for Loops Outline

1. for Loops Outline
2. A while Loop That Counts
3. Count-Controlled Loops
4. Count-Controlled Loop Flowchart
5. Arithmetic Assignment Operators
6. Increment & Decrement Operators
7. for Loops
8. for Loop Flowchart
9. Three Programs That Behave Identically
10. Identical Behavior: Proof
11. Identical Behavior: Proof (continued)
12. for Loop
13. for Loop Details
14. for Loop Application
15. for Loop Application (continued)
16. for Loop with an Explicit Increment
17. for Loop w/Explicit Increment (continued)
18. for Loop with a Negative Increment
19. for Loop with Named Constants
20. for Loop with Variables
21. for Loop with Expressions
22. for Loop with a float Counter: BAD BAD BAD
23. Why float Counters Are BAD BAD BAD
24. Replacing a float Counter with an int Counter
25. Debugging a for Loop
26. Debugging a for Loop: printf Statements in the Loop Body
27. Debugging a for Loop: printf Statements (continued)
28. Debugging a for Loop: Removing printf Statements
29. Nesting for Loops Inside if Blocks and Vice Versa
30. Nesting for Loop Inside if Block Example Run
31. Nested for Loops
32. Output of Nested for Loop Example
33. Changing the Loop Bounds Inside the Loop: BAD BAD BAD!
34. Changing the Loop Index Inside the Loop: BAD BAD BAD!
A while Loop That Counts

% cat whilecount.c
#include <stdio.h>
int main ()
{ /* main */
    int initial_value, final_value;
    int count;
    int sum;

    printf("What value would you like to ");
    printf("start counting at?\n");
    scanf("%d", &initial_value);
    printf("What value would you like to ");
    printf("stop counting at,\n");
    printf(" which must be greater than ");
    printf("or equal to %d?\n", initial_value);
    scanf("%d", &final_value);
    if (final_value < initial_value) {
        printf("ERROR: the final value %d
", final_value);
        printf(" is less than the initial ");
        printf("value %d.\n", initial_value);
        exit(-1);
    } /* if (final_value < initial_value) */
    sum = 0;
    count = initial_value;
    while (count <= final_value) {
        sum = sum + count;
        count = count + 1;
    } /* while (count <= final_value) */
    printf("The sum of the integers from ");
    printf("%d through %d is %d.\n", initial_value, final_value, sum);
    return 0;
} /* main */

% gcc -o whilecount whilecount.c
% whilecount
What value would you like to start counting at?
1
What value would you like to stop counting at, which must be greater than or equal to 1?
0
ERROR: the final value 0 is less than the initial value 1.
% whilecount
What value would you like to start counting at?
1
What value would you like to stop counting at, which must be greater than or equal to 1?
5
The sum of the integers from 1 through 5 is 15.
Count-Controlled Loops

On the previous slide, we saw a case of a loop that executes a specific number of iterations, by using a counter variable that is initialized to a particular initial value and is incremented (increased by 1) at the end of each iteration of the loop, until it goes beyond a particular final value:

```c
sum = 0;
count = initial_value;
while (count <= final_value) {
    sum = sum + count;
    count = count + 1;
} /* while (count <= final_value) */
```

We call this kind of loop a count-controlled loop. If we express a count-controlled loop as a while loop, then the general form is:

```c
counter = initial_value;
while (counter <= final_value) {
    statement1;
    statement2;
    ...
    counter = counter + 1;
} /* while (counter <= final_value) */
```

Count-controlled loops are among the most commonly used kinds of loops. They’re so common that we have a special construct for them, called a for loop.
Count-Controlled Loop Flowchart

counter = initial_value;
while (counter <= final_value) {
    statement_inside1;
    statement_inside2;
    ...
    counter = counter + 1;
}
statement_after;
Arithmetic Assignment Operators

Some while back, we saw the following:

\[ x = x + y; \]

We learned that this statement increases the value of \( x \) by \( y \). That is, the statement takes the old value of \( x \), adds \( y \) to it, then assigns the result of this addition to \( x \).

This kind of statement is so common that the C language has a special operator for it, called the *addition assignment operator*:

\[ x += y; \]

Note that the two statements above behave identically.

C also has arithmetic assignment operators for the other arithmetic operations:

<table>
<thead>
<tr>
<th>This:</th>
<th>is identical to this:</th>
<th>Operation name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x ) += ( y );</td>
<td>( x = x + y; )</td>
<td>Addition assignment</td>
</tr>
<tr>
<td>( x ) -= ( y );</td>
<td>( x = x - y; )</td>
<td>Subtraction assignment</td>
</tr>
<tr>
<td>( x ) *= ( y );</td>
<td>( x = x * y; )</td>
<td>Multiplication assignment</td>
</tr>
<tr>
<td>( x ) /= ( y );</td>
<td>( x = x / y; )</td>
<td>Division assignment</td>
</tr>
<tr>
<td>( x ) %= ( y );</td>
<td>( x = x % y; )</td>
<td>Remainder assignment</td>
</tr>
</tbody>
</table>

Note that these arithmetic assignment operators are *syntactic sugar*: they don’t add any new capability to the language, but they make programming more convenient.
Increment & Decrement Operators

One of the most common addition assignments is:

\[ x = x + 1; \]

We learned that this statement increases the value of \( x \) by 1. That is, the statement takes the old value of \( x \), adds 1 to it, then assigns the result of this addition to \( x \).

For this statement, we could use the addition assignment operator:

\[ x += 1; \]

But this statement is MUCH MUCH MUCH more common than

\[ x += y; \]

for generic \( y \), so the C language has another special operator, called the increment operator:

\[ x++; \]

C also has a decrement operator, which subtracts 1:

<table>
<thead>
<tr>
<th>This:</th>
<th>is identical to this:</th>
<th>is identical to this:</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x++; )</td>
<td>( x += 1; )</td>
<td>( x = x + 1; )</td>
<td>Increment</td>
</tr>
<tr>
<td>( x--; )</td>
<td>( x -= 1; )</td>
<td>( x = x - 1; )</td>
<td>Decrement</td>
</tr>
</tbody>
</table>

Note that the increment and decrement operators are syntactic sugar, just like the arithmetic assignment operators.
for Loops

A for loop has this form:

```c
for (counter = initial_value;
     counter <= final_value; counter++) {
    statement1;
    statement2;
    ...
} /* for counter */
```

A for loop behaves exactly the same as a count-controlled while loop:

```c
counter = initial_value;
while (counter <= final_value) {
    statement1;
    statement2;
    ...
    counter = counter + 1;
} /* while (counter <= final_value) */
```
for Loop Flowchart

for (counter = initial_value; counter <= final_value; counter++) {
    statement_inside1;
    statement_inside2;
    ...
}
statement_after;

for (counter = initial_value; counter <= final_value; counter++) {
    statement_inside1;
    statement_inside2;
    ...
    counter = counter + 1
}
statement_after;
Three Programs That Behave Identically

```c
#include <stdio.h>

int main ()
{ /* main */
    int count;
    int sum = 0;

    count = 1;
    sum = sum + count;
    count = count + 1;
    sum = sum + count;
    count = count + 1;
    sum = sum + count;
    count = count + 1;
    sum = sum + count;
    count = count + 1;
    sum = sum + count;
    count = count + 1;
    printf("count = %d\n", count);
    printf("sum = %d\n", sum);
    return 0;
} /* main */
```

```c
#include <stdio.h>

int main ()
{ /* main */
    int count;
    int sum = 0;

    count = 1;
    while (count <= 5) {
        sum = sum + count;
        count = count + 1;
    } /* while (count <= 5) */
    printf("count = %d\n", count);
    printf("sum = %d\n", sum);
    return 0;
} /* main */
```

```c
#include <stdio.h>

int main ()
{ /* main */
    int count;
    int sum = 0;

    for (count = 1; count <= 5; count++) {
        sum = sum + count;
    } /* for count */
    printf("count = %d\n", count);
    printf("sum = %d\n", sum);
    return 0;
} /* main */
```
Identical Behavior: Proof

% cat manycountstmts.c
#include <stdio.h>

int main ()
{ /* main */
    int count;
    int sum = 0;

    count = 1;
    sum = sum + count;
    count = count + 1;
    sum = sum + count;
    count = count + 1;
    sum = sum + count;
    count = count + 1;
    sum = sum + count;
    count = count + 1;
    sum = sum + count;
    count = count + 1;
    sum = sum + count;
    printf("count = %d\n", count);
    printf("sum = %d\n", sum);
    return 0;
} /* main */

% gcc -o manycountstmts manycountstmts.c
% manycountstmts
count = 6
sum = 15

% cat while_loop.c
#include <stdio.h>

int main ()
{ /* main */
    int count;
    int sum = 0;

    count = 1;
    while (count <= 5) {
        sum = sum + count;
        count = count + 1;
    } /* while (count <= 5) */
    printf("count = %d\n", count);
    printf("sum = %d\n", sum);
    return 0;
} /* main */

% gcc -o while_loop while_loop.c
% while_loop
count = 6
sum = 15
Identical Behavior: Proof (continued)

% cat while_loop.c
#include <stdio.h>

int main ()
{ /* main */
    int count;
    int sum = 0;

    count = 1;
    while (count <= 5) {
        sum = sum + count;
        count = count + 1;
    } /* while (count <= 5) */
    printf("count = %d\n", count);
    printf("sum = %d\n", sum);
    return 0;
} /* main */
% gcc -o while_loop while_loop.c
% while_loop
count = 6
sum = 15

% cat for_loop.c
#include <stdio.h>

int main ()
{ /* main */
    int count;
    int sum = 0;

    for (count = 1; count <= 5; count++) {
        sum = sum + count;
    } /* for count */
    printf("count = %d\n", count);
    printf("sum = %d\n", sum);
    return 0;
} /* main */
% gcc -o for_loop for_loop.c
% for_loop
count = 6
sum = 15
for Loop

```c
#include <stdio.h>

int main ()
{ /* main */
    int product = 1;
    int count;

    for (count = 1; count <= 5; count++) {
        product = product * count;
    } /* for count */
    printf("After the loop: count = %d,", count);
    printf(" product = %d\n", product);
    return 0;
} /* main */
```

After the loop: count = 6, product = 120

When the `for` statement is encountered:

1. The loop initialization is performed; typically, the loop control variable (sometimes called the loop counter or the loop index) is assigned an initial value (sometimes called a lower bound).
2. The loop continuation condition is evaluated, and if the loop continuation condition evaluates to false (0), then the loop body is skipped, and the program continues on from the first statement after the loop block close.
3. Each statement inside the loop body is executed in sequence.
4. When the end of the loop body is reached (indicated by the block close associated with the block open of the `for` statement), the loop counter is changed by the loop change statement, typically (though not always) by incrementing.
5. The program jumps back up to step 2.

We refer to each trip through the body of the loop as an iteration.
for Loop Details

Suppose you have a for loop that looks like this:

```c
int product = 1;
int count;
for (count = 1; count <= 5; count++) {
    product *= count;
} /* for count */
```

The above program fragment behaves identically the same as:

```c
/* Program Trace */
int product = 1; /* product = 1 */
int count; /* count is undefined */
count = 1; /* count == 1, product == 1 */
product *= count; /* count == 1, product == 1 */
count++; /* count == 2, product == 1 */
product *= count; /* count == 2, product == 2 */
count++; /* count == 3, product == 3 */
product *= count; /* count == 3, product == 6 */
count++; /* count == 4, product == 6 */
product *= count; /* count == 4, product == 24 */
count++; /* count == 5, product == 24 */
product *= count; /* count == 5, product == 120 */
count++; /* count == 6, product == 120 */
```
for Loop Application

Suppose that there’s a line of a dozen students waiting for tickets for the next OU-Texas football game.

How many different orders can they have in line?

- The head of the line could be any student.
- The 2nd position in line could be any student except the student at the head of the line.
- The 3rd position in line could be any student except the student at the head of the line or the student in the 2nd position.

And so on.

Generalizing, we have that the number of different orders of the students is:

$$(12) \ (11) \ (10) \ ... \ (2) \ (1)$$

We can also express this in the other direction:

$$(1) \ (2) \ (3) \ ... \ (12)$$

In fact, for any number of students $n$, we have that the number of orders is:

$$(1) \ (2) \ (3) \ ... \ (n)$$

This arithmetic expression is called “$n$ factorial”, denoted $n!$

We say that there are $n!$ permutations, or orderings, of the $n$ students.
for Loop Application (continued)

The number of permutations of \( n \) objects is:

\[
P(n) = n! = (1) (2) (3) \ldots (n)
\]

Here’s a program that calculates permutations:

```c
#include <stdio.h>

int main ()
{
    int number_of_students;
    int permutations;
    int count;

    printf("How many students are in line for tickets?\n");
    scanf("%d", &number_of_students);
    permutations = 1;
    for (count = 1; count <= number_of_students; count++) {
        permutations = permutations * count;
    }
    printf("There are %d different orders in which\n", permutations);
    printf(" the %d students can stand in line.\n", number_of_students);
    return 0;
}
```

Here's the output of running the program:

```
How many students are in line for tickets?
12
There are 479001600 different orders in which the 12 students can stand in line.
```
for Loop with an Explicit Increment

The most common increment for a for loop is 1. For convenience, therefore, we typically use the increment operator for our loop change statement. For example:

```c
int product = 1;
int count;
for (count = 1; count <= 5; count++) {
    product *= count;
} /* for count */
```

On the other hand, we could state the loop increment explicitly in the for statement, by using, for example, an addition assignment operator:

```c
int product = 1;
int count;
for (count = 1; count <= 5; count += 1) {
    product *= count;
} /* for count */
```

The above two program fragments behave identically. Notice that both of the above loops have 5 iterations:

- count == 1
- count == 2
- count == 3
- count == 4
- count == 5
for Loop w/Explicit Increment (continued)

On the other hand, if the loop increment is not 1, then it must be explicitly stated, using, for example, an addition assignment operator:

```c
int product = 1;
int count;
for (count = 1; count <= 5; count += 2) {
    product *= count;
} /* for count */
```

Notice that the above loop has only 3 iterations:

- count == 1
- count == 3
- count == 5

The above program fragment behaves identically to:

```c
int product = 1;
int count;
count = 1;        /* count == 1, product == 1 */
product *= count; /* count == 1, product == 1 */
count += 2;       /* count == 3, product == 1 */
product *= count; /* count == 3, product == 3 */
count += 2;       /* count == 5, product == 3 */
product *= count; /* count == 5, product == 15 */
count += 2;       /* count == 7, product == 15 */
```
for Loop with a Negative Increment

Sometimes, we want to loop backwards, from a high initial value to a low final value. To do this, we use a negative loop increment; that is, we use the decrement operator --:

```c
#include <stdio.h>
#include <math.h>

int main ()
{
    const int input_digits = 4;
    const int base = 10;
    int base_power, input_value;
    int base_digit_value, output_digit;

    printf("Input an integer of no more ");
    printf("than %d digits: \n", input_digits);
    scanf("%d", &input_value);
    for (base_power = input_digits - 1;
         base_power >= 0; base_power--)
    {
        base_digit_value = pow(base, base_power);
        if (input_value >= base_digit_value)
        {
            output_digit =
                input_value / base_digit_value;
            printf("%2d^%1d: %1d \n",
                base, base_power, output_digit);
            input_value =
                input_value -
                output_digit * base_digit_value;
        } /* if (input_value >= base_digit_value) */
    } /* for base_power */
    return 0;
}
```

```
% gcc -o decimaldigits decimaldigits.c -lm
% decimaldigits
Input an integer of no more than 4 digits:
3984
10^3: 3
10^2: 9
10^1: 8
10^0: 4
```
for Loop with Named Constants

For the loop lower bound and upper bound, and the stride if there is one, we can use `int` named constants:

```c
#include <stdio.h>

int main ()
{
    const int initial_value = 1;
    const int final_value = 20;
    const int stride = 3;
    int count, sum = 0;

    for (count = initial_value;
         count <= final_value; count += stride) {
        sum = sum + count;
        printf("count = %d, sum = %d\n", count, sum);
    } /* for count */
    printf("After loop, count = %d, sum = %d.\n", count, sum);
    return 0;
}

% gcc -o loopbndconsts loopbndconsts.c
% loopbndconsts
count = 1, sum = 1
count = 4, sum = 5
count = 7, sum = 12
count = 10, sum = 22
count = 13, sum = 35
count = 16, sum = 51
count = 19, sum = 70
After loop, count = 22, sum = 70.
```

In fact, we should use `int` named constants rather than `int` literal constants: it’s much better programming practice, because it makes it much easier to change the loop bounds and the stride.
for Loop with Variables

For the loop lower bound, loop upper bound and loop stride, we can use int variables:

```
% cat loopbndvars.c
#include <stdio.h>

int main ()
{
    int initial_value, final_value, stride;
    int count, sum = 0;

    printf("What are the initial, final and ");
    printf("stride values?\n");
    scanf("%d %d %d", &initial_value, &final_value, &stride);
    for (count = initial_value;
        count <= final_value; count += stride) {
        sum = sum + count;
        printf("count = %d, sum = %d\n", count, sum);
    } /* for count */
    printf("After the loop, count = %d, sum = %d.\n", count, sum);
    return 0;
} /* main */
```

% gcc -o loopbndvars loopbndvars.c
% loopbndvars
What are the initial, final and stride values?
  1 7 2
  count = 1, sum = 1
  count = 3, sum = 4
  count = 5, sum = 9
  count = 7, sum = 16
After the loop, count = 9, sum = 16.
for Loop with Expressions

If we don’t happen to have a variable handy that represents one of the loop bounds or the loop increment, then we can use an expression:

```
% cat loopbndexprs.c
#include <stdio.h>
int main()
{ /* main */
    int initial_value, final_value, multiplier;
    int count, sum = 0;

    printf("What are the initial, final and ");
    printf("multiplier values?\n");
    scanf("%d %d %d", 
         &initial_value, &final_value, &multiplier);
    for (count = initial_value * multiplier; 
         count <= final_value * multiplier; 
         count += multiplier - 1) {
        sum = sum + count;
        printf("count = %d, sum = %d\n", count, sum);
    } /* for count */
    printf("After the loop, count = %d, sum = %d.\n", 
            count, sum);
    return 0;
} /* main */
```

% gcc -o loopbndexprs loopbndexprs.c
% loopbndexprs
What are the initial, final and multiplier values?
1 7 2
count = 2, sum = 2
count = 3, sum = 5
count = 4, sum = 9
count = 5, sum = 14
count = 6, sum = 20
count = 7, sum = 27
count = 8, sum = 35
count = 9, sum = 44
count = 10, sum = 54
count = 11, sum = 65
count = 12, sum = 77
count = 13, sum = 90
count = 14, sum = 104
After the loop, count = 15, sum = 104.
for Loop with a float Counter:  
BAD BAD BAD

All of the examples of `for` loops that we’ve seen so far have used `int` counters. In principle, C also supports `float` counters:

```c
#define forreal.c
#include <stdio.h>

int main ()
{
    float real_count;
    float sum = 0.0;

    for (real_count = 1.0;
         real_count <= 10.0; real_count++)
    {
        sum = sum + real_count;
    } /* for real_count */
    printf("After the loop:\n");
    printf(" real_count = %f, sum = %f\n", real_count, sum);
    return 0;
} /* main */
```

```
gcc -o forreal forreal.c

forreal
After the loop:
    real_count = 11.000000, sum = 55.000000
```

This is BAD BAD BAD. Why?
Why float Counters Are BAD BAD BAD

float counters are generally considered to be very poor programming practice, because a float counter is an approximation, and therefore a loop with lots of iterations will accumulate a lot of error in the counter, as the error from each approximation adds up:

% cat forreal2.c
#include <stdio.h>

int main ()
{ /* main */
    const float pi = 3.14;
    float radians;

    for (radians = 0; radians <= 100.0 * pi;
         radians = radians + pi / 5.0) {
        printf("radians = %19.15f\n", radians);
    } /* for radians */
    printf("After the loop:\n");
    printf(" 100 * pi = %19.15f\n", 100.0 * pi);
    printf(" radians = %19.15f\n", radians);
    return 0;
} /* main */
% gcc -o forreal2 forreal2.c
% forreal2
radians = 0.000000000000000
radians = 0.628000020980835
radians = 1.256000041961670
radians = 1.884000062942505
radians = 2.512000083923340
...
radians = 312.116149902343750
radians = 312.744140625000000
radians = 313.372131347656250
After the loop:
100 * pi = 314.000010490417480
radians = 314.000122070312500

This has been a deadly problem in real life. See:

Replacing a float Counter with an int Counter

Happily, we rarely need a float counter, because we can use an int counter and calculate the float value in the loop body:

```
#include <stdio.h>

int main ()
{ /* main */
    const float pi = 3.14;
    float radians;
    int radians_counter;

    for (radians_counter = 0;
        radians_counter <= 500; radians_counter++) {
        radians = radians_counter * pi / 5.0;
        printf("radians = %19.15f\n", radians);
    } /* for radians_counter */
    printf("After the loop:\n");
    printf(" 100.0 * pi = %19.15f\n", 100.0 * pi);
    printf(" radians = %19.15f\n", radians);
    printf(" radians_counter = %3d\n", radians_counter);
    return 0;
} /* main */
```

Notice that there’s no accumulated error from approximating float quantities, because each approximation is independent of the others.
Debugging a for Loop

Suppose you have a program that has a for loop, and it looks like the for loop has a bug in it:

```c
#include <stdio.h>

int main ()
{
    int initial_value, final_value, count;
    int sum = 0;

    printf("What are the summation limits?\n");
    scanf("%d %d", &initial_value, &final_value);
    for (count = initial_value;
         count <= final_value; count++) {
        sum = sum * count;
    } /* for count */
    printf("The sum from %d to %d is %d.\n", 
           initial_value, final_value, sum);
    return 0;
} /* main */
```

```
% gcc -o sumbad sumbad.c
% sumbad
What are the summation limits?
1 5
The sum from 1 to 5 is 0.
```

Assuming that the bug isn’t obvious just from looking, how do we figure out where the bug is?
Debugging a `for` Loop: `printf` Statements in the Loop Body

One thing we can try is to put some `printf` statements inside the loop body:

```c
#include <stdio.h>

int main ()
{
   /* main */
   int initial_value, final_value, count;
   int sum = 0;

   printf("What are the summation limits?\n");
   scanf("%d %d", &initial_value, &final_value);
   for (count = initial_value;
       count <= final_value; count++) {
      sum = sum * count;
      printf("count = %d, sum = %d\n", count, sum);
   } /* for count */
   printf("The sum from %d to %d is %d.\n", initial_value, final_value, sum);
   return 0;
} /* main */
```

Often, the output of the loop body `printf` statements will tell us where to find the bug.
Debugging a for Loop:
printf Statements (Continued)

When we’ve made a change, we can check to make sure things are going well using the same printf statements inside the loop body:

% cat sumgooddebug.c
#include <stdio.h>

int main ()
{ /* main */
   int initial_value, final_value, count;
   int sum = 0;

   printf("What are the summation limits?\n");
   scanf("%d %d", &initial_value, &final_value);
   for (count = initial_value; count <= final_value; count++) {
      sum = sum + count;
      printf("count = %d, sum = %d\n", count, sum);
   } /* for count */
   printf("The sum from %d to %d is %d.\n", initial_value, final_value, sum);
   return 0;
} /* main */
% gcc -o sumgooddebug sumgooddebug.c
% sumgooddebug
What are the summation limits?
1 5
1 count = 1, sum = 1
2 count = 2, sum = 3
3 count = 3, sum = 6
4 count = 4, sum = 10
5 count = 5, sum = 15
The sum from 1 to 5 is 15.
Debugging a for Loop: Removing printf Statements

Once we know that the loop is debugged, we can delete the printf statements inside the loop body:

```
% cat sumgood.c
#include <stdio.h>

int main ()
{
    /* main */
    int initial_value, final_value, count;
    int sum = 0;

    printf("What are the summation limits?\n");
    scanf("%d %d", &initial_value, &final_value);
    for (count = initial_value;
        count <= final_value; count++) {
        sum = sum + count;
    } /* for count */
    printf("The sum from %d to %d is %d.\n",
        initial_value, final_value, sum);
    return 0;
} /* main */
% gcc -o sumgood sumgood.c
% sumgood
What are the summation limits?
1 5
The sum from 1 to 5 is 15.
```
Nesting for Loops Inside if Blocks and Vice Versa

We can nest for loops inside if blocks and if blocks inside for loops:

```c
#include <stdio.h>

int main ()
{
    const int first_prime = 2;
    const int no_remainder = 0, increment = 1;
    int input_value, factor, remainder;
    char is_prime;

    printf("What integer greater than or equal to \%d\n", first_prime);
    printf(" would you like to check to see whether ");
    printf("it’s prime?\n");
    scanf("\%d", &input_value);
    if (input_value < first_prime) {
        printf("Sorry, I can’t determine whether \%d\n", input_value);
        printf(" is prime, because it isn’t ");
        printf("at least \%d.\n", first_prime);
    } /* if (input_value < first_prime) */
    else if (input_value == first_prime) {
        printf("Duh! Of course \%d is a prime!\n", first_prime);
    } /* if (input_value == first_prime) */
    else {
        is_prime = 1;
        factor = first_prime;
        while (is_prime && (factor < input_value)) {
            remainder =
            input_value - ((input_value / factor) * factor);
            if (remainder == no_remainder) {
                is_prime = 0;
            } /* if (remainder == no_remainder) */
            else {
                factor = factor + increment;
            } /* if (remainder == no_remainder)...else */
        } /* while (is_prime && (factor < input_value)) */
        if (is_prime) {
            printf("Yes! \%d is a prime!\n", input_value);
        } /* if (is_prime) */
        else {
            printf("Hey! \%d isn’t a prime!\n", input_value);
            printf("One of its factors is \%d.\n", factor);
        } /* if (is_prime)...else */
    } /* if (input_value == first_prime)...else */
    return 0;
} /* main */
```

We can also nest if blocks inside if blocks inside for loops, and for loops inside if blocks inside if blocks, and so on.
Nested for Loop Inside if Block
Example Run

% gcc -o itisprime itisprime.c
% itisprime
What integer greater than or equal to 2
would you like to check to see whether it’s prime?
1
Sorry, I can’t determine whether 1
is prime, because it isn’t at least 2.
% itisprime
What integer greater than or equal to 2
would you like to check to see whether it’s prime?
2
Duh! Of course 2 is a prime!
% itisprime
What integer greater than or equal to 2
would you like to check to see whether it’s prime?
3
Yes! 3 is a prime!
% itisprime
What integer greater than or equal to 2
would you like to check to see whether it’s prime?
4
Hey! 4 isn’t a prime!
One of its factors is 2.
% itisprime
What integer greater than or equal to 2
would you like to check to see whether it’s prime?
5
Yes! 5 is a prime!
% itisprime
What integer greater than or equal to 2
would you like to check to see whether it’s prime?
6
Hey! 6 isn’t a prime!
One of its factors is 2.
Nested for Loops

#include <stdio.h>

int main ()
{ /* main */
    const int first_prime = 2;
    const int no_remainder = 0;
    const int increment = 1, decrement = -1;
    int initial_value, final_value, loop_increment;
    int this_value, remainder, factor;
    char is_prime;

    printf("What are the loop bounds that you would like\n");
    printf("to check to see which numbers are prime?\n");
    scanf("%d %d", &initial_value, &final_value);
    if (initial_value < first_prime)
    {
        if (final_value < first_prime)
        {
            printf("Hey! None of the values you want are %d\n", first_prime);
            printf("or greater, so none of them can be primes.\n");
            exit(-1);
        } /* if (final_value < first_prime) */
        printf("No value less than %d is prime, so I’ll start at %d.\n", first_prime, first_prime);
        initial_value = first_prime;
    } /* if (initial_value < first_prime) */

    if (final_value < first_prime)
    {
        printf("No value less than %d is prime, ", first_prime);
        printf("so I’ll end at %d.\n", first_prime);
        final_value = first_prime;
    } /* if (final_value < first_prime) */

    if (initial_value > final_value)
    { loop_increment = decrement;
    } /* if (initial_value > final_value) */
    else
    { loop_increment = increment;
    } /* if (initial_value > final_value)...else */
    printf("Primes from %d to %d:\n", initial_value, final_value);
    for (this_value = initial_value;
         this_value <= final_value;
         this_value += loop_increment)
    {
        is_prime = 1;
        factor = first_prime;
        while (is_prime && (factor < this_value))
        {
            remainder =
            this_value - ((this_value / factor) * factor);
            if (remainder == no_remainder)
            {
                is_prime = 0;
            } /* if (remainder == no_remainder) */
            else
            {
                factor = factor + increment;
            } /* if (remainder == no_remainder)...else */
        } /* while (is_prime && (factor < this_value)) */
        if (is_prime)
        {
            printf("%d\n", this_value);
        } /* if (is_prime) */
    } /* for this_value */
    return 0;
} /* main */
Output of Nested for Loop Example

% gcc -o allprimes allprimes.c
% allprimes
What are the loop bounds that you would like to check to see which numbers are prime?
1 2
No value less than 2 is prime, so I’ll start at 2.
Primes from 2 to 2:
2
% allprimes
What are the loop bounds that you would like to check to see which numbers are prime?
2 1
No value less than 2 is prime, so I’ll end at 2.
Primes from 2 to 2:
2
% allprimes
What are the loop bounds that you would like to check to see which numbers are prime?
2 20
Primes from 2 to 20:
2
3
5
7
11
13
17
19
Changing the Loop Bounds Inside the Loop: BAD BAD BAD!

% cat loopbndschg.c
#include <stdio.h>

int main ()
{ /* main */
   int initial_value, final_value, maximum_value;
   int count, sum = 0;

   printf("What are the initial, final and ");
   printf("maximum values?\n");
   scanf("%d %d %d",
         &initial_value, &final_value, &maximum_value);
   for (count = initial_value; count <= final_value; count++) {
      sum = sum + count;
      if (sum > maximum_value) {
         /* BAD BAD BAD BAD BAD BAD BAD */
         final_value = final_value - 1; /* BAD BAD BAD */
      } /* if (sum > maximum_value) */
      printf("count = %d, sum = %d, final_value = %d
",
             count, sum, final_value);
   } /* for count */
   printf("sum = %d\n", sum);
   return 0;
} /* main */

% gcc -o loopbndschg loopbndschg.c
% loopbndschg
What are the initial, final and maximum values?
1 5 3
count = 1, sum = 1, final_value = 5
count = 2, sum = 3, final_value = 5
count = 3, sum = 6, final_value = 4
count = 4, sum = 10, final_value = 3
sum = 10
Changing the Loop Index Inside the Loop:
BAD BAD BAD!

% cat looidxchg.c
#include <stdio.h>

int main ()
{ /* main */
    int initial_value, final_value, maximum_value;
    int count, sum = 0;

    printf("What are the initial, final ");
    printf("and maximum values?\n");
    scanf("%d %d %d",
        &initial_value, &final_value, &maximum_value);
    for (count = initial_value; count <= final_value; count++) {
        sum = sum + count;
        if (sum > maximum_value) {
            /* BAD BAD BAD
             * BAD BAD BAD
             * BAD BAD BAD
             */
            count = count + 1; /* BAD BAD BAD */
        }
        /* BAD BAD BAD
         * BAD BAD BAD
         * BAD BAD BAD
         */
    } /* if (sum > maximum_value) */
    printf("count = %d, sum = %d, final_value = %d\n",
            count, sum, final_value);
    return 0;
} /* main */

% gcc -o looidxchg looidxchg.c
% looidxchg
What are the initial, final and maximum values?
1 5 3
count = 1, sum = 1, final_value = 5
count = 2, sum = 3, final_value = 5
count = 4, sum = 6, final_value = 5
count = 6, sum = 11, final_value = 5
sum = 11