Lecture 3: Expressions, File I/O, and Program Arguments

Sarah Nadi
nadi@ualberta.ca
Department of Computing Science
University of Alberta

CMPUT 201 - Practical Programming Methodology

[With material/slides from Guohui Lin, Davood Rafei, and Michael Buro. Most examples taken from K.N. King's book]

Agenda

- Expressions
- FILE pointers
- Program arguments

Readings

- Textbook:
  - Ch4
  - Ch 22.1-22.3
  - Last part of Ch 13.7 (superficially for now)

Expressions

- The simplest expression is a variable (e.g., `x`) or a constant (e.g., `3`)
- More complicated expressions apply operators to operands, where an operand can itself be an expression

Basic C Operators

- arithmetic operators (+, -, *, /, %)
- relational operators (>, <, >=, <=,==)
- logical operators (!, &&, ||)
- assignment operators (=, +=, -=, *=, …)

Arithmetic Operators

Unary	Binary	
	Additive	**Multiplicative**
+ unary plus - unary minus	+ binary plus - binary minus	* multiplication / division % remainder

Unary operators require 1 operand (e.g., -1)
Binary operators require 2 operands (e.g., 4 * 5)

Arithmetic Operators

Unary	Binary	
	Additive	**Multiplicative**
+ unary plus - unary minus	+ binary plus - binary minus	* multiplication / division % remainder

Unary operators require 1 operand (e.g., -1)
Binary operators require 2 operands (e.g., 4 * 5)

Operator Precedence:

Highest	+ -	(unary)
	* / %	
Lowest:	+ -	(binary)

You can use parentheses to “override” the default precedence

Arithmetic Operators

right associative

Unary	Binary	
	Additive	**Multiplicative**
+ unary plus - unary minus	+ binary plus - binary minus	* multiplication / division % remainder

Unary operators require 1 operand (e.g., -1)
Binary operators require 2 operands (e.g., 4 * 5)

Operator Precedence:

Highest	+ -	(unary)
	* / %	
Lowest:	+ -	(binary)

You can use parentheses to "override" the default precedence

Arithmetic Operators

right associative

Unary	Binary	
	Additive	Multiplicative
+ unary plus - unary minus	+ binary plus - binary minus	* multiplication / division % remainder

left associative

Unary operators require 1 operand (e.g., -1)
Binary operators require 2 operands (e.g., 4 * 5)

Operator Precedence:

Highest	+ -	(unary)
	* / %	
Lowest:	+ -	(binary)

You can use parentheses to “override” the default precedence

Notes about Arithmetic Operators

- The result of +, -, and * depends on the types of the operands
  - If both are `int`, the result is an `int`
  - If both are `float`, the result is a `float`
  - If `int` and `float` are mixed, the result is a `float`

demo: binary_op.c

Important Notes about / and %

- When both operands are integers, the / operator “truncates” the result by dropping the fractional part of the value. Thus, 1/2 is 0, not 0.5
- The % operator requires integer operands. Otherwise, the program will not compile.
- Using zeros as the right operand of either / or % causes undefined behavior
- When negative operands are used with / or %
  - In C89: it depends on the implementation of the compiler. For e.g., -9/7 could be -1 or -2 (rounded down or up), and -9 % 7 could be -2 or 5 (sign depends on implementation)
  - In C99: result of division is always truncated towards zero (-9/7 = -1) and the result of i % j always has the same sign as i (- 9 % 7 = -2)

Assignment Operators

- Simple assignment operator =
  - left operand gets the value of the right operand. Example:
    ```
    i = 5; /* i is now 5 */
    j = i; /* j is now 5;
    k = 10 * i + j; /* k is now ?? */
    ```
- Compound assignment operators `+=, *=, /=, %=`
  - Allow shortening statements such as `i = i + 3` to `i += 3`

Type Conversion During Assignment

- In the assignment `v = e`, if `v` and `e` do not have the same type, value of `e` is converted to type of `v` during assignment

```
int i;
float f;
i = 72.99f; /* i is now 72 */
f = 136; /* f is now 136.0 */
```

Assignment Operators Have Side Effects

- Assignment operators modify their left operand
- Evaluating `i = 10` produces the result 10 and, as a side effect, assigns 10 to i.
- Multiple assignments can be chained together
  - `i = j = k = 0; //equivalent to i = (j = (k = 0));`
  - `i += j += k; //equivalent to i += (j += k);`
  - The assignment operator is right associative

demo: chaining.c

Assignment Operator Requires an Lvalue

- An *lvalue* is an object stored in computer memory, not a constant or the result of a computation.
- Variables are lvalues; expressions such as 10 or 2 * i are not
- The left operand of an assignment statement **must** be an lvalue

```
12 = i ; //wrong

i = 12; // correct

i + j  = 0; //wrong
```

Reverse Digits

- Programming project #1 (p71): Write a program that asks the user to enter a two-digit number, then prints the number with its digits reverse.
- The number must be read as a single number using %d.
- A session with the program should have the following appearance (underlined values show user input):

```
Enter a two-digit number: 28
The reversal is: 82
```

demo: reverse.c

Increment and Decrement Operators

- Increment operator: ++
- Decrement operator: --
- Increment and decrement operators can be postfix or prefix:
    - postfix: `i++; // (use the value of i then) increment it`
    - prefix: `--i; //decrement i (and then use its value)`
    - At the end, both postfix and prefix operators change the value of i, but their side effects are different

Postfix vs. Prefix: What is the value of i and j?

```
int i, j, k, m;
i = 1;
j = 2;
k = ++i + j++;
m = j++ - 1;
printf(“k=%d, m=%d\n”, k, m);
```

What is the value of k and m?

Answer using mentimeter

Expression Statements

- Any expression can be used as a statement by appending a semicolon

```
++i; // is a statement, but the result of this
expression is discarded since no one used it

i++; // is a statement. The value of i is
fetched but not used. However, the value of i
is actually incremented after executing this
statement. i.e., there is a side effect

i * j - 1; // the result of this expression is
computed and discarded
```

Summary: Precedence & Associativity of Operators

Precedence	Name	Symbol(s)	Associativity
1	increment (postfix) decrement (postfix)	++ --	left
2	increment (prefix) decrement (prefix) unary plus unary minus	++ -- + -	right
3	multiplicative	* / %	left
4	additive	+ -	left
5	assignment	= *= /= %= += −=	right

Summary: Precedence & Associativity of Operators

Precedence	Name	Symbol(s)	Associativity
1	increment (postfix) decrement (postfix)	++ --	left
2	increment (prefix) decrement (prefix) unary plus unary minus	++ -- + -	right
3	multiplicative	* / %	left
4	additive	+ -	left
5	assignment	= *= /= %= += −=	right

Use the table to add parentheses to the following expression:

a = b += c++ - d + --e / -f

Avoid Writing Expressions with Undefined behavior!

```
a = 5;
c = (b = a + 2) - (a = 1);
```

Avoid Writing Expressions with Undefined behavior!

```
a = 5;
c = (b = a + 2) - (a = 1);
```

What is the value of c?

Avoid Writing Expressions with Undefined behavior!

```
a = 5;
c = (b = a + 2) - (a = 1);
```

What is the value of c?

- It is better to break this down into a series of statements to specify the exact order you want things executed in:

```
a = 5;
b = a + 2;
a = 1;
c = b - a;
```

FILE Pointers

- **Streams**
- **File Operations**
- **Formatted I/O from files**

What if I want to read or write data from/to a file instead of from the keyboard and to the screen?

What if I want to read or write data from/to a file instead of from the keyboard and to the screen?

Use input/output redirection

Directly input/output to/from files in your program

Streams

- Stream refers to any source of input or any destination for output
- We have been using the keyboard as our input stream and the terminal/screen as our output stream so far
- C uses the type `FILE` to represent steams, including files

Standard Streams

- `<stdio.h>` already provides 3 standard streams:
    - `stdin` which means the keyboard
    - `stdout` which means the screen
    - `stderr` which means the screen
- The `printf` function we saw last class sends output to `stdout`
- The `scanf` function we saw last class reads input from `stdin`

Redirection

- Input Redirection:
  - Force to read from file instead of keyboard:

```
./myprogram < input.txt
```

- Output Redirection:
  - Force to print to file instead of screen

```
./myprogram > output.txt
```

  - Note that using output redirection means that everything is written to stdout. If you use stdout for your error messages and then redirect the output, you will not realize that something is wrong with your program. Using stderr is good practice to ensure errors are not missed.

Redirection: Separating Errors

- Separate the redirection of errors from regular output

```
./myprogram < input.txt > output.txt 2>
error.txt
```

FILE Pointers

- are used to access a stream
- have the type `FILE*` which is declared in `<stdio.h>` (we will understand what the * and the term “pointer” really mean later in the course)
- are declared as

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

Printing to a Specified Stream

- `printf` prints to the standard output stream (i.e., the screen)
- To print to a specified stream, use `fprintf`, which is defined as follows:

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

- `fprintf` uses the same conversion specifiers as printf. For example:

```
int x = 10;
fprintf(stdout, “%d”, x); //equivalent to printf(“%d”, x);
```

demo: stream.c

`FILE` Operations

- `fopen` declaration:

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

- fclose declaration:

```
int fclose (FILE* steam); //returns 0 if closed successfully
```

- Example of opening and closing a file:

```
FILE* fp = fopen("in.txt", "r"); //opens for reading
FILE* fp2 = fopen("out.txt", "w"); //opens for writing.
                                  //See book for other modes
fclose(fp);
fclose(fp2);
```

Reading from files

- Remember that `scanf` reads from the standard input stream (i.e., the keyboard)
- to read from a specified stream, use `fscanf`, which is defined as follows:

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

- `fscanf` uses the same conversion specifiers as `scanf`
- Every stream has two indicators associated with it: error indicator and end-of-file indicator

```
// This program illustrates how to check for errors before & while reading a file

#include <stdio.h>
#include <stdlib.h> //included to use exit and EXIT_FAILURE
int main (void){
    FILE *fp = fopen("test_read_file.txt", "r");
    int n;
    if (fp == NULL){ /* can’t open file */
        fprintf(stderr, "ERROR: input.txt does not exist.\n");
        exit(EXIT_FAILURE);
     }

    while (fscanf(fp, "%d", &n) != 1){
        if (ferror(fp)){
            /* read error*/
            fclose(fp);
            fprintf(stderr, "A read error has occurred. Program exited\n");
            exit(EXIT_FAILURE);
          }
        if (feof(fp)){
            /* end of file reached before integer is found */
            fclose(fp);
            printf("The input file does not contain lines that
                        begin with an integer\n”);
           return 0;

           }

        fscanf(fp, "%*[^\n]"); /* skips rest of line */
    }
    fclose(fp);
    printf("Found a line that begins with %d\n", n);
    return 0;
}
```

```
// This program illustrates how to check for errors before & while reading a file

#include <stdio.h>
#include <stdlib.h> //included to use exit and EXIT_FAILURE
int main (void){
    FILE *fp = fopen("test_read_file.txt", "r");
    int n;
    if (fp == NULL){ /* can't open file */
        fprintf(stderr, "ERROR: input.txt does not exist.\n");
        exit(EXIT_FAILURE);
     }
    while (fscanf(fp, "%d", &n) != 1){
        if (ferror(fp)){
            /* read error*/
            fclose(fp);
            fprintf(stderr, "A read error has occurred. Program exited\n");
            exit(EXIT_FAILURE);
         }
        if (feof(fp)){
            /* end of file reached before integer is found */
            fclose(fp);
            printf("The input file does not contain lines that
                        begin with an integer\n");
           return 0;

           }

        fscanf(fp, "%*[^\n]"); /* skips rest of line */
    }
    fclose(fp);
    printf("Found a line that begins with %d\n", n);
    return 0;
}
```

$\leftarrow$ **EXIT_FAILURE is a macro defined in the stdlib.h file to be a non-zero value.**

The exit function exits the whole program

```
// This program illustrates how to check for errors before & while reading a file

#include <stdio.h>
#include <stdlib.h> //included to use exit and EXIT_FAILURE
int main (void){
    FILE *fp = fopen("test_read_file.txt", "r");
    int n;
    if (fp == NULL){ /* can't open file */
        fprintf(stderr, "ERROR: input.txt does not exist.\n");
        exit(EXIT_FAILURE);
     }

    while (fscanf(fp, "%d", &n) != 1){
        if (ferror(fp)){
            /* read error*/
            fclose(fp);
            fprintf(stderr, "A read error has occurred. Program exited\n");
            exit(EXIT_FAILURE);
         }
        if (feof(fp)){
            /* end of file reached before integer is found */
            fclose(fp);
            printf("The input file does not contain lines that
                        begin with an integer\n");
           return 0;

           }

        fscanf(fp, "%*[^\n]"); /* skips rest of line */
    }
    fclose(fp);
    printf("Found a line that begins with %d\n", n);
    return 0;
}
```

$\leftarrow$ **EXIT_FAILURE is a macro defined in the stdlib.h file to be a non-zero value.**

The exit function exits the whole program

$\leftarrow$ **ferror is a library function that checks if an error occurred while operating on the stream**

```
// This program illustrates how to check for errors before & while reading a file

#include <stdio.h>
#include <stdlib.h> //included to use exit and EXIT_FAILURE
int main (void){
    FILE *fp = fopen("test_read_file.txt", "r");
    int n;
    if (fp == NULL){ /* can’t open file */
        fprintf(stderr, "ERROR: input.txt does not exist.\n");
        exit(EXIT_FAILURE);
     }
    while (fscanf(fp, "%d", &n) != 1){
        if (ferror(fp)){
            /* read error*/
            fclose(fp);
            fprintf(stderr, "A read error has occurred. Program exited\n");
            exit(EXIT_FAILURE);
         }
        if (feof(fp)){
            /* end of file reached before integer is found */
            fclose(fp);
            printf("The input file does not contain lines that
                        begin with an integer\n”);
           return 0;

           }

        fscanf(fp, "%*[^\n]"); /* skips rest of line */
    }
    fclose(fp);
    printf("Found a line that begins with %d\n", n);
    return 0;
}
```

EXIT_FAILURE is a macro defined in the stdlib.h file to be a non-zero value.

The exit function exits the whole program

ferror is a library function that checks if an error occurred while operating on the stream

foef is a library function that checks if end of file is reached

```
// This program illustrates how to check for errors before & while reading a file

#include <stdio.h>
#include <stdlib.h> //included to use exit and EXIT_FAILURE
int main (void){
    FILE *fp = fopen("test_read_file.txt", "r");
    int n;
    if (fp == NULL){ /* can’t open file */
        fprintf(stderr, "ERROR: input.txt does not exist.\n");
        exit(EXIT_FAILURE);
     }

    while (fscanf(fp, "%d", &n) != 1){
        if (ferror(fp)){
            /* read error*/
            fclose(fp);
            fprintf(stderr, "A read error has occurred. Program exited\n");
            exit(EXIT_FAILURE);
         }
        if (feof(fp)){
            /* end of file reached before integer is found */
            fclose(fp);
            printf("The input file does not contain lines that
                        begin with an integer\n”);
           return 0;

           }

        fscanf(fp, "%*[^\n]"); /* skips rest of line */
    }
    fclose(fp);
    printf("Found a line that begins with %d\n", n);
    return 0;
}
```

EXIT_FAILURE is a macro defined in the stdlib.h file to be a non-zero value.

The exit function exits the whole program

ferror is a library function that checks if an error occurred while operating on the stream

foef is a library function that checks if end of file is reached

Notice how it’s important to close the stream once done (or if you faced an error)

Printing to a File

```
FILE* fp = fopen("out.txt", "w");
int x = 10;
fprintf(fp, "%d", x);
fclose(fp);
```

What if we don’t want to hard-code the name of the file we want to read from or write to?

Program Arguments

Program Arguments

- Also called command-line arguments
- For example, I can pass the name of a file to the program as follows:

```
./demo input.txt
```

- “input.txt” in the above example is called a program argument, or a command-line argument

Accessing Command Line Arguments

- can be done by changing the definition of your main function as follows:

```
int main(int argc, char *argv[]) {
 ...
 }
```

- `argc` is the number of command-line arguments that have been passed to the program
- `argv` is an array of pointers to the string arguments, where `argv[0]` points to the program name
- We will understand the structure of `argv` in more detail when we cover Strings and pointers

Obtaining the Filename from the Command Line

- If your program runs as

```
./wordsearch input.txt
```

  - argc is 2
  - argv[0] points to the string “./wordsearch”
  - argv[1] points to the string “input.txt”

Obtaining the Filename from the Command Line

- If your program runs as

  ```
  ./wordsearch -i input.txt -s 5
  ```

  - argc is 5
  - argv[0] points to the string “./wordsearch”
  - argv[1] points to the string “-i”
  - argv[2] points to the string “input.txt”
  - argv[3] points to the string “-s”
  - argv[4] points to the string “5”

```
/* canopen.c Checks whether a file passed as an argument can be
   opened for reading */

#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
     FILE *fp;
    if (argc != 2) {
        printf("usage: canopen filename\n");
        exit(EXIT_FAILURE);
     }


    if ((fp = fopen(argv[1], "r")) == NULL) {
        printf("%s can't be opened\n", argv[1]);
        exit(EXIT_FAILURE);
     }

    printf("%s can be opened\n", argv[1]);
    fclose(fp);
    return 0;
}
```

demo: canopen.c

```c
/* canopen.c Checks whether a file passed as an argument can be
   opened for reading */

#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
     FILE *fp;
    if (argc != 2) {
        printf("usage: canopen filename\n");
        exit(EXIT_FAILURE);
     }


    if ((fp = fopen(argv[1], "r")) == NULL) {
        printf("%s can't be opened\n", argv[1]);
        exit(EXIT_FAILURE);
     }
    printf("%s can be opened\n", argv[1]);
    fclose(fp);
    return 0;
}
```

note the %s used to output a string

demo: canopen.c