

Computer Programming

Bitwise operators. Repetitive statements

Robert Varga

Technical University of Cluj-Napoca
Computer Science Department

Course 3



Contents

- 1 Bitwise operators
- 2 Repetitive statements
 - While loop
 - Do-while loop
 - For loop
- 3 Jump statements
- 4 Exercises



Bitwise operators

- they look at and modify the binary representation of numbers
- applicable to integer types
- in the past, they were faster than arithmetic operations
- on modern compilers there is no significant difference
- they are equivalent to set operations



Bitwise operators

Symbol	type	name	description
~	unary	one's complement	flip all bits
&	binary	bitwise and	apply logical and on all bits in parallel
	binary	bitwise or	apply logical or on all bits in parallel
^	binary	bitwise xor	apply logical xor (exclusive or) on all bits in parallel
<<	binary	left-shift	left shift the left operand by the right operand
>>	binary	right-shift	right shift the left operand by the right operand



Bitwise operators - more details (1)

- one's complement \sim
 - changes the state of each individual bit, including leading zeroes
 - utility: two's complement, set complement
- bitwise and $\&$
 - perform logical and on all bits of the two operands in parallel
 - logical and between two bits is one only if both are ones
 - utility: checking for active bits, changing bits to zero, set intersection
- bitwise or $|$
 - perform logical or on all bits of the two operands in parallel
 - logical or between two bits is one if at least one of them is one
 - utility: setting specific bits to one, set union



Bitwise operators - more details (2)

- bitwise xor \wedge
 - perform logical xor on all bits of the two operands in parallel
 - the xor of two bits is one if exactly one of them is one
 - utility: flipping specific bits, set difference
- left-shift $x \ll k$
 - move the bits of x to the left by k positions
 - zeroes are added to the right
 - in most cases equivalent to $x \cdot 2^k$
 - utility: multiplying by powers of 2
- right-shift $x \gg k$
 - move the bits of x to the right by k positions
 - add 0 bits to the left for positive numbers
 - add 1 bits to the left for negative numbers
 - in most cases equivalent to $\lfloor x/2^k \rfloor$
 - utility: division by powers of 2



Useful identities

$1 + 2 + 4 + \dots + 2^{k-1} = 2^k - 1$ = it is the number with k active bits

$2^k = 1 \ll k$ = it is the number with one active bit and k 0 bits

$\tilde{0}$ = is the number with all bits active

$0|x = x$, zero is neutral element for the or operator

$\tilde{0}|x = \tilde{0}$, or with $\tilde{0}$ is always $\tilde{0}$

$0&x = 0$, and with zero is always zero

$\tilde{0}&x = x$, $\tilde{0}$ is neutral element for the and operator

$0^{\wedge}x = x$, zero is neutral element for the xor operator

$x^{\wedge}x = 0$, anything xored with itself gives 0



Examples

```
1 //bitwise operations
2 int a = 5;
3 int b = 9;
4 int c = 2;
5 printf("%d\n", a & b);
6 printf("%d\n", a ^ c);
7 printf("%d\n", a | b);
8
9 //signed vs. unsigned
10 printf("%d vs %u\n", ~0, ~0);
11
12 //shifting and overflow
13 printf("%d\n", 1 << 30);
14 printf("%d\n", 1 << 31);
15 printf("%u\n", 1 << 31);
```




Examples

```

1 //bitwise operations
2 int a = 5;
3 int b = 9;
4 int c = 2;
5 printf("%d\n", a & b);
6 printf("%d\n", a ^ c);
7 printf("%d\n", a | b);
8
9 //signed vs. unsigned
10 printf("%d vs %u\n", ~0, ~0);
11
12 //shifting and overflow
13 printf("%d\n", 1 << 30);
14 printf("%d\n", 1 << 31);
15 printf("%u\n", 1 << 31);

```

- we must view the data in binary:
- $a = 0000 \dots 0000 \ 0101$
- $b = 0000 \dots 0000 \ 1001$
- $c = 0000 \dots 0000 \ 0010$
- $ = 0000 \dots 0000 \ 0001 = 1$
- $ = 0000 \dots 0000 \ 0111 = 7$
- $ = 0000 \dots 0000 \ 1101 = 13$
- -1 vs max unsigned int = $2^{32} - 1$
- no overflow = 2^{30}
- smallest possible int = -2^{31}
- positive value if unsigned = 2^{31}



Program 3.1 - Shifting vs. arithmetic

```
1 #include <stdio.h>
2
3 int main(){
4     int x;
5     scanf("%d", &x);
6     if (x >> 31 & 1)
7         printf("%d = %d",
8             x>>1, (x-1)/2);
9     else
10        printf("%d = %d",
11            x>>1, x/2);
12    return 0;
13 }
```

- the expression from the if statement checks the most significant bit of x
- this is the sign bit for two's complement
- for negative numbers shifting once to the right is equivalent to dividing by two and rounding down, but truncation is a different operator for negatives
- for positive numbers shifting once to the right is equivalent to dividing by two and rounding down



Correspondence to sets

- a number in binary can be viewed as the indicator vector of a set: the positions with active bits correspond to elements which are part of the set
- for example, the set:

$$S = \{1, 2, 5\}$$

- is equivalent to the following number in binary:

$$x_S = 100110_{(2)}$$

- the singleton set $S = \{k\}$, containing only the element k can be represented by the number:

$$x_S = 1 \ll k = 2^k$$



Correspondence to sets

- set operations have corresponding bitwise operators:
 - complement = \sim , intersection = $\&$, union = $|$, difference = \wedge
- indicator vector is also called bitmask, because it selects certain positions
- once positions are selected, operations can be applied specifically on these positions
- for example, the following flips the k th bit of the number x

$$x = x \wedge (1 \ll k)$$

- for example, the following activates the last two bits of y :

$$y = y | 3$$



Contents

- 1 Bitwise operators
- 2 Repetitive statements
 - While loop
 - Do-while loop
 - For loop
- 3 Jump statements
- 4 Exercises



While loop

- typical syntax:

```
while(expression)  
    statement;
```
- the expression is evaluated
- if it is true, then the statement is executed and then the execution returns to checking the expression
- if it is false, jump to the instruction after the loop
- if the statement is compound (it contains multiple instructions) it must be inside curly brackets { }
- can be read as: while the expression is true, repeatedly execute the statement



Program 3.2 - While loop

```
1 #include <stdio.h>
2
3 int main(){
4     int n = 10;
5     int s = 0;
6     while(n > 0){
7         s = s + n;
8         n--;
9     }
10    printf("sum = %d\n", s);
11    return 0;
12 }
```

- calculates the sum of the first n natural numbers
- the expression from the while loop is often called *halting condition*, even though it describes the conditions in which the statements are repeated
- what is the behavior if $n = 0$? if n is negative? if n is large?



Do-while loop

- typical syntax:

do

 statement;

while(expression)

- the statement is executed, then the expression is checked
- if it is true, then the execution returns to the statement
- if it is false, jump to the instruction after the loop
- if the statement is compound (it contains multiple instructions) it must be inside curly brackets {}
- can be read as: execute the statements, if the expression is true, then repeat



Program 3.3 - Do-while loop

```
1 #include <stdio.h>
2
3 int main(){
4     int n = 10;
5     int b = 2;
6     int nd = 0;
7     do{
8         nd++;
9         n /= b;
10    }while(n>0);
11    printf("digits = %d\n", nd);
12    return 0;
13 }
```

- determines the number of digits of n in base b
- the statements from the loop are executed at least once
- what is the behavior if $n = 0$? if n is negative? if n is large?



For loop

- typical syntax:

```
for(expr_init; expr_run; expr_change)  
    statement;
```
- the first expression is evaluated once at the beginning
- the run condition is checked (2nd expression)
- if true, the statement is executed and then the last expression followed by checking the run condition again (in a loop)
- if false, the execution resumes with statements after the loop
- if the statement is compound (it contains multiple instructions) it must be inside curly brackets { }
- can be read as: starting from `expr_init`, while `expr_run`, execute statement and `expr_change` repeatedly



For loop

- typical syntax:

```
for(expr_init; expr_run; expr_change)
    statement;
```

- equivalent to the following while syntax:

```
expr_init;
while(expr_run){
    statement;
    expr_change;
}
```

- typically used when the required iterations is known beforehand
- most often employs a counter or index variable



For loop - idioms

- counting up from 0 to n-1:
`for(int i=0; i<n; i++)`
- counting up from 1 to n:
`for(int i=1; i<=n; i++)`
- counting down from n-1 to 0:
`for(int i=n-1; i>=0; i--)`
- counting down from n to 1:
`for(int i=n; i>0; i--)`
- 0-indexing is preferred in most cases
- declaring i inside the for loop is good practice, available only since standard C99



Program 3.4 - Simple for loop

```
1 #include <stdio.h>
2
3 int main(){
4     int n = 10;
5     int sum = 0;
6     for(int i=1; i<=n; i++)
7         sum += i;
8     printf("sum = %d\n", sum);
9     return 0;
10 }
```

- sums the first n natural numbers
- we use a temporary counter variable to iterate
- what is the behavior if $n = 0$? if n is negative? if n is large?



Program 3.5 - Advanced for loop

```
1 #include <stdio.h>
2
3 int main(){
4     int n = 10;
5     int sum = 0;
6     for(; n>0; sum += n, n--);
7     printf("sum = %d\n", sum);
8     return 0;
9 }
```

- some expressions can be absent from the for loop
- but the two semicolons must appear
- multiple update expressions can be used, separated by commas
- what is the behavior if $n = 0$? if n is negative? if n is large?



Contents

- 1 Bitwise operators
- 2 Repetitive statements
 - While loop
 - Do-while loop
 - For loop
- 3 Jump statements**
- 4 Exercises



Jump statements


- they permit jumping to a different position in the code
- they are considered more low-level than loops
- ultimately, loop statements are translated into jump instructions
- usually, they produce code that is hard to read
- list of jump statements:
 - continue, break, goto, exit (function)



Continue statement

- can be used only inside loops
- jumps to the line which evaluates the run condition
- in case of a for loop, the update expression is evaluated first
- it is paired with the closest loop

```
while(expr){  
    instr1;  
    continue;  
    instr2;  
}
```

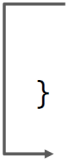
A diagram illustrating the behavior of the 'continue' statement. It shows a line from the 'continue;' statement that goes up and then left, ending with an arrow pointing to the 'while(expr){' line, indicating that the loop body is skipped and the condition is re-evaluated.



Break statement

- can be used only inside loops or switch statements
- jumps to the line which is after the loop (or switch)
- it is paired with the closest loop

```
while(expr){  
    instr1;  
    break;  
    instr2;  
}
```

A diagram consisting of a vertical line on the left, a horizontal line at the top connecting to the 'break;' statement, and another horizontal line at the bottom connecting to the closing brace '}', with a downward-pointing arrow at the end of the vertical line.



Goto statement

- can be used inside functions
- jumps to the line with the given label
- labels can be defined using a name followed by the colon symbol (*name:*)

```
goto et1;  
    instr1;  
et1:    ←  
instr2;
```

A diagram illustrating the execution of a goto statement. A horizontal line from the 'goto et1;' statement extends to the right, then turns 90 degrees downwards, and finally turns 90 degrees to the left, ending with an arrowhead pointing to the 'et1:' label.



exit function

`void exit(int exit_code)`

- can be used inside functions
- terminates the program with a given error code (integer)
- equivalent to the return statement from the main function
- mostly used when an error is encountered from which there is no possible recovery (missing file, failed memory allocation)
- it is a function from `stdlib.h` but has similar behavior to jump statements



Structural programming

- programs which use the 3 main elements: sequential, conditional and repetitive statements
- such programs can calculate any computable function
- promoted by Edsger Dijkstra in 1968
- avoid jump statements = they make the execution hard to follow for humans
- still, in some cases jump statements produce clean code



Contents

- 1 Bitwise operators
- 2 Repetitive statements
 - While loop
 - Do-while loop
 - For loop
- 3 Jump statements
- 4 Exercises



Program 3.6 - bug hunt

```
1 #include <stdio.h>
2 int main(){
3     int a, b;
4     scanf("%d%d", &a, &b);
5     int mn = a;
6     if (b < a)
7         mn = b;
8     int d;
9     for(int i = mn; i>0; i--){
10         if (a%i == 0 && b%i == 0){
11             d = i;
12             break;
13         }
14     }
15     printf("(%d, %d) = %d\n", a, b, d);
16     return 0;
17 }
```

- code purpose?
- find the bug



Program 3.7 - slow gcd

```
1 #include <stdio.h>
2 int main(){
3     int a, b;
4     scanf("%d%d", &a, &b);
5     int mn = a;
6     if (b < a)
7         mn = b;
8     int d = mn;
9     while(d > 0 && (a % d || b % d)){
10         d--;
11     }
12     printf("(%d, %d) = %d\n", a, b, d);
13     return 0;
14 }
```

- compare with previous
- is the bug gone?



Study problem - High IQ

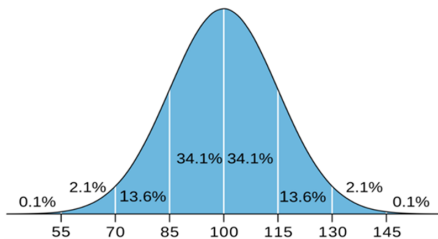
- What are the chances that among n people someone has an IQ higher than x ?



Study problem - High IQ

- we will assume the distribution of IQ is normal with mean 100 and standard deviation 15
- the probability that a single person has an IQ less than x can be calculated as:

$$p = \frac{1}{15\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{(t-100)^2}{2 \cdot 15^2}} dt$$





Study problem - High IQ

- for simplicity, we will ask for this probability p as input
- the "at least one person" = someone is hard to handle
- calculate the probability of the complementary event: none of the persons have IQ higher than x
- this is just the probability of one person having IQ over x , raised to the n th power
- print the complementary probability



Program 3.8 - High IQ

```
1 #include <stdio.h>
2
3 int main(){
4     int n;
5     double p;
6     puts("Nr. people?");
7     scanf("%d", &n);
8     puts("Probability of IQ <= x?");
9     scanf("%lf", &p);
10    p /= 100;
11    double ans = 1;
12    for(int i=0; i<n; i++)
13        ans = ans * p;
14    printf("Probabilitaty of IQ over x: %.2f %%\n",
15          (1-ans) * 100);
16    return 0;
17 }
```