, scanf и т.н., а техните алтернативи с f отпред – fprintf, fscanf и т.н. В случай, че искате да четете или пишете от конзолата, просто като файлов аргумент ползвате stdin, stdout или stderr.

Как можете да отворите файл? Сравнително лесно. Ако искате да ползвате някой от стандартно-отворените под друго име правите просто:

```
FILE* newName = stdin/stdout/stderr;
```

Ако пък желаете да отворите някой файл на диска се ползва:

```
FILE* fptr = fopen(const char* fileName, const char* mode);
```

fileName представлява името на файла (ако е в директорията, където е стартирано exe-то), или пълния път до файла.

mode указва за какво точно искаме да ползваме файла (четене, писане,...). В 99% от случаите ще ползвате някое от:

1. "r" (от "read"), което указва, че отваряте файла за четене
2. "w" (от "write"), което указва, че отваряте файла за писане
3. "a" (от "append), което указва, че отваряте файла за дописване (тоест не изтрива досегашното му съдържание, което би направило "w")

Можете да отворите файл за четене и писане едновременно, като добавите "+" след първата буква. Забележете, че "r+" е различно от "w+" – докато първото отваря файла за четене и писане, второто отново го отваря за четене и писане, но изтрива досегашното му съдържание (или го създава) преди да започне каквото и да било.
По подразбиране файлът се отваря за четене на текст. Има начин и да отворите файл за четене на binary информация. Това става с добавяне на втора буква "b" (от "binary") към mode. Така примерно "rb" би означавало, че отваряме двоичен файл за четене, а "wb" – че ще пишем директно двоична информация, а не текст. Може да срещнете и втора буква "t", която пък специфицира, че отваряте текстов файл, но тъй като това се прави по подразбиране много хора го пропускат.
Ако искате да затворите файл (за да сте сигурни, че съдържанието му е запазено, или за да го отворите малко по-късно) можете функцията fclose(FILE* file). Много често това се изпуска ако е в края на програмата, тъй като рутината по затварянето на дадена програма включва затваряне на всички отворени от нея файлове. Преди да се затвори пък даден файл се изпразват буферите му (за да е записана цялата информация).

След като сме отворили файл (или ползваме някой от стандартно-отворените) как можем да запишем или извлечем информация от там? Най-основните функции за четене и писане са fscanf и fprintf съответно.
Нека първо разгледаме четенето. Сигнатурата на функцията fscanf е:

```
int fscanf(FILE* file, const char* format, ... );
```

Многоточието в края означава произволен брой аргументи. Този брой аргументи може да е между нула и достатъчно много за всякакви практически нужди. В една правилно-написана програма броят аргументи зависи от аргумента format, но синтаксисът на езика позволява това да е нарушено (тоест да искаме да прочетем 3 аргумента, но да подадем повече или по-малко). Функцията чете информация от подадения stream, като записва част от данните в 3тия, 4тия и т.н. аргумент. Колко точно аргумента и какви ще са те функцията разбира от format. За да стане това трябва да имаме някакъв начин да означаваме информацията, която искаме да запишем в променливи. За целта са въведени специални символи, започващи с "%" (примерно "%d", "%s", "%lf"…). Те указват какво точно се очаква да бъде прочетено и в каква променлива да бъде запазено (тъй като функцията няма лесен начин да знае какви аргументи точно да очаква се подават void*-ри, които после се convert-ват към съответните типове (даже това не е много наложително да се прави, тъй като е достатъчно да се знае адреса на променливата и колко байта е тя). И тъй като има разлика дали четем char, double или string, съответните специални символи разграничават променливите по типове, не само по размер в байтове.
Ето и специалните символни последователности (спецификатори) за различните типове:

Спецификатор	Тип
%c	char
%d	int
%f	float
%s	string
%u	unsigned

Различни от десетична бройна система	
%o	octal (число в осмична бр. система)
%x	hexadecimal (число в шестнадесетична)

Също така ако искаме да прочетем числа в различни от стандартните по дължина думи (примерно 4 байта за int) можем да ползваме следните модификатори:

Модификатор + Спецификатор	Тип
%hd	short
%ld	long
%lld	long long
%lf	double
%Lf	long double

Така, например, нека разгледаме програма, която чете първо име, второ име, години и среден успех от FILE* in:

```
char firstName[32], lastName[32];
int age;
double gpa;
fscanf(in, "%s %s %d %lf", firstName, lastName, &age, &gpa);
```

Както казахме, аргументите от трети нататък са УКАЗАТЕЛИ, затова пред age и gpa слагаме амперсанд (той взима адреса на променлива, което е точно нещото, което ни трябва). Защо нямаме амперсанд и пред firstName и lastName тогава? Защото те са масиви от char-ове, а всеки масив всъщност е pointer към данни. Така подавайки само името на масива всъщност подаваме точно pointer, което трябваше и да направим.

Можем да четем и произволни символи навсякъде между аргументите. Нека например имаме на вход изречението "The student Alexander Georgiev is 23 years old and has GPA 5.50.". C-style четенето ни разрешава форматиране на входа и изхода, като горното изречение можехме да parse-нем по следния начин:

```
fscanf(in, "The student %s %s is %d years old and has GPA %lf",
      firstName, lastName, &age, &gpa);
```

или

```
fscanf(in, "%*s %*s %s %s %*s %d %*s %*s %*s %*s %*s %lf",
      firstName, lastName, &age, &gpa);
```

Можем да ползваме "%*" за да четем, но не записваме даден тип.

Да обобщим трудностите при C-style четенето. Важните неща са две – първо, кои модификатори и спецификатори за кои типове са. Второ – аргументите винаги са pointer-и, а не променливи! Второто е една от най-честите грешки, които неопитните програмисти правят при ползването му.

Писането чрез fprintf е доста подобно – аргументите са същите, както и спецификаторите. Единствената разлика е, че тук се подават стойностите на променливите, а не указатели към тях. С други думи – никъде не ползвате амперсанд. Примерно (при някакъв FILE* out, отворен за писане):

```
fprintf(out, "The student %s %s is %d years old and has GPA %lf.\n",
      firstName, lastName, age, gpa);
```

или

```
fprintf(out, "The student %s %s is %d years old and has GPA %lf.\n",
      "Alexander", "Georgiev", 23, 5.5);
```

Би изпечатало точно горния пример (но с повече символи след десетичната запетая).

Веднъж свикнали с горното форматиране, понякога то е по-удобно за печатане на смислен текст, в който има много променливи. Примерно как би изглеждал горният пример с потоково писане?

```
cout << "The student " << firstName << " " << lastName << " is "
     << age << " years old and has GPA " << gpa << "." << endl;
```

Не особено по-красиво; освен това трябва да внимавате къде трябва да има шпации и къде не. Разбира се, не можем да твърдим, че едното е по-удобно или другото – всичко е до навик на програмиста и лични предпочитания.

Накратко ще споменем също, че имаме и няколко начина да форматираме цели и дробни числа. Ако по някаква причина искаме да изпечатаме едно цяло число с ТОЧНО определен брой символа (примерно искаме да изпечатаме някаква таблица така, че да е подравнена), това лесно може да стане. "%0nd" печата int с точно n цифри, като допълва числото с водещи нули, ако то е по-късо. Примерно:

```
fprintf(out, "%05d %05d", 42, 1337);
```

би изпечатало „00042 01337". Ако пък не искаме водещи нули, а просто празно пространство (в случая шпации) можем да ползваме:

```
fprintf(out, "%5d %5d", 42, 1337);
```

което, от своя страна, би довело до изпечатването на „ 42 1337".
Нека като един по-ясен пример искаме да изпечатаме таблица (от произволни стойности) с numRows реда и numCols колони, подравнени в 8 позиции (без допълнителни шпации между тях). Следният цикъл би свършил тази работа за нас:

```
for (int row = 0; row < numRows; row++)
{
      for (int col = 0; col < numCols; col++)
      {
            fprintf(out, "%8d", rand());
      }
      fprintf(out, "\n");
}
```

Тестово не открих да можем да ползваме други place holder-и освен шпации или водещи нули.

Също така съществува подобно лесно форматиране на изхода на нецели числа. Често от нас се иска да ги изведем с определен брой знаци след десетичната запетая. По подразбиране този брой е 6. Тоест:

```
fprintf(out, "%lf", 42.0);
```

би изпечатало "42.000000".

Лесно можем да го модифицираме (примерно ако искаме да печатаме числата с точност от 3 знака след десетичната запетая):

```
fprintf(out, "%.3lf", 42.0);
```

печата „42.000".

Интересно е какво се случва, когато даваме точност по-малка от броя значещи цифри в числото. Тогава числото автоматично се закръгля до такова с определения брой знаци (като цифри 5 и нагоре се закръглят към горното, докато 4 и надолу към долното – тоест нормалното закръгляне). Така изразът:

```
fprintf(out, "%.3lf", 1337.56273);
```

би довел до изпечатването на „1337.563", а

```
fprintf(out, "%.2lf", 1337.56273);
```

води до „1337.56".

Тук набързо да вметна един трик – ако искате да изпечатате double променлива като integer число, може да ползвате „%.0lf", което би изпечатало числото, закръглено до най-близкото цяло такова без десетична точка.
Това в най-общи линии е за форматирането на ints & doubles.

Малко допълнителна (но що-годе важна) информация. Първо, какво, аджеба, връщат тези функции? И двете връщат int, който отразява или колко неща са били прочетени/записани успешно, или отрицателна стойност при грешка (всъщност връща константата EOF (End Of File), но тя е винаги отрицателна). В случая на fscanf върната стойност значи броя на успешно прочетени аргумента (който, ако всичко е наред, трябва да съвпада с броя спецификатори (съответно допълнителни аргументи)). fprintf пък връща броя успешно записани символа (тоест колко нови char-а са се появили в изходния файл). Ето два примера:

```
fscanf(in, "%d %lf", &integerNum, &floatNum);
```

при успешно четене би върнало 2, докато

```
fprintf(out, “%d %lf”, integerNum, floatNum);
```

би върнало броя цифри на integerNum + 1 (за шпацията) + броя цифри преди десетичната точка на floatNum + 1 (за десетичната точка) + 6 за цифрите след десетичната точка (които, както казахме, по подразбиране са 6).
Второто допълнително нещо – ако ползвате тези функции под, да кажем, Visual Studio, то ще ви вади едни warning-и, които казват, че тези функции са deprecated. Какво означава това? Това означава, че се считат за несигурни и е възможно да бъдат премахнати в бъдещи версии (е, едва ли ще бъдат премахнати, но пък че са несигурни – така си е). Какво имам предвид под несигурни? Ами има много начини, по които биха могли да бъдат счупени. Нека да вземем първия пример, който дадохме за fscanf.

```
char firstName[32], lastName[32];
int age;
double gpa;
fscanf(in, “%s %s %d %lf”, firstName, lastName, &age, &gpa);
```

Какво става ако някой въведе име, което е дълго, примерно, 100 символа? Еми всички символи след 32-рия ще бъдат записани в памет, която не сме алокирали (съответно има шанс да не е наша). Така програмата ни ще гръмне по един не особено хубав начин (runtime error на spoj0, за хората, които са ходили на ПрАнКА или Състезателно Програмиране). Има и други security проблеми, които не се handle-ват по особено добър начин в тези функции, затова и съответно въпросните warning-и в студиото.

А няма ли начин да прочетем цял ред, независимо какво съдържа той и колко е дълъг? (Няма ли друг живот?) Има разбира се! Функцията, която ви трябва, е fgets(). Нея можем да ползваме по следния начин:

```
fgets(char* destination, int maxNumberOfReadCharacters, FILE* source);
```

Тук destination е някакъв char масив, в който искаме да запишем реда. Вторият аргумент ни указва колко най-много символа да прочетем дори все още да не сме стигнали до края на реда (което ни предпазва от горния пример да излезем извън масива си). Третият аргумент е файлът, от който четем. Малка забележка – тъй като четем целия ред, последният символ преди терминиращата нула ще е знакът за нов ред. Това е що-годе сходно с BufferedReader в Java, където можем да четем именно ред по ред. Точно с fgets бихме реализирали буферирано четене в C/C++.

Още нещо – има ли начин да проверим дали имаме още нещо за четене (тоест дали файлът е свършил)? И това има. Този път функцията е feof(FILE* stream). Така нека например на интервю ни дадат следната задача: изпечатайте в конзолата всеки четен ред, записан във файла „evenLines.txt”, който се намира в директорията на изпълнимия ви файл. Програмата, която се иска от вас да напишете е следната (след като ги питате колко най-дълъг може да бъде един ред

и те, да кажем, ви отвърнат 80 символа, и също така дали индексирането започва от 0 или 1, и те ви отвърнат от 0):

```
FILE* in = fopen(“evenLines.txt”, “rt”);
char buff[128];
bool even = true;
while (!feof(in))
{
      fgets(buff, 127, in);
      if (even)
            fprintf(stdout, “%s”, buff);
      even = !even;
}
```

Накрая нещо, което не показахме на лекции, но набързо ще споменем тук. Това е пренасочването на входа и изхода в C/C++. Какво означава пренасочване? Еми примерно сме написали програма, която чете от файл, но изведнъж искаме да направим всички четения от стандартния вход (конзолата) или пък обратно (вместо от конзолата искаме да четем/пишем във файл). За това има 2 сравнително прости варианта (отново наследени от C).

Вариант 1:

```
freopen(const char* fileName, const char* mode, FILE* stream);
```

Тази функция започва ползването на файл с име fileName вместо досегашния файл stream. Така например

```
freopen(“inputFile.txt”, “rt”, stdin);
```

би направил всички следващи четения от конзолата да бъдат прочетени от файла inputFile.txt. Аналогично

```
freopen(“outputFile.txt”, “wt”, stdout);
```

би пренасочило всички писания в конзолата към файла outputFile.txt.

Вариант 2:

Като алтернатива можем да ползваме fprintf/fscanf и просто да сменим файловия указател. Примерно:

```
FILE* out = stdout;
fprintf(out, “Hello, console!\n”);
out = fopen(“textFile.txt”, “wt”);
fprintf(out, “Hello, file!\n”);
```

би изпечатало първото съобщение в конзолата (тъй като първият аргумент на fprintf е out, което в началото сме set-нали да бъде stdout), докато второто писане би било във файла textFile.txt, тъй като сме сменили към какво точно сочи out.