OPERATORS

CSC100 Introduction to programming in C/C++ (Spring 2024)

Marcus Birkenkrahe

March 5, 2024

Contents

1 README

- In this section of the course, we go beyond simple statements and turn to program flow and evaluation of logical conditions
- This section follows chapter 3 in Davenport/Vine (2015) and chapters 4 and 5 in King (2008)
- Practice workbooks, input files and PDF solution files in GitHub

2 Preamble

- **Algorithms** are the core of programming
- Example for an algorithm: *"When you come to a STOP sign, stop."*
- The human form of algorithm is **heuristics**
- Example for a heuristic: *"To get to the college, go straight."*
- For **programming**, you need both algorithms and heuristics
- Useful tools to master when designing algorithms:
  - **Pseudocode** (task flow description)
  - **Visual modeling** (task flow visualization)

3 Operators in C

- Mathematically, operators are really functions: `f(i,j)=i+j`
- C has many operators, both unary (`-1`) and binary (`1+1`)
- Types of operators in C:

OPERATOR	WHY	EXAMPLES	EXPRESSION
Arithmetic	compute	* + - / %	i * j + k
Relational	compare	< > <= >=	i > j
Equality	compare (in/equality)	== !=	i == j
Logical	confirm (truth)	&&	i && j
Assignment	change	=	i = j
Increment/decrement	change stepwise	++, +-	++i

- Note: there is no exponential operator (though there is a power function `pow` in `math.h` - see here for more information).
- Conditional operators used in C are important for program flow:

OPERATOR	DESCRIPTION	EXPRESSION	BOOLEAN VALUE
==	Equal	5 == 5	true
!=	Not equal	5 != 5	false
>	Greater than	5 > 5	false
<	Less than	5 < 5	false
>=	Greater than or equal to	5 >= 5	true
<=	Less than or equal to	5 <= 5	true

- The value of an evaluated conditional operator is **Boolean** (logical) - e.g. `2==2` evaluates as `TRUE` or `1`.

4 Operators in other languages

- Different programming languages differ greatly rgd. operators. For example, in the language R, the `|>` operator ("pipe") passes a data set to a function[1].

```
## pipe data set into function
mtcars |> head()  ## same as head(mtcars)
```

Output:

```
                  mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

- You already met the `>` and `>>` operators of the `bash` shell language that redirects standard output to a file:

```
## create empty file called "empty"
> empty
```

[1] Only from R version 4.1 - before that, you have to use the magrittr pipe operator `%>%`.

5 Boolean algebra

- What is algebra about?[2]
- Algebra allows you to form small worlds with fixed laws so that you know exactly what's going on - what the output must be for a given input. This certainty is what is responsible for much of the magic of mathematics.
- Boole's (or Boolean) algebra, or the algebra of logic, uses the values of TRUE (or 1) and FALSE (or 0) and the operators AND (or "conjunction"), OR (or "disjunction"), and NOT (or "negation").
- Truth tables are the traditional way of showing Boolean scenarios:

p	q	p AND q
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

p	q	p OR q
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

p	NOT p
TRUE	FALSE
FALSE	TRUE

- Using the three basic operators, other operators can be built. In electronics, and modeling, the "exclusive OR" operator or "XOR", is e.g. equivalent to (p AND NOT q) OR (NOT p AND q)

[2]Algebra is the branch of mathematics that allows you to represent problems in the form of abstract, or formal, expressions. The abstraction is encapsulated in the notion of a variable (an expression of changing value), and of an operator acting on one or more variables (a function having the variable as an argument, and using it to compute something).

p	q	p XOR q	P = p AND (NOT q)	Q = (NOT p) AND q	P OR Q
TRUE	TRUE	FALSE	FALSE	FALSE	FALSE
TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

- XOR is the operator that we've used in our BPMN models for pseudocode as a gateway operator - only one of its outcomes can be true but never both of them
- Algebraic operations are more elegant and insightful than truth tables. Watch "Proving Logical Equivalences without Truth Tables" (2012) as an example.

6 Order of operator operations

- In compound operations (multiple operators), you need to know the order of operator precedence
- C has almost 50 operators. The most unusual are compound increment/decrement operators[3]:

STATEMENT	COMPOUND	PREFIX	POSTFIX
i = i + 1;	i += 1;	++i;	i++;
j = j - 1;	j -= 1;	–i;	i–;

- `++` and `--` have side effects: they modify the values of their operands: the *prefix* operator `++i` increments `i+1` and then fetches the value `i`:

```
int i = 1;
printf("i is %d\n", ++i);  // prints "i is 2"
printf("i is %d\n", i);  // prints "i is 2"

i is 2
i is 2
```

- The *postfix* operator `++j` also means `j = j + 1` but here, the value of `j` is fetched, and then incremented.

[3]These operators were inherited from Ken Thompson's earlier B language. They are not faster just shorter and more convenient.

```
int j = 1;
printf("j is %d\n", j++);  // prints "j is 1"
printf("j is %d\n", j);  // prints "j is 2"

j is 1
j is 2
```

- Here is another illustration with an assignment of post and prefix increment operators:

```
int num1 = 10, num2 = 0;
puts("start: num1 = 10, num2 =0");

num2 = num1++;
printf("num2 = num1++, so num2 = %d, num1 = %d\n", num2, num1);

num1 = 10;
num2 = ++num1;
printf("num2 = ++num1, so num2 = %d, num1 = %d\n", num2, num1);

start: num1 = 10, num2 =0
num2 = num1++, so num2 = 10, num1 = 11
num2 = ++num1, so num2 = 11, num1 = 11
```

- The table 6 shows a partial list of operators and their order of precedence from 1 (highest precedence, i.e. evaluated first) to 5 (lowest precedence, i.e. evaluated last)

ORDER	OPERATOR	SYMBOL	ASSOCIATIVITY
1	increment (postfix)	`++`	left
	decrement (postfix)	`--`	
2	increment (prefix)	`++`	right
	decrement (prefix)	`--`	
	unary plus	`+`	
	unary minus	`-`	
3	multiplicative	`* / %`	left
4	additive	`+ -`	left
5	assignment	`= *= /= %= += -=`	right

EXPRESSION	EQUIVALENCE	ASSOCIATIVITY
i - j - k	(i - j) - k	left
i * j / k	(i * j) / k	left
-+j	- (+j)	right
i %=j	i = (i % j)	right
i +=j	i = (j + 1)	right

- Left/right *associativity* means that the operator groups from left/right. Examples:
- Write some of these out yourself and run examples. I found `%=` quite challenging: a modulus and assignment operator. `i %= j` computes `i%j` (i modulus j) and assigns it to `i`.
- What is the value of `i = 10` after running the code below?

```
int i = 10, j = 5;
i %= j; // compute modulus of i and j and assigns it to i
printf("i was 10 and is now %d = 10 %% 5\n", i);
```

```
i was 10 and is now 0 = 10 % 5
```

7 Compound if structures and input validation

7.1 TODO Booleans in C

- C evaluates all non-zero values as `TRUE` (`1`), and all zero values as `FALSE` (`0`):

```
if (3) {
  puts("3 is TRUE"); // non-zero expression
 }
if (!!0) puts("0 is FALSE"); // !0 is literally non-zero
```

```
3 is TRUE
```

- The Boolean operators AND, OR and NOT are represented in C by the logical operators `&&`, `||` and `!`, respectively

7.2 TODO ! operator (logical NOT)

- The ! operator is a "unary" operator that is evaluated from the left. It is `TRUE` when its argument is `FALSE` (`0`), and it is `FALSE` when its argument is `TRUE` (non-zero).

☒ If `i = 100`, what is `!i`?

The Boolean value of `100` is TRUE. Therefore, `!100` = `!TRUE` = `FALSE`

☒ If `j = 1.0e-15`, what is `!j`?

The Boolean value of `1.0e-15` is TRUE. Therefore, `!1.0e-15` = `!TRUE` = `FALSE`

☐ Let's check!

```
// declare and assign variables
int i = 100;
double j = 1.e-15;
// print output
printf("!%d is %d because %d is non-zero!\n", i, !i, i);
printf("!(%.1e) is %d because %.1e is non-zero!\n", j, !j, j);
```

```
!100 is 0 because 100 is non-zero!
!(1.0e-15) is 0 because 1.0e-15 is non-zero!
```

7.3 TODO && operator (logical AND)

- Evaluates a Boolean expression from left to right
- Its value is `TRUE` if and only if **both** sides of the operator are `TRUE`

☒ Example: guess the outcome first

```
if ( 3 > 1 && 5 == 10 )
  printf("The expression is TRUE.\n");
 else
   printf("The expression is FALSE.\n");
```

```
The expression is FALSE.
```

☐ Example: guess the outcome first

```
if (3 < 5 && 5 == 5 )
  printf("The expression is TRUE.\n");
 else
   printf
     ("The expression is FALSE.\n");

The expression is TRUE.
```

7.4 TODO || operator (logical OR)

- Evaluates a Boolean expression from left to right
- It is `FALSE` if and only **both** sides of the operator are `FALSE`
- It is `TRUE` if either side of the operator is `TRUE`

☒ Example: guess the outcome first

```
if ( 3 > 5 || 5 == 5 )
  printf("The expression is TRUE.\n");
 else
   printf("The expression is FALSE.\n");

The expression is TRUE.
```

☒ Example: guess the outcome first

```
if ( 3 > 5 || 6 < 5 )
  printf("The expression is TRUE.\n");
 else
   printf("The expression is FALSE.\n");

The expression is FALSE.
```

7.5 TODO Checking for upper and lower case

- Characters are represented by ASCII[4] character sets
- E.g. `a` and `A` are represented by the ASCII codes 97 and 65, resp.

[4] ASCII stands for the American Standard Code for Information Interchange.

- Let's check that.

```
echo "a A" > ./src/ascii
cat ./src/ascii
```

In **??**, two characters are scanned and then printed as characters and as integers:

```
char c1, c2;
scanf("%c %c", &c1, &c2);
printf("The ASCII value of %c is %d\n", c1, c1);
printf("The ASCII value of %c is %d\n", c2, c2);
```

- User-friendly programs should use compound conditions to check for both lower and upper case letters:

```
if (response == 'A' || response == 'a')
```

7.6 TODO Checking for a range of values

- To validate input, you often need to check a range of values
- This is a common use of compound conditions, logical and relational operators
- We first create an input file `num` with a number in it.

```
echo 5 > ./src/num
cat ./src/num
```

☐ What does the code in **??** do? Will it run? What will the output be for our choice of input?

```
int response = 0; // declare and initialize integer

scanf("%d", &response);  // scan integer input

// check if input was in range or not
if ( response < 1 || response > 10 ) {
```

```
 puts("Number not in range.");
} else {
 puts("Number in range.");
}
```

- How can you translate a range like `![1,10]` into a conditional expression? It means that we want to test if a number is outside of the closed interval `[1,10]`.
- The numbers that fulfil this condition are smaller than 1 or greater than 10, hence the condition is `x < 1 || x > 10`.
- This is more conveniently written as `x < 1 || 10 < x`.

7.7 TODO Let's practice

Open and complete the `operators.org` practice file.

8 Let's practice!

- Download the practice file `8_operator_practice.org` from GitHub as bit.ly/op-practice.
- Complete the file and upload it to Canvas.

9 References

- Davenport/Vine (2015) C Programming for the Absolute Beginner (3ed). Cengage Learning.
- GVSUmath (Aug 10, 2012). Proving Logical Equivalences without Truth Tables [video]. URL: youtu.be/iPbLzl2kMHA.
- Kernighan/Ritchie (1978). The C Programming Language (1st). Prentice Hall.
- King (2008). C Programming - A modern approach (2e). W A Norton.
- Orgmode.org (n.d.). 16 Working with Source Code [website]. URL: orgmode.org