CS11001/CS11002
Programming and Data Structures (PDS)

(Theory: 3-1-0)

Assignments

Example

```
#include<stdio.h>
main()
{
  float a = -7.89., b = 3;
  int c;
  typedef unsigned long newlong;
  newlong d;
   c = (int) a + b;
   d=c;

   printf("%d\n",c);
   printf("%x\n",d);
}
```

Typecasting Again

- float a = 7.89, b = 3.21;
  int c; c = (int)(a + b);

What is the value of c?

The parentheses around the expression a + b implies that the typecasting is to be done after the evaluation of the expression. The following variant has a different effect:

- float a = 7.89, b = 3.21; int c; c = (int)a + b;
- What is the value of c now?

Assignments also return a value.

- int a, b, c; c = (a = 8) + (b = 13);
- Here a is assigned the value 8 and b the value 13. The values (8 and 13) returned by these assignments are then added and the sum 21 is stored in c.
- The assignment of c also returns a value, i.e., 21.
- Here we do not need this value.

Assignment is *right associative*

- For example,

  a = b = c = 0;

  is equivalent to a = (b = (c = 0));
- Here c is first assigned the value 0. This value is returned to assign b, i.e., b also gets the value 0. The value returned from this second assignment is then assigned to a. Thus after this statement all of a, b and c are assigned the value 0.

Generation of Expressions

- A constant is an expression.
- A (defined) variable is an expression.
- If E is an expression, then so also is (E).
- If E is an expression and *op* a unary operator defined in C, then *op* E is again an expression.
- If E1 and E2 are expressions and *op* is a binary operator defined in C, then E1 *op* E2 is again an expression.
- If V is a variable and E is an expression, then V = E is also an expression.

--- *These rules do not exhaust all possibilities for generating expressions, but form a handy set to start with.*

Examples

- 53 /* constant */
- -3.21 /* constant */
- 'a' /* constant */
- x /* variable */
- -x[0] /* unary negation on a variable */
- x + 5 /* addition of two subexpressions */
- (x + 5) /* parenthesized expression */
- (x) + (((5))) /* another parenthesized expression */
- y[78] / (x + 5) /* more complex expression */
- y[78] / x + 5 /* another complex expression */
- y / (x = 5) /* expression involving assignment */
- 1 + 32.5 / 'a' /* expression involving different data types */

Non-examples

- 5 3 /* space is not an operator and integer constants may not contain spaces */
- y *+ 5 /* *+ is not a defined operator */
- x (+ 5) /* badly placed parentheses */
- x = 5; /* semi-colons are not allowed in expressions */

Operators in C

Operator	**Meaning**	**Description**
-	unary negation	Applicable for integers and real numbers. Does not make enough sense for unsigned operands.
+	(binary) addition	Applicable for integers and real numbers.
-	(binary) subtraction	Applicable for integers and real numbers.
*	(binary) multiplication	Applicable for integers and real numbers.

Operators in C

/	(binary) division	For integers division means "quotient", whereas for real numbers division means "real division". If both the operands are integers, the integer quotient is calculated, whereas if (one or both) the operands are real numbers, real division is carried out.
%	(binary) remainder	Applicable only for integer operands.

Examples

- Here are examples of integer arithmetic:
  - 55 + 21 evaluates to 76.
  - 55 - 21 evaluates to 34.
  - 55 * 21 evaluates to 1155.
  - 55 / 21 evaluates to 2.
  - 55 % 21 evaluates to 13.

  Here are some examples of floating point arithmetic:
  - 55.0 + 21.0 evaluates to 76.0.
  - 55.0 - 21.0 evaluates to 34.0.
  - 55.0 * 21.0 evaluates to 1155.0.
  - 55.0 / 21.0 evaluates to 2.6190476 (approximately).
  - 55.0 % 21.0 is not defined.

  Note: C does not provide a built-in exponentiation operator.

Bitwise Operators

- Bitwise operations apply to unsigned integer operands and work on each individual bit.
- Bitwise operations on signed integers give results that depend on the compiler used, and so are not recommended in good programs.
- The following table summarizes the bitwise operations.
- For illustration we use two unsigned char operands a and b. We assume that a stores the value $237 = (11101101)_2$ and that b stores the value $174 = (10101110)_2$.

<table>
<tr><th>Operator</th><th>Meaning</th><th colspan="9">Example</th></tr>
<tr><td rowspan="3">&</td><td rowspan="3">AND</td><td>a = 237</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>b = 174</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>a & b is 172</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td rowspan="3">|</td><td rowspan="3">OR</td><td>a = 237</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>b = 174</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>a | b is 239</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td rowspan="3">^</td><td rowspan="3">EXOR</td><td>a = 237</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>b = 174</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>a ^ b is 67</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td rowspan="2">~</td><td rowspan="2">Complement</td><td>a = 237</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>~a is 18</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td rowspan="2">>></td><td rowspan="2">Right-shift</td><td>a = 237</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>a >> 2 is 59</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td rowspan="2"><<</td><td rowspan="2">Left-shift</td><td>b = 174</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>b << 1 is 92</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr>
</table>

Multiply by 2 (or powers of 2)

```
#include<stdio.h>
main()
{
 int a;
 int n;
  scanf("%d",&a);
  scanf("%d",&n);
  printf("Result: %d\n",a<<n);
}
```

Divide by 2 (or powers of 2)

```
#include<stdio.h>
main()
{
 int a;
 int n;
  scanf("%d",&a);
  scanf("%d",&n);
  printf("Result: %d\n",a>>n);
}
```

If the number is negative

- Suppose a=--5, n=1

+5: 0000 0000 0000 0101
    1111 1111 1111 1010
                      1

    1111 1111 1111 1011 >> 1: 1111 1111 1111 1101

What does this represent?

    0000 0000 0000 0010
                      1

    0000 0000 0000 0011 : +3

Therefore, the result is -3 (So, is it integer division ?)

Bit Complement Operator

- Consider an integer i. How do you make the last 4 bits 0?
- Method 1: i = i & 0xfff0;

  (requires the knowledge of the size of int)
- Method 2: i = (i >> 4)<<4; (requires two shifts)
- **Method 3: i = i & ~0xf;**
- **Concise Form: i &= ~0xf; (expressions like this when the variable being assigned to and the variable being operated on are same can be written like this).**

Extract the n^{th} bit

```
#include<stdio.h>
main()
{
    int i, n;
   int bit;
   scanf("%d",&i);
   scanf("%d",&n);
   bit = (i>>n)&1;
   printf("The %dth bit of %d is %d\n",n,i,bit);
}
```

Problem

- Can you use this code (method) to find out the binary representation of an integer value?
  - Write a C code and check.

Ternary Operator

- Consists of two symbols: ? and :
  - example,

    larger = (i > j) : i : j;
  - i and j are two test expressions.
  - Depending on whether i > j, larger (the variable on the left) is assigned.
    - if (i > j), larger = i
    - else (i,e i<=j), larger = j
  - This is the only operator in C which takes three operands.

Comma Operator

- int i, j;
- i=(j=1,j+10);
- What is the result? j=11.