OPERATORS

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

Marcus Birkenkrahe

February 22, 2025

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 the book by Davenport/Vine (2015) and chapters 4 and 5 in the book by 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**, with one argument, like `-1`, and **binary**, with two arguments, like `1+1`.
- A list of types of operators in C:
- Note: there is no exponential operator (though there is a power function `pow` in `math.h` [1]
- **Conditional** operators used in C are important for program flow:

[1] See here for more information.

Table 1: Operator types in C

OPERATOR	WHY USE IT	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`

Table 2: Conditional operators in C

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

- Conditional = the operator tests a condition:

  ```
  x == y // is x equal to y? if yes, then return TRUE
  ```

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

- The only **unary** operator is `!` also known as NOT: It merely inverts the Boolean or truth value of its argument.

  ```
  int x = 1;
  printf("If x = %d, then: NOT x = %d\n",x, !x);
  printf("If x = !%d, then: x = %d\n",!x, x);

  If x = 1, then: NOT x = 0
  If x = !0, then: x = 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[2].

```
## pipe data set into function
mtcars |> head(n=2)
## use data set as function argument
head(mtcars,n=2)
```

```
              mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21   6  160 110  3.9 2.620 16.46  0  1    4    4
Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4    4
              mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21   6  160 110  3.9 2.620 16.46  0  1    4    4
Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4    4
```

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

```
> empty  # create empty file called "empty"
ls -l empty  # shows the result
```

5 Boolean algebra

- What is algebra about?[3]
- Why algebra? 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 one way of showing Boolean relationships (there are many other ways, some more intuitive than others[4]):

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

[3] Algebra is a branch of mathematics that deals with **symbols** and the **rules** for combining them to express **relationships** and solve **equations**.

[4] **Logic Gates** represent Boolean expressions through digital circuits - the basis of computers. **Set theory** interprets Boolean operations as union, intersection, and complement. **Venn diagrams** visualize Boolean operations using overlapping circles. **Binary arithmetic** uses Boolean values 0 and 1 in computational operations = truth tables.

Table 3: Conjunction: ‘p AND q‘ for all values of p,q

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

Table 4: Disjunction: ‘p OR q‘ for all values of p,q

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

6 Exploring Boolean algebra

Let's explore Boolean algebra in three different ways to help absolutely everyone get a picture of what it means.

6.1 Conjunction: Logic gates (digital circuits)

- Go to CircuitVerse (circuitverse.org) and sign up for free with your Google Mail account.
- Create a logic gate that represents the operation `p AND q` for varying values of `p` and `q`:

  1. Select two **input** values.
  2. Select the "Logical conjunction" gate ("D").
  3. Select an **output** value.
  4. Combine the elements.
  5. Run through the truth values of the table.

Table 5: Inverse: ‘p‘ and ‘NOT p‘ for all values of p

p	NOT p
TRUE	FALSE
FALSE	TRUE

6. If you want to keep it, save it as a project.

- Your logic gate should look like this:

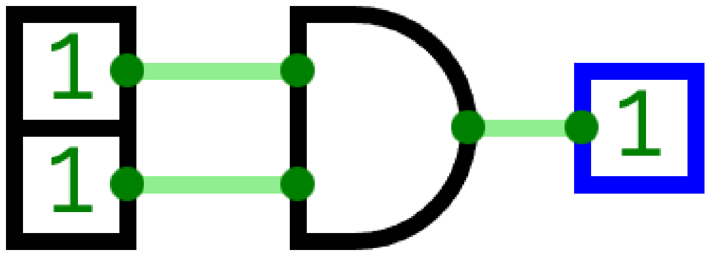

Figure 1: Source: CircuitVerse project

(Project link)

6.2 Disjunction: Set theory (vector algebra)

- Set up the sets in an R language code block

```
p <- c(TRUE, TRUE, FALSE, FALSE) # set of p values
q <- c(TRUE, FALSE, TRUE, FALSE) # set of q values
tt <- data.frame("p"=p,"q"=q)    # truth table setup
print(tt,row.names=FALSE)
```

```
     p     q
  TRUE  TRUE
  TRUE FALSE
 FALSE  TRUE
 FALSE FALSE
```

- Compute the

```
tt["p OR q"] <- p | q # check p OR q for every row of the table
print(tt,row.names=FALSE)
```

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

6.3 Inverse: Set theory diagram (Euler diagram)

- The box is the universe.

  `p` is represented by a circle inside the box.

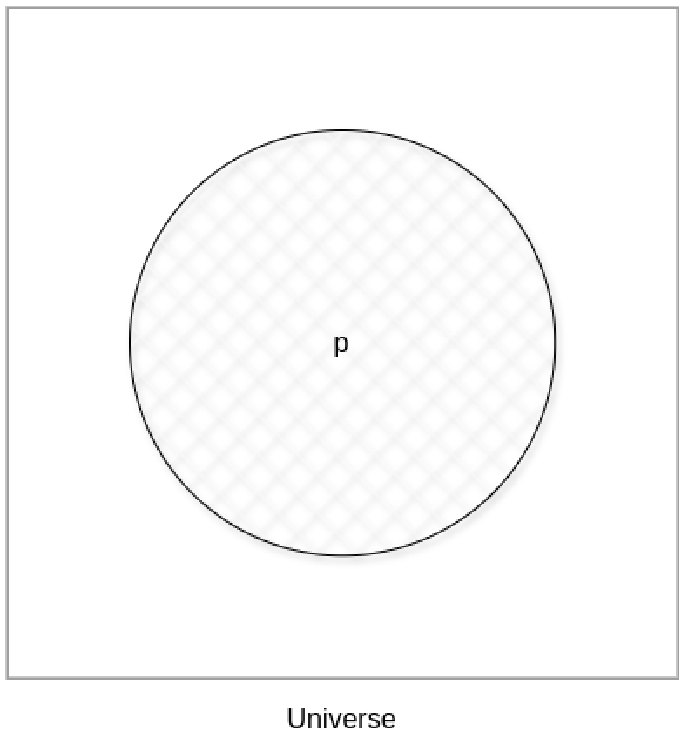

Figure 2: Euler diagram: p in the universe

- What is `NOT p` (`\not p`)?

  `NOT p` is the universe outside of `p`.

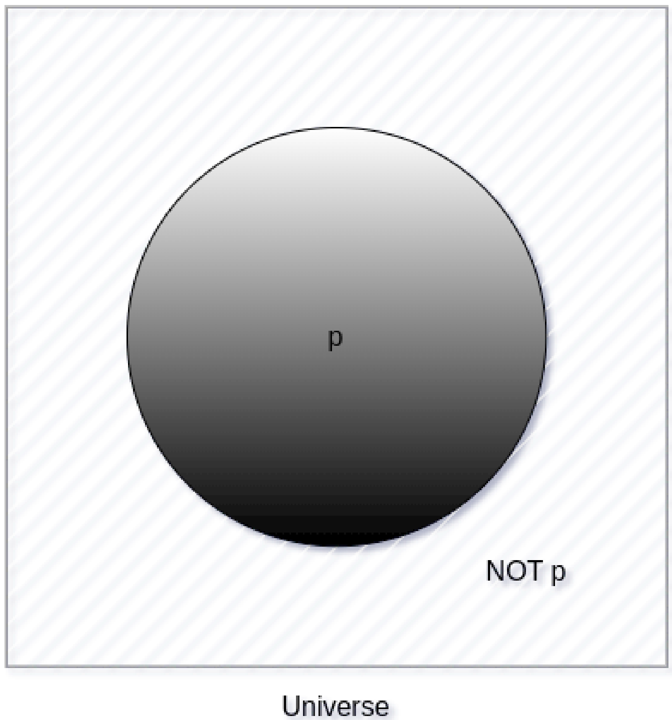

Figure 3: Euler diagram: p and NOT p in the universe

- Therefore, what is the Boolean equation for the universe?

  The universe is `p AND (NOT P)`.

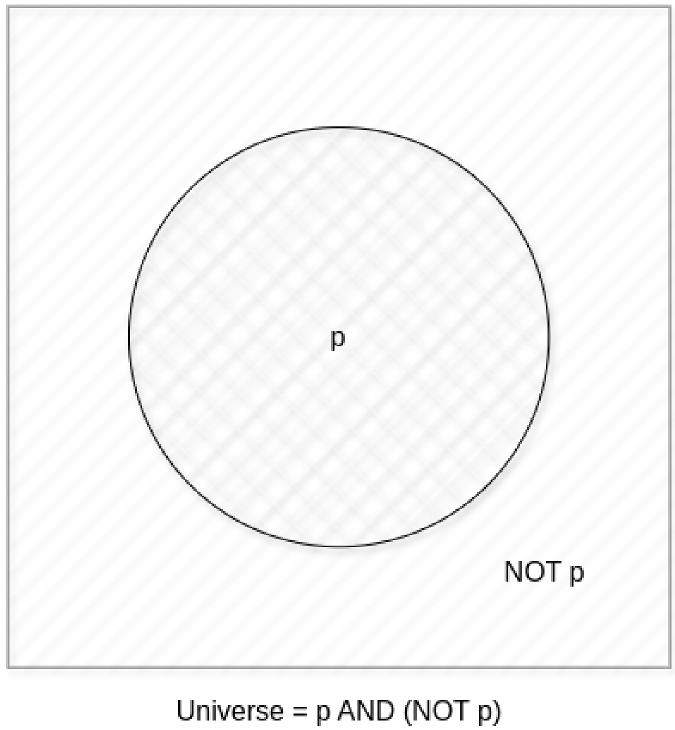

Figure 4: Euler diagram: p AND (NOT p) = Universe

7 Expanding Boolean algebra

- 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)`.

Table 6: Exclusive OR: ‘p XOR q‘ and its derivation

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

- How could you show the truth of the equivalence of `p XOR q` and `(p AND NOT q) OR (NOT p AND q)`?

  > You can show this computationally by going through all p,q $\in \{0,1\}$ - we're using a `for` loop here but we could also do it manually with values `p0=0`, `q0=0`, `p1=0`, `q1=1` or as array values.

- In R (vectorized Boolean operations):

```
## set up truth table with values for p and q
tt <- data.frame("p"=c(TRUE,TRUE,FALSE,FALSE),"q"=c(TRUE,FALSE,TRUE,FALSE))

## compute (p AND NOT q) OR (NOT p AND q) and add it to the table
tt["p XOR q"] <- (p & !q) | (!p & q)

## print resulting truth table
print(tt,row.names=FALSE)
```

```
    p     q p XOR q
  TRUE  TRUE   FALSE
  TRUE FALSE    TRUE
 FALSE  TRUE    TRUE
 FALSE FALSE   FALSE
```

- In C, `TRUE` is `1` and `FALSE` is `0` (we're going to analyze this later):

```
// non-loop approach without arrays
int p0=0,q0=0,p1=0,q1=1,p2=1,q2=0,p3=1,q3=1;
printf("\n%d %d %d %d\n",
       (p0 && !q0) || (!p0 && q0),
       (p1 && !q1) || (!p1 && q1),
       (p2 && !q2) || (!p2 && q2),
       (p3 && !q3) || (!p3 && q3));

// print p XOR q - the answer should be 0 1 1 0
for (int i=0;i<2;++i) { // 00 01 10 11
  for (int j=0;j<2;++j) {
    printf("%d ", (i && !j) || (!i && j)); //
  }
```

```
 }

// declare truth values p,q as array
int a[] = {0,1};
printf("\n%d %d %d %d\n",
       (a[0] && !a[0]) || (!a[0] && a[0]),
       (a[0] && !a[1]) || (!a[0] && a[1]),
       (a[1] && !a[0]) || (!a[1] && a[0]),
       (a[1] && !a[1]) || (!a[1] && a[1]));
```

```
0 1 1 0
0 1 1 0
0 1 1 0
```

- Result:

  Each row shows the results of `(p AND NOT q) OR (NOT p AND q)` from left to right for all values of `p` and `q`: the same as `p XOR q`:

Table 7: ‘p XOR q‘ with Boolean values and in C (with 0,1)

p	q	p XOR q	`printf`
TRUE	TRUE	FALSE	0
TRUE	FALSE	TRUE	1
FALSE	TRUE	TRUE	1
FALSE	FALSE	FALSE	0

- In R:

```
## set up truth table with values for p and q
tt <- data.frame("p"=c(TRUE,TRUE,FALSE,FALSE),"q"=c(TRUE,FALSE,TRUE,FALSE))

## compute (p AND NOT q) OR (NOT p AND q) and add it to the table
tt["p XOR q"] <- (p & !q) | (!p & q)

## print resulting truth table
print(tt,row.names=FALSE)
```

```
    p     q p XOR q
```

```
 TRUE  TRUE   FALSE
 TRUE FALSE    TRUE
FALSE  TRUE    TRUE
FALSE FALSE   FALSE
```

- Algebraic operations are way more elegant and insightful than truth tables. Watch "Proving Logical Equivalences without Truth Tables" (2012) as an example.

8 Order of operator operations (codealong)

- In compound operations (multiple operators), you need to know the order of operator precedence.
- C has almost 50 operators - more than keywords. The most unusual are compound increment/decrement operators[5]:

Table 8: Compound prefix and postfix operators in C

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);  // increments i, then 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.

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

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

```
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++; // assign num1 to num2 and then add 1 to num1
printf("postfix: num2 = num1++, so num2 = %d, num1 = %d\n", num2, num1);

num1 = 10;     // reset num1 to 10
num2 = ++num1; // add 1 to num1 and then assign it to num2
printf("prefix:  num2 = ++num1, so num2 = %d, num1 = %d\n", num2, num1);
```

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

- The table below 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)

Table 9: Order of precedence of arithmetic operators in C

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

- Left/right *associativity* means that the operator groups from left/right. Examples:

Table 10: Associativity of operators in C

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

- 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
```

9 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
0 is FALSE
```

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

10 ! 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! You can validate these arguments computationally:

```
// 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!
```

11 && 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.
```

12 || 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.
```

13 Proving Boolean equivalence with code

- Problem: show that `p XOR q` and `(p AND NOT q) OR (NOT p AND q)` are equivalent.
- Pseudocode:

```
ALGORITHM: compute the expressions:
           A. (p XOR q)
           B. ((p AND NOT q) OR (NOT p AND q))
Input: all truth values of p and q (stored in a file)
       |p0=0|q0=0|
       |p0=0|q0=1|
       |p0=1|q0=0|
       |p0=1|q0=1|
Output: evaluation of A and B

Begin:
   // Declare values to Boolean variables

   // Read in values from input file

   // Print A = p XOR q for all values of p and q

   // Print B = (p AND NOT q) OR (NOT p AND q) for all values of p and q
End
```

- Create the input file `demorgan` (or generate it manually on Windoze):

```
echo "0 0" >  demorgan
echo "0 1" >> demorgan
echo "1 0" >> demorgan
echo "1 1" >> demorgan
cat demorgan
```

- C code (without loops or arrays)

```
// Declare Boolean variables
int p0,p1,p2,p3,q0,q1,q2,q3;

// Read in values from input file
```

```
scanf("%d%d%d%d%d%d%d%d",&p0,&q0,&p1,&q1,&p2,&q2,&p3,&q3);

// Check that input was correctly read
printf("%d%d\n%d%d\n%d%d\n%d%d\n",p0,q0,p1,q1,p2,q2,p3,q3);

// Print A = p XOR q for all values of p and q
printf("p XOR q: %d %d %d %d\n",0,1,1,0);

// Print B = (p AND NOT q) OR (NOT p AND q) for all values of p and q
printf("p = %d, q = %d,(p AND !q) OR (!p AND q) = %-2d\n",p0,q0,(p0 && !q0) || (!p
printf("p = %d, q = %d,(p AND !q) OR (!p AND q) = %-2d\n",p1,q1,(p1 && !q1) || (!p
printf("p = %d, q = %d,(p AND !q) OR (!p AND q) = %-2d\n",p2,q2,(p2 && !q2) || (!p
printf("p = %d, q = %d,(p AND !q) OR (!p AND q) = %-2d\n",p3,q3,(p3 && !q3) || (!p

printf("\n.........Q.E.D.\n");
```

```
00
01
10
11
p XOR q: 0 1 1 0
p = 0, q = 0,(p AND !q) OR (!p AND q) = 0
p = 0, q = 1,(p AND !q) OR (!p AND q) = 1
p = 1, q = 0,(p AND !q) OR (!p AND q) = 1
p = 1, q = 1,(p AND !q) OR (!p AND q) = 0

.........Q.E.D.
```

- You could also dispense with reading the values (since they're constant) and set the values in the code - this makes it shorter:

```
// Declare and assign values to Boolean variables
int p0=0,q0=0,p1=0,q1=1,p2=1,q2=0,p3=1,q3=1;

// Print A = p XOR q for all values of p and q
printf("%d %d %d %d\n",0,1,1,0);

// Print B = (p AND NOT q) OR (NOT p AND q) for all values of p and q
printf("%-2d",(p0 && !q0) || (!p0 && q0));
printf("%-2d",(p1 && !q1) || (!p1 && q1));
```

```
printf("%-2d",(p2 && !q2) || (!p2 && q2));
printf("%-2d",(p3 && !q3) || (!p3 && q3));

printf("\n.........Q.E.D.\n");
```

```
0 1 1 0
0 1 1 0
.........Q.E.D.
```

14 Checking for upper and lower case

- Characters are represented by ASCII[6] character sets
- E.g. `a` and `A` are represented by the ASCII codes 97 and 65, resp.
- Let's check that.

  ```
  echo "a A" > ascii
  cat 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);
  ```

- What happens if you use the format specifier `%c%c` for `scanf`? Try it.

  > Answer: Instead of the ASCII value for 'A' you get the ASCII value for the space, because after picking up the `a`, `scanf` finds the space (it only expects a string literal, and the space is one of those).

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

  ```
  if (response == 'A' || response == 'a') // accept if either a or A is response
  ```

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

15 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 11 > num
cat 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`.

16 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