CSE / ENGR 142 Programming I

Iteration

Chapter 5

Read Sections 5.1-5.6, 5.10

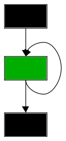
- 5.1 Introduction & While Statement
- 5.2 While example
- 5.3 For Loop
- 5.4 Looping with a fixed bound
- 5.5 Loop design
- 5.6 Nested Loops
- 5.10 Debugging Loops

What’s “Wrong” with HW1?

- User has to rerun the program for every new pair of years
  - Wouldn’t it be nice if the program could process repeated requests?
- Program ends immediately if user types a bad input
  - Wouldn’t it be nice the program politely asked the user again (and again, etc. if necessary)?

One More Type of Control Flow

Sometimes we want to repeat a block of code. This is called a *loop.*



Loops

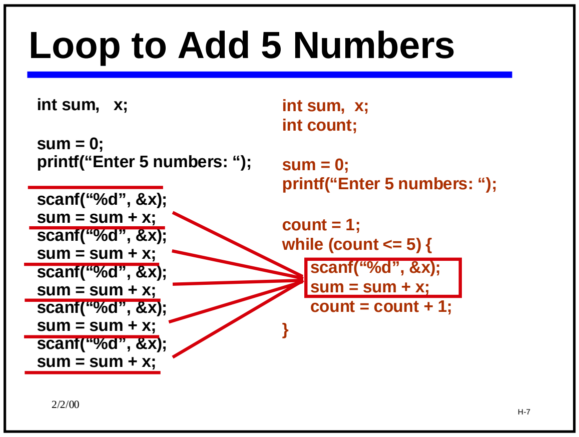
- A “loop” is a repeated (“iterated”) sequence of statements
- Like conditionals, loops (iteration) will give us a huge increase in the power of our programs
- Alert: loops are harder to master than *if* statements
  - Even experienced programmers often make subtle errors when writing loops

Motivating Loops

Problem: add 5 numbers entered at the keyboard. Here’s a solution:

```
int sum;
int x1, x2, x3, x4, x5;

printf(“Enter 5 numbers: ”);
scanf(“%d%d%d%d%d”, &x1, &x2, &x3, &x4, &x5);
sum = x1 + x2 + x3 + x4 + x5;
```

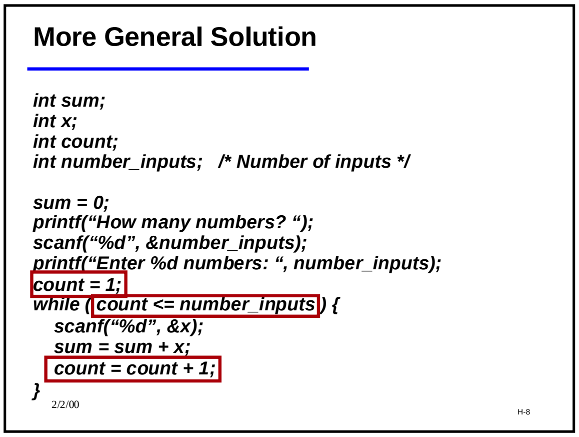
This works perfectly!
But... what if we had 15 numbers? or 50? or 5000?



Loop to Add 5 Numbers

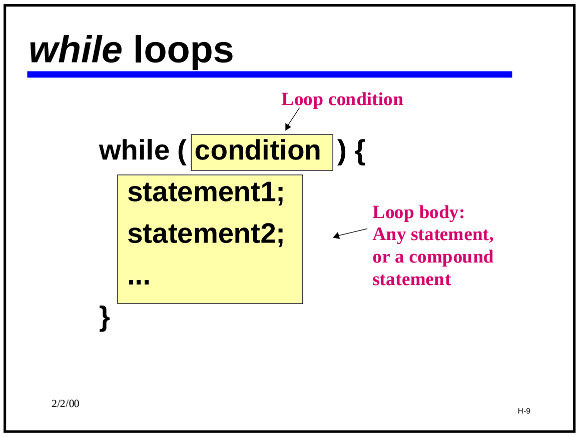
```
int sum,   x;

sum = 0;
printf(“Enter 5 numbers: “);

scanf(“%d”, &x);
sum = sum + x;
scanf(“%d”, &x);
sum = sum + x;
scanf(“%d”, &x);
sum = sum + x;
scanf(“%d”, &x);
sum = sum + x;
scanf(“%d”, &x);
sum = sum + x;
```

```
int sum,  x;
int count;

sum = 0;
printf(“Enter 5 numbers: “);

count = 1;
while (count <= 5) {
    scanf(“%d”, &x);
    sum = sum + x;
    count = count + 1;
}
```



More General Solution

```
int sum;
int x;
int count;
int number_inputs;    /* Number of inputs */

sum = 0;
printf(“How many numbers? “);
scanf(“%d”, &number_inputs);
printf(“Enter %d numbers: “, number_inputs);
count = 1;
while (count <= number_inputs) {
    scanf(“%d”, &x);
    sum = sum + x;
    count = count + 1;
}
```

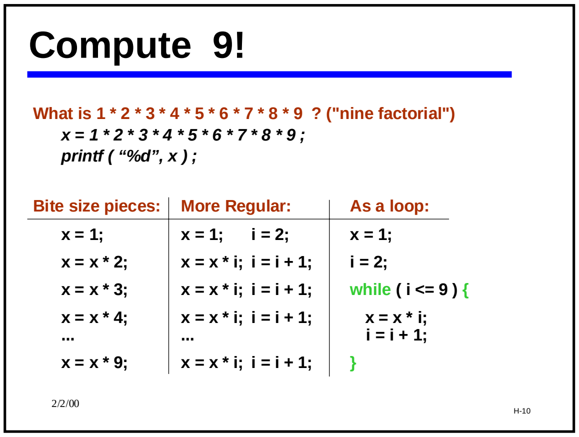


while loops

Loop condition

```
while ( condition ) {
    statement1;
    statement2;
    ...
}
```

Loop body: Any statement, or a compound statement

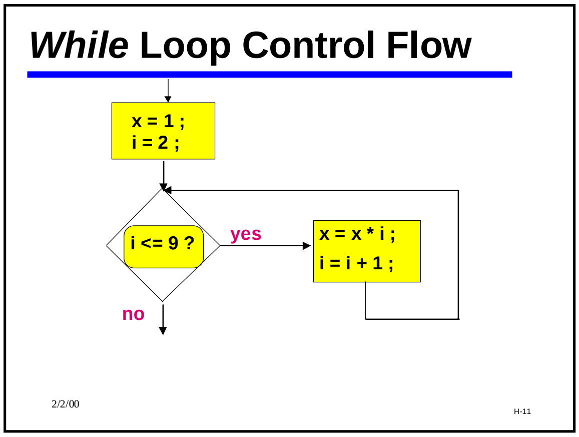


Compute 9!

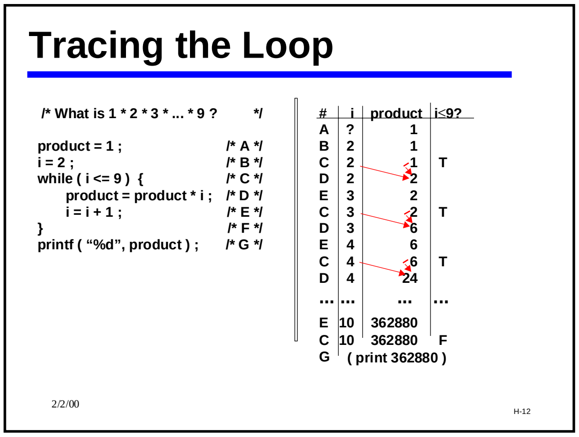
What is 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 ? ("nine factorial")

```
    x = 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 ;
    printf ( “%d”, x ) ;
```

Bite size pieces:	More Regular:	As a loop:
x = 1;	x = 1; i = 2;	x = 1;
x = x * 2;	x = x * i; i = i + 1;	i = 2;
x = x * 3;	x = x * i; i = i + 1;	while (i <= 9) {
x = x * 4;	x = x * i; i = i + 1;	x = x * i;
...	...	i = i + 1;
x = x * 9;	x = x * i; i = i + 1;	}



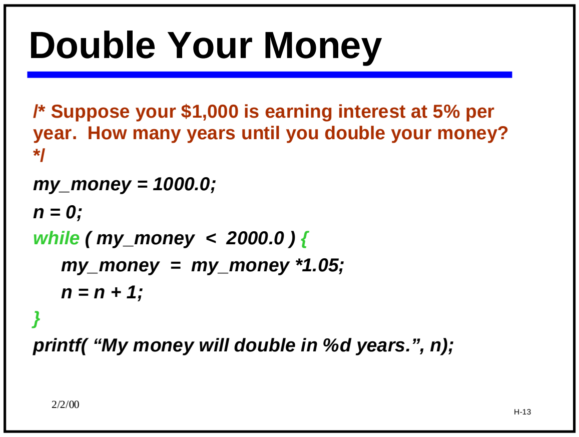
While Loop Control Flow



Tracing the Loop

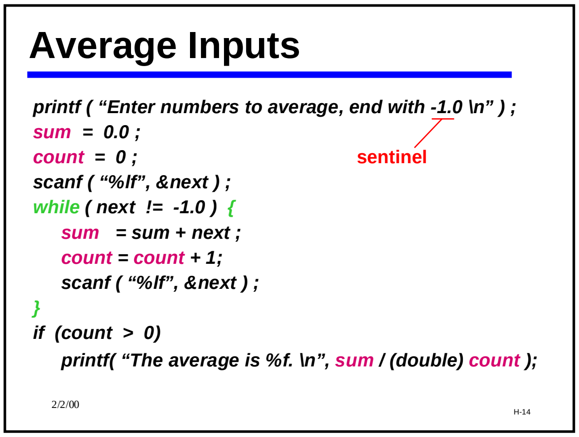
```
 /* What is 1 * 2 * 3 * ... * 9 ?          */

product = 1 ;                       /* A */
i = 2 ;                             /* B */
while ( i <= 9 )  {                 /* C */
        product = product * i ;     /* D */
        i = i + 1 ;                 /* E */
}                                   /* F */
printf ( “%d”, product ) ;          /* G */
```

#	i	product	i≤9?
A	?	1	
B	2	1	
C	2	1	T
D	2	2	
E	3	2	
C	3	2	T
D	3	6	
E	4	6	
C	4	6	T
D	4	24	
...	...	...	...
E	10	362880	
C	10	362880	F
G	(print 362880)		



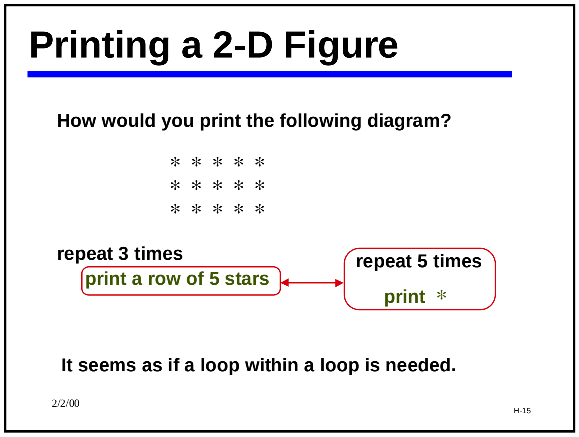
Double Your Money

```
/* Suppose your $1,000 is earning interest at 5% per
year.  How many years until you double your money?
*/
my_money = 1000.0;
n = 0;
while ( my_money  <  2000.0 ) {
    my_money  =  my_money *1.05;
    n = n + 1;
}
printf( “My money will double in %d years.”, n);
```



Average Inputs

```
printf ( “Enter numbers to average, end with -1.0 \n” ) ;
sum  =  0.0 ;
count  =  0 ;                                    sentinel
scanf ( “%lf”, &next ) ;
while ( next  !=  -1.0 )  {
    sum    = sum + next ;
    count = count + 1;
    scanf ( “%lf”, &next ) ;
}
if  (count  >  0)
    printf( “The average is %f. \n”, sum / (double) count );
```



Printing a 2-D Figure

How would you print the following diagram?

```
* * * * *
* * * * *
* * * * *
```

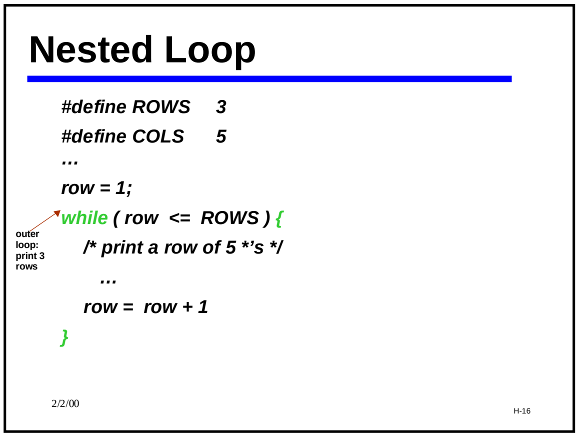
repeat 3 times

print a row of 5 stars

repeat 5 times

print *

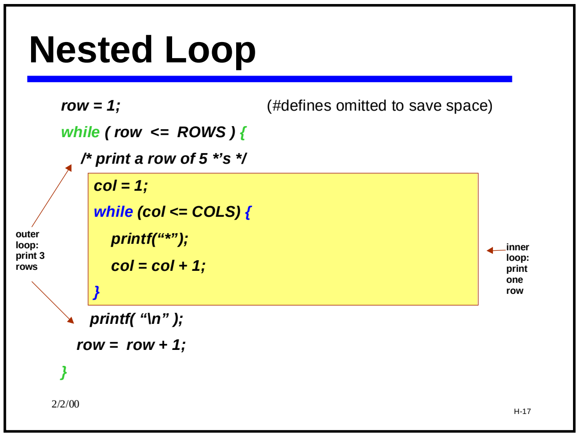
It seems as if a loop within a loop is needed.



Nested Loop

```
#define ROWS     3
#define COLS      5
...
row = 1;
while ( row  <=  ROWS ) {
    /* print a row of 5 *’s */
      ...
    row =  row + 1
}
```

outer loop: print 3 rows

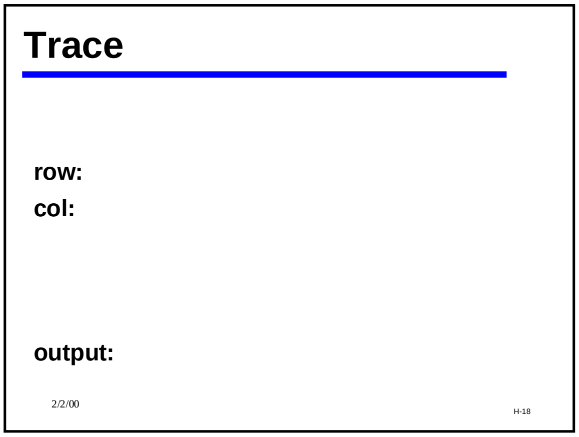


Nested Loop

```
row = 1;                          (#defines omitted to save space)
while ( row  <=  ROWS ) {
    /* print a row of 5 *’s */
    col = 1;
    while (col <= COLS) {
        printf(“*”);
        col = col + 1;
    }
    printf( “\n” );
   row =  row + 1;
}
```

outer loop: print 3 rows

inner loop: print one row



Trace

row:

col:

output:

Print a Multiplication Table

	1	2	3
1	1	2	3
2	2	4	6
3	3	6	9
4	4	8	12

	1	2	3
1	1 * 1	1 * 2	1 * 3
2	2 * 1	2 * 2	2 * 3
3	3 * 1	3 * 2	3 * 3
4	4 * 1	4 * 2	4 * 3

Print Row 2

	1	2	3
1	1	2	3
2	2	4	6
3	3	6	9
4	4	8	12

```
col = 1;
while (col <= 3) {
     printf(“%4d”, 2 * col );
     col = col + 1;
}
printf(“\n”);
```

row number

Nested Loops

Print 4 rows

Print one row

```
row = 1;
while (row <= 4)  {
    col = 1;
    while (col <= 3) {
         printf(“%4d”,  row  * col );
         col = col + 1;
    }
    printf(“\n”);
    row = row + 1;
}
```

Loop Trace

```
row  col
 1    1     print 1
      2     print 2
      3     print 3
          print  \n
 2    1     print 2
      2     print 4
      3     print 6
          print  \n
 3    1     print 3
      2     print 6
      3     print 9
          print  \n
 4    1     print 4
      2     print 8
      3     print 12
          print  \n
```

Loop Trace (Detailed)

```
row  col             statement

 1   ?                1a
 1   ?  (TRUE)        1b
 1   1                2a
 1   1  (TRUE)        2b
 1   1   print 1      3
 1   2                2c
 1   2   (TRUE)       2b
 1   2   print 2      3
 1   3                 2c
 1   3   (TRUE)       2b
 1   3   print 3      3
 1   4                 2c
 1   4   (FALSE)      2b
 1   4    print \n    4
 2   4                1c
 2   4   (TRUE)       1b
 2   1                2a
          . . .
```

Notes About Loop Conditions

- They offer all the same possibilities as conditions in *if*-statements
  - Can use &&, ||, !
- Condition is reevaluated each time through the loop
- A common loop pattern: counting the times through the loop
  - Occurs so often there is a separate statement type based on that pattern: the *for*-statement

for Loops

```
/* What is 1 * 2 * 3 * ... * n ? */

product = 1 ;
i = 2 ;                         /* initialize */
while ( i <= n ) {              /* test */
      product = product * i ;
      i = i+1;                  /* update */
}
printf ( “%d”, product ) ;
```

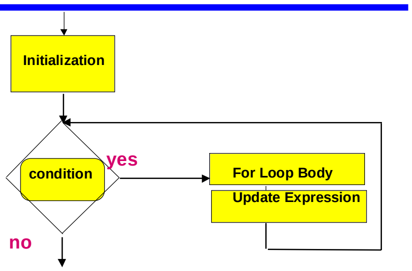
```
product = 1 ;
for ( i = 2 ; i <= n ; i = i+1 ) {
      product = product * i ;
}
printf ( “%d”, product ) ;
```

for Loops Syntax

```
for ( initialization;
      condition;
      update expression ) {
    statement1;
    statement2;
    ...
}
```

“Update” is written at the front of the loop, but executed at the end

for Loop Control Flow



for Loops vs *while* Loops

- Any *for* loop can be written as a *while* loop
- These two loops mean exactly the same thing:

```
for (initialization; condition; update)
      statement;

initialization;
while (condition) {
      statement;
      update
}
```

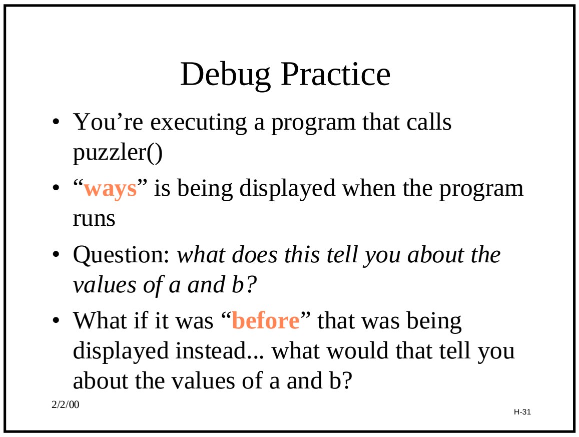
- So *for* provides no new capabilities, but the notation is often convenient.

Counting in *for* Loops

```
/* Print n asterisks */
for ( count = 1 ; count <= n ;  count = count + 1 ) {
    printf ( “*” ) ;
}
```

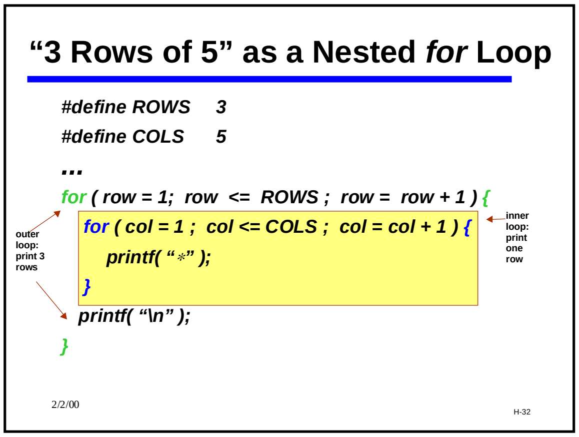
```
/* Different style of counting */
for (  count = 0 ; count < n ;  count = count + 1 ) {
    printf ( “*” );
}
/* could also use count <= n-1 */
```

```
void puzzler (int a, int b) {

        if (a == b || a <= 0 || b <= 0) {
                printf ("look");
                return;
        }
        switch (a) {
        case 1:
                if (b <= a) {
                        printf ("both");
                        return;
                }
                printf ("ways");
                break;
        case 2:
                if ((b < 3*a) && (b % 2 == 0)) {
                        printf ("before");
                }
                printf ("crossing");
                break;
        default:
                if (b > a) printf (“the”);
                else
                        printf ("street");
        }
}
```



Debug Practice

- You’re executing a program that calls puzzler()
- “**ways**” is being displayed when the program runs
- Question: *what does this tell you about the values of a and b?*
- What if it was “**before**” that was being displayed instead... what would that tell you about the values of a and b?

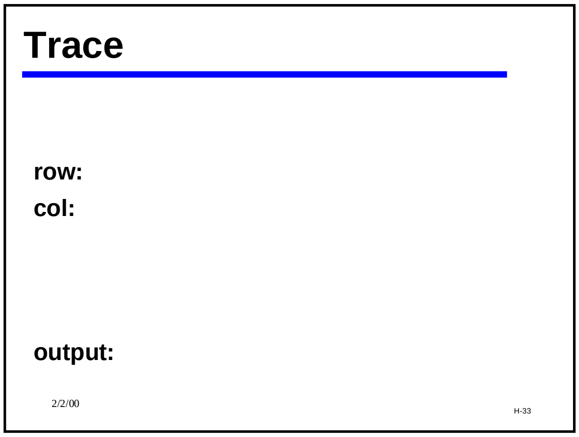


“3 Rows of 5” as a Nested *for* Loop

```
#define ROWS    3
#define COLS    5
...
for ( row = 1;  row  <=  ROWS ;  row =  row + 1 ) {
    for ( col = 1 ;  col <= COLS ;  col = col + 1 ) {
        printf( “*” );
    }
    printf( “\n” );
}
```

outer loop: print 3 rows

inner loop: print one row

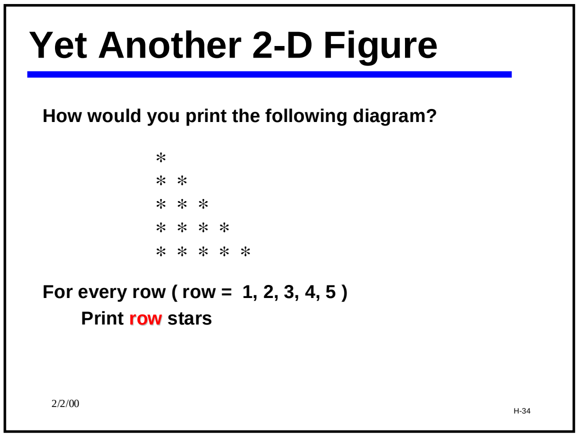


Trace

row:

col:

output:



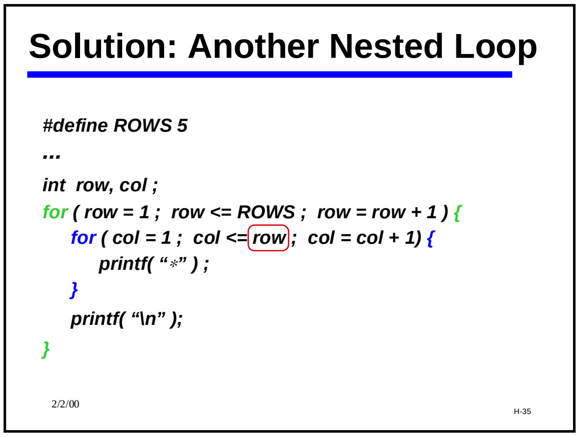
Yet Another 2-D Figure

How would you print the following diagram?

```
*
* *
* * *
* * * *
* * * * *
```

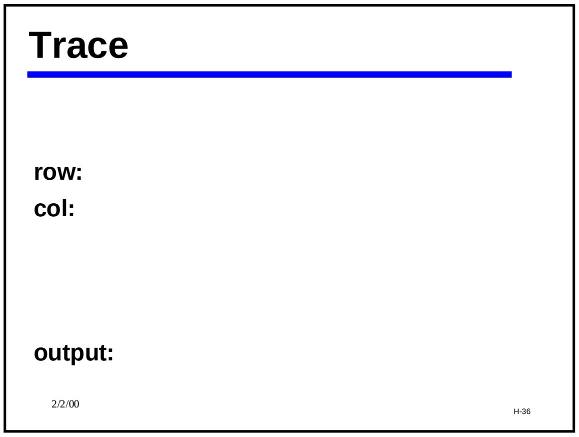
For every row (row = 1, 2, 3, 4, 5)

Print row stars



Solution: Another Nested Loop

```
#define ROWS 5
...
int  row, col ;
for ( row = 1 ;  row <= ROWS ;  row = row + 1 ) {
    for ( col = 1 ;  col <= row ;  col = col + 1) {
        printf( “*” ) ;
    }
    printf( “\n” );
}
```

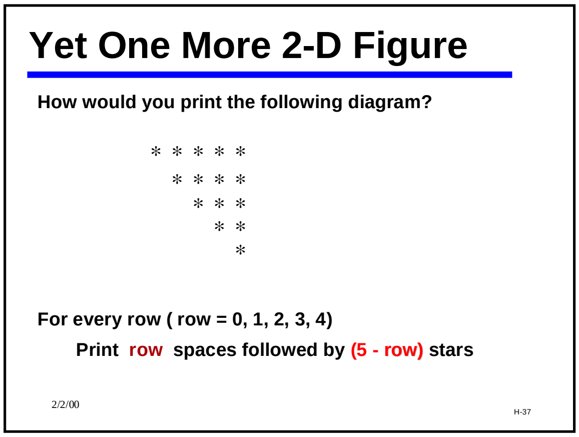


Trace

row:

col:

output:



Yet One More 2-D Figure

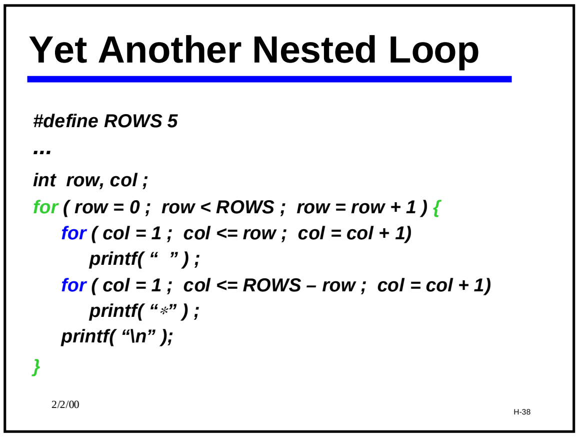
How would you print the following diagram?

```
* * * * *
  * * * *
    * * *
      * *
        *
```

For every row (row = 0, 1, 2, 3, 4)

Print row spaces followed by (5 - row) stars

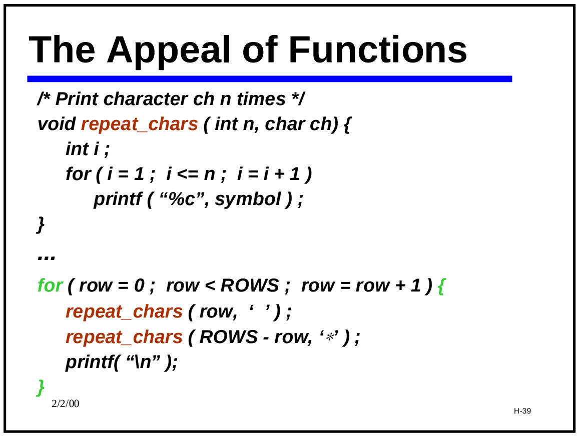
2/2/00 H-37



Yet Another Nested Loop

```
#define ROWS 5
...
int row, col ;
for ( row = 0 ; row < ROWS ; row = row + 1 ) {
    for ( col = 1 ; col <= row ; col = col + 1)
        printf( “  ” ) ;
    for ( col = 1 ; col <= ROWS – row ; col = col + 1)
        printf( “*” ) ;
    printf( “\n” );
}
```

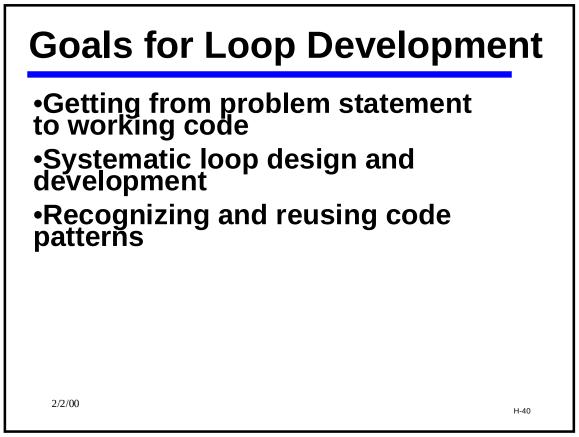
2/2/00 H-38



The Appeal of Functions

```
/* Print character ch n times */
void repeat_chars ( int n, char ch) {
    int i ;
    for ( i = 1 ; i <= n ; i = i + 1 )
        printf ( “%c”, symbol ) ;
}
...
for ( row = 0 ; row < ROWS ; row = row + 1 ) {
    repeat_chars ( row, ‘ ’ ) ;
    repeat_chars ( ROWS - row, ‘*’ ) ;
    printf( “\n” );
}
```

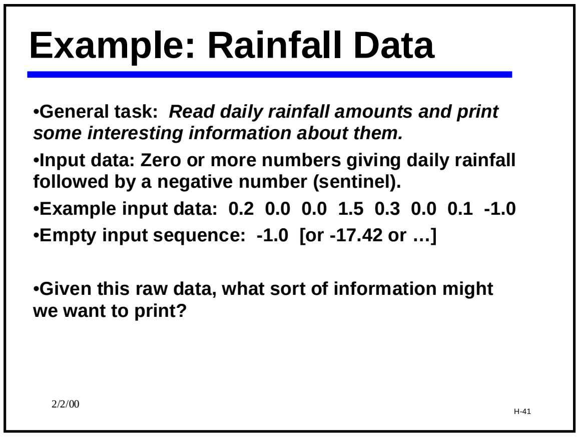
2/2/00 H-39



Goals for Loop Development

- Getting from problem statement to working code
- Systematic loop design and development
- Recognizing and reusing code patterns

2/2/00 H-40

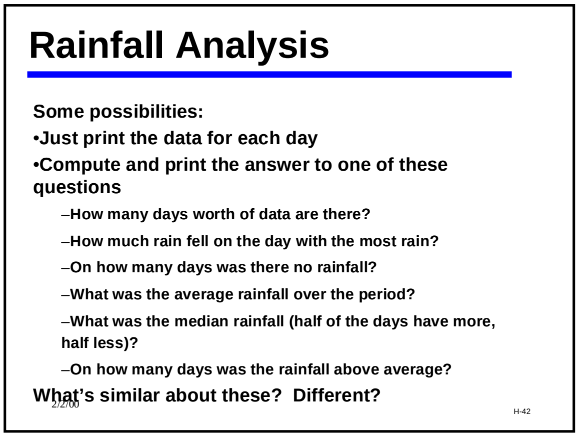


Example: Rainfall Data

- General task: *Read daily rainfall amounts and print some interesting information about them.*
- Input data: Zero or more numbers giving daily rainfall followed by a negative number (sentinel).
- Example input data: 0.2 0.0 0.0 1.5 0.3 0.0 0.1 -1.0
- Empty input sequence: -1.0 [or -17.42 or …]

- Given this raw data, what sort of information might we want to print?

2/2/00 H-41



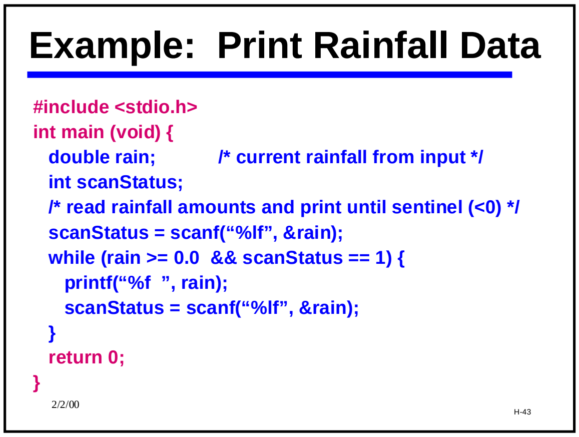
Rainfall Analysis

Some possibilities:

- Just print the data for each day
- Compute and print the answer to one of these questions
  - How many days worth of data are there?
  - How much rain fell on the day with the most rain?
  - On how many days was there no rainfall?
  - What was the average rainfall over the period?
  - What was the median rainfall (half of the days have more, half less)?
  - On how many days was the rainfall above average?

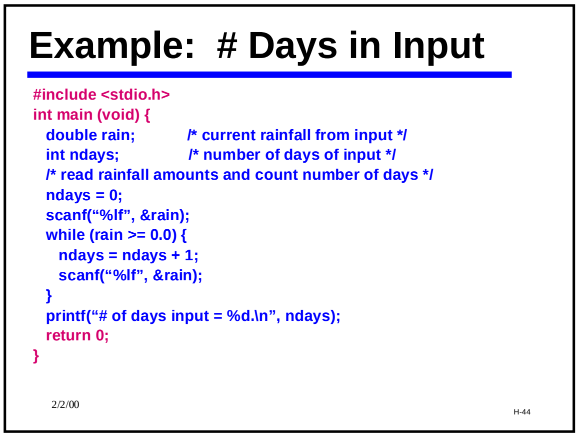
What’s similar about these? Different?

2/2/00 H-42



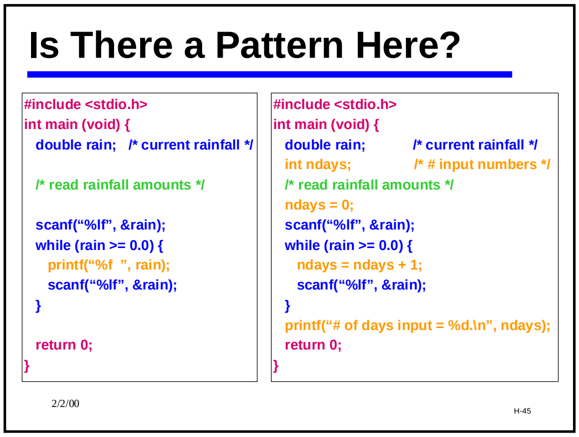
Example: Print Rainfall Data

```
#include <stdio.h>
int main (void) {
   double rain;          /* current rainfall from input */
   int scanStatus;
   /* read rainfall amounts and print until sentinel (<0) */
   scanStatus = scanf(“%lf”, &rain);
   while (rain >= 0.0  && scanStatus == 1) {
      printf(“%f  ”, rain);
      scanStatus = scanf(“%lf”, &rain);
   }
   return 0;
}
```



Example: # Days in Input

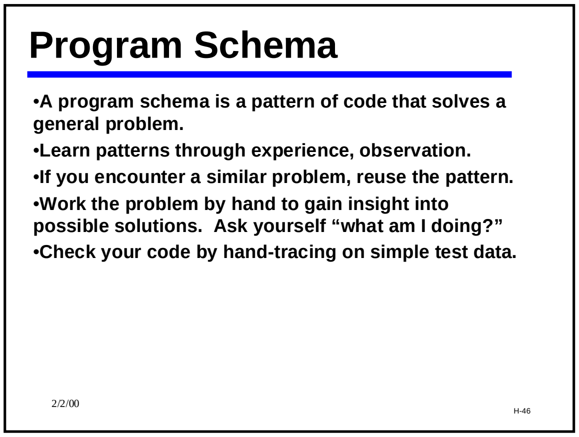
```
#include <stdio.h>
int main (void) {
   double rain;          /* current rainfall from input */
   int ndays;            /* number of days of input */
   /* read rainfall amounts and count number of days */
   ndays = 0;
   scanf(“%lf”, &rain);
   while (rain >= 0.0) {
      ndays = ndays + 1;
      scanf(“%lf”, &rain);
   }
   printf(“# of days input = %d.\n”, ndays);
   return 0;
}
```



Is There a Pattern Here?

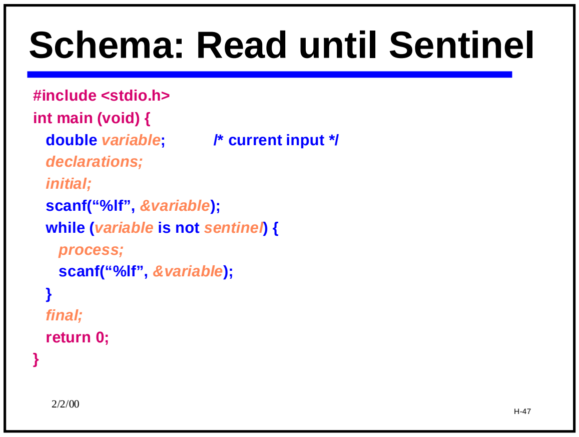
```
#include <stdio.h>
int main (void) {
   double rain;   /* current rainfall */

   /* read rainfall amounts */

   scanf(“%lf”, &rain);
   while (rain >= 0.0) {
      printf(“%f  ”, rain);
      scanf(“%lf”, &rain);
   }

   return 0;
}
```

```
#include <stdio.h>
int main (void) {
   double rain;        /* current rainfall */
   int ndays;          /* # input numbers */
   /* read rainfall amounts */
   ndays = 0;
   scanf(“%lf”, &rain);
   while (rain >= 0.0) {
      ndays = ndays + 1;
      scanf(“%lf”, &rain);
   }
   printf(“# of days input = %d.\n”, ndays);
   return 0;
}
```



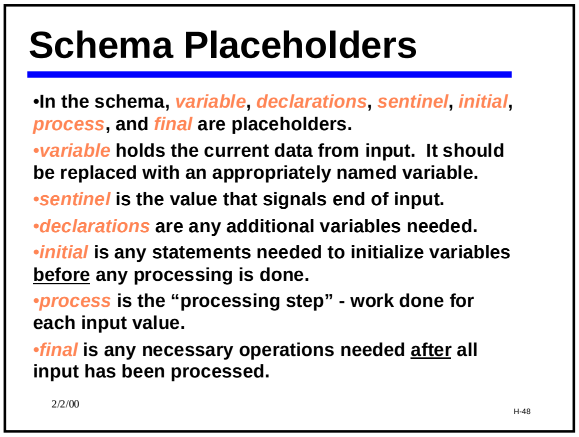
Program Schema

- •A program schema is a pattern of code that solves a general problem.
- •Learn patterns through experience, observation.
- •If you encounter a similar problem, reuse the pattern.
- •Work the problem by hand to gain insight into possible solutions. Ask yourself “what am I doing?”
- •Check your code by hand-tracing on simple test data.



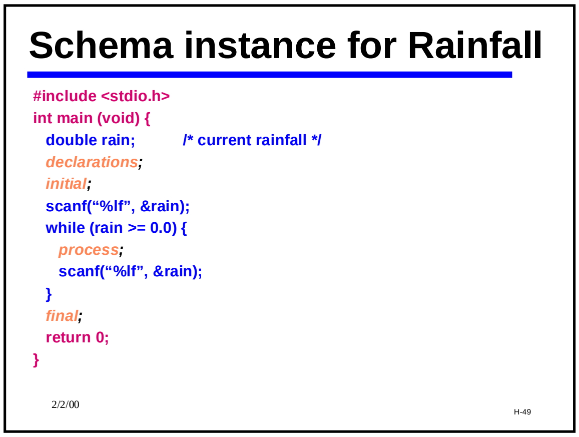
Schema: Read until Sentinel

```
#include <stdio.h>
int main (void) {
   double variable;          /* current input */
   declarations;
   initial;
   scanf(“%lf”, &variable);
   while (variable is not sentinel) {
      process;
      scanf(“%lf”, &variable);
   }
   final;
   return 0;
}
```



Schema Placeholders

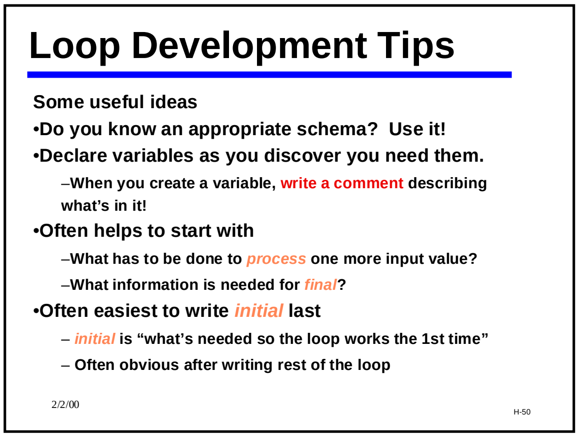
- •In the schema, *variable*, *declarations*, *sentinel*, *initial*, *process*, and *final* are placeholders.
- •*variable* holds the current data from input. It should be replaced with an appropriately named variable.
- •*sentinel* is the value that signals end of input.
- •*declarations* are any additional variables needed.
- •*initial* is any statements needed to initialize variables before any processing is done.
- •*process* is the “processing step” - work done for each input value.
- •*final* is any necessary operations needed after all input has been processed.



Schema instance for Rainfall

```
#include <stdio.h>
int main (void) {
   double rain;          /* current rainfall */
   declarations;
   initial;
   scanf("%lf", &rain);
   while (rain >= 0.0) {
      process;
      scanf("%lf", &rain);
   }
   final;
   return 0;
}
```

2/2/00 H-49

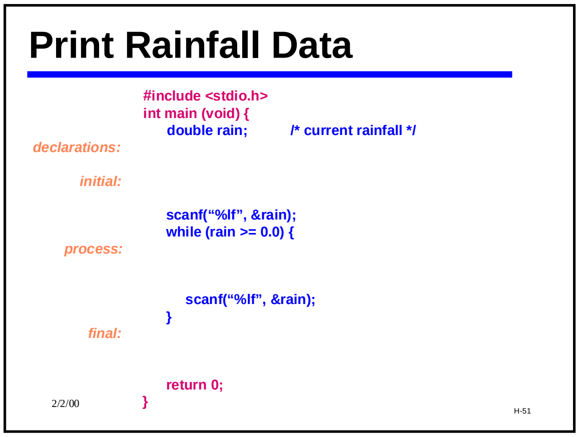


Loop Development Tips

Some useful ideas

- Do you know an appropriate schema? Use it!
- Declare variables as you discover you need them.
  - When you create a variable, write a comment describing what's in it!
- Often helps to start with
  - What has to be done to *process* one more input value?
  - What information is needed for *final*?
- Often easiest to write *initial* last
  - *initial* is "what's needed so the loop works the 1st time"
  - Often obvious after writing rest of the loop

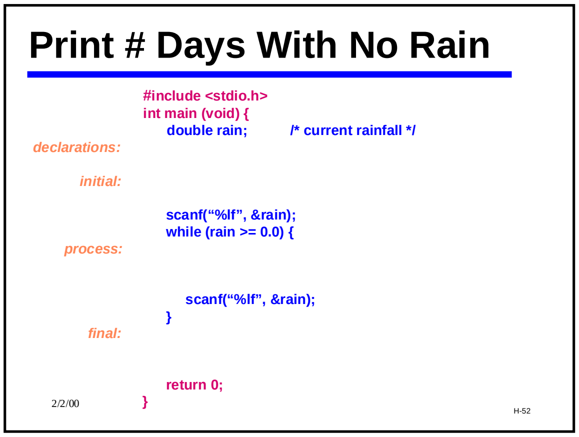
2/2/00 H-50



Print Rainfall Data

```
                 #include <stdio.h>
                 int main (void) {
                     double rain;          /* current rainfall */
declarations:

        initial:

                     scanf("%lf", &rain);
                     while (rain >= 0.0) {
     process:


                         scanf("%lf", &rain);
                     }
         final:


                     return 0;
                 }
```

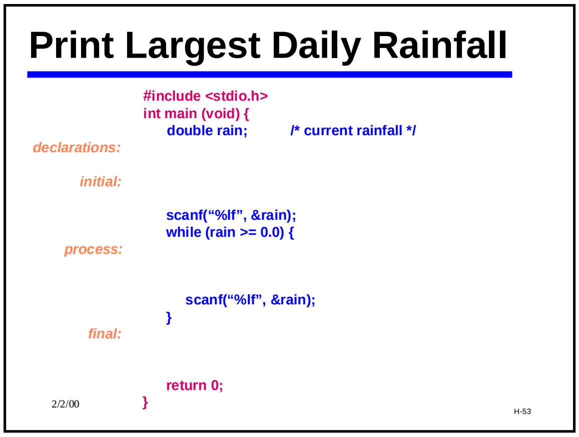
2/2/00 H-51



Print # Days With No Rain

```
                 #include <stdio.h>
                 int main (void) {
                     double rain;          /* current rainfall */
declarations:

        initial:

                     scanf("%lf", &rain);
                     while (rain >= 0.0) {
     process:


                         scanf("%lf", &rain);
                     }
         final:


                     return 0;
                 }
```

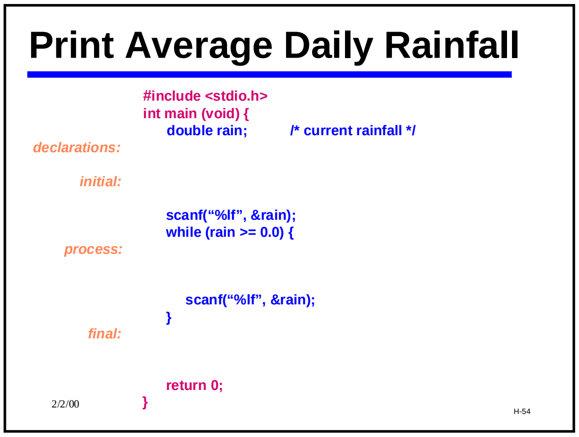
2/2/00 H-52



Print Largest Daily Rainfall

```
                 #include <stdio.h>
                 int main (void) {
                     double rain;          /* current rainfall */
declarations:

        initial:

                     scanf("%lf", &rain);
                     while (rain >= 0.0) {
     process:


                         scanf("%lf", &rain);
                     }
         final:


                     return 0;
                 }
```

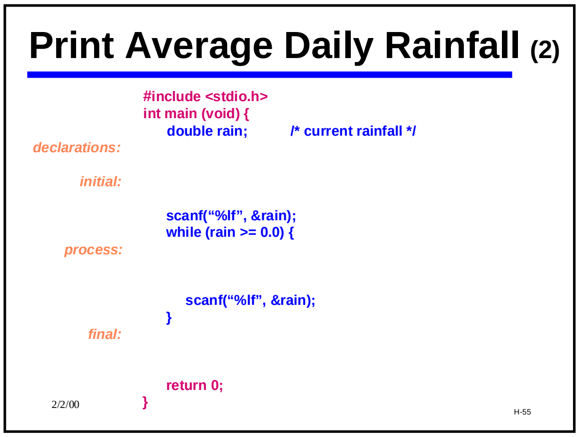
2/2/00 H-53



Print Average Daily Rainfall

```
                 #include <stdio.h>
                 int main (void) {
                     double rain;          /* current rainfall */
declarations:

        initial:

                     scanf("%lf", &rain);
                     while (rain >= 0.0) {
     process:


                         scanf("%lf", &rain);
                     }
         final:


                     return 0;
                 }
```

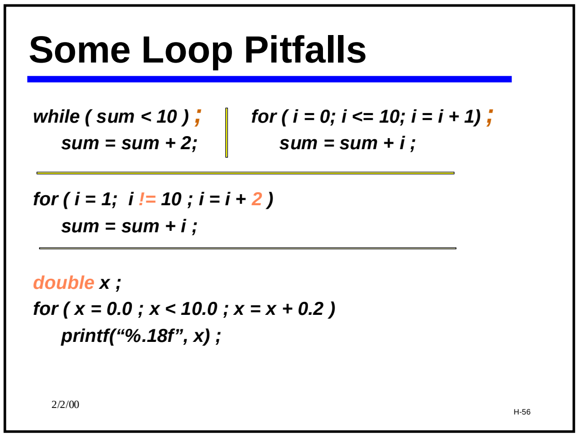
2/2/00 H-54



Print Average Daily Rainfall (2)

```
                #include <stdio.h>
                int main (void) {
                    double rain;        /* current rainfall */
declarations:

      initial:

                    scanf("%lf", &rain);
                    while (rain >= 0.0) {
     process:

                        scanf("%lf", &rain);
                    }
       final:

                    return 0;
                }
```

2/2/00 H-55



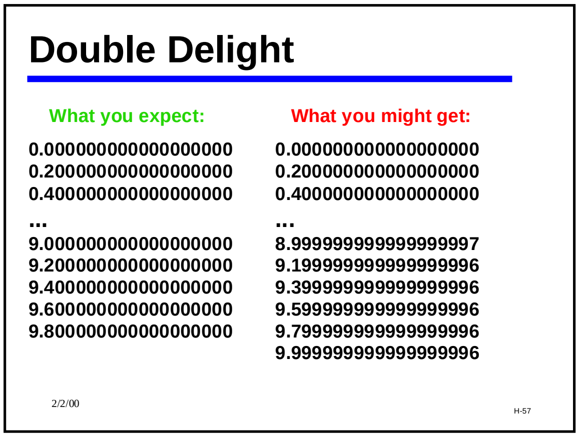
Some Loop Pitfalls

```
while ( sum < 10 ) ;          |  for ( i = 0; i <= 10; i = i + 1) ;
    sum = sum + 2;            |      sum = sum + i ;
```

```
for ( i = 1;  i != 10 ; i = i + 2 )
    sum = sum + i ;
```

```
double x ;
for ( x = 0.0 ; x < 10.0 ; x = x + 0.2 )
    printf("%.18f", x) ;
```

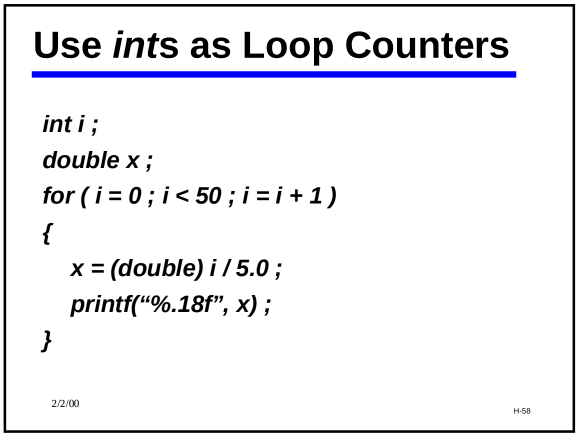
2/2/00 H-56



Double Delight

```
  What you expect:        What you might get:
0.000000000000000000    0.000000000000000000
0.200000000000000000    0.200000000000000000
0.400000000000000000    0.400000000000000000
...                     ...
9.000000000000000000    8.999999999999999997
9.200000000000000000    9.199999999999999996
9.400000000000000000    9.399999999999999996
9.600000000000000000    9.599999999999999996
9.800000000000000000    9.799999999999999996
                        9.999999999999999996
```

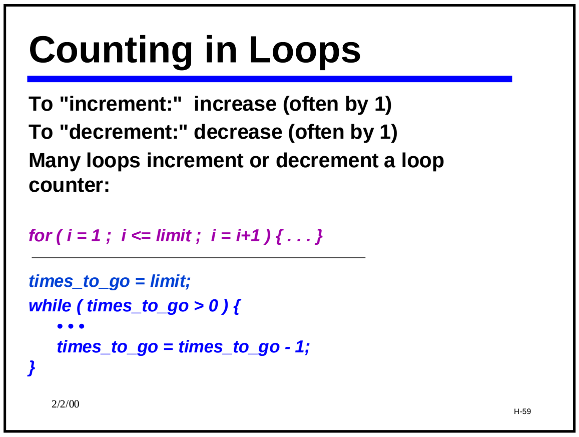
2/2/00 H-57



Use *ints* as Loop Counters

```
int i ;
double x ;
for ( i = 0 ; i < 50 ; i = i + 1 )
{
    x = (double) i / 5.0 ;
    printf("%.18f", x) ;
}
```

2/2/00 H-58



Counting in Loops

To "increment:" increase (often by 1)

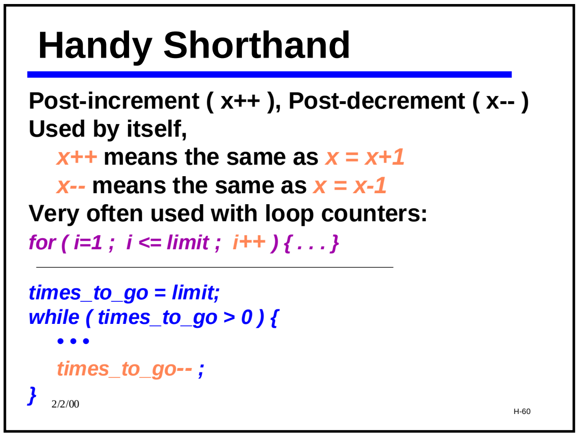
To "decrement:" decrease (often by 1)

Many loops increment or decrement a loop counter:

```
for ( i = 1 ;  i <= limit ;  i = i+1 ) { . . . }
```

```
times_to_go = limit;
while ( times_to_go > 0 ) {
    • • •
    times_to_go = times_to_go - 1;
}
```

2/2/00 H-59



Handy Shorthand

Post-increment (x++), Post-decrement (x--)

Used by itself,

x++ means the same as *x = x+1*

x-- means the same as *x = x-1*

Very often used with loop counters:

```
for ( i=1 ;  i <= limit ;  i++ ) { . . . }
```

```
times_to_go = limit;
while ( times_to_go > 0 ) {
    • • •
    times_to_go-- ;
}
```

2/2/00 H-60

Surgeon General's Warning

- ++ and -- are unary operators.
- Pre-increment (++x) and pre-decrement (--x) exist, too.
- For CSE142, use only in isolation. Don't combine these with other operators in expressions!

  E.g., don't try x = y++ / (3 * --x--)

Iteration Summary

- General pattern:
  - initialize
  - test
  - do stuff
  - update
  - go back to re-test, re-do stuff, re-update, ...
- "while" and "for" are equally general in C
- use "for" when initialize/test/update are closely related and simple, especially when counting

Event-Driven Programming

- Modern programs tend to be "event-driven"
  - – Program starts, sets itself up.
  - – Program enters a loop, waiting for some event or command to happen:
    - mouse click, key click, timer, menu selection, etc.
  - – Program performs operation ("handles" the event or command)
  - – Program goes back to its wait loop
- GP142 programs follow this model

Simple Command Interpreter

Read in "commands" and execute them.

Input - single characters

a -- execute command A by calling *A_handler( )*

b -- execute command B by calling *B_handler( )*

q -- quit

Pseudocode for main loop:

```
get next command
if a, execute command A
if b, execute command B
if q, signal quit
```

Command Interpreter Loop Control Schema

repeat until quit signal

use variable "done" to indicate when done

```
set done to false
while not done
    body statements
    if quit command, set done to true
```

Command Interpreter main ( )

```
#define FALSE 0
#define TRUE   1

int main(void) {
    char command;
    int done;

    done = FALSE;
    while (! done){
                        /* Input command from user */
        command = ReadCommand( );

        switch (command){
        case ‘A’:
        case ‘a’:
            A_handler();        /* Execute command A */
            break;
        case ‘B’:
        case ‘b’:
            B_handler();         /* Execute command B */
            break;
        case ‘Q’:
        case ‘q’:
            done = TRUE;         /* quit */
            break;
        default:
            printf(“Unrecognized command\n”);
        }
    }
    return 0;
}
```