




Software a zpracování dat ve fyzice částic I

... aneb úvod do světa Linuxu a vybraného software používaného ve fyzice vysokých energií pohledem mírně pokročilého uživatele.



(poslední úprava: 6. května 2020)

Neházejte rajčata po autorovi, fakt dělal co mohl.

Program (1)

- Přehled aplikací ve fyzice vysokých energií aneb s čím vším přijde člověk do styku 
- Základy Linuxu 
 - základní příkazy, shell, cykly, základní utility
 - jednoduché skripty; další utility (grep, sort, awk)
 - instalace Debian Linuxu (pro zájemce)
- Od Pascalu k C++: 
 - srovnání programovacích jazyků Pascal a C, ukazatele
 - použití debuggeru
 - pár slov o C++

Program (2)

- Zpracování dat (ROOT) 
 - základní datové struktury (grafy, histogramy, funkce)
 - fitování, jednoduchá makra, vlastní funkce
 - stromy, výběrová pravidla
- Sazba dokumentů v LaTeXu 
 - základní styly a prostředí
 - matematické vzorce, tabulky, obrázky
 - časté typografické prohřešky a některé finty

V dalším semestru navazuje druhá část o simulacích, sběru a zpracování dat
http://www-ucjf.troja.mff.cuni.cz/~davidek/vyuka/comphep_2.pdf

Připojení k serveru (1)

- Pracujeme na serveru **ipnp21**, připojení pomocí **ssh** s možností posílat "další okna" (X11 forwarding):
 - z libovolného Unixu příkazem:

```
ssh -X [username@]ipnp21.troja.mff.cuni.cz
```
 - z Windows je několik možností:
 - **PuTTY + X-Win32/Exceed v pasívním módu**
 - stažení libovolného ssh-klienta (např. PuTTY) nejlépe do domovského adresáře (v PUČ je to H:\). Konfigurace (verze 0.59):
 - Session: hostname: ipnp21.troja.mff.cuni.cz
 - Session: connection type: SSH
 - Connection → SSH → Preferred SSH Protocol version: 2
 - Connection → SSH → X11: Enable X11 forwarding
 - spuštění programu X-Win32/Exceed v pasívním módu (stačí např. spustit Start → All programs → X-Win32 8.0 → X-Win32)
 - při prvním otevíraném okně povolit přístup vždy
 - další okno: nové PuTTY nebo spustit ze stávajícího (**xterm &**, **emacs &**, ...)

Připojení k serveru (2)

- **X-Win32 v "aktivním" módu:**

- nepotřebujeme PuTTY, zato musíme konfigurovat X-Win32 spuštěním X-Win32 8.0 → X-Config → Relace → Ručně:

- způsob připojení: StarNetSSH
- název relace: ipnp21
- hostitel: ipnp21.troja.mff.cuni.cz
- příkaz: xterm

Nezadávejte přihlašovací jméno a heslo, konfigurace se totiž v PUČ ukládá na daném počítači !!!

- po spuštění se pak otevře xterm, z něhož lze pouštět libovolné aplikace (`xterm &`, `emacs &`, ...)

- **instalace Cygwin:**

- pak ale musíme použít

`ssh -Y [username@]ipnp21.troja.mff.cuni.cz (trusted X11)`

Kapitola 1: Stručný přehled software používaného ve fyzice částic

... aneb s čím vším se můžeme v životě setkat
a nemělo by nás to rozházet

Operační systémy

Upozornění: jde o částečně subjektivní pohled !!

- V podstatě 2 základní typy: Unix a Windows (Mac je v zásadě Unix)
 - Windows se používají spíš jako přátelské rozhraní (čtení mailů, multimédia, příprava prezentací...), někde i na sběr a analýzu dat (ATLAS SCT)
 - Unix (tj. Linux, SunOS,) se používá na serverech, které slouží jako mailservery, výpočetní farmy (MC simulace, analýza dat), často i sběr dat. Speciální konfigurace mohou sloužit i jako síťové prvky (gateway, router, DNS)
- Výhody a nevýhody: záleží na jedinci...., ale myslím že bez Unixu to nejde.

Programovací jazyky

- Fortran, C, C++
 - Fortran je starý, je pro něj napsána celá řada aplikací a užitečných knihoven. Je relativně jednoduchý (skoro jako Pascal), ale postrádá některé možnosti (rekurze, přímé ovládání hardware)
 - C je trochu složitější, ale jde tam všechno :-))).
 - C++ umožňuje moderní objektový přístup k řešení komplexních problémů. Tudy asi vede cesta do budoucna... ale je to dost složité.
- Dnešní stav: Fortran téměř mrtev, všude číhá C++

Generátory srážek částic

- Co dělají:
 - generují srážku částic (např. elektron-positron, proton-proton,), vznikne spousta částic (procesy QED, QCD a různé modely hadronizace).
 - Parametry vzniklých částic (kinematické proměnné, náboj, ...) včetně "rodokmenu" se zapíše do nějakého seznamu.
 - Seznam částic slouží nejčastěji jako vstup do programů, které simulují průchod částic detektorem.
- Existuje jich celá řada, nejznámější jsou [Pythia](#) a [Herwig](#). Pro modelování některých speciálních procesů existují speciální generátory.....

Simulace průchodu částic detektorem (1)

- Co dělají:
 - jako vstup berou buď určitou částici (studium chování detektoru např. na testovacím svazku) nebo výstup z generátorů srážek částic (studium odezvy detektoru pro zvolené procesy)
 - vstupní částice jsou "prohnány detektorem", kde se simuluje průchod částic hmotným prostředím (ionizace, brzdné, přechodové a Čerenkovo záření, elektromagnetické a hadronové spršky, průchod magn. polem...)
 - výstupem je simulace odezvy v jednotlivých typech a částech detektoru.

Simulace průchodu částic detektorem (2)

- Dva základní přístupy:
 - plná simulace: simulují se jednotlivé procesy do detailů.
Příklad: [Geant3](#) (Fortran), [Geant4](#) (C++)
 - rychlá simulace: odezvy v jednotlivých částech detektoru jsou parametrizovány podle známých/očekávaných vlastností (kalibrace, rozlišení), proto jsou šité na tělo danému experimentu. Příklad: [ATLFAST](#) pro ATLAS

Analýza dat (1)

- 2 nejrozšířenější aplikace:
 - **Paw** (Physics Analysis Workstation), založeno na Fortran/C.
 - vlastní makrojazyk COMIS, podobný Fortranu
 - podpora pro uživatelské funkce psané ve Fortranu či C
 - jednodušší syntaxe, jednodušší pro začátečníky (několik příkazů stačí)
 - **ROOT** - důsledně založený na C++:
 - makrojazyk je čisté C++, stejné zacházení v interaktivní i batch-verzi.
 - objektově-orientovaný, modulární, široké možnosti (např. řízení sběru a online zobrazení dat)
 - relativně nový, stále se vyvíjí
 - lze importovat datové objekty z Paw (histogramy, ntuply)

Analýza dat (2)

- Použitelnost:
 - obě aplikace (Paw, ROOT) fungují pod všemi OS
 - současný stav: Paw téměř mrtvé, všichni používají ROOT (a proto se budeme zabývat ROOTem)
 - existují/existovaly i další aplikace (např. JAS), ale moc se neprosadily.....

Kapitola 2: Operační systém Linux



...aneb expertem (relativně 😊)
snadno a rychle

Základy Linuxu (1)

- Trocha historie:
 - operační systém Unix vyvíjen od roku 1969. Základní myšlenka – jádro oddělené od interpreteru příkazů.
<http://www.root.cz/clanky/historie-os-unix/>
 - kvůli implementaci byl vyvinut programovací jazyk C
 - autorem Linuxu je Linus Torvalds, který napsal první jádro (1991)
- Vlastnosti Linuxu:
 - současně může běžet mnoho programů (multitasking)
 - na jednom stroji může pracovat současně více uživatelů
 - OS funguje na různých platformách a mnoha procesorech

Základy Linuxu (2)

- Základní příkazy:
 - některé stejné jako v DOS/Win:
 - `cd` – změna adresáře
 - `pwd` – zjištění současného adresáře
 - `mkdir` – vytvoření adresáře
 - `echo` – výpis parametru (řetězec, proměnná) na obrazovku
 - jiné se trochu liší:
 - `ls` – výpis souborů/podadresářů (**dir** v DOS)
 - `rm` – smazání souboru/adresáře (**del** v DOS)
 - `cp` – kopie souboru/adresáře (**copy** v DOS)
 - `man` – jednoduchý manuál k danému příkazu (**help** v DOS)

Pozor na rozdíl mezi malými/velkými písmeny !

Základy Linuxu (3)

- Použití 'wildcards':
 - stejné jako v jiných operačních systémech
 - příklady: `ls *.ps`; `ls -rtl obr?.ps`
- 'Hidden files': začínají tečkou, zobrazí se pouze při výpisu pomocí: `ls -a`
- Shell – interpret příkazů:
 - mnoho shellů (bash, zsh, ksh, csh, tcsh), liší se v detailech
 - (ne)interaktivní shell, login shell
 - startovací skripty (nastavení některých proměnných, aliasů, provedení některých úkonů)

- nastavení proměnných obecně (bash, zsh):
 - `export CISLO=195 ; export ADR=/home/davidek/bin`
 - `export PATH="$PATH:$HOME/bin"`

Platí pro dané "okno" a jeho dceřinné procesy. Bez příkazu `export` se nastavení dále nedědí.
 - další operace: `echo $CISLO ; unset ADR`
- významné proměnné nastavené shellem:
 - SHELL login shell
 - PATH prohledávaná cesta ke spustitelným souborům
 - HOME domovský adresář
 - PWD aktuální adresář
 - USER jméno uživatele
 - PS1 základní prompt

Proměnné v shellu (2)

- Některé proměnné se nastavují automaticky. Kde ?
 - bash (v uvedeném pořadí):
 - interaktivní login shell: `/etc/profile`, `~/.bash_profile`, `~/.bash_login`, `~/.profile`
 - interaktivní non-login shell: `/etc/bash.bashrc`, `~/.bashrc`
 - zsh (v uvedeném pořadí, pokud odpovídá typ shellu):
 - všechny typy: `/etc/zsh/zshenv`, `~/.zshenv`
 - login shell: `/etc/zsh/zprofile`, `~/.zprofile`
 - interaktivní shell: `/etc/zsh/zshrc`, `~/.zshrc`
 - login shell: `/etc/zsh/zlogin`, `~/.zlogin`
- Změna login-shellu: `chsh`
 - účinnost od dalšího nalogování
 - pozor na správnou cestu k shellu!! (viz též `/etc/shells`)

Aliases

- Nastavení užitečných zkratek. Obvykle nastavujeme permanentně ve startovacích skriptech pro příslušný interaktivní shell:
 - `alias ls='ls --color=auto -F -T 0'`
 - `alias la='ls -la'`
 - `alias rm='rm -i'`
 - Někdy se mohou hodit tzv. globální aliasy (alias se expanduje uvnitř příkazu):
 - `alias -g lxplus='lxplus.cern.ch' # pouze zsh`

Linux pro mírně pokročilé (1)

- Vstup a výstup: existují 3 základní zařízení:
 - standardní vstup (stdin; default: klávesnice)
 - standardní výstup (stdout; default: terminál na obrazovce)
 - standardní výstup chyb (stderr; default: terminál)
- Přesměrování vstupu a výstupu:
 - vstup ze souboru: `muj_program < muj.vstup`
 - výstup do souboru: `ls -l /etc > etc.vypis`
 - výstup stdout i stderr do souboru:
`muj_program > muj.vystup 2>&1`

Linux pro mírně pokročilé (2)

- Linux pracuje se zařízením stejně jako se souborem:
 - výstup na zařízení: `ls -l /etc > /dev/null`
- Příkaz "pipe" (|): výstup z posledního příkazu je zpracován jako vstup příkazu následujícího:
 - `ls -l $HOME | less`
 - `cat muj.soubor | grep "Slovo" | less`

Lze řetězit prakticky donekonečna.
- Symbolický link - místo kopie přímo odkaz na původní soubor/adresář:
 - `ln -s /etc/profile hlavni.profil`

Linux pro mírně pokročilé (3)

- Vlastnosti souborů/adresářů:

- určují přístupová práva pro vlastníka, skupinu a ostatní

```
drwxr-xr-x  3 davidek users      4096 Jan 21 14:37 atmel/
```

```
-rw-r--r--  1 davidek users    239190 Jan 22 12:51 nc21.select
```

- mění se příkazem `chmod`:

- `chmod g+w,o-r nc21.select` (případně: `chmod 660 nc21.select`)

- `chmod o-x atmel` (případně: `chmod 754 atmel`)

- `chmod -R +X *` (rekurzivně, dodá právo +x jen pro adresáře)

- Vlastník souborů/adresářů:

- změna se provádí příkazem `chown`

- `chown pribyll:users nc21.select`

- `chown -R pribyll:users atmel` (rekurzivně, tj. včetně podadresářů)

Měnit vlastníka může jen root, uživatel může měnit jen skupinu.

Linux pro mírně pokročilé (4)

- Kompresní utility:
 - `gzip` (*.gz), `bzip2` (*.bz2), `compress` (*.Z)
 - opačný krok: `gunzip`, `bunzip2`, `uncompress`
- Archivace adresářů (případně s kompresí): `tar`
 - vytvoření archívu: `tar -cvf /tmp/home.tar $HOME/*`
(`tar -cvzf /tmp/home.tgz $HOME/*`)
 - zjištění obsahu archívu: `tar -tvf /tmp/home.tar`
 - rozbalení archívu: `cd / ; tar -xvf /tmp/home.tar`

Zachovávají se i symbolické linky !

Linux pro mírně pokročilé (5)

- Příkaz "cyklu" `for-done` - užitečný při práci se skupinami souborů:
 - `for i in *.ps ; do echo $i ; done`
 - `for i in {11..25} ; do rm -f test${i}.ps ; done # pouze zsh`
 - `for i in `seq 11 25` ; do rm -f test${i}.ps ; done # bash`

či seznamy:

- `export SEZNAM=`cat muj_seznam.txt` ;
for i in $SEZNAM ; do cp $i /scratch/davidek/ ; done`

Vytvoření seznamu je obecným modelem pro naplnění proměnné hodnotou (seznamem hodnot) získanou jako výstup nějakého příkazu. Pozor na "zpětné apostrofy".

Linux pro mírně pokročilé (6)

- Podmínečný příkaz: `if-then-elif-else-fi`
 - `if [...] ; then echo "Is true"; else echo "Is false" ; fi`
 - různé formy testované podmínky:
 - porovnání řetězců: `if ["$PWD" = "/home/davidek/test"]`
 - numerické relace: `if [$CISLO -lt 10]`
 - existence souborů: `if [-f /etc/passwd]` (podobně též adresářů či symbolických linků)
 - Co dělá následující příkaz ?

```
export first=$HOME/.startup;  
if [ "$SHELL" = "/bin/bash" ]; then cp /etc/profile $first;  
elif [ "$SHELL" = "/usr/bin/zsh" ]; then cp /etc/zprofile \  
$first; else echo "Non-standard shell !" ; fi
```

Linux pro mírně pokročilé (7)

- Co kde najdeme, aneb základní adresářová struktura:
 - **/** – celý svět
 - **/etc** – všechny konfigurace
 - **/etc/init.d** – startovací skripty, spouštějí se při bootu
 - **/home** – domovské adresáře
 - **/boot** – jádro, soubory potřebné k naboťování
 - **/var** – často měnící se obsah (např. logy)
 - **/usr** – obsah, který se moc nemění (např. programy a dokumentace)
 - **/usr/local** – lokální soubory pro daný stroj
 - **/tmp**, **/var/tmp** – dočasné soubory

Linux pro mírně pokročilé (8)

- Užitečné klávesové zkratky:
 - **Ctrl-r** (^r): zpětné hledání příkazu/výrazu (xterm, emacs, ...)
 - **Ctrl-s** (^s): dopředné hledání příkazu/výrazu (xterm, emacs, ...)
 - **Ctrl-g** (^g): konec (nejen hledacího) režimu
 - **Ctrl-l** (^l): obnovení (překreslení) obrazovky
 - **Ctrl-q** (^q): dojde-li k zablokování xterm ... (některé shelly, resp nastavení, nemají rády **Ctrl-s**)

Linux pro mírně pokročilé (9)

- Editory v Linuxu – dvě základní kategorie:

1. liga: vim, emacs

- umí úplně všechno (zvýraznění syntaxe, podpora pro různé programovací jazyky, klávesová makra, sloupcové operace, ...) ☺
- mají varianty v textovém i grafickém režimu ☺
- ne zcela intuitivní ovládání ☹ (složitější více-hmaty, v grafickém režimu lze ale většinu funkcí ovládat myší přes menu)

2. liga: např. pico, mcedit, jed, ...

- (relativně) intuitivní ovládání, jednoduché i pro uživatele Windows ☺
- nemají širokou podporu pro různé jazyky, pro některé však mají alespoň zvýrazněnou syntaxi (vyjma pico) ☹ Pouze v textovém režimu.
- Nefunguje-li **Fx**, pomůže **Esc-x** (např. mcedit v Putty). Označení bloku: Shift+myš

Existuje celá řada dalších editorů (kate, gedit, nedit, ...), ale určitě se vyplatí naučit se jeden z dvojice prvoligových.

Pár praktických informací (1)

- Přenos dat mezi různými stroji:
 - Unix – Unix:
 - pomocí tzv. secure copy (**scp**)
`scp local_file username@remote_machine:path_from_home_dir`
`scp username@remote_machine:path_from_home_dir local_file`
Existují i grafické aplikace (**secpanel**), stejnou službu umí ale i Midnight Commander
 - prostou kopií (**cp**), pokud stroje sdílejí disky např. přes NFS, AFS
 - Windows – Unix:
 - pomocí **WinScp**, volně k dispozici na internetu
 - Unix – Windows:
 - přístup teoreticky možný přes sambu (**smb**), ovšem sdílení musí být povoleno na vzdáleném Windows-serveru

Pár praktických informací (2)

- Správa adresářů a souborů pomocí Midnight Commander (**mc**)
 - chová se stejně jako známý Norton Commander a jeho klony
 - umožňuje i přesuny souborů mezi různými stroji, např:
 - `cd /#sh:davidek@ipnp10/home/davidek/public`
 - `cd /#ftp:ftp2.debian.cz/debian`
- Kvóty: celková velikost souborů v \$HOME je obvykle omezena
 - 2 typy: "soft" (jednorázově lze překročit na několik dní) × "hard"
 - po vypršení časového limitu po překročení soft kvóty se nelze nalogovat !
 - co dělat, jsem-li na limitu:
 - promazat Cache od různých programů (např. `$HOME/.mozilla/*/*/Cache`)
 - smazat nepotřebné soubory, např. nepoužívané zkompileované programy, od nichž mám zdrojové kódy
 - uplatit administrátora, aby zvýšil kvótu ☺

Jednoduché shell-skripty (1)

- Prakticky: jen posloupnost příkazů, lze volat s parametry:

```
#!/bin/bash
```

```
if [ -f $1 ]; then
```

```
    gzip $1
```

```
else
```

```
    echo "Soubor neexistuje."
```

```
    exit 1
```

```
fi
```

```
exit 0
```

- specialita: jméno shellu, který příkazy bude interpretovat
- \$1 je první parametr, např. jméno souboru
- návratová hodnota v případě neúspěchu
- návratová hodnota v případě úspěchu

Jednoduché shell-skripty (2)

- Přístup k parametrům:
 - \$0 jméno volaného skriptu (tj. jméno souboru)
 - \$1,\$2 první, druhý předaný parametr při volání
 - \$* všechny parametry ("\$*"="\$1 \$2 \$3...")
 - @\$ všechny parametry ("\$@"="\$1" "\$2" "\$3"...)
- Přístup k dalším užitečným informacím:
 - \$# počet předaných parametrů
 - \$\$ číslo procesu, pod kterým se shell-skript spustil
 - \$? návratová hodnota předchozího příkazu (nula znamená bezchybné provedení)
- Příkaz **shift**: maže původní obsah parametru \$1, ostatní o jeden dolů posune (\$0 zůstává, \$# se příslušně změní)

Jednoduché shell-skripty (3)

- S tím už se dají dělat slušná kouzla... viz příklady
- Úkol: napište shell-skript **kopie**, který zkopíruje zadaný soubor do zadaného adresáře:
 - volání: **kopie soubor adresář**
 - testuje existenci souboru i adresáře, v případě neexistence skončí a vrátí hodnotu 1
 - testuje úspěšnost kopírování a v případě neúspěchu vrátí hodnotu 2
 - pokud vše proběhne v pořádku, vrátí hodnotu 0

Další utility (1)

Následující utility zpracovávají vstup po celých řádcích:

- **grep** - vyhledávání určitého řetězce v souborech:
 - `grep -i "hledany vyraz" muj_soubor.txt`
 - `grep -v "vyskyt" muj_soubor.txt`
 - `grep -il "195\.113" /etc/*`
 - umí i tzv. regulární výrazy (viz literatura uvedená na konci kapitoly)
- **sort** – setřídění podle zadaného "sloupce" (oddělené mezerou). Numericky nebo po znacích:
 - `sort -n -k 2 muj_soubor.txt`
 - `sort -r muj_soubor.txt`

Další utility (2)

- Mocná zbraň **awk**: umožňuje sofistikované zpracování textu v závislosti na obsahu jednotlivých "polí" na řádce. Důsledně používá syntaxi C:
 - tisk prvního a třetího pole (sloupce):
 - `awk '{print $1, $3}' muj_soubor.txt`
 - `awk '{printf("%d, %6.4f\n",$1,$3)}' muj_soubor.txt`
 - výběr řádku s daným obsahem v určitém poli:
`awk '{if ($3 == "vyraz") {print $0}}' muj_soubor.txt`
 - některé akce lze požadovat jen na začátku (tj. před zpracováním první řádky), resp. na konci: BEGIN, END. Např. sečtení prvků prvního sloupce:
`awk 'BEGIN{s=0}; {s+=$1}; END{printf("Soucet %d\n",s)}' \muj_soubor.txt`
 - řídicí konstrukce můžeme zapsat i do souboru (`awk -f ...`)

O čem jsme se ještě nezmnili.....

- Na Unix-serverech lze spouštět i časově náročné úlohy, přičemž po spuštění se uživatel může odpojit a sledovat stav běhu programu při každém dalším zalogování
 - **nohup** (nechá program běžet i po odlogování uživatele)
 - `nohup muj_program > vystup 2>&1 &`
 - ingredience: spuštění na pozadí (&), přesměrování výstupu
 - **screen**:
 - běžné spuštění programu, nemusím ani přesměrovat výstup (výhoda "okamžitého zápisu" oproti přesměrování)
 - k dispozici na většině serverů včetně ipnp21
 - podrobnosti např. na <http://www-ucjf.troja.mff.cuni.cz/~davidek/linux1/>
 - **fronty**:
 - systém umožňuje počítání úloh na volných procesorech, ostatní úlohy mezitím čekají ve frontách
 - k dispozici na velkých serverech, kde je velké množství uživatelů

Literatura pro další vzdělávání

- O Linuxu existuje mnoho zajímavých knih a materiálů na Webu. V češtině jsou k dispozici například:
 - M.G.Sobell: *Linux – praktický průvodce*
 - kolektiv autorů: *Linux – dokumentační projekt*
 - E.Siever: *Linux v kostce*
 - http://atrey.karlin.mff.cuni.cz/~johanka/vyuka/pohadky_unix.html
- Mnoho užitečných informací a rad najdete také na serverech <http://www.linux.cz> a <http://www.root.cz/>

Příklad na závěr

- Napište složený příkaz či shell-skript, který provede následující:
 - do pomocného adresáře rozbalí archiv
`~davidek/prednasky/comphep/shell/big.tgz`
 - sečte délku v blocích všech souborů a vypíše ji spolu s původní délkou archívu
 - v každém souboru typu `*.dat` spočítá aritmetický průměr z čísel uvedených v prvním sloupci a vypíše je (soubor, průměr)
 - uklidí po sobě, tj. smaže pomocný adresář i s jeho obsahem

Nápověda: lze použít např. příkazy `mkdir`, `cd`, `tar`, `du -s`, `du -a`, `echo`, `grep`, `for-done`, `awk`, `rm`

Teď jste již pokročilí uživatelé Linuxu !

Instalace Debian Linuxu

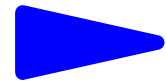
- Proč právě Debian? A proč ne :-))) (no dobře, protože já ho používám). Nicméně rozdíly mezi jednotlivými distribucemi spočívají v detailech....
- Jednoduchá konfigurace: stroj trvale připojen k síti, žádná speciální zařízení (tiskárna, CDW, zip-drive, modem,)
- Instalace rozumného minima software, kompilace vlastního jádra (stihneme-li)
- Dokumentace: existuje celá řada HOWTOs, např.:
 - <http://www-ucjf.troja.mff.cuni.cz/~davidek/linux1/>
 - <http://www.togaware.com/linux/survivor/>
- Obecně platí: **If all has failed, read the instruction !**

Kapitola 3: Pár slov o C/C++

Stručné srovnání Pascalu a C,
ukazatele. Triviální program
v C, použití debuggeru.

Základní principy v C++,
jednoduchý příklad

Několik poznámek o Fortranu a
srovnání s C pro zájemce v příloze.



Pascal vs. C (1)

- Pascal i C jsou standardní programovací jazyky, oba mají své objektově-orientované "pokračovatele" (Delphi, C++)
- Některé vlastnosti jsou stejné či velmi podobné:
 - základní struktura: hierarchické uspořádání (hlavní program, doprovázen dalšími procedurami a/nebo funkcemi)
 - základní datové typy stejné (liší se jen v označení), existují globální i lokální proměnné
 - přiřazení hodnot, podmínky a typy cyklů prakticky stejné, mírně se liší v syntaxi
 - možnost rekurze
 - vstupní a výstupní operace (otevření/zavření souboru, čtení/zápis do souboru či na standardní vstup/výstup) podobné, liší se jen v syntaxi. C má širší možnosti.

Pascal vs. C (2)

- volný formát zdrojového textu (lze více příkazů na řádek, každý příkaz končí středníkem)
- složené příkazy se uzavírají do skupin
 - Pascal: `begin ... end;`
 - C: `{ ... }`
- Samozřejmě existují rozdíly (neúplný výčet):
 - Pascal nerozlišuje malá/velká písmena, C ano !!
 - index polí: v Pascalu běží (obvykle) od 1, v C od nuly !!
 - předávání proměnných do procedur/funkcí:
 - Pascal: hodnotou (`i: integer`) i odkazem (`var i: integer`)
 - C: pouze hodnotou (`int i`), předávání odkazem se dělá přes ukazatele (pointery)
 - časté použití ukazatelů v C, aritmetika ukazatelů (C)

Pascal vs. C (3)

Základní struktura programu

- Hlavní program je definován na začátku souboru

```
program tilerow(input,output)
```

- Procedures a funkce jsou definovány níže

- Hlavní program je vždy funkce main, vrací celé číslo

```
int main()
```

- Definice ostatních funkcí – 2 možnosti:
 - před hlavním programem
 - za hlavním programem:
 - musím ale definovat na začátku funkční prototypy, aby překladač mohl provést kontrolu typů

Pascal vs. C (4)

```

program tilerow(input,output);
var .... ; {def globálních proměnných}
begin
....   {vlastní programový kód}
end.

```

```

procedure segment(i: integer);
var ...; {def lokálních proměnných}
begin
...     {vlastní programový kód}
end;

```

```

#include <stdio.h> //hlavičkový soubor
....           // def globálních proměnných,
               // je-li opravdu potřeba
void segment(int i); // def úplného
                    // funkčního prototypu
int main() {
... // def lokálních proměnných + kód
return 0; // návratová hodnota
}

```

```

void segment(int i) {
... // def lokálních proměnných + kód
}

```

Pascal vs. C (5)

- Typy základních proměnných:

```
var i: integer; a: real;
```

```
int i; float a; double b;
```

- Podmínka, přiřazení hodnoty:

```
if i = 1 then
begin
  writeln(i); i:=i+2;
end;
```

```
if (i == 1) {      Pozor, 2x "=" !!
  printf("%d\n",i);
  i=i+2;          Zde není znak ":" !!
}
```

- Některé C-operátory v Pascalu nejsou:

```
i:=i+5;
```

```
i+=5;
```

Podobně existují v C i další operátory: -= *= /= %=

```
i:=i+1;
```

```
i++; // post-inkrement
```

```
i:=i+1; j:=i;
```

```
j=++i; // pre-inkrement
```

Podobně existuje v C i post/pre-dekrement: i--; --i;

Pascal vs. C (6)

- Definice pole, naplnění, for-cyklus:

```
var pole: array[1..10] of real;
for i:=1 to 10 do
  pole[i]:=sqrt(a+1);
```

```
float pole[10];
for(i=0; i<10; ++i)
  pole[i]=sqrt(a+1);
```

První prvek má index nula !!

Pozor na přetečení indexu pole !!

- Vícerozměrná pole:

```
var pole2d: array[1..10] of
  array[1..5] of real;
pole2d[i,j]:= 5.5;
```

```
float pole2d[10][5];
pole2d[i][j]=5.5;
pole2d[i,j] = 3;
```

Častá chyba, čárka je operátor a znamená něco jiného !!

Pascal vs. C (7)

- I/O operace:

- společné definice:

```
var f: text;
```

```
var i:integer; x:real;
```

```
#include <stdio.h>
```

```
FILE *f;
```

```
int i; float x;
```

- čtení ze souboru:

```
assign(f,'jmeno.dat'); reset(f);
```

```
read(f,i,x);
```

```
close(f);
```

```
f=fopen("jmeno.dat","r");
```

```
fscanf(f,"%d %f",&i,&x);
```

```
fclose(f);
```

referenční operátor ("volání odkazem" přes ukazatele)

```
f=fopen("jmeno.dat","w");
```

```
fprintf(f,"%3d %6.1f\n",i,x);
```

```
fclose(f);
```

- zápis do souboru:

```
rewrite(f);
```

```
writeln(f,i:3,x:6:1);
```

```
close(f);
```


Ukazatele v C (1)

- Ukazatel obsahuje adresu paměti, kde je uložena hodnota dané proměnné. Jednoduchý příklad:

```
float *p; // ukazatel na typ float, zatím nikam neukazuje
```

```
float r=3; // reálná proměnná, inicializovaná na hodnotu 3
```

```
*p=5; // NELZE, neboť p zatím nikam neukazuje
```

```
p=&r; // p ukazuje na proměnnou r (obsahuje její adresu)
```

```
printf("%6.2f %6.2f\n",r,*p);
```

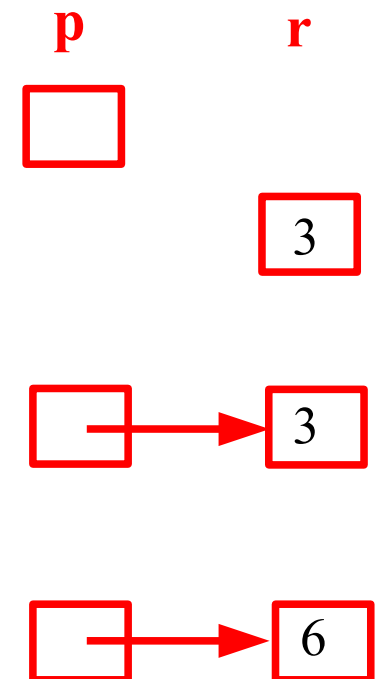
```
*p=6; // nyní již mohu přiřadit
```

```
printf("%6.2f %6.2f\n",r,*p);
```

– co dostanu na výstupu ?

3.00 3.00

6.00 6.00



Ukazatele v C (2)

- Díky ukazatelům lze tedy efektivně zařídit "volání odkazem" i v C. Příklad:

```
float r1=3, r2=5.5;
```

```
prehod(&r1,&r2); // předávám hodnotou adresy r1,r2
```

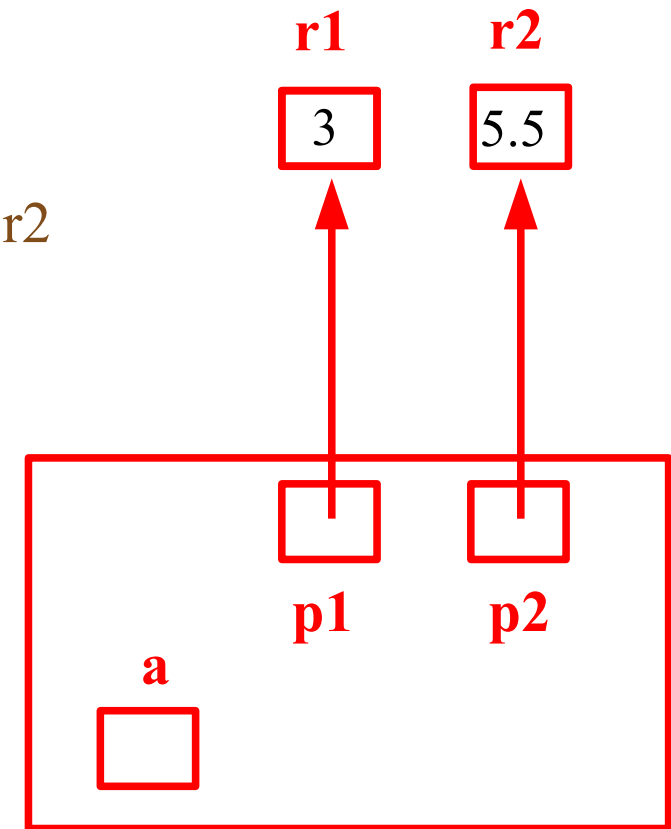
```
printf("Obsah r1 = %6.2f, r2 = %6.2f\n",r1,r2);
```

```
void prehod(float *p1, float *p2) {
```

```
    float a;
```

```
    a=*p1; *p1=*p2; *p2=a;
```

```
}
```



- na stejném principu fungují např. funkce `scanf()`, `fscanf()`, ...
- Pokud bych potřeboval předat vlastní ukazatel odkazem, musím použít konstrukci ukazatel na ukazatel na daný typ (`float **p`)

Ukazatele v C (3)

- Ukazatele a pole:
 - jméno pole je současně ukazatel na jeho první prvek ($p \equiv \&p[0]$)
 - ukazatelová aritmetika:
 - uvažme `double *pt; double p[10]`
 - necht' `pt=p`; potom `*(pt+5)` je ekvivalentní výrazu `p[5]`. Vtip je v tom, že prvky pole jsou v paměti uloženy za sebou

```
double p[10];
```

```
napln_pole(p);
```

```
....
```

```
void napln_pole(double *pt) {  
    *pt=5.1; // ekvivalentní pt[0]=5.1  
    *(pt+3)=7; // ekvivalentní pt[3]=7  
}
```

Pozor na přetečení rozsahu pole,
došlo by k nekontrolovanému
chování programu !!

Překladač to nehlídá !!

Příklad v C

- Napište program `prumery.c`, který:
 - v hlavním programu založí dvě pole `x[20],y[20]`
 - naplní tato pole čísla ze souboru `~davidek/prednasky/comphep/cpp/xy.dat`. Čtení proved'te v proceduře `void cti(float *x, float *y, int *n)`, která vrací mj. i počet údajů `n` ve zmíněném souboru ($n \leq 20$)
 - spočítá aritmetický průměr z hodnot v každém poli `x,y`. Průměry počítejte ve funkci `float prumer(float *x, int n)`, která vrací právě výsledný aritmetický průměr.
 - vytiskne výsledné průměry pro pole `x` a `y`
- Překlad programu:
 - pomocí skriptu `~davidek/bin/cecko prumery.c`
 - vytvoří se program `prumery`

Debugger (1)

- Debugger je nástroj na ladění programů. Pod Linuxem je k dispozici debugger `gdb` s grafickou nadstavbou `ddd`:
 - trasování běhu programu zdrojovým kódem
 - zobrazení obsahu proměnných v každé fázi programu
 - nastavení **break-pointů** (pozastavení na určité řádce) a **watch-pointů** (pozastavení běhu programu, změnil-li se obsah udané proměnné)
 - jak začít ? Jednoduše: `ddd muj_program`
- Spuštění uvnitř debuggeru:
 - nastavení break-pointu na začátek (první příkaz programu)
 - C kód: `break main`
 - Fortran kód: `break MAIN__`
 - start laděného programu: `run`

Debugger (2)

- Trasování běhu programu:
 - krok o 1 příkaz: **step** (krok s ponořením do podprogramů), **next** (krok bez ponoření do nižší úrovně)
 - více kroků: **c** (continue – až do dalšího break-pointu či watch-pointu), také možnost pokračování až do určitého řádku (výběr v menu)
- Zobrazení proměnných (včetně polí): **print promenna** nebo umístěním kurzoru myši na jméno proměnné
 - problém s proměnnými z common-bloku (Fortran), řešení viz <http://www-ucjf.troja.mff.cuni.cz/~davidek/linux1/>
- **Zkusme:** ladění programu **prumery**, zastavení běhu po dokončení čtení ze vstupního souboru.

Objekty a C++ (1)

- Principy objektově-orientovaného programování:
 - základními prvky jsou objekty:
 - odpovídají "reálným" objektům světa naší aplikace (např. konto, sklad, vozidlo)
 - obsahují vlastní data a interface (tzv. metody, tj. procedury a funkce) na zacházení s nimi. Vlastní datové prvky obvykle nejsou přímo přístupné (zapouzdření)
 - objekty spolu vzájemně spolupracují, tj. volají své služby jiných objektů
 - možnost konstrukce odvozených objektů (dědičnost)

neptej se nejdříve co má program udělat, ale s čím to má udělat

Objekty a C++ (2)

- Objekty obsahují vždy – kromě datových prvků a metod přístupu k nim – dvě speciální metody:
 - konstruktor
 - vytváří instanci daného objektu, inicializuje proměnné
 - má stejné jméno jako objekt (`void Object(...)`)
 - může jich být více, liší se počtem či typem parametrů (přetížené funkce)
 - volá se automaticky při vytvoření instance objektu (`new Object`)
 - destruktork:
 - "uklízí po sobě", např. uvolňuje paměť dynamicky přidělenou datovým prvkům dané instanci objektu
 - stejné jméno jako objekt, ale s vlnkou (`void ~Object()`)
 - právě jeden, bez parametrů
 - volá se automaticky při destrukci instance objektu (`delete Object`)

Jednoduchý příklad v C++ (1)

- Napišme program, který umožňuje základní aritmetické operace s komplexními čísly:
 - objekt Complex:
 - data: obsahují vlastní reálnou a imaginární část
 - metody:
 - nastavení reálné a imaginární části
 - zjištění reálné a imaginární části, absolutní hodnota, ...
 - definice operátorů (např. přiřazení, součet, ...), což zjednodušuje zápis programu

Jednoduchý příklad v C++ (2)

```

class Complex {
private:
    double reart, impart;
public:
    Complex();
    Complex(const double a, const double b);
    ~Complex();
    void Set(const double a, const double b=0);
    double GetRe() const;
    double GetIm() const;
    double GetAbs() const;
    Complex& operator=(const Complex &c);
    Complex operator+(const Complex &c);
};

```

- definice třídy Complex
 - datové prvky, přístupné jen pomocí metod tohoto objektu
 - veřejná část: metody
 - 2 přetížené konstruktory, volá se jeden či druhý podle počtu a typu parametrů
 - destruktork
- ostatní metody na zacházení s datovými prvky
- definice operátorů pro snadnější zápis standardních operací (pro pokročilé)

Jednoduchý příklad v C++ (3)

```
Complex::Complex() {  
    Set(0,0);  
}
```

```
Complex::Complex(const double a, const  
    double b) {  
    Set(a,b);  
}
```

```
Complex::~~Complex() {}
```

- implementace konstruktorů:
 - inicializace proměnných, případně vytvoření dynamických proměnných
 - nastavení Re a Im části
- destruktory:
 - zde je prázdný, není co dělat
 - typická akce: uvolnění dynamické paměti alokované v konstruktoru

```
int main() {
```

```
    Complex *a = new Complex(1,5);
```

```
    Complex *b = new Complex(3,2);
```

```
    Complex c;
```

```
    c = *a + *b;
```

```
    std::cout << "Abs(c) = " <<  
        c.GetAbs() << std::endl;
```

```
    delete a; delete b;
```

```
    return 0;
```

```
}
```

- příklad použití v hlavním programu
 - založení dynamických objektů typu Complex a naplnění
 - new volá konstruktor dané třídy, vrací ukazatel na nově vytvořenou instanci
 - objekty jsou v dynamické části paměti
 - statický objekt typu Complex:
 - založení, ale neplní se žádné hodnoty
 - přiřazení součtu výše uvedených objektů, využití přetíženého operátoru
 - tisk absolutní hodnoty
 - uvolnění dynamické paměti
 - návratová hodnota hlavního programu

Kompletní příklad včetně implementace všech zmíněných funkcí a operátorů viz [~davidek/prednasky/comphep/cpp/complex.C](https://github.com/davidek/prednasky/comphep/cpp/complex.C)

Další výdobytky C++

- C++ implementuje standardní volání odkazem tak jak ho známe z Pascalu:

```
prehod(x,y);  
....  
void prehod(float& a, float& b) {  
    float x = a; a = b; b = x;  
}
```

- Proměnné lze definovat i "uprostřed funkcí" či bloků, platnost takové proměnné je ale omezena na daný blok

```
for(int i=0; i<10; ++i) {  
    if (i < 5) { pole[i] = 2*i+1; } else { ... }  
}  
  
// zde již není proměnná i definována
```



Literatura (1)

- Dostupné v češtině:
 - P.Herout: *Učebnice jazyka C*, Kopp 2005 (ISBN 80-7232-220-6)
 - výklad programovacích postupů, časté srovnání s Pascallem, řešené příklady
 - 2 díly, 1.díl je určitě důležitější
 - J.Liberty: *Naučte se C++ za 21 dní*, Computer Press 2002 (ISBN 80-7226-774-4)
 - S.Racek, M.Kvoch: *Třídy a objekty v C++*, Kopp 1995 (ISBN 80-7232-017-3)

Literatura (2)

- Dostupné na síti:
 - J.Soulie: *C++ language tutorial*,
<http://www.cplusplus.com/doc/tutorial/>
 - B.Eckel: *Thinking in C++*
 - HTML verze: <http://jamesthornton.com/eckel/>
 - PDF verze:
<http://www.planetpdf.com/developer/article.asp?ContentID=6634>

Kapitola 4: Zpracování dat

- základy **Paw** aneb nebojte se staré pracky 
 - poměrně jednoduchý, ale starší program
 - založený na Fortranu, má interface do C
- základy **ROOTu**: 
 - moderní objektově-orientovaný nástroj na analýzu dat
 - založen na C++, syntaxe a makra důsledně C++
 - oproti Paw složitější, ale jde v něm udělat (skoro) všechno

**ALMOST
DEAD !!**

Úvod do ROOTu (1)

- Mocný nástroj nejen na analýzu dat, založený na C++, objektově orientovaný
- Vlastní interpret příkazů **CINT** (čisté C++), uživatelská makra a funkce také v C++
- Základní datové struktury:
 - grafy, funkce, histrogramy
 - ntuply, "zobecněné ntuply" (tree)
- Dokumentace:
 - manuály, příklady, FAQ i zdrojové kódy najdeme na adrese <http://root.cern.ch>
 - známe-li trochu Paw, pomůže nám "konverzní tabulka příkazů" <http://couet.home.cern.ch/couet/root/ht05.html>

Úvod do ROOTu (2)

- **Pozor:** většina funkcí má více variant použití (různý počet parametrů různých typů), jde o tzv. **přetížené funkce** (C++)
- Základní informace:

- spuštění: `root` (`root -l` vynechá zobrazení úvodního loga)
- příkazy lze psát buď přímo do příkazové řádky, nebo do tzv. makra. Příklad `test1.C`:

```
void test1() {  
    ....  
}
```


- spuštění maker uvnitř ROOTu: `.x test1.C`
- konverze existujících dat z Paw do formátu ROOT (spec. program, spouští se přímo v shellu):

`h2root file.hbook [file.root]`


Grafy

- Grafy se používají k zobrazení množiny dvojic $[x,y]$ (případně s chybami $[ex,ey]$), těmito body lze prokládat různé křivky (tzv. fity)
 - načtení hodnot do obyčejných jednorozměrných polí x,y,ex,ey
 - vytvoření a nakreslení grafu:


```
TGraphErrors *gr = new TGraphErrors(nfit,x,y,ex,ey) //nfit=počet bodů
gr->Draw("AP") //A=nakreslí osy, P=polymarker pro každý bod
```
 - vlastní fit:
 - k dispozici je sada předdefinovaných funkcí (exponenciála, Gauss, polynom N-tého stupně), např.:


```
gr->Fit("expo") ; gr->Fit("gaus"); gr->Fit("pol2")
```
 - fitovat lze i vlastní funkcí, v omezeném rozsahu atd. Více o fitech viz strana 71 

Funkce

- Funkce slouží k zobrazení určité závislosti a také k fitování (proložení určité parametrické funkce body grafu nebo spektrem histogramu)
- Mohou být 1-dim (TF1) nebo 2-dimenzionální (TF2)
- Příklady použití:
 - `TF1 *fun=new TF1("fun","sin(x)/x",-1,1) // definice funkce`
 - `fun->Draw() // zobrazení funkce`
 - fitování viz strana 71 

Histogramy (1)

- Histogram je jednoduchý datový útvar, zobrazující např. tvar nějakého spektra. Typy histogramů:
 - 1-dim (**TH1F**, **TH1D**), 2-dimenzionální (**TH2F**, **TH2D**)
 - profilové (**TProfile**): pro všechny případy patřící k dané hodnotě na ose X (tj. do daného binu) spočítá na ose Y průměr příslušných hodnot a jejich rozptyl (RMS)
- Základní úkony:
 - `TH1F *his=new TH1F("his","Muj histogram",100,0,1) //`
založení histogramu, 100 binů na ose X v rozsahu 0-1
 - `his->Fill(0.75) //` vložení jedné hodnoty
 - `his->Draw() //` zobrazení histogramu
 - `his->Fit("pol1") //` fit polynomem 1.stupně (pol1)

Histogramy (2)

- Input/Output operace (týká se obecně všech objektů v ROOTu, nejen histogramů):
 - načtení histogramu ze souboru:
 - `TFile *f=new TFile("data.root","OLD")` // otevření, read-only mod
 - `f->cd()` // změna adresáře – nyní v hlavním stromu souboru
 - `f->ls()` // výpis objektů v daném adresáři souboru
 - `hist->Draw()` // zobrazení histogramu "hist"
 - uložení histogramu do souboru:
 - `TFile *f=new TFile("data.root","NEW")` // otevření nového souboru
 - `hist->Write()` // zapsání histogramu "hist" do otevřeného souboru
 - `f->Close()` // zavření souboru
 - poznámka: je-li otevřeno více souborů současně, musíme specifikovat odkud/kam objekt čteme/zapisujeme. Viz manuál.

Více o fitech (1)

- Graf/histogram můžeme fitovat:
 - předdefinovanými funkcemi (exponenciála, Gauss, polynom), např. `his->Fit("gaus")`
 - vlastními jednoduchými funkcemi:

```
TF1 *f1 =new TF1("f1","[0]+x*[1]") //definice funkce pro fit se dvěma  
parametry [0] a [1]
```

```
f1->SetParameters(-2,1) // nastavení počátečních parametrů [0]=-2 a  
[1]=1
```

```
his->Fit("f1") // vlastní fit funkcí f1
```

Pozor – alternativa nastavení parametrů:

```
f1->SetParameter(0,-2) // nastavení počáteční hodnoty parametru [0]=-2
```

```
f1->SetParameter(1,1) // nastavení počáteční hodnoty parametru [1]=1
```

Více o fitech (2)

- fit obecně složitými funkcemi:

```
Double_t myfit(Double_t *v, Double_t *par) { // definice funkce
    if (v[0] < par[0]) {
        return(0);
    } else {
        return(par[1]*sqrt(v[0]*v[0] - par[0]*par[0]));
    }
}

TF1 *fit=new TF1("fit",myfit,-5,5,2); // 2= počet parametrů
fit->SetParameters(10,2);          // nastavení počátečních parametrů
hist->Fit("fit");                  // vlastní fit funkcí myfit
```

Poznámka: funkce "myfit" může mít libovolný počet proměnných (v[0],v[1],....) i parametrů (par[0],par[1],...)

Více o fitech (3)

- Fit histogramu s omezením (odkud,kam)
 - předdefinovaná funkce: `his->Fit("gaus","R","",0.5,1.5)`
 - vlastní jednoduchá funkce:


```
TF1 *fitf=new TF1("fitf","[0]+[1]/x",0.5,1.5)
fitf->SetParameters(-2,1) // počáteční parametry
his->Fit("fitf","R")      // fit v uvedeném rozsahu
his->Fit("fitf","R","", -1,5) // fit v jiném rozsahu
```
 - pro obecně složité funkce se to dělá zcela analogicky
- Dále lze omezit rozsah hodnot pro jednotlivé parametry či hodnoty parametrů přímo zafixovat – podrobnosti viz manuál.

Více o fitech (4)

- Získání parametrů fitu:

- předdefinované funkce:

```
TF1 *gfit = his->GetFunction("gaus");  
Double_t gsig=gfit->GetParameter(2);
```

- vlastní funkce (jednoduchá i obecně složitá): funkce je již zavedená, proto stačí prosté

```
Double_t slope=fitf->GetParameter(1);
```

Další údaje viz např. <http://root.cern.ch/root/HowtoFit.html>

Statistika a výsledky fitů

- Oba druhy informací lze zobrazit spolu s daným histogramem či grafem (TGraph, TGraphErrors):
 - statistické informace o histogramu (počet případů, střední hodnota, RMS, ...):
 - zapnutí/vypnutí: `gStyle->SetOptStat(1/0)`
 - nastavení zobrazených položek: `gStyle->SetOptStat(1100111)`
 - výsledky fitu:
 - zapnutí/vypnutí: `gStyle->SetOptFit(1/0)`
 - nastavení zobrazených položek: `gStyle->SetOptFit(101)`
- Statistický "box" lze na grafu libovolně umístit pomocí myši
- Zmíněná nastavení jsou globální, tj. platí pro všechny histogramy a grafy. Lze udělat i jednotlivě (viz manuál)

Konečně pár příkladů (1)

- **Příklad 1:** fitování vektorů (Paw / ROOT):
 - načtěte vektory x, y ze souboru
`~davidek/prednasky/comphep/paw/p2fit.dat`
 - body $[x, y]$ proložte polynom 2. stupně. Předpokládejte, že všechny body $y[i]$ jsou zatíženy chybou 0.8
 - výsledek fitu uložte do vektoru `par` a vypište ho do souboru `out.dat`
- **Příklad 2:** fit histogramu uživatelskou funkcí (Paw / ROOT):
 - načtěte histogram č. 10 ze souboru
`~davidek/prednasky/comphep/FortranC/histc.hbook`
 - histogram nafitujte lineární závislostí typu $a + b * x$, kterou si ale z cvičných důvodů napište jako uživatelskou funkci (se dvěma parametry). Funkce buď ve Fortranu/C nebo v C++.


Konečně pár příkladů (2)

- **Příklad 3:** vytvoření vlastního histogramu, naplnění hodnotami a fit teoretickou závislostí (Paw / ROOT):
 - pomocí uživatelské funkce
 - Paw: `soucet.f` (Fortran), případně `soucet.c` (C)
 - ROOT: `soucet.C` (C++)
 - založte histogram v rozsahu (0; 2) a naplňte ho 10000 případy součtů dvou nezávislých náhodných čísel rovnoměrně rozdělených na intervalu (0; 1)
 - matematicky odvodte tvar spektra a nafitujte ho uživatelskou funkcí (se dvěma parametry: **střed rozdělení** a **výška píku**)
 - histogram uložte do souboru `soucet.hbook` / `soucet.root`

Ntuply (1)

- Jedná se o "multi-dimenzionální tabulky". Lze zkoumat závislost jakékoli proměnné na libovolné kombinaci ostatních proměnných – **to je pravá analýza dat !!**
- **ROOT-Ntuply** odpovídají **RWN v Paw** (tj. velká tabulka, kde sloupce jsou proměnné a řádky jednotlivé případy), zatímco **Trees** jsou obecné struktury obsahující libovolné objekty.
- Základní operace s ntuply:
 - založení: `TNtuple *nt = new TNtuple("nt","pulse","time:samp")`
 - naplnění jedné řádky: `nt->Fill(time,samp)`
 - načtení: `nt->ReadFile("soubor.dat") //soubor o 2
sloupcích čísel, funguje od verze 4.0`
 - výpis vlastností: `nt->Print()`

Ntuple (2)

- Zobrazení proměnných:
 - celé spektrum jedné proměnné: `nt->Draw("time")`
 - spektrum s výběrem: `nt->Draw("samp","time>10")`
 - závislost 2 proměnných: `nt->Draw("samp:time")`
 - profil 2 proměnných: `nt->Draw("samp:time"," ","prof")`
- I/O operace (čtení ntuplu ze souboru a uložení ntuplu do souboru na disk) – stejně jako u histogramů (viz strana 70) 
 - příklad zápisu:

```
TFile *f=new TFile("ntuple.root","NEW");  
nt->Write();  
f->Close();
```

Ntuple (3)

- Složitější operace: projekce do připraveného histogramu
 - jedna proměnná:

```
TH1F *h=new TH1F("h","my time",100,0,100);  
nt->Draw("time >> +h"); // + znamená přidat !!
```
 - profil 2 proměnných:

```
TProfile *hp=new TProfile("hp","profil",100,0,10);  
nt->Draw("samp:time >> +hp");
```
- Tímto způsobem zobrazíme požadovanou závislost do histogramu o zadaných mezích, binningu atd. a můžeme výsledek dále zkoumat (fity různými závislostmi, ...)

Stručně o cotech

- Výběry (cuts) se v ROOTu dělají pomocí stringu:
 - a la C: `char cut[60] = "(time > 10) && (samp > 0)";`
Potřebujeme-li vytvořit cut pomocí čísla uloženého v nějaké proměnné, použijeme `sprintf(...)`
 - a la ROOT: `TString cut = "(time > 10) && (samp >0)";`
Případná čísla dodáme pomocí přetíženého operátoru `+`.
- Můžeme využít třídy `TCut`, kdy mohu využít navíc i další operace:
 - `TCut c1 = "(time > 10)";`
 - `TCut c2 = "(samp > 0)";`
 - `TCut cut = c1 && c2;`
- Použití cutu: `nt->Draw("samp:time",cut);`

Příklad (Paw / ROOT)

- Načtěte ntuple obsahující veličiny x, y, z ze souboru `~davidek/prednasky/comphep/paw/xyz.dat`
- Podívejte se, jak závisí jednotlivé proměnné na sobě. Jde o rovnoměrná rozdělení ?
- Speciálně prozkoumejte závislost $\sqrt{x^2+y^2}$ na ostatních proměnných.
- Vyberte závislost s nejvyšší korelací a vyprojektujte ji do profilového histogramu. Některé případy očividně závislost nesplňují – vyřežte je.
- Výsledek nafitujte vhodnou funkcí a odhadněte, jak asi byla data vytvořena.
- Vytvořený ntuple uložte na disk. Porovnejte délky obou souborů.

Ntuply a funkce (1)

- Občas potřebujeme pro každý případ v ntuplu udělat nějakou operaci (určit aritmetický průměr z hodnot pole, vybrat maximální hodnotu z několika proměnných, ...), kterou nelze jednoduše zapsat do příkazu v ROOTu => **použití C++ funkce**

- Kreslení a složitější cuty: v tomto případě není potřeba vytvářet žádný speciální skelet, stačí

```
Double_t myfunc(Double_t a, Double_t b) {
    if (b>0) return(a) ; else return(-a);
}
```

```
nt->Draw("myfunc(samp,time)");
```

- Provedení nějaké akce pro každý případ (řádku):
 - zde již musíme vytvořit skelet pomocí

```
nt->MakeClass("myclass") // vytvoří myclass.C, myclass.h
```

Ntuply a funkce (2)

- Dále vytvoříme cyklus přes všechny případy:

```
gROOT->LoadMacro("myclass.C"); //ekvivalent .L myclass.C
myclass *trida = new myclass(nt);
for (Int_t i=0; i< nt->GetEntries(); ++i) {
    nt->GetEntry(i);
    w=trida->x*trida->x+trida->y*trida->y+trida->z*trida->z;
    hist->Fill(trida->x+trida->y,1/sqrt(w));
}
```

- Do takto vytvořených funkcí lze dopsat libovolný kód. Funkce může mít i parametry (musíme je ručně dopsat do hlavičky). Příklady použití:
 - sčítání energie z určitých kanálů
 - použití složitých cutů, plnění histogramu s určitou vahou

Ntuply a funkce (3)

- **Příklad:** využijte skutečnou C++ funkci (tj. opravdu definovanou jako samostatnou funkci) při zobrazování závislosti `sqrt(x*x+y*y)` v předchozím příkladě (viz strana 82).



Trees (1)

- Obecnější ntuply, které mohou obsahovat jakékoli objekty:
 - proměnné jakéhokoli typu (int, float, double)
 - pole, případně i pole proměnné délky pro každý event
 - další objekty (třeba i další Tree, ...)
- Zacházení s Tree je stejné jako s ntuply (metody **Draw**, **MakeClass**, použití cutů, ...), prakticky se liší jen inicializace – dvě možnosti:
 - založení větve (**TBranch**) obsahující strukturu, která pak obsahuje jednotlivé proměnné
 - založení větve pro každou proměnnou zvlášť (častější případ, nemusím číst všechny proměnné v Tree)

Trees (2)

- Založení jednoduchého Tree (statický TTree, lze i dynamicky)

```
TTree t1("t1","a simple Tree with simple variables");
```

```
Float_t p[3]; Double_t random; Int_t ev;
```

```
t1.Branch("random",&random,"random/D");
```

```
t1.Branch("ev",&ev,"ev/I");
```

```
t1.Branch("p",p,"p[3]/F");
```

zde není referenční operátor,
p je totiž ukazatel na pole p

- Plnění – typicky ve smyčce:

- nastavení jednotlivých proměnných (p[0]-p[2], random, ev)

- zavolání metody `t1.Fill()`

- Zápis na disk:

- otevření souboru: `TFile f("tree1.root","RECREATE");`

- zápis na disk a zavření souboru: `t1.Write();`

Trees (3)

- Čtení Tree (dynamický `TTree`, lze samozřejmě i staticky):

- otevření souboru, inicializace

```
TFile *f = new TFile("tree1.root");
TTree *t1 = (TTree*)f->Get("t1");
Float_t p[3]; Double_t random; Int_t ev;
t1->SetBranchAddress("random",&random);
t1->SetBranchAddress("ev",&ev);
t1->SetBranchAddress("p",p);
```

} Přiřazení adresy
proměnné danému
jménu větve

- vlastní čtení:

```
for(Int_t i=0; i<t1->GetEntries(); ++i) {
    t1->GetEntry(i); ....
}
```

celý příklad viz [~davidek/prednasky/comphep/root/tree.C](https://github.com/davidek/prednasky/comphep/root/tree.C)

Trees (4)

- Poznámky k předchozímu příkladu:
 - nemusíme číst/mapovat všechny větve, `SetBranchAddress()` voláme jen pro ty, s nimiž chceme pracovat
 - na rozdíl od Ntuplů jsme nepoužili `MakeClass()`, který by vytvořil správný skelet. My jsme si skelet vytvořili sami inicializací jednotlivých větví (`SetBranchAddress()`)
 - skelet vytvořený `MakeClass()` inicializuje všechny větve
 - pokud máme větví hodně a chceme použít jen některé, vyplatí se nepotřebné větve inaktivovat pomocí `SetBranchStatus()` - viz ROOT manuál

Trees (5)

- Pole proměnné délky:

- příklad použití: v jednotlivých případech ("řádcích") máme různý počet částic, jejichž vlastnosti chceme do pole v Tree uložit
- Tree musí obsahovat proměnnou, která udává délku pole v každé "řádce"

```
TTree t2("t2","tree with variable array length");
```

```
Int_t length; Float_t p[1000];
```

```
t2.Branch("length",&length,"length/I");
```

```
t2.Branch("array",p,"p[length]/F");
```

- v poli **p** se do Tree na dané řádce ukládá jen prvních **length** hodnot
 - vícerozměrná pole: proměnná délka pouze v jednom (posledním) indexu
- Další příklady najdete také v </usr/share/doc/root/tutorials/tree/>

Trees (6)

- Pole proměnné délky se dnes obvykle řeší přes **std::vector**
 - plnění vektoru metodou `push_back(...)`, přístup k položce metodou `at(index)`
 - Tree nemusí obsahovat skutečnou délku vektoru na dané řádce, zjišťuje se pomocí metody `size()`
 - zápis a čtení se formálně liší (objekt vs. ukazatel na objekt)
 - viz též příklad </usr/share/doc/root-system/tutorials/tree/hvector.C>

```
#include <vector>
std::vector<float> vpx;
TTree *t3 = new TTree("t3","vectors");
t3->Branch("vpx",&vpx);
for(Int_t i=0; i<NEVENTS; ++i) {
    vpx.clear();
    for(Int_t j=0; j<.... ; ++j) {
        vpx.push_back(...);
    }
    t3->Fill();
}
```

```
#include <vector>
std::vector<float> *vpx = 0;
TTree *t3 = (TTree)f->Get("t3");
t3->SetBranchAddress("vpx",&vpx);
for(Int_t i=0; i<t3->GetEntries(); ++i) {
    t3->GetEntry(i);
    for(Int_t j=0; j<vpx->size(); ++j) {
        item = vpx->at(j); // =(*vpx)[j]
    }
}
```

- Vícerozměrná pole pomocí **std::vector**

- na rozdíl od klasických polí můžeme mít různý počet položek ve všech rozměrech

- příklad čtení:

```
#include <vector>
std::vector<<std::vector<float>> > *AntiKt_etaconst = 0;
TTree *t = (TTree)f->Get("tree");
t->SetBranchAddress("AntiKt_etaconst",&AntiKt_etaconst);
for(Int_t i=0; i<t->GetEntries(); ++i) {
    t->GetEntry(i);
    for(Int_t j=0; j<njet; ++j) {
        for(Int_t k=0; k<(*AntiKt4_etaconst)[j].size(); ++k) {
            if (fabs((*AntiKt_etaconst)[j][k]) < 1.5) {
                ....
            }
        }
    }
}
```

- takový program je potřeba nejprve zkompilovat (.L myprogram.C++), jinak nebude fungovat

Příklady (1)

Příklad 1: mějme 2 stripové křemíkové detektory s 1000 stripy, uspořádané tak, že stripy jsou v obou detektorech vzájemně kolmé. Detektor ozařujeme svazkem částic, přičemž známe polohu každé částice v rovině XY (např. nějakým nezávislým měřením).

Detektor má tzv. binární čtení, tj. daný strip je nebo není zasažen. Data z detektoru jsou zapsána ve dvou polích `xstrip[1000]`, `ystrip[1000]` typu `Bool_t`, kde každý zasažený strip má hodnotu `True`, ostatní `False`. Navíc je ke každému eventu k dispozici údaj o skutečné poloze částice v rovině XY (reálné proměnné `xgen`, `ygen`) – viz soubor

~davidek/prednasky/comphep/root/strip_hit_map.root

Příklady (2)

- Úkoly: napište 2 ROOT-makra:
 1. redukujte uvedený soubor na Tree obsahující pouze seznam zasažených stripů (`xhits[nxhit]`, `yhits[nyhit]`). Jsou-li zasaženy dva sousední stripy, považujte je za jeden zásah o délce 2 – údaje ukládejte do proměnných `dxhits[nxhit]`, `dyhits[nyhit]`. Nakonec připojte údaj o skutečné poloze částice (`xgen`, `ygen`) z původního Tree a výsledný Tree uložte do souboru `strip_hit_map_1.root`. Všimněte si velikosti obou souborů.
 2. na takto získaném Tree proveďte následující analýzy:
 - určete rozteč stripů v obou směrech za předpokladu, že hodnoty `xgen`, `ygen` jsou uvedené v μm
 - určete, za jakých podmínek jsou zasaženy dva sousední stripy
 - ze změřených údajů (`xhits[nxhit]`, `yhits[nyhit]`) určete polohu a profil svazku a nafitujte příslušnou funkcí. Porovnejte s údaji získanými ze skutečné polohy částice (`xgen`, `ygen`).

Příklady (3)

Příklad 2: uvažme stejný detektor jako v příkladu 1, který ovšem šumí, tj. v jednotlivých případech je zasaženo více stripů.

Data z detektoru jsou zapsána ve stejném formátu jako v příkladu 1 – viz soubor
`~davidek/prednasky/comphep/root/strip_hit_map_noise.root`

- Úkoly:
 - proved'te stejnou konverzi jako v příkladu 1 a vytvořte soubor `strip_hit_map_noise_1.root`
 - na takto získaném Tree proved'te následující analýzy:
 - určete opět polohu a profil svazku ze změřených údajů (`xhits[nxhit]`, `yhits[nyhit]`)
 - určete četnost šumu a jeho rozdělení, opět pomocí změřených údajů `xhits[nxhit]`, `yhits[nyhit]`

Překlad programů používajících ROOT (1)

- Složitá makra lze přeložit (zkompilovat), díky tomu pak poběží výrazně rychleji (třeba i 100x !!). Co je potřeba udělat:
 - poctivě definovat všechny proměnné (CINT je velmi tolerantní, ale při překladu to už nebude fungovat)
 - vložit hlavičkové soubory používaných ROOT-struktur (tj. konstrukce typu `#include "TH1F.h"`)
 - vlastní překlad v ROOTu: `.L mymacro.C++`
 - vytvoří se shared-library `mymacro_C.so`
 - při každém dalším spuštění ROOTu stačí jen natáhnout tuto knihovnu (`.L mymacro_C.so`) a spustit vybranou funkci: `myfunc()`

Překlad programů používajících ROOT (2)

- Externí programy využívající ROOT, ale spustitelné samostatně (tj. mimo interaktivního prostředí CINT):
 - programy jsou napsány normálně v C++ stejně jako makra pro interaktivní verzi
 - pozor na jméno zdrojového souboru, mělo mít tvar: *.C, *.cc, *.cp, *.cxx, *.cpp, *.CPP, *.c++ (kvůli spouštění C++ pre-processoru)
 - při překladu musíme jen přikompilovat knihovny ROOTu. Podrobnosti najdeme na adrese

<http://root.cern.ch/root/HowtoLink.html>

- kompilační skript: `~davidek/bin/gccroot`
- Výhody externích programů: lze použít i funkce z jiných knihoven (speciální funkce z CERNLIB, Pythie,...)

Grafický výstup z ROOTu (1)

- ROOT umožňuje výstup do mnoha různých grafických formátů (PS, EPS, GIF, PDF, ...). Na výběr máme dvě možnosti jak obrázek exportovat:
 - ručně pomocí menu File v daném grafickém okně třídy TCanvas
 - příkazem typu `c1->Print("obr.ps")`, kde c1 je pointer na příslušné grafické okno třídy TCanvas (nebo i pod-okno typu TPad). Podle přípony se vytvoří příslušný typ grafického souboru.
 - do jednoho souboru lze postupně vytisknout i více obrázků, tj. soubor bude mít více stránek. Podrobnosti viz

<http://root.cern.ch/root/html//TPad.html#TPad:Print>

Grafický výstup z ROOTu (2)

- Jaký formát je lepší ? Záleží k čemu:
 - **PostScript (PS)**: určen především k přímému tisku
 - **Encapsulated PostScript (EPS)**: jde o "PS s definovaným bounding boxem", dobře se škáluje. Vhodný zejména ke vkládání do textů (TeX, LaTeX, ...)
 - **Graphic Interchange Format (GIF)**: hodí se pro použití v prezentacích typu PowerPoint, OpenOffice, ...
 - **Portable Document Format (PDF)**: hodí se jednak k přímému tisku, lze dobře škálovat (vektorový charakter). Navíc lze vkládat i do dokumentů zpracovávaných pomocí pdflatex.

Grafický výstup z ROOTu (3)

- Další možnosti uschování obrázku z ROOTu:
 - "Save as ROOT macros (*.C)"
 - ROOT vytvoří se C++ kód, kde se vytváří graf/histogram, osy, popisky, atd. Naplnění histogramu je řešeno pomocí série příkazů typu `SetBinContent()`
 - v takovém kódu lze např. jednoduše měnit popisky os a vzhled objektu (barvy, typy markeru atd)
 - "Save as ROOT files (*.root)"
 - do zmíněného souboru se uloží přímo objekt typu `TCanvas`, který lze znovu nakreslit pomocí `TCanvas::Draw()`

Kapitola 5: Sazba dokumentů v LaTeXu



... o psaní dokumentů a dodržování základních typografických pravidel. Sázení matematických formulí, vkládání obrázků a jiné finty ...

Programy na tvorbu dokumentů

- Rozdělení podle vlastnosti **WhatYouSeeIsWhatYouGet**:
 - obecné, WYSIWYG: Word, FrameMaker
 - specializované, WYSIWYG: Quark
 - non-WYSIWYG: TeX, sgml, ...

K většině non-WYSIWYG existují tzv. grafické front-endy (SciWord, LyX, ...)

- Většina programů je typu WYSIWYG. Nevýhody:
 - psaní rozsáhlejších děl je náročné (stránkové zlomy, křížové odkazy, zpracování celého textu)
 - umožňují jednoduše porušovat typografická pravidla (dělení slov, typy fontů, rozměry, rozvržení stránek)
 - problém s kompatibilitou mezi různými systémy

TeX a jeho klony (1)

- Typografický systém **TeX** (autor Donald E. Knuth)
 - propracovaný systém podporující sazbu matematiky, udržování rozsáhlých textů
 - má zakódovány základní typografická pravidla
 - existuje na všech platformách, zdrojový text je ASCII (vlastně jde o specifický druh sgml)
 - k dispozici je řada rozšíření (stylů) podporujících např. psaní not (MusicTeX), kreslení Feynmanových diagramů, psaní v netradičních znakových sadách (hebrejščina, arabština, čínské a japonské znaky)
- Jak **TeX** funguje:
 - zdrojový kód obsahuje "značky" vymežující např. typy fontů, strukturální prvky (kapitoly, odkazy, ...), matematické vzorce, místa pro vložení grafiky atd.

TeX a jeho klony (2)

- překlad zdrojového kódu (dělení slov, umístění grafických objektů, vytvoření vzájemných vazeb) → vytvoření grafického formátu
- Základní TeX je relativně složitý. Existuje ale nadstavba ("makrojazyk") **LaTeX** (autor Leslie Lamport):
 - mnoho předdefinovaných prostředí pro vzorce, tabulky, obrázky, vzhled dokumentů (klasický, transparence, ...)
 - menší nároky na znalosti uživatele (jednodušší než TeX), ale kdykoli se lze ponořit zpět na úroveň samotného TeXu
 - klony: **pdflatex** (přímá tvorba PDF, klikací odkazy)

Podpora češtiny v LaTeXu (1)

- LaTeX akceptuje znaky v různých kódováních, také umí české dělení slov a vytváření českých popisků (obrázky, tabulky, ...)
- Starší přístup: použití binárek `cslatex`, `pdfcslatex`. Hlavička dokumentu

```
\usepackage{czech}
\usepackage[utf8]{inputenc}
```
- Nový přístup: použití Babel + `latex` (`pdflatex`). Hlavička

```
\usepackage[czech]{babel}
\usepackage[utf8]{inputenc}
\usepackage[IL2]{fontenc}
```
- V obou případech předpokládáme vstupní kódování češtiny v UTF8, na výstupu jsou pak použity Computer Modern fonty v kódování IL2 (tzv. CSfonty)

Podpora češtiny v LaTeXu (2)

- Alternativně lze použít Latin Modern fonty, k dispozici v kódování T1 a IL2

```
\usepackage[czech]{babel}  
\usepackage[utf8]{inputenc}  
\usepackage{lmodern}  
\usepackage[T1]{fontenc}
```

Literatura, dokumentace (1)

- Původní literatura:
 - D.E.Knuth: *The TeXBook* (ISBN 0-201-13447-0, brožované ISBN 0-201-13448-9)
 - L.Lamport: *LaTeX – A Document Preparation System* (ISBN 0-201-52983-1)
 - M.Goossens et al: *The LaTeX Companion* (ISBN 0-201-54199-8)
 - M.Goossens et al: *The LaTeX Graphics Companion* (ISBN 0-201-85469-4)

Literatura, dokumentace (2)

- Dostupné v češtině:
 - P.Olšák: *Typografický systém TeX*
 - P.Olšák: *TeXBook naruby*
 - J.Rybička: *LaTeX pro začátečníky* (2.vydání)
 - H.Kopka, P.Daly: *LaTeX – kompletní průvodce* (ISBN 80-7226-973-9)

Literatura, dokumentace (3)

- Dokumentace na Webu:
 - stránky České sdružení uživatelů TeXu (CSTUG)
<http://www.cstug.cz>
obsahují celou řadu odkazů týkajících se (La)TeXu, včetně PDF verzí Zpravodajů.
 - seriál o TeXu na serveru Root.cz
<http://www.root.cz/serialy/tex-pro-kazdeho/>
 - manuál **TeX at CERN**, popisuje i řadu užitečných balíčků
<http://consult.cern.ch/writeup/texatcern/>
 - T.Oetiker et al: **The not so short introduction to LaTeX2 ϵ**
<http://www.loria.fr/tex/general/lshort2e.dvi>

Úvod do LaTeXu (1)

- Základní struktura:
 - hlavička dokumentu (použité styly, kódování):

```
\documentclass[11pt]{article}
\usepackage{a4,czech,epsfig}
```
 - tělo dokumentu: obsahuje vlastní text a ostatní objekty:

```
\begin{document}
bla bla bla
\end{document}
```
- Příkazy (značky) LaTeXu:
 - začínají obráceným lomítkem (backslash)
 - povinné parametry ve složených, volitelné v hranatých závorkách

Úvod do LaTeXu (2)

- Znak se speciálním významem:
 - % ... vše za ním na dané řádce je komentář (zdrojový text)
 - \$... uvozuje matematické výrazy ($x+y>2$)
 - ~ ... nedělitelná mezera standardní velikosti. Použití:
 - psaní jmen: D.~E.~Knuth
 - jednopísmenné předložky: v~pátek
 - odkazy (viz dále)
 - prázdný řádek ... odděluje odstavce
 - Formát zdrojového textu:
 - více prázdných řádků se bere jako jeden prázdný řádek
 - jedna a více mezer je stále jen jedna mezera
- => volný formát zdrojového textu !**

Úvod do LaTeXu (3)

- Kapitoly, pod-kapitoly:

`\section{První kapitola}`

Text první kapitoly. Bla bla bla

`\subsection{Naše podkapitolka}`

Pozor ! Veškeré číslování si obstarává LaTeX sám.
Chceme-li číslování u daného prvku vynechat, napíšeme
v tomto případě: `\section*`{První kapitola}

- Vytváření obsahu: zcela automaticky příkazem `\tableofcontents`

Úvod do LaTeXu (4)

- Nejpoužívanější prostředí:

- **centrovaný text**

```
\begin{center}
```

Tento text se zobrazí centrovane \\ ve dvou řádcích

```
\end{center}
```

- **výčty**: obyčejné (*itemize*) či číslované (*enumerate*). Příklad:

... záleží na počasí:

```
\begin{itemize}
```

```
\item V~případě, že bude pršet, ...
```

```
\item Nebude-li pršet, ....
```

```
\item Ať tak či onak, ...
```

```
\end{itemize}
```

Zkrátka to nějak dopadne.

... záleží na počasí:

- V případě, že bude pršet,
- Nebude-li pršet, ...
- Ať tak či onak, ...

Zkrátka to nějak dopadne.

Úvod do LaTeXu (5)

– obrázky:

```
\begin{figure}
```

```
\begin{center}
```

```
\epsfig{file=soubor.eps,width=0.5\linewidth} %vyžaduje balíček epsfig
```

```
\includegraphics[width=0.5\linewidth]{soubor.eps} %vyžaduje graphicx
```

```
\end{center}
```

```
\caption{Můj krásný obrázek}
```

```
\end{figure}
```

Poznámky:

- pro grafy, histogramy apod preferujeme vektorové formáty obrázků (eps, pdf)
- latex umožňuje vložení *.eps, pdflatex naopak *.pdf. Zde lze ale využít automatické konverze eps → pdf
 - `\include{epstopdf}`
 - uvedení souboru bez přípony `\includegraphics{soubor}`, pokud najde jen obrázek v eps, konvertuje jej to pdf

Úvod do LaTeXu (6)

– tabulky:

`\begin{table}`

`\begin{center}`

`\begin{tabular}{|c|l|r|c|} \hline`

Případ & A & B & Poznámka `\ \hline`

1 & 0.5 & 2 & jednoduché `\ \hline`

2 & 0.1 & 10 & složitější `\ \hline`

`\end{tabular}`

`\end{center}`

`\caption{Jednoduchá tabulka}`

`\end{table}`

Případ	A	B	Poznámka
1	0.5	2	jednoduché
2	0.1	10	složitější

Poznámky:

Tabulka 1: Jednoduchá tabulka

- spojení několika cel: `\multicolumn`
- podtržení jen některých sloupců: `\cline{2-3}` místo `\hline`

Úvod do LaTeXu (7)

- Poznámky o tabulkách a obrázcích
 - obrázek je tzv. plovoucí objekt, LaTeX ho umístí uje na vhodné místo, tedy ne nutně přesně tam, kde ho máme v textu my (lze částečně ovlivnit nepovinným parametrem).
 - popis může být nad či pod obrázkem (podle toho, kde příkaz `\caption` uvedeme).
 - Číslování je automatické, nastavením lze ovlivnit vzhled.

Úvod do LaTeXu (8)

- Křížové odkazy:
 - pomocí `\label` a `\ref`, parametrem je symbolické jméno. Konkrétní číslo se místo `\ref` přiřadí až při zpracování.
 - příklad:

```
\section{Kapitola o odkazech} \label{kap:odkazy}
```

Tato kapitola je věnována odkazům....

```
\section{Kapitola o něčem jiném}
```

Jak jsme zmínili v kapitole `\ref{kap:odkazy}`,

Poznámky:

- každý typ objektu (kapitoly, tabulky, obrázky, rovnice) se čísluje zvlášť. Při odkazování na rovnice musíme závorky napsat ručně, tedy:

viz rovnice `(\ref{eq:rozpad})`

- odkazy na literaturu: speciální příkazy `\bibitem`, `\cite`

Matematické výrazy (1)

- Veškeré proměnné (nikoli čísla či funkce !) se automaticky sází jiným fontem (mat. kurzíva). Matematické prostředí má několik typů, mírně se liší způsobem zobrazení:

- **textový styl:** pro psaní jednoduchých výrazů přímo do řádek v textu. Příklad:

Chování funkce záleží na znaménku čitatele $x-y$

- **vzorcový styl:** matematické výrazy/vzorce/rovnice na samostatném řádku. Existují 3 možnosti:

- jeden výraz, nečíslováno: $x^2 + y^2 = 1$ (je to ekvivalentní použití prostředí *displaymath* (tj. `\begin{displaymath} ... \end{displaymath}`))
- jeden výraz, číslováno: prostředí *equation*
- více výrazů: prostředí *eqnarray*

Podívejme se na dva příklady:

Matematické výrazy (2)

– **jeden výraz/rovnice:**

```
\begin{equation}
```

$$x^2 + y^2 = 1$$

```
\end{equation}
```

Rovnice bude na zvláštní řádce, vodorovně centrovaná, vertikálně mírně odsazená, očíslovaná vpravo:

$$x^2 + y^2 = 1 \tag{1}$$

Pokud bychom to chtěli jinak (např. žádné centrování, číslování vlevo), lze to změnit v hlavičce dokumentu a bude to platit pro všechny rovnice.

Chceme-li se na rovnici v textu odkazovat, přidáme uvnitř prostředí příkaz `\label`.

Matematické výrazy (3)

– více výrazů/rovníc:

```
\begin{eqnarray}
x^2 + y^2 & = & 1 \\
3x + y & = & 2
\end{eqnarray}
```

Soustava rovnic bude na zvláštních řádcích, vertikálně mírně odsazená, horizontálně centrovaná na znaménko "=". Obě rovnice budou číslované vpravo:

$$x^2 + y^2 = 1 \quad (1)$$

$$3x + y = 2 \quad (2)$$

Takto lze psát i dlouhé výrazy, kdy nečísloujeme první řádek (příkaz `\nonumber`).

Matematické výrazy (4)

- Konstrukce matematických výrazů - **indexy**:
 - horní indexy a mocniny jsou uvozeny stříškou "^". Je-li indexový výraz delší než jeden znak, uzavřeme jej do skupiny pomocí {}, např.:

$$e^{-iE}$$

- dolní indexy: uvozeny podtržítkem "_", pro vícepísmenné indexy opět použijeme skupiny
- indexy mohou být víceúrovňové, tj. horní index může mít dolní index atd, např.:

$$e^{iT_{jk}}$$

se zobrazí jako:

$$e^{iT_{jk}}$$

Matematické výrazy (5)

- výraz může mít horní i dolní index současně, indexy mohou být i vlevo od výrazu. Např.:

`\Lambda^{\mu}_{\nu} \quad ^{235}_{92}\mathrm{U}`

se zobrazí:

(`\phantom` je kvůli zarovnání !)

$$\Lambda_{\nu}^{\mu} \quad {}^{235}_{92}\mathrm{U}$$

- **matematika a mezery:**

- mezery ve zdrojovém kódu nehrají roli (TeX je zvolí dle typografických pravidel). Chceme-li donutit TeX vložit nestandardní mezeru, použijeme některý z následujících příkazů
- malá, střední, normální mezera: `\,$` `;$` `\ $`
- velká (čtverčiková) a 2x velká mezera: `\quad$` `\qquad$`
- záporná mezera: `\!$` (velikost odpovídá malé mezeře `\,$`)

Matematické výrazy (6)

- **zlomky:**

- příkaz `\frac{čitatel}{jmenovatel}`
- mohou být vnořené

- **závorky:**

- několik velikostí (od nejmenší do největší): `\bigl(` , `\biggl(` , `\Bigl(` , `\Biggl(` . Právě závorky pomocí `\bigr)` atd. Znak "(" odpovídá nejmenší závorce.

((((

- stejným způsobem se sází hranaté závorky, ale pozor na složené: `\bigl\{` (odlišení od skupin)

- automatická velikost závorek: `\left(\frac{1}{\frac{3}{7}}\right)`

$$\left(\frac{1}{2 \times \frac{3}{7}}\right)$$

Matematické výrazy (7)

- **sumy a integrály:**

- příkazy `\sum`, `\int`, `\oint`, příklad:

```
\begin{eqnarray}
```

```
\sum_{i=1}^n \frac{1}{i \times (i+1)} & = & 1 - \frac{1}{n+1} \\\
```

```
\int_0^{\infty} \mathrm{d}x \cos x^2 & = & \sqrt{\frac{\pi}{8}} \\\
```

```
\int\limits_{0}^{\infty} \mathrm{d}x \; e^{-x} & = & 1 \\\
```

```
\oint \mathrm{d}S f(s) & = & 1
```

```
\end{eqnarray}
```

$$\sum_{i=1}^n \frac{1}{i \times (i + 1)} = 1 - \frac{1}{n + 1}$$

$$\int_0^{\infty} dx \cos x^2 = \sqrt{\frac{\pi}{8}}$$

$$\int_0^{\infty} dx e^{-x} = 1$$

$$\oint dS f(s) = 1$$

Všiměme si: příkaz `\limits` posune u integrálu horní mez výše a dolní mez níže – platí ve vzorcovém stylu. V textovém stylu by se meze zobrazily ještě "blíž sobě".

Matematické výrazy (8)

- **jména funkcí:**
 - nesází se matematickou kurzívou, ale normálním fontem
 - většina funkcí je dostupná jako příkaz: \sin , \cos , \lim , ...
- Další vymoženosti (matice, vektory, speciální znaky, ...) najdete v manuálech.

Jak LaTeXovat

- Vytvoření, zpracování a tisk LaTeX-dokumentů:
 - vytvoření zdrojového textu (např. v Emacsu, který má makra pro podporu psaní LaTeX-dokumentů)
 - spuštění LaTeXu:
 - `latex mujdoc.tex` (případně `cslatex`)
 - obvykle je potřeba spustit dávku několikrát, aby se vytvořily správně křížové odkazy, obsahy atd (píší se do pomocných souborů `*.aux`, `*.toc`, `*.lot`, `*.lof`, ...). Dávka na to sama upozorní.
 - prohlížení:
 - dávka `latex` vytvoří soubor typu `*.dvi`, který se dá prohlížet pod Linuxem pomocí programu `xdvi` (ve Windows např. `windvi`, `Yap`)
 - pro tisk (a prohlížení případných vložených grafických objektů) je potřeba vytvořit PostScript: `dvips -o mujdoc.ps mujdoc.dvi`
 - prohlížení PostScriptu: programy `ghostview`, `gv`

Časté typografické prohřešky (1)

- Fyzikální jednotky, prvky, částice atd ve výrazech psané matematickou kurzívou. Nejde o proměnné, proto se mají sázet běžným fontem:
 - jednotky: $M = 100 \sim \mathrm{GeV}$ (pozor na mezeru !)
 - mezera závisí na typu fontu, může být i menší (\sim ; nebo \sim)
 - použití balíčku siunitx: $M = \mathrm{SI}\{100\}\{\mathrm{GeV}\}$
 - prvky: $^{12}_6 \mathrm{C}$ (mezera kvůli zarovnání)
 - částice: $\mathrm{Z}^0 \rightarrow \mathrm{e}^+ \mathrm{e}^-$
 - indexy s fyzikálním významem: p_{T} nebo $p_{\scriptsize \mathrm{T}}$

Na poslední dva případy se často zapomíná.... Sázejme alespoň stejné věci stále stejným způsobem !!

Časté typografické prohřešky (2)

Také lze využít vlastní definice v hlavičce dokumentu, například: `\newcommand{\pt}{p_{\mathrm{T}}}`

- výhoda: mohu to kdykoli jednoduše změnit....
- poznámka pro experty: poslední definici lze použít pouze v matematickém režimu. Univerzální definice může vypadat:

`\newcommand{\pt}{\ensuremath{p_{\mathrm{T}}}}`

- Nepoužívání nezlomitelné mezery (~). Správně má být:
 - v odkazech: odstavec~`\ref{sec:uvod}`
 - ve jménech: D.~E.~Knuth
 - u jednopísmenných předložek: v~lese

Nezlomitelná mezera má dvojí funkci !

LaTeX pro pokročilé – typy a velikosti písma

- Základním typem je stojací písmo (`\textup{}`.) Další používané typy:
 - italika: `\textit{text italikou}` *text italikou*
 - slanted: `\textsl{text podobný italice}`
 - bezpatkové písmo: `\textsf{bezpatkové písmo}`
 - tučné písmo: `\textbf{tučný titulek}` **tučný titulek**

Pokud potřebujeme nějaké netradiční fonty, lze je jednoduše vložit do systému.

- Velikosti písma: základní velikost `\normalsize` můžeme
 - zvětšovat: `\large` < `\Large` < `\LARGE` < `\huge` < `\Huge`
 - zmenšovat: `\small` > `\footnotesize` > `\scriptsize` > `\tiny`

Obecně platí: příliš mnoho typů/velikostí písma v dokumentu ztěžuje čitelnost => zbytečně neplýtváme

LaTeX pro pokročilé – umístování plovoucích objektů (1)

- Plovoucí objekty (prostředí *table*, *figure*) LaTeX umístí na první vhodné místo. Ne tedy nutně tam, kam jsme určili ve zdrojovém textu. Typografická omezení:
 - objekt příliš velký, nevejde se na danou stránku
 - poměr textu a plovoucích objektů na dané stránce (např. jedna řádka textu doplněná velkým obrázkem je nepřijatelná)
 - omezení na maximální počet plovoucích objektů na jedné stránce
- Nepovinné parametry plovoucích objektů: `[htbp]`
 - jedná se o "doporučení", kam má LaTeX objekt umístit. Nejsou-li uvedeny, předpokládá se výše uvedené pořadí. Syntaxe: `\begin{table}[hbt]`

LaTeX pro pokročilé – umístování plovoucích objektů (2)

- Jak to funguje ? TeX se snaží umístit objekt dle našeho přání (při dodržení typografických pravidel):
 - h=here – objekt se umístí přesně tam, kam by měl podle zdrojovému textu
 - t=top, b=bottom – objekt se umístí na začátek, resp. konec stránky
 - p=page – objekt se umístí na zvláštní stránku na konci dokumentu či kapitoly

Z toho vyplývá:

- chceme-li mít dva obrázky umístěné na jedné stránce, první je lépe definovat s umístěním "t" a druhý s umístěním "b"
- pokud se jeden objekt umístí jako "p" na konec dokumentu, všechny ostatní už budou nutně za ním (pořadí se zachovává)

LaTeX pro pokročilé – umístování plovoucích objektů (3)

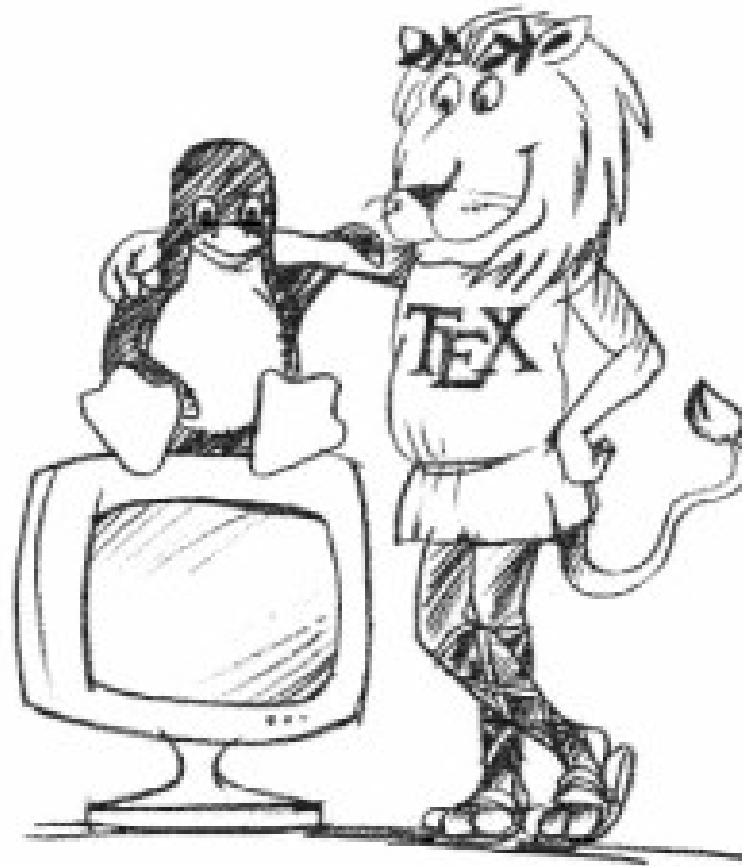
- Co lze ovlivnit, aniž bychom LaTeX příliš znásilňovali (nastavení se provádí v hlavičce dokumentu):
 - maximální počty plovoucích objektů umístěných:
 - na začátku jedné stránky (volba "t"): `\setcounter{topnumber}{2}`
 - na konci jedné stránky (volba "b"): `\setcounter{bottomnumber}{1}`
 - celkový počet na jedné stránce: `\setcounter{totalnumber}{3}`
 - maximální podíly pro plovoucí objekty umístěné:
 - nahoře na stránce ("top"): `\renewcommand\topfraction{0.7}`
 - dole na stránce ("bottom"): `\renewcommand\bottomfraction{0.3}`
 - minimální podíl textu na stránce (proto nelze mít velký obrázek a jednu řádku): `\renewcommand\textfraction{0.2}`

Uvedené hodnoty jsou přednastavené ve stylu *article*. V jiných stylech to může být jinak.

LaTeX pro pokročilé – o čem ještě nebyla řeč

- Neúplný výčet toho, co dalšího ještě LaTeX umí:
 - poznámky pod čarou (příkaz `\footnote{}`)
 - kejkle s vkládanými objekty (oříznutí, škálování, rotace)
 - definice vlastních příkazů, prostředí a stylů, makra s parametry
 - snadné vytváření rejstříků, třídění podle různých abeced
- Dále existuje celá řada podpůrných programů, např.:
 - test syntaktické správnosti (`lacheck`)
 - vkládání vlnky "~" za jednopísmenné předložky (makro `tildify.el` pro Emacs)
 - převod zdrojového LaTeX-textu do HTML (`latex2html`)
 - tvorba fontů či obrázků: `MetaFont`, `MetaPost`

... at' vám všechny aplikace dobře slouží !



That's all, folks.

Přílohy

1. Fortran vs. C
2. Paw

Příloha 1: Pár slov o Fortranu a C

... aneb velmi stručně o rozdílech v syntaxi obou programovacích jazyků, kompilaci programů pod Linuxem a využití knihoven CERNLIB.

Dodatek pro zvědavce: pohádka o vlastních knihovnách

Fortran vs. C (1)

- Fortran původně určen na vědecko-technické výpočty. C umožňuje operace na velmi nízké úrovni (např. bitové operace, ovládání hardware, ...)
- Fortran nerozlišuje malá/velká písmena, C ano.
- Formát zdrojového textu:
 - Fortran: každý příkaz na jednom řádku s možností pokračovacího řádku. Jen sloupce 7-72 !! (sloupce 1-5 určeny pro návěští, 6=značka pro pokračovací řádek)
 - C: žádné omezení, příkaz končí středníkem.
- Parametry procedur a funkcí:
 - Fortran: volání odkazem
 - C: volání hodnotou (odkazem jedině přes pointery)

Fortran vs. C (2)

- Globální proměnné:
 - Fortran: řeší se přes common-bloky
 - C: definice proměnné vně všech funkcí
- Lokální proměnné: v různých procedurách jsou nezávislé (stejně ve Fortranu, C, Pascalu ...)
- Pole:
 - Fortran: první index je 1, u vícerozměrných polí se mění nejrychleji první index (způsob uložení pole v paměti)
 - C: první index je 0, u vícerozměrných polí se mění nejrychleji poslední index.

... podívejme se na velmi jednoduchý příklad ...

Fortran vs. C (3)

```
program test
```

```
implicit none
```

```
integer i,j
```

```
real a(4,2)
```

* naplneni pole a:

```
do j=1,2
```

```
  do i=1,4
```

```
    a(i,j)=10.*j+float(i)/10.
```

```
  enddo
```

```
enddo
```

```
#include <stdio.h>
```

```
main()
```

```
{
```

```
  int i,j;
```

```
  float a[2][4];
```

```
  /* naplneni pole a */
```

```
  for(j=0; j<2; j++){
```

```
    for(i=0; i<4; ++i)
```

```
      a[j][i]=10*j+((float)i)/10;
```

```
  }
```

Fortran vs. C (4)

```
do j=1,2
  write(*,11)(a(i,j),i=1,4)
enddo
11 format(4f6.2)
end
```

- Kompilace pomocí `g77` (viz [~davidek/bin/gf](#))
- Jméno zdrojového souboru: `*.f`, `*.for`, `*.FOR`

```
for(j=0; j<2; j++){
  for(i=0; i<4; ++i)
    printf("%6.2f",a[j][i]);
  printf("\n");
}
```

- Kompilace pomocí `gcc` (viz [~davidek/bin/c](#))
- Jméno zdrojového souboru:
 - C: `*.c`
 - C++: `*.C`, `*.cc`, `*.cp`, `*.cxx`, `*.cpp`, `*.CPP`, `*.c++`

Knihovny CERNLIB (1)

- Obsahují celou škálu užitečných procedur a funkcí:
 - generátory náhodných čísel podle různých rozdělení
 - maticové a vektorové operace
 - řešení soustav rovnic
 - speciální funkce (Bessel, erf, Landau, ...), numerické integrace
 - routiny z aplikací **Paw**, **Pythia/Jetset**, **Geant3**, **Herwig**, **Minuit** (minimalizace mnoha-dim funkcionálů)
 - a mnohé další ...
- Dokumentaci najdeme na Webu:

<http://wwwinfo.cern.ch/asdoc/cernlib.html>

Knihovny CERNLIB (2)

- Většinou napsané ve Fortranu, tedy jednoduché volání (F). Použití v C:
 - využití `cfortran.h` a příslušné hlavičky pro danou knihovnu (jednoduché, neboť máme proměnné stejného typu jako ve Fortranu)
 - obecné použití Fortran-funkce z knihovny: definice funkce (s podtržítkem) v hlavičce, pozor na přetypování proměnných, zvláště pak polí (neboť ve Fortranu se volá odkazem) !!
- Jak zkompilujeme program s knihovnami ?
 - fáze překladač (překlad našeho kódu) a fáze linkování (sestavení našeho kódu s knihovními funkcemi...)
 - při linkování obecně záleží na pořadí knihoven, ale v případě CERNLIB je to ošetřeno

Knihovny CERNLIB (3)

– kompilační skripty:

- `~davidek/bin/g77l` (Fortran)
- `~davidek/bin/gccl` (C)

- **Příklad:** program **matice** naplní matici 3x3 náhodnými čísly z intervalu (0,1) a vektor pravých stran (v intervalu (-2,2)). Výsledkem je řešení soustavy rovnic pomocí konstrukce inverzní matice. Výpis mezivýsledků, program musí fungovat stejně ve Fortranu i C !!

– nápověda:

- použití CERNLIB funkcí/procedur `rndm` a `rinv`
- v případě řešení v C pomocí `cfortran.h` nezapomeňte vložit také ostatní hlavičkové soubory knihoven CERNLIB, ve kterých se výše uvedené funkce/procedury nacházejí (viz dokumentace na Webu)

Knihovny CERNLIB (4)

- Jak to v praxi bude vypadat ?

- Fortran – triviální (viz příklad FortranC/matrice.f)

```

real x,dummy,mat(3,3),res(3)    ! definice reálných proměnných
x=rndm(dummy)                   ! získání náhodného čísla
call rinv(3,mat,3,res,ifail)     ! inverze reálné matice

```

- C s využitím cfortran.h (viz příklad FortranC/matrice-1.c)

```

#include <cfortran.h>             /* vložení hlavičkových souborů */
#include <packlib.h>
#include <kernlib.h>

float x,dummy,mat[3][3],res[3]; /* definice reálných proměnných */
x=RNDM(dummy);                  /* získání náhodného čísla */
RINV(3,mat,3,res,ifail);        /* inverze reálné matice */

```

A musím navíc dát pozor na "transpozici matice".....

Knihovny CERNLIB (5)

- čisté C (viz FortranC/matrice-2.c) aneb něco pro experty:

```
float rndm_(float *r);           /* definice prototypu funkce */  
void rinv_(int *d, float *m, int *n, float *wrk, int *ifail); /* dtto */  
  
float x,dummy,mat[3][3],res[3]; /* definice reálných proměnných */  
x=rinv_(&dummy);                /* pozor na volání odkazem */  
k=3; rinv_(&k,mat[0],&k,res,&ifail); /* pozor na volání odkazem */
```

A samozřejmě musíme dát zase pozor na "transpozici matice"...

A zase příklad

- Ve Fortranu napište program **integ.f**, který numericky integruje funkci **exp(-x)** na intervalu (0,2)

- použijte knihovní funkci **GAUSS** z mathlib/kernlib, Fortran-syntaxe:

external FCE

vysledek = GAUSS(FCE,odkud,kam,presnost)

kde funkce **FCE** právě definuje uvedenou exponenciálu.
Všechny proměnné jsou typu **real**.

- Napište stejný program v C s využitím výše uvedené knihovní funkce:
 - program **integ-1.c**, s využitím **cfortran.h**
 - program **integ-2.c**, s přímým voláním knihovní funkce (bez využití **cfortran.h**)

Common bloky ve Fortranu, ale co v C ? (1)

- Ve Fortranu se takto obecně definují globální proměnné:

```
integer i,j
```

```
real a,b
```

```
common /moje/a,b,i,j
```

- Přístup k těmto proměnným mám v těch funkcích, kde tento blok uvedu. Pozor – pokaždé musím definovat typ proměnných.
- Fortran-knihovny/aplikace využívají zvláštní **common bloky** pro přístup k datům, proto to musím umět i v C:

```
struct {
```

```
    float a,b; int i,j;
```

```
} moje_ ; /* pozor na podtržitko - stejne jako u funkci */
```

Common bloky ve Fortranu, ale co v C ? (2)

- Jednoduchý příklad: použití **common bloku** v programu, který vyrobí histogram pro použití v Paw:
 - Fortran: `~davidek/prednasky/comphep/FortranC/hist.f`
 - C: `~davidek/prednasky/comphep/FortranC/hist.c`
- Oba programy fungují stejně, i výstup mají stejný (přesvědčíme se, až se dostaneme k Paw)
- Uvedené použití **common bloku** je specifické (rezervace dynamické paměti, kterou Fortran nemá). Klasické předdefinované bloky se v knihovnách/aplikacích používají např. pro přístup k parametrům a výsledkům fitu (**pawpar**, **hcfits** v Paw), seznamu generovaných částic (**pyjets** v Pythii), ...

Vlastní knihovny pro fajnšmekry (1)

- Vytvořme vlastní knihovnu obsahující funkce skalární součin a normu vektoru o obecné dimenzi n :
 - provedeme ve Fortranu (ale zcela stejně by to šlo i v C, rozdíl jen ve volání odkazem/hodnotou)
 - každá funkce ve vlastním souboru: `sksoucin.f`, `norma.f`
 - překlad pomocí `g77`, vytvoření object-souborů (`g77 -c...`)
 - vytvoření statické knihovny:
`ar -r libvect.a sksoucin.o norma.o`
- Napišme program, který volá uvedené knihovní funkce:
 - Fortran: `~davidek/prednasky/comphep/FortranC/vektory.f`
 - C: `~davidek/prednasky/comphep/FortranC/vektory.c`
(samozřejmě přístup bez `cfortran.h`, tj. přímé volání)

Vlastní knihovny pro fajnšmekry (2)

- Jak překládat takové programy ? Stejně jako v případě CERNLIB:

- Fortran: `g77 program.f libvektor.a`

- C - dvojstupňový překlad:

- `gcc -c program.c # pouze překlad, nelinkuje se`

- `g77 program.o libvect.a # linkuje object-soubor s knihovnou`

Příloha 2: Paw

- vektory, histogramy, fity
- Ntuply, cuty, funkce

Paw

- **PAW** (Physics Analysis Workstation) – nástroj pro analýzu dat. Základní datové struktury:
 - vektory (maximálně 3-dim)
 - histogramy (1-dim, 2-dim, profilový)
 - ntuply ("multi-dimenzionální tabulky")
- Vlastní interpret příkazů **COMIS** (podobný Fortranu), relativně jednoduchá syntaxe. Podpora uživatelských funkcí ve Fortranu i v C (složitější, ale jde to taky)
- Dokumentace:
 - manuál: <http://paw.web.cern.ch/paw>
 - FAQ:
<http://paw.web.cern.ch/wwwasd/cgi-bin/listpawfaqs.pl>
 - nápověda v Paw: **help příkaz** nebo **us příkaz**

Paw: vektory (1)

- Vektory používáme při prokládání křivky množinou bodů (fitování) a přenos mezivýsledků (prvky histogramů, výsledky fitů, ...)
- Vytvoření:
 - vektor reálných prvků: `ve/cre par(3) r 0. 1.5 -1.`
 - celoločíselný vektor: `ve/cre idh(10) i 5*0 2 4 5 2*-1`
- Zrušení vektoru:
 - daného vektoru: `ve/del par`
 - všech dosud definovaných vektorů: `ve/del *`
- Čtení ze/ zápis do souboru, na obrazovku:
 - `ve/read xx,yy soubor.dat`
 - `ve/wri xx data.dat 'f10.5' ; ve/print xx`

Paw: vektory (2)

- Zobrazení: `ve/plot par` , seznam vektorů: `ve/list`
- Aritmetické operace:
 - škálování (`ve/op/vscale`), přičtení čísla (`ve/op/vbias`)
 - sčítání, odčítání vektorů (`ve/op/vadd`; `ve/op/vsub`), násobení vektorů (`ve/op/vmult`)
 - složitější aritmetické kejkle pomocí **SIGMA** (speciální jednotka v Paw, viz manuál)
- Fitování: proložení parametrické funkce množinou bodů $[xx(i); yy(i) \pm ey(i)]$ metodou nejmenších čtverců:
 - vestavěné funkce: Gauss (**g**), polynom N-tého stupně (**pN**), exponenciála (**e**)

příklad: `ve/fit xx yy ey p2`

Paw: vektory (3)

– uživatelská funkce:

```
ve/cre par(3) r 10. 0. 1. ; ve/fit xx yy ey fce.f ! 3 par
```

kde soubor `fce.f` vypadá např. takto (jméno souboru je totožné se jménem funkce !):

```
real function fce(x)
```

```
common/pawpar/par(3)
```

```
fce=par(1)+par(2)*x+par(3)/x**2
```

```
end
```

Pozor: musím zadat počáteční hodnoty parametrů !!

– Lze fitovat i funkcí psanou v C, **Paw** pak zavolá nejdříve překladač a funkce se "natáhne" do paměti. Stejně to jde udělat i s Fortran funkcí, podrobnosti viz

<http://paw.web.cern.ch/wwwasd/cgi-bin/listpawfaqs.pl/16>

Paw: histogramy (1)

- Jednoduchý datový útvar, zobrazující např. tvar spektra, funkci, ...
- Založení histogramu (COMIS)
 - 1-dim: `1dh 101 'muj hist' 100 10. 110.`
 - 2-dim: `2dh 999 'x-y' 100 0. 1. 50 100. 200.`
 - profil: `prof 1001 'profil fce' 50 -2. 9. !!`
- Založení histogramu v naší vlastní Fortran-funkci:
 - 1-dim: `call hbook1(101,'muj hist',100,10.,110.,0.)`
 - 2-dim: `call hbook2(999,'x-y',100,0.,1.,50,100.,200.,0.)`
 - profil: `call hbprof(1001,'profil fce',50,-2.,9.,ymin,ymax,'')`

Vlastní funkci zavoláme z **COMISu**: `call fce.f` Pozor – jméno souboru musí být totožné se jménem funkce !!!


Paw: histogramy (2)

- Plnění histogramu:
 - v rámci Fortran-funkce: `call hf1(101,xx,w)`
 - pomocí vektorů: `ve/hfill xx 105`
 - projekcí z ntuplu...
- Čtení ze/zápis do souboru:
 - `hi/file 1 histo.dat ; hi/pl 10`
 - `hi/file 99 newhist.dat 1024 'N'; hrout 10; close 99`
- Jednoduché operace:
 - zobrazení: `hi/plot 101` , seznam histogramů: `hi/list`
 - kopírování: `hi/copy 101 205`
 - mazání (`hi/del 101`), sčítání (`hi/op/add 101 102 103`), odčítání (`hi/op/sub 101 102 103`), ...

Paw: histogramy (3)

- Fitování histogramu: podobně jako v případě vektorů buď pomocí vestavěných funkcí, nebo uživatelskou funkcí. Příklady:
 - Gauss: `hi/fit 101 g`
 - Gauss, finta 1: `ve/cre gg(3) r; hi/fit 101 g ! 0 gg` (výsledek fitu se objeví ve vektoru `gg`, ale žádné počáteční hodnoty)
 - Gauss, finta 2: `ve/cre gg(3) r 10. 1. 0.5; hi/fit 101 g ! 3 gg` (fit s nastavenými počátečními hodnotami parametrů)
 - uživatelská funkce: stejně jako v případě vektorů...
- Dodatek k fitování: jednotlivé parametry je možné omezit (min a max hodnota) či zcela zafixovat. Podrobnosti viz manuál.

Paw: statistika a výsledky fitů

- Oba druhy informací lze zobrazit spolu s daným histogramem či vektory:
 - statistické informace o histogramu (počet případů, střední hodnota, RMS, ...):
 - zapnutí/vypnutí: `opt stat / opt nsta`
 - nastavení zobrazených položek: `set stat 1100111`
 - výsledky fitu:
 - zapnutí/vypnutí: `opt fit / opt nfit`
 - nastavení zobrazených položek: `set fit 101`
- **Příklady:** viz strana 76, 77 

Paw: základy Ntuplů (1)

- Jedná se o "multi-dimenzionální tabulky". Lze zkoumat závislost jakékoli proměnné na libovolné kombinaci ostatních proměnných – **to je pravá analýza dat !!**
- Dva základní typy:
 - **RWN** (Row-Wise Ntuple): obvykle se používá pro malý počet proměnných
 - **CWN** (Column-Wise Ntuple): velké datové soubory, složen z jednotlivých bloků

S oběma typy se zachází stejně, rozdíl je pouze ve vytváření.
- Jednoduché **RWN**-tuply (vytvořené přímo v Paw):
 - založení: `nt/cre 11 'pulse shape' 2 !! time samp`
 - načtení údajů: `nt/read 11 soubor.dat` (předpokládá se soubor o 2 sloupcích reálných čísel)

Paw: základy Ntuplů (2)

- Složitější Ntuply se vytvářejí obvykle v různých programech (Fortran, C) a otvírají se v Paw pomocí
 - `hi/file 1 ntuple.dat`
- Základní zacházení s ntuplem:
 - zjištění vlasností: `nt/pri 1 1`
 - zobrazení proměnné: `nt/plot 1 1.samp`
 - zobrazení proměnné s výběrem: `nt/plot 1 1.samp time>90.`
 - použití tzv. cutu: `cut 99 (time>0.).and.(samp<100.) ;`
`nt/plot 1 1.samp $99`
 - závislost 2 proměnných: `nt/plot 1 1.samp%time`
 - profil 2 proměnných: `nt/plot 1 1.samp%time $99 ! ! ! prof`


Automaticky se založí histogram (1d, 2d, profil) a naplní se.

Paw: základy Ntuplů (3)

- Složitější operace s ntuply: založení vlastního histogramu (volíme si vlastní rozsah, binning, ...), požadovanou závislost do něj projektujeme:
 - jedna proměnná: `1dh 101 'veličina time' 100 0. 100. ;
nt/proj 101 11.time $99`
 - 2 proměnné: `2dh 102 'samp-time' 100 0. 100. 50 -10. 30. ;
nt/proj 102 11.samp%time`
 - profil 2 proměnných: `prof 103 'samp-time' 100 0.
100. !! ; nt/proj 103 11.samp%time $99`

Další finty: zpracování pouze části ntuplů (zadáva se počet prvků a číslo prvního prvku, ...) - viz manuál **Paw**
- Tím zobrazíme požadovanou závislost do histogramu a pak ji můžeme zkoumat (fitovat různými funkcemi atd.)

Paw: pár slov o cotech

- Cut = podmínka, za které je daný případ (řádka) v ntuplech zpracována (např. zanesena do histogramu)
- Základní operace:
 - vytvoření: `cut 45 (time>-10).or.(samp<100.)`
 - zobrazení: `cut 45 ; cut 0`
 - smazání: `cut 45 - ; cut 0 -`
- Cuty lze (téměř libovolně) kombinovat:
 - k dispozici logické operátory: **AND, OR, NOT**, používá se Fortran syntaxe:
 - `cut 91 time>0 ; cut 92 samp<100; cut 93 (10<time<20.)`
 - `cut 99 $91.and.$92.and.(.not.($93))`
- **Příklad:** viz strana 82 

Ntuply a funkce (1)

- Občas potřebujeme pro každý případ v ntuplu udělat nějakou operaci (určit aritmetický průměr z hodnot pole, vybrat maximální hodnotu z několika proměnných, ...), kterou nelze jednoduše zapsat do příkazu v Paw => **použití Fortran (případně C) funkce**
- Jak na to:
 - nejdříve musíme vytvořit skelet funkce pomocí
`nt/uwfunc 99 funkce.f` | vytvoří soubor funkce.f pro ntuple č.99
a pak do ní dopíšeme náš kód.
 - použití:
 - kreslení složitých funkcí: `nt/plot 99.funkce.f`
 - použití složitějších cutů: `nt/plot 99.x y>0.and.funkce.f>1.`
 - provedení akce pro každý případ: `nt/loop 99 funkce.f`

Ntupy a funkce (2)

- Do takto vytvořených funkcí / částí programu lze dopsat libovolný kód. Funkce může mít i parametry, se kterými se volá (musíme je ručně dopsat do hlavičky) – hodí se např. pro počítání energie z určitého kanálu.
- **Příklad:** využijte Fortran/C funkci při zobrazování závislosti $\text{sqrt}(x^{**2}+y^{**2})$ v předchozím příkladě (viz strana 82).



Vytváření Paw-Ntuplů v externích programech (1)

- Zápis/výstup dat z našich vlastních programů do ntuplu, případně histogramů. Trochu se liší pro **RWN** a **CWN**.
- Potřebujeme procedury/funkce z knihovny **HBOOK**

http://paw.web.cern.ch/wwwasdoc/hbook_html3/hboomain.html

některé už známe (založení a plnění histogramů – strana 153) 

- **RWN** (uvedme jen Fortran-kód, v C analogicky):
 - hlavička: definice common-bloků a proměnných ntuplu:

```

common/PAWC/H(10000) ! prostor pro objekty Paw
parameter(ncol=5)    ! počet proměnných (sloupců) v RWN
real xtuple(ncol)   ! pole hodnot v jedné řádce RWN
character*5 chtags(ncol) ! jména proměnných RWN
data chtags/'event','x_val','y_val','z_val','trig'/

```

Vytváření Paw-Ntuplů v externích programech (2)

- otevření nového souboru a založení ntuplu:

```
call hlimit(10000)
```

! rezervace paměti pro objekty Paw

```
call hropen(99,'ntup','mujtup.dat','N',1024,istat) ! otevření souboru
```

```
call hbookn(10,'muj nazev',ncol,'ntup',100,htags) ! založení RWN
```

- uložení případu ("jedné řádky") do RWN:

```
call hfn(10,xtuple)
```

Obvykle v cyklu, předtím musíme prvky xtuple(i) naplnit.

- uzavření vstupu/výstupu (stejně pro oba druhy ntuplů):

```
call hrout(10,istat,'')
```

```
call hrend('ntup')
```

```
close(99)
```

Vytváření Paw-Ntuplů v externích programech (3)

- **CWN:** obecnější filozofie – obsahuje 1 či více common-bloků, proměnná může být i pole (dokonce s proměnnou délkou, ale to je specialita – viz manuál **HBOOK**)
 - hlavička: definice common-bloků a proměnných ntuplu:
 - `common/PAWC/H(10000)` ! prostor pro objekty Paw
 - `integer evt,trig`
 - `real xyz(3)`
 - `common/mujcwn/evt,xyz,trig` ! blok proměnných v ntuplu
 - otevření nového souboru a založení ntuplu:
 - `call hlimit(10000)` ! rezervace paměti pro objekty Paw
 - `call hropen(99,'ntup','mujtup.dat','N',1024,istat)` ! otevření souboru
 - `call hbnt(10,'muj nazev','')` ! založení CWN
 - `call hbname(10,'mujcwn',evt,'event:I, vtx(3):R, trig:I')` ! proměnné s definovaným typem (integer, real)

Vytváření Paw-Ntuplů v externích programech (4)

- uložení případu ("jedné řádky") do CWN:

```
call hfnt(10)
```

Obvykle v cyklu, předtím musíme proměnné našeho common-bloku naplnit.

- uzavření vstupu/výstupu – stejně jako v případě RWN:

```
call hrout(10,istat,'')
```

```
call hrend('ntup')
```

```
close(99)
```

Grafický výstup z Paw

- **Paw** umí exportovat obrázky (tj. grafy, histogramy, ...) ve formátech **PS**, **EPS**, **GIF**. Jak to provést:
 - nastavit, aby se poslední (všechny) obrázky ukládaly do paměti: `opt zfl1` (`opt zfl`)
 - výstup do souboru: `hi/pl 101 ; pic/print obr.ps` (formát je dán příponou)