C++ PROGRAMMING

DR. YIHSIANG LIOW (DECEMBER 22, 2025)

Contents

16. Scope

OBJECTIVES

- Understand the scope of a variable
- Understand how C++ searches for a variable
- Understand the importance of using minimal scopes

I' ve already mentioned scopes of variables in several places. In this set of notes, I will collect what you already know about scopes. So much of it is review. I will also add a few new scope rules.

Scope

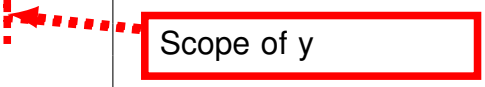
A variable has a name, a value, and you should know by now, a variable also has a **scope**. This refers to the place in your code where you can refer to that variable.

Here's an old example (very very very old ...):

```
x = 42; // x not declared yet bozo!!!

int x = 0;
std::cout << x << std::endl;
```

In general the scope of a variable is from the **point of declaration** to the **end of the block where it is declared**. During the execution of the program, when the point of execution exits that block, the variable is destroyed. Try this:

```
#include <iostream>

int main()
{
    int x = 0;
    std::cin >> x;

    if (x == 42)
    {
       std::cout << "here we go ... " << std::endl;
       int y = x + 1;
       std::cout << "y: " << y << std::endl;
    }
    std::cout << "x: " << x << std::endl;

    return 0;
```



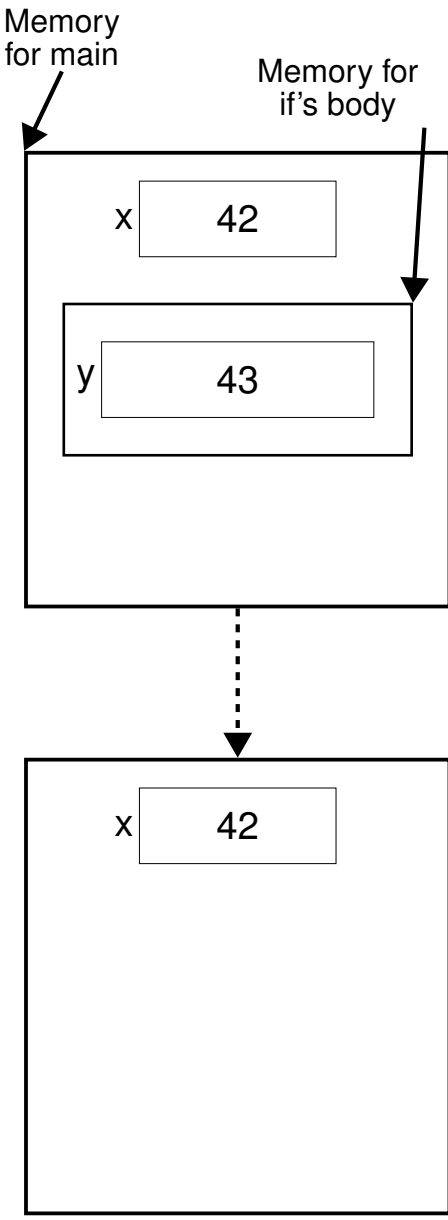
Of course you should know by now that this won't work (try it):

```
#include <iostream>

int main()
{
     int x = 0;
     std::cin >> x;

     if (x == 42)
```

```
    {
        std::cout << "here we go ..." << std::endl;
        int y = x + 1;
        std::cout << "y: " << y << std::endl;
    }
    std::cout << "y:  " << y << std::endl;

    return 0;
}
```

You should visualize the variables in your code as being created in blocks of memory spaces. Suppose the user enters 42 for `x` and we are about to execute the print statement in the body of the `if` statement. The memory looks like this:

```
#include <iostream>

int main()
{
     int x = 0;
     std::cin >> x;

     if (x == 42)
     {
          std::cout << "here we go ..." << std::endl;
          int y = x + 1;
          std::cout << "y:  " << y << std::endl;
     }
     std::cout << "y: " << y << std::endl;

     return 0;
}
```




```
#include <iostream>

int main()
{
     int x = 0;
     std::cin >> x;

     if (x == 42)
     {
          std::cout << "here we go ..." <<
          std::endl;
          int y = x + 1;
          std::cout << "y: " << y << std::endl;
     }
     std::cout << "y:  " << y << std::endl;

     return 0;
}
```

As you can see, immediately on exiting the `if` statement, the **variable in the body of the `if` statement is destroyed**.

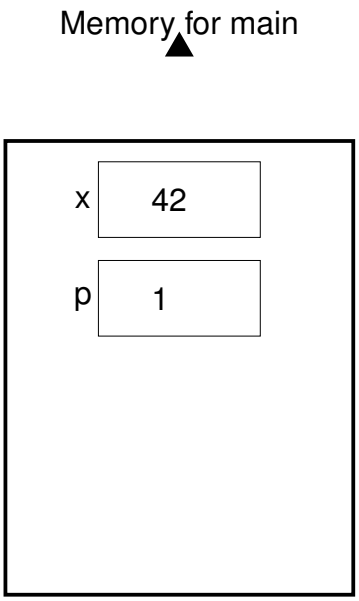
Of course `x` was created in the `main()` block. So `x` is not destroyed.

This is one of the reasons why proper indentation is so important. Not only does it help you read the logic of your code, it tells you when variables are destroyed.

`for`-loops and `while`-loops

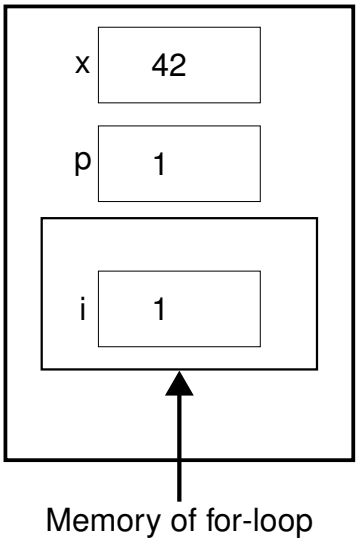
You should think of the `for`-loop as having a block and the body of the for-loop as having an inner block.

```
#include <iostream>

int main()
{
      int x = 42, p = 1;

      for (int i = 1; i <= x; ++i)
      {
            int y = i * i;
            p *= y;
      }
      std::cout << p << std::endl;

      return 0;
}
```



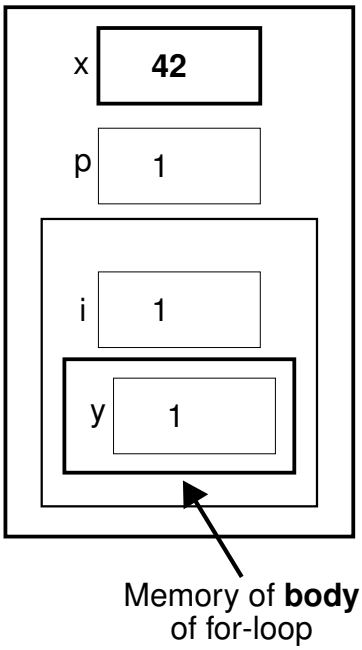
As we enter the `for`-loop, we get a block of memory for the `for`-loop. Variable `i` in the `for`-loop resides in this block.

```
#include <iostream>

int main()
{
      int x = 42, p = 1;
      for (int i = 1; i <= x; ++i)
      {
          int y = i * i;
          p *= y;
      }
      std::cout << p << std::endl;

      return 0;
}
```



When we step into the body of the `for`-loop, we have **another** block. Variable `y` is created in this inner block:

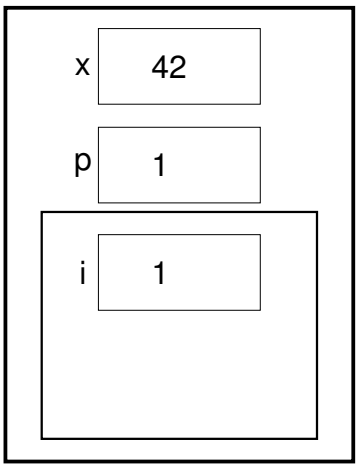
```
#include <iostream>

int main()
{
      int x = 42, p = 1;
      for (int i = 1; i <= x; ++i)
      {
            int y = i * i;
            p *= y;
      }
      std::cout << p << std::endl;

      return 0;
```



Memory of **body** of for-loop

When the last statement of the body of the `for`-loop is executed, we exit the body and variable `y` is destroyed:

```
#include <iostream>

int main()
{
      int x = 42, p = 1;

      for (int i = 1; i <= x; ++i)
      {
            int y = i * i;
            p *= y;
      }
      std::cout << p << std::endl;

      return 0;
```

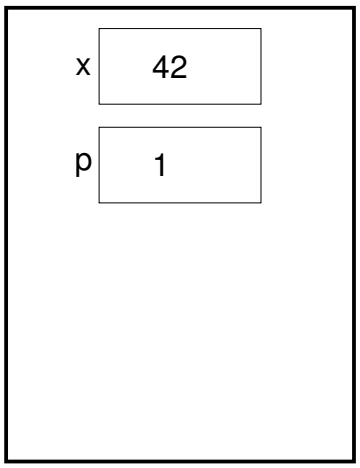


Of course when we enter the body of the `for`-loop the second time (when `i` = 2), variable `y` is created again.

Get it?

And when we exit the `for`-loop altogether, variable `i` is destroyed because it is created in the `for`-loop:

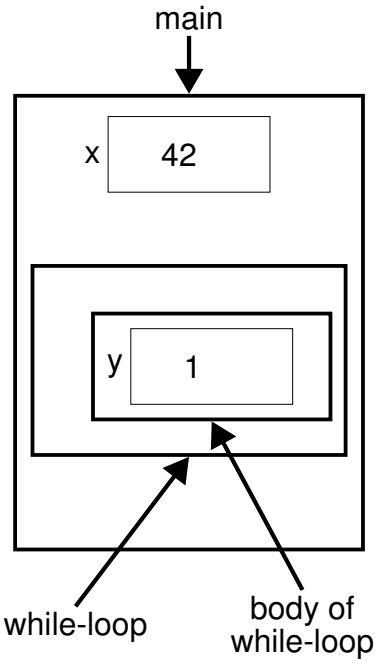
```
#include <iostream>

int main()
{
      int x = 42, p = 1;

      for (int i = 1; i <= x; ++i)
      {
            int y = i * i;
            p *= y;
      }
      std::cout << p << std::endl;

      return 0;
```



Note that variable `x` and `p` are still in scope; they are not destroyed.

What about the `while`-loop? The idea is similar: The `while`-loop has it' s memory and the body of the `while`-loop has an inner block.

```
#include <iostream>

int main()
{
      ...
      int x = 42;

      while (...)
      {
            int y = 24;
            ...
      }

      ...
}
```



So remember this: each loop has it' s own memory for variables and the body of the loop itself has its own memory.

Exercise -1.0.1. What is the problem? Fix it!

```
std::cout << "Compute product of squares.";
std::cout << "Enter 0 to stop.\textbackslash n";

int prod = 1;
int x;
std::cin >> x;
while (x != 0)
{
     int square = x * x;
     prod = prod * square;
     std::cout << "Product:" << prod << "\n";
     std::cin >> x;
}

std::cout << "\nProduct:" << prod << std::endl;
std::cout << "Last square: " << square
          << std::endl;
```

If you need **several** temporary variables in a `for`-loop you can do this:

```
for (int i = 0, s = 5, t = 100; i < 10; i++)
{
    ...
}
```

For instance if you want to print the running partial sums of summing 1 to 100 you can do this:

```
for (int i = 0, s = 0; i < 10; i++)
{
    s += i;
    std::cout << i << ' ' << s << '\n';
}
```

Of course note that in this case `s` is local to the `for`-loop and hence goes out of scope when the `for`-loop is done, i.e., it cannot be referenced outside the `for`-loop:

```
for (int i = 0, s = 0; i < 10; i++)
{
    s += i;
    std::cout << i << ' ' s << '\n';
}
std::cout << s << '\n'; // BADDDDDDDDDDDD!!!!
```

Exercise -1.0.2. Can you also create variables in the header of an `if` statement? Can this work:

```
int i = 0;

if (int x = i * i < 10)
{

    ... do something with i and x ...

}
     ... do not use x!!! ...
```

Variables with the same name

Pay attention!!! This is something new!!!

You can actually create variables with the same name ... provided they are either declared in non-overlapping blocks or one is defined in an inner block.

Here' s the case where names are created in **disjoint blocks**:

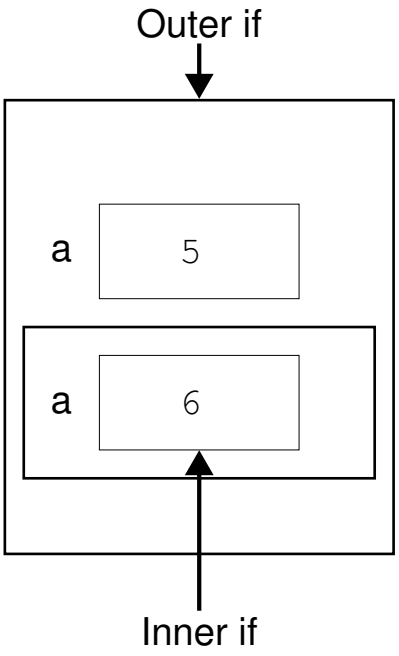
```
...

for (int i = 0; i < 10; ++i)
{
     int x, y, z;
     ...
}

for (int i = 0; i < 100; ++i)
{
     ...
     int y;

}

...
```

No problem! You won't get a re-declaration because the previous i is already destroyed!

Do you see why there won' t be conflicts or confusion to C++?

Here' s the case where two names are created in **nested blocks**:

```
int x = 0, y = 0;
std::cin >> x >> y;

if (x == 0)
{
     int a = 5;
     std::cout << a << std::endl;

     if (y == 1)
     {
          int a = 6;
          std::cout << a << std::endl;
     }
}
```



YIKES!!!

Note that the first print statement is not ambiguous. (Why?)

But what about the second? Which `a` is used in the second print statement???... go to next section to find out ...

WARNING: Although technically you can do the above (i.e. have two variables with the same name nest within blocks, it makes the code hard to read. Not only that, some languages do not allow this at all. Therefore you should **AVOID DECLARING VARIABLES WITH THE SAME NAME IN NESTED BLOCKS**. The only **exception** is a temporary (scratch) variable for instance like the counter variable of a `for`-loop

```
for (int i = 0; i < 100; i++)
{
      ...
      for (int i = -10; i < 10; i++)
      {
            ...
      }
      ...
}
```

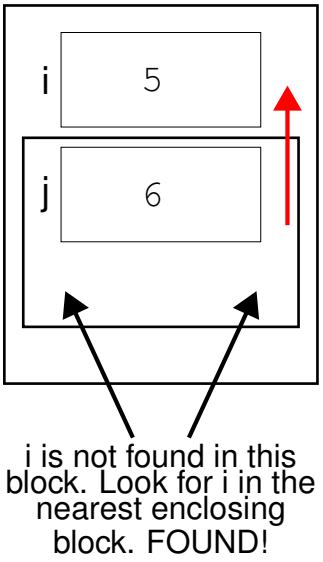
and only when the logic is easy to understand. But even then, you might still be shooting yourself in your foot later if you need to change the logic of your code.

How does C++ search for a variable?

In looking for a variable C++ will always start from the block where the execution occurs. If the variable cannot be found, C++ will look for that variable in the nearest outer enclosing block.

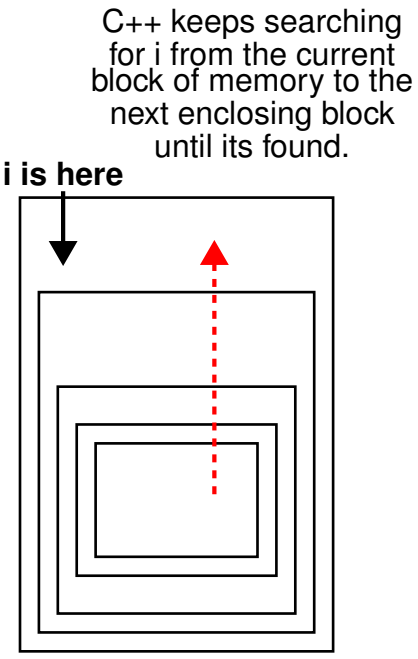
Here's a simple example:

```
#include <iostream>

int main()
{
      int i = 5;

      if ( i > 0 )
      {
            int j = 6;
            std::cout << i;
      }

      return 0;
}
```



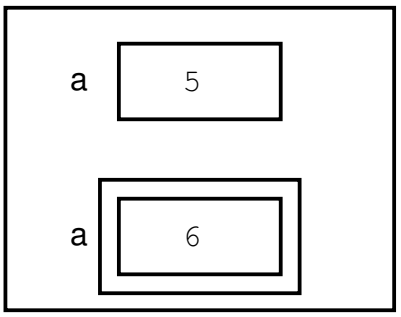
Of course C++ might need to search in several enclosing blocks before it's found.

```
#include <iostream>

int main()
{
      int i = 5;
      while (i > 0)
      {
            int j = 6;

            for (int k = 0; k < j; ++k)
            {
                  std::cout << i;
            }
            --i;
      }
}
```



Exercise -1.0.3. Now look at the example from the previous section. Why is C++ not confused?

```
int x = 0, y = 0;
std::cin >> x >> y;

if (x == 0)
{
      int a = 5;
      std::cout << a << std::endl;
      if (y == 1)
      {
            int a = 6;
            std::cout << a << std::endl;
      }
}
```



In this case when the variable `a` in the inner if-statement is created, we say that the innermost `a` **hides** the outer `a`.

Exercise -1.0.4. What is the output? (Or is there an error?)

```
int x = 0, y = 1;

if (x == 0)
{
      int a = 5;
      if (y == 1)
      {
            std::cout << a << std::endl;
            int a = 6;
            std::cout << a << std::endl;
      }
}
```

Exercise -1.0.5. What is the output? (Or is there an error?)

```
int x = 0, y = 1;

for (int x = 5; x < 10; x++)
{
     std::cout << x << ' ' << y << std::endl;
     int y = 5;
     while (y > 0)
     {
          int x = y + 1;
          std::cout << x << ' ' << y << std::endl;
          y++;
     }
     std::cout << x << ' ' << y << std::endl;
}
```

Exercise -1.0.6. What is the output? (Or is there an error?)

```
int i = 5, j = 6;
std::cout << i << ' ' << j << std::endl;
for (int i = 0; i < 3; i++)
{
      std::cout << i << ' ' << j << std::endl;
      if (i < j)
      {
           std::cout << i << ' ' << j << std::endl;
           int i = 24;
           std::cout << i << ' ' << j << std::endl;
      }
      else
      {
           std::cout << i << ' ' << j << std::endl;
           int j = 42;
           std::cout << i << ' ' << j << std::endl;
      }
      std::cout << i << ' ' << j << std::endl;
}
std::cout << i << ' ' << j << std::endl;
```

Exercise -1.0.7. What is the output? (Or is there an error?)

```
int i = 5, j = 6;}
std::cout << i << ' ' << j << std::endl;
for (int i = 0; i < 3; i++)
{
     int i = 8;
     std::cout << i << ' ' << j << std::endl;
     if (i < j)
     {
          std::cout << i << ' ' << j << std::endl;
          int i = 24;
          std::cout << i << ' ' << j << std::endl;
     }
     else
     {
          std::cout << i << ' ' << j << std::endl;
          int j = 42;
          std::cout << i << ' ' << j << std::endl;
     }
     std::cout << i << ' ' << j << std::endl;
     int i = 9;
}
std::cout << i << ' ' << j << std::endl;
```

Principle of minimal scope

Here's the principle of minimal scope:

Keep the scope of variables as small as possible.

(The above is suppose to blink).

For instance this is NOT GOOD:

```
// compute the factorial of n
int i = 0;
int prod = 1;

int n = 0;
std::cin >> n;

for (i = 1; i <= n; i++)
{
    prod *= i;
}
std::cout << prod << std::endl;
```

This is BETTER

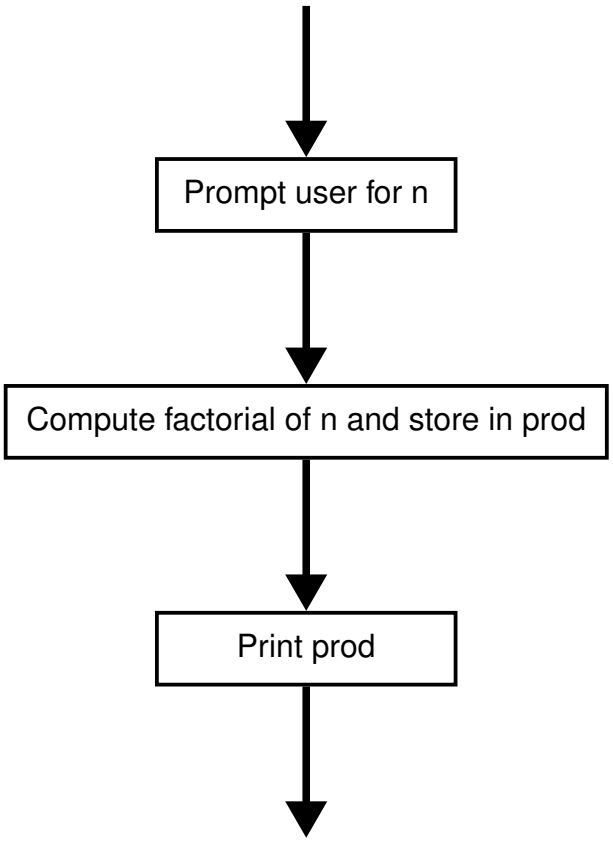
```
// compute the factorial of n
int prod = 1;

int n = 0;
std::cin >> n;

for (int i = 1; i <= n; i++)
{
     prod *= i;
}
std::cout << prod << std::endl;
```

Exercise -1.0.8. What's the difference?

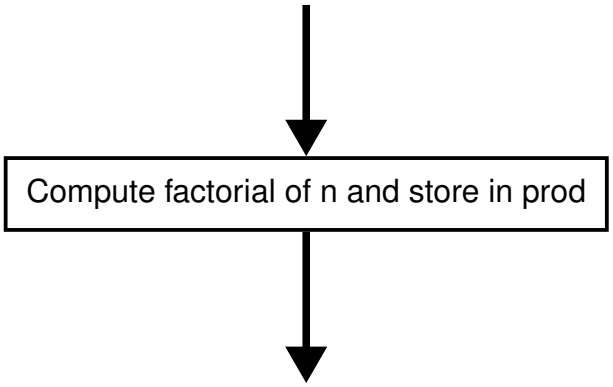
Why? Frequently when you're solving a problem (not just a programming problem), you go through a sequence of steps of computation or transformation on the data. Each step usually requires some input data to modify something (such as a variable) or to perform a task

(such as printing something to the screen in a game).

For our program above you should be able to see these steps:

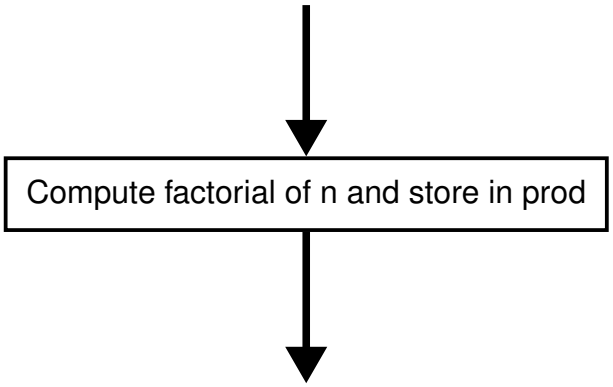


A well-written program will always try to minimize the inputs and outputs (or change) within a step such as this:



Why? Because the more stuff you have in a step the more complex it will be. That means that the code is harder to read, debug, and

maintain for future change. In the real world, software engineers write new code and maintain code base (maintain = correct, add and delete). Now look at this:



What variables are involved in this step? Don't think about the internals of this chunk of code. Think of the code as a black box. You need n (that's the input) and you want to compute the factorial of n (that's the output) and you want to put that value in to prod (that's the transformation).

So two variables take part in this step.

Now look at the code from the first program (the part in bold):

```
// compute the factorial of n
int i = 0;
int prod = 1;

int n = 0;
std::cin >> n;

for (i = 1; i <= n; i++)
{
    prod *= i;
}

std::cout << prod << std::endl;
```

You see clearly that besides "sending in" variables `n` and `prod` to this step, **you also "send in" variable i**.

Now look at the improved version:

```
// compute the factorial of n
int prod = 1;
```

```
int n = 0;
std::cin >> n;

for (int i = 1; i <= n; i++)
{
    prod *= i;
}

std::cout << prod << std::endl;
```

Variable `i` is internal to the step that computes the factorial of `n` and stores it in `prod`. Outside the code that computes and stores the factorial of `n` in `prod`, variable `i` does not exist.

Get it?

This also prevent "side effects" in code. (I've already mentioned this.) If a variable has a large scope, and you modify it in some place, it might have unexpected effects. Here's something similar to a previous example – it's important enough to repeat!!! For instance suppose work in a team and you're given this code segment:

```
...
int i = 42;
...
for (; i < 100; i++)
{
    ...
}
....
```

where you expect the `for`-loop to run from `i` = 42 to `i` = 99. One of your colleagues might see the variable `i` and add some code like so:

```
...
int i = 42;
...
for (i = 1000; i > 0; --i)
{
     ...
}
...
for (; i < 100; i++)
{
     ...
}
....
```

So guess what? When you use `i`, the initial value for you is now 0 and not 42 and your program doesn't work anymore. Tough. What's worse is when you try to debug, you might not even see it because you kept saying to yourself "But, but, but ... I didn't change anything ever since it was working!!! ... and my code looks the same as before!!!"

This is one of the most important reasons why we also try to restrict the scope of a variable. If it's meant to be a "scratch" variable, then create it in such a way that it gets destroy ASAP.

As you can see from the example, one way to minimize scopes is to declare only when you need to and in the **innermost block** whenever possible. For our example, we declare `i` within the for-loop block.

NOTE: Some people declare all variables at the beginning of a function. That's because **some** programming languages specify that **all** declaration statements must come before other types of statements. That's not the case for C++ or other modern programming languages.

Now let's go back to that program again. There's one other spot where you can minimize the scope of a variable.

```
// compute the factorial of n
int prod = 1;

int n = 0;
std::cin >> n;

for (int i = 1; i <= n; i++)
```

```
{
     prod *= i;
}

std::cout << prod << std::endl;
```

Note that `prod` is not used until the `for`-loop. So you can write this:

```
// compute the factorial of n
int n = 0;
std::cin >> n;

int prod = 1;
for (int i = 1; i <= n; i++)
{
     prod *= i;
}
std::cout << prod << std::endl;
```

This however is **not as crucial** as the previous change because `prod` is not destroyed. Furthermore `prod` is part of the larger goal of computing the factorial. Hence it's OK to declare `prod` early so that readers of your code can see it early, especially if you mention `prod` in your comments like this:

```
// compute the factorial of n and store it in prod
int n = 0;
int prod = 1;

std::cin >> n;

for (int i = 0; i <= n; i++)
{
     prod *= i;
}

std::cout << prod << std::endl;
```

See that?

Exercise -1.0.9. Rewrite the following program so that variables have minimal scopes.

```
#include <iostream>

int main()
{
     int answer, i, first, last, sum, term;
     std::cout << "Find sum? (1 -- yes or 0 - no) ";
     std::cin >> answer;

     if (answer == 1)
     {
          std::cout << "First int:";
          std::cin >> first;
          std::cout << "Last int:"
          std::cin >> last;

          sum = 0;
          for (i = first; i <= last; ++i)
          {
               term = i * i * i * i;
               sum += term;
          }
          std::cout << "Sum:" << sum << std::endl;
     }

     return 0;
}
```

Exercise -1.0.10. The following program computes the sum

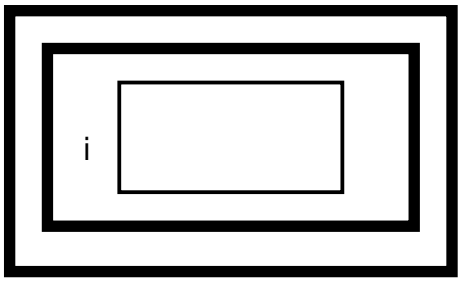
$$(1/1) + (1/1)(1/2) + (1/1)(1/2)(1/3) + (1/1)(1/2)(1/3)(1/4) + ...$$

(that's in mathematical notation and not C++ notation) where the number of terms is specified by the user. Verify by hand that it does not work correctly. Explain why and correct the program; there are several problems.

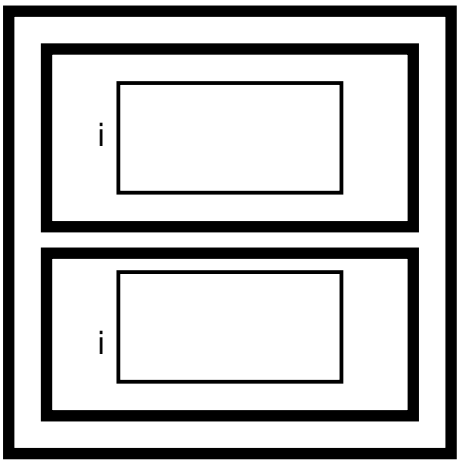
```
int n = 0, i = 0, j = 0, term = 0, sum = 0;

std::cin >> n;

for (i = 1; i <= n; ++i)
{
     // Compute i-th term, i.e. 1/1*...*1/i
     for (j = 0; j < i; ++j)
     {
          term *= 1.0 / j;
     }

     // Accumulate term into sum by addition
     sum += term;
}

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

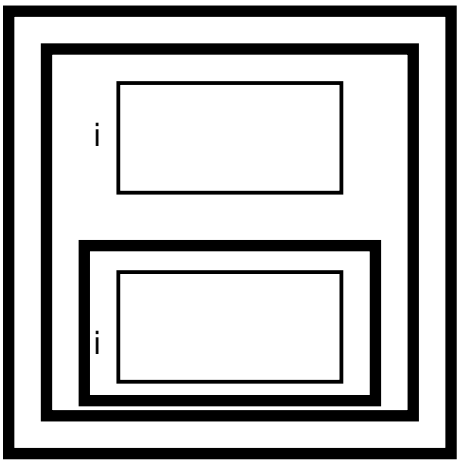
Summary The scope of a variable is where the variable can be accessed. This is from the declaration of the variable to the end of the block where it is declared.

```
...
{
    ...
    double i = 0.0;
    ...
}
...
```



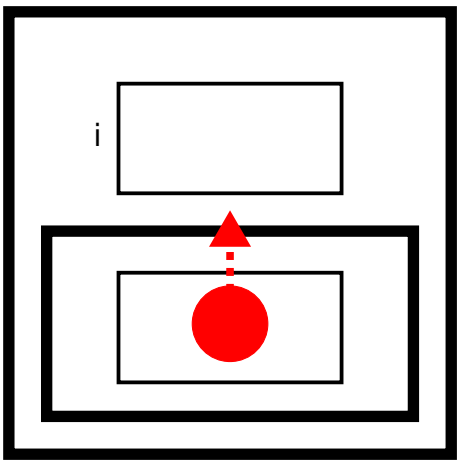
If a variable name is used in two declarations, then the two blocks where they are declared do not overlap or one is nested within the other.





If the program tries to access a variable and it's not found in the current block, then the program will look for the variable in the **nearest** outer block.

Program execution in this block and looks for i Program will keep looking in the next outer block until it's found or if not, we have an error



The `for`-loop and `while`-loop create a block and their bodies create a block within the loop's block.