14. Loops

Objectives

Don't worry … I won't be introducing more loop syntax.

In this section we'll look at more examples. I'll also give some general comments, suggestions, techniques, warnings etc.

We'll start off with two reviews … general advice on when you should use a `for`-loop and when you should use `while`-loop ...

When do I use the `for`-Loop?

I've already shown you that you can always rewrite a `for`-loop into a `while`-loop and vice versa.

So ... how do you choose which type of loop to use?

The general advice is simple: If your loop has a control (index or counter) variable that runs across a sequence of values without user intervention and the control variable changes in a predictable way, you should use the `for`-loop. The user might need to "configure" this sequence: for instance, by specifying a starting value and an ending value and how to step through the intermediate values. But once that's done, the control variable in the `for`-loop should run without user intervention.

Otherwise you should use the `while`-loop.

Look at this familiar example:

```
int start = 0, end = 0, step = 0;
std::cin >> start >> end >> step;

int sum = 0;
for (int i = start; i <= end; i += step)
{
    sum += i;
}
std::cout << sum << std::endl;
```

Note that the loop is controlled by `i` and `i` steps through a sequence with the starting and ending values known before the loop starts.

Note that the control variable is only used in loop through some statements to achieve a goal. Look at the above example: The control variable is defined within the `for`-loop. After the `for`-loop, the control variable is destroyed.

The next section shows you a situation where the data to process is entered by the user and hence cannot be predicted ahead of time.

When do I use the `while`-Loop?

Suppose you want to write a program that adds numbers for a user but you do not know in advance how many numbers to add. Of course you need to give the user an opportunity to stop the adding process! There are two ways of doing this. Either you can ask the user each time if he/she wants to enter a number to add or to stop, or ... you can use a **sentinel value**. This is a special value entered by the user to indicate that the processing loop should stop.

Let's try the simplest possible example. This program prompts the user until he/she enters a sentinel value. I'm choosing -1 to be the sentinel value.

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

std::cout << "gimme a number: ";
std::cin >> i;
while (i != -1)
{
    sum += i;
    std::cout << "gimme a number: ";
    std::cin >> i;
}
std::cout << "sum: " << sum << std::endl;
```

prompt user for data
while value of data is not sentinel value:
    process data
    prompt user for data

Note that there is code duplication:

prompt user for data
while value of data is not sentinel value:
    process data
    prompt user for data

Removing Duplication in the sentinel `while`-loop

Look at the sentinel while-loop again:

prompt user for data
while value of data is not sentinel value:
    process data
    prompt user for data

Remember that there is code duplication. If you want to get rid of code duplication, you can do this:

while (1): // don't forget that 1 is type cast to true
    prompt user for data
    if value is the sentinel value:
        break
    process data

Note that some people do not like `break`s because it's harder to tell when you get out of a loop because you can issue a `break` statement anywhere in the body of the `while` loop. They prefer the exit condition in the header of the `while` loop so that visually one can tell quickly when you get out of the loop.

Using the above technique, the following program:

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

std::cout << "gimme a number: "; // INPUT
std::cin >> i;
while (i != -1) // CONDITION TO REPEAT LOOP
{
    sum += i;

    std::cout << "gimme a number: "; // INPUT
    std::cin >> i;
}
std::cout << "sum: " << sum << std::endl;
```

becomes:

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

while (1)
{
    std::cout << "gimme a number: "; // INPUT
    std::cin >> i;
    if (i == -1) break; // CONDITION TO EXIT LOOP

    sum += i;
}
std::cout << "sum: " << sum << std::endl;
```

Make sure you study this program carefully!!!

Exercise. Rewrite the above program so that in addition to exiting the loop when the term entered in 0 (i.e., use a sentinel value of 0), the program also exits the loop when the sum exceeds 10.

Exercise. Using this structure of the `while`-loop for data processing, rewrite your previous program that computes the product of integers entered by the user. Use a sentinel value of 0.

Example: sentinel value

Here's an easy example. Write a simple cash register program. The program prompts the cashier for the number of items and the unit price per line item (yes, it's a lousy point-of-sale program). There is a tax of 20% (this is probably in New York).

```
const double TAX = 0.2;
double total = 0.0;
double price = 0.0;
int num = 0;

while (1)
{
    std::cout << "units: ";
    std::cin >> num;

    std::cout << "unit price: ";
    std::cin >> price;

    double cost = num * price * (1 + TAX);
    std::cout << "please pay " << cost << std::endl;
}
```

The problem is that the program doesn't stop!!!

Exercise. Add code to this program so that when the cashier enters 0 for number of items, the program gets out of the loop.

Exercise. Add code to print the warning “units cannot be negative” and `continue` when `num` is negative.

Exercise. Add code to print the warning “unit price cannot be negative” and `continue` when `price` is negative.

Exercise. Add a running total so that the cashier can double check the amount in the till at the end of the day.

Of course for a real business application, the POS register would be networked to a database server containing the product catalog and you simply scan the UPC to retrieve the unit price. The data computed is stored in a database directly. And of course you can have more than one line item per sale.

Note that with multiple `continue`s, it makes the program harder to read. This is similar to the situation where there are multiple `break`s in the body of the while-loop. Can you rewrite the above program so that there are no `continue`s?

Other conditions

So far we have seen two different ways for a loop to terminate:

- when a control variable has finished running over it's sequence (`for`-loop)
- when a data input is a sentinel value (`while`-loop)

Of course loops can terminate based on other types of conditions. It's impossible to list all possible cases!!! You have to analyze each case carefully to see what's the right thing to do.

Example: counting digits

Here's a problem where the condition does not depend on continually prompting the user for an input. Therefore there is **no sentinel value**. You will see that there is also no control variable – so you definitely should not expect a `for`-loop.

Pay attention to the way I slowly work through the problem until I see the loop.

Suppose the user enters an integer and you want to count the number of digits in that integer.

For instance if the user enters 134, your program must display 3. If the user enters 97531, your program must display 5.

So how do you do that? If you count the digits from the right to left, you can think of moving a finger one digit at a time and counting each digit your finger passed by:

```
97531^
9753^1      1 digit
975^31      2 digits
97^531      3 digits
9^7531      4 digits
^97531      5 digits
```

(^ = your finger).

Now you notice that the digits you've passed does not affect the rest of your digit-counting process. So let's throw them away:

```
97531^
9753^       1 digit
975^        2 digits
97^         3 digits
9^          4 digits
^           5 digits
```

Now, the symbol for our finger doesn't help. Let's get rid of that too:

```
97531
9753        1 digit
975         2 digits
97          3 digits
9           4 digits
            5 digits
```

Well ... that sure looks like removing the rightmost digit (one at a time) to me. And we already know how to do that: just do integer division by 10!!! If we do that, then the last number should be zero:

97531	
9753	1 digit
975	2 digits
97	3 digits
9	4 digits
0	5 digits

Of course the number that is going up (the 1, 2, 3, 4, 5) requires a variable. Let's call it x. And let's say the number we're working on (the number entered by the user) is placed in variable n.

Now when does the process **stop**? When we see a zero in the left column of the above simulation, right? So the process seems to be this:

```
prompt user for integer n
if n is not zero, n = n / 10 and x = 1
if n is not zero, n = n / 10 and x = 2
if n is not zero, n = n / 10 and x = 3
if n is not zero, n = n / 10 and x = 4
if n is not zero, n = n / 10 and x = 5
n is zero. Stop.
```

We need to write this as an algorithm. Obviously there's a loop here!!! You want the statements

```
if n is not zero, n = n / 10 and x = 1
if n is not zero, n = n / 10 and x = 2
if n is not zero, n = n / 10 and x = 3
if n is not zero, n = n / 10 and x = 4
if n is not zero, n = n / 10 and x = 5
```

to be replaced by one statement. You can “combine” them like this:

```
if n is not zero, n = n / 10 and x = x + 1
if n is not zero, n = n / 10 and x = x + 1
if n is not zero, n = n / 10 and x = x + 1
if n is not zero, n = n / 10 and x = x + 1
if n is not zero, n = n / 10 and x = x + 1
```

AHA! They are all the same! Of course x does not have an initial value. You need to initialize it. And based on the above example, you can see that x should be initialized to 0. Now you can write the following very easily:

```
while n is not 0:
        n = n / 10
        x = x + 1
```

Here's the code:

```
int n = 97531;
int numDigits = 0;

while (n != 0)
```

```
{
    n /= 10;
    ++numDigits;
    std::cout << n << ' ' << numDigits << std::endl;
}
std::cout << numDigits << std::endl;
```

I've inserted print statements to get the program to show us the intermediate steps. And instead of `x`, I'm using `numDigits` which is a lot more descriptive.

Yes, of course you can rewrite the `while`-loop as a `for`-loop:

```
int numDigits;
for (numDigits = 0; n != 0; ++numDigits)
{
    n /= 10;
    std::cout << n << ' ' << numDigits << std::endl;
}
```

Although it's not wrong, note that the `for`-loop does not really use a control variable: the variable updated (i.e. `numDigits`) is the variable in the boolean expression (i.e. `n`). This is an indication that the `while`-loop is probably best.

Remember: If the loop doesn't look like a `for`-loop (i.e., there's no single variable that runs across a sequence of values), then don't use it. Use the `while`-loop instead.

Exercise. There's just one problem. Run it with n = 0. What do you get? How many digit(s) do you see in the number 0? Of course the number 0 has 1 digit. This is a special case. So the pseudocode looks like this:

```
if n is 0:
   numDigits = _____
else
   use the above digit removing algorithm to compute numDigits
```

Correct the program. Test your program thoroughly.

Error-checking for a sentinel `while`-loop

The following is a structure for the `while`-loop that includes checks for breaking and data validation:

```
some initialization
while (1):
        get data
        if data is sentinel value:
                break
        if data is not valid:
                print warning
                continue

        process data
```

OR

```
some initialization
while (1):
        get data
        if data is sentinel value:
                break
        if data is not valid:
                print warning
        else:
                process data
```

The second version removes the presence of `continue`. Make sure you understand that they work the same!!! You can use either version in your programs.

Exercise. Write a program that prompts the user for positive integers and computes the sum of the square root of the integers entered by the user. Of course, you can't compute the square root of a negative number, right? The program terminates when the user enters 0. The program prints a warning when a negative value is entered.

Although this is usually not mentioned in most textbooks, the above is a very standard structure of the loop used in the industry, if you need to use it at all. In general, alternative flow of execution must be bunched up together in a spot which is easy to find. We won't have time to talk too much about this topic, but in writing robust software there are concepts such as assertions, exceptions, pre-conditions, post-conditions, etc. which are all related to catching or verification of error conditions. They are all related to flow of control. These controls are also usually placed at the “usual” places, near the top or the bottom of certain structures. I won't elaborate on “assertions, exceptions, pre-conditions, post-conditions” – you will learn some of these in future classes.

Avoiding `break` and `continue`

The basic principle in programming is simple: don't do crazy things!

In other words, don't try to surprise the reader (including yourself). Writing crazy stuff throws people off. It makes debugging really difficult – that includes yourself!!!

One corollary is that you should limit the number of exit points for your loops. This means not having too many `break`s in your loops. Likewise limit the number of `continue`s.

Exercise. Rewrite this without the break.

```
int prod = 1;
for (int i = 1; i <= 10; i++)
{
    if (i == 5) break;
    prod *= i;
}
```

Exercise. Rewrite the above without the `continue`.

```
int prod = 1;
for (int i = 1; i <= 10; i++)
{
    if (i == 5) continue;
    prod *= i;
}
```

“Pushing” the Control/Index/Counter Variable in the `for`-Loop (DIY)

Here's the program from the previous section:

```
int prod = 1;
for (int i = 1; i <= 10; i++)
{
    if (i == 5) break;
    prod *= i;
}
```

One method of terminating a for-loop early is what I call “pushing the control variable”. It looks like this:

```
int prod = 1;
for (int i = 1; i <= 10; i++)
{
    if (i == 5)
    {
        i == 11; // force i to have value to stop
                 // the for-loop
        continue;
    }
    prod *= i;
}
```

Unfortunately this method requires knowing the bound of the values assumed by `i` and hence can be dangerous. Therefore ... DON'T DO IT!

In fact, the general advice is this: **A much as possible, do not modify the control/index/counter variable in the body of the `for`-loop.**

WARNING: `break` and `continue` are shallow!

It's very important to remember that the `break` statement only breaks out of the loop the program is **currently** in!!!

```
for (int i = 0; i < 5; i++)
{
    std::cout << "i: " << i << std::endl;
    for (int j = 0; j < 5; i++)
    {
        if (j == 3) break; // WHERE DO YOU GO?
        std::cout << "  j: " << j << std::endl;
    }
}
```

Likewise `continue` only applies to the current loop:

```
for (int i = 0; i < 5; i++)
{
    std::cout << i << std::endl;
    for (int j = 0; j < 5; i++)
    {
        if (j == 3) continue; // WHERE DO YOU GO?
        std::cout << j << std::endl;
    }
}
```

This is the same for nested `while`-loops or a `for`-loop containing a `while`-loop.

Breaking out of Deeply Nested Loops

Now what if the `break` in the above example is really meant to break out of BOTH `for`-loops?

One way to accomplish this would be to use a special boolean flag to indicate that a complete break is needed. Here's an example. Suppose you have this code:

```
for (int i = 0; i < 10; i++)
{
    ... SOME STATEMENTS ...

    for (int j = 0; j < i; j++)
    {
        if (j == 3)
        {
           // WANT TO BREAK OUT OF *BOTH* LOOPS
        }
    }

    ... SOME STATEMENTS ...
}
```

You modify it in this manner:

```
for (int i = 0; i < 10 && !stop; i++)
{
    ... SOME STATEMENTS ...

    bool stop = false;
    for (int j = 0; j < i; j++)
    {
        if ( j == 3 )
        {
           stop = true;
           break; // break of of inner loop
        }
    }
    if (stop) break; // break out of outer loop

    ... SOME STATEMENTS ...
}
```

So in general (i.e. for both the `for`-loop and the `while`-loop) you have something like this:

```
begin of loop1

    bool stop = false
    begin of loop2
        if ([bool expr]):
            stop = true
            break
    end of loop2
```

```
end of loop1
```

As you can see things can go **pretty ugly**. The best is to try to simplify the actual `for`-loop or even the whole program whenever possible rather than trying to insert controls to break out in the middle of the `for`-loop (or `while`-loop).

Exercise. Here's a simple drill. You own a certain number of banks (at most 100) and each bank has a certain number of accounts (at most 1000). This program helps you calculate how much you have altogether in the banks. When entering the amount in each account at a bank, if you enter 0, you will immediately go to the next bank and if you enter -1, the program will print the total and stop. Complete the program!

```
double total = 0;
for (int bank = 0; bank < 100; ++bank)
{
    std::cout << "bank #" << i << '\n';

    for (int account = 0; account <= 1000; ++account)
    {
        std::cout << "account #" << account << "\n";
        std::cout << "amount (0:next bank, -1:exit)";
        double amount = 0;
        std::cin >> amount;
        if (amount == 0)
        {
             // exit this for-loop and go to
             // next bank


        }
        if (amount == -1)
        {
             // exit both for-loops and go to
             // print total


        }
        total += amount;
    }

}

// print total
std::cout << total << '\n';
```

Numerical Approximation

Look at this sum:

$$t = \frac{1}{1 * 1} + \frac{1}{2 * 2} + \frac{1}{3 * 3} + \frac{1}{4 * 4} + \frac{1}{5 * 5}$$

This is a sum of 5 terms. It's easy to write a program to compute the sum – you have seen similar examples before.

Exercise. Write a program that computes t. Should you use a `for`-loop or a `while`-loop? The value you get should be approximately 1.46361.

Exercise. Now modify the above program to compute

$$\frac{1}{1 * 1} + \frac{1}{2 * 2} + \frac{1}{3 * 3} + \frac{1}{4 * 4} + \frac{1}{5 * 5} + \frac{1}{6 * 6}$$

The value should be 1.49139. (If you get 1 then you know you're wrong.)

Exercise. The first program of this section computes a sum of 5 terms. The second computes a sum of six terms. Now do the obvious: Rewrite your program so that it prompts the user how many terms to use in the sum. Let's say the variable for the number of terms is `n`.

Exercise. Modify your program to print the following values:

$$\frac{1}{1 * 1}$$

$$\frac{1}{1 * 1} + \frac{1}{2 * 2}$$

$$\frac{1}{1 * 1} + \frac{1}{2 * 2} + \frac{1}{3 * 3}$$

$$\frac{1}{1 * 1} + \frac{1}{2 * 2} + \frac{1}{3 * 3} + \frac{1}{4 * 4}$$

$$\frac{1}{1 * 1} + \frac{1}{2 * 2} + \frac{1}{3 * 3} + \frac{1}{4 * 4} + \frac{1}{5 * 5}$$

up to

$$\frac{1}{1 * 1} + \frac{1}{2 * 2} + \frac{1}{3 * 3} + \ldots + \frac{1}{10 * 10}$$

You should get this:

```
1
```

```
1.25
1.36111
1.42361
1.46361
1.49139
1.5118
1.52742
1.53977
1.54977
```

Now let the printout go up to a sum of 20 terms, then 30 terms, etc. Keep going until the first 5 decimal places stops changing.

Basically, what you're doing in the previous exercises is to compute the approximations to

```
          1       1       1       1       1
s = ----- + ----- + ----- + ----- + ----- + ...
    1 * 1   2 * 2   3 * 3   4 * 4   5 * 5
```

Note that this goes on forever. The reason why it stops changing after a while is that the term you're adding to the sum gets smaller and smaller "very quickly":

```
  1
-----
i * i
```

If you want to know more, you should take Calc 2. This is not a Calc class.

(To the math folks: If you keep going forever you get

$$\frac{\pi^2}{6}$$

correct?)

Computations of approximations is important for many kinds of applications including scientific, mathematical, and business applications.

Now suppose you work for NASA and you need to compute the above sum

```
          1       1       1       1       1
s = ----- + ----- + ----- + ----- + ----- + ...
    1 * 1   2 * 2   3 * 3   4 * 4   5 * 5
```

(which goes on forever) up to 10 decimal places. You can do what we did early: Keep including more and more terms. You need to get C++ to reveal more decimal places (use `std::setprecision()`). That would be really time consuming.

Another way is to let the `for`-loop run without an upper limit for `i` (how would you do that?) and when you see the first 10 decimal places not changing you can write it down (if you can see it scrolling by) and stop the program manually. But this might take a while and if you go out to

lunch, the `for`-loop keeps running even after the first 10 decimal places stop changing. This is a waste of CPU time.

We need to figure out how to stop the loop AUTOMATICALLY.

By the way if you look at the term:

```
  1
-----
i * i
```

obviously you want to avoid integer division. So in C++ you should do

```
1.0 / (i * i)
```

But besides fixing the division problem, you have another: recall that an `int` value in C++ does not go on forever. If you want to represent bigger numbers, you should do

```
1.0 / (double(i) * i)
```

i.e. even the denominator multiplication should be “doubled”-up.

Exercise. Repeat the above example with this:

```
      1       1       1       1       1
s = ----- + ----- + ----- + ----- + ----- + ...
    1*1*1   2*2*2   3*3*3   4*4*4   5*5*5
```

Exercise. Repeat the above example with this:

```
      1       1       1
s = ----- + ----- + ----- + ...
    1 * 1   3 * 3   5 * 5
```

Exercise. Repeat the above example with this:

```
      1       1       1       1
s = ----- - ----- + ----- - ----- ...
    1 * 1   3 * 3   5 * 5   7 * 7
```

(WARNING: Note the alternating + and – signs.)

Exercise. Repeat the above example with this:

```
    1    1      1        1          1
s = - + --- + ----- + ------- + --------- + ...
    1   1*2   1*2*3   1*2*3*4   1*2*3*4*5
```

Numerical Approximation: Stopping the Approximation Process

To really answer the last question, you need to know a little more math. So I'm going to simplify the problem.

Instead of stopping the computation when the further computations will not change the first 10 decimal places, I'll replace the stopping condition with this: Stop the approximation process when the **first 5 decimal places of two consecutive approximations are the same**. I'll give you the code structure for the general problem.

What I mean is this. Suppose the following are values from an approximation and you want to stop when two consecutive approximations have the same first 3 decimal places:

1.1 1.123 1.1345 1.1355 **1.1356** 1.1357

You should stop at the number in bold: Look at its first 3 decimal places and the first decimal places of the previous number. Get it?

OK. Let's do it.

Note that since you need to compare **two consecutive approximations**, that tells you that the computer is thinking about two approximations at the same time: You need two variables to hold consecutive approximations. Let's use the previous values as a simulation. The values computed are:

1.1 1.123 1.1345 1.1355 **1.1356** 1.1357

Let's call the two approximation variables a0, a1. You can also think of a0 and a1 scanning over the above numbers hand-in-hand:

```
1.1   1.123  1.1345  1.1355  1.1356  1.1357
a0    a1

1.1   1.123  1.1345  1.1355  1.1356  1.1357
      a0     a1

1.1   1.123  1.1345  1.1355  1.1356  1.1357
             a0     a1

1.1   1.123  1.1345  1.1355  1.1356  1.1357
                    a0     a1
```

and we stop at this point because the first three decimal places of a0 and a1 at this point are the same.

Hmmm ... it seems like we need to initialize a0 and a1 with the first two approximations. But there's something about the loop .... Look at the first two lines.

```
1.1     1.123   1.1345  1.1355  1.1356  1.1357
a0      a1

1.1     1.123   1.1345  1.1355  1.1356  1.1357
        a0      a1
```

The pseudocode for the above two data lines looks like this:

```
compute first approximation and put into a0
compute second approximation and put in to a1

compute second approximation and put into a0
compute third approximation and put in to a1
```

But a better way is this:

```
compute first approximation and put into a0
compute second approximation and put in to a1

a0 = a1
compute third approximation and put in to a1
```

See it? By the way, note that the computation of the next (third) approximation should depend on the current (second) approximation. You don't have to do it from scratch. For instance, to compute

```
  1       1       1
----- + ----- + -----
1 * 1   2 * 2   3 * 3
```

you just need to add `1.0/(3*3)` to

```
  1       1
----- + -----
1 * 1   2 * 2
```

What about the ending condition? The right thing to do is to first compute the new approximation and then terminate the loop if a0 and a1 are close enough, say they are less than 0.0001 apart. What does that mean? That means we stop the loop if either

$$a0 - a1 < 0.0001$$

or

$$a1 - a0 < 0.0001$$

In other words we stop if mathematically

$$| a1 - a0 | < 0.0001$$

where | ... | means “the absolute value of”. In C++ the absolute value is given by the `abs()` function for integer values and `fabs()` for double. The above becomes

```
fabs(a0 – a1) < 0.0001
```

You might need to `#include <cmath>` to use the `abs()` or `fabs()` function.

Exercise. Review … try this:

```
std::cout << abs(5) << std::endl;
std::cout << abs(-5) << std::endl;
std::cout << fabs(12.34) << std::endl;
std::cout << fabs(-12.34) << std::endl;
```

The pseudocode looks something like this:

```
error = 0.00001
approx0 =  [some value]
approx1 = [another value]
while ( |approx0 – approx1| >= error):
   approx1 = approx0
   compute approx1
```

This is correct but we can simplify the code. Look at the previous simulation:

```
1.1     1.123   1.1345  1.1355  1.1356  1.1357
a0      a1

1.1     1.123   1.1345  1.1355  1.1356  1.1357
        a0      a1

1.1     1.123   1.1345  1.1355  1.1356  1.1357
                a0      a1

1.1     1.123   1.1345  1.1355  1.1356  1.1357
                        a0      a1
```

It requires initializing a0 and a1 with the first two approximations. But I can initialize a0 with a dummy value say 0:

```
0.0     1.1     1.123   1.1345  1.1355  1.1356  1.1357
a0      a1

0.0     1.1     1.123   1.1345  1.1355  1.1356  1.1357
        a0      a1

0.0     1.1     1.123   1.1345  1.1355  1.1356  1.1357
                a0      a1
```

0.0	1.1	1.123	1.1345 a0	1.1355 a1	**1.1356**	1.1357
0.0	1.1	1.123	1.1345	1.1355 a0	**1.1356** a1	1.1357

As long as a1 is far enough from a0, I'm guaranteed that the approximation will be the same as the previous code.

For the above problem we have the following code:

```
double error = 0.00001;
double approx0 = 0.0;
double approx1 = 1.0; // i.e. 1.0 / (1 * 1)
int i = 1;
while (abs(approx0 - approx1) >= error)
{
    approx0 = approx1;
    i++;
    approx1 += 1.0 / (i * i);
    std::cout << approx1 << std::endl;
}
```

Verify by hand that your program works. For instance, use a calculator to compute 1 + 1/4 + 1/9 and look for this number in your output.

Here's a curious problem. Look at the following sequence of numbers:

$$1$$

$$1+\frac{1}{1}$$

$$1+\frac{1}{1+\frac{1}{1}}$$

$$1+\frac{1}{1+\frac{1}{1+\frac{1}{1}}}$$

$$1+\frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1}}}}$$

$$1+\frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1}}}}}$$

etc. Write a program to compute an approximation of this number.

Stop the loop when consecutive approximations differ by at most 0.0001.

Hint: What's the relationship between the last and the second last expressions above? If `x` is the second last, then the last is

$$1 + \frac{1}{1 + x}$$

See it?

Here's the code:

```
double error = 0.00001;
double x = 0;
double nextx = 1;
while (fabs(x - nextx) >= error)
{
    x = nextx;
    nextx = 1 + 1/(1 + x);
}
```

For the math geeks: What number is it? Recognize it?

Exercise. Here's an ancient algorithm for computing square roots known to the Babylonians. Suppose S is given and you want to compute the square root of S.

1. Start with an arbitrary positive start value x_0 (the closer to the root, the better).
2. Let x_{n+1} be the average of x_n and S/x_n i.e.

$$x_{n+1} = (1/2)(x_n + S/x_n)$$

3. Repeat step 2, until the desired accuracy is achieved.

In other words, if you repeat the above algorithm, the sequence of values $x_0, x_1, x_2, \ldots$, will get closer and closer to the actual square root of S.

To understand how it works, try to run this algorithm with S = 1 and x_0 = 1 to compute approximations to the square root of 1. Next try it with S = 2 and x_0 =1 to compute the square root of 2.

Write a program that prompts the user for an integer value S and computes an approximation. To start off, your first approximation x_0 should be computed as follows: x_0 is the integer such that

$$x_0^2 <= S <= (x_0+1)^2$$

(Use a `for`-loop.) You should stop when two consecutive approximations agree up to 5 decimal places. [Obviously you should not use the `pow()` or `sqrt()` function for this problem!!!]

Exercise. Here's another curious one: Design an approximation to this:

$$\sqrt{1}$$
$$\sqrt{1+\sqrt{1}}$$
$$\sqrt{1+\sqrt{1+\sqrt{1}}}$$
$$\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1}}}}$$
$$\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1}}}}}$$

if you go on forever.

[Hint: If x is

$$\sqrt{1+\sqrt{1+\sqrt{1}}}$$

what is

$$\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1}}}}$$

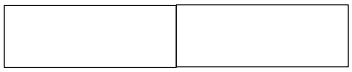
in terms of x?]

Queue

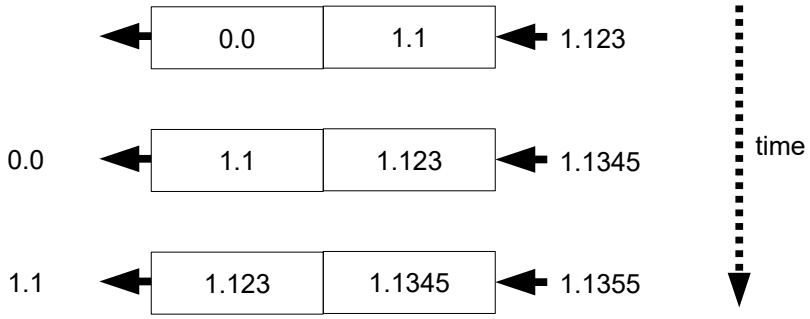
Look at our first example on approximations again. Once again we have two variables a0 and a1 and they scan over a bunch of values.

```
0.0     1.1     1.123   1.1345  1.1355  1.1356  1.1357
a0      a1

0.0     1.1     1.123   1.1345  1.1355  1.1356  1.1357
        a0      a1

0.0     1.1     1.123   1.1345  1.1355  1.1356  1.1357
                a0      a1

0.0     1.1     1.123   1.1345  1.1355  1.1356  1.1357
                        a0      a1

0.0     1.1     1.123   1.1345  1.1355  1.1356  1.1357
                                a0      a1
```

You can also think about it this way: Think of a data structure (i.e. something that contains data and comes with operations):



and data goes in at one end and comes out from the other end.



The structure you see, as a blackbox, can keep two values. There are also operations on this structure: You can take things out of the structure and you can put things into this structure. The important thing is that the thing you take out is also the thing that has been in there the longest. Such a structure is called a **first-in-first-out (FIFO)** data structure, also known as a **queue**.

Exercises

Q1. Recall that we already have a way of determining when a positive integer is prime. Write a program that prints the following

```
n          primes from  2 to n        (primes from 2 to n) / n
```

where n runs from 1 to 100. This prints the so-called density of primes. If you make a plot of n and (primes from 2 to n) / n, you will see an interesting pattern.

Q2. Modify the previous program so that you only print n when n is 100, 200, 300, etc. Let n run up to 1000000.