Kolegové a kolegyně,

kontaktní výuka je sice zakázána, nás však nic nezastaví. Předpokládám, že budu postupovat podle slidů dostupných na
`http://kam.mff.cuni.cz/~perm/programovani/NNPRG031/` – tam jsem zatím vyvěsil loňské slidy. Ačkoliv letošní slidy se mohou lišit, nepředpokládám, že rozdíly budou zásadní. Jako změny lze očekávat například mírné omezení grafových algoritmů a rozšíření softwarově-inženýrských položek (jako objektový návrh či verzovací systémy). Slidy budou upravovány průběžně (nejpozději v termínu, kdy by se konala příslušná přednáška). Nicméně potrvá-li současné omezení déle, přeskupím si práci a slidy zkusím přepsat co nejdříve. Paralelně se bude objevovat tento materiál. Tak tedy poznáte, kde již výklad měněn nebude. Máte-li dotazy, stále platí, že je třeba se ptát. Na dotazy plánuju odpovídat, ale ne nutně každý den. Občas se může stát, že vaše dotazy budou muset počkat za záležitostí s vyšší prioritou. Například již nyní tři maily čekají na odpověď, až sepíšu tento materiál a rozešlu.

Pracovat můžete pravidelně, jako bychom pracovali, kdyby výuka nebyla přerušena, nebo si studium můžete přeskupit dle libosti. Nicméně cvičení poběží korespondenčním způsobem. Dokud se budou cvičit základy programování v C#, není s tím zásadní problém (dotazy budete moci – či dokonce muset – vznášet mailem). Horší situace nastane, až přijde čas cvičení objektových návrhů. Tam je bezlatenční komunikační systém velmi užitečný a ještě nevím, jak se s problémem případně vypořádáme (nějaké nápady už ale mám, takže v případě nouze bychom měli být schopni vydržet téměř celý semestr).

Zde budou především komentáře ke slidům, které bych komentoval ústně. Bez těchto slidů tedy tento materiál nemusí dávat smysl.

Dosti organizačních záležitostí, pusťme se do práce.

1 17. března – placeholdery, streamy, 1. část diskrétní simulace

1.1 Placeholdery

Dosud jste se dozvěděli, že v C# vypisujeme výstup pomocí metod `Write` a `WriteLine`, které najdeme ve třídě `Console` umístěné v namespacu `System`. Všimli jste si, že tyto funkce zpracují celkem jakýkoliv typ (díky přetěžování funkcí), ale reagují vždy jen na jeden argument (ačkoliv se vám je možná podařilo zavolat s více argumenty). Chcete-li vypisovat výstup rozmístěný v několika proměnných, můžete použít tzv. placeholdery. Tento výraz společnost Mircosoft překládá jako *složené formátování.*

K dalším argumentům odkážeme tak, že první argument (funkcí `Write` a `WriteLine`) je tzv. `formátovací string.` Objeví-li se v něm složené závorky, v nich se očekává popis, co se sem má umístit. Nám zcela postačí informace, že tento string začíná číslem, které indexuje argumenty následující za tímto stringem, a to podobně jako při práci s polem. Tedy například takto:

```
int a=Convert.ToInt32(Console.ReadLine()),
int b=Convert.ToInt32(Console.ReadLine());
Console.WriteLine("{0} + {1} = {2}",a,b,a+b);
```

Výhodou může také být, že k jednomu argumentu můžeme přistoupit opakovaně:

```
Console.WriteLine("{0} + {1} = {2}, kdezto {0} * {1} = {3}.",a,b,a+b,a*b);
```

Další informace (pro naše účely až příliš podrobné) najdete například na:
`https://docs.microsoft.com/cs-cz/dotnet/standard/base-types/composite-formatting`

1.2 Streamy

Tato část výkladu se nachází ve slidech k 9. březnu (kdy jsme postupovali o trochu pomaleji).

Práce se soubory je při programování nutností. Zatím umíme pracovat se standardním vstupem a výstupem, což děláme voláním metod Read, Readline, Write, WriteLine. Jak uvidíme, se soubory se pracuje velmi podobně. Budeme totiž volat stejné metody. Jen nebudeme pracovat se standardním vstupem, ale se souborem, který pro tyto účely otevřeme. Soubor otevřený pro čteni (ve formátu proudu dat, tedy streamu) je v C# reprezentován objektem třídy `StreamReader`, obdobně soubor pro zápis je reprezentován objektem třídy `StreamWriter`. Nejjednodušší způsob otevření souboru je zavolání konstruktoru s jedním stringovým parametrem označujícím jméno souboru, se kterým budeme pracovat:

```
System.IO.StreamReader r= new System.IO.StreamReader(@"c:\temp\file.txt");
```

Takto otevřeme StreamReader. Vidíme, že je též v namespacu `System`, ovšem tentokrát v jeho podnamespacu `IO`. Máme-li tedy na začátku programu `using System.IO;` postačí napsat jen:

```
StreamReader r=new StreamReader(@"c:\temp\file.txt");
```

Povšimněte si znaku "obchodní a"(jinak zvaný zavináč) na začátku stringu. Copak tam asi dělá? Napřed jen napovím, vše prozradím až koncem odstavce: Co znamená `\r`, `\n`, `\t`?
Chceme-li říci, že řetězec neobsahuje řídicí sekvence, dáme před něj právě znak obchodního a (tím řekneme, že uvnitř se nic neinterpretuje). Alternativou by bylo řetězec přepsat takto: `"c:\\temp\\file.txt"`. To by fungovalo také.

Stream zavřeme metodou `Close`. Ke čtení a zápisu používáme kromě metod `Read, Readline, Write, Writeline` ještě `ReadToEnd`, která přečte celý stream až do konce. Chceme-li se zeptat (u streamu pro čtení), zda jsme na konci souboru, využijeme `EndOfStream`. Kontrolní otázka: Je to atribut, nebo wrapper? (Nápověda: Podívejte se do Studia.)

Přečtěte si ve slidech, co je možné se soubory dělat (kódování češtiny a další parametry konstruktoru).

1.3 Diskrétní simulace

Simulovat lze leccos. Nás budou zajímat optimalizační problémy, které jsou dostatečně náročné, aby bylo těžké si je představit. Ačkoliv cílem tak je optimali-

zace, nám v tuto chvíli postačí prostředí, které umí dotyčný systém simulovat (za daných podmínek).

Simulace se dělí na spojitou (vedoucí obvykle k soustavě diferenciálních rovnic) a diskrétní. Úlohou spojité simulace může být například šíření infekce, která nám výuku přerušila (kdy nás zajímá počet nakažených v jednotlivých okamžicích), typičtější úlohou spojité simulace může být šíření ethanolu v organismu po rychlém požití několika pumprlíků. Typickou úlohou diskrétní simulace (na naší Fakultě) jsou auta s pískem:

Máme stavbu, k jejímuž provedení potřebujeme písek. Máme několik nákladních automobilů, kterými lze písek přepravovat. Dále máme hromadu písku vzdáleného od staveniště. U hromady i na staveništi jsou pracovníci s lopatami (a pochopitelně i rouškami), kteří nakládají, resp. vykládají automobily. Jak máme práci zorganizovat, abychom písek převezli co nejdříve, potažmo co nejlevněji? Pro zjednodušení můžeme předpokládat, že všichni zúčastnění půjdou domů, až bude písek odvezen a do té doby jim za přítomnost na pracovišti platíme.

Úlohy spojité simulace obvykle vedou na systém diferenciálních rovnic, který neumíme vyřešit, nebo umíme dokázat jeho neřešitelnost. Úlohy spojité simulace můžeme zkusit diskretizovat - například zmíněné šíření naší oblíbené infekce můžeme diskretizovat tak, že budeme simulovat jednotlivé lidi pobíhající po Republice. Když se setká zdravý s nemocným, s danou pravděpodobností se infekce přenese, s další pravděpodobností se infekce zaktivuje a po pár hodinách začne šířit (stejným způsobem na všechny, které potká). Problém ovšem je popsat, jak se chová jeden každý obyvatel - třeba i menšího města. Úloha diskrétní simulace by nás čekala ještě při každém jednotlivém přenosu (který vůbec není triviální). Úloha by tedy byla diskretizovatelná, ale bylo by to zapotřebí udělat jiným způsobem. Náš model (byť nerealizovatelný) by měl výhodu, že ukáže, kdo patrně onemocní (a kdo ne).

My si na diskrétní simulaci demonstrujeme `objektový návrh`. Objektový návrh odkazuje k tomu, jak bychom problém vyřešili, kdybychom na to měli čas. Obecně v objektovém návrhu jde o to problém dekomponovat do tříd a objektů tak, aby spolu související agenda zůstala pospolu, ale aby se podle možností co nejvíce rozdělila (například proto, aby vedoucí týmu mohl práci rozdělit více nežli jednomu programátorovi). Při objektovém návrhu je třeba vhodně ukládat data (abychom je zbytečně nekompírovali, protože všichni víme, že kopírování je častým zdrojem chyb). Současně je ale potřebujeme na všech místech, kde se s nimi bude pracovat. Potřebujeme problém rozdělit na co nejmenší dílky (abychom mohli následně kód psát ve více lidech), ale současně potřebujeme, aby se celý systém podařilo složit dohromady (a ne aby se systém při pokusu o zprovoznění složil). U nás se obyčejně objektový návrh cvičí tak, že zkoušíme problém rozdělit na co nejvíce tříd a objektů tak, aby to ještě dávalo smysl. Ke cvičení totiž obvykle dostáváme jednodušší úlohy, protože na ty těžší bychom potřebovali několik týdnů (než bychom vůbec pochopili, o co v nich jde).

Úlohy vstupující do objektového návrhu jsou obvykle popsány poměrně vágně. Důvody jsou různé. Obvykle jde o náročný problém, který je problematické detailně popsat. Tyto detaily se navíc mohou začít měnit (v průběhu implementace). A tedy ani zadavatel nemá přesnou představu, co po nás vlastně

chce (to začne zjišťovat, až mu ukážeme, jak úlohu pochopili my). Objektový návrh by tedy měl být odolný vůči změnám různých parametrů. Simulujeme-li tedy šíření infekce, může se stát, že než systém dopíšeme, přijde jiná infekce (která se šíří trochu jinak) a my potřebujeme, aby bylo možné naše dílo použít. Taktéž se může stát, že nebudeme simulovat šíření viru, ale šíření ekonomické krize (která bude nejspíše následovat). Udělat ovšem objektový návrh pro simulaci infekce i hospodářské krize by už asi bylo výrazně náročnější. Společnými rysy však je, že bychom oboje nejspíše zkusili řešit diskrétní simulací.

Diskrétní simulace je zvláštní tím, že ačkoliv problém vypadá složitě, existuje poměrně snadný objektový návrh definující několik tříd, které jsou od sebe velmi slušně oddělené, aniž by to někomu vadilo. S konkrétním předmětem simulace má něco společného typicky jen jedna třída (ostatní na předmětu simulace nezávisejí). Typičtějšími úlohami diskrétní simulace (o kterých bude řeč) jsou (kromě aut s pískem) výtahy (a jízda jimi) nebo samoobsluha (a nákup v ní).

Třídy (a objekty), se kterými diskrétní simulace pracuje, jsou: *kalendář událostí, simulační jádro a třída obsahující rutiny pro obsluhu (tedy zpracování) konkrétních událostí*. Popsané je najdete ve slidech.

Kalendář událostí je datová struktura obsahující události tak, abychom byli schopni událost přidat, zrušit, přerozvrhnout (na jiný čas) a zjistit nejbližší událost (která nastane jako první). Simulační jádro vytáhne nejbližší událost (z kalendáře) a zajistí obsloužení (tedy vyvolá rutiny událost obsluhující, tyto rutiny mohou manipulovat s již rozvrženými událostmi - prostřednictvím simulačního jádra, které manipuluje s kalendářem).

Obvyklé řešení operuje s procesy (probíhajícími v simulaci). Každý proces je schopný určit ze současného stavu příští událost, kterou si rozvrhne v kalendáři. Tyto procesy je třeba vhodně navrhnout, ačkoliv v našich případech jsou očividné. U aut s pískem stačí, aby procesy odpovídaly autům. Auta buďto přejíždějí, nebo čekají (až budou naložené). Nic jiného se v simulaci neděje. U výtahů může být jedním procesem jeden cestující, dále bude jeden proces za každý výtah. Cestující buďto čekají na výtah, nebo jím jedou. Nic jiného se v simulaci neděje. Ovšem je třeba dát pozor na interakce mezi procesy. Jede-li výtah (a ví, kdy dojede do cíle), může se čas příjezdu změnit, pokud ho po odjezdu zastaví přišedší cestující v patře, kde dosud nikdo nebyl. Jeden proces tak může ovlivnit události jiného procesu (což je poměrně nepříjemné).

Každý proces je v nějakém stavu (jak uvádí slidy) a mezi těmito stavy různě přechází. Diskrétní simulace operuje jen s těmito změnami stavu. Probíhá-li proces, vlastně nás nezajímá. Pro nás je zajímavé, kdy se něco stane. Obsluhujeme-li tedy událost, z popisu (který si k ní musíme být schopni obstarat) zjistíme, jaká událost bude příští a kdy nastane. Tu si necháme rozvrhnout do kalendáře a až do této události pro simulaci přestáváme být zajímaví.

Simulační jádro tedy jednoduše bere události v pořadí, v jakém nastanou a obsluhuje. Simulace končí, je-li kalendář událostí prázdný. Na simulaci nás může zajímat plno věci, základní informací je čas konce (tedy kdy skončila obsluha poslední události).

Konkrétní situace: Auta s pískem se mohou sjet u zúžení vozovky (pro nás kritické sekce), kterou se smí projíždět jen jedním směrem. Jak simulovat

za těchto okolností? Je třeba čekání. Aktivní nebo pasívní. Při aktivním čekání se auto při každém výjezdu z kritické sekce podívá, zda je sekce volná v jeho směru. Při čekání pasívním se vyjíždějící auta podívají, zda je sekce volná (a pokud ano, probudí protijedoucí vozidla). Při paralelních výpočtech hrozí race condition (do kritické sekce vjede nepovolený počet procesů) nebo deadlock (alias vzájemné vyloučení, kdy na sebe procesy navzájem čekají způsobem, který neskončí). Podrobnosti budou nejspíše na Počítačových systémech (či Základech operačních systémů). Řešením je zamykání. Na to dojde řeč při povídání o vláknech.

Jak lze simulaci implementovat: Kalendář událostí může být obousměrný cyklický spojový seznam uspořádaný dle času události. Plánujeme procesy (o jakou událost jde, si pamatuje proces, kterého se událost týká), na kritické sekce se čeká ve frontách pasívně (poslední tedy odjíždějící probudí všechny čekající ve frontě - tedy rozvrhne jim vpád do kritické sekce teď). Pasívní čekání obyčejně méně zatěžuje procesor, občas způsobí uváznutí procesu (proces čeká, i když by nemusel), kdežto aktivní čekání připomíná Oslíka ze Shreka.

Popis události: Výčtovým datovým typem:
`enum nazev_typu{jedna_hodnota,druha_hodnota,treti_hodnota,pooddelovane_carkami, nekdy, muzeme, priradit, hodnotu=10, jindy,ne;}`
Příklad: `enum stav{stoji,jede,vyklada=5};`