Shell Programming

UNIX Shell

- The shell sits between you and the operating system, acting as a command interpreter

- The user interacts with the kernel through the shell. You can write text scripts to be acted upon by a shell

- It reads your terminal input and translates the commands into actions taken by the system. The shell is analogous to command.com in DOS

- When you log into the system you are given a default shell

UNIX Shell

- The original shell was the Bourne shell, sh
- Every Unix platform will either have the Bourne shell, or a Bourne compatible shell
- The default prompt for the Bourne shell is $ (or #, for the root user)
- Another popular shell is C Shell. The default prompt for the C shell is %

Shell Programming

- Why write shell scripts?
  - To avoid repetition:
    - To do a sequence of steps with standard Unix commands over and over again so why not do it all with just one command?
  - To automate difficult tasks:
    - Many commands have difficult options to remember every time

Shell Programming

- Write shell programs by creating scripts
- A shell script is a text file with Unix commands in it
- First line of script starts with #! which indicates to the kernel that the script is directly executable
- #! is followed with the name of shell (spaces are allowed) to execute, using the full path name. So to set up a Bourne shell script, the first line would be:
  #!/bin/sh or
  #! /bin/sh

Shell Programming

- The first line is followed by commands
- Within the scripts # indicates a comment from that point until the end of the line.
  - #! /bin/bash # bourne again shell
  - cd tmp
  - mkdir t
- Specify that the script is executable by setting the proper bits on the file with chmod:

  % chmod +x shell_script.sh
- To execute shell script as:

  % ./shell_script.sh or

  % shell_script.sh

Example Script

```
#! /bin/csh
echo "Hello $USER"
echo "This machine is `uname -n`"
# uname - print system information
# -n print network node hostname
echo "The calendar for this month is:"
cal
echo "You are running these processes:"
ps
```

```
Hello khuwaja
This machine is indigo
The calendar for this month is
 July 2016
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
You are running these processes:
  PID TTY          TIME CMD
 1952 pts/30   00:00:00 ps
 4894 pts/30   00:00:13 gedit
24926 pts/30   00:00:00 tcsh
```

Variable Names

- The name of a variable can contain only letters a to z or A to Z, numbers 0 to 9 or the underscore character _
- By convention, Unix Shell variables would have their names in UPPERCASE
- Examples for valid variable names
  _ALI
  TOKEN_A
  VAR_1
- Examples for invalid variable names
  2_VAR
  -VARIABLE
  VAR_A!

Defining Variables

- Variables are defined as:
  variable_name=variable_value

- NAME=“David Green"

- Variables of this type are called scalar variables. A scalar variable can hold only one value at a time

- The shell enables to store any value in a variable
  VAR1="Toronto"
  VAR2=100

Accessing Variables

- To access value stored in a variable, prefix its name with the dollar sign $

- For example, following script would access value of defined variable NAME and would print it on STDOUT

```
#!/bin/sh
NAME=“David Green"
echo $NAME
```

This would produce following value
David Green

Read-only Variables

- Shell provides a way to mark variables as read-only by using readonly command. After a variable is marked read-only, its value cannot be changed

- For example, following script would give error while trying to change the value of NAME

```
#!/bin/sh
NAME="David Green"
readonly NAME
NAME="Peter"
/bin/sh: NAME: This variable is read only
```

Example Script

```
#! /bin/sh
echo -n “Enter first name: “ # prompt for first name
                             # -n = no newline
read FNAME                   # read first name
echo -n Enter last name:     # prompt for second name
read LNAME
MESSAGE="Your name is: $LNAME, $FNAME"
echo $MESSAGE                # no double quotation
                             # necessary
Enter first name: Gulzar
Enter last name: Khuwaja
Your name is: Khuwaja, Gulzar
```

Shell Basic Operators

- Various operators supported by each shell
  - Arithmetic Operators
  - Relational Operators
  - Boolean Operators
  - String Operators

```
#!/bin/sh
val=’expr 2 + 2’        # spaces between operators and expressions
echo "Total value: $val“
```

Total value: 4

Arithmetic Operators

a=10
b=20

Operator	Description	Example
+	Addition - Adds values on either side of the operator	`expr $a + $b` will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
*	Multiplication - Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/	Division - Divides left hand operand by right hand operand	`expr $b / $a` will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
=	Assignment - Assign right operand in left operand	a=$b would assign value of b into a
==	Equality - Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!=	Not Equality - Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

- All conditional expressions inside square braces with spaces

Arithmetic Operators- Example

```
#!/bin/sh

a=10
b=20
val=`expr $a + $b`
echo "a + b : $val"

val=`expr $a - $b`
echo "a - b : $val"

val=`expr $a \* $b`
echo "a * b : $val"

val=`expr $b / $a`
echo "b / a : $val"

val=`expr $b % $a`
echo "b % a : $val"

if [ $a == $b ]
then
   echo "a is equal to b"
fi

if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

Output:

```
a + b : 30
a - b : -10
a * b : 200
b / a : 2
b % a : 0
a is not equal to b
```

Relational Operators

a=10
b=20

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not, if yes then condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	[$a -le $b] is true.

Relational Operators- Example

```
#!/bin/sh
a=10
b=20

if [ $a -eq $b ]
then
   echo "$a -eq $b : a is equal to b"
else
   echo "$a -eq $b: a is not equal to b"
fi

if [ $a -ne $b ]
then
   echo "$a -ne $b: a is not equal to b"
else
   echo "$a -ne $b : a is equal to b"
fi

if [ $a -gt $b ]
then
   echo "$a -gt $b: a is greater than b"
else
   echo "$a -gt $b: a is not greater than b"
fi

if [ $a -lt $b ]
then
   echo "$a -lt $b: a is less than b"
else
   echo "$a -lt $b: a is not less than b"
fi
```

Relational Operators- Example

```
if [ $a -ge $b ]
then
   echo "$a -ge $b: a is greater or  equal to b"
else
   echo "$a -ge $b: a is not greater or equal to b"
fi

if [ $a -le $b ]
then
   echo "$a -le $b: a is less or  equal to b"
else
   echo "$a -le $b: a is not less or equal to b"
fi
```

Output:

10 -eq 20: a is not equal to b

10 -ne 20: a is not equal to b

10 -gt 20: a is not greater than b

10 -lt 20: a is less than b

10 -ge 20: a is not greater or equal to b

10 -le 20: a is less or equal to b

Boolean Operators

- Assume variable a holds 10 and variable b holds 20

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical OR. If one of the operands is true then condition would be true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical AND. If both the operands are true then condition would be true otherwise it would be false.	[$a -lt 20 -a $b -gt 100] is false.

Boolean Operators- Example

```
#!/bin/sh

a=10
b=20

if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi

if [ $a -lt 100 -a $b -gt 15 ]
then
   echo "$a -lt 100 -a $b -gt 15 : returns true"
else
   echo "$a -lt 100 -a $b -gt 15 : returns false"
fi

if [ $a -lt 100 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi

if [ $a -lt 5 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi
```

```
10 != 20 : a is not equal to b
10 -lt 100 -a 20 -gt 15 : returns true
10 -lt 100 -o 20 -gt 100 : returns true
10 -lt 5 -o 20 -gt 100 : returns false
```

String Operators

a="abc"
b="efg"

Operator	Description	Example
=	Checks if the value of two operands are equal or not, if yes then condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero. If it is zero length then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero. If it is non-zero length then it returns true.	[-n $a] is not false.
str	Check if str is not the empty string. If it is empty then it returns false.	[$a] is not false.

String Operators- Example

```
#!/bin/sh

a="abc"
b="efg"

if [ $a = $b ]
then
   echo "$a = $b : a is equal to b"
else
   echo "$a = $b: a is not equal to b"
fi

if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi
```

```
if [ -z $a ]
then
   echo "-z $a : string length is zero"
else
   echo "-z $a : string length is not zero"
fi

if [ -n $a ]
then
   echo "-n $a : string length is not zero"
else
   echo "-n $a : string length is zero"
fi

if [ $a ]
then
   echo "$a : string is not empty"
else
   echo "$a : string is empty"
fi
```

String Operators- Example

abc = efg: a is not equal to b

abc != efg : a is not equal to b

-z abc : string length is not zero

-n abc : string length is not zero

abc : string is not empty

Decision Making

The if...else statement:

```
#!/bin/sh
a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
elif [ $a -gt $b ]
then
   echo "a is greater than b"
elif [ $a -lt $b ]
then
   echo "a is less than b"
else
   echo "None of the condition met"
fi
```

a is less than b

Decision Making

The case...esac Statement

```
#!/bin/sh

FRUIT="kiwi"

case "$FRUIT" in
   "apple") echo "Apple pie is quite tasty."
   ;;
   "banana") echo "I like banana nut bread."
   ;;
   "kiwi") echo "New Zealand is famous for kiwi."
   ;;
esac
```

New Zealand is famous for kiwi.

The while Loop

```
#!/bin/sh

a=0

while [ $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

```
0
1
2
3
4
5
6
7
8
9
```

The for Loop

```
#!/bin/sh

for var in 0 1 2 3 4 5 6 7 8 9
do
   echo $var
done
```

```
0
1
2
3
4
5
6
7
8
9
```

The until Loop

```
#!/bin/sh

i=1
until [ $i -gt 6 ]
do
    echo "Welcome $i times."
    i=‘expr  $i + 1’
done
```

```
Welcome 1 times.
Welcome 2 times.
Welcome 3 times.
Welcome 4 times.
Welcome 5 times.
Welcome 6 times.
```

The select Loop

```
#!/bin/sh
select DRINK in tea cofee water juice all none
do
  case $DRINK in
    tea|cofee|water|all)
      echo "Go to canteen"
      ;;
    juice|appe)
      echo "Available at home"
    ;;
    none)
      break
    ;;
    *) echo "ERROR: Invalid selection"
    ;;
  esac
done
```

```
$ test.sh
1) tea
2) cofee
3) water
4) juice
5) all
6) none
#? 4
Available at home
#? 6
$
```

The break Statement

```
#!/bin/sh

a=0

while [ $a -lt 10 ]
do
   echo $a
   if [ $a -eq 5 ]
   then
      break
   fi
   a=`expr $a + 1`
done
```

```
0
1
2
3
4
5
```

The continue Statement

```
#!/bin/sh

NUMS="1 2 3 4 5 6 7"

for NUM in $NUMS
do
   Q=`expr $NUM % 2`
   if [ $Q -eq 0 ]
   then
      echo "Number is an even number!!"
      continue
   fi
   echo "Found odd number"
done
```

```
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
```

Chapter 22

Input/Output

Introduction

- C's input/output library is the biggest and most important part of the standard library.
- The `<stdio.h>` header is the primary source of input/output functions, including `printf`, `scanf`, `putchar`, `getchar`, `puts`, and `gets`.

File Pointers

- Accessing a stream is done through a ***file pointer,*** which has type `FILE *`.
- The `FILE` type is declared in `<stdio.h>`.
- Additional file pointers can be declared as needed:

```
FILE *fp1, *fp2;
```

Standard Streams and Redirection

- `<stdio.h>` provides three standard streams:

File Pointer	***Stream***	***Default Meaning***
`stdin`	Standard input	Keyboard
`stdout`	Standard output	Screen
`stderr`	Standard error	Screen

- These streams are ready to use—we don’t declare them, and we don’t open or close them.

Standard Streams and Redirection

- A typical technique for forcing a program to obtain its input from a file instead of from the keyboard:

  ```
  demo <in.dat
  ```

  This technique is known as ***input redirection.***
- ***Output redirection*** is similar:

  ```
  demo >out.dat
  ```

  All data written to `stdout` will now go into the `out.dat` file instead of appearing on the screen.

Standard Streams and Redirection

- Input redirection and output redirection can be combined:

```
demo <in.dat >out.dat
```

- The < and > characters don't have to be adjacent to file names, and the order in which the redirected files are listed doesn't matter:

```
demo < in.dat > out.dat
demo >out.dat <in.dat
```

Text Files versus Binary Files

- `<stdio.h>` supports two kinds of files: text and binary.
- The bytes in a ***text file*** represent characters, allowing humans to examine or edit the file.
  - The source code for a C program is stored in a text file.
- In a ***binary file,*** bytes don’t necessarily represent characters.
  - Groups of bytes might represent other types of data, such as integers and floating-point numbers.
  - An executable C program is stored in a binary file.

Opening a File

- Opening a file for use as a stream requires a call of the `fopen` function.
- Prototype for `fopen`:

```
FILE *fopen(const char * restrict filename,
            const char * restrict mode);
```

- `filename` is the name of the file to be opened.
  - This argument may include information about the file's location, such as a drive specifier or path.
- `mode` is a "mode string" that specifies what operations we intend to perform on the file.

Opening a File

- The word `restrict` appears twice in the prototype for `fopen`.
- `restrict`, which is a C99 keyword, indicates that `filename` and `mode` should point to strings.
- The C89 prototype for `fopen` doesn't contain `restrict` but is otherwise identical.

Opening a File

- In Windows, be careful when the file name in a call of `fopen` includes the `\` character.
- The call

```
fopen("c:\project\test1.dat", "r")
```

will fail, because `\t` is treated as a character escape.

- One way to avoid the problem is to use `\\` instead of `\`:

```
fopen("c:\\project\\test1.dat", "r")
```

- An alternative is to use the `/` character instead of `\`:

```
fopen("c:/project/test1.dat", "r")
```

Opening a File

- `fopen` returns a file pointer that the program can (and usually will) save in a variable:

```
fp = fopen("in.dat", "r");
  /* opens in.dat for reading */
```

- When it can't open a file, `fopen` returns a null pointer.

Modes

- Factors that determine which mode string to pass to `fopen`:
  - Which operations are to be performed on the file
  - Whether the file contains text or binary data

Modes

- Mode strings for text files:

String	***Meaning***
`"r"`	Open for reading
`"w"`	Open for writing (file need not exist)
`"a"`	Open for appending (file need not exist)
`"r+"`	Open for reading and writing, starting at beginning
`"w+"`	Open for reading and writing (truncate if file exists)
`"a+"`	Open for reading and writing (append if file exists)

Modes

- Mode strings for binary files:

String	***Meaning***
`"rb"`	Open for reading
`"wb"`	Open for writing (file need not exist)
`"ab"`	Open for appending (file need not exist)
`"r+b"` or `"rb+"`	Open for reading and writing, starting at beginning
`"w+b"` or `"wb+"`	Open for reading and writing (truncate if file exists)
`"a+b"` or `"ab+"`	Open for reading and writing (append if file exists)

Modes

- Note that there are different mode strings for *writing* data and *appending* data.
- When data is written to a file, it normally overwrites what was previously there.
- When a file is opened for appending, data written to the file is added at the end.

Modes

- Special rules apply when a file is opened for both reading and writing.
  - Can't switch from reading to writing without first calling a file-positioning function unless the reading operation encountered the end of the file.
  - Can't switch from writing to reading without calling a file-positioning function.

Closing a File

- The `fclose` function allows a program to close a file that it's no longer using.
- The argument to `fclose` must be a file pointer obtained from a call of `fopen`.
- `fclose` returns zero if the file was closed successfully.
- Otherwise, it returns the error code `EOF` (a macro defined in `<stdio.h>`).

Closing a File

- The outline of a program that opens a file for reading:

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

#define FILE_NAME "example.dat"

int main(void)
{
  FILE *fp;

  fp = fopen(FILE_NAME, "r");
  if (fp == NULL) {
    printf("Can't open %s\n", FILE_NAME);
    exit(EXIT_FAILURE);
  }
  …
  fclose(fp);
  return 0;
}
```

Closing a File

- It's not unusual to see the call of `fopen` combined with the declaration of `fp`:

  ```
  FILE *fp = fopen(FILE_NAME, "r");
  ```

  or the test against `NULL`:

  ```
  if ((fp = fopen(FILE_NAME, "r")) == NULL) …
  ```

Miscellaneous File Operations

- The `remove` and `rename` functions allow a program to perform basic file management operations.
- Unlike most other functions in this section, `remove` and `rename` work with file *names* instead of file *pointers*.
- Both functions return zero if they succeed and a nonzero value if they fail.

Miscellaneous File Operations

- `remove` deletes a file:

```
remove("foo");
  /* deletes the file named "foo" */
```

- The effect of removing a file that's currently open is implementation-defined.

Miscellaneous File Operations

- `rename` changes the name of a file:

```
rename("foo", "bar");
  /* renames "foo" to "bar" */
```

- `rename` is handy for renaming a temporary file created using `fopen` if a program should decide to make it permanent.
  - If a file with the new name already exists, the effect is implementation-defined.
- `rename` may fail if asked to rename an open file.

Formatted I/O

- The next group of library functions use format strings to control reading and writing.
- `printf` and related functions are able to convert data from numeric form to character form during output.
- `scanf` and related functions are able to convert data from character form to numeric form during input.

The …`printf` Functions

- The `fprintf` and `printf` functions write a variable number of data items to an output stream, using a format string to control the appearance of the output.
- The prototypes for both functions end with the `...` symbol (an ***ellipsis***), which indicates a variable number of additional arguments:

```
int fprintf(FILE * restrict stream,
            const char * restrict format, ...);
int printf(const char * restrict format, ...);
```

- Both functions return the number of characters written; a negative return value indicates that an error occurred.

The …`printf` Functions

- `printf` always writes to `stdout`, whereas `fprintf` writes to the stream indicated by its first argument:

```
printf("Total: %d\n", total);
  /* writes to stdout */
fprintf(fp, "Total: %d\n", total);
  /* writes to fp */
```

- A call of `printf` is equivalent to a call of `fprintf` with `stdout` as the first argument.

...`printf` Conversion Specifications

- Both `printf` and `fprintf` require a format string containing ordinary characters and/or conversion specifications.
  - Ordinary characters are printed as is.
  - Conversion specifications describe how the remaining arguments are to be converted to character form for display.

Examples of …printf Conversion Specifications

- Examples showing the effect of flags on the %d conversion:

Conversion Specification	*Result of Applying Conversion to 123*	*Result of Applying Conversion to –123*
`%8d`	`•••••123`	`••••-123`
`%-8d`	`123•••••`	`-123••••`
`%+8d`	`••••+123`	`••••-123`
`% 8d`	`•••••123`	`••••-123`
`%08d`	`00000123`	`-0000123`
`%-+8d`	`+123••••`	`-123••••`
`%- 8d`	`•123••••`	`-123••••`
`%+08d`	`+0000123`	`-0000123`
`% 08d`	`•0000123`	`-0000123`

Examples of ...`printf` Conversion Specifications

- Examples showing the effect of the `#` flag on the `o`, `x`, `X`, `g`, and `G` conversions:

Conversion Specification	***Result of Applying Conversion to 123***	***Result of Applying Conversion to 123.0***
`%8o`	`•••••173`	
`%#8o`	`••••0173`	
`%8x`	`••••••7b`	
`%#8x`	`••••0x7b`	
`%8X`	`••••••7B`	
`%#8X`	`••••0X7B`	
`%8g`		`•••••123`
`%#8g`		`•123.000`
`%8G`		`•••••123`
`%#8G`		`•123.000`

Examples of ...`printf` Conversion Specifications

- Examples showing the effect of the minimum field width and precision on the `%s` conversion:

Conversion Specification	*Result of Applying Conversion to* **`"bogus"`**	*Result of Applying Conversion to* **`"buzzword"`**
`%6s`	`•bogus`	`buzzword`
`%-6s`	`bogus•`	`buzzword`
`%.4s`	`bogu`	`buzz`
`%6.4s`	`••bogu`	`••buzz`
`%-6.4s`	`bogu••`	`buzz••`

Examples of …`printf` Conversion Specifications

- Examples showing how the `%g` conversion displays some numbers in `%e` form and others in `%f` form:

Number	***Result of Applying*** **`%.4g`** ***Conversion to Number***
123456.	`1.235e+05`
12345.6	`1.235e+04`
1234.56	`1235`
123.456	`123.5`
12.3456	`12.35`
1.23456	`1.235`
.123456	`0.1235`
.0123456	`0.01235`
.00123456	`0.001235`
.000123456	`0.0001235`
.0000123456	`1.235e-05`
.00000123456	`1.235e-06`

The …`scanf` Functions

- `scanf` always reads from `stdin`, whereas `fscanf` reads from the stream indicated by its first argument:

```
scanf("%d%d", &i, &j);
  /* reads from stdin */
fscanf(fp, "%d%d", &i, &j);
  /* reads from fp */
```

- A call of `scanf` is equivalent to a call of `fscanf` with `stdin` as the first argument.

The …`scanf` Functions

- Errors that cause the …`scanf` functions to return prematurely:
  - ***Input failure*** (no more input characters could be read)
  - ***Matching failure*** (the input characters didn't match the format string)

Output Functions

- `putchar` writes one character to the `stdout` stream:

```
putchar(ch);     /* writes ch to stdout */
```

- `fputc` and `putc` write a character to an arbitrary stream:

```
fputc(ch, fp);   /* writes ch to fp */
putc(ch, fp);    /* writes ch to fp */
```

- `putc` is usually implemented as a macro (as well as a function), while `fputc` is implemented only as a function.

Output Functions

- `putchar` itself is usually a macro:

```
#define putchar(c) putc((c), stdout)
```

- Programmers usually prefer `putc`, which gives a faster program.
- If a write error occurs, all three functions set the error indicator for the stream and return `EOF`.
- Otherwise, they return the character that was written.

Input Functions

- `getchar` reads a character from `stdin`:

```
ch = getchar();
```

- `fgetc` and `getc` read a character from an arbitrary stream:

```
ch = fgetc(fp);
ch = getc(fp);
```

- All three functions treat the character as an `unsigned char` value (which is then converted to `int` type before it's returned).
- As a result, they never return a negative value other than `EOF`.

Input Functions

- `getc` is usually implemented as a macro (as well as a function), while `fgetc` is implemented only as a function.
- `getchar` is normally a macro as well:

```
#define getchar() getc(stdin)
```

- Programmers usually prefer `getc` over `fgetc`.

Program: Copying a File

- The `fcopy.c` program makes a copy of a file.
- The names of the original file and the new file will be specified on the command line when the program is executed.
- An example that uses `fcopy` to copy the file `f1.c` to `f2.c`:

```
fcopy f1.c f2.c
```

- `fcopy` will issue an error message if there aren't exactly two file names on the command line or if either file can't be opened.

Program: Copying a File

- Using "`rb`" and "`wb`" as the file modes enables `fcopy` to copy both text and binary files.
- If we used "`r`" and "`w`" instead, the program wouldn't necessarily be able to copy binary files.

fcopy.c

```
/* Copies a file */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *source_fp, *dest_fp;
  int ch;

  if (argc != 3) {
    fprintf(stderr, "usage: fcopy source dest\n");
    exit(EXIT_FAILURE);
  }
```

```
  if ((source_fp = fopen(argv[1], "rb")) == NULL) {
    fprintf(stderr, "Can't open %s\n", argv[1]);
    exit(EXIT_FAILURE);
  }

  if ((dest_fp = fopen(argv[2], "wb")) == NULL) {
    fprintf(stderr, "Can't open %s\n", argv[2]);
    fclose(source_fp);
    exit(EXIT_FAILURE);
  }

  while ((ch = getc(source_fp)) != EOF)
    putc(ch, dest_fp);

  fclose(source_fp);
  fclose(dest_fp);
  return 0;
}
```

Output Functions

- The `puts` function writes a string of characters to `stdout`:

```
puts("Hi, there!");  /* writes to stdout */
```

- After it writes the characters in the string, `puts` always adds a new-line character.

Output Functions

- `fputs` is a more general version of `puts`.
- Its second argument indicates the stream to which the output should be written:

```
fputs("Hi, there!", fp);  /* writes to fp */
```

- Unlike `puts`, the `fputs` function doesn't write a new-line character unless one is present in the string.
- Both functions return `EOF` if a write error occurs; otherwise, they return a nonnegative number.

Input Functions

- The `gets` function reads a line of input from `stdin`:

```
gets(str);  /* reads a line from stdin */
```

- `gets` reads characters one by one, storing them in the array pointed to by `str`, until it reads a new-line character (which it discards).
- `fgets` is a more general version of `gets` that can read from any stream.
- `fgets` is also safer than `gets`, since it limits the number of characters that it will store.

Input Functions

- A call of `fgets` that reads a line into a character array named `str`:

```
fgets(str, sizeof(str), fp);
```

- `fgets` will read characters until it reaches the first new-line character or `sizeof(str)` – 1 characters have been read.
- If it reads the new-line character, `fgets` stores it along with the other characters.

Input Functions

- Both `gets` and `fgets` return a null pointer if a read error occurs or they reach the end of the input stream before storing any characters.
- Otherwise, both return their first argument, which points to the array in which the input was stored.
- Both functions store a null character at the end of the string.

Block I/O

- The `fread` and `fwrite` functions allow a program to read and write large blocks of data in a single step.
- `fread` and `fwrite` are used primarily with binary streams, although—with care—it's possible to use them with text streams as well.

Block I/O

- `fwrite` is designed to copy an array from memory to a stream.
- Arguments in a call of `fwrite`:
  - Address of array
  - Size of each array element (in bytes)
  - Number of elements to write
  - File pointer
- A call of `fwrite` that writes the entire contents of the array `a`:

```
fwrite(a, sizeof(a[0]),
       sizeof(a) / sizeof(a[0]), fp);
```

Block I/O

- `fwrite` returns the number of elements actually written.
- This number will be less than the third argument if a write error occurs.

Block I/O

- `fread` will read the elements of an array from a stream.
- A call of `fread` that reads the contents of a file into the array `a`:

```
n = fread(a, sizeof(a[0]),
          sizeof(a) / sizeof(a[0]), fp);
```

- `fread`'s return value indicates the actual number of elements read.
- This number should equal the third argument unless the end of the input file was reached or a read error occurred.

Block I/O

- `fwrite` is convenient for a program that needs to store data in a file before terminating.
- Later, the program (or another program) can use `fread` to read the data back into memory.
- The data doesn't need to be in array form.
- A call of `fwrite` that writes a structure variable `s` to a file:

```
fwrite(&s, sizeof(s), 1, fp);
```

File Positioning

- Every stream has an associated ***file position.***
- When a file is opened, the file position is set at the beginning of the file.
  - In "append" mode, the initial file position may be at the beginning or end, depending on the implementation.
- When a read or write operation is performed, the file position advances automatically, providing sequential access to data.

File Positioning

- Although sequential access is fine for many applications, some programs need the ability to jump around within a file.
- If a file contains a series of records, we might want to jump directly to a particular record.
- `<stdio.h>` provides five functions that allow a program to determine the current file position or to change it.

File Positioning

- The `fseek` function changes the file position associated with the first argument (a file pointer).
- The third argument is one of three macros:

`SEEK_SET`	Beginning of file
`SEEK_CUR`	Current file position
`SEEK_END`	End of file

- The second argument, which has type `long int`, is a (possibly negative) byte count.

File Positioning

- Using `fseek` to move to the beginning of a file:

```
fseek(fp, 0L, SEEK_SET);
```

- Using `fseek` to move to the end of a file:

```
fseek(fp, 0L, SEEK_END);
```

- Using `fseek` to move back 10 bytes:

```
fseek(fp, -10L, SEEK_CUR);
```

- If an error occurs (the requested position doesn't exist, for example), `fseek` returns a nonzero value.

File Positioning

- The file-positioning functions are best used with binary streams.
- C doesn't prohibit programs from using them with text streams, but certain restrictions apply.
- For text streams, `fseek` can be used only to move to the beginning or end of a text stream or to return to a place that was visited previously.
- For binary streams, `fseek` isn't required to support calls in which the third argument is `SEEK_END`.

File Positioning

- The `ftell` function returns the current file position as a long integer.
- The value returned by `ftell` may be saved and later supplied to a call of `fseek`:

```
long file_pos;
…
file_pos = ftell(fp);
  /* saves current position */
…
fseek(fp, file_pos, SEEK_SET);
  /* returns to old position */
```

File Positioning

- If `fp` is a binary stream, the call `ftell(fp)` returns the current file position as a byte count, where zero represents the beginning of the file.
- If `fp` is a text stream, `ftell(fp)` isn't necessarily a byte count.

File Positioning

- The `rewind` function sets the file position at the beginning.
- The call `rewind(fp)` is nearly equivalent to `fseek(fp, 0L, SEEK_SET)`.