C++ PROGRAMMING

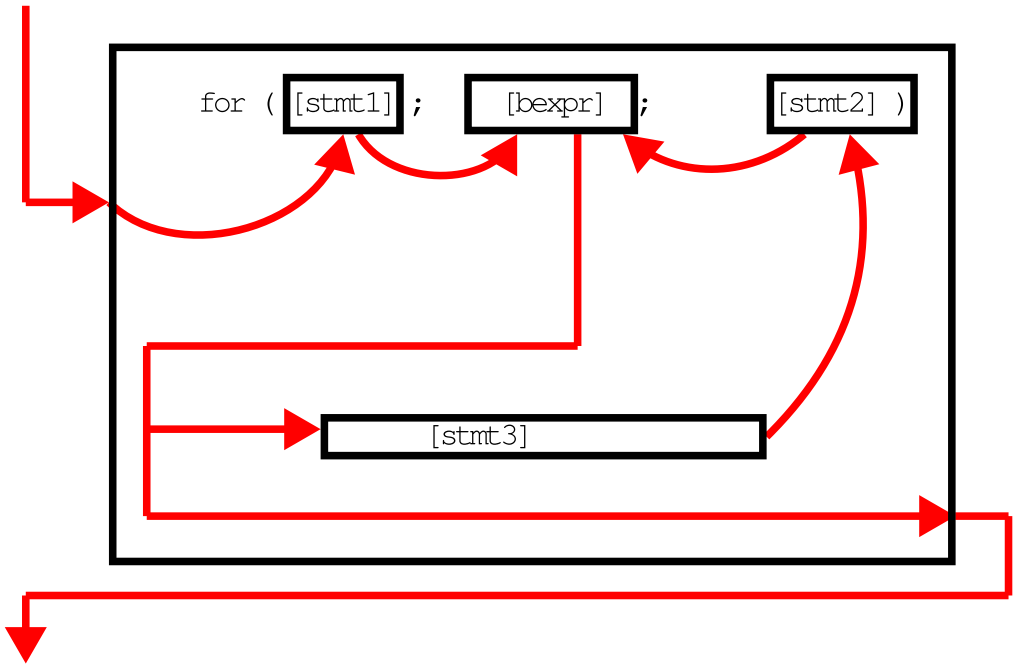
DR. YIHSIANG LIOW (JULY 1, 2025)

Contents

11. for Loops: Part 2

This is a continuation of our discussion of for-loops ...

The three parts of the for-loop



Look at the structure of the for-loop:

The statement looks like this:

```
for ([stmt1]; [bexpr]; [stmt2])
              [stmt3]
```

where `[stmt1]`, `[stmt2]`, `[stmt3]` are statements; `[stmt1]` and `[stmt2]` are written without a ';' since it's included in the above notation.

`[stmt1]` is executed **first** and it's executed exactly **once**.

After executing `[stmt1]`, `[bexpr]` is evaluated.

If the resulting value is `true`, `[stmt3]` is executed. After that `[stmt2]` is executed. And after that we repeat by evaluating `[bexpr]`.

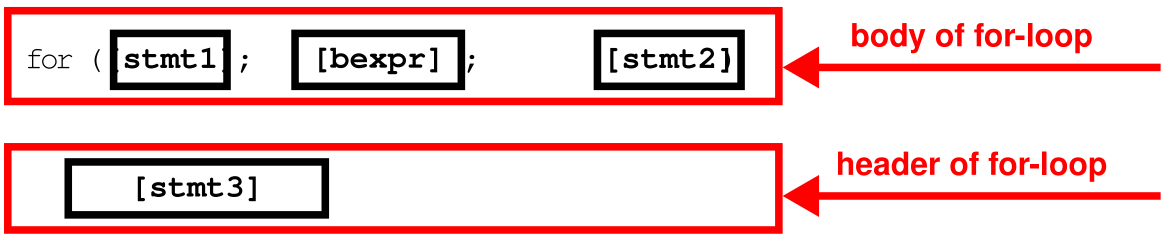
If the resulting value of the `[bexpr]` is `false`, the program exits the for-loop and executes the statement after the for-loop.

By the way, sometimes [stmt1] is sometimes called the **initialization** statement of the for-loop while [stmt2] is called the **update** statement. For the following code:

```
for (int i = 0; i < 100; i++)
{
    std::cout << i << std::endl;
}
```

i is called the **index or counter variable** of the for-loop.

Two final terms:



Leaving things out

Refer to the diagram from the previous section.

Actually it turns out that you can leave out `[stmt1]`. Try this

```
for (int i = 0; i < 5; i++)
{
    std::cout << "i have " << i
              << " head(s)"
              << std::endl;
}
```

And now modify it as follows:

```
int i = 0;
for (; i < 5; i++)
{
    std::cout << "i have " << i
              << " head(s)"
              << std::endl;
}
```

You can also leave the third part out as well!!!

```
int i = 0;
for (; i < 5;)
{
    std::cout << "i have " << i
              << " head(s)"
              << std::endl;
    i++;
}
```

What if the second part is blank? Try this:

```
int i = 0;
for (;;)
{
    std::cout << "i have " << i
              << " head(s)"
              << std::endl;
    i++;
}
```

(Pretty gross, isn't it? ...)

This tells you that if the boolean expression is missing, C++ will assume you want the boolean value to be **`true`**. Remember what I said before: you have to pay attention whenever C++ does something for you automatically.

By the way, the above program is executing an **infinite loop** (because it doesn't stop). This format of the for-loop is sometimes used in writing games, although we'll see that there's another type of loop called the **`while-loop`** that is frequently used instead. Although in the above

```
for (;;)
...
```

it seems like you can't get out of the loop. But, in fact, **you can get out of the loop in another way**. We'll talk about it later.

Why scopes?

Compare the following programs:

```
int i = 0;
int sum = 0;

for (i = 0; i < 10; i++)
{
    sum += i;
}

std::cout << sum << std::endl;
```

```
int sum = 0;

for (int i = 1; i < 11; i++)
{
    sum += i;
}

std::cout << sum << std::endl;
```

Their goals are the same: To compute and print the sum of integers from 1 to 10.

The only difference is in the scope of variable `i`. The scope of `i` is a lot smaller in the second program.

If you think about it, the variable `i` is sort of like a “scratch”variable when compared to `sum`. You don’t really need it once the computation of `sum` is completed. When you read the second program, you can very easily deduce the fact that `i` is a “scratch” variable from it’s scope: it exists only in the `for`-loop.

A complex problem is solved by breaking the problem into smaller and simpler subproblems. Once that’s done you solve the simpler subproblems and put these subsolutions together to solve the big problem. One should be careful of the unintended effects of a subsolution – a small piece of the solution overall – on another.

Here’s an example. This program prints the sum from 1 to 10 and then the sum from 1 to 100:

```
int i;
int sum = 0;

for (i = 1; i < 11; i++)
{
    sum += i;
}

// Print sum from 1 to 10
std::cout << sum << std::endl;

// Continue the summing process
for (; i < 101; i++)
{
    sum += i;
}

std::cout << sum << std::endl;
```

If you're working on a project with someone and you both have agreed that i is a scratch variable, he might use it like this in a computation of a formula involving `a, b, c, d, e, f`

```
i = a + b;
result = i + c + d + e + f;
```

He might just insert this into your code:

```
int i;
int sum = 0;

for (i = 1; i < 11; i++)
{
    sum += i;
}

std::cout << sum << std::endl;

i = a + b;
result = i + c + d + e + f;

for (; i < 101; i++)
{
    sum += i;
}

std::cout << sum << std::endl;
```

Do you see how his work has now destroyed your computation?

In general you want to have as little “communication”as possible between your subsolutions when they are meant to be independent.

One way to achieve that is to use small scopes. Look at this:

```
int i;
int sum = 0;

for (int i = }; i < 11; i++)
{
    sum += i;
}

std::cout << sum << std::endl;

i = a + b;
result = i + c + d + e + f;

for (int i = 1; i < 101; i++)
{
    sum += i;
}

std::cout << sum << std::endl;
```

There are really 3 i’s. The first is outside both for-loops. The second is created only when you enter the first for-loop; this is destroyed when you exit the first for-loop. The third is created only when you enter the second for-loop; this is destroyed when you exit the second for-loop. They all live in different “places”.

Even more examples

Now for examples where the body of the `for`-loop is more complicated.

Of course you can put `if`-statements in the `for`-loop. Run this:

```
int sum = 0;
for (int i = 1; i <= 10; ++i)
{
    if (i == 3 || i > 7)
    {
        sum += i;
        std::cout << i << " " << sum << std::endl;
    }
}
```

Exercise -1.0.1. Modify the above program by adding an else statement to the if statement that prints `i` and `" is not included"`.

Exercise -1.0.2. Write a program that randomly generates 20 integers between 0 and 3.

Exercise -1.0.3. Write a program that randomly generates 20 integers between 10 and 20 and continually sums up all of them, printing the sum in each iteration.

Mixing two for-loops

Of course you can put a `for`-loop in a `for`-loop to get a double `for`-loop:

```
for (int i = 0; i < 5; i++)
{
    std::cout << "i:" << i << std::endl;
    for (int j = 0; j < 3; j++)
    {
        std::cout << " j:" << j << std::endl;
    }
}
```

Read the output very carefully!!!

Exercise -1.0.4. Write a program that has a double for-loop and prints this

```
3
  1 2 3
4
  1 2 3
5
  1 2 3
```

Exercise -1.0.5. Write a program that has a double `for`-loop and prints this

```
3
  0 1 2 3
4
  0 1 2 3 4
5
  0 1 2 3 4 5
```

Exercise -1.0.6. Write a program that has a double for-loop and prints this

```
3
  0 1 2
4
  0 1 2 3
5
  0 1 2 3 4
```

Exercise -1.0.7. Write a program that has a double for-loop and prints this

```
3
  0 1 2 3 4 5 6
4
  0 1 2 3 4 5 6 7 8
5
  0 1 2 3 4 5 6 7 8 9 10
```

Exercise -1.0.8. Write a program that has a double for-loop and prints this

```
3
  3 4 5 6
4
  4 5 6 7 8
5
  5 6 7 8 9 10
```

Exercise -1.0.9. Write the following program that prints a calendar month. The program prompts the user for three integers: the month, the year, and the day-of-week for the first day of the month (with 0 representing Sunday, 1 representing Monday, etc.),. For instance, if the user entered

```
3 2008 6
```

It means that he/she wants the calendar for March 2008 and the first day of the month is a Saturday. The output is

```
March 2008
--------------------
Su Mo Tu We Th Fr Sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

Here's another test case:

```
3 2008 4
March 2008
---------------------

Su Mo Tu We Th Fr Sa
                1 2 3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

Of course your program should be smart enough to compute the number of days in the month (including leap year cases for the month of February.)

Some ASCII art

Now for some (silly) ASCII art.

Try this:

```
int numSpaces = 0;
std::cout << "How far away is the star? ";
std::cin >> numSpaces;

for (int i = 0; i < numSpaces; i++)
{
    std::cout << ' ';
}
std::cout << '*' << std::endl;
```

And this:

```
int numStars = 0;
std::cout << "How many stars? ";
std::cin >> numStars;

for (int i = 0; i < numStars; i++)
{
    std::cout << '*';
}
std::cout << std::endl;
```

And more ASCII art

It can only get worse ...

I want to draw a square of *. The square has width 5 and height 3. First I draw a horizontal line:

```
// Draw horizontal line
for (int i = 0; i < 5; i++)
{
    std::cout << '*';
}
```

OK. Now, I just draw the horizontal line 3 times:

```
for (int j = 0; j < 3; j++)
{
    // Draw horizontal line
    for (int i = 0; i < 5; i++)
    {
        std::cout << '*';
    }
}
```

Oops! How come I get a straight line??? Well, of course after drawing a horizontal line, I need to skip my cursor to the next line:

```
for (int j = 0; j < 3; j++)
{
    // Draw horizontal line
    for (int i = 0; i < 5; i++)
    {
        std::cout << '*';
    }
    std::cout << std::endl;
}
```

The boolean condition in the for-loop can be any boolean expression. So instead of drawing 3 lines, you can prompt the user for number of lines to draw instead.

```
int height = 0;
std::cin >> height;
for (int j = 0; j < height; j++)
{
    // Draw horizontal line
    for (int i = 0; i < 5; i++)
    {
        std::cout << '*';
    }
    std::cout << std::endl;
}
```

Exercise -1.0.10. Now modify the program so that it prompts the user for the width of the square. (Obviously you must draw the square with that width).

Unrolling the `for`-loop: How to see the `for`-loop

Sometimes it's hard to come up with the appropriate for-loop. In that case it's always easier to **"unroll"** the for-loop for several special cases by hand. Unrolling a for-loop is just writing down what the program executes without using the for-loop. For instance, unrolling the following:

```
for (int i = 0; i < 3; i++)
{
    std::cout << i << std::endl;
}

gives us
\begin{console}
std::cout << 0 << std::endl;
std::cout << 1 << std::endl;
std::cout << 2 << std::endl;
```

For instance, suppose you want the program to draw this:

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

if the user enters 4 (this is a triangle). And if he enters 3, the program draws:

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

So just take a special case, say when the user enters 3, and write the pseudocode **without the for-loop**:

```
draw a line of length 1
draw a line of length 2
draw a line of length 3
```

The pseudocode for height 4 is

```
draw a line of length 1
draw a line of length 2
draw a line of length 3
draw a line of length 4
```

Of course you want to write one program and not two! The program must work for both cases. We must find a way to "combine"them.

The first pseudocode ... **NOT C++!!!** ... can be written:

```
for i = 1, 2, 3:
           draw a line of length i
```

and the second can be written:

```
for i = 1, 2, 3, 4:
           draw a line of length i
```

Now they are almost the same! If the height entered by the user is kept in variable h, then the pseudocode is

```
for i = 1, 2, ..., h:
           draw a line of length i
```

AHA! **This pseudocode works for both!!!**

Of course you already know how to draw a line (of stars) of length i. So altogether the pseudocode becomes:

```
for i = 1, 2, ..., h:
         for j = 1, 2, ..., i:
               draw a star
         go to the next line
```

And finally a straightforward translation to C++ gives:

```
for (int i = 1; i <= h; i++)
{
    for (int j = 1; j <= i; j++)
    {
        std::cout << '*';
    }
    std::cout << std::endl;
}
```

Now we add the icing by prompting the user for h:

```
int h = 0;
std::cin >> h;
for (int i = 1; i <= h; i++)
{
    for (int j = 1; j <= i; j++)
    {
        std::cout << '*';
    }
std::cout << std::endl;
}
```

Unrolling the for-loop: Another example

What about this? If the user enters 3 your program has to draw:

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

If the user enters 4 your program has to draw

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

The pseudocode for the case when the user enters 3 is (without using the for-loop):

```
draw 2 spaces
draw 3 stars
draw 1 space
draw 3 stars
draw 3 stars
```

The trick is to make everything as uniform as possible. So write this:

```
draw 2 spaces
draw 3 stars
draw 1 space
draw 3 stars
draw 0 space
draw 3 stars
```

See the pattern yet? No?

The interleaving of two different things is confusing. Rewrite this as:

```
draw 2 spaces; draw 3 stars
draw 1 space; draw 3 stars
draw 0 space; draw 3 stars
```

What if the user enters 4? The pseudocode becomes

```
draw 3 spaces; draw 4 stars
draw 2 spaces; draw 4 stars
draw 1 space; draw 4 stars
draw 0 space; draw 4 stars
```

How do you combine the above cases into one common pseudocode?

The first is

```
for i = 2, 1, 0:
        draw i spaces; draw 3 stars
```

The second is

```
for i = 3, 2, 1, 0:
        draw i spaces; draw 4 stars
```

Now both pseudocodes are almost the same!!! Suppose the value entered by the user is kept in a variable n. Then you can (finally!) combine both pseudocodes into one:

```
for i = n-1, ..., 0:
        draw i spaces; draw n stars
```

Now you add in the details:

```
for i = n-1, ..., 0:
      // draw i spaces
      for j = 1, ..., i
          draw a space
      // draw n stars
      for j = 1, ..., n
          draw a star
      go to next line
```

And finally a straightforward translation to C++ yields:

```
for (int i = n - 1; i >= 0; i--)
{
    for (int j = 1; j <= i; j++)
    {
        std::cout << ' ';
    }
    for (int j = 1; j <= n; j++)
    {
        std::cout << '*';
    }
    std::cout << std::endl;
}
```

Adding the prompt (and some comments) we get:

```
int n = 0;
std::cin >> n;
for (int i = n-1; i >= 0; i--)
{
    // Print i spaces
    for (int j = 1; j <= i; j++)
    {
        std::cout << ' ';
    }
    // Print n stars
    for (int j = 1; j <= n; j++)
    {
        std::cout << '*';
    }
    std::cout << std::endl;
}
```

Exercise -1.0.11. Write a program that draws the following figure when the user enters 3

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

and when the user enters 4 it draws

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

[See next page for hints.]

Exercise -1.0.12. Write a program that draws the following figure when the user enters 3

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

and this when the user enters 4

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

[See next page for hints.]

WARNING: SPOILERS!!!

Here are the pseudocode (with explanations) for the last two ASCII art problems from the previous section.

Here's the first problem . . .

Exercise -1.0.13. Write a program that draws the following figure when the user enters 3

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

and when the user enters 4 it draws

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

and here's the pseudocode with explanation:

```
For n = 4:
    4 stars
    1 star, 2 spaces, 1 star
    1 star, 2 spaces, 1 star
    4 stars
For n = 5:
    5 stars
    1 star, 3 spaces, 1 star
    1 star, 3 spaces, 1 star
    1 star, 3 spaces, 1 star
    5 stars
```

In general, we have:

```
    n stars
    n-2 of [1 star, n-2 stars, 1 star]
    n stars
```

Therefore, the pseudocode is:

```
print n stars
print newline
for i = 1,..., n-2
    print 1 star
    print n-2 stars
    print 1 star
    print newline
print n stars
```

And after expanding some of the print statements we get:

```
//print n stars
for i=1 ,..., n: print '*'
print newline
for i = 1, ..., n-2
    print '*'
    //print n-2 stars
    for j=1,..., n-2: print '*'
    print '*'
    print newline
//print n stars
for i=1, ..., n: print '*'
```

Exercise -1.0.14. Write a program that draws the following figure when the user enters 3:

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

and this when the user enters 4:

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

```
When n = 3:
        2 spaces, 1 star
        1 space, 3 stars
        0 space, 5 stars

When n = 4:
        3 spaces, 1 star
        2 spaces, 3 stars
        1 space, 5 stars
        0 space, 7 stars
```

There are two columns of numbers to handle. In other words, each line (which is the body of a loop) has two numbers. The first column is easy - it's just from n-1 to 0. So the pseudocode looks like this:

```
for i = n-1 to 0
        i spaces, ??? stars
```

What about the second? The relationship between the second number and i is not so direct. But note that it starts with 1 and always goes up by 2 after each iteration. So the pseudocode looks like

```
numStars = 1
for i = n-1 to 0
        i spaces, numStars stars
        numStars = numStars + 2
```

In more details:

```
numStars = 1
for i = n-1 to 0
    print i spaces
    print numStars stars
    print newline
    numStars = numStars + 2
```

And after expanding some print statements we get

```
numStars = 1
for i = n-1 to 0
    //print i spaces
    for j=1, . . . , i: print ”
    // print numStars stars
    for j=1, . . . , numStars: print ’*’
    print newline
    numStars = numStars + 2
```

break

Remember `break` from the `switch` statement? If you execute `break`, you immediately go out of the `switch` block. `break` can also be used in a `for`-loop.

```
First run this:
for (int i = 0; i < 1000; i += 2)
{
std::cout << "entered body of for-loop \n";
std::cout << i << std::endl;
std::cout << "exiting body of for-loop \n";
}
```

Now run this:

```
for (int i = 0; i < 1000; i += 2)
{
    std::cout << "entered body of for-loop\n";
    std::cout << i << std::endl;
    if (i == 100) break;
    std::cout << "exiting body of for-loop\n";
}
```

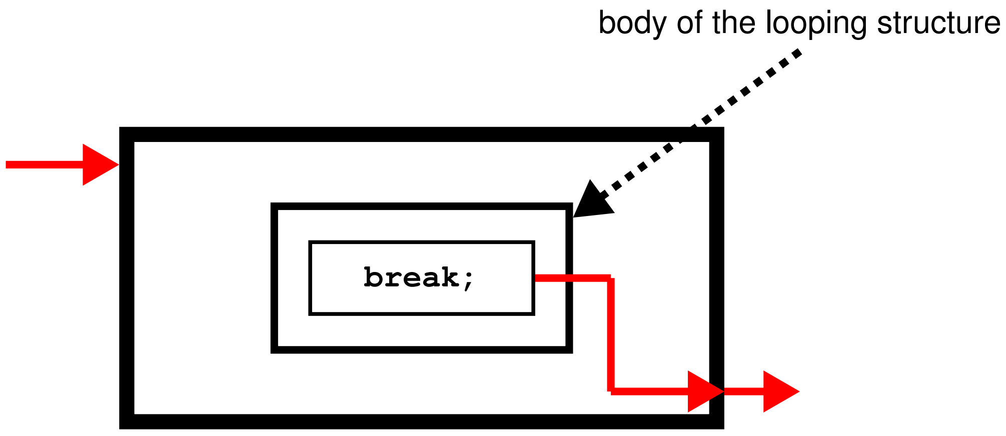
See the point of `break`?

The `break` when executed in a `for`-loop will get out of that `for`-loop. That's all there is to it.

Of course some times you can avoid the break. For the above program you can rewrite it as

```
for (int i = 0; i < 101; i += 2)
{
    std::cout << i << std::endl;
}
```

There's a general principle (or good practice) that you should **avoid having too many exit points out of a loop**. Too many exit points make the code difficult to trace.



It's very important to note that the execution of break only exits the current for-loop. Make sure you trace and run try this:

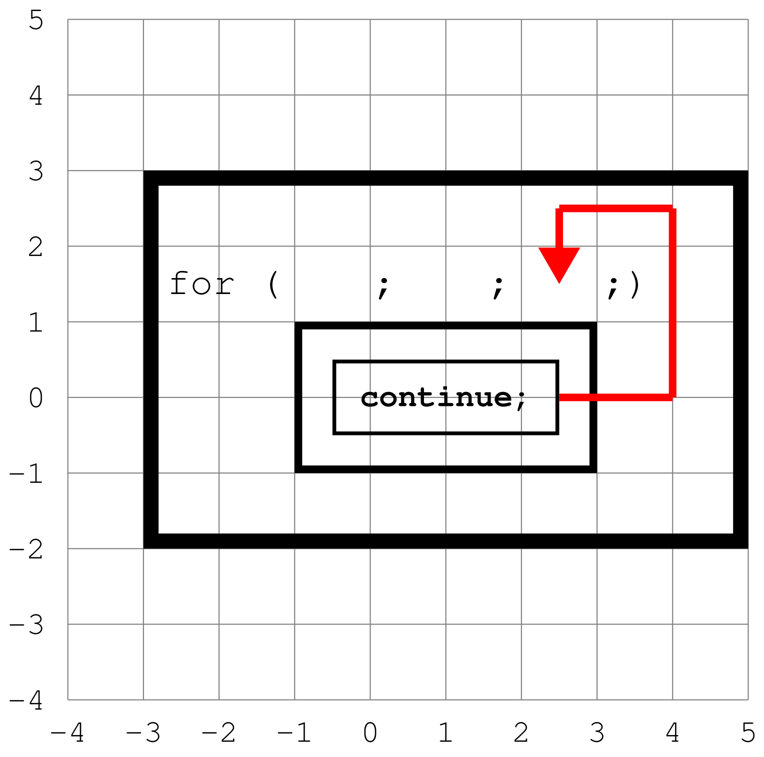
```
for (int i = 0; i < 5; i++)
{
    std::cout << "A: " << i << std::endl;
    for (int j = 0; j < 5; j++)
    {
        std::cout << "B: " << j << std::endl;
        if (j == 1) break;
        std::cout << "C: " << j << std::endl;
    }
    std::cout << "D: " << i << std::endl;
}
```

continue

Another way to control the flow of execution in a for-loop is by using the `continue` statement. Try this and tell me what it does:

```
for (int i = 0; i < 101; i += 2)
{
    std::cout << "top of loop" << std::endl;
    std::cout << i << std::endl;
    if (i > 10) continue;
    std::cout << "bottom of loop" << std::endl;
}
```

A picture tells a thousand words:



Exercise -1.0.15. Write a program that prompts the user for `start`, `end`, and `skip` (altogether there are three integers) and prints the sum of all integers from start to end except for `skip`. Here's the skeleton

```
int start = 0, skip = 0, end = 0;
std::cin >> start >> end >> skip;
int sum = 0;
for (         ;          ;          )
{
    if (          )          ;
    sum +=          ;
}
std::cout << sum << std::endl;
```

For instance, if `start` is -5, `end` is 10, and `skip` is 7, then your program should compute the sum of integers from -5 to 10 (inclusive) except for 7.

Divisors and primes

Quick review of divisors and primes (See previous notes on integers for more information.)

Let n be a positive integer at least 1 and let d be a positive integer greater than 0. We say that d **divides** n or that d is a **divisor** of n if you can find an integer x such that dx = n. (Yes, we've seen this before, in earlier notes and also in the prerequisite math courses.) You already know that the statement

"d divides n"

in C++ is

$$n\%d == 0$$

Of course for d to be a divisor of n, d must be at most n.

Exercise -1.0.16. Write a program that prompts the user for n and prints all the (positive) divisors of a given positive integer n. For instance, if the user entered 10 for n, the program should print

```
1
2
5
10
```

If the user entered 11, the program should print

```
1
11
```

Don't forget that when the user entered 1 for n, your program should print only one 1:

```
1
```

Once you're done with your program, modify it so that if the user entered an integer less than or equal to 0, it prints

```
Wrong input. Run again and enter a positive int.
```

A **prime** is a positive integer which is greater than 1 and is divisible by only 1 and itself.

Let's try to get C++ to list primes.

Look at the statement: "A prime is a positive integer which is greater than 1 and is divisible by only 1 and itself."

Let's say the value of the integer is in variable n. There are two conditions to check on n

- n greater than 1
- n is divisible by only 1 and itself

The first condition is easy. The second one is tricky. But its not too bad.

Suppose n has value 7. We just need to check that 2, 3, 4, 5, and 6 cannot divide 7 to conclude that 7 is a prime.

Hmmm .... "2, 3, 4, 5, and 6"... sure sounds like a for-loop.

What about "cannot divide"??? Well "i divides n"is the same as saying n % i is zero. So for "i does not divide n", the remainder when you divide n by i must be nonzero, i.e. n % i is not zero. (Duh.)

Try this:

```
int n = 7;
for (int i = 2; i < n; ++i)
{
    std::cout << i << ' '<< n % i << std::endl;
}
```

Now try this:

```
int n = 25;
for (int i = 2; i < n; i++)
{
    std::cout << i << ' '<< n % i << std::endl;
}
```

AHA! If you do get a nonzero value, it's not a prime and you can stop the check right?

```
int n = 25;
for (int i = 2; i < n; ++i)
{
    std::cout << n % i << std::endl;
    if (n % i == 0) break;
}
```

But . . . of course you still need to know if your `n` is a prime!!! You can't tell from the above code. Why? Because it's possible to get out of the loop in **two** different ways:

- you could have gotten out of the loop because `i` reached `n` (therefore `n` is a prime); or
- `break` was executed (therefore `n` is not a prime)

We need to **disambiguate** the exit condition of the for-loop!!!

To do that, we create a variable that tells us why we are out of the for-loop. Let's call it `flag`. I'll let our program get the value of `n` from us so that it's easier to test the program:

```
int n;
std::cin >> n;
int flag;
for (int i = 2; i < n; ++i)
{
    std::cout << n % i << std::endl;
    if (n % i == 0)
    {
        flag = 1; // flag equals 1 means n not prime
        break;
    }
}
```

But what if `n` is indeed a prime? It would pass all the tests (i.e.the body of the if statement will not execute). In that case `flag` will not be assigned a value!!! We had better give `flag` an initial value:

```
int n = 25;
int flag = 0; // flag equals 0 means n is prime
for (int i = 2; i < n; ++i)
{
    std::cout << n % i << std::endl;
    if (n % i == 0)
    {
        flag = 1; // flag equals 1 means n not prime
        break;
    }
}
// At this point ...
// if flag is 0, then n is prime
// if flag is 1, then n is not prime \\
```

Let's insert a print statement after the loop:

```
int n = 25;
int flag = 0; // flag equals 0 means n is prime
for (int i = 2; i < n; ++i)
{
    std::cout << n % i << std::endl;
    if (n % i == 0)
    {
        flag = 1; // flag equals 1 means n not prime
        break;
    }
}

std::cout << (flag == 0 ? "prime" : "not prime")
          << std::endl;
```

(Review the ternary operator if you have forgotten about it.)

Go ahead and test your program with a couple of values for n.

Note that `flag` only needs to have two possible values because the only purpose of `flag` is to tell us why we exit the loop and there are only two reasons for exiting the for-loop. Although the program works, the variable **name** `flag` can be better. The **purpose** of `flag` is to tell if `n` is a `prime`. So ...

I will use a **`bool`** variable **`isprime`** instead. Why boolean? Because this variable only needs to take on **two possible values**. Of course if there are three exit points in the for-loop, then you should not use a boolean variable.

```
int n = 25;
bool isprime = true;
for (int i = 2; i < n; ++i)
{
    std::cout << n % i << std::endl;
    if (n % i == 0)
    {
        isprime = false;
        break;
    }
}
std::cout << (isprime ? "prime" : "not prime") << std::endl;
```

Let us now remove the print statement in the for-loop:

```
int n = 25;
bool isprime = true;
for (int i = 2; i < n; ++i)
{
    if (n % i == 0)
    {
        isprime = false;
        break;
    }
}

std::cout << (isprime ? "prime" : "not prime")
          << std::endl;
```

Study the above program very carefully. Make sure you see that the choice of the name of the variable `isprime` makes the program easier to understand.

Note that it's possible to remove the `break` statement; study this program very carefully:

```
bool isprime = true;
for (int i = 2; i < n && isprime; i++)
{
    if (n % i == 0)
    {
        isprime = false;
    }
}

std::cout << (isprime ? "prime" : "not prime")
          << std::endl;
```

Note that the loop keeps running as long as

```
i < n && isprime
```

is `true`. The only way to get out of the for-loop is when the boolean expression

```
i < n && isprime
```

evaluates to false which is when `i` reaches `n` or when `isprime` is `false`.

Exercise -1.0.17. Print all the primes from 2 to 100. (Recall that 1 is not a prime.) The pseudocode is

```
for n running from 2 to 100:
    if n is prime, print n
```

(Don't forget that a C++ int variable can take values up to about 2 billion.) And of course you know how to check "n is prime" using the code I've shown you above. Rework your program so that it prompts the users for int values for int variables a and b and prints the primes from a to b. Run your program, listing the primes in the range from 1,000,000,000 to 2,000,000,000. Observe the speed/performance of your program.

Note that it's obvious that, for instance when you're testing if n = 87 is a prime, you need **not** test if 86 is a divisor of n = 87. 86 is obviously too big to divide 87!!!

It can be proven that instead of testing i from 2 to n - 1, you really only need to test up to the **square root of n**. (I won't prove this. You can take, for instance, a class on cryptography to find out why.)

```
bool isprime = true;
for (int i = 2; i <= sqrt(n) && isprime; ++i)
{
    if (n % i == 0)
    {
        isprime = false;
    }
}

std::cout << (isprime ? "prime" : "not prime")
          << std::endl;
```

(Don't forget: if you want to use the `sqrt` function, you must add `#include < cmath>` at the top of your code. Also don't forget that if you're using MS VS, you might need to convert the value of `n` to a `double`, i.e. you should use `sqrt(double(n))` instead of `sqrt(n)`.)

Now, for instance when you test if 1023457 is prime, your program will run `i` from 2 to 1011 instead of from 2 to 1023456. That's quite a lot of CPU time saved!!!

Exercise -1.0.18. Redo your earlier primes printing program using this improved algorithm. Run your program, listing the primes in the range from 1,000,000,000 to 2,000,000,000. Observe the improved speed/performance of your program.

But there's still something else you can do. Note that `sqrt(n)` is computed many times, once every time the boolean expression in the for-loop is evaluated. We can compute and store the square root of n to avoid recomputation:

```
bool isprime = true;
int sqrtn = sqrt(n);
for (int i = 2; i <= sqrtn && isprime; ++i)
{
    if (n % i == 0)
    {
        isprime = false;
    }
}

std::cout << (isprime ? "prime" : "not prime")
          << std::endl;
```

Exercise -1.0.19. Redo your earlier primes printing program using this improved algorithm. Run your program, listing the primes in the range from 1,000,000,000 to 2,000,000,000. Observe the improved speed/performance of your program.

There's yet another thing we can do. Note that `sqrtn` is used only in the for-loop. We can actually declare `sqrtn` at the initialization part of the for-loop so that the scope of `sqrtn` stays inside the loop:

```
bool isprime = true;
for (int i = 2, sqrtn = sqrt(n); i <= sqrtn && isprime; ++i)
{
    if (n % i == 0)
    {
        isprime = false;
    }
}

std::cout << (isprime ? "prime" : "not prime")
          << std::endl;
```

Note that this does not improve the speed of your program. It simply shrinks the scope of variable `sqrtn`.

Exercise -1.0.20. Write a program that prompts the user for n and prints p and p + 2 where p and p + 2 are both primes and at most n.

For instance, if the user entered 20 your program should print

```
1. 3 5
2. 5 7
3. 11 13
4. 17 19
```

For instance, note that 5 and 7 are both primes and 7 is 5 + 2. Note however that 7 and 9 are not printed since 9 is not a prime. These pairs of primes are called "twin primes". It has been conjectured for more than 150 years that there are infinitely many twin primes. This is one of the most famous open questions in Math. The largest known twin primes (as of 2010) have more than 100000 decimal digits. If you find something bigger than that let me know. Don't forget that with what you know now, you can't handle integers significantly larger than 2 billion, i.e. your int variables can't handle more than 10 digits!!!

Exercise -1.0.21. Write a program that prompts the user for n and prints the longest chain of consecutive composite integers at most n. A composite integer is an integer greater than 1 that is not a prime. For instance, if the user entered 10 for n, then the following are the numbers from 2 to n with the composites underlined:

$$2,\ 3,\ \underline{4,}\ 5,\ \underline{6,}\ 7,\ \underline{8,\ 9,\ 10}$$

The longest chain of composites is 8, 9, 10. Therefore your program should print

```
8 9 10
```

Brute force search

Computer Science, like Mathematics, seeks to find solutions to problems. One way to look for solutions is through “brute-force”search. This means going over all possible candidates and seeing which ones are the right solutions.

Let’s try one. Suppose you want to find **integer** solution(s) to this equation:

$$x^3 = 729$$

If x is a negative integer, x^3 is negative. So we are definitely only interested in positive integers. Furthermore x must be less than 729 (well ... a lot less!). So we can try x = 0, 1, 2, ..., 729 and see which one satisfies the above equation.

WHOA!!!

That sure looks suspiciously like a for-loop to me ...

Exercise -1.0.22. Write a for-loop with an index variable running from 0 to 729 to find (and print ... of course!) solution(s) to the equation

$$x^3 = 729$$

Well . . . the above example is kind of silly since we could have computed the cube root of 729 and see if it’s an integer value . . . but what if you have ***two*** variables in the equation? Suppose you want to solve for positive integers x and y satisfying

$$x^2 + y^2 = 13^2$$

Well first of all each x and y must be at most 13, right? So all you need to do is to check all possible 0, ..., 13 for x and 0, ..., 13 for y. In other words you need to check all the following cases:

x = 0, y = 0
x = 0, y = 1
x = 0, y = 2
...
x = 0, y = 13
x = 1, y = 0

x = 1, y = 1
x = 1, y = 2
...
x = 1, y = 13
x = 2, y = 0
x = 2, y = 1
x = 2, y = 2
...
x = 2, y = 13
...
...
...
x = 13, y = 13

That sure looks like a double-nested for-loop to me! Here's the program:

```
for (int x = 0; x <= 13; ++x)
{
    for (int y = 0; y <= 13; ++y)
    {
        if (x * x + y * y == 13 * 13)
        {
           std::cout << x << ',' << y << '\n';
        }
    }
}
```

Now note that you see some "repetitions". I'm saying that if x = a, y = b is a solution, then x = b, y = a is also a solution because in the equation above, you can switch x and y. What if you do not want to include such repetitions? One way would be to ensure x ¡= y. In other words you try to get your program to run through these values of x,y:

x = 0, y = 0
x = 0, y = 1
x = 0, y = 2
...
x = 0, y = 13
x = 1, y = 1 (note that y starts with 1 and not 0)
x = 1, y = 2
x = 1, y = 3
...
x = 1, y = 13

x = 2, y = 2 (note that y starts with 2 and not 0)
x = 2, y = 3
x = 2, y = 4
...
x = 2, y = 13
...
...
...
x = 13, y = 13

```
for (int x = 0; x <= 13; ++x)
{
    for (int y = x; y <= 13; ++y)
    {
        if (x * x + y * y == 13 * 13)
        {
           std::cout << x << ',' << y << '\n';
        }
    }
}
```

Of course you realize that $x^2 + y^2 = 13^2$ gives the solution of the right-angle triangle with hypotenuse of length 13 and integer sides. Why not find all such triangles with hypotenuse from 1 to 100??? This means solving

$$x^2 + y^2 = z^2$$

where 1 ¡= z ¡= 100.

```
for (int z = 1; z <= 100; ++z)
{
    for (int x = 0; x <= z; ++x)
    {
        for (int y = x; y <= z; ++y)
        {
            if (x * x + y * y == z * z)
            {
               std::cout << x << ','
                         << y << ','
                         << z << '\n';
            }
        }
    }
}
```

This prints the sides of all right angle triangles with integer sides where the hypotenuse is between 1 to 100.

Exercise -1.0.23. Can you improve the performance of the above program?

Exercise -1.0.24. Find all integer solutions to the equation

$$x^4 + 2y^4 = 10001$$

Exercise -1.0.25. Here's something from wikipedia.org:

```
1729 is known as the Hardy-Ramanujan number, after a
famous anecdote of the British mathematician
G. H. Hardy regarding a
hospital visit to the Indian mathematician
SrinivasaRamanujan. In Hardy's words:

`` I remember once going to see him when he was ill at
Putney. I had ridden in taxi cab number 1729
and remarked that the number seemed to me rather a dull
one, and that I hoped it was not an unfavorable omen.
"No," he replied, "it is a very interesting number;
it is the smallest number expressible as the sum of two
cubes in two different ways."

The quotation is sometimes expressed using the
term "positive cubes", as the admission of negative
perfect cubes (the cube of a negative integer) gives
the smallest solution as 91 (which is a factor of 1729):

     91 = 6^3 + (-5)^3 =
     4^3 + 3^3

Of course, equating "smallest" with "most negative", as
opposed to "closest to zero" gives rise to solutions
like -91, -189, -1729, and further negative numbers.
This ambiguity is eliminated by the term "positive cubes".

Numbers such as

     1729 = 1^3 + 12^3 =
     9^3 + 10^3

that are the smallest number that can be expressed as the
sum of two cubes in n distinct ways have been dubbed
taxicab numbers. 1729 is the second taxicab number (the
first is 2 = 1^3 + 1^3). The number was also found in one
of Ramanujan's notebooks dated years before the incident.
```

Find all positive integer solutions to the equation

$$x^3 + y^3 = 1729$$

(Hint: Write a double for-loop.)

Exercise -1.0.26. The above exercise verified that there are exactly two integer solutions. It does not verify that 1729 is the "**smallest number expressible as the sum of two cubes in two different ways**". To do that you need to solve this:

$$x^3 + y^3 = z$$

Of course you should print out x, y, and z. Look at all the solutions and find the value of z with exactly two integer solutions. Is it 1729? (Or is Ramanujan totally wrong all these years ...)

Exercise -1.0.27. Write a program that prints all positive fractions m/n from 0 to 5 where m and n are positive and at most 10. The fractions need not be reduced or in any particular order or unique.

Brute force search: polynomial factorization

Here's another brute force search program. Let's write a program to factorize a degree 2 polynomial into two polynomials with integer coefficients. For instance

$$x^2 - 1 = (x + 1)(x - 1)$$

Therefore when the user enters 1 0 -1 for polynomial $x^2 - 1$, the program produces (x + 1)(x - 1). Here's an execution of the program:

```
1 0 -1
1x^2 + 0x + -1 = (1x + 1)(1x + -1)
```

Assume that your program will only handle coefficients from -20 to 20.

We know that

$$(ax + b)(cx + d) = acx^2 + (ad + bc)x + bd$$

Suppose the user entered A, B, C for polynomial

$$Ax^2 + Bx + C$$

then we basically want to find a, b, c, d such that

$$(ax + b)(cx + d) = acx^2 + (ad + bc)x + bd = Ax^2 + Bx + C$$

which is the same as saying:

$$\begin{aligned} A &= ac \\ B &= ad + bc \\ C &= bd \end{aligned}$$

For instance in the case of the polynomial $x^2 - 1$, we want to find a, b, c, d such that

$$\begin{aligned} 1 &= a * c \\ 0 &= a * d + b * c \\ -1 &= b * d \end{aligned}$$

Here's the program:

```
int A, B, C;
std::cin >> A >> B >> C;

int a, b, c, d;

bool found = false;

for (a = -20; a <= 20 && !found; ++a)
{
    for (b = -20; b <= 20 && !found; ++b)
    {
        for (c = -20; c <= 20 && !found; ++c)
        {
        for (d = -20; d <= 20 && !found; ++d)
        {
            if (a * c == A
                 && B == a * d + b * c
                 && b * d == C)
            {
                    found = true;
            }
        }
     }
}
if (found)
{
   std::cout << a << ' '<< b
             << ' ' << c << ' '<< d << '\n';
}
}
```

Of course you can change the range of values for a, b, c, d if you like. In fact, it's a good idea to create a constant, say N, for 20 in the above program so that a, b, c, d ranges from -N to N and you can change the value of N easily.

Note that this is a **brute force** search for a factorization of the given polynomial because it's **not smart**. For instance, if the given polynomial is

$$15x^2 + \ldots$$

then since you want

$$15x^2 + \ldots = (ax + b)(cx + d) = acx^2 + (ad + bc)x + bd$$

then of course

$$15 = ac$$

and the right thing to do is to factorize 15. You would get 15 = 1x15 = 3x5 = 5x3 = 15x1 = -1x-15 = -3x-5 = -5x-3 and then a and c can only be

$$\begin{aligned} a = 1, c = 15 \\ a = 3, c = 5 \\ a = 5, c = 3 \\ a = 15, c = 1 \\ a = -1, c = -15 \\ a = -3, c = -5 \\ a = -5, c = -3 \\ a = -15, c = -1 \end{aligned}$$

The above program however tries 41 possible values for a (i.e., from -20 to 20) and also 41 possible values for c. This means that the program actually tries 41 x 41 possible cases for a and c. Including b and d, the program tries

$$41 \times 41 \times 41 \times 41 = 2825761$$

cases!!! This is no big deal for a modern-day computer of course. But if we want to allow more values for a, b, c, d, say from -100 to 100, then there are about

$$201 \times 201 \times 201 \times 201 = 1632240801$$

You can try this range of values for the above program. You'll see that the program will run very slowly. Brute force algorithms are not smart at all.

Exercise -1.0.28. Write a program that prompts the user for n and if n is even, attempts to rewrite n as a sum of at most two primes. For instance if the user entered 2, the program prints:

```
2
2
```

If the user entered 4 the program prints

```
4
2 + 2
```

If the user entered 12, the program prints

```
12
5 + 7
```

Using your program, attempt to answer this question: Are there even integers which are not a sum of at most two primes? (For more information, google for “Goldbach conjecture”. This is also a very famous conjecture in Math that is as yet unproven.)

Summary

The following summarizes the pre- and post-increment operators:

`++i`increments the value of`i`
`i++`increments the value of`i`
`j = (++i)` increments the value of i and then assign new value of `i` to `j`
`j = (i++)` give the value of `i` to `j` and then increments the value of `I`

The following summarizes the pre- and post-decrement operators

`--i`decrements the value of `i`
`i--`decrements the value of `i`
`j = (--i)` decrements the value of i and then assign new value of `i` to `j`
`j = (i--)` give the value of `i` to `j` and then decrements the value of `i`

The following summarizes the augmented assignment operators: suppose `[op]` is an operator such as +, -, *, /, %, then

`x [op]= y` is the same as `x = x [op] y`

For instance x += y is the same as x = x + y. The augmented assignment operator returns the new augmented value. For instance:

```
z = (x += y)
```

The for-loop statement looks like this:

for (`[stmt1]; [bool expr]; [stmt2]`)`[stmt3]`

where `[stmt3]` can be either a statement of a block of statements. The for-loop statement executes as follows:

1. Execute `[stmt1]`, go to 2.
2. If the `[bool expr]` is true, go to 3, otherwise goto 5
3. Execute `[stmt3]`, go to 4.
4. Execute `[stmt2]`, go to 2.
5. Exit the `for`-loop statement to the statement after the for-loop.

Of course since `[stmt3]` can be any statement or a block, a for-loop can contain an if statement, if-else statement, a switch-case statement, or even a for-loop.

The scope of a variable is from its point of declaration to the end of the smallest block where it was declared.

Exercises

Q1. Find all positive integer solutions to the equation

$$x^3 + y^4 = 1729$$

(Hint: Write a double for-loop.)

Q2. Write a program that draws the following figure when the user enters 3

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

and when the user enters 4 it draws

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

Q3. Write a program that draws the following figure when the user enters 3

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

and this when the user enters 5

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

and this when the user enters 6

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

Q4. Write a program that draws the following figure when the user enters 3

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

and when the user enters 5 it draws

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

and when the user enters 7 the program draws this:

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

and when the user enters 9 the program draws this:

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

Etc.

Q5. Write a program that draws the following figure when the user enters 3

```
 +
/ \
| |
---
```

and when the user enters 5 it draws

```
  +
 |
 / \
/   \
|   |
|   |
-----
```

and when the user enters 7 the program draws this:

```
   +
   |
   |
  / \
 /   \
/     \
|     |
|     |
|     |
-------
```

Etc.

Q6. Write a program that draws the following figure when the user enters 7 it draws

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

and when the user enters 9 it draws

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

and when the user enters 11 it draws

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

Q7. Write a program that draws the following given 2 and 2:

```
**********
*        *
* X      *
*        *
*        *
*        *
*        *
*        *
*        *
**********
```

and this when the user enters 7 and 3:

```
**********
*        *
*        *
*        *
*        *
*        *
*        *
*  X     *
*        *
**********
```

Q8. Write a program that prompts the user for n and computes

1 + 1/(1) + 1/(1x2) + 1/(1x2x3) + 1/(1x2x3x4) + . . . + 1/(1x2x3x...xn)

Q9. Write a program that prompts the user for a double d and computes the largest n such that n^2 + n is at most d.

Q10. Write a program that prompts the user for an integer n and displays all the non-squares from 1 to n and print the number of these integers. (An integer is a non-square if it is not the square of another integer. For instance 9 is a square since it is the square of 3, however 6 is not a square.)

Q11. A number n is said to be perfect if it is the sum of all its divisors less then n. For instance 6 is perfect since the divisors of 6 are 1, 2, 3, 6 and the sum of the divisors less than 6 is 1 + 2 + 3 = 6. Write a program that prompts the user for n and prints all perfect numbers from 1 to n.