NOI.PH-YTP Junior Track: C++ Programming 1

A technical introduction to C++

Aldrich Ellis Asuncion

Contents

1 Introduction

This module introduces you to:

- the C++ standards
- how to use the command line in Linux
- how to compile and run C++ programs
- variables and basic data types in C++
- input and output in C++

I assume those of you reading this already know how to program, and are familiar with variables, data types, and input/output in either Python or C++. This module is *not* a typical beginner's introduction to programming. We will go through many details about how C++ works, so that you have a more accurate mental model of what exactly things like "declaring a variable" do. This will also help you understand some C++ syntax and conventions we'll encounter later on.

On the other hand, the goal of this module is still to equip you with practical C++ knowledge which can apply in competitive programming. We'll still be omitting a lot of technical details. Some of the things said here will not be strictly correct, or somewhat simplified compared to actual reality. This is done to make the explanations simpler.

Sections 2.3 to 2.6 are taken from the existing NOI.PH Training handout "Technical Stuff 1: Technical Stuff (and C++ Garbage)", prepared by Payton Yao, Kevin Atienza, and Vernon Gutierrez, with some minor modifications.

2 Compiling and running C++ programs

2.1 Programming languages and their implementations

Programming languages are standards. Much like how human languages have standard vocabulary and grammar, described by dictionaries and grammar rules, programming languages are also defined by standards. (Of course, human languages are far more fluid than programming languages.)

- A working group in the International Organization for Standardization (ISO) work on and publish the ISO C++ Standard.
- Python is defined by the Python Language Reference.

Remark 2.1. You are not meant to read the language standards in order to *learn how to program*. The language standards are there for people interested in making compilers and interpreters for a given language, and for those who want to investigate *really niche* behavior in a language.

The C programming language was created in 1972. C++ was then created in the 80s and 90s as an enhanced version of C. The first ever *language standard* for C++ was C++98 (released in 1998), with a minor standard update, C++03 (released in 2003). C++03 was then the major standard for a while. A lot of older tutorials and books out which you might see might still teach C++03.

(From this point on, I assume you understand that the C++XX standard would be released in the year 20XX. So I don't need to keep repeating myself.)

The next version, C++11 was a major revision. C++11 overhauled how many things are meant to be done in C++, and personally, I view C++11 onwards as an entirely different programming language from C++03 that just so happens to share a lot of syntax. Because of all the changes, C++11 is now the new normal. When you hear people say "modern C++", this typically refers to C++11 or later. And since then, there's been a new version of the standard out every 3 years, so C++14, C++17, C++20, ...

Throughout these handouts, I'll assume that we're working with **C++17**, as that's the default in many compilers at the time of writing. A lot of the features added in later versions of the C++ standard aren't relevant to competitive programming. But in case there are some useful syntax tricks or library functions available in later versions, I'll make sure to mention the versions for C++20 or later.

Remark 2.2. For those curious, a quick cheat sheet of version differences can be found here: `https://github.com/AnthonyCalandra/modern-cpp-features`.

Informally, programming is the act of telling a computer what to do. Humans write instructions in the form of **source code** adhering to a programming language, and these instructions are then executed by a computer. A **programming language implementation** is a program that takes source code, and allows for the instructions written in that language to be executed. There are two general types of programming language implementations, which can be informally described as:

Interpreters. Interpreters read code and execute the code.

Compilers. Compilers read code and translate it into a new form, which can then be executed.

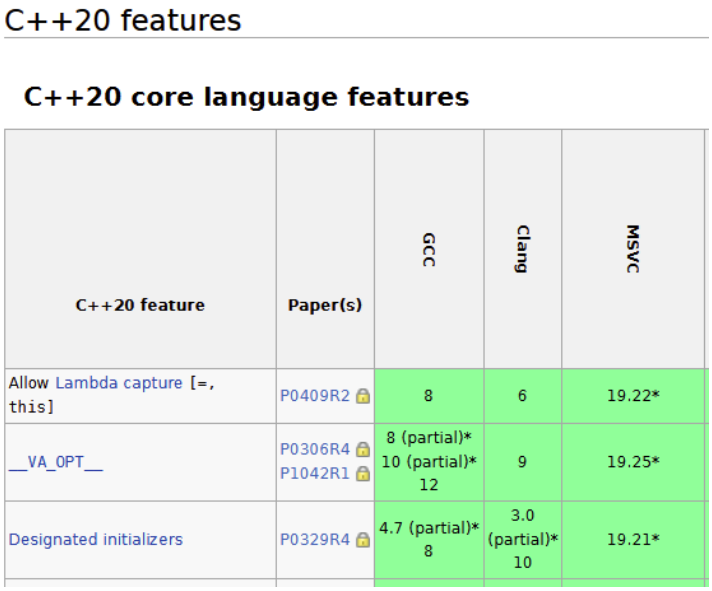
These two types of programming language implementations are not mutually exclusive. Some implementations do a bit of both (The reference implementation for Python has both a compilation step and a interpretation step.) For C++, it's quite simple: C++ is commonly implemented using a compiler. There are three major C++ compilers out there:

- The **GNU Compiler Collection (GCC)** is a collection of compilers, which includes support for C++. The C compiler in this collection is called GCC, while the C++ compiler in this collection is called **G++**.
- **clang** is a C++ compiler part of the LLVM project. (What the LLVM project is, is well outside the scope of these handouts.)
- **Microsoft Visual C++ (MSVC)** is a proprietary C++ compiler released and maintained by Microsoft. Used in some industries, like game development, but is not common use in competitive programming.

The standard C++ compiler you should use is GCC. There are some GCC-specific features which have widespread use in competitive programming. You'll find that online judges typically support the GCC compiler, while having limited support for other compilers.

Remark 2.3. A list of all the programming languages supported by CodeForces, alongside some technical details, can be found at `https://codeforces.com/topic/121739/en14`. Make sure to check the version list at the bottom of the page to find the latest version of this post.

Knowing what compiler you have is useful outside of competitive programming, because not all language features of a standard are supported by all versions of a compiler. For example, Figure 1 shows a portion of the cpp reference page on compiler support. Notice that complete and bug-free support for an entire language standard is usually not done all at once. Some features might be implemented in a later version of a compiler compared to other features.



C++20 features

C++20 core language features

C++20 feature	Paper(s)	GCC	Clang	MSVC
Allow Lambda capture [=, this]	P0409R2	8	6	19.22*
__VA_OPT__	P0306R4 P1042R1	8 (partial)* 10 (partial)* 12	9	19.25*
Designated initializers	P0329R4	4.7 (partial)* 8	3.0 (partial)* 10	19.21*

Figure 1: An example of how some C++ features are supported by some compiler versions only.

As an example, as of September 2024, according to https://gcc.gnu.org/projects/cxx-status.html,

- GCC uses C++17 as the default, and supports a large portion of the C++17 standard (with some very minor exceptions)
- GCC has *experimental* support for the C++20 and C++23 standards, as well as the upcoming C++26 standard.

However, for competitive programming, you can largely ignore many of the new features in C++20 or later. Consider all the above information as a fun fact that might be useful later if you decide to go deeper in the C++ rabbit hole in your future studies or career. But for now, these handouts will assume the C++17 standard.

2.2 References for C++

Please use the documentation at en.cppreference.com as a language reference. Much of the C++ material I will be providing in the handouts are based on the following references.

- The C++ tutorial hosted at learncpp.com.
- Federico Busato's open-access modern C++ programming course.

An easier tutorial than the ones above is the one hosted at cplusplus.com. This tutorial is more informal and is not as updated in terms of language features in C++14 and later. This can be useful if you want an alternate, easier explanation of a topic to accompany the NOI.PH-YTP handouts.

2.3 The command line

You already know how to code. For some of you, you might have used an online compiler, compiling and debugging your programs through that platform. For others, you might have used an Integrated Development Environment (IDE) like CodeBlocks or DevC++.[1] But in order to be successful and fast for contests, you need to be used to doing everything through the **command line** which is what we'll introduce here.

You might be asking: "Why?" If everything can be done on an IDE, why bother learning to use the command line? There are two answers, both equally valid. The first and more practical one is that you may not have the exact IDE you're comfortable with available at all times. However, every computer will have some form of command line available, and every computer meant for programmers will have the command line tools set up. The second reason is about understanding. IDEs give way too much magic and sometimes we miss all the small things that happen behind the scenes.

Note that this is only a short introduction to the command line. Mastering its use takes years of experience to do extremely intricate things, but basic usage allowing you to navigate the file system, compile code, and run programs you can learn in the space of an hour or so.

At this point, it's important to note that there are some major differences between using the command line on Windows and on Linux. Since *NOI.PH onsite rounds and IOI all use Linux*, we will only describe the **Linux** terminal. For Mac or Windows users, you need to take the following extra steps:

- For **Mac** users, you can mostly follow along since the differences between Mac and Linux are small. However, note that the built-in C++ compiler is Clang. You may want to install GCC to access GCC-specific features[2] (there are a few handy ones for competitive programming), but this is not required. Note that the IOI uses GCC.
- If you are on **Windows**, I strongly recommend installing WSL.[3] WSL allows you to pretend that your computer is running Linux even if you're on Windows. Then, C++ already comes installed with the Linux distribution. Then set up WSL for VS Code.[4] (VS Code is a text editor. We recommend using it.) 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.

As a historical note, there was once an operating system called Unix developed at Bell Labs in the 1970s. It's this original operating system whose design philosophy Linux and Mac copied. Because they come from similar roots, much of the core command line tools are the same with some small differences in some of the options they can be given. Advanced users will also note a striking similarity in the way system files are organized, for example Linux has `/home/` where Mac has `/Users/`.

[1] IDEs are these huge pieces of software that combine multiple tools in one, allowing you to edit source code, compile programs, debug them, and more, all in one convenient program.

[2] http://cs.millersville.edu/~gzoppetti/InstallingGccMac.html

[3] https://learn.microsoft.com/en-us/windows/wsl/install

[4] https://code.visualstudio.com/docs/remote/wsl

2.4 Basic commands

The first step is to learn how to open the command line terminal. If you're on Ubuntu, open the Gnome Terminal. (And if you're not on Ubuntu, you probably already know what you're doing anyway.) For Mac users, you can use the built-in Terminal, just search for it using Spotlight. You can also opt to download something called iTerm2 which is a much-improved version of Terminal with lots of useful new features.

The important commands are as follows:

- `cd` - change directory. Use as `cd ~/Downloads`.
- `ls` - list files in current directory. Use as `ls`
- `cp` - copy a file. Use as `cp source.txt destination.txt`
- `mv` - move a file from one place to another. Use as `mv original.txt new.txt`. Interestingly, this is also the command used to rename files.
- `rm` - delete a file. Use as `rm a.txt`
- `mkdir` - make a new directory. Use as `mkdir new_directory`
- `rmdir` - delete an empty directory. Use as `rmdir new_directory`

We encourage you to try these commands out. Your computer won't break if you type a command wrong. (Just make sure not to delete or overwrite your important files!) You can hit the tab key to auto-complete.

You may have noticed that we did `cd ~/Downloads`. The symbol `~` is a shorthand for `/home/user` in Linux (`/Users/user` in Mac) where `user` is the username of your computer account.[5] (For example, it's `/home/kevin` in my computer.) We'll have a bigger discussion on Unix commands and the environment in the future, but this is enough for now.

You can read more about the Unix command line at:
http://www.ee.surrey.ac.uk/Teaching/Unix/

[5] To check that this is true, enter `echo ~` in the terminal. (`echo` just prints something, e.g., `echo "Hello World!"`)

2.5 Compiling and executing programs

2.5.1 Compiling programs

We will assume that the GCC compiler is installed. Now that you're all done setting up, it's time to start compiling! If you save your source code file as `source.cpp`, you can compile it using the command `g++ source.cpp`.

By default it saves the resulting executable as "`a.out`". You can change the name for the resulting output file by adding `-o filename` to the command. For example, if my source code is "`example.cpp`" and I wanted to save the output executable as "`executable`" I should then type the command `g++ example.cpp -o executable`. Note that unlike Windows, you don't need to add an extension such as `.exe`; Linux knows that a file is executable using other means.

Try compiling a sample Hello World program this way before moving on, such as the one below.

```
#include <iostream>
int main() {
    std::cout << "Hello World!" << std::endl;
}
```

If you're having an error like `g++: error: source.cpp: No such file or directory`, please ensure you're in the right directory first.

> **Remark 2.4.** You can specify the C++ standard to be used, by adding `-std=c++XX` to the compile command, where `XX` indicate the version. For example, `g++ -std=c++14 source.cpp` compiles using the C++14 standard.

2.5.2 Running executable files

Now that you've compiled your program, it's time to run it! The command to run a program is `./program_name`. So if your program is saved as `a.out`, you should type "`./a.out`", and if it's saved as `executable`, you should type "`./executable`". Note that you have to be on the folder containing the compiled executable.

2.5.3 Killing running programs

In case you accidentally run code that never ends or you just want it to die for some other reason, what can you do? It's gonna block your command line and stop you from typing other commands, so you will want to kill it. You can opt to kill it using the System Monitor[6], but there's a much easier way: Just press `Ctrl + C`. This will kill the program that's hogging up the command line.

Try compiling and running the following program, then kill it just to try out `Ctrl + C`.

[6] the Linux equivalent of Windows' Task Manager

```cpp
#include <iostream>
using namespace std;
int main() {
    while (true) {
        cout << "wheee" << endl;
    }
}
```

2.6 Input redirection

2.6.1 Saving output to a file

There are cases when we want to save the output of our program to a file. The simple way is to highlight and copy things from the command line, paste it in your text editor, and save the file. But there's an easier and more precise way to do it: Use the `>` symbol.

To be precise, if we want to save the output of the program executable to a file called `output.txt`, we type the command `./executable > output.txt`

Try it yourself! Of course, you can change the output file name to anything you want and not just `output.txt`

It's important to note at this point that **this will completely erase the contents of the file you're saving into**, even if your program hasn't printed anything out yet. So be careful!

2.6.2 Reading input from a file

Another common case is when we want our program to read input from some file instead of us having to type the values manually. Again, there's a simple way to do that! Use the `<` symbol.

If we want to make the program executable read from a file called `input.txt`, we type the command `./executable < input.txt`

This is extremely useful for debugging your solutions for contests, especially for problems with large-ish input like graph problems.

2.6.3 Combining the two

There are cases where we have input from a file and we want to save the output to another file. For example, in old versions of Google Code Jam, input files are downloaded from their website and you have to upload the correct output file. Or maybe you just want to store the output for a pre-determined input file into a different file, presumably to inspect later. It turns out that you can do this by combining `<` and `>`.

An example command that combines them is `./executable < input.txt > output.txt`

Note that the command `./executable > output.txt < input.txt` is just as valid and does exactly the same thing.

3 Basic C++ programming concepts, basic types

The following section is an adapted version of the tutorial found at https://cplusplus.com, incorporating technical details from https://www.learncpp.com/, as well as competitive programming-specific information.

3.1 Structure of a program

3.1.1 Comments, headers, namespaces

Let's dissect the following "Hello World!" program. We'll discuss the boring (but necessary) stuff first.

```
1  /*
2      This is a multi-line comment
3      These lines are ignored
4  */
5  #include <iostream>
6
7  int main()
8  {
9      std::cout << "Hello World!" << std::endl; // Gives output
10     std::cout << "I'm a C++ program" << std::endl;
11     // I guess it's not just Hello World
12 }
```

- Lines 9 and 11 demonstrate **single-line comments**. Two forward slashes `//` indicate that the rest of the line is a comment. Comments are ignored by the compiler and are meant to give human-readable notes.
- Lines 1 to 4 demonstrate **multi-line comments**. Any text between `/*` and `*/` is a comment, even if the text spans multiple lines.
- Line 5 is an example of a **preprocessor directive**. Preprocessor directives start with `#`. These are one of the first things handled as part of the compilation process. The `#include` preprocessor directive tells the compiler to copy in the `<iostream>` header file, allowing us to use all the things contained in there, such as:
  - ▷ `std::cin`, used for input
  - ▷ `std::cout`, used for output
  - ▷ `std::endl`, used for outputting a newline and flushing the stream (more on this later)

There are many **header files** available in C++, which you can see on the next page. Including a header file allows access to part of the C++ standard library. We'll only be using some of these in competitive programming.

Figure 2: All the header files available at the time of writing. Notice that some headers are only available with some C++ standards (e.g. C++20 or later)

Before we proceed, let's discuss some common ways competitive programmers shorten their code.

- The preprocessor directive `#include <bits/stdc++.h>` includes every header file (with respect to the C++ standard version you are using). This header file is only available on the GCC compiler.

  It is okay to use this for contests, but for real-world programming, it is easier to organize code if you only include the headers you need.

- Every element made available in the C++ standard library has a name which starts with `std::`. We say that names beginning with `std::` **belong in the `std` namespace.** By adding the statement `using namespace std;`, we allows all elements in the `std` namespace to be accessed in an unqualified manner (without the `std::` prefix).

  While this makes accessing parts of the standard library easier, it means we cannot name our variables in ways that conflict with elements of the standard library. For example, there is an object in the standard library called `std::vector`. If we use `using namespace std;`, then we can access this by just typing `vector`, but now we can't name a variable "vector", because its name will conflict with the "vector" in the standard library.

Using these two tricks, and by removing the comments, we can write the previous program as follows.

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    cout << "Hello World!" << endl;
    cout << "I'm a C++ program" << endl;
}
```

You are free to use these tricks in your own code, as this training is for competitive programming. However, for the sake of clarity, I will often not use these tricks when writing the handouts.

3.1.2 Statements

Let's look at the following program. We've already discussed comments, so I've taken out all the comments.

```
#include <iostream>
int main()
{
    std::cout << "Hello World!" << std::endl;
    std::cout << "I'm a C++ program" << std::endl;
}
```

Lines 4 and 5 are **statements**. Statements in C++ always end with a semicolon `;`. You technically don't need to place statements on a new line, for example, the statements

```
int x = 5;
int y = x + 2;
std::cout << x * y << std::endl;
```

could be written as

```
int x = 5; int y = x + 2; std::cout << x * y << std::endl;
```

and this would be completely valid C++ code. However, for readability, it is preferable to place statements on separate lines.

The statements in this program display text on the screen using `std::cout`. `cout` stands for *character output*, and values to be outputted are inserted using the insertion operator `<<`.

We'll discuss input and output properly later.

3.1.3 The main function

Okay, we're almost done with the boring stuff. The last thing to talk about is the **main function**. A function is a group of statements which are run together, and the function with the name `main` has the special role of being the first function that is executed when a program is run. In fancy words, `main` is the **entry point** of the program.

We will discuss functions in general in a later handout (they're quite technical, and knowing the details is important). For now we'll focus on the main function.

```
1  #include <iostream>
2  int main()
3  {
4      std::cout << "Hello World!" << std::endl;
5      std::cout << "I'm a C++ program" << std::endl;
6  }
```

Here, line 2 initiates the declaration of a function named `main`. The statements in the function are enclosed in curly braces `{}`, seen on lines 3 and 6.

Okay, so actually, this main function here employs a couple of shortcuts. To properly discuss what's going on here, let's look at what this should look like if I took no shortcuts.

```
1  #include <iostream>
2  int main(int argc, char *argv[])
3  {
4      std::cout << "Hello World!" << std::endl;
5      std::cout << "I'm a C++ program" << std::endl;
6      return 0;
7  }
```

First, a standard convention[7] for any program is to return the value zero if it executed successfully, and some non-zero value if it didn't. The value of the non-zero value typically indicates the type of error that was encountered.

As this is a common convention, there is a special rule in place for the main function: If the end of the main function is reached without encountering a return statement, the effect is that of executing `return 0;`. So we can safely omit `return 0;` for the main function (but not for any other function!)

Next, to explain `int argc, char *argv[]`. Remember how when compiling or running a program, you can add a bunch of extra options (called **command-line arguments**) to each of the commands?

```
1  g++ source.cpp -std=c++17 -o myprogram
2  ./myprogram < input.txt > output.txt
```

[7] See, for example, https://stackoverflow.com/a/2371023

Well, by adding `int argc, char *argv[]`, you can actually detect and use additional options like these in your program. For example, if I write the following program:

```cpp
#include <iostream>
int main(int argc, char *argv[])
{
    std::cout << "You put " << argc << " arguments" << std::endl;
    std::cout << argv[0] << std::endl;
    std::cout << argv[1] << std::endl;
    std::cout << argv[2] << std::endl;
    std::cout << argv[3] << std::endl;
}
```

And then compile and run it with

```
g++ source.cpp -o myprogram
./myprogram correct horse battery staple
```

You'll get the following output:

```
You put 5 arguments
./myprogram
correct
horse
battery
```

Notice that `argc` will contain the number of arguments you put in, including the program invocation itself. And `argv` contains each of the arguments, so you can use them to dictate logic in your program.

The programs we write for competitive programming do not rely on command-line arguments. C++ permits us to omit `int argc, char *argv[]` in the definition of the main function if our program does not rely on command-line arguments.

Hence, from now on, we will work with main functions where we omit the final return statement and don't rely on command-line arguments. The most basic C++ program (which does nothing) is:

```cpp
int main()
{
    // some code here
}
```

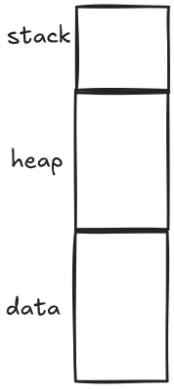
3.2 Variable declaration and initialization

3.2.1 A mental model for memory

Your computer has RAM (random-access memory), which contains all the data your computer is actively using at a given point in time. Whenever you run a program, your operating system gives that program its own part of RAM, which we call program memory.

For desktop computers running Linux, as in the case of competitive programming, we can think of program memory as divided into a few different areas, called segments.

Simplifying a bit, a program makes use of three segments: named **data**, **heap**, and **stack**. We'll use the following simple diagram to illustrate these parts of the program memory.



The only thing you need to know for now is that stack memory has limited size, while the data segment and the heap can hold larger amounts of data.

Within each segment, memory is organized into addresses, kind of like this. You don't need to be able to read the numbers. I just put legit-looking visuals so we feel like hackermans here.

addresses	memory
0x7fffb04da37c	57 68 79 20
0x7fffb04da380	61 72 65 20
0x7fffb04da384	79 6F 75 20
0x7fffb04da388	72 65 61 64
0x7fffb04da38c	69 6E 67 20
0x7fffb04da390	74 68 69 73
0x7fffb04da394	3F 20 47 61
0x7fffb04da398	6C 69 6E 67
0x7fffb04da39c	20 6D 6F 20
0x7fffb04da3A0	6E 61 6D 61
0x7fffb04da3A4	6E 2E 2E 2E

Addresses are the locations of data in memory. We can say, using this picture for example, that the data `61 72 65 20` is located at the address `0x7fffb04da380`. While us humans are used to referring to parts of data using things like variable names, your computer operates using addresses like these. All of your computer's data has a location[8] in memory.

[8] We will not distinguish between physical and virtual addresses because that's another rabbit hole oh no

In C++, we declare variables using the following syntax. ("identifier" is just a fancy word for "name")

```
<data type> <identifier>;
```

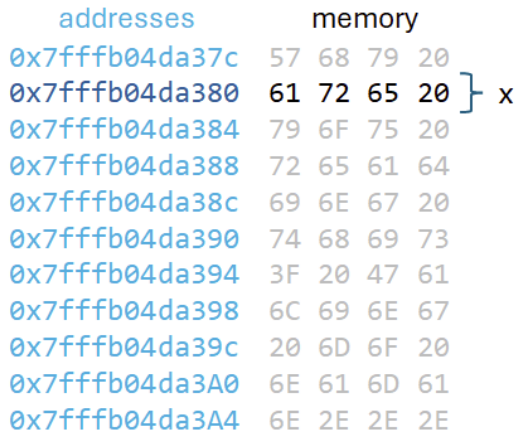
For example:

```
int x;      // declares an integer named x
float eps;  // declares a float named eps
```

What this does is allocate (or reserve) some part of memory for the variable in question. For example, when I issue the statement

```
int x;
```

I am telling the computer to go find some free space in memory that is big enough to hold an integer, and designate that to be `x`. So maybe our memory picture will now look like this.



Notice that we don't actually clear out the memory for `x`. When we declare a variable like this, the variable is **uninitialized**. It exists now, but it wasn't given a value. In some cases, the memory *is* cleared out, but we can't assume that.

If you try running this, you might get garbage or you might get 0, or your program might crash.

```
#include <iostream>
int main()
{
    int x;
    std::cout << x << std::endl;
}
```

There is one exception though. V**ariables declared globally (outside any function) are value-initialized.** This means the variable is initialized to zero (or empty, if that's more appropriate for a given type). Compare the following two programs.

```
int main() {
    int x; // x is uninitialized
}
```

```
int x;  // x is initialized to 0
int main() {
}
```

3.2.2 Initializing variables

We always want to ensure that our variables are initialized before using them. If we want to "clear out" the memory for `x` and ensure that it is 0, we explicitly do so.

```
#include <iostream>
int main()
{
    int x = 0;
    std::cout << x << std::endl;
}
```

In modern C++, it is recommended to use an alternate syntax for initialization, called **list initialization**. Instead of using the assignment operator =, we provide the initial value of the variable in curly braces, like this:

```
int x {42};     // x is an integer list-initialized to 42
```

We can try it out in an example program.

```
#include <iostream>
int main()
{
    int x {42};
    std::cout << x << std::endl;
}
```

Sometimes, we don't want to initialize our variable to a particular value. Maybe we're just declaring a variable to hold input, in which case the initial value doesn't matter. Since it's bad practice (and possibly error-prone) to leave variables in an uninitialized state, we can use **value initialization**. We simply add curly braces after the variable name, but don't provide a value, like this.

```
int x {};
```

In most cases, value initialization will initialize the variable to zero (or empty, if that's more appropriate for a given type). In such cases where zeroing occurs, this is called zero-initialization.

```
#include <iostream>
int main()
{
    int x {};
    std::cin >> x; // read input and store the result in x
    std::cout << x << std::endl;
}
```

This more clearly communicates that you are declaring a variable but do not care about its initial value, while still avoiding the use of uninitialized variables.

But of course, in competitive programming, you can save yourself some typing by not initializing variables which are going to immediately read from input anyway. I personally like writing the code which extracts input on the same line as the variable declaration, like this.

This way, the declaration is on the same line as the initialization.

```cpp
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int x; cin >> x;
    int n, q; cin >> n >> q;

    cout << x << endl;
}
```

3.2.3 C-style arrays

We will briefly[9] discuss **C-style arrays** here. As previously mentioned, C++ was built off of C, and C has its own array data type, on which other types, like `std::string` and `std::vector` are built on.

It is generally more difficult to work with C-style arrays instead of the modern container types provided by C++. In competitive programming, it is far easier to use containers such as `std::vector` for general use when you want to work with a "list of elements". However, it is still useful to learn about C-style arrays for various reasons:

- In cases where you have to declare multidimensional arrays, like when implementing dynamic programming or dealing with problems on grids, it can be easier to just use a C-style array.
- Learning C-style arrays will allows us to better understand how objects like `std::vector` work internally.

We can declare large amounts of data using arrays. An array is a series of elements of the same type that can be individually accessed by an index. The syntax to declare an array is as follows.

```
<data type> <identifier>[<number of elements>]
```

For example, here is how you declare an integer array of length 5, named `A`. The size of an array cannot be changed once it it declared.

```
int A[5]; // array of 5 uninitialized ints
```

This is equivalent to declaring 5 integer variables, named `A[0], A[1], A[2], A[3], A[4]`.

Note, however, that declaring an array like this inside a function still leaves the individual elements uninitialized! Modern C++ supports value initialization for arrays. The following declares an array where every element is value-initialized (set to zero).

```
int A[5] {}; // array of 5 ints, all set to 0
```

We can also outright denote the elements of the array using list initialization.

```
int A[5] {1, 2, 3, 4, 5}; // A: [1, 2, 3, 4, 5]

// if you provide less elements than the size of the array
// the remaining elements are value-initialized
int B[5] {1, 2, 3};       // B: [1, 2, 3, 0, 0]

// If you list-initialize an array, you can omit the size,
// and it will infer the size
int C[] {1, 2, 3};        // C: [1, 2, 3]
```

[9] This section is adapted from https://www.learncpp.com/, Chapters 17.8 to 17.13. I've removed many of the technical details which don't see much explicit use in competitive programming.

Remember how program memory is separated into three segments: data, heap, and stack? Variables declared inside a function get put on the stack, while variables declared globally (outside any function) get put on the data segment. And since the data segment has a larger capacity than the stack, for competitive programming, we should declare large amounts of data globally.

As an example, try running each of the following programs.

```cpp
#include <iostream>
using namespace std;
int main() {
    int A[10000000] {};
    cout << A[0] << endl;
}
```

```cpp
#include <iostream>
using namespace std;
int A[10000000] {};
int main() {
    cout << A[0] << endl;
}
```

The first program will crash, because you are placing all the data on the stack, which has a limited capacity. The second program places the array on the data segment, which has a higher capacity.

Remark 3.1. For advanced readers, you can read more about using braces to initialize variables here.

3.2.4 Using <algorithm> with C-style arrays

We will cover the <algorithm> header in more detail in a later handout, alongside all of the container types in C++. But for now, just so you can manipulate C-style arrays comfortably for basic use, let's introduce some commonly-used functions.

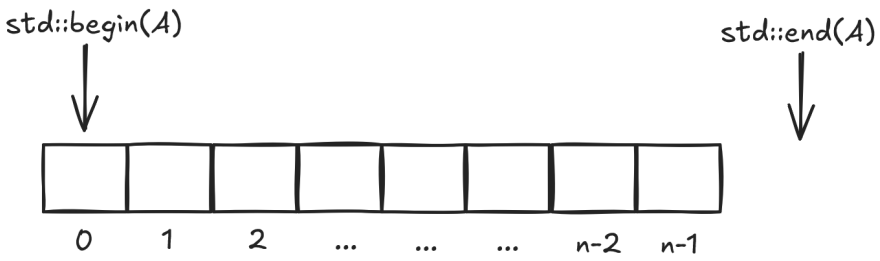
The proper way to use the functions in the <algorithm> header is using the `std::begin` and `std::end` functions from the <iterator> header. See the following example.

```
#include <iostream>
#include <algorithm> // std::sort, std::reverse, std::copy, std::fill
#include <iterator>  // std::begin, std::end, std::size

int main() {
    int A[] {3, 1, 4, 1, 5};
    int B[5] {};

    std::sort(std::begin(A), std::end(A));
    // A is now [1, 1, 3, 4, 5]

    std::reverse(std::begin(A), std::end(A));
    // A is now [5, 4, 3, 1, 1]

    std::copy(std::begin(A), std::end(A), std::begin(B));
    // This copies A into B, so B is now [5, 4, 3, 1, 1]

    std::fill(std::begin(A), std::end(A), -1);
    // This sets the value of every element of A to -1.

    int n = std::size(A);
    // n is 5, the number of elements in the array A.
}
```

Sometimes, you'll want to apply these functions on *subarrays*. A subarray is a contiguous portion of an array.

Some technical details are in order. When `A` is a C-style array, `std::begin(A)` refers to the location of the first element of the array, while `std::end(A)` refers to one location *after* the last element. In other words, you can think of

- `std::begin(A)` is equivalent to "location of `A[0]`"
- `std::end(A)` is equivalent to "location after `A[n-1]`"



If it seems weird that `std::end(A)` refers to the location *after* the last element, consider this. The functions in the <algorithm> library demonstrated above basically do a for loop that looks like this.

```
for (location = std::begin(A); location != std::end(A); ++location)
{
    // do things
}
```

The loop starts doing things at `std::begin(A)`. It does things as long as the location is *not* `std::end(A)`. For this for loop to make sense, `std::end(A)` has to be *after* the index of the last element.

Additionally, when `A` is a C-style array, the return values of `std::begin` and `std::end` can be "moved forward or backward" by adding/subtracting integers. For `std::begin`, you can just think of the added integer as the index you're starting at.

- `std::begin(A)+1` is equivalent to "location of `A[1]`"
- `std::begin(A)+k` is equivalent to "location of `A[k]`"

And for `std::end`, we can write out a similar sequence of

- `std::end(A)-1` is equivalent to "location after `A[n-2]`"
- `std::end(A)-k` is equivalent to "location after `A[n-k-1]`"

Let's look at some examples!

```
#include <iostream>
#include <algorithm> // std::sort, std::reverse, std::copy, std::fill
#include <iterator>  // std::begin, std::end, std::size

int main() {
    int A[10] {};

    std::fill(std::begin(A)+2, std::end(A), -1);
    // replace everything from A[2] onward with -1.

    std::fill(std::begin(A), std::end(A)-2, 42);
    // replace everything except the last two elements with 42.

    std::sort(std::begin(A)+1, std::end(A)-1);
    // sort the subarray from the 2nd element to the 2nd to the last
    ↪  element.
}
```

Remark 3.2. Prior to C++11, we didn't have `std::begin` and `std::end`. For C-style arrays `A` with `n` elements, instead of `std::begin(A)`, you write `A`. And instead of `std::end(A)`, you write `A + n`. This still works even after C++11, but using `std::begin` and `std::end` is much clearer and is the preferred syntax.

Adding and subtracting integers still works the same, so you would write syntax like `std::sort(A, A + n)` or `fill(A+2, A+n, -1)`. You might still see `(A, A + n)` instead of `(begin(A), end(A))` in a lot of code out there.

3.3 Input and output, data types, constants, operators

It is important that you are familiar with the basic data types and operators available in C++. These are well-explained the tutorial found in learncpp.com, so I won't replicate them here.

For this section, read through the following tutorial pages. You don't need to memorize every single detail, but at least try to follow along. Even if you already know C++, it doesn't hurt to go through the entire tutorial again. You might learn some new things!

- The entire Chapter 1 of https://www.learncpp.com/.
- The entire Chapter 4 of https://www.learncpp.com/.
- Chapters 5.1 to 5.3 of https://www.learncpp.com/.
- The entire Chapter 6 of https://www.learncpp.com/.

Additionally, for competitive programming, by adding these two lines before performing any input/output operations, we can make input and output operations *fast*.

```cpp
#include <iostream>

int main()
{
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);

    cout << "Hello World!" << endl;
}
```

Read https://usaco.guide/general/fast-io for more information on what these two lines do.