Expressions

Chapter 4

Arithmetic Operators

- Operators that perform arithmetic

- Binary Operators in C
  - "Binary" means "takes two operands"

+	Add
-	Subtract
*	Multiply
/	Divide
%	Remainder (Modulus)

Operators

13 / 5

/ is called an ***operator***.

13 and 5 are called ***operands***.

- operand = “the thing to be operated on”

13 / 5 is an ***expression***

- Represents the result of dividing 13 by 5

Operators

- What is the value of 13 / 5?
- Since 13 and 5 are both integers, the C compiler will call for integer division.
- 13/5 represents the ***quotient*** obtained when 13 is divided by 5
- Thus, the value of 13 / 5 is 2

Mixed Types

- What happens when we mix integer and floating point numbers in the same calculation?
- On each *operation* (*, /, +, -)
  - If both operands are integer
    - Do integer arithmetic
    - Produce an integer result
  - If ***either*** operand is floating point
    - Convert integer operand to floating point
    - Do floating point arithmetic
    - Produce a floating point result

Storing Floating Point Values

- When a floating point value is stored in an integer variable the fractional part is dropped.
  - *Value is not rounded!*

- Don't confuse storing a floating point value with outputting it with printf.
  - printf *outputs* a rounded value (if necessary).
  - Variable being output is not affected.

The Modulus Operator

- There is one more binary arithmetic operator, %
  - integer ***remainder***
  - a.k.a. “modulus”

  - “a % b” produces the remainder from a divided by b
    - where a and b are integers.

  - 10 % 3 is 1, since $10 = \overset{\text{quotient}}{3} * 3 + \overset{\text{remainder}}{1}$
  - 17 % 5 is 2

Modulus

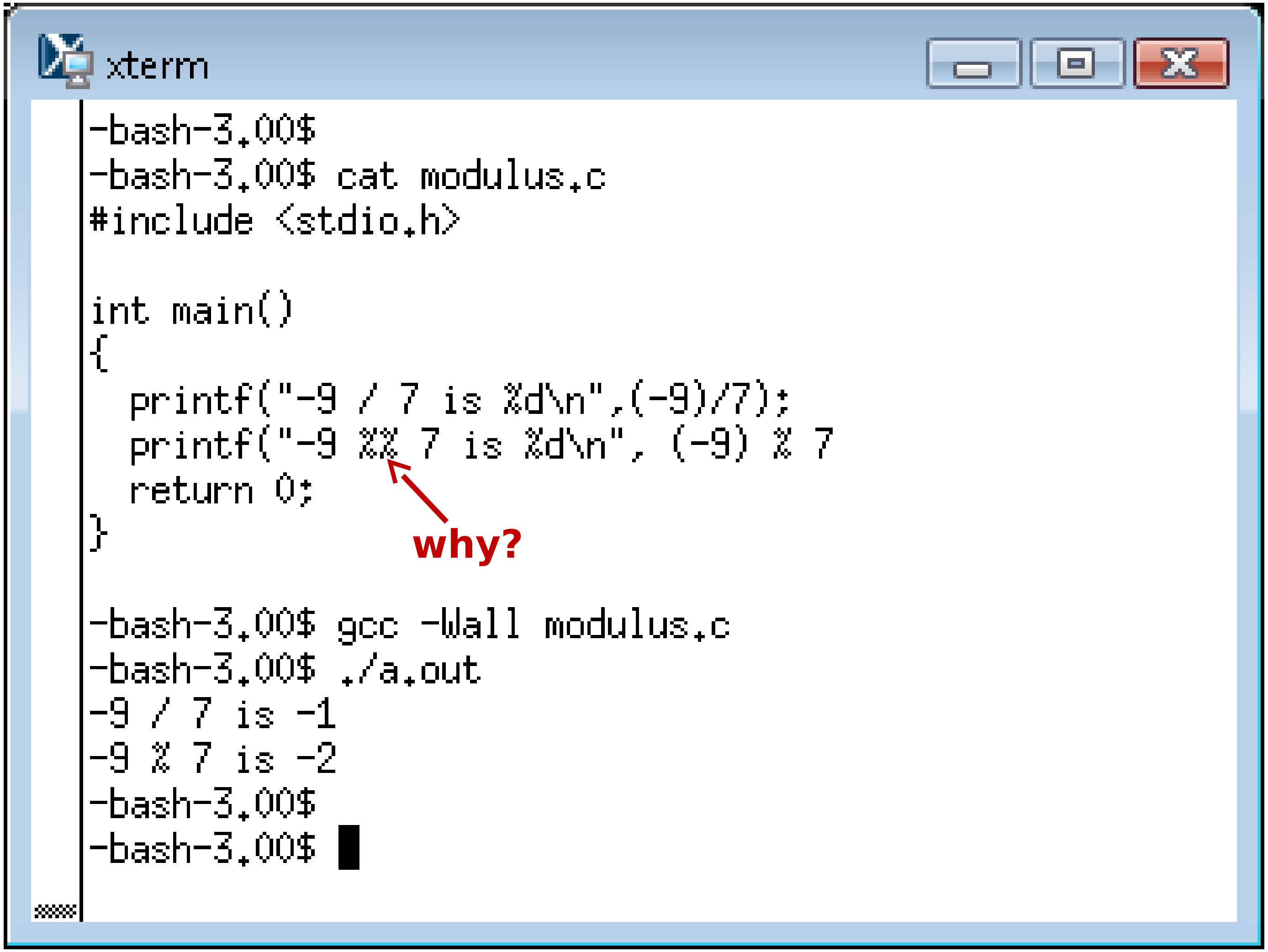
- The modulus operator with negative values is tricky!
  - See page 54 in textbook.

- C89 permits a negative result from integer division to be rounded in either direction.
  - -9 / 7 can be either -1 or -2
  - -9 % 7 can be either -2 or 5
  - implementation dependent.

Modulus

- C99 requires the result of an integer division to be truncated (rounded toward 0)
  - -9 / 7 must be -1
  - -9 % 7 must be -2
- In real life, modulus is rarely used with negative values.

Modulus on Unix Server



```
-bash-3.00$
-bash-3.00$ cat modulus.c
#include <stdio.h>

int main()
{
  printf("-9 / 7 is %d\n",(-9)/7);
  printf("-9 %% 7 is %d\n", (-9) % 7
  return 0;
}

-bash-3.00$ gcc -Wall modulus.c
-bash-3.00$ ./a.out
-9 / 7 is -1
-9 % 7 is -2
-bash-3.00$
-bash-3.00$
```

why?

Operators

13 / 5 + 2

What does this mean?

Divide 13 by 5 then add 2 ?

Add 5 and 2 and then divide 13 by the result ?

How does the compiler decide what we meant?

Operator Precedence

- The C language assigns a *precedence* to each operator.
- Without parentheses:
  - Operation with highest precedence is applied first.
  - Operations with lower precedence take the results of previous operations as their operands.

Operator Precedence

- / has higher precedence than +

- So

13 / 5 + 2

means divide 13 by 5 and then add 2.

or

(13 / 5) + 2

Parentheses Rule!

- Precedence doesn’t matter when we use parentheses.

(a + b) * (c + d)

Means

1. Compute a + b
2. Compute c + d
3. Multiply the results

Operator Precedence

- What about

  16 / 4 / 2

- Precedence can’t determine whether this is (16/4) / 2 or 16 / (4/2)

- The C language also defines *associativity* for operators

Associativity

- Associativity specifies what to do when there are multiple successive instances of the same operator in an expression.
- All binary arithmetic operators have

  left-to-right associativity

- Apply the leftmost operator first, and move to the right.
- So 16 / 4 / 2 means (16/4) / 2

Associativity

- Associativity is only significant for successive operators with the same precedence.
- And only when you don’t use parentheses to specify what to do first.

Associativity

- Associativity might or might not matter

- Doesn’t matter:
  - a + b + c + d
  - a * b * c * d

- Does matter:
  - a – b – c – d
  - a / b / d / c

Order of Operations

- * and / have equal, high precedence
- + and – have equal, lower precedence
- In a complex expression without parentheses, all * and / operations are applied first, left to right
- Then all + and – operations are applied, left to right.

A Reference for Precedence and Associativity

- Appendix A, page 735, tells the precedence and associativity of all operators in the C language
  - including many that we have not seen yet.

- Rule:
  - If you need to look it up, don’t.
  - Use parentheses to make your intentions clear.

Precedence and Associativity

- Memorize the fact that * and / are applied before + and –

and

- * and / are applied left to right
- + and – are applied left to right

- You probably learned the same rules in high school algegra.

Unary Opertors

- We can put + or - in front of an expression.
  - Only one operand: *unary* operator.
  - + has no effect.
  - - is equivalent to -1*
- Not often used.
  - Included in C for consistency with normal mathematical notation.
  - Used, sometimes, for readability

Assignment Operator

- So far as the C compiler is concerned, “=” is just another operator.
  - the “***assignment*** ” operator

- “a = b” means “Set a to the value of b”

Read "a gets b"

- Yields a value, just like “a+b” does.
  - The value is the contents of b.

Assignment Operator

- “a = b = c” is a legal statement.
- = associates right to left.

  (very low precedence)

- Above parses as

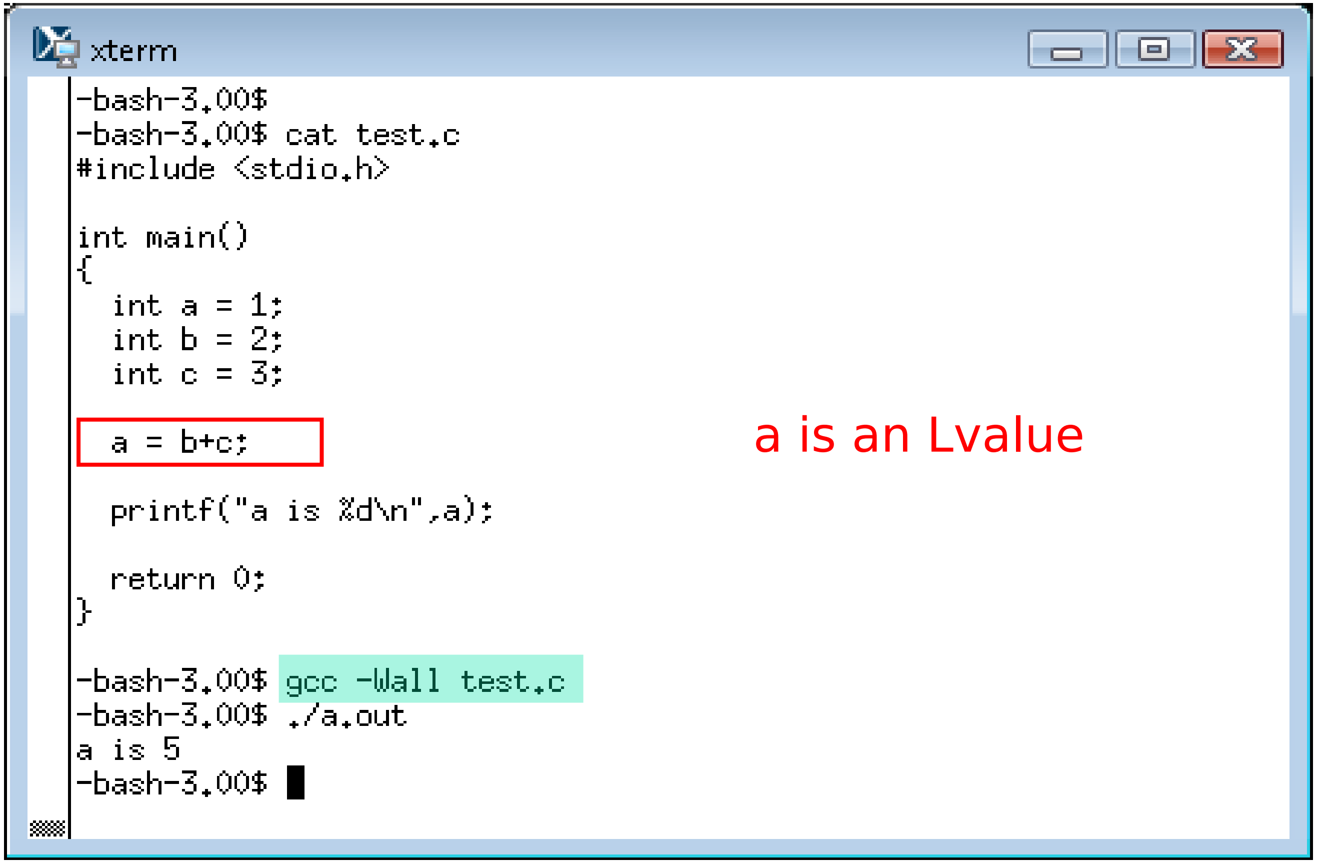
  a = (b = c)

- Both a and b are set to the value of c.

Lvalues

- The = operator is not symmetrical.
- Right Hand Side (RHS) can be any valid expression.
- Left Hand Side (LHS) must be something that has a memory address
  - Normally a variable name
  - Called an Lvalue.

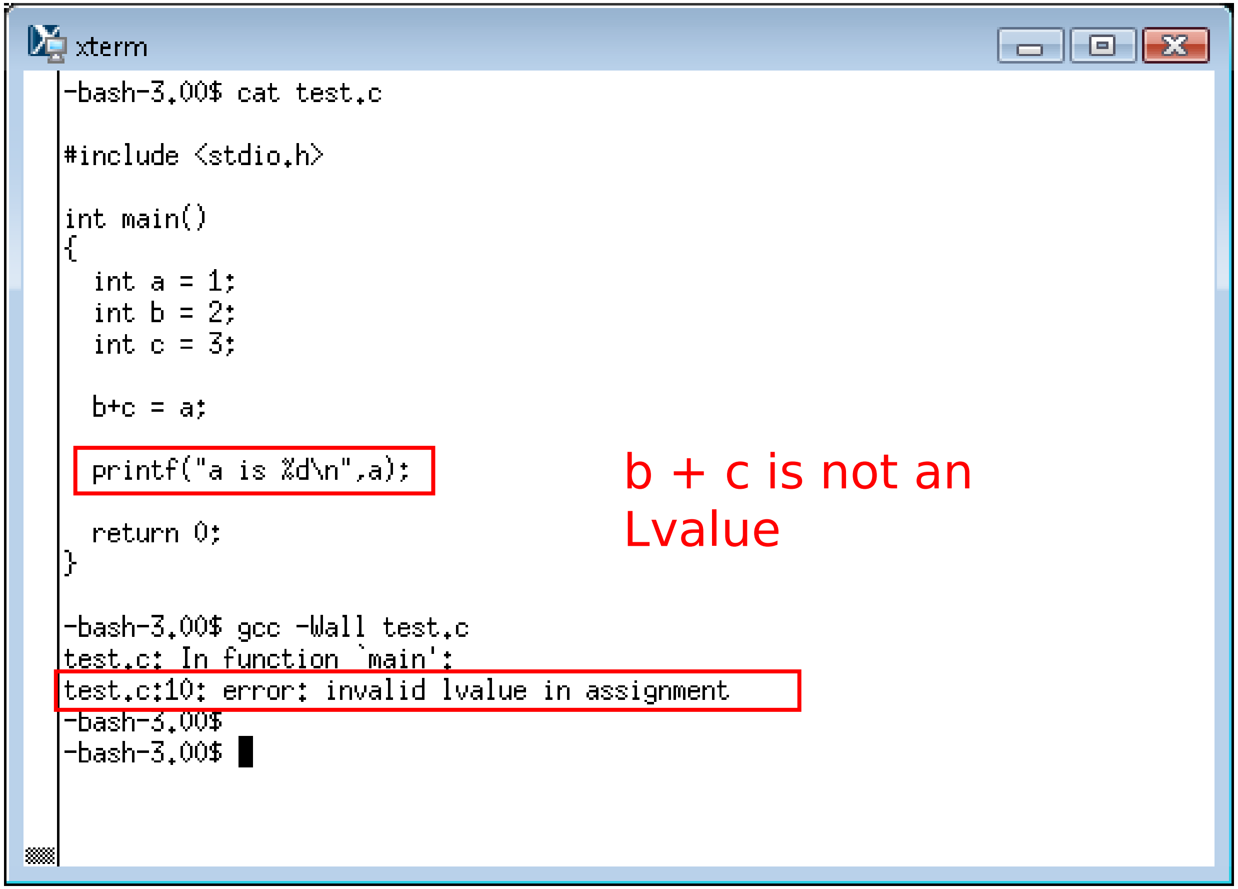
Valid Assignment Statment



```
-bash-3.00$
-bash-3.00$ cat test.c
#include <stdio.h>

int main()
{
  int a = 1;
  int b = 2;
  int c = 3;

  a = b+c;

  printf("a is %d\n",a);

  return 0;
}

-bash-3.00$ gcc -Wall test.c
-bash-3.00$ ./a.out
a is 5
-bash-3.00$
```

a is an Lvalue

Invalid Assignment Statement



```
-bash-3.00$ cat test.c

#include <stdio.h>

int main()
{
  int a = 1;
  int b = 2;
  int c = 3;

  b+c = a;

  printf("a is %d\n",a);

  return 0;
}

-bash-3.00$ gcc -Wall test.c
test.c: In function `main':
test.c:10: error: invalid lvalue in assignment
-bash-3.00$
-bash-3.00$ 
```

b + c is not an Lvalue

Combination Operators

- “x += y” is shorthand for “x = x+y”
- Other operators of this form:
  - -=
  - *=
  - /=
- Part of the C culture.
- OK to use in this class.

Increment and Decrement Operators

x = y++;

means

Set x to the value of y.

Then add 1 to y.

Called "***postincrement***" operator.

Post meaning increment *after* using.

Note that the "++" is written ***after*** the y.

Increment and Decrement Operators

x = ++y;

means

Add 1 to y.

Then set x to the new value of y.

Called “***preincrement*** ” operator.

Pre meaning increment *before* using.

++ is written ***before*** y.

Increment and Decrement Operators

- We also have *postdecrement* and *predecrement* operators

- x = y--;
- x = --y;

Increment and Decrement Operators

- OK to write these operators without an assignment.
- a++;

  just says add one to a.

Identical in effect to

```
        a += 1;
```

or

```
        a = a + 1;
```

Increment and Decrement Operators

- a++;

and

- ++a;

have the same effect.

“Pre” vs. “Post” is significant only when something else is done with the operand in the same statement.

Except: pre is slightly more efficient

Side Effects

- The increment and decrement operators are said to have “*side effects*.”
  - Changing the value of the operand in memory, in addition to changing its value in the expression being computed.
- Compare to “a + b” or just “-a”

Side Effects

- Side effects are common in C.
- Also a common source of errors.
  - Be careful with statements that include side effects.

Side Effects

- What does this do?

```
int x = 1;
double sum = 0.0;
...

  sum += x + (1.0/x++);
```

Is this the value of x before it was incremented, or afterward?

- You can’t be sure!

Increment and Decrement Operators

- Don’t use variable more than once in an expression if it is incremented or decremented.
- Result can vary from one compiler to another.
- Readers are sure to be confused!
- A good general rule:
  - If it is not completely obvious what a statement will do, don’t use it.

Summary

- C provides a lot of "short cuts"
- Saves typing.
- Makes code more concise.
  - But possibly more cryptic.
- Shortcuts are never necessary.
- Use them only if you are sure you understand them
  - and the meaning is clear to a reader.

Assignment

- Read Chapter 4
- Look at the Q & A section
  - Be sure you understand
- Look at the Exercises
  - Try them for yourself.
  - Note that solutions to most exercises are available on the author's web site.

    (http://www.knking.com/books/c2/answers/index.html)
- Look at the Programming Projects.
  - Think about how you would do them.
  - Try some for yourself.
- If anything doesn't make sense, ask for help!