**The Power of Computational Thinking**  
By Michael Franklin

Introduction:

Computational Thinking, a confusing subject at first, but when you boil it down it all comes back to one idea: “How do I solve this problem?”  
  
Computational Thinking teaches the Computation Thinking Process (CT Process) which gives a step-by-step guide to solving any problem, from a general perspective. Then CT goes into different strategies and “tools” that have very many similarities but all have their own use. Like a screwdriver versus a hammer. Sure, you could use a screwdriver to bang in a nail or try to pry a nail from the wall, but a hammer is just designed and made to fit that particular need far more comfortably than a screwdriver ever could be.

In this guide I will be walking you through my process of bringing the principals and procedures that CT has shown me to help solve this problem I have been wanting to confront for well over 3 months now.

My problem:

I want to write a program to teach people how to mentally tell if a number is equally divisible by another number quickly without a calculator.

Now I could just write a program where you input two numbers and it spit out “yep 12 is equally divisible by 4” but that really doesn’t solve the whole “teaching people how to tell” part of the problem, does it?  
  
Which brings me to the first key thing CT teaches: Decomposition and Abstraction.

Solution Breakdown:

Abstraction:

Abstraction is the process of determining, or simplifying, only what is necessary from a problem or procedure to better understand the problem or procedure.

Before we get too far deep into solving this problem, let’s take the time to figure out exactly what we are trying to solve, we what have to work with, and what want the solution to be. This is often called “Determining the Scope of a Problem” and is a form of Abstraction.

I stated before that I want to: “I want to write a program to teach people how to mentally tell if a number is equally divisible by another number quickly without a calculator.”.

* This seems a tad vague and some parts of this statement aren’t really needed to help form a solution.
  + “write a program” brings up a question of what language to use. I am very proficient in C++ so I will be coding in that.
  + “without a calculator” is a tad redundant if I already stated that I wanted to teach people how to do this mentally.
  + This program needs to “teach” people how to tell divisibility mentally and not just output that some number is equally divisible by another number.
* Also, we haven’t really stated any real constraints on what these input numbers should be. To make things easier to teach, let’s just have them be only integers, no decimals, and lets also put a bound on the divisors.
  + Let’s say 1 to 10 as those are common numbers people need to divide by (though when do you ever divide something by 1, but we’ll keep it for completeness).

With those concerns out of the way, let’s move on to:

Decomposition:

Decomposition is process of taking a complex problem or procedure and breaking it down into its sub parts; such that each subpart added together is the whole problem or procedure and that no subpart overlaps in detail or function with another subpart.

We can break down the different steps and modules of this program I want to make:

* Input and Validation:  
  We’ll need a way for the user to input what number they want to divide and what number they want to divide by. We also need to make sure these numbers are integers and the divisor is between 1 and 10.
* Divisor Algorithm Selection:  
  We need to input the user’s number into the correct algorithm based on what they entered as a divisor.
* Divisor Algorithms:  
  This portion will be stepping through the selected algorithm, possibly outputting each step and its result in a user-friendly way, as the number to analyzed by the algorithm.

* Output:  
  This module will be called on either during or after the selected Divisor Algorithm to print out what is happening to the user, this is probably the most crucial part as this is what will “teach” the user how to mentally tell if a number is equally divisible by another number”

Now this is just a quick overview of the things that I will need to make work in order to solve the bigger main problem. It may seem that my problems have “multiplied” but I promise breaking things up like this makes thing much easier to work with.

With these two ideas (Abstraction and Decomposition) we can really start diving into each of these modules applying these processes, and other tools I’ll show as we get to them, as we go through each module of this program.

Input and Validation:

Looking back at how I decomposed this module, it has two things it needs to do.

* Prompt the user for two integers: A number they want to divide by, and a divisor between 1 and 10.
  + Since I’ll be using C++, I can use cout and cin operators to grab this input.
* Validate the inputs are integers and the second input is between 1 through 10
  + This can be accomplished via a while loops that have those two checks in it
  + That would look something like
    - Get Input1 and Input2 from user.
    - While ((input2 is Not a Number) or (input2 is Not a Number)) or (input2 is not between 1 and 10))
      * Output Error to User that inputs need to be numbers and Input2 is between 1 and 10, then loop to get new input from the user.
        + In C++ I must also call cin.clear() to clear the input buffer for new input every loop.
      * C++ has a cin.fail() that will return true if anything is input that starts with an non-number. For example, inputting a letter or a decimal point first instead of a number.

Divisor Algorithm Selection:  
This will be a small simple function that has a switch in it that calls the other Divisor Algorithm functions based on which divisor is chosen.

Divisor Algorithms:

Coming to the divisor algorithms themselves, first we must determine how to tell if a given integer is divisible by a divisor. To that end I have done some simple research and found some very nice and concise “math tricks” I can use:

* 1 divides any number equally.
* A number is divisible by 2 if its last digit is 0, 2, 4, 6, or 8.
* A number is divisible by 3 if the sum of its digits is divisible by 3.
* A number is divisible by 4 if the number's last two digits are divisible by 4.
* A number is divisible by 5 if its last digit is 0 or 5.
* A number is divisible by 6 if its last digit is 0, 2, 4, 6, or 8 **and** if the **sum of its digits** is divisible by 3.
* Remove the last digit, double it, subtract it from the truncated original number and continue doing this until only one digit remains. If this is 0 or 7, then the original number is divisible by 7.
* A number is divisible by 8 if the last three digits are divisible 8.
* A number is divisible by 9 if the sum of its digits is divisible by 9.
* A number is divisible by 10 if its last digit is 0.

Looking at these “math tricks”:

* It seems there are a lot of “last x digits is” and “sums of digits”. So those two processes could be decomposed into their own functions for easy use everywhere.
* 7 seems like a tricky number to determine, we will cover this one more in depth as we get to it.

Now that we will be talking about them more, lets lay out what an “Algorithm” really is.

Algorithms:

In short, an Algorithm has:

* Unambiguous and ordered steps.
* Specific halting criteria.
* One of more inputs and produces one or more outputs.
* Consistency – a unique set of inputs will always output the same unique set of outputs.

For an example, I have made an algorithm to find the sum of all the digits of any integer:

* Algorithm: GetDIgitSum
* Parameters:
* a – any nonnegative integer
* b – a positive integer that holds the total amount of digits of a
* t – nonnegative integer output for the sum, starts at 0
* Returns: a nonnegative integer
* GetDigitSum(int a)
* {
* -- initialize t to 0 before we start looping
* t = 0
* -- initialize b using the formula to find the number of digits in a, rounding down
* b = floor(log(a) + 1)
* -- while b is greater than 0 keep looping
* while(b > 0)
* {
* -- First, get the value of the leftmost digit of a and add it to t
* -- to get the value of the leftmost digit of a:
* -- divide a by 10 to the b minus 1 power, then round that value down
* t = t + floor(a / 10^(b – 1))
* -- Next, “remove” the leftmost digit from a by:
* -- getting the value of the leftmost digit of a, as above
* -- multiply that value by 10 to the power of b minus 1 to get
* -- the “positional value” of the leftmost digit of a, then subtract that from a
* a = a – (floor(a / 10^(b – 1)) \* 10^(b – 1))
* -- decrement b to analyze the next digit of a during the next iteration
* b = b – 1
* }
* -- return the calculated sum of all a’s digits
* return t
* }

A little bit complex, but remember that a computer doesn’t know how to find the sum of all the digits of a number like a human does. Sure, I could convert the integer to something like a string and then just “get the first digit and the second digit”, turn those back into integers and add them together, but I really want to avoid the mess of switching variable types. There are many ways to solve a problem, each having pros and cons, which solution you choose is up to you.

Speaking of multiple solutions, the problem of getting the last x digits of a number also has many solutions. I chose to stick closely to the process I used above and came up with this:

* Algorithm: GetLastDigits
* Parameters:
* n – any nonnegative integer
* d – a positive integer that holds the total amount of digits to obtain from n (starting from the right going left)
* p – nonnegative integer that holds the current digit to be analyzed, starts at 0.
* t – nonnegative integer output for digits retrieved, starts at 0
* Returns: a nonnegative integer
* int GetLastDigits(int n, int d)
* {
* -- initialize t and p to 0 before we start looping
* t = 0
* p = 0
* -- while p is less than d keep looping
* while(p < d)
* {
* -- First, get the value of the pth digit from the end of n and add it to t
* -- to get the value of the pth digit of n:
* -- use modulo of n by 10 and multiply that by 10 to the p power
* -- then round that value down
* t = t + floor((n % 10) \* (10 ^ p))
* -- divide n by 10 to trim off this leftmost digit, this will allow the next digit to
* -- be analyzed by modulo if needed
* n = n / 10
* -- increment p, this allows for the next digit to be added properly to t if needed.
* p = p + 1
* }
* -- return the retrieved digits of n
* return t
* }

Very similar algorithms, that both will do much of the heavy lifting in determining divisibility for most of the divisors. Let’s see what these two algorithms help solve:

1 is the easiest to solve:

* 1 divides any number equally.
  + I can just print a simple statement explaining this, no math needed.

2, 4, 5, 8, and 10 all will use GetLastDigits:

2, 5, and 10 are simply solve as follows:

* A number is divisible by 2 if its last digit is 0, 2, 4, 6, or 8.
  + This will use GetLastDigits(input,1), to get the last digit of input.
    - Then I can just check if this digit is a 0, 2, 4, 6, or 8.
* A number is divisible by 5 if its last digit is 0 or 5.
  + This will use GetLastDigits(input,1), to get the last digit of input.
    - Then I can just check if this digit is a 0 or 5.
* A number is divisible by 10 if its last digit is 0.
  + This will use GetLastDigits(input,1), to get the last digit of input.
    - Then I can just check if this digit is a 0.

4 and 8 are a tad trickier:

* A number is divisible by 4 if the number's last two digits are divisible by 4.
  + This will use GetLastDigits(input,2), to get the last two digits of input.
    - Then I can check if this smaller number is divisible by 4 using modulo, or using a stored “times table array” for 4 that lists all the products of 4 up to 100.
* A number is divisible by 8 if the last three digits are divisible 8.
  + This will use GetLastDigits(input,3), to get the last three digits of input.
    - Then I can check if this smaller number is divisible by 8 using modulo, or using a stored “times table array” for 8 that lists all the products of 8 up to 1000.
      * To make this easier, I could also use a second “trick with 8’s products”:
        + If the third digit is even, then just follow the normal times tables for 8

480 is divisible by 8 as 80 is a product of 8.

* + - * + If the third digit is odd, all the times tables will be reduced by 4.

376 is divisible by 8 as 76 is 4 less than 80 which is a product of 8.

3 and 9 will use GetDigitSum:

* A number is divisible by 3 if the sum of its digits is divisible by 3.
  + This will use GetDigitSum(input), to get the sum of the digits of input.
    - I can then check if this smaller number is small enough, less than 10, and if so check if the number is 3, 6 or 9 as those are all the products of 3 less than 10.
      * If it is not smaller, I can just run this smaller number through GetDigitSum again to make it smaller.
* A number is divisible by 9 if the sum of its digits is divisible by 9.
  + This will use GetDigitSum(input), to get the sum of the digits of input.
    - I can then check if this smaller number is small enough, less than 10, and if so check if the number is 9 as this is the only product of 9 less than 10.
      * If it is not smaller, I can just run this smaller number through GetDigitSum again to make it smaller.

6 combines the checks of 2 and 3:

* A number is divisible by 6 if its last digit is 0, 2, 4, 6, or 8 **and** if the **sum of its digits** is divisible by 3.
  + 6 is the product of 2 and 3. Therefore, the input must meet both requirements for 2 and 3.
    - This is true of any divisor that has two different factors, such as 12 (4 and 3) or even 48 (8 and 6), but these larger divisors are beyond the scope of this solution.

7 is the odd ball of the group as it has its own algorithm:

* Remove the last digit, double it, subtract it from the truncated original number and continue doing this until only one digit remains. If this is 0 or 7, then the original number is divisible by 7.

* Algorithm: DivideBy7
* Parameters:
* n – any nonnegative integer
* d – a positive integer that holds the total amount of digits to obtain from n (starting from the right going left)
* l – holds the truncated last digit of n
* div – Boolean that holds if n is divisible by 7 or not
* Returns: a Boolean (1 if n is divisible by 7, 0 if not)
* bool DivideBy7(int n)
* {
* -- initialize div as false before we start looping
* div = false
* -- initialize d using the formula to find the number of digits in a, rounding down
* d = floor(log(number) + 1)
* -- while we have more than 1 digit, keep looping
* while (d > 1)
* {
* -- get the leftmost digit of n
* l = GetLastDigits(n, 1)
* -- double l
* l = l \* 2
* -- strip of the leftmost digit of n
* n = floor(n / 10)
* -- set n equal to the difference between n and l
* n = n - l
* -- recalculate the new number of digits of n to see if we need to loop again
* d = floor(log(number) + 1)
* }
* -- now that we only have 1 digit left, check if it is 0 or 7
* if (n = 0 or n = 7)
* {
* -- if it is, then the original n is divisible by 7
* div = true
* }
* -- return if n is divisible by 7
* return div
* }

It’s a little complex and might be hard to see how this and the previous two algorithms work. A good way to get a better grasp on these algorithms is to step through them one line at a time and build what is called a Trace Table.

Trace Tables:

A Trace Table is a table that has the iterations of an algorithm as either the row or column header and the different parameters of the algorthim as the opposite header. The table is populated with all the different values of each parameter during each iteration, started with iteration 0 (default values before the algorithm starts proper).

Here is an example of DivideBy7’s Trace Table using the number 3892 as n:

(Remember that after iteration 1, the code will have gone through the while loop once)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iteration | n | d | l | div |
| 0 | 4515 | 4 | 0 | 0 |
| 1 | 441 | 3 | 10 | 0 |
| 2 | 42 | 2 | 2 | 0 |
| 3 | 0 | 1 | 4 | 1 |

Notice how n changes drastically from iteration. That is because:  
5 is the last digit in 4515

5 \* 2 = 10

Removing 5 from 4515 leaves 451  
451 – 10 = 441  
repeat this every iteration and you end up with 0, which is either 0 or 7 and thus this algorithm shows that 4515 is divisible by 7. In fact, 4515 is 7 \* 645, proving this algorithm works.

Now, there is a problem with this algorithm as numbers that are divisible by 7, like 3892, do cause it to break as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iteration | n | d | l | div |
| 0 | 3892 | 4 | 0 | 0 |
| 1 | 385 | 3 | 4 | 0 |
| 2 | 28 | 2 | 10 | 0 |
| 3 | -14 | 2 | 16 | 0 |
| 4 | -9 | 1 | 8 | 0 |

As you can see this error pops up with only some unique digit sequences, and would only be found by trial and error, or by understanding the algorithm by way of something like a Trace Table.

The fix for this is quite simple though. Change the line:  
-- set n equal to the difference between n and l

n = n – l  
  
to:

-- set n equal to the absolute difference between n and l

n = abs(n – l)

this would result in the following correct Trace Table for 3892:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iteration | n | d | l | div |
| 0 | 3892 | 4 | 0 | 0 |
| 1 | 385 | 3 | 4 | 0 |
| 2 | 28 | 2 | 10 | 0 |
| 3 | 14 | 2 | 16 | 0 |
| 4 | 7 | 1 | 8 | 1 |

As 2 – 16 = -14, but the absolute value of -14 is 14.

On iteration 4, 1 – 8 = -7, but the absolute value of -7 is 7, showing 3892 is, as it should be, divisible by 7.

That covers all the algorithms and procedures for how the computer figures out if any given integer is divisible by a given divisor (1 – 10). (Negative integers can but be put through the abs function to get the absolute value of the number as we can safely ignore the negative sign for all these algorithms. Now, how to we represent all this to the user? Will we just show them how the computer figured it out? Certainly not. We will show them how they themselves can do it. Which brings us to:  
  
Output:

This module is all about presenting a communicating with the user. The user doesn’t need to know what my algorithms are doing, they simply want to know how what is important to them. In this case, “how to mentally tell if a number is divisible by another number”. The “math tricks” I found will be very helpful and I will be displaying the appropriate divisor’s math trick first before going step by step with the input number as an example for the user.

There are a few ways that I can go about this, either keeping all the “output logic” completely separate from the algorithms and other logic (the usual and “proper” way of programming), or I could integrate the “output logic” into each of my algorithms so that I only have to call the algorithm and it will send the appropriate logic to the user. I think I am going to do the latter. Now I could do this the “proper” way, but that way assumes you have a much larger and more complex program with a few, “foundational steps”, such as having all your variables you want to print tracked separately outside the algorithm function, that my small, simple program just doesn’t have. As such, integrating the output logic into each of my algorithms well be the best way in this case.  
  
The way I am going to go about this is by asking the question, “how would a human do what the computer is doing right now?”, and print that answer out to the computer. For example, my GetDigitSum algorithm will print out something like this:  
  
Get the sum of all the digits of 1544.  
1 + 5 + 4 + 4 = 14  
  
Two lines, instead of the crazy while looping mess that would just confuse a user.

To achieve this, I could put the first line at the top of the algorithm, to tell the user what

we are about to do.

The second line might require a few modifications to my existing algorithms, such as putting in a few checks to see what iteration we are on and maybe moving parts of the calculations to their own variables to be displayed, but that’s ok.

**Don’t be afraid to go back and reevaluate the previous parts of your solution if new problems present themselves. “Trial and error”, is part of the problem-solving process too.**

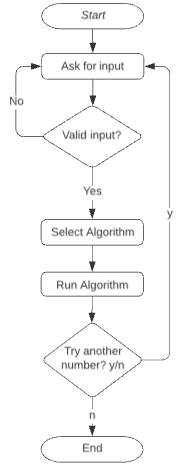
In the end, output is less “calculation logic” and more “where will I put this output statement”. This module is all about deciding what to say and when/where to say it.

Putting it all together:  
Now that we have out modules, or parts, of the solution all built and working, we can lay out how each will flow into and interact with the other to get a good overview of how the entire solution will flow. To assist in this, we can use a diagram:

Diagrams:

Diagrams are very powerful and useful ways to get a different or more general point of view of how data or parts of a problem work and look. If you are ever stuck on something, try finding a way to draw it into a diagram, like a graph or chart.

In this case we will use a diagram called a flowchart:



See how with this we can get a very quick and easy look at how the finished program, the solution to my original problem, will flow. As you can see it is a very simple program, but still it required a bit of work to figure out all that needed to be done.

Closing Statements:

Now all that is left is to convert all this “logic” into the proper C++ syntax and compile, then we can truly say we are “done”. Though that does bring up a very good question. How do you know when a problem has been “solved”? Surely, there is so much more I can do to make this program better, such as adding in a “how to find a quotient mentally” module, or maybe I could include more divisors to teach the user how to divide by bigger numbers that these tricks don’t cover. I could list tens, hundreds, of things that I could add into the program that would make it a better “solution”, but you have to ask yourself “where do I draw the line”.

A good way to make sure you don’t keep adding and adding unneeded complexity into your solution is to look back at the “scope” of your problem. This was part of the very first thing we did during the initial abstraction process. However, there is another tool you can use in addition to abstraction to help you define scope – Use Cases.

Use Cases:

A use case is a scenario or manner in which you know or believe your solution will be used in.

In the case of my program, (the solution):

* I imagine this program would get a lot of use in Elementary schools where younger students have learned basic division and are getting into the higher math classes.
* This program also could be an interesting utility to get minds thinking about how numbers interact and how sometimes, the straight forward solution isn’t always the most efficient.

When thinking though use cases you are intentionally formulating the restraints for your solution to fit those use cases. I could add more divisors to my program, but younger students will probably not need anything higher than 10 for “quick math”. I could add in “a how to get the quotient of a number mentally” but that doesn’t fall into the original scope of this problem that I abstracted out.

So, “How do you know when you have solved the problem?”. The answer is, “When you feel you have solved all the issues of each use case you can think of for your problem”.

Conclusion:

Computational Thinking can be a hard subject to get to grips with at first, with all its terms and similar looking principals and processes for solving different problems. Though, once you take a step back and realize it’s all pointing in the same direction, things do get easier to understand. You have decomposition, abstraction, and even use cases to help pick apart and understand what the problem you are looking at is. Then you take the problem one part at a time, decomposing and abstracting as you go, to get more and more clarity, and solve each part one at a time. CT also provides many tools and principals to help overcome some of the more tedious or complex issues of a problem. From, algorithms that take a complex process and break it down step by step, to learning to draw different diagrams that give a good overview and “birds eye” view of data or the process of a solution you’ve come up with. CT isn’t about how to solve a limited selection of problems, its about how to solve any problem, carefully and methodically, and gives you specific tools to help solve specific problems you might face along the way.

In the end, how you solve your problems is up to you. However, if you use the tools and principals from Computational Thinking, things get a little easier to understand, manage, and work through.