ComputeCanada

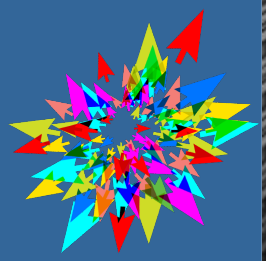
Ontario Summer School on HPC

- **LINUX/SHELL programming**

Isaac Ye

HPTC @York University

compute canada | calcul canada



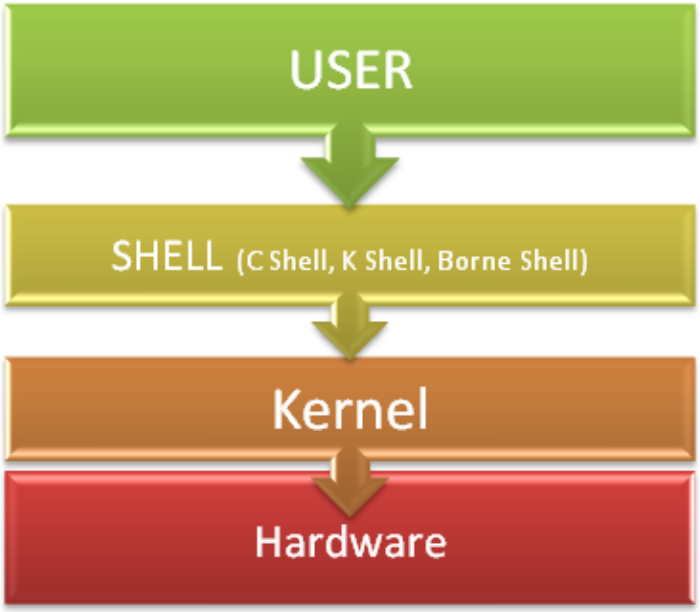
Overview

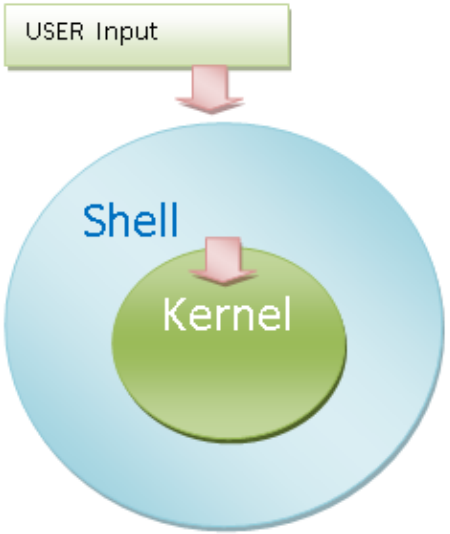
Session II
(SHELL programming)

1) SHELL basics
2) Programming I
    1) Variables
    2) Quote
    3) Environment variables
    4) Read/Substitution
    5) Arithmetic calculation
3) Programming II
    1) Conditional Statement
    2) SHELL parameters
    3) Iteration statement
4) Makefile

What is Shell?

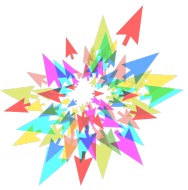
- Shell is the interface between end user and the system





The Shell of LINUX

- LINUX has a variety of different shells:
  - Bourne shell (sh), C shell (csh), Korn shell (ksh), TC shell (tcsh), Bourne Again shell (bash).
- Certainly the most popular shell is “bash”(Default at SHARCNET). Bash is an sh-compatible shell that incorporates useful features from the Korn shell (ksh) and C shell (csh).
- It is intended to conform to the IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools standard.
- It offers functional improvements over sh for both programming and interactive use.



History

- **1979**: Bourne Shell (/bin/sh)
  - first UNIX shell
  - still widely used as the LCD of shells
- **1981**: C shell (csh)
  - part of BSD UNIX
  - commands and syntax which resembled C
  - introduced aliases, job control
- **1988**: Bourne again shell (bash)
  - developed as part of GNU project (default shell in Linux)
  - incorporated much from csh, ksh and others
  - introduced command-line editing, functions, integer arithmetic, etc.

The first bash program (Hello World)

- There are two major text editors in LINUX: vi, emacs (or xemacs).
- So fire up a text editor; for example:

```
$ vi
```

and type the following inside it:

```
#!/bin/bash
echo "Hello World"
```

- The first line tells LINUX to use the bash interpreter to run this script. We call it hello.sh. Then, make the script executable:

```
$ chmod 700 hello.sh
$ ./hello.sh
$ Hello World
```

The second bash program

- We write a program that copies all files into a directory, and then deletes the directory along with its contents. This can be done with the following commands:

```
$ mkdir trash
$ cp * trash
$ rm -rf trash
```

- Instead of having to type all that interactively on the shell, write a shell program instead:

```
$ cat trash.sh
    #!/bin/bash
    #this script deletes some files
    mkdir trash
    cp * trash
    rm -rf trash
    echo "Deleted all files!"
```

Variables

- We can use variables as in any programming languages. Their values are always stored as strings, but there are mathematical operators in the shell language that will convert variables to numbers for calculations.
- We have no need to declare a variable, just assigning a value to its reference will create it.
- Example

```
#!/bin/bash
STR="Hello World!"
echo $STR
```

- Line 2 creates a variable called STR and assigns the string *"Hello World!"* to it. Then the value of this variable is retrieved by putting the '$' in at the beginning.

Warning !

- The shell programming language does not type-cast its variables. This means that a variable can hold number data or character data.

```
Count = 0
Count = Sunday
```

- Switching the TYPE of a variable can lead to confusion for the writer of the script or someone trying to modify it, so it is recommended to use a variable for only a single TYPE of data in a script.

Single and Double Quote

- When assigning character data containing spaces or special characters, the data must be enclosed in either single or double quotes.
  - Using double quotes to show a string of characters will allow any variables in the quotes to be resolved

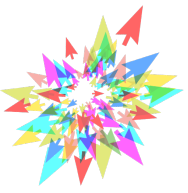
```
$ var="test string"
$ newvar="Value of var is $var"
$ echo $newvar

Value of var is test string
```

- Value of var is test string
  - Using single quotes to show a string of characters will not allow variable resolution

```
$ var='test string'
$ newvar='Value of var is $var'
$ echo $newvar

Value of var is $var
```



The export command

- The export command puts a variable into the environment so it will be accessible to child processes. For instance:

```
$ x=hello
$ bash                    #Run a child Shell
$ echo $x
                           #Nothing in x
$ exit                    #Return to parent
$ export x
$ bash
$ echo $x

hello                     #It's there
```

- If the child modifies x, it will not modify the parent's original value. Verify this by changing x in the following way:

```
$ x=ciao
$ exit
$ echo $x

hello
```

Exercise #1

1. Make a bash script for the following features
    1. Echo ‘Hello World’ using STR=“Hello World” variable
    2. Refer to slide 10, make sure the difference between ‘’ and “”
2. Make a trash.sh based on the slide 7
3. Open 4 child shells and make sure where you are in.

note) You can copy all exercise scripts

- cp /home/isaac/ss15_s2.gz ~
- tar zxvf ss15_s2.gz

Exercise #1 (Cont'd)

- How to check which shell layer you are in?

```
[isaac@orc-login1:~] bash
[isaac@orc-login1:~] ps
  PID TTY          TIME CMD
17800 pts/13   00:00:00 bash
18468 pts/13   00:00:00 bash
18497 pts/13   00:00:00 ps
[isaac@orc-login1:~] bash
[isaac@orc-login1:~] ps
  PID TTY          TIME CMD
17800 pts/13   00:00:00 bash
18468 pts/13   00:00:00 bash
18498 pts/13   00:00:00 bash
18526 pts/13   00:00:00 ps
[isaac@orc-login1:~] exit
exit
[isaac@orc-login1:~] ps
  PID TTY          TIME CMD
17800 pts/13   00:00:00 bash
18468 pts/13   00:00:00 bash
18527 pts/13   00:00:00 ps
[isaac@orc-login1:~] exit
exit
[isaac@orc-login1:~] ps
  PID TTY          TIME CMD
17800 pts/13   00:00:00 bash
18574 pts/13   00:00:00 ps
```

Environmental Variables

- There are two types of variables:
  - Local variables
  - Environmental variables

- Environmental variables are set by the system and can usually be found by using the env command. Environmental variables hold special values. For instance:

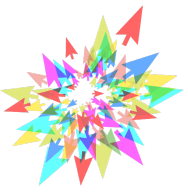
```
$ echo $SHELL
/bin/bash
$ echo $PATH
/opt/sharcnet/openmpi/1.8.3/gcc/bin:/opt/sharcnet/gcc/4.8.2/bin
```

Environmental Variables

- Environmental variables are defined in /etc/profile, /etc/profile.d/ and ~/.bash_profile. These files are the initialization files and they are read when bash shell is invoked.

- When a login shell exits, bash reads ~/.bash_logout

- The startup is more complex; for example, if bash is used interactively, then /etc/bashrc or ~/.bashrc are read. See the man page for more details.



Environmental Variables

- HOME: The default argument (home directory) for cd.
- PATH: The search path for commands. It is a colon-separated list of directories that are searched when you type a command.

- Usually, we type in the commands in the following way:

```
$ ./command
```

- By setting PATH=$PATH:. our working directory is included in the search path for commands, and we simply type:

```
$ command
```

Environmental Variables

- If we type in

```
$ mkdir ~/bin
```

and we include the following lines in the ~/.bashrc:

```
PATH=$PATH:$HOME/bin
```

```
$ source ~/.bashrc
```

- We obtain that the directory /home/userid/bin is included in the search path for commands.

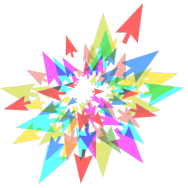
Environment Variables

- LOGNAME (USER): contains the user name
- HOSTNAME: contains the computer name.
- PS1: sequence of characters shown before the prompt

```
\t          hour
\d          date
\w          current directory
\W          last part of the current directory
\u          user name
\h          hostname
\$          prompt character
```

- Example:

```
$ PS1 = 'hi \u *'
hi $USERID* __
```

Exercise #2

- Put following log-out comment into `~/.bash_logout`

  `"Good-bye $USER!"`

- Add `/home/$USER/ss15_2/Ex2` directory into the existing path
- Copy trash.sh from Ex1 into `/home/$USER/ss15_2/Ex2` and run it without (./)
- Change the prompt setting PS1 environment variable by export command

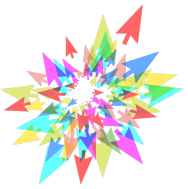
  `(export PS1='I am \u @\w')`

Read command

- The read command allows you to prompt for input and store it in a variable.
- Example:

```
#!/bin/bash
echo -n "Enter name of file to delete: "
read filename
echo "Type 'y' to remove it, 'n' to change your mind ... "
rm -i $filename
echo "Done! That was YOUR decision!"
```

- Line 2 prompts for a string that is read in line 3. Line 4 uses the interactive remove (`rm -i`) to ask the user for confirmation.



Command Substitution

- The backquote “ ` ” is different from the single quote “ ´ ”. It is used for command

substitution: `command`

```
$ LIST=`ls`
$ echo $LIST
hello.sh read.sh

$ PS1="`pwd`>"
/home/userid/work> _
```

- We can perform the command substitution by means of `$(command)`

```
$ LIST=$(ls)
$ echo $LIST
hello.sh read.sh
```

Exercise #3 (Cont'd)

```
#!/bin/bash
echo "Enter the directory you want to backup"
read dir
BCKFILE=backup-$(date +%d-%m-%y).tar.gz
BCKUP=/tmp/$BCKFILE
tar -cPzf $BCKUP $dir
echo "Compression is done! Enter the destination"
read newdir
mv $BCKUP $newdir
echo "Your backup file $BCKFILE is at $newdir"
```

Arithmetic Evaluation

- The let statement can be used to do mathematical functions:

```
$ let X=10+2*7
$ echo $X
24
$ let Y=X+2*4
$ echo $Y
32
```

- An arithmetic expression can be evaluated by `$[expression]` or `$((expression))`

```
$ echo "$((123+20))"
143
$ VALORE=$[123+20]
$ echo "$[123*$VALORE]"
17589 32
```

Arithmetic Evaluation

- Available operators: +, -, /, *, %, ++, —, **
- Example

```
$ cat arithmetic.sh
#!/bin/bash
echo -n "Enter the first number: "; read x
echo -n "Enter the second number: "; read y
add=$(($x + $y))
sub=$(($x - $y))
mul=$(($x * $y))
div=$(($x / $y))
mod=$(($x % $y))
# print out the answers:
echo "Sum: $add"
echo "Difference: $sub"
echo "Product: $mul"
echo "Quotient: $div"
echo "Remainder: $mod"
```

Case Statement

- Used to execute statements based on specific values. Often used in place of an if statement if there are a large number of conditions.
- Value used can be an expression
- each set of statements must be ended by a pair of semicolons;
- a *) is used to accept any value not matched with list of values

```
case $var in
    val1)
        statements;;
    val2)
        statements;;
    *)
        statements;;
esac
```

Example (case.sh)

```
$ cat case.sh
#!/bin/bash
echo -n "Enter a number 1 < x < 10: "
read x
      case $x in
         1) echo "Value of x is 1.";;
         2) echo "Value of x is 2.";;
         3) echo "Value of x is 3.";;
         4) echo "Value of x is 4.";;
         5) echo "Value of x is 5.";;
         6) echo "Value of x is 6.";;
         7) echo "Value of x is 7.";;
         8) echo "Value of x is 8.";;
         9) echo "Value of x is 9.";;
         0 | 10) echo "wrong number.";;
         *) echo "Unrecognized value.";;
    esac
```

Exercise #3-1

1. Mae a ‘conv.sh’ script to convert km —> m and m—> cm.

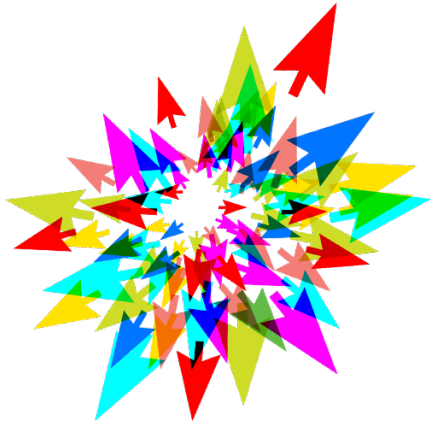
   Here is the expected run

```
$ ./conv.sh
Choose mode: (1) km --> m (2) m --> cm
1
Enter speed
10
10 km is 10000 m
$ ./conv.sh
Choose mode: (1) km --> m (2) m --> cm
2
Enter speed
1
1 m is 100 cm
```

Exercise #3-2

1. Make a ‘runbackup.sh’ script to backup a directory into the designated location
   - Read the target directory
   - It should ‘tar’ all files in the target directory and compress it as zip file
     - Hint) tar -zcPvf /tmp/**backup-$(date +%d-%m-%y).tar.gz**
   - Read the destination directory
   - mv the file to the destination directory
   - Print “`Your backup file backup-DD-MM-YY.tar.gz is at /work/USERID`”

compute canada | calcul canada



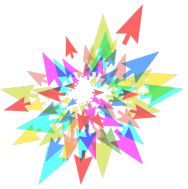
Break!

Conditional Statements

- Conditionals let us decide whether to perform an action or not, this decision is taken by evaluating an expression. The most basic form is:

```
if [ expression ];
     then
          statements
     elif [ expression ];
     then
          statements
     else
          statements
fi
```

- the elif (else if) and else sections are optional
- Put spaces after [and before], and around the operators and operands.



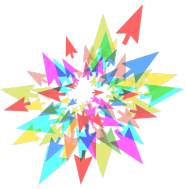
Expressions

- An expression can be: String comparison, Numeric comparison, File operators and Logical operators and it is represented by [expression]:

```
 =   compare if two strings are equal
!=   compare if two strings are not equal
-n   evaluate if string length is greater than zero
-z   evaluate if string length is equal to zero
```

- String Comparisons:

```
[ s1  = s2 ] (true if s1 same as s2, else false)
[ s1 != s2 ] (true if s1 not same as s2, else false)
[ s1 ]       (true if s1 is not empty, else false)
[ -n s1 ]    (true if s1 has a length greater then 0, else false)
[ -z s2 ]    (true if s2 has a length of 0, otherwise false)
```



Expressions

- Number Comparisons:

```
-eq  compare if two numbers are equal
-ge  compare if one number is greater than or equal to a number
-le  compare if one number is less than or equal to a number
-ne  compare if two numbers are not equal
-gt  compare if one number is greater than another number
-lt  compare if one number is less than another number
```

- Examples:

```
[ n1 -eq n2 ]      (true if n1 same as n2, else false)
[ n1 -ge n2 ]      (true if n1greater then or equal to n2, else false)
[ n1 -le n2 ]      (true if n1 less then or equal to n2, else false)
[ n1 -ne n2 ]      (true if n1 is not same as n2, else false)
[ n1 -gt n2 ]      (true if n1 greater then n2, else false)
[ n1 -lt n2 ]      (true if n1 less then n2, else false)
```

Examples

```
$ cat user.sh
 #!/bin/bash
 echo -n "Enter your login name: "
 read name
 if [ "$name" = "$USER" ];
     then
         echo "Hello, $name. How are you today ?"
     else
         echo "You are not $USER, so who are you ?"
 fi
```

Exercise #4

- Make a bash script with the following features
  - Ask to enter/read login name and if the name is same with the userid print “Hello Isaac! How are you today?” and move on to the next step, otherwise print “You are not Isaac, Bye!” and break. (Hint: exit 1 for termination)
  - Ask to enter a number from 1 to 10.
  - If the number is greater than 10, print “You entered $value which is over the limit” and stop
  - If the number is less than 1, write ““You entered $value which is under the limit”” and stop

Exercise #4 (Cont'd)

```
#!/bin/bash
echo -n "Enter your login name: "
read name
if [ "$name" = "$USER" ];
then
     echo "Hello, $name. How are you today ?"
else
     echo "You are not $USER, Bye~!"
     exit 1
fi

echo "Enter a number 1 to 10"
read num
if [ $num -lt 10 ]; then
     if [ $num -gt 1 ]; then
          echo "You enter a right number 1< $num <10"
     else
          echo "You enter $num which is under the limit"
     fi
else
     echo "You enter $num which is over the limit"
fi
```

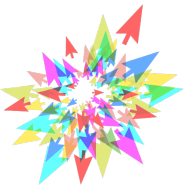
Expressions

- Files operators:

```
-d   check if path given is a directory
-f   check if path given is a file
-e   check if file name exists
-r   check if read permission is set for file or directory
-s   check if a file has a length greater than 0
-w   check if write permission is set for a file or directory
```

- Examples:

```
[ -d fname ] (true if fname is a directory, otherwise false)
[ -f fname ] (true if fname is a file, otherwise false)
[ -e fname ] (true if fname exists, otherwise false)
[ -s fname ] (true if fname length is greater then 0, else false)
[ -r fname ] (true if fname has the read permission, else false)
[ -w fname ] (true if fname has the write permission, else false)
```



Example

```
#!/bin/bash
if [ -f /etc/fstab ];
     then
     cp /etc/fstab .
          echo "Done."
     else
     echo "This file does not exist."
          exit 1
     fi
```

Expressions

- Logical operators:

```
!    negate (NOT) a logical expression
-a   logically AND two logical expressions
-o   logically OR two logical expressions
```

- Example:

```
#!/bin/bash
     echo -n "Enter a number 1 < x < 10:"
     read num
     if [ "$num" -gt 1 -a "$num" -lt 10 ];
     then
          echo "You enter a right number 1< $num <10"
     else
          echo "You enter $num which is out of the range"
     fi
```

Expressions

- Logical operators:

```
&&   logically AND two logical expressions
||   logically OR two logical expressions
```

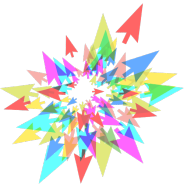
- Example:

```
#!/bin/bash
     echo -n "Enter a number 1 < x < 10: "
     read num
     if [ "$number" -gt 1 ] && [ "$number" -lt 10 ];
     then
          echo "You enter a right number 1< $num <10"
     else
          echo "You enter $num which is under the limit"
     fi
```

Example

```
#!/bin/bash
echo "Enter a path: "; read dir

if [ -d $dir ]; then
     cd $dir
     echo "I am in $dir and it contains"; ls
else
     echo "The directory $dir does not exit";
     exit 1
fi
```

Shell Parameters

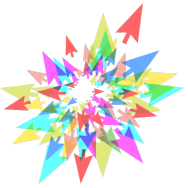
- Positional parameters are assigned from the shell's argument when it is invoked. Positional parameter "N" may be referenced as "${N}", or as "$N" when "N" consists of a single digit.

```
$#   is the number of parameters passed
$0   returns the name of the shell script running as well as its
     location in the file system
$*   gives a single word containing all the parameters passed
     to the script
$@   gives an array of words containing all the parameters
     passed to the script
```

Shell Parameters (Cont'd)

```
$ cat sparameters.sh
#!/bin/bash
echo "$# - number of parameters"
echo "$0 - name of the script"
echo "$1 - first parameters"
echo "$2 - second parameters"
echo "$* - single word with all parameters"
echo "$@ - single array with all parameters"


$ ./sparameters.sh A B C
3 - number of parameters
sparameters.sh - name of the script
A - first parameters
B - second parameters
A B C - single word with all parameters
A B C - single array with all parameters
```



Example (case.sh)

```
$ cat case.sh
#!/bin/bash
echo -n "Enter a number 1 < x < 10: "
read x
      case $x in
         1) echo "Value of x is 1.";;
         2) echo "Value of x is 2.";;
         3) echo "Value of x is 3.";;
         4) echo "Value of x is 4.";;
         5) echo "Value of x is 5.";;
         6) echo "Value of x is 6.";;
         7) echo "Value of x is 7.";;
         8) echo "Value of x is 8.";;
         9) echo "Value of x is 9.";;
         0 | 10) echo "wrong number.";;
         *) echo "Unrecognized value.";;
    esac
```

Iteration Statements

- The for structure is used when you are looping through a range of variables.

```
for var in list
    do
      statements
    done
```

- statements are executed with var set to each value in the list.
- Example

```
#!/bin/bash
    let sum=0
    for num in 1 2 3 4 5
       do
         let "sum = $sum + $num"
       done
    echo $sum
```

Iteration Statements

```
#!/bin/bash
for x in paper pencil pen
  do
       echo "The value of variable x is: $x"
       sleep 1
    done

$ ./iterfor.sh
The value of variable x is: paper
The value of variable x is: pencil
The value of variable x is: pen
```

Exercise #5

- Write a bash script to move files into the designated location and change the permission of all files. The script should have the following feature
  - Obtain the target directory as ‘argument’ (ex, ./runbaskcup.sh /home/isaac/test)
  - If no argument is provided, print “No target information provided” and break
  - Check if there is a backup directory in /scratch/$USER/old and make a directory if necessary.
  - move into ss15/BCK directory (from the example gzip file)
  - Using ‘for’ iteration, mv each file into /scratch/$USER/old and change the permission into ‘700’
  - Once it is done, list /scratch/$USER/old

Exercise #5 (Cont'd)

```
#!/bin/bash
BCKDIR=/scratch/$USER/old

if [ $# -eq 0 ]; then
     echo "No target information provided"
     exit 1
fi
if [ ! -d  "$BCKDIR" ]; then
     mkdir -p "$BCKDIR"
fi

echo "The following files in $1 will be saved in the old
directory"

cd $1
for file in *
do
     cp $file "$BCKDIR"
     chmod 700 "$BCKDIR/$file"
done

ls -l "$BCKDIR"
```

Makefile

- Compiling the source code files can be tiring, especially when you have to include several source files and type the compiling command every time you need to compile. Makefiles are the solution to simplify this task.

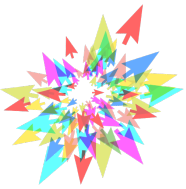
- Example:

  main.cpp, hello.cpp, factorial.cpp, functions.h

  (You can find these in ss15/Ex3)

- Make sure your module:

  intel/15.0.2, mkl/11.1.4



Simple compilation

- The trivial way to compile is

```
$ icc main.cpp hello.cpp factorial.cpp -o hello
$ ls -l hello
-rwxrwxr-x 1 isaac isaac 25616 May 24 00:19 hello
```

- What if there are thousands of source code?
- Makefile is a good solution to manage a large project with many different compilation options.

Makefile #1

- General syntax:

```
target [target …] : [dependent …]
      [command …]
```

- The simplest one would be

```
$ cat Makefile
run: main.cpp hello.cpp factorial.cpp functions.h
      icc -o run main.cpp hello.cpp factorial.cpp -I.
```

- Be careful to have a tab before ‘icc’ as a separator

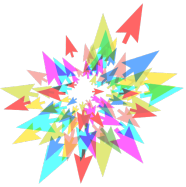
```
$ make
icc -o run main.cpp hello.cpp factorial.cpp -I.
$ ./run
Hello World!
The factorial of 5 is 120
```

Makefile variables

- There are built-in variables for makefile

C	C++	FORTRAN	ARCHIVE
CC	CXX	FC	AR
CFLAGS	CXXFLAGS	FFLAGS	ARFLAGS

```
LIBS: library link
INCLUDE: header file location
DEPS: dependent files
```



Makefile #2

- Makefile variables can be used as followed

```
$ cat Makefile
CC = icc
CFLAGS = -I.

run: main.cpp hello.cpp factorial.cpp functions.h
        $(CC) -o run main.cpp hello.cpp factorial.cpp $(CFLAGS)

clean:
    rm -rf run *.o *.core
```

- 'clean' target is used to remove the previous outputs and compilation error cores.

Makefile #3

- Makefile has Macros:

```
$@ : name of the file to be made
$? : name of the changed dependents
$< : name of the related file that caused the action
$* : the prefix shared by target and dependent files
$^ : name of all dependent files separated by spaces
```

- For example, we could use this as follows

```
run: main.cpp hello.cpp factorial.cpp functions.h
        $(CC) $(CFLAGS) $? -o $@
```

- Implicit rule for the construction of .o (obj) files out of .cpp (src)

```
.o.cpp:
        $(CC) $(CFLAGS) -c $<
```

Makefile #3 (Cont'd)

- Makefile using the special macros:

```
$ cat Makefile
CC = icc
CFLAGS = -I.
DEPS = functions.h

%.o: %.cpp $(DEPS)
        $(CC) $(CFLAGS) -c -o $@ $<
run: main.o hello.o factorial.o
        $(CC) $(CFLAGS) -o $@ main.o hello.o factorial.o

clean:
     rm -rf run *.o *.core
$ make
icc -I. -c -o main.o main.cpp
icc -I. -c -o hello.o hello.cpp
icc -I. -c -o factorial.o factorial.cpp
icc -I. -o run main.o hello.o factorial.o
```

Makefile #4

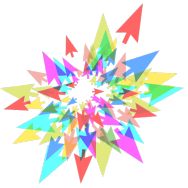
- Makefile using the special macros:

```
$ cat Makefile
CC = icc
CFLAGS = -I.
DEPS = functions.h
OBJ  = main.o hello.o factorial.o

%.o: %.cpp $(DEPS)
        $(CC) $(CFLAGS) -c -o $@ $<
run: $(OBJ)
        $(CC) $(CFLAGS) -o $@ $^
clean:
     rm -rf run *.o *.core
```

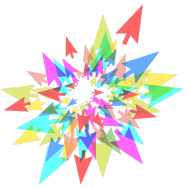
Exercise (Job Submission)

- Monte Carlo-type simulations
  - once the experiment is designed and parameters set, we need to submit vast numbers of jobs to the queue
  - can speed this process dramatically using a script to do the submissions
- Notes:
  This is most easily accomplished by having the program take its parameters either on the command-line, or from a file that is specified on the command-line; similarly, output should either go to stdout or to a file specified on the command-line
  - makes it easy to submit from the same directory
  - “for each set of parameters, submit a job with those parameters”

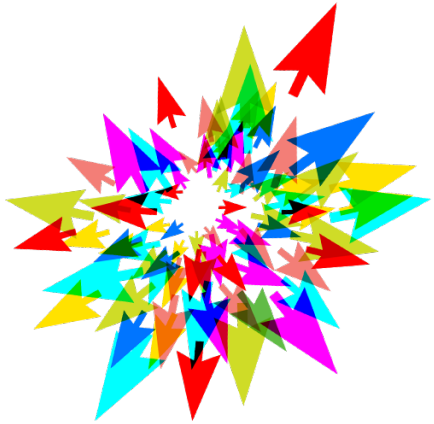


Exercise (Job Submission)

```
#!/bin/bash
#
# DEST_DIR is the base directory for submission # EXENAME is the name
of the executable
#
DEST_DIR=/work/$USER/MC
EXENAME=hello_param
cd ${DEST_DIR}
for trial in 1 2 3 4 5; do
    for param in 1 2 3; do
      echo "Submitting trial_${trial} - param_${param}..."
      sqsub -q serial -o OUTPUT-${trial}.${param}.txt ./${EXENAME} $
      {trial}-${param}
    done
done
```

compute canada | calcul canada



Thank you !

For further questions,