Installation

Mac and Linux already come with a C++ compiler pre-installed, so you don't need to do anything.

If you are on Mac, the built-in C++ compiler is Clang. You may want to install GCC to access GCC-specific features (there are a few handy ones for competitive programming), but this is not required: Installing GCC for Mac Users.

If you are on Windows, I strongly recommend installing WSL: Install WSL | Microsoft Learn. Then, C++ already comes installed with the Linux distribution. Then set up WSL for VS Code: Developing in the Windows Subsystem for Linux with Visual Studio Code. WSL has a separate file system from the Windows file system. To locate your WSL files (e.g. for submitting to online judges), look for the "Linux" folder at the bottom of the navigation pane in File Explorer. If you use the terminal within VS Code connected to WSL, you are effectively using Linux and can pretend that your OS is Linux, so follow the commands for Linux when compiling, running, listing files, etc.

Compilation

Python code can be run immediately. In C++, there is an intermediate step called "compilation," which takes code and turns it into an executable program.

Previously, if your workflow in Python was:

1. Write code
2. Run code
3. Repeat steps 1 and 2 until the code has no bugs

Now in C++ it will be:

1. Write code
2. Compile code
3. Run code
4. Repeat steps 1 to 3 until the code has no bugs

For more details on how to compile and run simple C++ programs, see this: [Tutorial] The command line: how to read input from file without #ifdef and much more - Codeforces.

One thing you will miss when moving from Python to C++ is easy-to-understand error messages. C++ compile errors are usually cryptic and give *way* too much info, especially when you misuse stuff from the standard library. My only practical advice is to ask for help from someone more experienced when you don't understand an error until you get used to it.

Miscellaneous Trivial Differences

In C++, white space has no meaning. Blocks are delineated using curly braces instead of colons and spaces. Conditions in `if`, `for`, and `while` blocks must be enclosed in parentheses, and statements must end with semicolons.

Python	**C++**

`if x == 1:`
`    x += 1`
 	
`while x > 0:`
`    x -= 1`
 	
`def foo():`
`    return 42`
	`if (x == 1) {`
	`    x += 1;`
	`}`
	`while (x > 0) {`
	`    x -= 1;`
	`}`
	
	`int foo() {`
	`    return 42;`
	`}`

The curly braces can be omitted from `if`, `else`, `for`, and `while` (but not function definitions): the next statement, and only the next statement, becomes the content of the block. See the example below. In the C++ code, the indentation misleads you into thinking that both statements are part of the block, but such is not the case: remember that white space has no meaning. The meaning of the C++ and Python programs below are the same: the second statement is not considered part of the `if` block. To avoid problems like this, some style guides recommend always enclosing blocks in curly braces, even if it is just one statement.

Python	**C++**

`if x == 1:`
`    x += 1`
`x += 2`
	`if (x == 1)`
	`    x += 1;`
	`    x += 2;`

You can ignore this recommendation in conditional chains. There is no `elif` in C++. The only reason it exists in Python is that nested conditional chains without `elif` require deep indentation, which can get hard to read. Making the curly braces optional enables avoiding deep indentation in C++. Notice how the “Ugly C++” and “C++” below are exactly the same, just that the curly braces are omitted from all but the last `else` blocks, and whitespaces massaged to better suggest the code’s intent to the reader.

Python	Ugly Python	Ugly C++	C++
`if x == 1:`	`if x == 1:`	`if (x == 1) {`	`if (x == 1) {`
`    x += 1`	`  x += 1`	`  x += 1;`	`    x += 1;`
`elif x == 2:`	`else:`	`} else {`	`} else if (x == 2) {`
`    x += 2`	`  if x == 2:`	`  if (x == 2) {`	`    x += 2;`
`elif x == 3:`	`    x += 2`	`    x += 2;`	`} else if (x == 3) {`
`    x += 3`	`  else:`	`  } else {`	`    x += 3;`
`else:`	`    if x == 3:`	`    if (x == 3) {`	`} else {`
`    x += 4`	`      x += 3`	`      x += 3;`	`    x += 4;`
	`    else:`	`    } else {`	`}`
	`      x += 4`	`      x += 4;`	
		`    }`	
		`  }`	
		`}`	

In C++, single-line comments start with `//` and multi-line comments are enclosed in `/*` and `*/`.

Python	C++
`# This is a single-line comment`	`// This is a single-line comment`
`'''`	`/* This is a multi-line comment.`
`Multi-line comments don't really`	`Of course, starting each line with //`
`exist in Python,`	`also works. */`
`but they are usually faked with`	
`multi-line strings.`	
`'''`	

In C++, boolean values are not capitalized. There are no list comprehensions, no `else` clauses in loops, no `pass` statements (empty blocks are allowed), no unpacking assignments, no argument unpacking, and no keyword arguments. If you don't know what any of these are, you won't need them. There are `break` and `continue` statements though. If you don't know what these are, you'll survive without them. There are conditional expressions and variadic arguments, but the syntax is different and ugly. Again, you'll survive without them.

Standard Input, Standard Output, and Libraries

In C++, reading input and writing output are not part of the language itself. Instead, they need to be imported (included) from the standard library. The syntax for input and output is pretty weird but you'll get used to it.

Notice that in Python, every print statement by default adds a newline character at the end (automatically moving the next printed value to the next line). You can also specify multiple values to print and Python by default automatically puts spaces in between. However, in C++, you need to explicitly specify each space in between and the newline at the end.

Python	**C++ (without using namespace)**

`print(1, 2, 3)`	`#include <iostream>`
	
	`std::cout << 1 << " " << 2 << " " << 3 << std::endl;`

	C++ (using namespace)

	`#include <iostream>`
	`using namespace std;`
	
	`cout << 1 << " " << 2 << " " << 3 << endl;`

In C++, entities from libraries are usually organized into namespaces. If you know what namespaces are in Python, it's not exactly the same concept in C++. In C++, library developers have a way to give "family names" to things in their library, so that they reduce name clashes with other people's libraries. In contests, you only have the standard library, so you don't really need to care about the details, except to note that everything from the standard library must be prefixed with `std::`. You can also put the statement `using namespace std;` at the top of your program. This lets you omit all the `std::` from the rest of the program. This is usually discouraged for real software because it defeats the point of namespaces in the first place: to reduce ambiguity when lots of different libraries from different developers are used. However, in a contest setting, you don't have access to any other libraries anyway so this is fine. For conciseness, all succeeding examples assume there is a `using namespace std;` at the top of the program.

Python	**C++**

`x = int(input())`
`x, y = input().split()`
	`cin >> x;`
	`cin >> x >> y;`

Three things to note about how input works in C++:

1. You don't need to call a split function, and you shouldn't put spaces as operands in `cin` (unlike `cout`). By default, C++ skips all whitespace in the input and tries to read the next *token* into the next variable that appears in the statement. A token is kind of like a word in English but can include digits and special characters too. By default, it is everything until the next whitespace character.
2. You don't specify the type while reading input (the type is specified elsewhere, explained below).
3. If the next token cannot be converted into the type of the next variable, `cin` fails but the program does not crash. Instead, the next variable and all succeeding variables in the current and succeeding statements get unexpected values. There are rules on how the values are determined, but they are not important to know. For practical purposes, you'll know you made a mistake in reading the input when your program is not working and you get weird values upon printing your variables before even doing any processing.

There is a shortcut for including the whole standard library. For contests, this is more convenient since you don't need to remember the import statement for the standard library feature you need. However, it slows down the compilation process a bit. On your own machine (but probably not in onsite contests), you can remove this disadvantage by precompiling the header file:
https://codeforces.com/blog/entry/70957?#comment-619188.

C++

```cpp
#include <bits/stdc++.h>
```

Note: there are some complications in getting this to work on macOS: How to include bits/stdc++.h header file on macOS - Codeforces. Warning: I don't own a Mac and don't know how to make it work.

If you don't `#include <bits/stdc++.h>`, you need to include stuff one by one. For example, since `cin`, `cout`, and `endl` are in the header `iostream`, you need to `#include <iostream>` (only once and you already get everything in it). You can refer to C++ Reference to know exactly which header contains something. For example, at the top of std::cin, std::wcin - cppreference.com, it says that `cin` is "defined in header`<iostream>`". Likewise, std::queue - cppreference.com says that `queue` is "defined in header `<queue>`".

Fast Input & Output · USACO Guide

The Main Function

In Python, you can write statements without any function definitions. In C++, most statements need to be inside function definitions. The exceptions are `using` statements, global variable declarations (explained below), and `#include` (technically not a statement but explaining takes too long). Any other code that you want to run when the program is run, code you'd just "put outside" in Python, must be put inside a function named "main" in C++. So in fact, the examples above are all incomplete because they have statements outside function definitions. But for conciseness, I will continue to present examples that way. Just pretend they are enclosed in the main function.

The main function must return an integer. In contests, 100% of the time, just return zero. Nonzero return values have a special meaning: they tell other programs that use your program that it did not execute successfully. This is useful for real software. But you don't need to care.

Python	**C++**

`print("Hello, World!")`	`#include <bits/stdc++.h>`
	`using namespace std;`
	
	`int main() {`
	`    cout << "Hello, World!" << endl;`
	`    return 0;`
	`}`

Forgetting the return statement does not result in a compile error. In Python, omitting the return statement makes the return value `None`, but in C++, the default return value is whatever garbage happens to be lying in memory at the time (e.g. set by another program that ran before and occupied the same memory). This is called *undefined behavior*. This is a pretty nasty gotcha: your program might run perfectly fine, but in some platforms like Codeforces, it is a runtime error if the return value is nonzero.

Declarations and Types

In C++, all variables must be declared in advance before use. Additionally, the variable must be declared with a type.

Python	**C++**
	`int x;`

`x = 1`	`x = 1;`
`print(x)`	`cout << x << endl;`

The general syntax is `<type> <name>;`

A declaration statement and an assignment statement can be combined into one.

The general syntax is `<type> <name> = <value>;`

However, note that having multiple declaration statements with the same variable name in the same block is a compile error. That's different from assigning the same variable multiple times, which is allowed.

Python	**C++**
`x = 1`	`int x = 1;`
`x = 2`	`x = 2;`
	C++ (wrong)
	`int x = 1;`
	`int x = 2;`

It's possible to declare and assign multiple variables of the same type in one statement by separating them with commas.

C++
`int x, y, z;`
`int a = 1, b = 2, c = 3;`
`int n, m = 5; // only m is assigned, n's value is undefined`

Note that using variables in expressions before they are assigned any value is allowed. The values of the variables will be whatever happens to be lying in memory at the time the program is run. This is another instance of *undefined behavior*. Almost always you'd rather C++ throw an error rather than produce a garbage result and leave you wondering, but that's not how it works. If you see your program outputting garbage, one of the likely bugs is unassigned variables.

As with normal variables, parameter types and the return type need to be specified for function definitions.

Python	**C++**

`def foo(x, y):`
`    return x + y`
	`int foo(int x, int y) {`
	`    return x + y;`
	`}`

The general syntax is

```
<return type> <function name>(<parameters>) {
    <body>
}
```

Where `<parameters>` is a comma-separated list of `<param type> <param name>`

The return type can be `void`, meaning there is no return value. There can be no return statements in a `void` function. The result of a `void` function call cannot be assigned or used in another expression. The `void` type can only be used for return types of functions. Variables cannot have this type.

Unlike in Python, in C++, once a variable has been declared, its type cannot be changed. You're also not allowed to assign a value to a variable if their types don't match. Likewise, all arguments passed to a function must be of the same type as the corresponding parameter in the definition. Mismatched types produce compile errors.

The following are some of the built-in types in C++:

Syntax	**Type**
`int`	32-bit integer
`long long`	64-bit integer
`double`	Double-precision floating-point number
`char`	Character (a single "letter" in a string)
`std::string` (or just `string` if `using namespace std`)	String
`bool`	Boolean

There are many more built-in types than these, but these are the most useful ones and really the only ones you need. Most of these will be explained later.

A variable declared inside any block (including loops and conditionals) only exists in that block. In other words, its *scope* is limited to the block.

C++	**C++ (wrong)**

`int y;`
`cin >> y;`
`int x;`
`if (y == 1) {`
`    x = 1;`
`} else {`
`    x = 2;`
`}`
 	
`cout << x << endl;`
	`int y;`
	`cin >> y;`
	
	`if (y == 1) {`
	`    int x = 1;`
	`} else {`
	`    int x = 2;`
	`    // A totally different variable.`
	`    // This could be named z and the`
	`    // meaning of the program remains`
	`    // unchanged.`
	`}`
	`// compile error: x undeclared`
	`cout << x << endl;`

This means that the same name can be used in different blocks to refer to different variables. This applies to function parameters as well, so different functions can have the same parameter names.

Variables can be declared globally too, in which case, it is accessible from all functions. More precisely, the rule is like this: a variable is accessible in the block (including loops and conditionals) it is declared, and all the blocks inside that block (including loops and conditionals), letting “outside any function definition” be the implicit super-block containing all blocks. Within the same block, there can’t be two variables declared with the same name. However, an inner block can have the same variable name as that of another in an outer block. In this case, what happens is that the inner block variable *shadows* the outer block variable. In other words, all occurrences of the name in the inner block refer to the inner block variable and not the outer block variable. Try running the examples below to understand.

Python	**C++**

`x = 1`
`def foo():`
	`int x = 1;`
	`int foo() {`

```python
    x = 2
    print(x)


foo()
print(x)
```

```cpp
    int x = 2;
    cout << x << endl;
}
int main() {
    foo();
    cout << x << endl;
}
```

Python

```python

x = 2
if True:
    x = 3
    print(x)

print(x)
```

C++

```cpp
int main() {
    int x = 2;
    if (true) {
        int x = 3;
        cout << x << endl;
    }
    cout << x << endl;
}
```

These rules are exactly the same as in Python, except that in C++, conditionals and loops create new blocks, and variables created inside them have limited scope: they shadow variables that were declared prior (if any) and they cease to exist beyond the block. Notice how the Python and C++ programs in the last example aren't really equivalent: they produce different output due to different scoping rules.

This is one place where C++ shines compared to Python. In Python, you can easily mess with variables that meant something else if you're not careful with the names. One annoying bug I occasionally make is accidentally reusing a global or function-scoped variable as a loop variable. In C++, you can still make this bug, but requiring declaring everything and making the scopes of variables much more limited makes it less likely. You can also enable compiler warnings (explained later) to help catch bugs like this.

In both Python and C++, there is a way to access a shadowed global variable. In Python, but not in C++, there is a way to access shadowed local variables. However, declaring a variable that shadows another when you need to use both in the same expression is a bad idea anyway. Just don't do it.

Limited-Precision Numbers

What follows assumes some familiarity with how numbers are represented in computers, that is, in binary. If you aren't familiar, read this first: Binary Numbers | Brilliant Math & Science Wiki.

In Python, integers can grow arbitrarily big. In C++, integers have maximum magnitudes, and there are several integer types that differ in what the maximum is. There are really only two you need to know: `int` and `long long`.

An `int` can have a value in the range [-2^{31}, 2^{31}), for a total of 2^{32} different values. This limit is imposed by the number of bits used to store an `int` in memory – 32 bits implies only 2^{32} distinct representable values, as the mapping between a bit string and a value needs to be one-to-one. To represent both positive and negative numbers, almost all computers today use Two's Complement.

Similarly, a `long long` is stored using 64 bits and can have a value in the range [-2^{63}, 2^{63}). There's a stupid historical reason why it's `long long` rather than just `long` but you don't need to care. It is what it is. A `long` is the same as an `int`.

What happens when the result of a computation would exceed the allowed range of values of your chosen data type? Try this.

C++

```cpp
int x = 2147483648;
cout << x + 1 << endl;
```

In C++, the sum of two positive integers can be a negative integer! This error is called "overflow." More technically, what happens is that the computer does "grade school arithmetic" on your numbers, but in binary, and only writes the least significant 32 (or 64, depending on the type) bits in the result. Two large integers whose most significant bit (MSB) is 0 can sum into an integer whose MSB is 1. But in Two's Complement, a number with MSB 1 is a negative number. Weirder things can happen for multiplication.

Even in Python, there is no such thing as a "number" type. Numbers are either integers or floating-point numbers. This fact is just hidden from you in Python. A floating-point number is kind of like a real number, except that since computers have finite memory, they also have a limit on the number of representable digits. In almost all cases in C++, you should use `double` if you need to compute with fractional values. A `double` can contain about 15 decimal places of precision. This limitation makes computing with them prone to errors. For example, try the following (even in Python):

```
ceil((0.1 + 0.1 + 0.1) / 3 * 10)
```

Because of floating-point limited precision, if there is a way to solve a problem using only integers, even if the most obvious way is to resort to floating-point numbers, you should use integers. Properly working with floating-point numbers requires a separate tutorial. For now, try to avoid them as much as you can (for very simple problems, they might be fine).

In Python, you never cared about the type of the result of an arithmetic operation, but because of potential overflow or precision errors, in C++, you need to care. The result of performing any arithmetic operation:

- On two `int`s is an `int` (so overflow happens instead of the result automatically becoming `long long`)
- On two `long long`s is a `long long` (so overflow can still happen – magnitudes in programming contests are usually, but not always, set to fit in `long long`)
- On two `double`s is a `double`
- On an `int` and a `long long` is a `long long`
- On an `int` and a `double` is a `double`
- On a `long long` and a `double` is a `double`

In general:

1. If the operands are of the same type, the result is also of that type.
2. If the operands are of different types, the result is *widened* into the more "general" of the two types: `double` being more general than `long long` being more general than `int`.

Notice that rule #1 has a pretty unintuitive consequence: the result of dividing two integer-typed numbers is going to be an integer-typed number. If the result has a fractional part, it just gets thrown away. This behavior is called *integer division*. Before you learned about fractions and decimal points, when you did division drills in school, you just outputted the whole-number quotient and the remainder. Integer division is like that "Grade 3 quotient."

C++

```cpp
int x = 5;
cout << x / 2 << endl;
```

Integer division actually exists in Python too, except that you explicitly tell Python you want integer division instead of floating-point division by using double slashes. In C++, there is only one division operator. Whether it performs integer division or floating-point division depends on the types of the operands.

So what if you really want to divide two integers but get a floating-point result? You can explicitly *cast* a value into another type by putting the type in parenthesis and prepending it to the value. The cast takes higher precedence than the arithmetic operator, but you can add more parentheses if you are paranoid like me.

C++

```cpp
int x = 5;
cout << (double) x / 2 << endl;
```

If one of the operands is a literal, you can also just write the floating-point version of the literal:

C++

```cpp
int x = 5;
cout << x / 2.0 << endl;
```

Rule #1 creates another gotcha for bitwise operations:

C++

```cpp
int x = 40;
cout << (1 << x) << endl;
```

To avoid overflow, you can use the `long long` version of the integer literal, which is just the number with an `LL` appended to it.

C++

```cpp
int x = 40;
cout << (1LL << x) << endl;
```

Consistent with Rule #1, you can assign an `int` value to a `long long` or `double` variable and a `long long` to a `double` variable, but the reverse assignments require explicit casts.

C++

```cpp

int x = 1;
long long y = x;
double z = x;
z = y;
y = (long long) z;
x = (int) z;
```

```
x = (int) y;
```

Note that because a “smaller” data type stores fewer bits than a “larger” one, casting may cause a loss of information: casting from a floating-point to an integer type discards the fractional part (no rounding) and casting from a `long long` to an `int` only keeps the least significant 32 bits. This for instance means that casting a positive `long long` can result in a negative `int`.

Why not always use long long - Codeforces
Does long long int affect time complexity in comparison to the use of int? - Codeforces

Vectors and Arrays

The equivalent of a Python list in C++ is a vector. std::vector - cppreference.com has a straightforward example showing how to use it (near the end of the page). Notice how the declaration must also specify in angle brackets the type of elements to be put inside the vector, and that there can be only one such type. This type can be any other valid type, `int`, `long long`, `string`, or even `vector`! Just remember to specify the innermost non-vector type. Here’s an example with a vector of vectors. The for-loops will be explained later.

Python

```python
n_rows = 3
n_cols = 4



table = [
    [0 for j in range(n_cols)]
    for i in range(n_rows)
]

for i in range(n_rows):
    for j in range(n_cols):
        table[i][j] = i * j


for row in table:
    for cell in row:
        print(cell, end=' ')
    print()
```

C++

```cpp
int n_rows = 3;
int n_cols = 4;

// You can write this in one line.
// It's just crowded here.
vector<vector<int>> table(
    n_rows,
    vector<int>(n_cols, 0)
);

for (int i = 0; i < n_rows; i++) {
    for (int j = 0; j < n_cols; j++) {
        table[i][j] = i * j;
    }
}
for (vector<int>& row : table) {
    for (int cell : row) {
        cout << cell << ' ';
    }
    cout << endl;
```

```python
table.append([])
table[-1].append(1)

for row in table:
    for cell in row:
        print(cell, end=' ')
    print()
```

```cpp
}

table.push_back(vector<int>());
table.back().push_back(1);

for (vector<int>& row : table) {
    for (int cell : row) {
        cout << cell << ' ';
    }
    cout << endl;
}
```

C++ provides a more low-level type of collection of items called an *array*. The difference with a vector is that you must provide its size in advance, and this size is fixed throughout the lifetime of the program. Unlike vectors, arrays do not smartly grow and shrink as elements are added and removed. In fact, there is no concept of adding or removing elements with arrays at all. Instead, a fixed-size number of elements are just there all the time. The initial values in the array depend on where it is declared. If it is declared inside a function, it will initially contain garbage values. If it is declared globally, the values are initialized to zero (or the reasonable “zero” value for the contained type – for example, the empty string for arrays of strings or the empty vector for arrays of vectors).

C++

```cpp
#include <bits/stdc++.h>
using namespace std;

const int MAX_NUM_ROWS = 10;
const int MAX_NUM_COLUMNS = 10;
int table[MAX_NUM_ROWS][MAX_NUM_COLUMNS];
int main() {
    int num_rows, num_columns;
    cin >> num_rows >> num_columns;
    for (int i = 0; i < num_rows; i++) {
        for (int j = 0; j < num_columns; j++) {
            table[i][j] = i * j;
        }
    }
    for (int i = 0; i < num_rows; i++) {
        for (int j = 0; j < num_columns; j++) {
            cout << table[i][j] << ' ';
        }
        cout << endl;
    }
}
```

Notice how sizes of arrays must be a literal or a `const` (for “constant”). In other words, they must already be known *at compile time*. They cannot be some non-`const` variable whose value will only be known *at run time*, such as given as input. For this reason, when using arrays, you typically specify their sizes to be the maximum possible you expect based on the constraints specified in the problem, but then access only a specific portion of them on specific input sizes. Also, notice how unlike with a vector of vectors, with an array of arrays, each inner array must be of the same, unchangeable size. For fewer or more dimensions, just reduce or add square brackets in the declaration.

Nowadays, the only place where I use arrays in competitive programming is when writing DP solutions (don’t worry if you don’t know what that is yet). The only reason is that the syntax for declaring multi-dimensional vectors is fugly, especially when you need to go to 4, 5, or 6 dimensions, which is pretty normal for DP. For everything else, I just use vectors. I find myself making fewer errors with them, and while using vectors can be slower, I find that in competitive programming nowadays, the speed difference usually doesn’t matter or can be made insignificant (but this is better explained in a separate lesson, going in-depth into how Python lists and C++ vectors actually work). You will definitely encounter other people’s code using arrays a lot more, so you should know about them.

Since I can’t use local variables to specify the sizes anyway, whenever I use arrays, I just declare them globally. This avoids the garbage initial value problem and more:

- Fugly syntax for functions with arrays, especially multi-dimensional arrays, as parameters
- For very technical reasons, the total amount of memory requested together with a function call, which includes all the memory needed for function parameters, local non-array variables, and locally-declared arrays cannot be too big. Otherwise, the program can crash with what’s called a *stack overflow error*. This severely limits the size a locally-declared array can have.

For-Loops

There are two kinds of for-loops in C++. The first one is the same as the Python for-loop, only with a slightly different syntax:

Python	**C++**

`for x in a:`
`    s += x`
	`for (int x : a) {`
	`    s += x;`
	`}`

Notice that the type of the loop variable must be specified. It should match the type of the items in the container being iterated over (remember, only one such type can exist). Unlike in Python, the loop variable specification is also a variable declaration, and the variable's scope is limited to the for-loop block. Also, you cannot use this type of for-loop on arrays.

Notice how when iterating over a vector of vectors, we add an ampersand at the end of `vector<int>`.

Python	**C++**

`for row in table:`
`    for cell in row:`
`        print(cell, end=' ')`
`    print()`
	`for (vector<int>& row : table) {`
	`    for (int cell : row) {`
	`        cout << cell << ' ';`
	`    }`
	`    cout << endl;`
	`}`

Without the ampersand, C++ will make a copy of one of the inner vectors at each iteration. This slows the program down and is almost always not what you want. Also, if it is a copy, then any changes, such as `push_back`s, made to `row` inside the loop do not change `table`. It would not be equivalent to the Python version, where appending to `row` inside the loop would change `table`. This is a slight lie. In fact, in the C++ version, re-assigning (as in using the `=` operator) `row` is going to change `table` as well. On the other hand, in the Python version, it won't. You almost never want to re-assign anyway so just don't, and you can pretend it's exactly like Python.

Finally, you can append items to a Python list while iterating over it and it will work as expected, but the same is not true for C++ vectors.

The second type of C++ loop is a kind of hack for the lack of a `range` function.

Python	**C++**	**C++ (equivalent while loop)**

`for i in range(n):`
`    s += i`
	`for (int i = 0; i < n; i += 1) {`
	`    s += i;`
		`{`
		`    int i = 0;`

	`}`	`    while (i < n) {`
		`        s += i;`
		`        i += 1;`
		`    }`
		`}`

Notice that even in Python, you can do away with the `range` function using while loops. We like using for-loops anyway because it matches the way we think about algorithms better. In C++, “for all integers in some range” is achieved through a thin layer of “syntactic sugar” over the while-loop. The for-loop and the while-loop above are literally compiled into the same thing. In general,

```
for (initialization; condition; update) {
    loop body
}
```

is equivalent to

```
{
    initialization;
    while (condition) {
        loop body
        update
    }
}
```

You can use this equivalence to understand exactly how the for-loop works, but it shouldn’t be the way you think about it. This extra syntax is really trying to suggest the “for integers in range” Python-style for-loop, and this is how you’ll most often use it. Notice though that since `initialization`, `condition`, and `update` can be arbitrary C++ statements, you’re allowed to do weird things that you can’t normally do with Python for-loops. Most of these weird things are discouraged, but there are a few handy and reasonable ones (plus how to do a few things that you can also do in Python):

- Subtract instead of add in the update statement to loop in reverse.
- Change the amount added to the loop counter to simulate “skip-by” in Python.
- Make the inequality in the condition nonstrict to mean “loop until end inclusive” (rather than doing +1 or -1). This is particularly nice for looping in reverse.
- Change the update statement from addition into multiplication to go through all powers of a number.
- Make the condition `i * i <= n` to go up to square root of `n`. (The effect of the extra multiplication on the running time is usually negligible. On the other hand, computing square roots is a floating-point operation, which may yield imprecise results.)

As before, the loop counter's scope is limited to the for-loop block (if the counter is declared inside the parenthesis): notice how the while-loop equivalent needs to have braces around it to be truly equivalent to the for-loop.

By the way, because incrementing by 1 is so common, there is a shortcut for it in C++. Double plus just means "increment by 1." This is where the name of the language comes from: it's the C programming language with an increment. You will see for-loops more commonly written like the one on the right below instead of the left one.

Ugly C++	**C++**

`for (int i = 0; i < n; i += 1) {`
`    s += i;`
`}`
	`for (int i = 0; i < n; i++) {`
	`    s += i;`
	`}`

The double plus can be put before or after the variable to increment. There is a difference in meaning between the two but this difference only manifests itself if you use the increment expression within a larger expression. Yes, you can do that. And yes, doing that is a bad idea anyway, so you can just not care about the difference.

Strings and Characters

C++ makes a distinction between strings and individual characters. Strings must be enclosed in double quotes while characters must be enclosed in single quotes. This is unlike Python where everything is just strings and either single or double quotes may be used. Indexing a string in C++ returns a character, whereas in Python, indexing a string returns another string.

In Python, there are `ord` and `chr` functions for performing "arithmetic with letters." In C++, you can directly apply arithmetic operations on characters: the operands will be widened into their numeric representation.

Note that if you need to print the result of the arithmetic operation as a character, you need to cast the result back to `char`, as the resulting type has widened.

Python	**C++**

`print(`
`    f'{i}th letter:',`
	`cout`
	`    << i << "th letter: "`

`chr(ord('A') + i)`	`<< (char)('A' + i)`
`)`	`<< endl;`

Technically, `std::string` isn't a built-in type in C++ (notice the `std`). That's because a string can be thought of as a `char` array and you can just have `char[]` in your code. That's how it was done in C. In C++, use `std::string` anyway for convenience. The practical thing to note here is that string literals are actually `char` arrays and applying the `+` operation on two string literals does not give the result you expect. See this for more details: Concatenate two string literals - c++. You never need to do this anyway. You can however append string literals (and individual characters) to a variable declared with type `std::string` using the `+=` operator: std::basic_string<CharT,Traits,Allocator>::operator+= - cppreference.com.

Making C++ More Programmer-Friendly

C++ is an ugly language for competitive programming. One reason: it gives the programmer too much control and has way too many features which are irrelevant to competitive programming. Another reason: it prioritizes efficiency over everything else. Efficiency is good, but C++ gains it at the expense of ease of programming. It'll assume the programmer is smart. By default, it does not do the safety checks (i.e. index out of bounds or uninitialized variables) or error tracebacks (showing which lines of the program caused a crash) you expect sane programming languages to have, because those cost computing time. In C++, the sum of two positive numbers can be a negative number. A C++ program will happily produce garbage outputs instead of crashing even when it does a nonsense computation. When it does crash, you get a big fat "Segmentation Fault" with no helpful info on what exactly went wrong.

Unfortunately, C++ has become sort of the standard in competitive programming. Simply because C++ exists, is fairly popular, and is one of the fastest languages available, problems today are designed so that solutions with suboptimal Big-O written in C++ don't pass. Otherwise, C++ programmers have an unfair advantage. As a result, problem constraints and time limits are often calibrated for C++, and it happens that even solutions with correct Big-O in Python get TLE. So we have no choice but to learn C++ and deal with its ugliness.

We've somewhat dealt with the feature bloat issue by only learning the important bits. We can deal with the safety checks issue too: it is possible to add *compilation flags* to the compilation command to turn on some checks, at the cost of making the compilation time longer and the compiled program slower. This is fine. When you submit your code to an online judge, it isn't compiled with these flags so retains its speed. You can just accept the extra slowness while testing. Often, you won't even notice the slowness anyway.

For more details see: Catching silly mistakes with GCC - Codeforces. And don't miss this helpful comment on that blog.

Additional References

Robin Yu’s C++ tutorial with exercises for practice: C++ for Competitive Programming