08. `if` and `if-else`: Part 2

This is a continuation of “if and if-else: Part 1”.

Enough said. Move on!!!

`if-else` statements

Run this twice by entering 42 and then 24 for input:

```
int a = 0;
std::cout << "C++: What's your favorite number?"
          << std::endl << "User: ";
std::cin >> a;

if (a == 42)
  std::cout << "C++: Mine too!" <<std::endl;
if (a != 42)
  std::cout << "C++: boo!" << std::endl;
```

You want to either do one thing or another based on whether a boolean condition is true or false (respectively). The above does work ... *but* it is the **WRONG** way of doing it.

Now change the above to this:

```
int a = 0;
std::cout << "C++: What's your favorite number?"
          << std::endl << "User: ";
std::cin >> a;

if (a == 42)
    std::cout << "C++: Mine too!" << std::endl;
else
    std::cout << "C++: boo!" << std::endl;
```

You should **NEVER** have a program like the earlier one with two if's with opposite conditions next to each other:

```
if (a == 42)
  std::cout << "C++: Mine too!" <<std::endl;
if (a != 42)     // BAD!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  std::cout << "C++: boo!" << std::endl;
```

In other words if you can rewrite a bunch of ifs with if-else, then you should. Why? Because this avoids evaluating boolean expressions unnecessarily and hence speed up your program. (There are other reasons).

The **second** way (the if-else) is the **right** way:

```
if (a == 42)
    std::cout << "C++: Mine too!" << std::endl;
else
    std::cout << "C++: boo!" << std::endl;
```

The format of the if-else statement is

`if (`*`[bool expr]`*`)`

```
        [stmt1]
    else
        [stmt2]
```

If *[bool expr]* evaluates to `true`, *[stmt1]* executes. When that's completed, the else part is bypassed. If *[bool expr]* evaluates to `false` *[stmt2]* is executed.

Obviously you can't write an else statement without an if!!!

Exercise. Here's the first if-else warm up exercise. No sweat required to finish it. First try running this program so that you know what it does:

```
int numHeads = 0;
std::cout << "How many heads do you have? ";
std::cin >> numHeads;
if (numHeads == 2)
    std::cout << "Are you Zaphod?" << std::endl;
if (numHeads != 2)
    std::cout << "You are not Zaphod" << std::endl;
```

Next modify it to use if-else instead of two if's. Run the program and test it. Here are some tests:

```
How many heads do you have? 1
You are not Zaphod
```

And here's an execution of the same program:

```
How many heads do you have? 2
Are you Zaphod?
```

Exercise. Run the following program:

```
std::cout << "How much do you have? ";
double savings = 0.0;
std::cin >> savings;

if (savings > 100)
    std::cout << "Can I borrow a ten?" << std::endl;
if (savings <= 100)
    std::cout << "Life is tough" << std::endl;
```

Rewrite the program using if-else.

Exercise. Write a program that prompts the user for an integer and tells you if the input is even or odd. Here's an execution:

```
gimme an int: 42
even!!!
```

Here's another:

```
gimme an int: 35
odd!!!
```

Exercise. No skeleton code here. This is a slight modification of a previous program. Write a program that prompts the user for his/her height (in ft) and weight (in lbs) and prints “you will live more than 100 years!!!” if the product of the height and weight is at least 200.45. (This is not supported by any form of research.) Otherwise it prints “you will not live more than 100 years”. Here's an execution of the program:

```
6.1 180.180
you will live more than 100 years!!!
```

Here's another

```
3.4 23.4
you will not live more than 100 years
```

Exercise. Write a program that prompts the user for his employee code which is either 0 or 1. If the employee code is 0, his/her hourly rate is $123.45. If the employee code is 1, his/her hourly rate is $7.99. Prompt the user for the number of hours he/she worked this week. Finally print the total amount he/she made this week. Here's an informal description of the program that you should follow (obviously you have to translate this to C++!!!):

```
Declare constant HOURLY_RATE_0 and initialize it to 123.45.
Declare constant HOURLY_RATE_1 and initialize it to 7.99.

Declare variable employee_code.
Get employee_code from user.

Declare variable hourly_rate.
If employee_code is 0:
   set hourly_rate to HOURLY_RATE_0
Otherwise:
   set hourly_rate to HOURLY_RATE_1

Get number_of_hours from user.
Print total pay for this week.
```

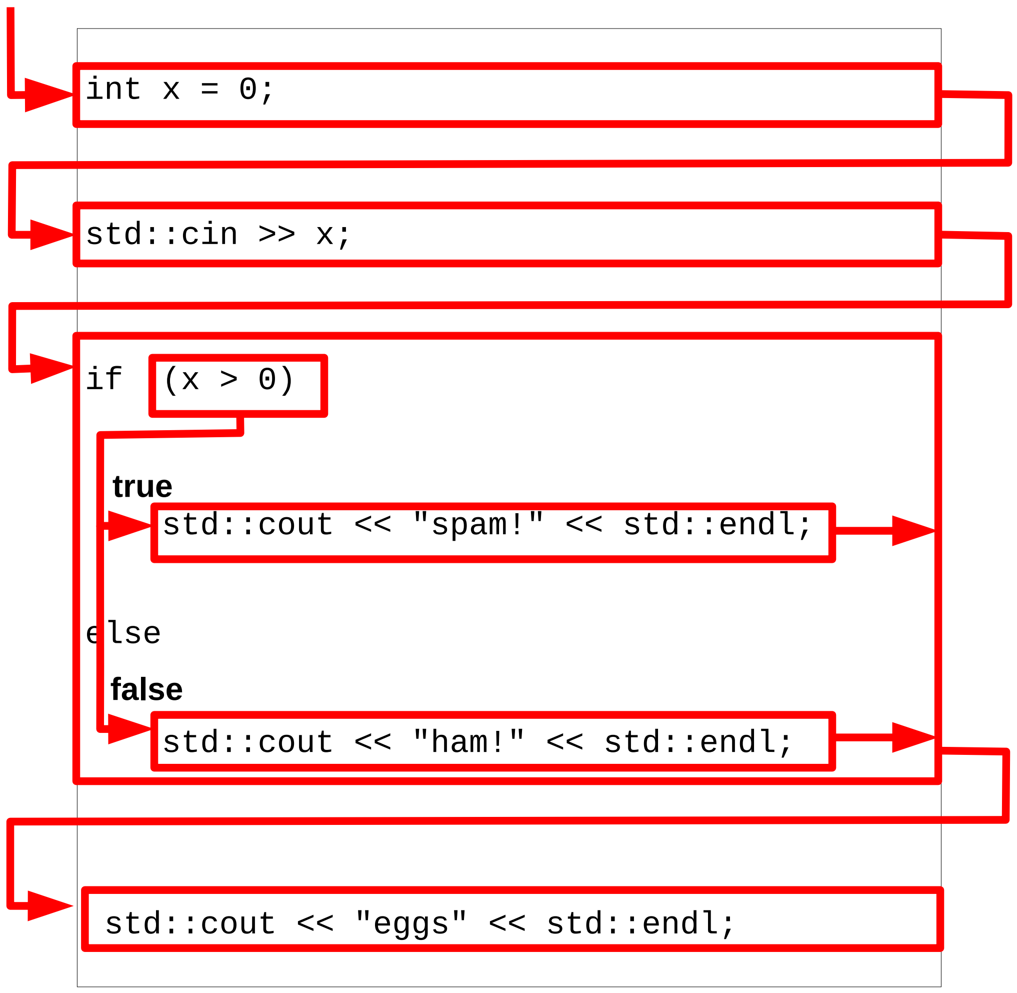
Mental picture: flow of execution for `if -else`

The following picture shows you the flow of execution of a simple program containing the if-else statement.

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

if (x > 0)
    std::cout << "spam!" << std::endl;
else
    std::cout << "ham!" << std::endl;

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



```
int x = 0;

std::cin >> x;

if  (x > 0)
        std::cout << "spam!" << std::endl;
else
        std::cout << "ham!" << std::endl;

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

Multiplication game: Part 2

Recall our multiplication game that tests if the user can multiply 97 and 94. Here's an execution of the program:

```
What is the product of 97 and 94? 1
```

Here's another:

```
What is the product of 97 and 94? 9118
You smart dawg!
```

Here's the pseudocode:

```
Declare integer variable guess
Print a prompt string to user
Prompt user for integer value for guess
If guess is 9118
      print congratulatory message
```

Now improve the program so that it does the following:

```
Declare integer variable guess
Print a prompt string to user
Prompt user for integer value for guess
If guess is 9118
      print congratulatory message
Otherwise
      print the correct answer
```

Here's an execution:

```
What is the product of 97 and 94? 1
Nope ... it's 9118.
```

Here's another:

```
What is the product of 97 and 94? 9118
You smart dawg!
```

Multiplication game: Part 3

We're going to sell our multiplication game. But no one is going to buy it if all it does is to ask the same question again and again!

Modify your program so that the program prompts the user for the product of two integers between 90 and 99. The integers should be randomly generated.

The pseudocode becomes:

```
Generate two random integers between 90 and 99
      and give the values to variables x and y
Print a prompt string to user to guess the
      product of x and y

Declare integer variable guess
Prompt user for integer value for guess

If guess is the product of x and y
      print congratulatory message
Otherwise
      print the product of x and y
```

Note the value of the product of x and y occurs twice. We can compute the product just once and store it in a variable:

```
Generate two random integers between 90 and 99
      and give the values to variables x and y
Store the product of x and y in variable answer
Print a prompt string to user to guess the
      product of x and y

Declare integer variable guess
Prompt user for integer value for guess

If guess is answer
      print congratulatory message
Otherwise
      print answer
```

To test your program you need to know the answer. So just get the program to print the answer temporarily before prompting the user. You can get rid of revealing the answer when you have fully tested your code.

Here's an execution of the program:

```
What is the product of 91 and 94? (answer: 8554) 1
Nope ... it's 8554
```

Here's another execution of the same program:

```
What is the product of 96 and 94? (answer: 9024) 9024
You smart dawg!
```

Of course after it works you get rid of the `(answer: ...)` output and the execution becomes::

```
What is the product of 91 and 94? 1
Nope ... it's 8554
```

```
What is the product of 96 and 94? 9024
You smart dawg!
```

if-else blocks

It should not be too surprising that you can execute blocks for if-else. You can have a block controlled by the if-part, or a block controlled by the else-part, or both.

```
std::cout << "Head Surgery Cost Planner\n"
          << "First 3: $100 each.\n"
          << "After that: $2 each.\n"
          << "Less than 3: extra 500 charge"
          << std::endl;

int extraHeads = 0;
int cost = 0;

std::cout << "How many extra heads do you want? ";
std::cin >> extraHeads;

if (extraHeads <= 3)
{
    cost = extraHeads * 100 + 500;
    std::cout << "Please go to Cheapskate Room."
              << std::endl;
}
else
{
    cost = 3 * 100 + (extraHeads – 3) * 2;
    std::cout << "Please go to Deluxe Room."
              << std::endl;
}
```

Exercise. Once you see code like the above you immediately ask yourself this question: How many constants do you see in the above program? (I can see four). Rewrite it with constants instead of hard-coding integer values.

Exercise. Write a program that computes the area of either a circle or a right-angle triangle. Recall that you can declare character variables. It turns out that you can compare character values too, using the == operator. The type for the character values is `char`. Here's a code skeleton for you:

```
std::cout << "Area Calculator!!!\n";
std::cout << "I can do circles or triangles!\n"
          << "Option (c or t): ";
char option;
std::cin >> option;
if (option == 'c')
{
    // Prompt the user for radius and display the
    // area of the circle with the given radius.
}
else // if it's not 'c', assume it is 't'
```

```
{
    // Prompt the user for the base and height of
    // the right-angled triangle and display the
    // area.
}
```

Coding style for `if` and `if-else`

ONE SPACE AFTER if:

BAD (but practiced by some)

```
if(x < 0)
    foo = 1;
```

GOOD

```
if (x < 0)
    foo = 1;
```

ONE-LINE if:

It's OK to write a one-line if statement if the body is a short statement and not a block and it improves readability.

OK BUT YUCKY:

```
x = 0;

if (x < 0)
    y = 0;
```

BETTER

```
x = 0;

if (x < 0) y = 0;
```

Maximum/minimum computation

Trace this (very very very important program by hand). Understand what it does. Finally rewrite it using the one-line if format.

```
int w = 1, x = 3, y = 0, z = 5;
int max;

max = w;
if (max < x)
    max = x;
if (max < y)
    max = y;
if (max < z)
    max = z;
```

Note that you can also use the bubblesort algorithm to compute the maximum value in w, x, y, z. However the above maximum computation algorithm does not require swapping values (the values in w, x, y, z are not changed) and therefore it uses less computation.

Exercise. Write a program that prompts the user for 6 integer values, prints the 6 values, and then prints the maximum of the 6 values.

Exercise. Refer to the previous exercise. What happens when you change all the < to the > operator? (This is also very very very important).

Coding style for blocks

ALIGN BLOCK DELIMITERS TO HEADER:

BAD (but used by some and is common in Java):

```
int main() {
    return 0;
}
```

and

```
if (x < 0) {
    foo = 42;
    bar = 43;
}
```

HORRIFYING:

```
if (x < 0)
    {
    foo = 42;
    bar = 43;
    }
```

ABOMINABLE:

```
if (x < 0)
    {
        foo = 42;
        bar = 43;
    }
```

GOOD:

```
int main()
{
    return 0;
}
```

and

```
if (x < 0)
{
    foo = 42;
    bar = 43;
}
```

The same applies to if-else:

BAD

```
if (x < 0) {
    foo = 42;
    bar = 43;
}
else {
    foo = 43;
    bar = 42;
}
```

GOOD

```
if (x < 0)
{
    foo = 42;
    bar = 43;
}
else
{
    foo = 43;
    bar = 42;
}
```

ALWAYS USE BLOCKS IF NOT ONE-LINERS:
C/C++ programmers tend to use blocks even when the block has only one statement except when using the one-line if style. This creates a unified look and prevents difficult-to-find bugs caused by “dangling else” (see later).

OK BUT NOT COMMON:

```
if (x < 0)
    foo = 42;
else
    foo = 24;
```

COMMON:

```
if (x < 0)
{
    foo = 42;
}
else
{
    foo = 24;
}
```

Boolean expressions again

We started with boolean expressions such as

```
x > 0
```

and went on to expressions such as

```
x > y
```

and then to

```
x * x + y * y != 0
```

and finally to

```
a * a + b * b < c * c + d * d
```

i.e.

`[expr1] [boolop] [expr2]`

where *`[expr1]`* and *`[expr2]`* are boolean expressions and *`[boolop]`* is a binary boolean operator accepting two numeric values. Let's recall a few more binary operators.

First of all let's not forget that we also have the unary ! operator. The following prints x if **it is not true that x is 0**.

```
int x = 0;
std::cin >> x;
if (!(x == 0))
    std::cout << x << std::endl;
```

Exercise. Rewrite the above program so that the ! operator is not used. Your resulting program therefore has to evaluate one boolean operator and not two.

Exercise. Simplify the boolean expression in the program so that the program behaves the same after your modification:

```
int x = 0;
std::cin >> x;
if (!(x <= 42) || !(x >= 42))
    std::cout << x << std::endl;
```

But note you should not simplify expressions (boolean or numeric) if by doing so it makes your program harder to read. Furthermore, modern-day compilers are actually smart enough to simplify it for you automatically!
Of course we also have the boolean operators && and `||`. So we can

have the following boolean expression:

```
(x < y) && (y < z)
```

Note that `(x < y)` and `(y < z)` are boolean expressions. So in general we can have

[boolexpr1] [boolop] [boolexpr2]

where *[boolexpr1]* and *[boolexpr2]* are boolean expressions and *[boolop]* is either && or `||`.

This should be a DIY review:

Exercise. Complete the truth table for &&:

true && true	is	_____
true && false	is	_____
false && true	is	_____
false && false	is	_____

Exercise. Complete the truth table for `||`:

true || true	is	_____
true || false	is	_____
false || true	is	_____
false || false	is	_____

You should check with your previous notes.

Exercise. You qualify for free head surgery if you have one head, less than three arms, more fingers on your left than you right hand, and finally either you are from Earth or the product of your highest level reached on Pacman and Galaxian is less than 6 (obviously you need an extra head in that case ...) Of course if you have not played Pacman, your high level is 0; likewise for Galaxian. Complete this program:

```
int numHeads = 0;
int numArms = 0;
int numLeftFingers = 0;
int numRightFingers = 0;
bool earthling = true;
int pacManLevel = 0;
int galaxianLevel = 0;

// prompt for values for all variables
// For boolean variable enter 1 for true and 0 for false


if (                                )
    std::cout << "You qualify for our free head surgery!"
        << std::endl;
else
    std::cout << "Try again at newheads.com" << std::endl;
```

You should create some test cases by hand and then test your code against your test cases.

Nested `if-else`

Now the plot thickens. Since we can put any statement inside an `if` or an `else`, and `if` and `if-else` are themselves statements, that means that we can nest if and if-else statements.

That's mind-boggling. Let's have an example.

Run this with input 1. Run it again with input 2. Repeat that with 3. Finally try it with -1.

```
int numHeads = 0;
std::cout << "How many heads do you have? " << std::endl;
std::cout << numHeads;

if (numHeads == 1)
{
    std::cout << "Let me introduce you to our surgeons."
              << std::endl;
}
else
{
    if (numHeads == 2)
    {
        std::cout << "Are you Zaphod?" << std::endl;
    }
    else
    {
        std::cout << "Hmmm ... neither 1 nor 2 ..."
                  << std::endl;
    }
}
```

Exercise. Draw a picture showing the flow of execution of the above code segment.

Here's another example. Run this with inputs 1, 2, 3, -1.

```
int numHeads = 0;
std::cout << "How many heads do you have? ");
std::cin >> numHeads;

if (numHeads < 1)
{
    std::cout << "Huh?" << std::endl;
}
else
{
    if (numHeads == 1)
    {
        std::cout << "Let me introduce you to our "
                  << "surgeons." << std::endl;
    }
    else
    {
       if (numHeads == 2)
       {
           std::cout << "Are you Zaphod?");
       }
       else
       {
           std::cout << "You have way too many."
                     << std::endl;
       }
    }
}
```

Exercise. What is the output when x has value 0.

```
if (x < 5)
{
    std::cout << "1" << std::endl;
}
else
{
    if (x == 0)
    {
        std::cout << "2" << std::endl;
    }
    else
    {
        if (x > 42)
        {
            std::cout << "3" << std::endl;
        }
    }
}
```

By the way look at this program:

```
if (score < 100)
{
    std::cout << "lousy gamer" << std::endl;
}
```

```
else
{
    if (100 <= score && score < 200)
    {
        std::cout << "not bad" << std::endl;
    }
    else
    {
        if (200 <= score)
        {
            std::cout << "great!" << std::endl;
        }
    }
}
```

Do you see why the above is the same as:

```
if (score < 100)
{
    std::cout << "lousy gamer" << std::endl;
}
else
{
    if (score < 200)
    {
        std::cout << "not bad" << std::endl;
    }
    else
    {
        if (200 <= score)
        {
            std::cout << "great!" << std::endl;
        }
    }
}
```

Exercise. Write a program that prompts the user for his/her age. If the age is less than 10, print “read the hobbit”, if it's between 11 and 15, print “read lord of the ring”, if it's 16, print “read hitchhiker's guide to the galaxy”, and otherwise, print “read chica chica boom boom”.

Exercise. Write a program that converts a numeric grade to a letter grade. The program prompts the user for a double, and if the double is less than 60, it prints 'F'; if it is at least 60 but less than 70, it prints 'D';if it is at least 70 but less than 80, it prints 'C';if it is at least 80 but less than 90, it prints 'B'; and if it's at least 90, it prints 'A'.

The dangling else problem

Here's a trick question (and it's another reason why C/C++/Java programmers use { } even when the block has only one statement). So ...

PAY ATTENTION

First figure out the output (without a compiler) when x has a value of 0.

```
if (x < 0)
    if (x == -2) std::cout << "1" << std::endl;
else
    std::cout << "2" << std::endl;
```

Now run the program.

Mamamia ... ?!?

This is called the "dangling else" problem.

What actually happens is that an `else` always follows the closest `if`. This means that the above code is better indented this way:

```
if (x < 0)
    if (x == -2) std::cout << "1" << std::endl;
    else
        std::cout << "2" << std::endl;
```

Of course the C++ compiler doesn't care about whitespaces. However this version of indentation makes it easier for human beings to follow the logic.

Now if what I intended was actually something like the previous version, then I would have to do this:

```
if (x < 0)
{
    if (x == -2) std::cout << "1" << std::endl;
}
else
    std::cout << "2" << std::endl;
```

In this case, the closest if-part that the else-part can attach to is the if-part on the first line since the else-part cannot "go inside" the block.

This is the reason why it's a good practice to ALWAYS use blocks even when the if-part or the else-part has only one line to execute. It prevents the misleading dangling else problem. So if I write this:

```
if (x < 0)
{
    if (x == -2)
    {
        std::cout << "1" << std::endl;
    }
}
else
{
    std::cout << "2" << std::endl;
}
```
there will be no confusion.

Coding style for nested `if-else`

NOT SO GOOD:

```
if (x < 5)
{
    foo = 42;
}
else
{
    if (x < 8)
    {
        foo = 43;
    }
    else
    {
        if (x == 9)
        {
            foo = 44;
        }
        else
        {
            foo = 45;
        }
    }
}
```

GOOD:

```
if (x < 5)
{
    foo = 42;
}
else if (x < 8)
{
    foo = 43;
}
else if (x == 9)
{
    foo = 44;
}
else
{
    foo = 45;
}
```

Exercise. Rewrite your previous program:

```
int numHeads = 0;
std::cout << "How many heads do you have? ");
std::cin >> numHeads;

if (numHeads < 1)
{
    std::cout << "Huh?" << std::endl;
}
else
{
    if (numHeads == 1)
    {
        std::cout << "Let me introduce you to our "
                  << "surgeons." << std::endl;
    }
    else
    {
        if (numHeads == 2)
        {
            std::cout << "Are you Zaphod?");
        }
        else
        {
            std::cout << "You have way too "many."
                      << std::endl;
        }
    }
}
```

using the above format.

Multiplication game: Part 4

Here comes one more improvement to our multiplication game. We're going to give the user a second chance. Furthermore we'll give a hint. We will tell him/her is his/her answer is too high or too low. If the user gets the correct answer the first time, we'll give 2 points. If the user gets it right the second time, we'll give 1 point. Otherwise we'll give zero. (What else can you give anyway?)

Here's the pseudocode:

```
Generate two random integers between 90 and 99
      and give the values to variables x and y
Store the product of x and y in variable answer
Print a prompt string to user to guess the
      product of x and y

Declare integer variable score and initialize it
      to 0
Declare integer variable guess
Prompt user for integer value for guess

If guess is answer
      Print congratulatory message
      score = 2
Otherwise if guess is less than the answer:
      Tell user the guess is too low
      Print a prompt string to user to guess
            the product of x and y
      If guess is answer
            print congratulatory message
            score = 1
      Otherwise
            print answer
Otherwise if guess is greater than the answer:
      Tell user the guess is too low
      Print a prompt string to user to guess
            the product of x and y
      If guess is answer
            print congratulatory message
            score = 1
      Otherwise
            print answer

Print user's score
```

Here are several executions to give you a feel for what the program should do:

Case: correct

```
What is the product of 91 and 94? (answer: 8554) 8554
You smart dawg!
Score: 2
```

Case: low-low

```
What is the product of 91 and 94? (answer: 8554) 1
Too low!
What is the product of 91 and 94? (answer: 8554) 1
Nope ... it's 8554
Score: 0
```

Case: low-correct

```
What is the product of 91 and 94? (answer: 8554) 1
Too low!
What is the product of 91 and 94? (answer: 8554) 8554
You smart dawg!
Score: 1
```

Case: low-high

```
What is the product of 91 and 94? (answer: 8554) 1
Too low!
What is the product of 91 and 94? (answer: 8554) 9000
Nope ... it's 8554
Score: 0
```

Case: high-low

```
What is the product of 91 and 94? (answer: 8554) 9000
Too high!
What is the product of 91 and 94? (answer: 8554) 1
Nope ... it's 8554
Score: 0
```

Case: high-correct

```
What is the product of 91 and 94? (answer: 8554) 9000
Too high!
What is the product of 91 and 94? (answer: 8554) 8554
You smart dawg!
Score: 1
```

Cleanup – refactoring

In writing software, even when you're done, you should look at your code to see if you can improve on it. You do not want to change the behavior of the code. Rather you want to reorganize the code to make it easier to read or to make it more efficient. Such an activity is called **refactoring**.

This is something that all problem-solving processes involve, not just software development. In particular, this occurs a lot in fundamental research as well. For instance mathematicians always rewrite their proofs to improve the readability. Sometimes mathematicians will even derive different proofs of the same result in order to see things in a totally different way. Anyway, let's get back to programming.

You realized that there is a lot of duplicate code. There is a lot of duplication when the program gives the user a second try:

```
Generate two random integers between 90 and 99
      and give the values to variables x and y
Store the product of x and y in variable answer
Print a prompt string to user to guess the
      product of x and y

Declare integer variable score and initialize it
      to 0
Declare integer variable guess
Prompt user for integer value for guess

If guess is answer
      Print congratulatory message
      score = 2
Otherwise if guess is less than the answer:
      Tell user the guess is too low
      Print a prompt string to user to guess
            the product of x and y
      If guess is answer
            print congratulatory message
            score = 1
      Otherwise
            print answer
Otherwise if guess is greater than the answer:
      Tell user the guess is too low
      Print a prompt string to user to guess
            the product of x and y
      If guess is answer
            print congratulatory message
            score = 1
      Otherwise
            print answer

Print user's score
```

Not everything is similar between the two chunks of code. But you can see that the similarities is around this spot:

```
If guess is answer
      Print congratulatory message
      score = 2
Otherwise if guess is less than the answer:
      Tell user the guess is too low
      Print a prompt string to user to guess
            the product of x and y
      If guess is answer
            print congratulatory message
            score = 1
      Otherwise
            print answer
Otherwise if guess is greater than the answer:
      Tell user the guess is too low
      Print a prompt string to user to guess
            the product of x and y
      If guess is answer
            print congratulatory message
            score = 1
      Otherwise
            print answer
```

We want to write one chunk of code and not two. To combine the chunks (they reside under two different boolean conditions right now) we need to have a common condition for both – and that's when the first guess is incorrect:

```
If guess is answer
      Print congratulatory message
      score = 2
Otherwise
      If guess is less than the answer:
            Tell user the guess is too low
      Otherwise
            Tell user the guess is too high
      Print a prompt string to user to guess
            the product of x and y
      If guess is answer
            print congratulatory message
            score = 1
      Otherwise
            print answer

Print user's score
```

Now using this pseudocode, clean up your program and see how many lines of code shorter your program gets.

Find computation

Suppose you're a bunch of integer values in variables, say

a, b, c, d

and you're interested in who has a particular value. If the target value is in a, you want to print 0. If it's in b, you want to print 1, etc. In other words, we think of a has having position 0 and if the target value is found in a, we think of the target at position 0. I will save the position in the variable `index`. The code would look like this:

```
if (target == a)
{
    index = 0;
}
else if (target == b)
{
    index = 1;
}
else if (target == c)
{
    index = 2;
}
else if (target == d)
{
    index = 3;
}
std::cout << index << '\n';
```

If the target is not found in a, b, c, d, we will set `index` to -1:

```
if (target == a)
{
    index = 0;
}
else if (target == b)
{
    index = 1;
}
else if (target == c)
{
    index = 2;
}
else if (target == d)
{
    index = 3;
}
else
{
    index = -1;
}
std::cout << index << '\n';
```

Exercise. Complete the program by prompting the user for a, b, c, d and target and print `index`.

Exercise. Do the same as above but prompt the user for 5 integer values a, b, c, d, e.

Exercise. Note that the above finds the position of a value by searching from the first variable to the last (a to d). What if you search backwards? For instance if the target value is found in both a and d, your program should print 3 and not 0.

Make sure you understand and remember this algorithm.

Count computation

Suppose you have 4 integer variables (say a, b, c, d) and you want to count the number of times a value (say in `target`) occurs in these integer variables. The pseudocode would look like this:

```
count = 0
if target == a:
    increment count by 1
if target == b:
    increment count by 1
if target == c:
    increment count by 1
if target == d:
    increment count by 1
print count
```

Exercise. Write a program that prompts the user for 4 integers, stores them in variables, a, b, c, d. The program then prompts the user for an integer value and stores it in `target`. Convert the above pseudocode into C++ and print `count`.

Make sure you understand and remember this algorithm.

Typecast between `bool` and `ints`

And don't forget that, when needed, C++ will automatically type cast boolean values to integers by casting the value true to 1 and false to 0. C++ will also can also type cast numeric values (ints, doubles, floats) to boolean values by casting nonzero values to true and zero to false:

`bool` →	`int`
`true`	`1`
`false`	`0`

numeric (int/double/float) →	`bool`
nonzero (example: 1, 2.3)	`true`
zero (example: 0, 0.0)	`false`

Try this:

```
if (1)
    std::cout << "i'm the queen of england"
              << std::endl;
```

Change 1 to 4 and run the program again.

(The above is not a random example. It's important because we're going into loops very soon.)

Exercise. What is the boolean value of 1 && 2? What about 1 && 0? Do it “by hand”. Next verify your computation by running this:

```
std::cout << (1 && 2) << std::endl;
std::cout << (1 && 0) << std::endl;
```

I also warn you again that type casting does not change the value of a variable:

```
int x = 42;
std::cout << (x && 1) << std::endl;
std::cout << x << std::endl;
```

(The value of `x` is type cast to `true`. `x` is still 42.)

This is really the same as calling the `sqrt()` (square root) function with a variable. The value of the variable is not changed.

```
double x = 2.0;
std::cout << x << std::endl;
std::cout << sqrt(x) << std::endl;
std::cout << x << std::endl;
```

Assignment based on conditions

Here's a trick to simplify an assignment via an `if-else` statement. Suppose you have this program:

```
bool newMember = true;
double loanRate = 0.0;

if (newMember)
    loanRate = 0.01;
else
    loanRate = 0.20;

// Yeah that's what credit cards companies do ...
```

You can rewrite this as:

```
bool newMember = true;
double loanRate = 0.01;
if (!newMember)
    loanRate = 0.20;
```

Yet another way to do this is:

```
bool newMember = true;
double loanRate = newMember * 0.01
                  + (!newMember) * 0.20;
```

There is however a **problem** with this technique and some disapprove of it. That is, you're using a boolean value (true or false) as a numeric value (1 or 0). Some people believes that automatic type casting like this is dangerous and misleading and that **types should be kept as separate as possible**.

To make such people happy (example: your tech leader / manager might be one) and also not to mislead the reader, you should always **explicitly type cast**. For instance above should be written:

```
bool newMember = true;
double loanRate = int(newMember) * 0.01
                  + int(!newMember) * 0.20;
```

Exercise. Write a program that gives the variable `heartAttackRate` 0.01, 0.02, 0.03, and 0.94 depending on whether `numHeads` is 1, 2, 3 or at least 4 (respectively). Complete this:

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

double heartAttackRate = ___________________________
std::cout << heartAttackRate << std::endl;
```

The ternary operator

So far our operators accept one value, for example:

```
–y
!b
```

or two values, for example:

```
x + y
b1 && b2
```

There is a curious operator that accepts ***three***. Try this program:

```
int x = 5;
std::cout << (x == 5 ? 42 : 0) << std::endl;
```

Run this with different values for variable x.

Get the point of the `(?:)` operator yet? The format looks like this:

(*[bool expr]* ? *[expr1]* : *[expr2]*)

Basically, if the *`[bool expr]`* (a boolean expression) evaluates to `true`, C++ will evaluate *`[expr1]`* and give you its value; otherwise it gives you the value from *`[expr2]`*.

Here's another example. First try to guess what the program does. Next run it to verify.

```
int x = 0, y = 1, z = 2;

int a = (x + y < z ? x : y + z);
std::cout << a << std::endl;
```

Do you see that it does the same thing as this:

```
int x = 0, y = 1, z = 2;

int a = 0;
if (x + y < z)
{
   a = x;
}
else
{
   a = y + z;
}
std::cout << a << std::endl;
```

Here's another example. Try this die game:

```
int die_roll = 1 + rand() % 6 ;
std::cout << "value of die roll: " << die_roll
          << '\n';
```

```
std::cout << "You "
          << (die_roll == 6 ? "win!" : "lose")
          << '\n';
```

Get it?

Exercise. Write a dice game. Roll two dice. If the dice both give you the same number, print “win”. Otherwise your program prints “lose”. Your program should also print the values of the dice roll. Use the ternary operator.

Exercise. Write a dice-coin game. First, you roll the dice. If the dice values are the same you gain $1.50 and the game ends. Otherwise you pay $1. However you have a second chance: you toss a coin. If you get a head (use random integer value of 0) you gain $1, otherwise (i.e., for a tail), you gain $0.50. Your program prints all the dice values and the result of the coin toss and finally your gain (which can be positive or negative). Run this program 10 times. What is you average gain? Use the ternary operator whenever possible.

Exercise. Rewrite the following code fragment so that it’s cleaner

```
int x, y;
std::cin >> x >> y;
if (x >= 42)
{
    std::cout << x << ' ';
    if (y < 10)
    {
        std::cout << y << ' ';
    }
    else
    {
        std::cout << '-' << ' ';
    }
}
else
{
    std::cout << '*' << ' ';
    if (y < 10)
    {
        std::cout << y << ' ';
    }
    else
    {
        std::cout << '-' << ' ';
    }
}
```

Summary

The if and if-else statement looks like these:

```
      if ([bool expr])
            [stmt]

      if ([bool expr])
            [stmt1]
      else
            [stmt2]
```

where *[bool expr]* is a boolean expression and *[stmt]*, *[stmt1]*, and *[stmt2]* are either statements or blocks of statements.

The else statement will always be associated with the closest if statement. In other words

```
if (x < 0)
    if (x == -2) std::cout << "1" << std::endl;
else
    std::cout << "2" << std::endl;
```

is better written this way:

```
if (x < 0)
    if (x == -2)
        std::cout << "1" << std::endl;
    else
        std::cout << "2" << std::endl;
```

The `rand()` function returns a “random” integer between 0 and `RAND_MAX`. Number. The `srand()` function “seeds” the `rand()` function. You might need to “`#include <cstdlib>`” to use the `srand()` and `rand()` functions. One way to seed rand() is to call

```
      srand((unsigned) time(0))
```

in which case you need to “`#include <ctime>`”.

```
      double(rand()) / RAND_MAX
```

will generate “random” doubles between 0 and 1. To generate a random integer between a and b (two integers with a <= b), you can use

```
      int(double(rand()) / RAND_MAX * (b – a)) + a
```

or

```
      rand() % (b – a + 1) + a
```

(The second method is not as general nor as random as the first. For instance you cannot use it to generate random number between 0 and 1000000 in MS .NET STUDIO).

The following code segment swaps the values in a and b (and `t` has the value of a):

```
t = a;      a = b;      b = t;
```

Using the "swapping trick", you can always ensure that a pair of variables are in ascending (or descending order):

```
if (a > b)
{
    t = a;
    a = b;
    b = t;
}
```

Given for instance four variables a, b, c, and d, you can apply the above to the pairs a,b and b,c and c,d to ensure that the largest value among a, b, c, d is in d:

```
if (a > b) { swap values of a and b }
if (b > c) { swap values of b and c }
if (c > d) { swap values of c and d }
```

This is called one pass of the sorting process. You can then apply the process to a, b, c for a second pass, and then to a, b for a third pass. This will sort the values of a, b, c, d into ascending order.

Just as there is automatic type casting from int to double, there is also automatic type type casting between bool and int or double:

bool →	int
true	1
false	0

numeric (int/double/float) →	bool
nonzero (example: 1, 2.3)	true
zero (example: 0, 0.0)	false

For instance the following is a valid C++ statement:

```
if (1)
    std::cout << "that's all folks";
```

The ternary operator looks like this:

`(`*`[bool expr]`* `?` *`[expr1]`* `:` *`[expr2]`*`)`

This whole expression evaluates to *`[expr1]`* if *`[bool expr]`* is `true`; otherwise it evaluates to *`[expr2]`*.

Repetition code and breaking deeply nested code and deep expressions

(There are some solutions are the end of this pdf.)

Here's one way to compute sum from 1 to 5 is

```
int s = 1 + 2 + 3 + 4 + 5;
```

Another way to do this is to do this:

```
int s = 0;

s = s + 1;

s = s + 2;

s = s + 3;

s = s + 4;

s = s + 5;
```

And to make all repetition code exactly the same, you do this:

```
int i = 1;
int s = 0;

s = s + i;
i = i + 1;

s = s + i;
i = i + 1;

s = s + i;
i = i + 1;

s = s + i;
i = i + 1;

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

You have already seen this before.

The reason why you want to think/write your code using repeating code is because later we can use loop controls to tell C++ to repeat a block of code by giving the loop structure just one block of the code.

Exercise. Compute the sum of squares from 1 to 25, i.e., s = 1*1 + 2*2 + 3*3 + 4*4 + 5*5 using the same idea as above.

Exercise. Get an integer n from the user and then compute the sum of the rightmost 4 digits of n. Do this with repetition code.

Maximum/minimum computation

Recall I have already talked about the max/min computation. There are two ways of doing it.

Here's the max computation that you have already seen:

```
int w = 1, x = 3, y = 0, z = 5;
int max;

max = w;
if (max < x)
    max = x;
if (max < y)
    max = y;
if (max < z)
    max = z;
```

Notice that other than the first statement, the rest of the code is more or less a repetition of the same kind of code:

```
max = w;

if (max < [variable])
    max = [variable];
```

There's another **TERRIBLE** way to do the above:

```
int w = 1, x = 3, y = 0, z = 5;

if (w >= x && w >= y && w >= z)
    max = w;
else
{
    if (x >= w && x >= y && x >= z)
        max = x;
    else
    {
        if (y >= w && y >= x && y >= z)
            max = y;
        else
            max = z;
    }
}
```

Of course you can try to make it "neater":

```
int w = 1, x = 3, y = 0, z = 5;

if (w >= x && w >= y && w >= z)
    max = w;
else if (x >= w && x >= y && x >= z)
    max = x;
else if (y >= w && y >= x && y >= z)
    max = y;
else
    max = z;
```

But you are computing way too many boolean conditions. If z is the maximum value, how many boolean operators are you evaluating? Compare this to the first max computation.

Secondly, and this is probably even more important:

Exercise. Suppose you need to add another variable to your w, x, y, z for max computation:

```
int w = 1, x = 3, y = 0, z = 5, a = 7;
int max;

// ... max computation ...
```

Complete this using the first method and then using the second method. Do I need to explain anymore that the first method is way better?

Part of the problem is that the boolean expression in

```
if (w >= x && w >= y && w >= z)
    max = w;
```

depends on a huge number of variables.

The earlier better solution is to consider one variable at a time.

In general if the boolean is made up of expressions which **do not depend on each other**, it's usually best to process them **separately**.

Count computation

Here's the pseudocode for the count computation again:

```
count = 0
if target == a:
    increment count by 1
if target == b:
    increment count by 1
if target == c:
    increment count by 1
if target == d:
    increment count by 1
print count
```

Again, you see repetition of code.

Another way to do the above, a **TERRIBLE** way, is to do this:

```
count = 0
if target != a && target != b && target != c && target != d
    count = 0
else if target == a && target != b && target != c && target != d
    count 1
else if target != a && target == b && target != c && target != d
    count 1
else if target != a && target != b && target == c && target != d
    count 1
else if target != a && target != b && target == c && target != d
    count 1
else if target != a && target != b && target != c && target == d
    count 1
else if target == a && target == b && target != c && target != d
    count 2
….
```

I don't even want to complete this!!! This is clearly a really TERRIBLE algorithm. There is a total of 2^4 = 16 boolean expressions. (Why?) YIKES!

And furthermore, if I add another variable e to the above, I would have to change a lot of code, as well as adding some more code.

Exercise. Get integer `x`, `y` from the user and print `x`, `y`, except that if `x` is negative you print `0` instead of `x` and if `y` is negative you print `0` instead of `y`. Here's a test case

```
2 3
2 3
```

Here's another test case

```
-1 3
0 3
```

and another test:

```
2 -2
2 0
```

and finally

```
-5 -2
0 0
```

You can write your program with boolean conditions where each boolean condition involves **both** x and y. But instead of that, write your program so that you have boolean expressions where each boolean expression involves only **one** variable.

Multiplication game: Part 4

Here's the pseudocode:

```
Generate two random integers between 90 and 99
      and give the values to variables x and y
Store the product of x and y in variable answer
Print a prompt string to user to guess the
      product of x and y

Declare integer variable score and initialize it
      to 0
Declare integer variable guess
Prompt user for integer value for guess

If guess is answer
      Print congratulatory message
      score = 2
Otherwise if guess is less than the answer:
      Tell user the guess is too low
      Print a prompt string to user to guess
            the product of x and y
      If guess is answer
            print congratulatory message
            score = 1
      Otherwise
            print answer
Otherwise if guess is greater than the answer:
      Tell user the guess is too low
      Print a prompt string to user to guess
            the product of x and y
      If guess is answer
            print congratulatory message
            score = 1
      Otherwise
            print answer

Print user's score
```

Sometimes, you have deeply nested code. What if the above, where you give the user at most 2 chances to answer a question, if changed to allowing the user to have **3** chances or even **4** chances? If you use the above method, you would have a very deeply nested code structure.

Another way to allow the user 3 chances is to avoid deeply nested code as follows. First, since you do not want to nest the 3 cases, you want something like this:

```
ask user for an answer
ask user for an answer
ask user for an answer
```

But of course you want the second “ask user for an answer” to know if the first “ask user for an answer” gave rise to a correct answer or not.

```
ask user for an answer
if first answer is not correct:
    ask user for an answer
ask user for an answer
```

Likewise the third “ask user for an answer” also need to know if a correct answer was given. So instead of “first answer is not correct”, we probably want ““answer is not correct” that can be used or the second and third “answer user for an answer”.

```
ask user for an answer
if answer is not correct
    ask user for an answer
if answer is not correct
    ask user for an answer
```

Let’s make “answer is not correct” into a boolean variable:

```
bool answer_is_correct

ask user for an answer
if answer from user is correct,
    answer_is_correct = true

if !answer_is_correct:
    ask user for an answer
    if answer from user is correct,
        answer_is_correct = true

if !answer_is_correct:
    ask user for an answer
    if answer from user is correct,
        answer_is_correct = true
```

You want to think of `answer_is_correct` as a **communication device** from the first “answer user for an answer” with the second and third, and also from the second to the third. For this example, the repeated blocks of code are **not completely independent**. Other than the first block, subsequent blocks need to know if the user has given a correct answer.

Get it?

Exercise. Write the multiplication game that allows the user to have at most 3 tries.

Exercise. Get an integer n from the user and then compute the sum of the rightmost 4 digits of n. Once a digit 0 is added, you should stop the summation. Do this with repetition code. Here’s a test case

```
1534315
13
```

Here's another:

```
1534015
6
```

(after adding `5` and `1`, on seeing a `0`, the summation stops. Therefore the sum is `6`) and another

```
1534205
5
```

Solutions

Exercise. Compute the sum of squares from 1 to 25, i.e., s = 1*1 + 2*2 + 3*3 + 4*4 + 5*5 using the same idea as above.
Solution.

```
#include <iostream>

int main()
{
    int s = 0;
    int i = 1;

    s = s + i;
    i = i + 1;

    s = s + i;
    i = i + 1;

    s = s + i;
    i = i + 1;

    s = s + i;
    i = i + 1;

    s = s + i;
    i = i + 1;

    std::cout << s << '\n';

    return 0;
}
```

Exercise. Get an integer n from the user and then compute the sum of the leftmost 4 digits of n. Do this with repetition code.
Solution.

```
#include <iostream>

int main()
{
    int n;
    std::cin >> n;

    int s = 0;

    s = s + n % 10;
    n = n / 10;

    s = s + n % 10;
    n = n / 10;

    s = s + n % 10;
    n = n / 10;

    s = s + n % 10;
    n = n / 10;

    std::cout << s << '\n';

    return 0;
}
```

Exercise. Get integer `x`, `y` from the user and print `x`, `y`, except that if `x` is negative you print `0` instead of `x` and if `y` is negative you print `0` instead of `y`. Here's a test case

```
2 3
2 3
```

Here's another test case

```
-1 3
0 3
```

and another test:

```
2 -2
2 0
```

and finally

```
-5 -2
0 0
```

You can write your program with boolean conditions where each boolean condition involves both `x` and `y`. But instead of that, write your program so that you have boolean expressions where each boolean expression involves only one variable.

Solution.

```
#include <iostream>

int main()
{
    int x, y;
    std::cin >> x >> y;

    if (x < 0)
    {
        x = 0;
    }
    std::cout << x << ' ';

    if (y < 0)
    {
        y = 0;
    }
    std::cout << y << ' ';

    return 0;
}
```

Exercise. Get an integer n from the user and then compute the sum of the rightmost 4 digits of n. Once a digit 0 is added, you should stop the summation. Do this with repetition code. Here's a test case

```
1534315
13
```

Here's another:

```
1534015
6
```

(after adding `5` and `1`, on seeing a `0`, the summation stops. Therefore the sum is 6) and another

```
1534205
5
```

Here's the idea. First here's the code to sum the 4 rightmost digits:

```
#include <iostream>

int main()
{
    int n;
    std::cin >> n;

    int s = 0;

    s = s + n % 10;
    n = n / 10;

    s = s + n % 10;
    n = n / 10;

    s = s + n % 10;
    n = n / 10;

    s = s + n % 10;
    n = n / 10;

    std::cout << s << '\n';

    return 0;
}
```

Each block needs to know if a zero was encountered. So …

```
#include <iostream>

int main()
{
    int n;
    std::cin >> n;

    int s = 0;
    bool zero_not_found = true;

    if (zero_not_found)
    {
        s = s + n % 10;
        n = n / 10;
    }

    if (zero_not_found)
    {
        s = s + n % 10;
        n = n / 10;
    }

    if (zero_not_found)
    {
        s = s + n % 10;
        n = n / 10;
    }

    if (zero_not_found)
    {
        s = s + n % 10;
        n = n / 10;
    }

    std::cout << s << '\n';

    return 0;
}
```

Get it? Think about what will happen if you don't use this communication device `zero_not_found`. You will most likely have a deeply nested if-else structure in your code:

```
    if (n % 10 != 0)             // 1st digit != 0
    {
        s = s + n % 10;
        n = n / 10;

        if (n % 10 != 0)         // 2nd digit != 0
        {
            s = s + n % 10;
            n = n / 10;

            if (n % 10 != 0)     // 3rd digit != 0
            {
                s = s + n % 10;
                n = n / 10;

                if (n % 10 != 0) // 4th digit != 0
                {
                    s = s + n % 10;
                    n = n / 10;
                }
            }
        }
    }
```

Later when I talk about loops and all the repeating blocks of code are placed in a loop structure, you'll see that your can control a loop to make it stop "prematurely".

Exercise

You have more than enough exercises in these notes already. Make sure you go over all of them on your own.

Some Mental Math (DIY)

This section has nothing to do with programming. It's just for fun.

Suppose I told you that it's possible to multiply 98 and 97 very quickly without paper and pencil or calculators or C++. Would you believe me? It does require some practice, but there is a way to multiply two values a little less than 100 very quickly.

Let me show you how with the 98 x 97 example. You subtract the numbers from 100. So 100 – 98 = 2 and 100 – 97 = 3. Write this down on a piece of paper:

98	97
2	3

Now look at 98 and 3. Compute 98 – 3. This is 95. You will need this number. Note that 97 – 2 is also 95. So there are two different ways of getting 95 with the two diagonals. Write this down:

98	97	**95**
2	3	

Now look at the 2 and 3. Multiply them together. You get 6. Write this down:

98	97	95**06**
2	3	

That's it. Here's another example: 96 x 95. First you have:

96	95
4	5

Next you get

96	95	**91**
4	5	

and finally

96	95	91**20**
4	5	

The answer is 9120. With some practice you will be able to do the computation in your head. (Using algebra, you can actually prove that the method does work.) So now you can show it off to your math teacher.