Control with Functions.

Starting with Something Trivial.

Suppose I wanted to see a list of all the numbers, starting with 1, then 2, then 3, and so on. Never mind about why I'd want something like that: perhaps I sometimes forget what comes after 17, who knows. It is just a beginning example.

How do you print all the numbers starting with 1? One step at a time: first print 1, then print all the numbers starting with 2.

How do you print all the numbers starting with 2? One step at a time: first print 2, then print all the numbers starting with 3.

How do you print all the numbers starting with 3? One step at a time: first print 3, then print all the numbers starting with 4.

The answer is always the same. In general, to print all the numbers starting with N, first print N, then print all the numbers staring with N+1. It is easily written as a C++ function

```
void print_all_the_numbers_starting_with(const int N)
{ print(N);
  new_line();
  print_all_the_numbers_starting_with(N+1); }

void main()
{ print_all_the_numbers_starting_with(1); }
```

It would be an annoying program to run. It would just print out a sequence of numbers as fast as it could, never stopping. But that is exactly what it was supposed to do. The example problem was printing out all the numbers starting with 1, and that is an endless task. A program that isn't endless would be wrong.

Even this little program could be made partly useful. When it prints the value of N, it could just as easily print a lot of other things too. I could make it print N squared and N cubed perhaps:

```
void print_all_the_numbers_starting_with(const int N)
{ print("N=");
  print(N);
  print(" squared=");
  print(N*N);
  print(" cubed=");
  print(N*N*N);
  new_line();
  print_all_the_numbers_starting_with(N+1); }
```

It would produce a table looking like this:

```
N=1 squared=1 cubed=1
N=2 squared=4 cubed=8
N=3 squared=9 cubed=27
N=4 squared=16 cubed=64
N=5 squared=25 cubed=125
```

The table would of course go on for ever, but if you stopped the program quickly enough, you would at least be able to use it to look up squares and cubes if ever you wanted to know them.

At this point, it is worth observing that "print_all_the_numbers_starting_with" isn't a very good name for the function any more. It is a frequent cause of great confusion in programming, when one function is enhanced to perform a different task, but the programmer forgets to change its name. What would you think of a function called "square_root" that actually performs centigrade to fahrenheit conversions?

A List of Limited Length

Now suppose that I don't want to see a list of all the numbers starting from 1, but just the first few. I want to see all the numbers between 1 and 20. How do you print all the numbers between 1 and 20? One at a time. First print the first one, then print all the rest.

How do you print all the numbers between 1 and 20? First print 1, then print all the numbers between 2 and 20.

How do you print all the numbers between 2 and 20? First print 2, then print all the numbers between 3 and 20.

How do you print all the numbers between 3 and 20? First print 3, then print all the numbers between 4 and 20.

and so on and so on. Generalising it, the way to print all the numbers between A and B is to first print A, the print all the numbers between A+1 and B. Just for a second, pretend you haven't seen anything wrong with this idea, and see how easy it is to code as a C++ function:

```
void print_all_numbers_between(const int A, const int B)
{ print(A);
  new_line();
  print_all_numbers_between(A+1, B); }
```

Of course, the problem is that the logic is incorrect. The generalisation "*the way to print all the numbers between A and B is to first print A, the print all the numbers between A+1 and B*" is not always correct. The way to print all the numbers from 21 to 20 is to do nothing at all, because there are no numbers left in that range. The correct plan needs to say that if A>B, nothing should be done.

There are two equally good ways of writing this in C++:

```
void print_all_numbers_between(const int A, const int B)
{ if (A > B)
  { }
  else
  { print(A);
    new_line();
    print_all_numbers_between(A+1, B); } }
```

or

```
void print_all_numbers_between(const int A, const int B)
{ if (A <= B)
  { print(A);
    new_line();
    print_all_numbers_between(A+1, B); } }
```

And a third formulation is equally good: The way to print all the numbers between A and B is to first print A. Then, only if there are more numbers to print, because B is bigger than A, print all the remaining numbers between A+1 and B.

```
void print_all_numbers_between(const int A, const int B)
{ print(A);
  new_line();
  if (B > A)
    print_all_numbers_between(A+1, B); }
```

The only difference is if somebody uses the function incorrectly. If somebody, perhaps in `main()`, specifically says `print_all_numbers_between(37, 10);`, what is supposed to happen? Should it print nothing at all, or should it print the numbers backwards from 37 to 10, or should it just print 37, or should it tell the programmer through an error message that he is an idiot?

Of course, just as before, the function isn't limited to just printing the value of A, it could do a lot of other things too. Perhaps I would like to have a table showing conversions between centigrade and fahrenheit, limited to a reasonable range. Let's say for temperatures between A and B, that way there will be less to change.

```
void print_temperature_table_between(const int A, const int B)
{ print(A);
  print(" centigrade is ");
  print(A*9.0/5.0+32.0);
  print(" fahrenheit, ");
  print(A);
  print(" fahrenheit is ");
  print((A-32)*5.0/9.0);
  print(" centigrade\n");
  if (B > A)
    print_temperature_table_between(A+1, B); }

void main()
{ print_temperature_table_between(0, 100); }
```

Another perfectly reasonable formulation is: How do you print all the numbers between A and B? First you print all the numbers between A and B-1 if they are any, and after that you print B.

```
void print_all_numbers_between(const int A, const int B)
{ if (B > A)
    print_all_numbers_between(A, B-1);
  print(B);
  new_line(); }
```

This works just as well, and there are more exotic ideas too. If you've got a lot of things to do, it can be quite discouraging to think of doing just one thing and then you've got all the rest to do. Instead, if you've got a hundred things to do, you might do the first fifty, then do the remaining fifty.

To print all the numbers between 1 and 100, first print all the numbers between 1 and 50, then print all the numbers between 51 and 100.

Generalising, to print all the numbers between A and B, first print all the numbers between A and (A+B)/2, then print all the numbers between (A+B)/2+1 and B. Except of course that if A and B are the same, all you do is print A.

```
void print_all_numbers_between(const int A, const int B)
{ if (A == B)
  { print(A);
    new_line(); }
  else
  { const int middle = (A+B)/2;
    print_all_numbers_between(A, middle);
    print_all_numbers_between(middle+1, B); } }
```

That version is quite alarming, and looking at it can cause panic. But there is no reason for alarm: you hardly ever have to understand anything like that. Programming goes in the other direction - you have the plan of action in English (or whatever you prefer), and produce the C++ function from that. The description above, "*to print all the numbers between A and B, first print all the numbers between A and (A+B)/2, then print all the numbers between (A+B)/2+1 and B. Except of course that if A and B are the same, all you do is print A*", is the logical idea. Given that, producing the C++ version is a simple matter of translation.

Working Out what a Function Does.

Occasionally, you do have to work out some understanding of a function written by somebody else. There are some simple techniques that are really helpful. Here's an example of a mystery function

```
void function(const int X)
{ print(X);
  if (X > 0)
    function(X-1);
  print(X); }
```

What does it do? It might be hard to work out all at once, but there is one very simple case. If X is zero, you've got nothing even to think about. If X is zero, the whole function is exactly the same as `{ print(0); print(0); }`. That doesn't go very far, but it is at least one solid fact that was easy to find.

Remember it. Nothing can change it. We know that `function(0)` would print out "00". There is no question of that. No matter what the circumstances, `function(0)` just prints "00".

Does that really help? Yes. We know exactly what `function(0)` does, and that means that we know what `function(X-1)` would do if X is 1, it's just the same thing. So what does `function(1)` do? First it prints 1, then because 1>0 is true it prints 00, then finally it prints 1 again. `function(1)` just prints "1001".

Working out the one simple mindless case makes another case become simple. From knowing what happens if X is 0, we can easily see what happens if X is 1. `function(1)` will always print 1001.

And now, with no thought at all, we know what `function(2)` does. It prints 2, then because 2>0 it prints 1001, the finally it prints 2 again. `function(2)` just prints "210012".

From that, you easily work out what `function(3)` does, and it is clear that the pattern never changes. You know exactly what the function will do for any given value of X. Just from working out the one simple obvious fact, and repeatedly using what you already know to deduce one more fact. This is just the same as induction in mathematics.

Here's another one.

```
    void potato(const int X)
    { const int A=X/10, B=X%10;
      if (A!=0)
        potato(A);
      print(B);
      print("*"); }
```

It looks tricky, but the same technique will solve the problem.

If the part that says "`if (A!=0) potato(A)`" wasn't there, the function would be trivial. It would turn into

```
    void easy_potato(const int X)
    { const int A=X/10, B=X%10;
      print(B);
      print("*"); }
```

(In case you had forgotten, X%10 always comes out as the last digit of X: 1234 divided by 10 is 123 remainder 4, so 1234%10 is 4)

So this reduced function would just print the last digit of X followed by a star, and that would be that.

Without the "`if (A!=0) potato(A)`", we know exactly what the function would do. And if A happens to be zero, `A!=0` is false, so "`if (A!=0) potato(A)`" does absolutely nothing at all. Exactly as if it wasn't there.

If we could arrange for A to be zero, the potato function would be solved. And as A is set to `X/10`, any time X is less than 10, A will be zero.

Given nay number less than 10, potato just prints the last (and only) digit of that number, followed by a star.

With that knowledge, the "`if (A!=0) potato(A)`" part would not be at all frightening if A had any value between 1 and 9. It would just print that number followed by a star. A will be between 1 and 9 whenever X is between 10 and 99. Pick one random example: 57.

Potato(57) calculates A=5 and B=7, and because 5>0 it then performs three steps:

`potato(5)` then `print(7)` then `print("*")`

We already worked out exactly what potato(5) does, it prints 5 followed by a star. So now we know what potato(57) does: it prints 5 followed by a star followed by 7 followed by a star. The same reasoning applies to any number between 10 and 99: potato just prints each of the digits of the number followed by a star.

Now we know potato's exact behaviour for any number 0 to 99. With that, we can work out what it does with any number 100 to 999. Take 573 for example.

Potato(579) calculates A=57 and B=9, and because 57>0 it then performs three steps:
`potato(57)` then `print(9)` then `print(“*”)`
We have already deduced that potato(57) does nothing more than printing “`5*7*`”, so now we know immediately that potato(579) will print “`5*7*9*`”.

And of course, this pattern of reasoning can go on for ever. The potato function can be given any positive number. It will print all the digits of that number, each followed by its own star.

If I provided a boring little function like this:

```
void say_digit(const int d)
{ if (d==0)
    print(“zero”);
  else if (d==1)
    print(“one”);
  else if (d==2)
    print(“two”);
  else if (d==3)
    print(“three”);
  else if (d==4)
    print(“four”);
  else if (d==5)
    print(“five”);
  else if (d==6)
    print(“six”);
  else if (d==7)
    print(“seven”);
  else if (d==8)
    print(“eight”);
  else
    print(“nine”); }
```

I could use it in a trivially modified potato function:

```
void new_potato(const int X)
{ const int A=X/10, B=X%10;
  if (A!=0)
    potato(A);
  say_digit(B);
  print(“-”); }
```

and I would have something that speaks numbers, as when making cheques tamper-proof -

```
{ print(“Pay to the order of Ms. Jumbo Mouster the sum of “);
  new_potato(579);
  print(“ dollars\n”); }
```