C++ PROGRAMMING

DR. YIHSIANG LIOW (JUNE 30, 2025)

Contents

10. for Loops: Part 1

OBJECTIVES

- Use increment, decrement and augmented assignment operators
- Write `for`-loops
- Write block of statements for the body of the for loops

We'll now begin our study of loops. I've already many times that a loop allows you to repeat the execution of a chunk of code.

I'll also talk a little more about the important concept of scopes

Why loops?

What are loops? Well, a loop is just something that will **repeat** aa statement or a bunch of statements (i.e., a block). Why do we need loops? Because we are too lazy and **way too smart** to do redundant work.

For instance, look at the following multiplication game. Instead of asking a single multiplication question (I'm using 2 spaces for indentation for the notes but in your work you should still use 4 spaces):

```
#include <iostream>
#include <ctime>
int main()
  {
    srand((unsigned int) time(0));
    int x = rand() % 10 + 90;
    int y = rand() % 10 + 90;
    int guess = 0;
    std::cout << "What is " << x << " * " << y << "?";
    std::cin >> guess;
    int product = x * y;
    if (guess < product)
    {
      std::cout << "Incorrect! Too low!"
      << "The answer is " << product
      << std::endl;
    }
    else if (guess == product)
    {
      std::cout << "Correct!" << std::endl;
    }
    else
    {
      std::cout << "Incorrect! Too high!"
      << "The answer is" << product
      << std::endl;
    }
}
```

what if you want to ask two questions? You can copy and paste your code:

```
include < iostream>
include < ctime>
int main()
{
    srand((unsigned int) time(0));
    int x = rand() % 10 + 90;
    int y = rand() % 10 + 90;
    int guess = 0;
    std::cout << "What is " << x << " * " << y << "?";
    std::cin >> guess;
    int product = x * y;
    if (guess < product)
    {
      std::cout << "Incorrect! Too low!"
      << "The answer is " << product
      << std::endl;
    }
    else if (guess == product)
    {
      std::cout << "Correct!" << std::endl;
    }
    else
    {
      std::cout << "Incorrect! Too high!"
      << "The answer is" << product
      << std::endl;
    }
    x = rand() % 10 + 90;
    y = rand() % 10 + 90;
    std::cout << "What is " << x << " * " << y << "?";
    std::cin >> guess;
    product = x * y;
    if (guess < product)
    {
      std::cout << "Incorrect! Too low!"
      << "The answer is " << product
      << std::endl;
    }
    else if (guess == product)
    {
      std::cout << "Correct!" << std::endl;
    }
    else
    {
      std::cout << "Incorrect! Too high!"
      << "The answer is" << product
      << std::endl;
    }
}
```

(Of course variables `x`, `y`, `guess`, and `product` are already de-

clared, so you don't declare them again!)

Great ...

But ... what if you want to ask **10** number of questions he/she wants to answer?

And ... what if you want to modify the program? Each time you modify one part of the program you have to modify all the other copies!!! That's not smart!!!

That's when you need to know how to write loops.

Or, take for instance, a computer game. Most games with graphics involve continually moving an image by a small amount to simulate smooth motion. In this case you want to repeat each small change until the user stops the game or when the game ends.

Similarly, in business applications, for instance in a payroll program, the program will process the pay for each employee kept in a file until all records are read. (Never mind if you don't understand what a file is. The point is that you need repetitions.)

Anyway, you will need a way to tell C++ to continually execute a statement or a block of statements until a condition is reached.

C++ understands three different types of loops: the **`for`-loop**, the **`while`-loop** and the **`do-while`-loop**.
We' ll talk about the `for`-loop first. But before we talk about the `for`-loop, we will need to talk about some operators which are frequently used with the `for`-loop.

In this set of notes, because of the way C++ (and Java) programmers write their loops, I also want to talk about "scope".

Increment and decrement operators

You have already seen the +, -, *, /, % operators for numeric values. Now for a few more.

Exercise -1.0.1. Run this:

```
{
int i = 1;

i++;
std::cout << i << std::endl;

i = 1;
++i;
std::cout << i << std::endl;

i = 1;
i--;
std::cout << i << std::endl;

i = 1;
--i;
std::cout << i << std::endl;
}
```

Change the initial value of **i** to any other number and run the program again.

The ++ and – are the increment and decrement operators respectively. When you apply ++ **in front** of the variable, you are using the **pre-increment** operator:

```
++i;
```

And when you use the ++ **after** the variable, you are using the **post-increment** operator.

```
i++;
```

Although they both add **1** to `i`, there is a difference. First of all, when you apply ++ to a variable, not only is the value of the variable changed, a copy of the value is returned. The next two exercises will make everything clear.

Exercise -1.0.2. Run this:

```
int i = 5;
int j = ++i; // i is incremented and the new value of
             // i is given to j
std::cout << i << " " << j << std::endl;
```

Exercise -1.0.3. Now run this:

```
int i = 5;
int j = i++; // The value of i is given to j and
             // *then* i is incremented
std::cout << i << " " << j << std::endl;
```

Make sure to compare the above two programs!!! In other words, the difference is in the **order** of incrementing and giving the value of the variable

The same applies to the pre- and the post-decrement operators.

Exercise -1.0.4. Run this.

```
int i = 5;
int j = --i;
std::cout << i << " " << j << std::endl;
```

Exercise -1.0.5. And this:

```
int i = 5;
int j = i--;
std::cout << i << " " << j << std::endl;
```

So now you know three different ways to add 1 to variable i:

```
i = i + 1; i++; ++i;
```

Choices are nice, right? Well, except for people who cant make up their minds. Here's a hint: the increment operators are faster. Another thing. Do not try to confuse yourself (or others) by using the increment and decrement operators in an expression. For instance the following is **BADDDDDDD** although it does run:

```
a = (++i) + (k--) * (++j);
```

This is the same as:

```
++i;
++j;
a = i + k * j;
--k;
```

which is much easier to read.

Exercise -1.0.6. First try to figure out the output without your C++ compiler:

```
int i = 0;
int j = (++i) + (++i);
std::cout << i << ' ' << j << '\n';
```

Now verify with your C++ compiler. Explain to yourself what happens when the pre-increment operator is applied multiple times to the same variable in an expression. Now do the same with this:

```
int i = 0;
int j = (i++) + (i++);
std::cout << i << ' ' << j << '\n';
```

Exercise -1.0.7. We have been using the pre-++ and post-++ on *int* variables. Can you do that to *double* variables?

Let me summarize what we now know about 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`

and here' s the summary for 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`

Gotchas

Try all these gotchas and remember them.

```
int i = 5;
(i++)++;
std::cout << i + ' ' + j << '\n';
```

```
int i = 5;
++(i++);
std::cout << i << ' ' << j << '\n';
```

```
int i = 5;
--(i++);
std::cout << i << ' ' << j << '\n';
```

```
std::cout << (++2) << '\n';
```

Augmented assignment operators

Exercise -1.0.8. Run this

```
int i = 1, j = 5;

i += 2;
std::cout << i << std::endl;

i = 1;
i += j;
std::cout << i << std::endl;
```

Change the initial values of `i` and `j` and run the program again.

Exercise -1.0.9. Run this

```
int i = 1, j = 5;

i -= 2;
std::cout << i << std::endl;

i = 1;
i -= j;
std::cout << i << std::endl;
```

Run the program again with different initial values for `i` and `j`.

Exercise -1.0.10. Run this

```
int i = 1, j = 5;

i *= 2;
std::cout << i << std::endl;

i = 1;
i *= j;
std::cout << i << std::endl;
```

In general, suppose *[op]* is an operator such as +, –, *, /, %, then

```
x [op]= y
```

is the same as

```
x = x [op] y
```

For instance,

`x %= y + 5`	is the same as	`x = x % (y + 5)`
`y /= c`	is the same as	`y = y / c`

etc. Note that the left hand version (using the augmented assignment operators) is faster than using an operator and then assignment.

Exercise -1.0.11. First figure out the output on your own:

```
int i = 10, j = 5;

i /= 2;
std::cout << i << std::endl;

i = 1;
i /= j;
std::cout << i << std::endl;
```

Now verify with your C++ compiler.

Exercise -1.0.12. Figure out the output without your compiler:

```
int i = 9, j = 5;

i %= 2;
std::cout << i << std::endl;

i = 1;
i %= j;
std::cout << i << std::endl;
```

Verify with your compiler.

Exercise -1.0.13. Figure out the output without your compiler:

```
int i = 2, j = -3, k = 5;

i %= j + k;
std::cout << i << std::endl;

i = 1;
i *= j;
std::cout << i << std::endl;
```

Verify with your compiler.

Exercise -1.0.14. One more exercise ...

```
int i = 9, j = 5;

i += i + j;
std::cout << i << std::endl;
```

By the way, you now have ***four*** different ways to add one to variable *i*:

```
++i;      i++;      i += 1;      i = i + 1;
```

(C++ programming is terrible for people who can't decide. Right?) In general, the operators on the left are faster than the ones to their right.

Exercise -1.0.15. Rewrite the following using an augmented assignment operator.

```
int heads = 0, extraHeads = 0;
std::cin >> heads;
std::cin >> extraHeads;
heads = heads + extraHeads;
std::cout << heads << std::endl;
```

Exercise -1.0.16. Rewrite the following using an augmented assignment operator.

```
double princ = 0.0, rate = 0.0;
std::cin >> princ;
std::cin >> rate;
princ = princ * (1 + rate);
std::cout << princ << std::endl;
```

Exercise -1.0.17. In fact, the augmented operators actually return a value, i.e., besides modifying the value of a variable, they evaluate to a value. Try this experiment:

```
int i = 0;
int j = (i += 1); // i is incremented *AND* (i += 1)
// gives the new value of i
std::cout << i << ' ' << j << std::endl;
```

Exercise -1.0.18. What is the output (without using your compiler of course)?

```
int i = 5;
int j = (i += 5);
int k = (j += i);
i = (k -= j + 1);
std::cout << i << ' ' << j << ' ' << k << std::endl;
```

Exercise -1.0.19. Here' s a horrifying program. First try to figure out the output on your own.

```
int i = 0, j = 1;
int k = (++i) * (i += 1) + (j *= i) * (i--);
std::cout << i << ' ' << j << ' ' << k << '\n';
```

Now verify with your C++ compiler. You should ***never*** write such an expression in actual programs. And if you join a company where programmers write such code ... well ... good luck to you!!!

Exercise -1.0.20. Can we increment an `int` by a `double`?

```
int i = 5;
double j = 1.5;
i += j;
std::cout << i << ' ' << j << std::endl;
std::cout << (i += j) << ' ' << j << std::endl;
```

Exercise -1.0.21. Can we increment a `double` by a `double`?

```
double i = 5.5;
int j = 1;
i += j;
std::cout << i << ' ' << j << std::endl;
std::cout << (i += j) << ' ' << j << std::endl;
```

Exercise -1.0.22. How about incrementing a `double` by an `int`?

```
double i = 5.5;
int j = 1;
i += j;
std::cout << i << ' ' << j << std::endl;
std::cout << (i += j) << ' ' << j << std::endl;
```

Exercise -1.0.23. Test if `-=`, `*=`, `/=` work with `doubles`. What about `%=`? `for`-loop

for-loop

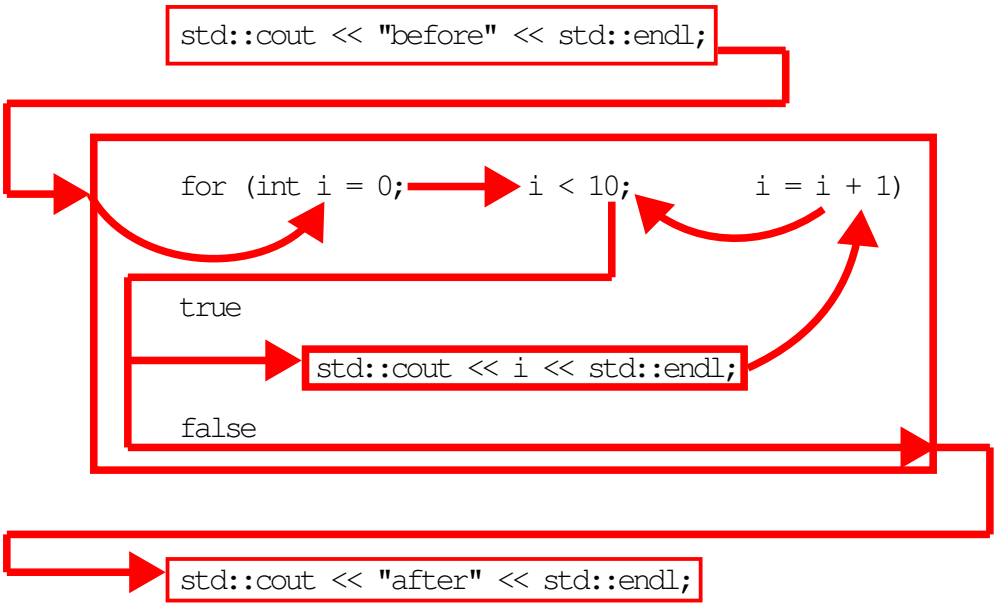
Exercise -1.0.24. Run this

```
std::cout << "before" << std::endl;

for (int i = 0; i < 10; i = i + 1)
    std::cout << i << std::endl;

std::cout << "after" << std::endl;
```

Note that the statement in bold, the `for`-loop, is a single statement.



Here's how the for-loop controls the execution:

```
std::cout << "before" << std::endl;
for (int i = 0; i < 10; i = i + 1)
true
std::cout << i << std::endl;
false
std::cout << "after" << std::endl;
```

Exercise -1.0.25. Rewrite the for-loop using the ++ operator (either pre-increment or post-increment).

More examples

The for-loop is a brand new concept. So let's do more drills before your first warm-up.

Try this:

```
std::cout << "before" << std::endl;

for (int i = 3; i < 7; i++)
    std::cout << i << std::endl;

std::cout << "after" << std::endl;
```

Try this:

```
std::cout << "before" << std::endl;

for (int i = 3; i <= 7; i++)
    std::cout << i << std::endl;

std::cout << "after" << std::endl;
```

Try this:

```
std::cout << "before" << std::endl;

for (int i = 3; i <= 7; i += 2)
    std::cout << i << std::endl;

std::cout << "after" << std::endl;
```

Try this:

```
std::cout << "before" << std::endl;

for (int i = 5; i >= 0; i--)
    std::cout << i << std::endl;

std::cout << "after" << std::endl;
```

Exercise -1.0.26. Warm-up time!!! Modify the previous program so that the output is

```
10
8
6
4
```

Exercise -1.0.27. Write a program that prints 1000 `"hello world!"`s ... oh ... and you have to do it in 1 minute. So either you are a killer typist or you have to use the `for`-loop.

Blocks

Of course it's not too surprising that you can have **blocks** with the `for`-loop:

```
std::cout << "entering the for-loop ...\n";
for (int i = 10; i > 0; i--)
{
        std::cout << "entered the block ...\n";
        std::cout << "i: " << i;
        std::cout << "exiting the block ...\n";
}
std::cout << "done with the for-loop ...\n";
```

It shouldn't be surprising that you can do this:

```
std::cout << "entering the for-loop ...\n";
int x = 10, y = 0;
for (int i = x; i > y; i--)
{
        std::cout << "entered the block ...\n";
        std::cout << "i: " << i;
        std::cout << "exiting the block ...\n";
}
std::cout << "done with the for-loop ...\n";
```

By the way, just like in the case of `if-` and `if-else` statements, C/C++ programming tend to use blocks for the for-loop even when the body of the for-loop has only one statement:

```
for (int i = 0; i < 10; i++)
{
        std::cout << i << std::endl;
}
```

Exercise -1.0.28. Remember our program that prints a table of squares?

```
n   n^2
--- ---
0   0
1   1
2   4
```

Write a program that prints the squares of all positive integers from 0 to 100. Your program must be less than 10 lines long and you have 3 minutes to do that - so I don't mean you have a very long

`std::cout`!!! Here's some pseudocode that might help:

```
Print "n   n^2\n"
Print "--- ---\n"
For n running from 0 to 100 (inclusive):
    print n, spaces, square of n, and newline
```

Exercise -1.0.29. Modify your program so that it prompts the user for integers and assign them to integer variables `start` and `end`. Print a table of squares from `start` to `end`.

Exercise -1.0.30. Can you use a `double` variable to control the for-loop? Try this

```
std::cout << "before" << std::endl;

for (double x = 3.1; x < 7.2; x += 0.1)
{
        std::cout << x << std::endl;
}

std::cout << "after" << std::endl;
```

Exercise -1.0.31. Write a program that prints a table of cubes (like the program on squares above).

Scope

The scope of a variable refers to where you can reference the variable. In the following program:

```
for (int i = 10; i > 0; i--)
{
    std::cout << i;
    std::cout << " ... ";
}
std::cout << "blast off!" << std::endl;
```

The variable `i` is created in the `for`-loop. Once you step out of the `for`-loop, the variable `i` goes out of scope and is **destroyed**.

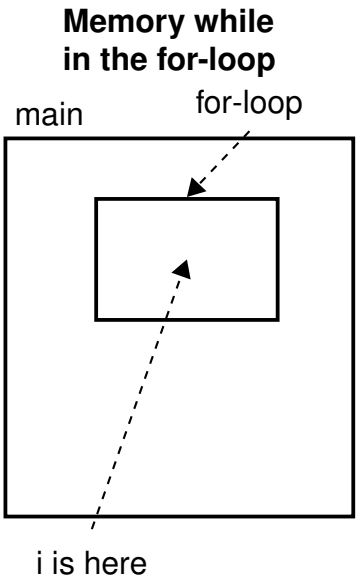
THIS IS VERY IMPORTANT!!!

Try this:

```
#include <iostream>
int main()
{
    for (int i = 10; i > 0; i--)
    {
        std::cout << i;
        std::cout << " ... ";
    }

    std::cout << "blast off!" << std::endl;
    std::cout << i << std::endl;

    return 0;
}
```



(WARNING: If you're using Microsoft Visual Studio, you MUST Disable Language Extensions. Some versions of MSVS do NOT destroy variables created inside a `for`-loop.)

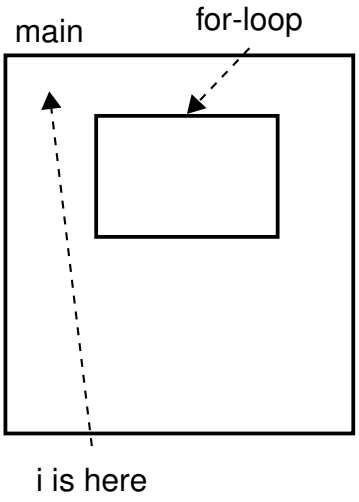
Now try this:

```
#include <iostream>
int main()
{
    int i = 0;
    for (int i = 10; i > 0; i--)
    {
        std::cout << i;
        std::cout << " ... ";
    }

    std::cout << "blast off!" << std::endl;
    std::cout << i << std::endl;

    return 0;
}
```

Memory while in the for-loop



There are other scenarios and other scope rules but this is enough for the time being.

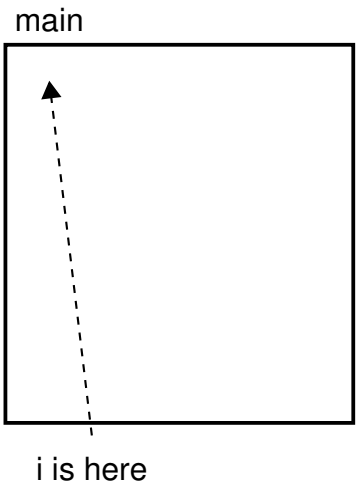
It's really important to understand the scope of a for-loop. The presence of a block (and not just a statement) that is controlled by the for-loop visually makes things slightly more complicated. So let's look at the basic for-loop again.

Here's a simple for-loop:

```
for (int i = 0; i < 10; i++)
    std::cout << i << std::endl;

std::cout << i << std::endl;
```

Memory after the for-loop

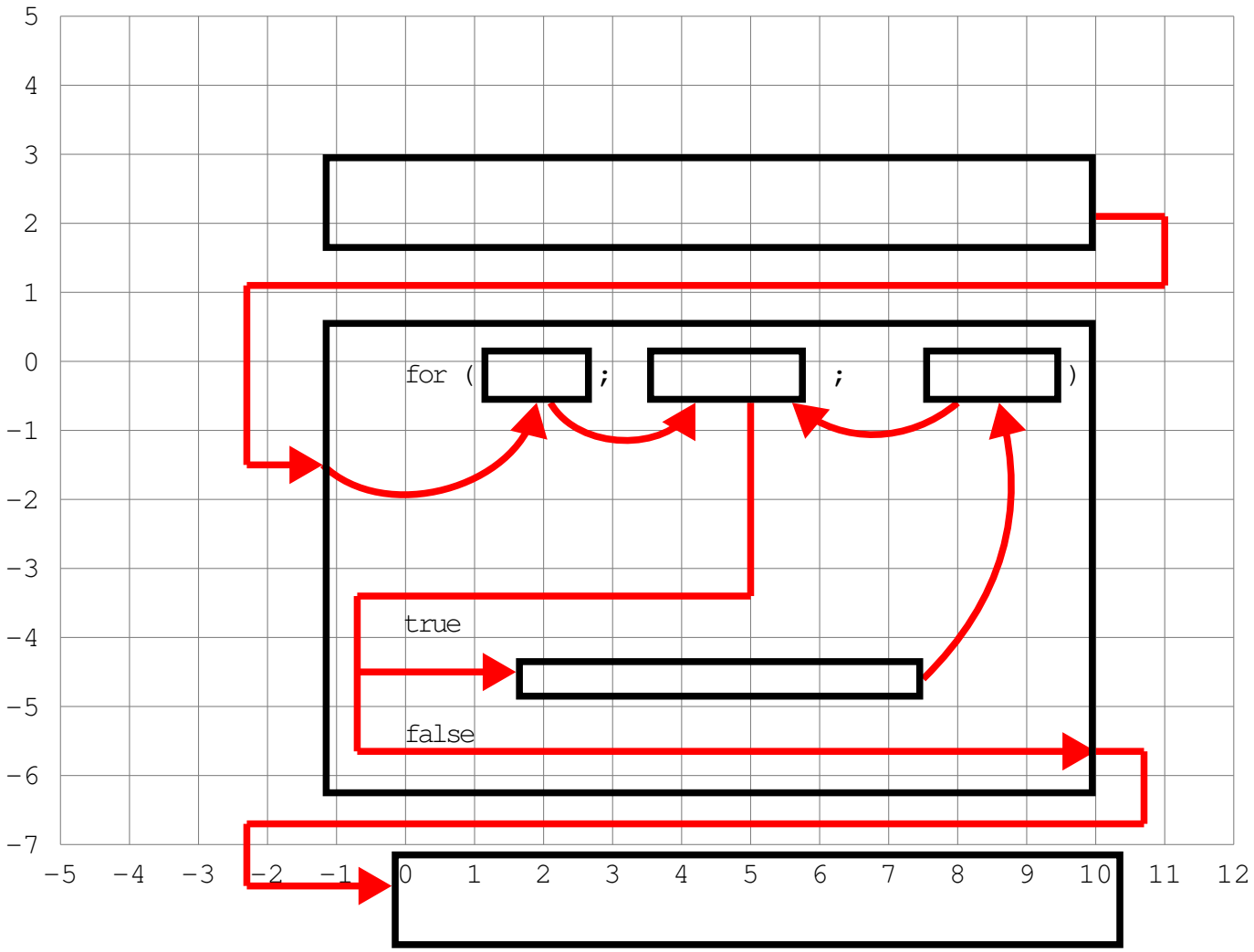


Again you see that the `i` lives inside the `for`-loop and dies and once the `for`-loop is finished, the `i` is destroyed.

There are other scope rules but this is enough for the time being.

Mental picture of flow of execution

Recall that the flow of execution for the for-loop looks like this:



For you to be a competent programmer, it's extremely important to "visualize"the flow of execution and roughly how the computer thinks. So I'm going to simulate the execution of a simple for-loop.

Note that the above simulation is only a ***model***. Furthermore, it does not explain how the computer knew that 4 ¡ 5 was true. And what does "true"really mean to a computer anyway? In fact, does a computer "know"what's a "4" (as in four)??? For details like these you will have to take Assembly Language (CISS360) and Computer Architecture (CISS420).

Exercise -1.0.32. There's one place where you can use the pre-increment operator and another where you can use +=. Modify the above code.

Exercise -1.0.33. Do the same trace with this program:

```
int s = 10;
for (int i = 10; i < 14; i = i + 1)
{
    s = s + i;
}
std::cout << s << std::endl;
```

What is the output? Now run the program with your C++ compiler and verify.

Exercise -1.0.34. Do the same trace with this program:

```
int s = 0;
for (int i = 3; i < 10; i += 2)
{
    s = s + i;
}
std::cout << s << std::endl;
```

What is the output? Now run the program with your C++ compiler and verify.

Exercise -1.0.35. Do the same trace with this program:

```
int s = 0;
for (int i = 0; i < 10; i = i + 1);
    s = s + i;
std::cout << s << std::endl;
```

What is the output (or is there an error)? Now run the program with your C++ compiler and verify.

Using `for`-loop to compute sums

Look at the following program:

```
int sum = 0;
int i = 1;

sum = sum + i;
i = i + 1;

sum = sum + i;
i = i + 1;

sum = sum + i;
i = i + 1;

sum = sum + i;
i = i + 1;

sum = sum + i;
i = i + 1;

sum = sum + i;
i = i + 1;

sum = sum + i;
i = i + 1;

sum = sum + i;
i = i + 1;

sum = sum + i;
i = i + 1;

sum = sum + i;
i = i + 1;

sum = sum + i;
i = i + 1;

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

OK. . . . looks like a long program . . . but most of it repeats.

Now look at this one:

```
int sum = 0;
for (int i = 1; i <= 10; i = i + 1)
{
    sum = sum + i;
}
std::cout << sum << std::endl;
```

Do you see that they do the same thing? Make sure you trace both programs by hand and see that the actual computations are the same.

Exercise -1.0.36. Modify the `i = i + 1` so that the plus augmented operator is used. (Don't peek ahead!)

Exercise -1.0.37. You can get C++ to tell us what's happening in the for-loop by pausing the loop like this:

```
int sum = 0;
for (int i = 1; i <= 3; i++)
{
     char c;
     std::cout << "enter a char to continue ...";
     std::cin >> c;

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

Make sure you try this. This is a **very basic technique** for slowing down a loop so that you can follow what's happening and do **debugging**.

Exercise -1.0.38. Rewrite the program to use an augmented operator. (Yes, there's one other place where you can use an augmented operator.)

Exercise -1.0.39. Modify the above program to compute the sum of all integers from 1 to 100. You want to remove the pause in the middle of the program:

Exercise -1.0.40. Modify the above program to compute the sum of squares from 1^2 to 5^2. Verify your program by calculating this sum with a calculator. Modify your program to compute the sum of squares from 1^2 to 100^2. Of course the number is going to be really huge . . . you do not want to check this one with your calculator!

Exercise -1.0.41. Modify the above program to compute the sum of reciprocals of squares from $1/1^2$ to $1/5^2$. (What type of values are you adding to `sum`? What should the type of `sum` be?) Verify your program by calculating this `sum` with a calculator. Modify your program to compute the sum of reciprocals of squares from $1/1^2$ to $1/100^2$. Compare to the previous exercise and note that the sum stabilizes quickly since the terms you're adding to the sum shrink very quickly. In this case we say that the sum **converges**. You can try to modify your program to add up to $1/1000000^2$. You'll find that it won't change the sum much.

Exercise -1.0.42. Modify the above program to compute

$$1/1 + 1/2 + 1/3 + \ldots + 1/100$$

In this case, the sum seems to stabilize since each term you add to the sum shrinks. However the sum is actually NOT stabilizing. It is actually growing . . . but very slowly. You can try to compute the sum up to 1/1000000.

In general the computation of a sum using a loop looks like this:

sum = 0
for i running from a, a+1, a+2, . . . , b:
  sum = sum + (some term)

For instance:

```
sum = 0;
for (int i = 1; i <= 100; ++i)
{   sum += i;
}
```

In this case the term to add to `sum` is `i`. If you want to sum squares you can do this:

```
sum = 0;
for (int i = 1; i <= 100; ++i)
{
  sum += i * i;
}
```

or this:

```
sum = 0;
for (int i = 1; i <= 100; ++i)
{
  int term = i * i;
  sum += term;
}
```

Of course for such cases, the term can be computed from `i`. There's no reason why the term cannot be supplied by, for instance, a user. Here's one that adds integers specified by a user:

```
int num_accts;
std::cout << "how many bank accounts do you have? ";
std::cin >> num_accts;

double sum = 0;
for (int i = 1; i <= num_accts; ++i)
{
        std::cout << "enter amt in bank acct #"
                  << i << "? $";
        double amt;
        std::cin >> amt;
        sum += amt;
        std::cout << "so far ... you have $"
                  << sum << std::endl;
}
std::cout << "total ... you have $"
          << sum << std::endl;
```

Exercise -1.0.43. Write a program that first asks a user how many houses he/she has. Next the program asks the user for the number of rooms in each house. Finally, the program prints the total number of rooms he/she has for all the properties owned by the user.

Exercise -1.0.44. Write a program that first asks a user how many houses he/she has. Next the program asks the user for the number of rooms in each house and the average square footage for the rooms in that house. Finally, the program prints the total square footage of all the rooms across all the properties owned by the user.

Using for-loop to compute products

Exercise -1.0.45. Trace this by hand:

```
int product = 1;
for (int i = 1; i <= 5; i++)
{
    product *= i;
    std::cout << i << ' ' << product << std::endl;
}
std::cout << product << std::endl;
```

What is the output (or is there an error)? Now run the program with your C++ compiler and verify.

By the way, the product of all positive integer from 1 to 5 (i.e. 1x2x3x4x5) is called the **factorial** of 5. The factorial of 10 is just

1 x 2 x 3 x 4 x 5 x 6 x 7 x 8 x 9 x 10

In general, the factorial of n is such an important quantity that we have a shorthand notation for it: The factorial of n is written n!. So

10! = 1 x 2 x 3 x 4 x 5 x 6 x 7 x 8 x 9 x 10

Note that the factorial of 0 is defined to be 1.

0! = 1

Study the above code. Do you see that it computes the factorial of 5?

The factorial appears in Computer Science, Math, Physics, Chemistry, Finance, etc. So you bet it's important.

Exercise -1.0.46. Compute by hand the factorial of n for n = 0, 1, 2, 3, 4, 5, 6. Use your calculator to verify your work. Using C++, compute the factorial of 10.

Exercise -1.0.47. Why did I initialize `product` with `1` and not `0`? What if you run this program instead?

```
int product = 0;
for (int i = 1; i <= 5; i++)
{
    product *= i;
    std::cout << i << ' ' << product << std::endl;
}
```

And what if I initialize the `product` to 2? (See why it has to be initialized to `1` now?)

Exercise -1.0.48. Write a program that prompts the user for n and prints the product of **odd** integers greater than 0 and less than n.

Exercise -1.0.49. Write a program that prompts the user for n (an integer value at least 0) and prints 2-to-the-power of n. For instance, if the user enters 0, the program prints 1; if the user enters 3, the program prints 8; if the user entered 10, the program prints 1024; Etc.

Exercise -1.0.50. Write a program that prompts the user for `x` (a `double`) and then `n` (an `int` value at least `0`) and prints `x`-to-the-power of `n`. Check your program with several cases, testing it either by hand or by using a calculator.

Exercise -1.0.51. Write a program that prompts the user for `x` (a `double`) and then `n` (an `int` value ***which can be negative***) and prints `x`-to-the-power of `n`. Check your program with several cases, testing it either by hand or by using a calculator.

Exercise -1.0.52. Besides the factorial function n!, there's another quantity that is extremely important in Computer Science, Math, Physics, and you-name-it. It's called the n-choose-r function. It's sometimes written C(n,r). It depends on the factorial. Let n and r be positive integers where r is at most n. Then n-choose-r is given by

$$n! / (r! (n - r)!)$$

n-choose-r is the total numbers of ways to choose r things from n distinct things. For instance, suppose you are an IT manager and have a team of 40 programmers, and you have to form a team of 5

software engineers. The total numbers of ways to form this team is 40-choose-5, i.e.,

40! / (5! x (40 - 5)!) = 40! / (5! x 35!)

However 40! is too huge to compute. Use your program to compute 5-choose-2. Verify the correctness of your program by doing this computation by hand. Here are 5 symbols:

a, b, c, d, e

Write down all possible selections of 2 symbols. Let me begin for you:

a, b
a, c

Note that the selection of a, b is considered the same as the selection of b, a. So do not include b, a. Likewise only include b, e and not e, b.

Using for-Loop to compute min/max

Recall that if you have int variables a, b, c, d, and you want to compute the minimum, you can do this (pseudocode only!):

```
min = a
if b < min:
   min = b
if c < min:
   min = c
if d < min:
   min = d
```

In this case min is an integer variable. (Re-study your notes on the if statement if you don't recall this!!!)

The idea of course works for doubles too. Of course the min variable must be declared to be a double variable. Make sure you understand the above!

The idea for the computation of the maximum is similar.

Now the above sure looks like a loop. If the values are not stored in variables a, b, c, d but for instance is supplied by the user (so we need to do std::cin), then we have a very compact way to compute the minimum of 4 values supplied by the user. Instead of doing

```
prompt user for a
min = a

prompt user for b
if b < min:
   min = b

prompt user for c
if c < min:
   min = c

prompt user for d
if d < min:
   min = d

print min
```

we can do this:

```
prompt user for x
min = x

// 3 more values
for i = 0, 1, 2:
    prompt user for x
    if x < min:
       min = x

print min
```

(user inputs are all stored using x).

The C++ code is then:

```
int x;
int min;

std::cin >> x;
min = x;

// 3 more values, so i = 0, 1, 2 works.
for (int i = 0; i < 3; i++)
{
    std::cin >> x;
    if (x < min)
    {
        min = x;
    }
}
std::cout << min << std::endl;
```

Not only that, this code can be EASILY modified to compute the minimum of 10 values:

```
int x;
int min;

std::cin >> x;
min = x;

// i = 0, 1, 2, 3, 4, 5, 6, 7, 8 --- 9 more values
for (int i = 0; i < 9; i++)
{
    std::cin >> x;
    if (x < min)
    {
        min = x;
    }
}
std::cout << min << std::endl;
```

The old method without loops would take a lot more time to write!

Not only that, the above method works when you allow the user to specify how many values the user wants to input!!

```
int x;
int min;
int n; // number of values for the min computation

std::cin >> n;
std::cin >> x;
min = x;

for (int i = 0; i < n - 1; i++)
{
    std::cin >> x;
    if (x < min)
    {
        min = x;
    }
}

std::cout << min << std::endl;
```

As you can see, the loop is indispensable!!!

Exercise -1.0.53. Write a program that prompts the user for n and then prompts the user for n double values and then prints the maxi-

mum of the values enters by the user.

Exercise -1.0.54. Write a program that prompt the user for n and then prompts the user for n double values and then prints the minimum of the **absolute values** of the values entered by the user. Recall from MATH104 that the absolute value of x is just x without the negative sign. So for instance

$$|42| = 42$$

$$|-24| = 24$$

The absolute value functions were mentioned in the set of notes on doubles. For your convenience, here's the information again. For C++, if you need the absolute value of an integer value this is how yo do it:

```
#include <iostream>
#include <cstdlib>

int main()
{
    int x = 42, y = -24;
    std::cout << abs(42) << '\n';
    std::cout << abs(-42)<< '\n';
    std::cout << abs(x) << '\n';
    std::cout << abs(y) << '\n';
    return 0;
}
```

If you need to compute the absolute value of doubles or floats you do this:

```
#include <iostream>
#include <cmath>
int main()
{
    int x = 4.2, y = -2.4;
    std::cout << fabs(42) << '\n';
    std::cout << fabs(-42)<< '\n';
    std::cout << fabs(x) << '\n';
    std::cout << fabs(y) << '\n';
}
return 0;
```

The boolean condition

Here's one of our first few programs:

```
std::cout << "entering the for-loop ...\n";
for (int i = 10; i > 0; i--)
{
    std::cout << "entered the block ...\n";
    std::cout << "i: " << i;
    std::cout << "exiting the block ...\n";
}
std::cout << "done with the for-loop ...\n";
```

Look at this boolean expression:

```
std::cout << "entering the for-loop ...\n";
for (int i = 10; i > 0; i--)
{
    std::cout << "entered the block ...\n";
    std::cout << "i: " << i;
    std::cout << "exiting the block ...\n";
}
std::cout << "done with the for-loop ...\n";
```

One of the most common misconceptions about the for-loop (and actually applies to all loops) is that many first-time programmers think that if the boolean expression

$$\mathbf{i > 0}$$

is false at **any point** in the loop, you will break you out of the for-loop immediately. That's **NOT TRUE**. Look at the picture on page 12 again. The check on whether to break out of the loop occurs at a specific point. If I do this instead:

```
std::cout << "entering the for-loop ...\n";
for (int i = 10; i > 0; i--)
{
    std::cout << "entered the block ...\n";
    std::cout << "i: " << i;
    i = 0;
    std::cout << "exiting the block ...\n";
}
std::cout << "done with the for-loop ...\n";
```

The program would still execute the last statement in the body of the for-loop:

```
std::cout << "exiting the block ... n";
```

because the boolean check comes after this print statement.

MAKE SURE YOU REMEMBER THAT!

Generating Equally Spaced Points on a Line

In the next two sections, we will generate points in a range which are equally spaced. This is what I mean:

Suppose I tell you that I want 5 equally spaced points on the interval [0, 4] that includes the end points 0 and 4. You would say:

$$0, 1, 2, 3, 4$$

There are 5 values. They are equally spaced: consecutive values differ by 1. The first value 0 is the left endpoint of [0, 4] and the last value 4 is the right endpoint of [0, 4].

What if I ask you for 3 equally spaced points on [0, 4] including both end points? Well this means that you want 3 values, say we call them

$$x0, x1, x2$$

where x0 is 0 and x2 is 4. What about x1? x1 is right in the middle. So it must be

$$x1 = (x2 – x1) / 2 = 2$$

So the values are

$$0, 2, 4$$

Great! Done!

What if I ask you for 4 equally points in [0, 4]? Let's call then

$$x0, x1, x2, x3$$

Of course

$$x0 = 0 \; x3 = 4$$

What about x1 and x2? Well since they are equally spaced, let's say the gap between x0 and x1 is d. Since the points are equally spaced, the gap is of course also the gap between x1,x2 and also x2,x3. There are altogether 3 gaps and they add up to 4. So

$$d = 4/3$$

(as a real number, i.e., double. The above is math, not C++) In other words:

d = 1.3333.... (up to 4 decimal places)

This mean that

$$\begin{aligned} x0 &= 0 \\ x1 &= 0 + d = 1.3333.... \\ x2 &= x1 + d = 2.6666... \\ x3 &= x2 + d = 4 \end{aligned}$$

What if I want **7** points in the same interval? Say the points are

x0, x1, x2, x3, x4, x5, x6

There are **6** gaps. Call the gap d. The gaps add up to 4. Therefore

$$6d = 4$$

i.e.,

$$d = 4/6 = 0.6666...$$

and the values of the points are

$$\begin{aligned} x0 &= 0 \\ x1 &= x0 + d \\ x2 &= x1 + d \\ x3 &= x2 + d \\ x4 &= x3 + d \\ x5 &= x4 + d \\ x6 &= x5 + d \end{aligned}$$

Writing this as a C++ program I get

```
double x = 0;
double d = double(4) / 6
for (int i = 0; i < 7; ++i)
{
    std::cout << x << '\n';
    x += d
}
```

Run this and check that you do get 7 points which are equally spaced.

Exercise -1.0.55. Modify the following code to generate equally spaced points in [0, 4] if the number of points is n.

```
int n;
std::cin >> n;
double x = 0;
double d = double(4) / 6;
for (int i = 0; i < 7; ++i)
{
    std::cout << x << '\n';
    x += d
}
```

Test your program for different values of `n`.

Exercise -1.0.56. If the interval is not [0, 4] but [0, b] and the number of points is n, then modify the following code as appropriate:

```
int n = 0;
double b = 0.0;
std::cin >> b;
std::cin >> n;

double x = 0;
double d = double(4) / 6;
for (int i = 0; i < 7; ++i)
{
    std::cout << x << '\n';
    x += d
}
```

Test your code thoroughly.

Exercise -1.0.57. If the interval is [a, b] and the number of points is n, then modify the following code as appropriate:

```
int n = 0;
double a = 0.0, b = 0.0;
std::cin >> a >> b;
std::cin >> n;

double x = 0;
double d = double(4) / 6;
for (int i = 0; i < 7; ++i)
{
    std::cout << x << '\n';
    x += d
}
```

Test your code thoroughly. [Hint: Move the interval so that the left endpoint is 0. The interval becomes [0, b – a]. Get the points using the previous exercise. Now move the points back to [a, b].

WARNING . . . INCOMING SPOILER!!!

Here's the answer to the above. If you are generating n equally spaced points in the interval a, b] (and the left and right end points are included), the code is

```
double x = a;
double d = double(b - a) / (n - 1);
for (int i = 0; i < n; ++i)
{
    std::cout << x << '\n';
    x += d
}
```

Note that because of rounding errors (remember that doubles and floats are not exact!), the last point might not be exactly the value b. You can stop the for-loop earlier and manually use b for the last point:

```
double x = a;
double d = double(b - a) / (n - 1);
for (int i = 0; i < n - 1; ++i)
{
    std::cout << x << '\n';
    x += d;
}

x = b;
std::cout << x << '\n';
```

In the next couple of sections, we will be using these equally spaced

points to do some computations. So in the general case, the code will look something like this:

```
double x = a;
double d = double(b - a) / (n - 1);
for (int i = 0; i < n - 1; ++i)
{
{... do something with x ...}
x += d;
}
x = b;
... do something with x ...
```

Sums again: Area computation

I'm going to use the `for`-loop to compute a certain sum. This time it's going to be the area under a curve. The computation of areas is extremely important in many areas of science.

The idea is actually very simple: you need to know how to compute the area of rectangles and you need to be able to add lots of these areas.

The idea is pretty simple and illustrates the power of being able to do perform repetitions quickly.

First you draw the graph of the function, for example, y = f(x) = x*x, the standard parabola):

Suppose you want to compute the area under this curve and above the x-axis and from x = 2 to x = 7:

You can make an approximation by computing the areas of the rectangles with

x=2 to x=3 with height f(2)
x=3 to x=4 with height f(3)
x=4 to x=5 with height f(4)
x=5 to x=6 with height f(5)
x=6 to x=7 with height f(6)

There are altogether 5 rectangles. Draw these rectangles into the graph above.

Note that for each rectangle, the base has width 1. Also, for each rectangle, I am using the left endpoint on the width to determine the height. The areas of these rectangles is smaller than the area under the parabola from x = 2 to 2 = 7.

Exercise -1.0.58. Write a for-loop to compute the sum of the above rectangles. Write down the sum of areas. We'll need it for comparison later.

Now we "move"the sum of areas of our rectangles towards the parabola

in the following way: we will use **more** rectangles. Previously the base of the rectangles had a width of 1. Now I'm going to use rectangles with base of length 0.5. Altogether we now have 10 rectangles:

x=2.0 to x=2.5 with height f(2.0)
x=2.5 to x=3.0 with height f(2.5)
x=3.0 to x=3.5 with height f(3.0)
x=3.5 to x=4.0 with height f(3.5)
x=4.0 to x=4.5 with height f(4.0)
x=4.5 to x=5.0 with height f(4.5)
x=5.0 to x=5.5 with height f(5.0)
x=5.5 to x=6.0 with height f(5.5)
x=6.0 to x=6.5 with height f(6.0)
x=6.5 to x=7.0 with height f(6.5)

Draw these 10 rectangles into the above graph.

Do you see that the total area of these rectangles are closer to the actual area under the parabola than the one from our first approximation?

Do you see that, intuitively, the total area of the rectangles gets closer and closer to the area under the parabola from x = 2 to x = 7 as the number of rectangles used increases? Here's the case where the length of the base of the rectangles is 0.25:

and here's the case where the length is 0.125

Using the code that computes equally spaced point, the sum of the areas of these rectangles can thus be computed as follows:

```
double a = 2;
double b = 7;
int n = 11; // 10 rects mean 11 points
double sum = 0;
double x = 0;
double d = double(b - a) / (n - 1);
for (int i = 0; i < n - 1; ++i)
{
    sum += d * (x * x);
    x += d;
}

// Why do we NOT need to execute the following?
// x = b;
// sum += d * (x * x);
```

Exercise -1.0.59. Compute area when 20 rectangles are used.

Exercise -1.0.60. Now do the same where you use 100 rectangles.

Now write a program that prompts the user for the number of rectangles! Say we call this variable n. Run the program with n = 1000 rectangles!!! You bet the value computed by your program is extremely close to the real area of the parabola. Of course you wouldn't add the areas of 1000 rectangles by hand – it would take too long! That's what a program is for – if you know C++!!!

Exercise -1.0.61. If you know some Calculus 1, you know that the area under the parabola from x = 2 to x = 7 is in fact exactly

$$(1/3)(7 * 7 * 7 - -2 * 2 * 2)$$

Use your calculator (or C++) to compute this value and compare against the value obtained by your program with n = 1000 rectangles.

Exercise -1.0.62. The above prompts the user for n, the number of rectangles to use for the area computation using a for-loop. Write a nested for-loop where the outer for-loop supplies values of n running from 1000 to 1000000 by increments of 1000. The pseudocode looks

like this:

for n = 1000, 2000, 3000, . . . , 1000000:
compute and print the area using n rectangles

Exercise -1.0.63. Modify the above program so that instead of computing the area under the parabola from x = 2 to x = 7, the program prompts the user for a, b, n and computes the area under the parabola from x = a to x = b using n rectangles.

Exercise -1.0.64. What is the area under the curve y = x*x*x from x=0 to x=10?

The computation of areas under a curve is important. For instance in Physics, the work done by a force F is the area under the force-displacement graph where you plot the force acting on an object and the displacement of the object. By Newton's law of universal gravitation, we know the gravitational force between two objects which are apart from each other at a fixed distance. So if a rocket wants to move away from earth, you can compute the work done moving the rocket to a distance from the surface of earth to a far away point in space. Knowing the work to be done moving the rocket from earth through a required distance, you can then compute the amount of chemical fuel that is needed to combust in order to convert the chemical energy into required work done. Of course there are many other applications of the computation of areas.

Root finding: An inefficient method

Another very common problem in Math and Sciences is finding solution(s) to an equation such as finding solutions to quadratic equations:

$$2 * x * x + 4 * x - 3 = 0$$

This is easy since we have the quadratic equation formula. But this is highly non-trivial if the equation is more complex such as a degree 5 equation:

$$2 * x * x * x * x * x + 4 * x * x - 3 * x + 10 = 0$$

or even one with trigonometric, logartihmic, and/or exponential functions!!!

For instance, what if I want to find a solution to

$$3 * x * x - 10 * x * sin(x) - 500 = 0$$

in the range [0, 20]? You don't recall a solution to such an equation from your math classes right?!?

Since we don't have a formula to use, we just have to plot the graph and visually see where the function touches the x-axis. In the same way, we can compute the values of the above expression

$$3 * x * x - 10 * x * sin(x) - 500$$

for many many many many many values in the range [0, 20] and see which value of x will give a value of

$$3 * x * x - 10 * x * sin(x) - 500$$

that is very close to 0. For instance we can test 1,000,000 values in the range [0,20]. If we do this by hand, we would go crazy. But ...

We know C++!!!!

All we need to do is to run a for-loop of 1,000,000 values in the range [0,20] and find the value of x that will get use as close as possible to the y-axis. Close to the y-axis means

$$|f(x)|$$

is close to zero. In other words, if you're finding the point closest to the y-axis it means that you need to find the smallest $|f(x)|$ for the 1,000,000 x values in the range.

Exercise -1.0.65. Using a `for`-loop, find a value for x from 1 to 5 such that x sin(x) + 1 is as close as possible to the x-axis.

Maximum and minimum of functions

Using the idea of generating lots and lots of equally spaced points on an interval, you can very quickly compute the approximate maximum and minimum of functions. Note that the max or min is only an approximation since your points might miss the actual maximum and minimum.

Exercise -1.0.66. Write a program to compute and print the maximum value of the sine function on the interval [0, 10] using 1,000,000 equally spaced points. The sine function is already mentioned in the notes on floating point values.

Exercise -1.0.67. Modify the above program so that it prints the value of x where the maximum occurs.

Exercise -1.0.68. Compute and print the maximum and minimum value of the function

$$f(x) = x \sin(x)$$

on the interval [0, 20] using 1,000,000 points. When you're done, modify the program to also print the x value where the maximum and minimum occurs.