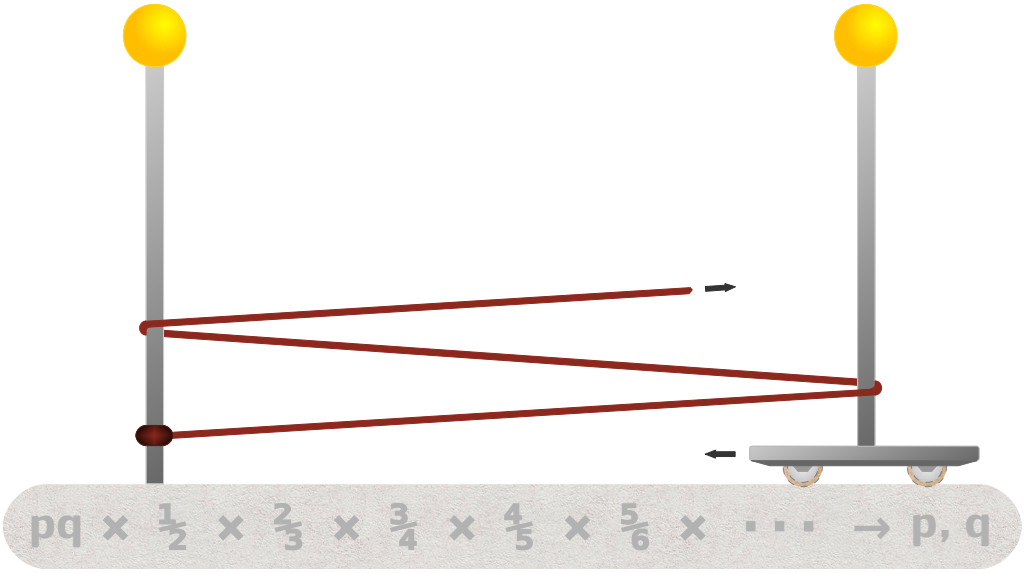
PROGRAMMING

— FOR THE —

MATHEMATICIAN



BUILD ADVANCED CRYPTOGRAPHIC TOOLS IN FIVE PAGES

A MINIMALIST INTRODUCTION TO CORE C++ AND HOW TO WORK WITH NUMBERS OF ARBITRARY SIZE & PRECISION

Ukraine 2020

&Number theory motivations

UNSTOPPABLE FUNCTIONALITY

Data transmission is never limited to equipment boundaries just like energy is transformed, never destroyed. Some tools are engineered to take advantage of equipment with poor computation isolation while other tools intercept and remember the endless signals born in the palm of your hand and broadcast beyond the solar system. Information security gives you the confidence that no entity can make sense of your data without the key, no matter how far your message has gone or how many devices attempted to break it.

Modern cryptography has made guesswork so impractical, the adversary must resort to some human weakness that leads directly to the key. Modern encryption keys may not exceed the length of short stories yet the number of guess possibilities for such a length is not comprehensible. And unlike the scientific equipment that is limited to the physical world, mathematics—being the language of the universe—opens a playground to the numerous entities analyzing the protocol in question. “May have cryptographic implications” has become the phrase of panic behind the conference room doors of the NSA—the agency whose approval determines number theory publication rights.

Algorithms are forever, no rusting parts, no lubrication required. Just as the world of mathematics is accessible to any entity, classified functions can always be discovered and duplicated no matter the guarding procedure. And just as algorithms exist in a perfect world, some functions pose incredible difficulties while others remain truly unbreakable.

TIMELESS FAME

Given that mathematics can never be irrelevant, mathematicians will always be remembered for their impressive work. Currently, there are a handful of unsolved mathematical challenges each holding a million-dollar prize and anyone with pencil and paper are entered to win. These kind of solutions would help us further understand the possibilities and impossibilities of reality, while smaller challenges attempt to measure the current state of cryptography. See claymath.org for the millions, EFF primes, rule 30 prize of $30,000 (cellular automata,) and the last-standing RSA numbers—progress for which halted as only the long semiprimes remain.

ULTIMATE TRUTH

Mathematical proofs are considered the most important part of integer manipulation and building new ideas. Scientific measurement can only be so accurate, but mathematical “aha” moments—as addicting as they are—give you the confidence to build and describe reality without fail. Mathematical truth being the result of logical deduction, anyone willing to consider your proof will be convinced of its truth regardless of the reader’s background. They need not rely on witnesses or systems of belief, and the numbers you have them consider will never lie or turn away.

UNLIMITED ENGINEERING

Like the brave problem-solvers who climb rope-free, analyzing every rock surface and applying the right grip technique, mathematicians can be compared to those who crave that most rewarding rush. A rigorous dig for discoveries—from anywhere, at any time. No expense for tools and not a single noise complaint. It’s the combination of these motivations that drive mathematicians to the outer reaches of infinity as they remain silent and undisturbed.

Flipping the sign

C++ is a language of beauty and speed. You will learn how to build and run algorithms for the exhaustive crunching of any and all numbers beginning with variables of limited size constraints. From there, you will have the skills to string together and manipulate individual string digits so as to represent values of arbitrary size or decimal expansion precision with total control and plenty of room for interjecting code—all as simple as dealing explicitly with our familiar ten counting symbols—zero through nine.

Whatever your numerical engine requirements, this book alone gives you everything you need to build advanced cryptographic tools with Command Line Interface, advanced cryptographic engines for networking and messaging apps, machine learning algorithms, basic game simulations, random number generators, hash functions, lossy & lossless compression, search & sorting, character & base conversion, and of course—the complex testing for counter-examples up to some limit—giving you the statistical confidence to pursue mathematical proof. The following shows you how to run your first few lines of code where you will be multiplying by negative-one. This operation turns negatives into positives and positives into negatives. Next, you will force two variables to swap values, without using temporary copies of those values.

Software package repositories for GNU+Linux operating systems have all the developer tools you need. **Open a terminal and use the following command as root** to install Geany and g++ on your computer.

```
apt install geany g++
```

Geany is a fast and lightweight text editor and Integrated Development Environment (IDE) in which you can write and run code. g++ is the GNU compiler for C++ which allows written code to run. The compiler operates in the background automatically and displays errors in your code as you will see in the lower box in Geany. **Create a folder somewhere on your machine.** Open Geany and reproduce the following code by **typing it into the blank Geany page.** This is how all your projects will begin.

(To zoom in or enlarge the code within Geany as shown in the sample, hold Control and scroll your mouse.) Now you can get code-highlighting and set this document as a C++ source file. Go to **Document >> Set Filetype >> Programming Languages >> C++ source file.** And finally, **save the file as anything.cpp** within the newly created folder (Geany will automatically suggest the .cpp extension.) This folder is now the home directory of your coding project—which can create, write to, and read from files within this folder. You may now build then run the program. Use the following keyboard shortcuts and in order.

build F9 run F5

1 Variables

"Keep imagining components until they are permanent memory,
only then can you effortlessly simulate the larger structures
where that elegant solution shows right up." -anon

In the sample code, the space between the two curly braces is there for all your code while you need not worry about the surrounding garbage. In that sample program, using a single line of code you multiplied a pre-set value by negative-one. You can try erasing and inserting different values—or you can replace both 1984 and -1 by variables who can be modified by other operations as the program is running. In this next sample, you are declaring or making two variables—the only ones we will use throughout this book. Note the semicolon after the declarations, they are required for lines that strictly do something... we will place semicolons after operations.

```
int main()
{   int variable_1;
    long big_number_SUB2;
}
```

The first variable is of **type** int which stands for integer. It can hold values as high as just over two billion and values as low as their negatives. Here, you have **declared** "variable_1" which can have just about any random name beginning with a character.

The second variable is of **type** long—the type with extra capacity. It can hold values as long as eighteen digits, and their negatives. This type takes up more space than int which means it's impractical when you're declaring large chunks of RAM. This will become clear in chapter three—declaring variables in bulk. Here, you **declared** "big_number_SUB2" which can have just about any random name beginning with a character.

You are strongly encouraged to painstakingly reproduce each and every sample block of code, and test the compiler until something breaks. Here, the compiler tells you the two variables are unused. You have not printed them in the terminal and they are left unmodified. Let us assign values to these variables, otherwise they take up space without storing anything useful. In fact right now, these variables contain whatever garbage value was left in memory by other programs. You can initialize these variables manually here and now, or they can be initialized by another operation somewhere in the code.

```
int main()
{   int variable_1 = 343;
    long big_number_SUB2 = 100000000000000000;

    cout << variable_1;
    cout << " and this one ";
    cout << big_number_SUB2;
}
```

big_number_SUB2 has been **initialized**, its value is now a total of eighteen digits long. You may have noticed by now that "cout" means print to screen. You can print variable values as well as some sentence with numbers included—within the quotes that is. This string being printed between the outputs of the two variables makes the formatting clear as there is a separation between the two numbers. Try removing that middle out-stream.

Code is executed in a procedural fashion meaning the program reads and executes the first line, then the second line, and so on. It is possible to initialize variables manually, as the program is running. In the following sample, the terminal will ask that you use your keyboard to enter a new value for variable_1 even though it had been set to 343 previously. variable_1 will simply be overwritten. Notice the operator << has been swapped to >> for the "cin >>" command.

```
int main()
{   int variable_1 = 343;
    long big_number_SUB2 = 100000000000000000;

    cin >> variable_1;
    cout << variable_1 + big_number_SUB2;
}
```

variable_1 could have been left uninitialized as it was being declared. This is because **you are asked to modify variable_1** anyway. Here, it gains value no matter what happens. The program then prints one value: **(variable_1 + big_number_SUB6)**. Note that only variable_1 is being printed... plus something. And C++ automatically and temporarily converts variable_1 to type long since int cannot hold eighteen-digit numbers.

```
int main()
{   int variable_1 = 343;
    long big_number_SUB2 = 100000000000000000;

    cin >> variable_1;
    big_number_SUB2 += variable_1;
    cout << big_number_SUB2;
}
```

+= means big_number_SUB2 **is now equal to** itself plus something. This modifies big_number_SUB2 right then and there, it gets a new value which is the sum of the two variables after you have given your keyboard-input. Since the summing operation here is saved in memory before being printed, only big_number_SUB2 can hold such a large value and so it is written on the left side of the operation.

	a = b;	a is now equal to b		
	a++;	increment a by 1		
	a--;	decrement a by 1		
cout << a + b;	a += b;	a is now equal to (a + b)	a = (b + c);	a is now equal to (b + c)
cout << a - b;	a -= b;	a is now equal to (a - b)	a = (b - c);	a is now equal to (b - c)
cout << a * b;	a *= b;	a is now equal to (a * b)	a = (b * c);	a is now equal to (b * c)
cout << a / b;	a /= b;	a is now equal to (a / b)	a = (b / c);	a is now equal to (b / c)
cout << a % b;	a %= b;	a is now equal to (a mod b)	a = (b % c);	a is now equal to (b mod c)
a & b are NOT modified		Only a is modified		Only a is modified

Eight mathematical operation types are shown in the table. Parentheses help the compiler identify what operation to perform first. Although parentheses are usually unnecessary for small operations, we will always include them for clarity. Division operations are treated slightly differently since both int and long don't have decimal expansions. Any such digits are stripped away from some resulting value. If we declare a variable of type int then initialize it to equal 100, a can still be divided by three or other numbers. The result would be exactly 33. And dividing by ten simply strips the last digit of a whether it's 100 or 123. If a is initialized to 123, a /= 10; yields exactly 12.

Let's say we declare two variables of type int—a and b. Additionally, we initialize our variables so a = 6 and b = 12—twice that of a. Now let's attempt to swap the values of a and b the traditional way. By the time this program ends, a should be equal to b and b should be equal to a. Here, we must introduce a third temporary variable since the operation a = b; means a and b are equal hence value a is lost.

```
int main()
{   int a = 6, b = 12, temporary; //Declarations and initializations on the same line! (Separated by commas.)

    temporary = a;
    a = b;
    b = temporary;

    cout << a << "\n" << b; //This is a comment, it's created using two forward slashes. \n means print new line.
    //                      This cout prints multiple items in the same line! (Separated by the << operators.)
}
```

This alternate solution avoids the use of an extra variable. a and b are initialized in that order, as well as printed in that order, however, the resulting output is -17 then 57, separated by a few spaces.

```
int main()
{   int a = 57, b = -17;

    b += a;
    a -= b;
    b += a;
    a *= -1;

    cout << a << "    " << b;
}
```

This sample project swaps the individual digits for two-digit values. a is declared and initialized to 29 but its value when printed is 92.

```
int main()
{   int a = 29, b;

    b = (a % 10);
    a /= 10;
    a += (b * 10);

    cout << a;
}
```

One day you'll incorporate file I/O, and one day you'll write keys to storage. Modern flash and SSD devices come with more memory than is displayed and when written to, do not immediately overwrite data marked for deletion. This increases device life-span while undermining security. "Encrypt & replace file" does not explicitly mean the unencrypted version is overwritten when you completely flood device memory. Encrypt device format beforehand or encrypt data on a separate device whose memory can be demolished effectively without the chance of reverse and recovery.

infosec tip

```cpp
int main()
{   long volts, amps;

    cout << "Enter voltage: ";
    cin >> volts;

    cout << "Enter amperage: ";
    cin >> amps;

    cout << "\nThat's " << (volts * amps) << " watts!";
    if(volts == amps) {cout << " A perfect square!";}
}
```

This sample demonstrates the **if statement**. It has curly braces of its own which make up its body. Here, this body of code is executed only if volts are equal to amps. And since the execution is a short and simple cout, the curly braces reside on the same line. Please enable indentation guides for extra clarity as our code from here will contain multiple blocks. Go to **View >> Show Indentation Guides.**

==	*equal to*
!=	*not equal to*
>	*greater than*
>=	*greater than or equal to*
<	*less than*
<=	*less than or equal to*

if

```cpp
if((volts * amps) >= 1000000000)
{   cout << "Gigawatts!\n";
    cout << "V = " << volts;
    cout << "\nI = " << amps;
}
```

```cpp
if((volts * amps) >= 1000000000)
{   cout << "Gigawatts!\n";

    //This if is nested in the 1st.
    if((volts % 2) == 0)
    {   cout << "V is even.";
    }
}
```

If statements are always evaluated by the program. The block of code in their bodies, however, is executed only if some condition is true. The program then continues onto other lines of code.

if else

```cpp
if((volts * amps) >= 1000000000)
{   cout << "Gigawatts!\n";
}

else
{   cout << "Insignificant power.";
}
```

```cpp
int a = 47;

if(((a * a) + a) != ((a + a) * a))
{   cout << "Non-commutative!";
}

else {cout << "Try again.";}
```

The if statement here has an else statement to go along with it. A satisfied if statement means its body is executed normally, otherwise the body of the else statement is executed. And the else statement has nothing to check or compare in order to be executed.

if else if else if else

```cpp
cout << "(1) Get package. (2) Couple pakage. (3) Verify coupling.";

cout << "\n\nEnter option: ";

int user_option;
cin >> user_option;

     if(user_option == 1) {cout << "Welcome, new user!";}
else if(user_option == 2) {cout << "Insert private files. ";}
else if(user_option == 3) {cout << "Insert public files.";}
else {cout << "Invalid option.";}
```

This is just another if else. In between however, there exist additional possibilities of whatever values a list of variables gain. If one of those statements are satisfied, its body is executed and the program continues to the next line after this array of comparison statements—without testing the rest of the else if statements. And of course, if no statements are satisfied then only the body of the else is executed.

In the last sample above, the if has been moved to the right for clarity, we can always space out parentheses, statements, and operations if it makes it easier to read. The following two items conclude chapter 1. And, or. These are symbolized using && for and, and || for or—two Unix pipes. The Unix pipe is on (Shift + backslash.) if((a == 6) && (b == 12)) executes the if body only if a = 6 *and* b = 12.

&& ||

```cpp
long a = 1e17;

//1e17 means 1*(10^17.) It has 18 digits.
if((a == 1000) || (a == 1e10) || (a == 1e17))
{   cout << "a has 3, 10, or 17 zeros.";
}
```

```cpp
int a, b, c, d;

cin >> a;
cin >> b;
cin >> c;
cin >> d;

if((a == 2) &&
   (b == 3) &&
   (c == 5) &&
   (d == 7))
{   cout << "You entered ";
    cout << "the first ";
    cout << "four primes!";
}
```

Do not underestimate your government! You should always backup and cache your data especially if sensitive. This allows for contingency plans and action on your behalf. Give your storage devices common sealed containers and place or bury them guardedly in public infrastructure while under adequate cover from aerial view.

infosec tip

2 Loops

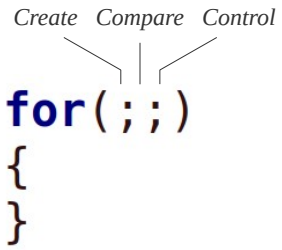
"It's like a dog chasing its tail. Observers are helplessly consumed by it until the dog has counted down to zero." -anon

Chapter 1 had just covered 99% of the material in this book, however, you need not worry if you feel unprepared to continue. The remaining two chapters and the remainder of this book heavily reiterate chapter 1. Chapter 2 only gives us the ability to repeat commands n times, or until some condition of your choice becomes true. And chapter 3 gives us the ability to declare variables in bulk, where each variable name is simply zero through n. This gives us a convenient numerical way of accessing the value behind each variable.

```
//Detects gravitational waves by testing
//the local properties of integers.
//Magnitude log: prints deviation from 4 over time.

for(;;)
{   if((2 + 2) != 4)
    {   cout << 2 + 2 << "\a\n";
    }
}
```

This loop runs the if statement without end. And the if statement checks if **2 + 2** is equal to **4**. This happens millions of times per second. When **2 + 2 ≠ 4**, the value of **2 + 2** (whatever it may be at that moment) is printed to the terminal and so we get a log of the peaks and troughs for some high-energy event. Additionally, we get an alert sounded for every time **2 + 2** deviates from **4**. This is due to the **\a** escape sequence placed before the **\n** or new line. Older hardware will sound the cylindrical speaker soldered to the motherboard to execute this **\a** alert. Perhaps such a detector would be functional if we just add a sensitivity knob which swaps **2 + 2** with operations of increasing complexity. Or perhaps it might be the timing of value return for stretched semiconductor circuits that interfere with computation.



Imagine the for loop symbols are just a friendly little spider, its brain in the center will do all the work of numerical comparison. We can fit "**a < b**" right between the semicolons, this tells the spider to execute the loop body while **a** is less than **b**. And the spider keeps checking if **a** really is less than **b** in order to execute the block of code again. Then **a** or **b** can be incremented or decremented somewhere in the loop body so that the program can break out. We could leave it at that note as this gives us all we need to know about loops, however, we can write elegant for loops using its remaining two properties. Recall that we can choose to include or exclude spaces and tabs anywhere in our code in order to clarify our "translation of ideas to computation." Additionally, we can print any variable, anywhere to see what the program is doing, this can help find code errors—what programmers call bugs, this helps us debug our code.

```
int a = 0;
for(; a < 10; a++)
{   cout << "The value of a is " << a << "\n";
}
```

```
for(int a = 0; a < 10; a++)
{   cout << "The value of a is " << a << "\n";
}
```

The two loops here accomplish the same thing. Both loops print something ten times. **a** is incremented by one after being compared to **10**. Once the loop body has been executed once, **a** is compared to **10** again, if it's less than **10**, the loop body is executed again and so on. As soon as **a** is equal to **10**, the program breaks out of the loop and continues running whatever lines are next. In the first sample, **a** being declared outside of the loop means **a** can be later used by other lines of code or other loops. Its value will be whatever this loop left it with, however, it can be reset to zero at any time by saying **a = 0;** somewhere after this loop.

The second of the two samples is the one we'll be using for most iterative operations since beginning with **0** gives us a convenient way to call upon the **0th** variable and so on—this will make sense in chapter three. This loop declares or creates variable **a** inside the for loop declaration which saves on space and ensures that **a** is destroyed after this loop ends. Here, **a** is local to the loop, once it's utilized by the loop, it can no longer be called upon as it does not exist. **a** is local to this loop. It is said that **a**'s domain or **scope** is this loop, rather than **global.**

`break;`

break; tells the program to break out of a loop as soon as this command is encountered. We can even have an if statement in the loop which breaks out of the loop on some condition, break; being inside the if body.

`continue;`

continue; tells the program to jump to the beginning of a loop as soon as this command is encountered, forcing the spider to make another comparison . The loop then iterates again if the comparison is true. And continue; can exist within an if statement—within that loop.

`return 0;`

return 0; tells the program to terminate or halt as soon as this command is encountered. return 0; has no placement constraints, it can be placed anywhere in the program. These three commands are useful but rarely used.

3 Arrays

"Every once in a while, you can escape the trade-off between time, complexity, and space. Computational imagination is where you hover symbols closely together and look for new meaning." -anon

Arrays are no different than int or long, an array simply consists of numerous variables of type long or int. And each variable name is just an automatic enumeration from zero to (n – 1.) That's because variable zero is one of the variables. We can have for example, an array of size ten, which holds ten variables whose names are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Here, variable 0 is the first while variable 9 is the tenth. The value of any one of these variables can be just over two billion if type int, and eighteen digits in length if type long. And variable names being numerical rather than alphabetical means we can move about the array and access each variable value arithmetically.

```
int plaintext[600];

int table[100000] = {0};

long table_snapshot[10] = {122, 9009, 3};
```

plaintext	0	1	2	3	...	597	598	599

The array **plaintext[]** is essentially a group of 600 variables of type int. We can print the value of its first **element** by saying **cout << plaintext[0];** and we can add to it by saying **cout << plaintext[0] + 2;** however, we would be dealing with a garbage value since **element 0** was not initialized. This means the current value of **element 0** is whatever was left in its RAM location by a previously ran program using the same memory.

The remaining two sample arrays are initialized upon declaration, which is not always necessary as our program may later fill some array with genuine values. **table[0]** is set to the value **0** and C++ automatically assignes **0** to its remaining elements, all the way down to **element 599**. C++ does this only if we initialize one or more array elements upon declaration.
table_snapshot[0] is set to **122**.
table_snapshot[1] is set to **9009**.
table_snapshot[2] is set to **3**.
And all remaining **table_snapshot[]** elements are set to **0**.

```
int sieve[100000] = {1, 1};
for(int prime = 2; prime < 317; prime++) //317 is sqrt(100000)
{   for(int a = prime + prime; a < 100000; a += prime) {sieve[a] = 1;}
}

for(int a = 0; a < 100000; a++)
{   cout << sieve[a];
}
```

sieve[0] is equal to **1**.
sieve[1] is equal to **1**.
sieve[2] is equal to **0**. (prime)
sieve[3] is equal to **0**. (prime)
sieve[4] is equal to **1**.
sieve[5] is equal to **0**. (prime)
sieve[6] is equal to **1**.
sieve[7] is equal to **0**. (prime)

sieve[] is declared and its first two elements are initialized to **1** and **1**. The for loop immediately after contains another for loop inside. This inner for loop is said to be nested in the first. Once the loops are finished processing **sieve[]**, the last loop prints the value of each and every variable or element of the **sieve[]** array. If we run the program, we will see quite a few ones and zeros who are scattered throughout this output. Those processing loops had computed the sieve of Eratosthenes on an interval of 0 to 100,000. Zeros represent primes and ones represent composites, except for the first two since 0 and 1 are neither prime nor composite. We can later use this sieve table as-is to check if some number is prime by saying
if(sieve[777] == 0) {cout << "yes";}
This is possible because the first zero in the sieve is element 2 therefore zeros are conveniently mapped to actual prime values.

```
for(int a = 0; a < 100000; a++)
{   if(sieve[a] == 0) {cout << a << "\n";}
}
```

If we replace the last for loop with this one (differs only by the body content,) we will get a print-out of actual primes. This is because what's printed is not the values of each element but the element name itself—if its value is zero. **a** serves as a simple counter which we increase by one. Whatever **a** is at the moment, if the value of **element a** is zero, **a** is printed.

```
int read_bookmark = 0;

int write_bookmark = 0;

int write_bookmark_crypt = 0;
```

We can fill arrays and later read from them, segments at a time—perhaps we're processing chunks of data rather than as a whole, and arrays can be a handful if there's exhaustive computation in between the read cycles. For this, we'll need a way to bookmark which element we are to read from (or write to) next when we're done storing computed values elsewhere. However many elements we read from an array is the number we add to the read-bookmark immediately after, this way we always come back to an element unread. **read_bookmark** would be the element name which is updated dynamically. We can even read from and write to the same array since it would be inefficient to just declare a second array. These ordinary declarations demonstrate meaningful variable names, good programming practice means naming things based on function or something strongly related. Additionally, larger arrays can be declared using "static." Depending on RAM size, we can declare large arrays by saying **static int table[100000000];**

```
if(table[a + 1] > 9) {table[a + 1] %= 10;}
```

The last if statement demonstrates element accessibility. We are inquiring about the element one-over to the right of **a**.

Appendix

Randomness

Notice the new necessary #include directives. This loop prints 80 random digits generated by rand() which produces a large integer and so this value undergoes mod 10 in order to utilize the last digit. These digits sprouted from the seed **time(0)** within the **srand()** function. **time(0)** is the Unix time —the number of seconds that have passed since **Jan 1, 1970.**

```
#include <cstdlib> //For rand() and srand().
#include <ctime>   //For time function.
#include <iostream>
using namespace std;

int main()
{   srand(time(0));

    for(int a = 0; a < 80; a++)
    {   cout << rand() % 10;
    }

    cout << "\n\nseed = " << time(0);
}
```

This example generates random numbers **without Unix time.** Instead, there are fifty **user-defined seeds**, each eight digits long. What the user doesn't notice is that by the time they're done with input, they will have entered 400 digits. But that's not all. Each user-defined seed is used to fill the entire array again, constructively. Essentially, there are fifty random strings of size 1,000 who are combined to form one scrambled string of size 1,000. The combination method makes use of the all-way function used in the One-time pad to achieve unlimited plausible deniability or perfect secrecy. See the Authorship documentation link below to see the proof.

```
int Table_private[1000] = {0};
cout << "Enter 8 random digits, repeat 50 times.\n\n";

int user_seed[50];
for(int a = 0; a < 50; a++)
{   if(a < 9) {cout << " " << (a + 1) << " of 50: ";}
    else {cout << (a + 1) << " of 50: ";}

    cin >> user_seed[a];
}

for(int a = 0; a < 50; a++)
{   srand(user_seed[a]);

    for(int b = 0; b < 1000; b++)
    {   Table_private[b] += (rand() % 10);
        Table_private[b] %= 10;
    }
}

for(int a = 0; a < 1000; a++)
{   cout << Table_private[a];
}
```

Any data open to observation—raw or as the result of use and modification—indirectly reveals the generator from which sprouted the random numbers. Had they sprouted from a twenty-digit seed, the difficulty of decrypting some message would be guessing that seed. Message time-stamps and file modification dates well-approximate the Unix time seed of that moment and anyone with heavy resources can easily brute-force a narrow set of values. Some random number generators include mouse movements, image sensor noise, and garbage left in memory for their seeds but if some procedure doesn't match the overall difficulty, the weakest link will be analyzed at every angle.

infosec tip

File I/O

Notice again, one new necessary #include directive. Once **ofstream** declares the randomly-chosen name **out_stream** then we can replace **cout** with **out_stream** and write 1,000 digits to a file from array **Table_private[]**. We can replace the file name **Table.private** to our liking. This file is a simple blank or used text file. And C++ will create one for you in the folder of the current **.cpp** file if one does not already exist. Every time the program runs, **Table.private** is overwritten with the same 1,000 digits.

```
#include <fstream> //For file I/O
#include <iostream>
using namespace std;

int main()
{   int Table_private[1000] = {2, 3, 5, 8};

    ofstream out_stream;
    out_stream.open("Table.private");

    for(int a = 0; a < 1000; a++)
    {   out_stream << Table_private[a];
    }

    out_stream.close();
}
```

For file input, we must introduce type char, it holds values up to 127 and their negatives. Although type char can handle characters and takes up significantly less space, it's quite a headache. The use of type char is minimized in this example. Once **ifstream** declares the randomly-chosen name **in_stream** then we can replace cin with in_stream and read 1,000 didits from the file **Table.private** as created or placed in the directory of the current **.cpp** file.

```
char Table_private[1000];

ifstream in_stream;
in_stream.open("Table.private");
for(int a = 0; a < 1000; a++)
{   in_stream >> Table_private[a];
}
in_stream.close();

for(int a = 0; a < 1000; a++)
{       if(Table_private[a] == 48) {Table_private[a] = 0;}
    else if(Table_private[a] == 49) {Table_private[a] = 1;}
    else if(Table_private[a] == 50) {Table_private[a] = 2;}
    else if(Table_private[a] == 51) {Table_private[a] = 3;}
    else if(Table_private[a] == 52) {Table_private[a] = 4;}
    else if(Table_private[a] == 53) {Table_private[a] = 5;}
    else if(Table_private[a] == 54) {Table_private[a] = 6;}
    else if(Table_private[a] == 55) {Table_private[a] = 7;}
    else if(Table_private[a] == 56) {Table_private[a] = 8;}
    else if(Table_private[a] == 57) {Table_private[a] = 9;}
    else        {cout << "File is corrupted!\n"; return 0;}
} //Converts each input to digits, with input safety.

for(int a = 0; a < 1000; a++)
{   cout << int(Table_private[a]);
} //Prints for a visual. Uses int() to convert to digits!

ofstream out_stream;
out_stream.open("out");
for(int a = 0; a < 1000; a++)
{   out_stream << int(Table_private[a]);
} //Writes to file as before except int() converts to digits!
out_stream.close();
```

It is possible to construct large clutter-free technical projects using variables, loops, and arrays as shown here: github.com/compromise-evident