University of Maryland at College Park

# A Brief Introduction to Shell Scripts, Bash, and DICE

Shuvra S. Bhattacharyya
Dept. of ECE

Version: Jan. 19, 2024

# Scripts

- Motivation: Using scripts is an important method for improving software productivity.
- A script can be viewed as way to connect groups of programs that may be written in different languages [Loukides 1997].
- Languages for writing scripts usually provide great flexibility in how groups of programs can be connected.

# Bash

- Bash = “Bourne again Shell”
  - GNU replacement for the Bourne shell
- As with other shells, you can use Bash as an interactive command interpreter (at the shell prompt) or as a programming language (using Bash scripts)
- As the first line in a Bash script, use:

  ```
  #!/usr/bin/env bash
  ```

  - This uses the default version of Bash in your environment
- Fundamental Bash/UNIX commands and utilities include:

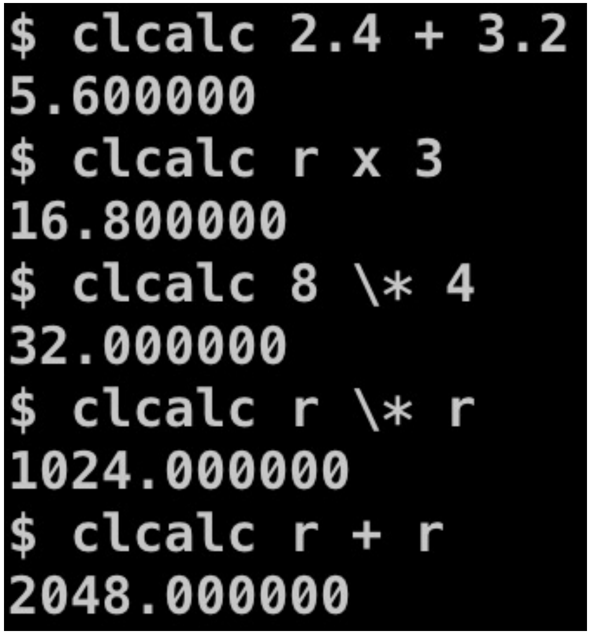
  ```
  cat/more/less, cd, cp, echo, grep, ls, man,
  mkdir, mv, rm, rmdir
  ```

- Useful Bash reference: C. Newham and B. Rosenblatt. *Learning the Bash shell*. O'Reilly & Associates, Inc., third edition, 2005.

# Script Example: Preview of Some Specific Features and Conventions

- Using **`$#`** to get the argument count of a script or function
- Exiting with a non-zero status upon error detection
- **`$UXTMP`**: DICE user space for storing temporary files. Can be cleaned using **`dxclntmp`**.
- Using a Bash script to "wrap" one or more binary executables
- Using **`$?`** to get return status/value
- Using **`[ -f <filename> ]`** to test for file existence
- Using **`` `<command>` ``** to capture standard output
- Displaying error messages using **`>&2`** (redirection to standard error)

# Calculator Example



```
$ clcalc 2.4 + 3.2
5.600000
$ clcalc r x 3
16.800000
$ clcalc 8 \* 4
32.000000
$ clcalc r \* r
1024.000000
$ clcalc r + r
2048.000000
```

```
# A command line calculator (clcalc) as a
# basic Bash programming example.
#
# In the script name, "cl" stands for
# "command line".
#
# For multiplication, use \* or x (lower
# case) as the operator.
#
# Valid operators are: +, -, /, \*, x.
#
# Numbers can be used as operands, as well
# as the special operand "r", which refers
# to the last result computed by clcalc.
```

This example illustrates methods for robust script implementation, which is important in complex/team projects.

S. S. Bhattacharyya, *A Brief Introduction to Shell Scripts, Bash, and DICE*, University of Maryland at College Park, Jan., 2024, slide 5.

# Script Example 1

```
#!/usr/bin/env bash
# A command line calculator (clcalc) as a basic
# Bash programming example.
# For multiplication, use \* or x (lower case) as the operator.
# Valid operators are: +, -, /, \*, x.
# Numbers can be used as operands, as well as the special
# operand "r",which refers to the last result computed by
# clcalc.

lxprog="clcalc"
lxtmp="$UXTMP/$lxprog-tmp.txt"
lxoperand1=""
lxoperand2=""
lxresultfile="$UXTMP/$lxprog-result.txt"

if [ $# -ne 3 ]; then
    >&2 echo "$lxprog error: arg count"
    exit 1
fi
```

# Variables in Bash

Adapted From https://tldp.org/LDP/abs/html/untyped.html (visited on 02/01/2022):

- Unlike many other programming languages, Bash does not segregate its variables by "type."
- Bash variables are character strings.
- But, depending on context, Bash permits arithmetic operations and comparisons on variables.
  - The determining factor is whether the value of a variable contains only digits.

# Script Example 2

```
if [ "$1" = "r" ]; then
    if ! [ -f "$lxresultfile" ]; then
        >&2 echo "$lxprog error: no result available"
        exit 1
    fi
    lxoperand1=`cat "$lxresultfile"`
else
    lxoperand1="$1"
fi

if [ "$3" = "r" ]; then
    if ! [ -f "$lxresultfile" ]; then
        >&2 echo "$lxprog error: no result available"
        exit 1
    fi
    lxoperand2=`cat "$lxresultfile"`
else
    lxoperand2="$3"
fi
```

# Script Example 3

```
# Special handling of multiplication
if [ "$2" = "x" ]; then
    lxoperator="*"
else
    lxoperator="$2"
fi

clccore.exe "$lxoperand1" "$lxoperator" "$lxoperand2" > "$lxtmp"

if [ $? -ne 0 ]; then
    >&2 echo "$lxprog error: invalid calculation"
    exit 1
fi

mv "$lxtmp" "$lxresultfile"
cat "$lxresultfile"
```

Note: the correct path needs to be provided in the call to `clccore.exe` if this executable is not in the system path.

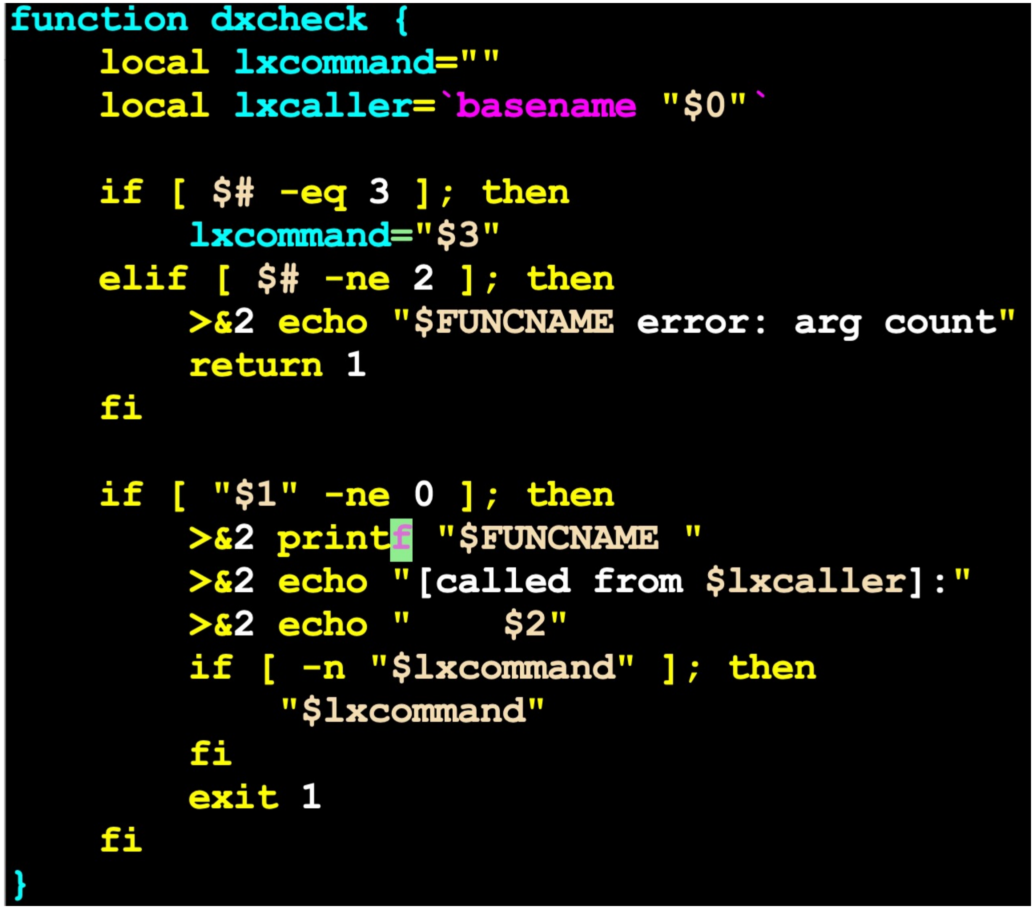
# Summary of Demonstrated Features and Conventions

- Using `$#` to get the argument count of a script or function
- Exiting with a non-zero status upon error detection
- `$UXTMP`: DICE user space for storing temporary files. Can be cleaned using `dxclntmp`
- Using a Bash script to "wrap" one or more binary executables (`clccore.exe` in this case)
- Using `$?` to get return status/value
- Using `[ -f <filename> ]` to test for file existence
  - There are also `-d` and `-a` tests
- Using `` `<command>` `` to capture standard output

# Summary of Demonstrated Features and Conventions (continued)

- `echo` to display messages (there is also `printf`, which does not automatically append a newline).
- Displaying error messages using `>&2` (redirection to standard error)
- Using quotes around strings (e.g., `lxtmp="$UXTMP/$lxprog-tmp.txt"`)
  - Robust when there are spaces in variable values
  - More consistent syntax coloring in editors
- Using `#!/usr/bin/env bash` to reference the default version of Bash in the user's environment.
- Using "$1", "$2", etc. to access positional arguments from a script.

# dxcheck

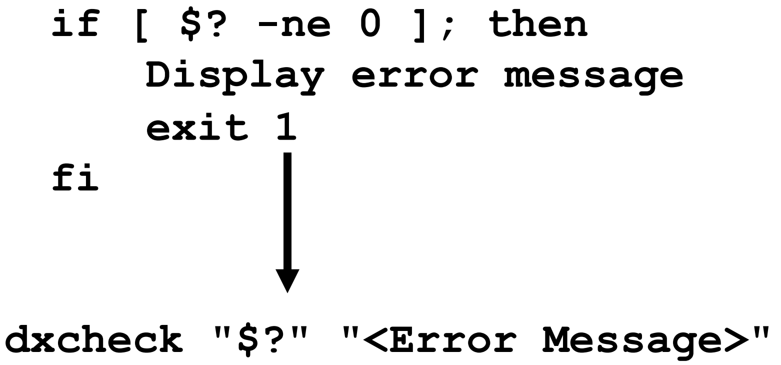


```
function dxcheck {
    local lxcommand=""
    local lxcaller=`basename "$0"`

    if [ $# -eq 3 ]; then
        lxcommand="$3"
    elif [ $# -ne 2 ]; then
        >&2 echo "$FUNCNAME error: arg count"
        return 1
    fi

    if [ "$1" -ne 0 ]; then
        >&2 printf "$FUNCNAME "
        >&2 echo "[called from $lxcaller]:"
        >&2 echo "    $2"
        if [ -n "$lxcommand" ]; then
            "$lxcommand"
        fi
        exit 1
    fi
}
```

**dxcheck** is DICE function that facilitates validation of return/exit status values



```
if [ $? -ne 0 ]; then
    Display error message
    exit 1
fi
```

```
dxcheck "$?" "<Error Message>"
```

This is a *function* that is intended to be called from *scripts*.

Similar functionality can be provided in a more concise form by jointly using the Bash **set** (with the **-e** option) and **trap** (trap on **ERR**) commands; however this approach is a little less flexible to work with.

S. S. Bhattacharyya, *A Brief Introduction to Shell Scripts, Bash, and DICE*, University of Maryland at College Park, Jan., 2024, slide 12.

# Bash Functions

- Run faster than scripts because they are in the memory of the shell
- Functions can help to decompose the functionality of a complex script into smaller, modular components
- When you **`source`** a script that contains a function definition, the function can be used in the remainder of the calling Bash session
- Functions do not run in separate processes, as scripts do
  - Therefore, if you execute the **`exit`** command from a function, the calling process exits.
  - To avoid this behavior, use the **`return`** command instead from within functions.
- If a function and a script have the same name, the function takes precedence
- As with scripts, positional arguments are accessed using "$1", "$2", etc.
- **`$FUNCNAME`** gives the name of the currently executing (innermost) function.
- The **`local`** keyword is used to ensure that variable definitions are local to the function (e.g., they don't clutter the caller's environment).

# What is DICE?

- Website: `http://www.ece.umd.edu/DSPCAD/projects/dice/dice.htm`
- A **`Bash`**-based project development environment that emphasizes
  - Cross-platform, command-driven operation
  - Language-agnostic operation; integration across heterogeneous design languages
  - Support for model-based design
  - Unit testing, and test-driven design
  - Ease of learning, use, and interoperability for interdisciplinary design teams
- The DICE package provides many useful utilities in the form of **`Bash`** scripts and functions.

# What DICE is *not*

- A shell
- A software synthesis tool
- A compiler
- A replacement for language-specific development tools and IDEs
- A debugger, simulator, or transcoder

Instead, DICE is a command-line solution to utilize all of these existing kinds of tools more effectively, especially for cross-platform design.

# Utility Scripts Provided in DICE for Efficient Directory Navigation

- The DICE utilities for directory navigation allow one to label directories with arbitrary (user-defined) identifiers,
- … and to move to directories by simply referencing these identifiers (rather than the complete directory path).
- This makes it very easy to “jump” from one directory to another.
- The main DICE utility related to directory navigation is **`dlk`** (the Directory LinKing utility)
- Other navigation-related utilities include **`rlk`**, and **`plk`**.

# Using the **`dlk`** Utility

- Usage: **`dlk <label>`**
  - This assigns a label to a directory.
  - In our script usage documentation, a string surrounded by **`<..>`** represents a placeholder for a user-specified command argument
- When a label **`<label>`** is assigned with **`dlk`**, a file named **`<label>.txt`** is created in the **`$UXGO`** directory.
- **`dlk`** label names can be of arbitrary length, but should contain only alphanumeric characters (e.g., no spaces).
- Once one runs **`dlk <label>,`** the user can return to the same directory at any time (during the same login session or a subsequent session) by running the DICE “g” command:
  - **`g <label>:`** cd (change directory) to the directory whose label is **`<label>`**.

# `dlk` example

- Example usage:

```
cd ~/mywork/proj/proj1
dlk p1
cd ~/myplay
g p1
```

- After the above sequence of commands, the user will end up in **`~/mywork/proj/proj1`** (assuming that this directory exists).

# Other navigation-related scripts in DICE

- **`rlk <label>`**
  - Remove the label associated with a directory
  - This is useful for conserving space or reducing clutter in the label cache (**`$UXGO`**) if one is no longer going to use the label.
- **`plk <label>`**
  - This works like **`g <label>`**, except that the new directory is effectively pushed onto the directory stack so that one can return to the original directory with **`popd`**.

# Moving and Copying Files Across Directories

- **`dxcu <arg>`**
  - move to “DICE user clipboard,” which is a repository for storing files and directories as they are “copied”, “cut”, and “pasted”
  - <arg> can be a file or directory
  - dxcu moves the specified file or directory from the current working directory *to* the DICE user clipboard
- **`dxpar <arg>`**
  - move (“paste”) from DICE user clipboard and remove from clipboard
  - <arg> can be a file or directory
  - dxpar effectively moves the specified file or directory *from* the DICE user clipboard to the current working directory
- **`dxco <arg>`** and **`dxpa <arg>`**
  - These work like their cousins dxcu and dxpar, except that they *copy* rather than move the specified files or directories

# Utilities for moving and copying: continued

- **`dxparl`** and **`dxpal`** are variations of **`dxpar`** and **`dxpa`**, respectively, that implicitly reference the last file/directory transferred (LFDT) by **`dxcu`** or **`dxco`**
- Each call to **`dxcu`** or **`dxco`** has the side-effect of updating an internal (shell) variable that stores the name of the LFDT
- **`dxparl`** and **`dxpal`** take **no arguments** — they transfer the LFDT from the DICE user clipboard to the current working directory

# Example

- Suppose **`proj1`** and **`proj2`** are project directories that have been previously labeled as **`pr1`** and **`pr2`**, respectively, by **`dlk`**
- Suppose there is a file called **`utilities.c`** in the **`proj1`** directory
- This file can be copied to the **`proj2`** directory with the following steps:

```
g pr1
dxco utilities.c
g pr2
dxparl
```

# DICE utilities for archiving and extracting directories, 1

- `dxpack`: archives a directory (recursively including all sub-directories) as a gzipped tar file (.tar.gz).
- Usage: `dxpack <directory_name>`
  - The directory name can be followed by an optional "/"
  - Example usage: `dxpack project`
  - Example usage: `dxpack my_files/`

# DICE utilities for archiving and extracting directories, 2

- `dxunpack`: Extract the contents of a tar.gz archive
- Usage: `dxunpack <archive_name>`

  The trailing .tar.gz in the <archive_name> can be omitted or included — it works either way.

  - Example usage: `dxunpack project2` (extracts from project2.tar.gz)
  - Example usage: `dxunpack my_files.tar.gz` (extracts from my_files.tar.gz)
- Note: The archive (.tar.gz file) is removed as a side effect of the `dxunpack` utility

# Summary of DICE Features

- Cross-platform design, implementation, and testing
- Lightweight conventions
- Language-agnostic
- Unit testing support
- Supported on Linux, MacOS, and Windows/Cygwin
- Easy to learn
- IDICE: Instructional Extensions

# References

- [Bhattacharyya 2011] S. S. Bhattacharyya, W. Plishker, C. Shen, N. Sane, and G. Zaki. *The DSPCAD integrative command line environment: Introduction to DICE version 1.1*. Technical Report UMIACS-TR-2011-10, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011. http://drum.lib.umd.edu/handle/1903/11422.
- [Newham 2005] C. Newham and B. Rosenblatt. *Learning the Bash shell*. O'Reilly & Associates, Inc., third edition, 2005.
- C. Ramey and B. Fox. Bash Reference Manual. Free Software Foundation, Inc., December 2020. url https://www.gnu.org/software/bash/manual/bash.pdf.