NOI.PH Training: Technical Stuff 2

More Technical Stuff (and More C++ Garbage)

Kevin Charles Atienza

Contents

1 Introduction

C++ is kinda garbage. This doc will be kinda garbage too, but I hope it's helpful garbage.

Note: A similar caveat applies here as in the previous module in this series (Technical Stuff 1): Some of the things said here will not be strictly correct, or somewhat simplified compared to actual reality, to make the explanations simpler.

1.1 Why learn more technical stuff?

One notable difference between IMO[1] and IOI[2] is that in the latter, not only do you need to come up with a correct solution (or algorithm), but you also have to *implement* it.

The closest analog of implementing an algorithm as a program, in a math olympiad, is writing proofs in math's "technical/formal language." However, in math, you don't need to write a computer-checkable proof—ultimately, a human grader (or graders) will evaluate your proof, so you don't need to be overly precise and pedantic. Although these graders have a high standard of rigor, at some point, they have to evaluate the level of understanding and insight you obtained in a problem, and use some amount of subjective judgement to determine the number of points you get.

In contrast, in the informatics olympiad, evaluation and grading is completely automated.[3] This is unsurprising; one of the most important goals of programming and informatics is automation. However, this also means that a (dumb) computer will "evaluate" your program, so you have to be overly precise and pedantic. Indeed, it doesn't even read your code—it just compiles it, runs it, and checks if the output is correct. It doesn't give you partial points (for a subtask) whether you made a tiny mistake (such as a typo) or a major one (such as if your solution has a fundamental flaw).

For this reason, it's all the more important for you, as a competitive programmer, to understand the tools you're using (your computer and your programming language) at a deeper level. As programmers learn to master their programming skills, it's not uncommon to find people experiencing a several-fold increase in productivity; not only do they code faster, but they also need to spend less mental effort coding, which means more time and energy to think about solutions and attack more problems. You may have already experienced this to some extent; you're probably already a much better programmer than when you were starting out. You may even be a faster typist now! However, you still have much to learn, so this module will help you level up your technical knowledge.

Of course, the most effective technique to improve your implementation skills is good old **practice**.[4] But at some point, you really have to just sit down and read about the finer technical details—you can't learn everything through mere practice and trial-and-error. Some of these technical matters are unintuitive/counterintuitive, some are seldom seen/hard to discover, and some are simply genuinely hard.

[1] International Mathematical Olympiad
[2] International Olympiad in Informatics
[3] This may change in the (far) future, but you can count on it for now.
[4] Actually, this is an advantage: The good thing about implementation skills, unlike problem-solving skills, is that it's very trainable. The skill ceiling is still quite high, but not as much if your main goal is just improving your implementation skills for competitive programming.

2 The cache

Consider the following two programs:

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

const int R = 5000;
const int C = 20000;

int A[R][C];
int main() {
    mt19937 rng(11);
    for (int i = 0; i < R; i++) {
        for (int j = 0; j < C; j++) {
            A[i][j] = uniform_int_distribution<int>(0, 100)(rng);
        }
    }

    // prefix sums
    for (int i = 1; i < R; i++) {
        for (int j = 1; j < C; j++) {
            A[i][j] += A[i][j-1] + A[i-1][j] - A[i-1][j-1];
        }
    }
}
```

This is just a normal 2D prefix sum program, and we know it runs in $\mathcal{O}(RC)$ time. So I ran the program on my computer, and it finished in 0.891sec. (I compiled it with `-O3` and timed it with bash's `time` command.) Looks good so far.

However, suppose we swap the two loops of the prefix sum as follows:

```cpp
    // prefix sums
    for (int j = 1; j < C; j++) {
        for (int i = 1; i < R; i++) {
            A[i][j] += A[i][j-1] + A[i-1][j] - A[i-1][j-1];
        }
    }
```

Conceptually, we're computing exactly the same table, just in a different order. The number of operations is still exactly the same, so it's still $\mathcal{O}(RC)$ time. Now I run the program again, but this time, it finished in 2.196sec, more than double the previous program! What happened?? Shouldn't they have finished in roughly the same time?

Well, what really happened is that the cache got involved!

The **cache** is an optimization developed by computer engineers. They realized that RAM access is somewhat slow, so they invented an intermediate storage—the cache—to speed up programs. The cache is smaller than the RAM, but it's closer to the CPU so it can be accessed faster.

The cache is meant to be "invisible"—your programs are not meant to deal with it and should work the same with or without it. So it doesn't your program's correctness. However, you still have to think about it since it affects your program's *efficiency*.

2.1 So, how does the cache work?

When a program access a part of a RAM, it copies that chunk of RAM to the cache and uses the cache instead. This way, if the program tries to access a nearby part of RAM, it may already be in the cache, so accessing it is much faster.

Several chunks of RAM may be in the cache at any time. When accessing a memory location, the cache is checked first; if it's already there, then it just uses the cache, saving a RAM access! This is called a **cache hit**. If it's not there—**cache miss**—it loads the appropriate chunk from RAM to the cache.

If the cache is full, it drops one of the chunks in the cache to make room. It tries to keep the number of cache misses low by being *smart* about which chunk to drop. For example, it may drop the least-recently used chunk in the cache.[5]

Of course, if a chunk is dropped from the cache, the updated version of the chunk has to be written to the RAM. So as you can imagine, it is no easy task to ensure that the cache and the RAM are in sync. But you don't need to worry about it since it's already automatically being done for you. (Thank the engineers!)

2.2 Consequences of the cache

Because of the cache, and the fact that *contiguous* chunks of RAM are loaded onto it, the order in which you access memory affects the number of cache hits and misses, and thus affect your program's running time, sometimes quite noticeably. For example:

- Accessing an array in random order may be noticeably slower than accessing them in sequential order.
- Another instance when this is noticeable is when accessing 2D (or 3D, etc.) arrays: row-by-row access can be faster than column-by-column access.

Exercise 2.1. Given what you know about the cache, can you explain the above examples?

Hint: You may have to recall that a 1D array is located in a contiguous chunk of memory, and each row of a 2D array is basically a 1D array.

This explains why our two programs differ in runtime so much!

Some algorithms are more cache-friendly than others. It's one reason why linear-time sorting (such as radix sort) can be slower in practice than comparison-based sorting (such as quicksort), even if it supposedly has a "better" time complexity. The RAM access pattern of quicksort is just cache-friendlier than that of radix-sort.

The size of the cache may affect the running time of a program in drastic ways. For example, an array that's just below the cache size can be loaded onto the cache fully, but a slightly larger array just above the cache size cannot, so there'll be more cache misses, and the slowdown may be more significant than big-$\mathcal{O}$ analysis would suggest.

So if we have to mind the cache when programming, how big is it? The size of the cache may vary from computer to computer—it ranges from 16KB to 32768KB—and how much they speed things up also vary from computer to computer.[6] So it's not a good idea to design your program around a specific cache size.

[5] I don't actually know the actual algorithm; so we just assume that it's smart enough!

[6] In fact, the cache is divided into three levels—L1, L2, L3—each with different sizes and varying speeds.

There are so-called cache-oblivious algorithms that are cache-friendly no matter what the size of the cache is. You don't have to learn them though. For the purposes of competitive programming, it's enough to think about the RAM access pattern of your program. Accessing your arrays in random or disjointed ways may be worse than accessing them in order.

So here's a takeaway as a competitive programmer: *Be aware that the cache exists. Code with that in mind.* However, keep in mind as well that this is basically *constant optimization*, so it shouldn't be the first thing to think about—ultimately, the big-$\mathcal{O}$ analysis is what's important.

3 Stack vs heap memory

The compiler splits the available memory into two parts: the **stack memory** and the **heap memory**.[7] Of course, this only matters to the compiler; the OS doesn't care and treats them all the same.

Everything you create via the `new` keyword are created in the heap. Everything else (variables, and arrays not created via `new`) are created in the stack.[8] In the stack, the compiler usually allocates sequentially; that is, whenever the compiler needs to allocate in the stack (say, you just declared a variable), it just allocates the next one. In contrast, it doesn't (always) do this with the heap.

Note that whenever you call a function, the compiler has the task of allocating memory for the arguments and local variables of the function. Whenever you return from a function, these variables are no longer needed, so it deallocates them from the stack.

Now, notice that you can only return from the most recent function call (that you haven't returned from yet). This is just like a regular stack data structure! We have the following correspondence:

- Calling a function ⇔ Pushing onto a stack
- Returning from a function ⇔ Popping from a stack

Indeed, this is exactly what happens with the local variables: The compiler will only ever need to deallocate the most-recently allocated variables. In other words, the stack memory is actually being used like a stack (data structure)! This is why it's called "stack memory" in the first place.

This is another reason why, when doing recursion, each recursive call gets its own copy of the local variables. All the compiler is doing is it's allocating more and more memory from the stack.

The stack is limited, so if it runs out, the program crashes. This is called a *stack overflow* error—the stack literally overflows. The most common way this is triggered is via infinite (or very very deep) recursion.

You can change the stack size via the bash command `ulimit -s`, e.g., `ulimit -s 10000` to set it to 10000 KB (the unit is KB). If you run `ulimit -s` without passing anything, the current stack limit is printed.

Note that this is not a compiler option! It's an OS option. Usually, the computer you'll be assigned in the competition and the judges' computers will have the same limit, and it is usually generously high. It is the contest organizers' responsibility to set these things up. But in your computer at home, it will usually be set to a low value by default, so you should change your local setup by running, say, `ulimit -s 1000000`.

You can have this be run automatically every time you open a new terminal by putting it in your `~/.bashrc` file. (Don't forget to add a newline at the end!) If you don't know what `~/.bashrc` is, ask in Discord!

[7] There's also something called the data segment, but you don't need to worry about it for now. (Think of it conceptually as part of the heap.)

[8] The "stack" memory isn't the same as the "stack" data structure, though as you will see later, the stack memory is mostly used as a stack.

3.1 A pitfall

Suppose, for some reason, you wanted a function that *allocated* memory, similar to `new` or `malloc`? Perhaps you just learned about pointers and the `&` operator, so you tried to write something like this:

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

// allocate an int, and initialize it with the value n+3
int* foo(int n) {
    int x = n+3;
    return &x;
}

int main() {
    int* y = foo(4);
    cout << *y << '\n'; // you expect to see 7
}
```

You try to compile it, and thankfully, it succeeded! You run it, expecting to see the output `7`, and you got...segmentation fault.

After a few moments, you notice that the compilation didn't actually go that smoothly, because `g++` gave the following warning:

```
my_code.cpp: In function ‘int* foo(int)’:
my_code.cpp:6:9: warning: address of local variable ‘x’ returned [-Wreturn-local-addr]
     int x = n+3;
         ^
```

Okay, it says that the address of the variable `x` was returned by your function, which was what you intended. But for some reason, the compiler doesn't like that. (But it didn't outright reject it, since it's *technically* valid.) Why?

Desperate, you try another tactic. You just learned about C++ references and thought that if pointers didn't work, then maybe references would? So you code this:

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

// allocate an int, and initialize it with the value n+3
int& foo(int n) {
    int x = n+3;
    return x;
}

int main() {
    int& y = foo(4);
    cout << y << '\n'; // you expect to see 7
}
```

Okay, you compile it, and it succeeded again, albeit with a similar warning ("`reference to local variable ‘x’ returned`"). You run it...and get another segmentation fault. What's happening??

Well, there's actually a simple explanation. Remember that local variables are allocated in the stack, and they're deallocated as soon as you return from the function call. But this means that when you return from the `foo` call, the local variable `x` will be deallocated, which means

that its original address will now be considered vacant, and you can't use it anymore! This is why the program crashed when it tried to read from it via `*y` (since `y` was assigned the address of `x`).

In other words, this attempt to do memory allocation is doomed to fail, since it's returning addresses that are in the stack, which are deallocated as soon as the function call finishes. To fix this, what you want is to allocate from the heap, since they're never deallocated (except if the program ends, or you deallocate it explicitly). In other words, don't bypass the librarian! Stick with the `new` keyword.

4 More on C++ constructors

Suppose you coded something like this, where a **struct** has an attribute whose type is another **struct** you wrote:

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

struct Dog {
    string name;
    int age;

    Dog(string n, int a) {
        name = n;
        age = a;
    }
};

struct Person {
    string name;
    vector<string> name_of_friends;
    Dog pet;

    Person(string n, vector<string> nof, Dog p) {
        name = n;
        name_of_friends = nof;
        pet = p;
    }
};

int main() {
    Person p("Cloud", {"Tifa", "Aerith", "Yuffie"}, Dog("Anakin", 48));
    // something something. Your code goes here
}
```

It looks fine, right? You're pretty happy with it, so try to compile it. Unfortunately, g++ spits out this garbage at you:

```
my_code.cpp: In constructor 'Person::Person(std::__cxx11::string, std::vector<std::__cxx11:...
my_code.cpp:19:49: error: no matching function for call to 'Dog::Dog()'
     Person(string n, vector<string> nof, Dog p) {
                                                 ^
my_code.cpp:8:5: note: candidate: Dog::Dog(std::__cxx11::string, int)
     Dog(string n, int a) {
     ^~~
my_code.cpp:8:5: note:   candidate expects 2 arguments, 0 provided
my_code.cpp:4:8: note: candidate: Dog::Dog(const Dog&)
 struct Dog {
        ^~~
my_code.cpp:4:8: note:   candidate expects 1 argument, 0 provided
my_code.cpp:4:8: note: candidate: Dog::Dog(Dog&&)
my_code.cpp:4:8: note:   candidate expects 1 argument, 0 provided
```

What does it all mean??

Well, it means that for some reason, the compiler is looking for an *empty constructor* of the Dog struct (that is, a constructor that requires no arguments), but couldn't find one, so it gives up and decides that your code is broken. "But why?" you ask, "I never created a Dog with an empty constructor!"

The explanation is that whenever you write something like

```
Something s;
```

the empty constructor of the type `Something` is being called! There are some weird exceptions to this, e.g., the line `int x;` inside a function doesn't call the empty constructor of `int`—it's a bit hard to explain, so we won't.[9] But at least it's always true for any `struct` that you write.

In particular, when we wrote `Dog pet;`, the empty constructor of `Dog` was called. So it doesn't matter even if we reassigned `pet` anyway in the constructor via `pet = p`; the empty constructor has already been called. And since `Dog` doesn't have an empty constructor, `g++` doesn't like your program.

To fix this, we write the constructor this way:

```
    Person(string n, vector<string> nof, Dog p): pet(p) {
        name = n;
        name_of_friends = nof;
    }
```

The thing after the colon '`:`' tells the compiler which constructor to use when initializing the attributes of this `Person` the moment that it's created (via this constructor). For example, consider this code:

```
struct Foo {
    int a, b, c;
    Foo(int _a, int _b, int _c) {
        a = _a;
        b = _b;
        c = _c;
    }
};
struct Bar {
    Foo foo;
    Bar(): foo(10, 20, 30) {}
};
```

The line `foo(10, 20, 30)` tells the compiler to *initialize* `foo` as if you declared it like

```
Foo foo(10, 20, 30);
```

So using this syntax, the compiler doesn't attempt to call the empty constructor anymore, and we're good!

"Hold on!" You ask. "I didn't define a constructor for `Dog` that takes in a `Dog` as an argument! So I shouldn't be able to write the following line:"

```
Pet pet(p);  // assume 'p' is of type Pet
```

"So why does it work?" Good question! You have a keen eye.

It turns out that C++ automatically adds such a constructor! For every struct you define, say `Foo`, it automatically adds a constructor whose type signature is `Foo(Foo foo)`, and what it does is it copies the object you passed! This is called the *copy constructor*. So this is why it works!

In fact, using this syntax, we can write our constructors more neatly, like this:

[9] If you're curious and would like to know more, feel free to ask in Discord!

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

struct Dog {
    string name;
    int age;

    Dog(string n, int a): name(n), age(a) {}
};

struct Person {
    string name;
    vector<string> name_of_friends;
    Dog pet;

    Person(string n, vector<string> nof, Dog p): name(n), name_of_friends(nof), pet(p) {}
};

int main() {
    Person p("Cloud", {"Tifa", "Aerith", "Yuffie"}, Dog("Anakin", 48));
    // something something. Your code goes here
}
```

Not only is it shorter, it actually communicates your intent better!

By the way, with this syntax, you can reuse names like so:

```cpp
struct Dog {
    string name;
    int age;

    Dog(string name, int age): name(name), age(age) {}
};
```

The compiler will just know what you mean. This is convenient so you don't need to invent separate names for the attribute and the parameter of the constructor.

5 Templates

If you think about it, `vector` is not really a type, but some sort of type *factory*. You pass it a type, and it returns another type. For example, if you pass `int`, then it returns `vector<int>`, if you pass `char`, then it returns `vector<char>`, and `vector<int>` and `vector<char>` are considered distinct types. (You can check this by trying to assign a `vector<int>` value to a `vector<char>` variable and noting that the compiler will reject it.)

You can also make something like `vector` by using *templates*!

For example, suppose we want to create a slightly more general version of `vector`, called `offset_vector`. Whereas `vector`s always have valid indices in the interval $[0, n)$ for some n, `offset_vector` will have valid indices in some interval $[l, r)$ for some integers l and r with $l \leq r$. These integers will be passed through the constructor.

Of course, you can just use a `vector` and just subtract `l` whenever you read from it or write to it, but that's cumbersome, and error-prone!

Let's see how we can do it without templates first. The following is an attempt to define an `offset_vector` of `char`s:

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

struct offset_vector {
    int l, r;
    vector<char> a;

    offset_vector(int l, int r): l(l), r(r), a(r - l) {
        assert(l <= r);
    }

    char get(int i) {
        assert(l <= i && i < r);
        return a[i - l];
    }

    void set(int i, char c) {
        assert(l <= i && i < r);
        a[i - l] = c;
    }
};

// sample usage
int main() {
    offset_vector v(5, 10);
    v.set(7, 'b');
    v.set(8, 'i');
    v.set(9, 'g');
    cout << v.get(7) << v.get(8) << v.get(9) << '\n'; // prints: big
    v.set(8, 'a');
    cout << v.get(7) << v.get(8) << v.get(9) << '\n'; // prints: bag
    // v.set(10, 's');  // this is an out-of-bounds error, so I commented it out
}
```

It works! However, this implemention is always an `offset_vector` of `char`s, so if you want to make an `offset_vector` of `int`s, you'd have to copy-paste this and replace `char` with `int` everywhere. If you need another one, you copy-paste again.

With `template`s, you can tell the compiler to automatically do this copy-paste and replace-

ment process. Here's how you do it:

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

// Here, 'T' is our type argument.
template<class T>
struct offset_vector {
    int l, r;
    vector<T> a;

    offset_vector(int l, int r): l(l), r(r), a(r - l) {
        assert(l <= r);
    }

    T get(int i) {
        assert(l <= i && i < r);
        return a[i - l];
    }

    void set(int i, T c) {
        assert(l <= i && i < r);
        a[i - l] = c;
    }
};

// sample usage
int main() {
    offset_vector<char> v(5, 10);  // note that we passed 'char' here
    v.set(7, 'b');
    v.set(8, 'i');
    v.set(9, 'g');
    cout << v.get(7) << v.get(8) << v.get(9) << '\n'; // prints: big
    v.set(8, 'a');
    cout << v.get(7) << v.get(8) << v.get(9) << '\n'; // prints: bag
    // v.set(10, 's');  // this is an out-of-bounds error, so I commented it out
}
```

There's nothing special about the name `T`. You can use another name if you want, though `T` is the most-commonly used one. Of course, if your template needs more than one type argument (such as STL's `pair<X, Y>` which takes two), then you'll have to use two different names.[10]

Structs are not the only place you can use **`template`**s in. You can also use it for functions. For example:

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

template<class T>
T square(T v) {
    return v * v;
}

// sample usage
int main() {
    double a = square(1.5);
```

[10] If you've used `tuple`, then you might notice that `tuple` is a bit more special than the others since it accepts an arbitrary number of arguments rather than a fixed number! You can also make something like this, but it's **very** complicated in C++, so I suggest not bothering with it at all. (It's easier in many other languages.)

```
    int b = square(10);
    int c = square(1 << 20);
    long long d = square(1 << 20);
    long long e = square(1LL << 20);

    cout << a << '\n'; // prints 2.25
    cout << b << '\n'; // prints 100
    cout << c << '\n'; // prints 0 (because of int overflow)
    cout << d << '\n'; // prints 0 (because of int overflow)
    cout << e << '\n'; // prints 1099511627776 (which is 2 raised to 40)
}
```

Note that you don't have to pass the type this time—the compiler infers the correct type based on the argument you passed. This is why `d` became `0` even though it's a **`long long`**; you passed in an **`int`**, so the compiler used the **`int`** version of the function `square`.

Finally, this feature is actually what "Template" means in "Standard *Template* Library"—after all, pretty much the whole STL is made up of **`template`**s!

Most programmers call this feature *generics*. "Generics" is the language-neutral term, while "templates" is C++'s version of it.

6 Default arguments

You can make some arguments in a function optional by providing a default value. For example, in the following, `sep` and `end` are optional, with default values `" "` and `"\n"`.

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

template<class T>
void print_separated(vector<T>& vec, string sep = " ", string end = "\n") {
    for (int i = 0; i < vec.size(); i++) {
        if (i > 0) cout << sep;
        cout << vec[i];
    }
    cout << end;
}

int main() {
    vector<char> a = {'a', 'b', 'c'};
    print_separated(a);
    print_separated(a, "--");
    vector<int> b = {10, 20, 30};
    print_separated(b, " ", "!");
    print_separated(b);
}
```

Note that the `&` here means *pass by reference*. It was discussed briefly in the previous module. Here is the output:

```
a b c
a--b--c
10 20 30!10 20 30
```

Note that there's no new line character after the exclamation point since we replaced the default value of `end` with a string that has no new line character. Also, note that there's no way to replace `end` without passing in `sep`, that's why in the third call, I passed in `" "` (even if it's the same as the default).

By the way, the `print_separated` function is very useful and convenient. I encourage you to implement it (or something similar). You can include it as part of your "boilerplate"—the preliminary code you include in all your solutions. You can write your boilerplate code once quickly near the beginning of the contest, then whenever you want to write a new program, just copy that file over instead of starting from an empty file.

7 Operator overloading

You can overload most of the C++ operators so they work with your custom types (structs). For example, here's how to override the addition operator:

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

struct MyInt {
    int v;
    MyInt(int v): v(v) {}

    MyInt operator+(MyInt other) {  // overrides the operator +
        return MyInt(v + other.v);
    }
};

// sample usage
int main() {
    MyInt x(10), y(20);
    MyInt z = x + y;  // call the overloaded +
    cout << z.v << '\n'; // 30
}
```

In other words, you define a method named `operator+` to override +. To override multiplication, you define `operator*`. There's also `operator-`, `operator/`, `operator%`, `operator[]`, and so on.

Exercise 7.1. Created a struct named `modded_int` which is like an `int` but automatically reduces all numbers mod $10^9 + 7$.

Tip: Define `1000000007` as a `const int` somewhere so you don't type it everywhere. (Write it as `1'000'000'007`. It means the same thing, but lets you easily see that you got the correct number of zeroes.)

Caution: Make sure it doesn't overflow! I suggest using `long long` for the attribute instead of `int`. You can use `int` if you want, though you'd have to typecast everywhere which is annoying.

8 More on C++ references

Note: This section talks about the C++ feature called "references", which was briefly discussed in the Technical Stuff 1 module. Read about it first before reading this part.

You've already seen that you can pass references as arguments to functions. This is useful if you want to pass a big object, but don't want to copy it every time. You can do it with pointers, but with references, the syntax is simpler, since it is still the same type (not a pointer type); it just happens to share the same memory location as the variable that was passed.

Well, you can return references too!

8.1 References as return values

Here's one example. Suppose you found our implementation of `offset_vector` to be annoying to use, since you have to do `.get(i)` and `.set(i, something)` every time instead of the usual array-indexing syntax. Well, you can improve things by overriding the indexing operator `operator[]` (as explained in section 7) like so:

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

template<class T>
struct offset_vector {
    int l, r;
    vector<T> a;

    offset_vector(int l, int r): l(l), r(r), a(r - l) {
        assert(l <= r);
    }

    T operator[](int i) {
        assert(l <= i && i < r);
        return a[i - l];
    }

    void set(int i, T c) {
        assert(l <= i && i < r);
        a[i - l] = c;
    }
};

// sample usage
int main() {
    offset_vector<char> v(5, 10);  // note that we passed 'char' here
    v.set(7, 'b');
    v.set(8, 'i');
    v.set(9, 'g');
    cout << v[7] << v[8] << v[9] << '\n'; // prints: big
    v.set(8, 'a');
    cout << v[7] << v[8] << v[9] << '\n'; // prints: bag
}
```

Okay, this worked...somewhat. Notice that we kept the method `set`. This is because if you removed it, and wrote something like `v[7] = 'b';`, then the compiler would complain, saying something like `lvalue required as left operand of assignment`. What's this lvalue thing??

We don't know (yet) what an lvalue is, but it should be clear why the compiler is complain-

ing: `v[7]` is just a *raw value* and doesn't refer to any address! So you can't assign to it, in the same way you can't write "`7 = 9;`".

But now, it should be clear how to fix this—references! The following now works:

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

template<class T>
struct offset_vector {
    int l, r;
    vector<T> a;

    offset_vector(int l, int r): l(l), r(r), a(r - l) {
        assert(l <= r);
    }

    T& operator[](int i) {
        assert(l <= i && i < r);
        return a[i - l];
    }
};

// sample usage
int main() {
    offset_vector<char> v(5, 10);  // note that we passed 'char' here
    v[7] = 'b';
    v[8] = 'i';
    v[9] = 'g';
    cout << v[7] << v[8] << v[9] << '\n'; // prints: big
    v[8] = 'a';
    cout << v[7] << v[8] << v[9] << '\n'; // prints: bag
}
```

Notice that the return type is `T&`, not `T`, which is why we can assign to it now!

> **Exercise 8.1.** Why doesn't the error described in subsection 3.1 happen in this case?

As a side note, you can think of an "lvalue" as basically *something that can be on the left side of an assignment* (such as `x = y`). And as it turns out, references are lvalues, which is why the assignment `v[7] = 'b'` works.

8.2 References as attributes

You can also have attributes that are references, such as in the following:

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

struct Foo {
    vector<int>& a;

    Foo(vector<int>& a): a(a) {}
};

// sample usage
```

```cpp
int main() {
    vector<int> b = {31, 41, 59};
    Foo f(b);

    // illustration that b and f.a refer to the same thing:
    cout << b[2] << " " << f.a[2] << '\n'; // 59 59
    f.a[2] = 592;
    cout << b[2] << " " << f.a[2] << '\n'; // 592 592
    b[2] = 60;
    cout << b[2] << " " << f.a[2] << '\n'; // 60 60
}
```

As expected, `b` and `f.a` refer to the same thing.

For example, this is useful if you want to add a thin "wrapper" **`struct`** over another type, perhaps to modify its functionality a bit, but you don't want to incur additional performance penalties, like in the following:

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

struct PaddedVector {
    vector<int>& vec;
    int padding;

    PaddedVector(vector<int>& vec, int padding = -1): vec(vec), padding(padding) {}

    int operator[](int i) {
        if (0 <= i && i < vec.size())
            return vec[i];
        return padding;
    }
};

// print the contents of 'thing' across indices l (inclusive) to r (exclusive) on a line
template<class T>
void print_range(T& thing, int l, int r) {
    for (int i = l; i < r; i++) {
        if (i > l) cout << ' ';
        cout << thing[i];
    }
    cout << '\n';
}

// sample usage
int main() {
    vector<int> vec = {31, 41, 59};

    PaddedVector pvec1(vec);
    PaddedVector pvec2(vec, 99);

    // print the contents of the padded vectors
    // across an index range that's normally out of bounds for 'vec':
    print_range(pvec1, -3, 6); // -1 -1 -1 31 41 59 -1 -1 -1
    print_range(pvec2, -3, 6); // 99 99 99 31 41 59 99 99 99
}
```

If we didn't use references, then the vector will have to be copied everytime you created an instance of `PaddedVector`, which is slow. (Of course, if you actually *do* need separate copies, then don't use references.)

This pattern is called the *delegation* pattern—you're making a new type, but it *mostly*

delegates its duties (methods) to one of its attributes (whose type is another, existing type).[11]

There's one gotcha with this though: You *have* to initialize an attribute that's a reference via the copy constructor; otherwise, the reference wouldn't really make sense! This is in contrast to pointers, which can refer to the address 0 (so it's a null pointer) or some other random 64-bit address (even if it's unusable).

[11] In so-called "object-oriented programming", some things that can be done with the delegation pattern are better done with *inheritance*. If you don't know what that is, don't worry! If you're curious, ask in Discord!

9 How to use the docs effectively

Still unwritten. If you want to learn about this, let us know in Discord.

Notes for now:

- IOI and most other contests provide C++ documentation.
- It's counterproductive to learn everything by trial and error. You have to read docs at some point.
- You're not supposed to memorize all the technical details of a language or its libraries. In fact, that's quite unreasonable. This fact extends to real-world programming; most professional programmers just google stuff extensively.
- Learn how to search through the docs.

10 Random number generators

Still unwritten. If you want to learn about this, let us know in Discord.

Notes for now:

- In principle, it's hard to generate truly random numbers. Most languages use pseudorandom number generators instead.
- C++ has several builtin RNGs, some inherited from C. Use `mt19937`.
- Seed it with system time!

```
unsigned seed = chrono::system_clock::now().time_since_epoch().count();
mt19937 rnd(seed);
```

  Also prevents some kinds of "hacking". (More relevant in some contests like CF rounds.) Remember that PRNGs are predictable once you know the seed, so your solution can be "hacked" (that is, someone can find a case that breaks it) if your seed is known.
- Note that this doesn't always prevent hacking by an *adaptive judge*.
- Essential for writing a program that generates *random* test cases, e.g., for stress testing.

Exercise 10.1. Oi Ghenn recently learned quicksort and also heard about a thing called the "*median-of-five trick*", which is supposed to speed up quicksort. He didn't actually bother reading about this median-of-five thing and simply guessed what it's supposed to do. So he implemented his version of quicksort like this:

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

unsigned seed = 314159;  // totally random seed with no pattern at all!!!
mt19937 rnd(seed);

template<class T>
void my_sort(T l, T r) {
    int n = distance(l, r);
    if (n <= 10) {  // for small n, use a fast builtin sort
        sort(l, r);
    } else {  // for large n, perform a quicksort step

        // take five random elements and put them in front
        const int M = 5;
        for (int i = 0; i < M; i++) {
            int j = uniform_int_distribution<int>(i, n - 1)(rnd);
            swap(*(l + i), *(l + j));
        }

        // take the median of these five as pivot
        my_sort(l, l + M);
        swap(*l, *(l + M/2));

        // l now points to the pivot. now partition
        T u = l + 1, v = r - 1;
        for (int i = 0; i < n - 1; i++) {
            if (*u < *l) {
                u++;
            } else if (*l < *v) {
                v--;
            } else {
                swap(*u, *v);
                v--;
            }
        }
        assert(u == v + 1);  // sanity check
        swap(*l, *v);  // put the pivot in the middle

        // recursively sort
        my_sort(l, v);
        my_sort(u, r);
    }
}

template<class T>
void print_separated(vector<T>& vec, string sep = " ", string end = "\n") {
    ... // defined in the same way as before, hence omitted
}

int main() {
    int n; cin >> n;
    vector<int> a(n);
    for (int& v: a) cin >> v;
    my_sort(a.begin(), a.end());
    print_separated(a);
}
```

Can you provide a "hacking case" for this solution? Assume $1 \leq n \leq 10^6$ and each element of the array is between -10^9 and 10^9, inclusive.

Note: Median-of-fives is indeed a trick related to quicksort, but the above code is **not** what it does! Oi Ghenn really shouldn't have been lazy and just read about it more.

Exercise 10.2. Franz Ferdinand Talampakan is trying to solve the following problem:

> *We say that two integer arrays A and B are almost the same if all the following are satisfied:*
>
> - $|A| = |B|$*, that is,* A *and* B *have the same length.*
> - $A[i] = B[i]$ *for every index* i*, except at most one index. In other words,* A *and* B *are equal except possibly in one position.*
> - *If* $A[i] \neq B[i]$*, then* $|A[i] - B[i]| = 1$*. In other words, if the values differ, then they differ by at most one.*
>
> *Given two arrays* B *and* S*, find the number of subarrays of* B *that are almost the same as* S*.*
>
> **Constraints:**
>
> - $1 \leq |S| \leq |B| \leq 10^6$.
> - $-10^9 \leq S[i], B[i] \leq 10^9$.

Franz came up with a rather clever solution! He noticed that two arrays A and B of length n are almost the same if and only if

$$(A[0] - B[0])^2 + (A[1] - B[1])^2 + \ldots + (A[n-1] - B[n-1])^2 \leq 1.$$

This may seem like a roundabout way to express almost-sameness, but it's actually helpful, because Franz knew that he can optimize the computation of the expression above for all length-$|S|$ subarrays of B using something called "FFT." Anyway, before applying FFT, Franz first implemented a quadratic-time solution, presumably to do stress testing later.

However, Franz foresaw a possible problem: overflow! The numbers are certainly going to exceed the limits of 64-bit integers, so the check may fail. Luckily, Franz came up with a workaround: just compute the expression modulo some random m, where m is small enough, say, a 32-bit integer! This prevents overflow. Also, he performs this check twice just to be *extra* sure.

Anyway, here is his quadratic-time code (which he'll optimize to $\mathcal{O}(n \log n)$ later with FFT):

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

using ll = long long;

// totally random seed with no pattern at all!!!
unsigned base_seed = 46692016;
mt19937 rnd;

template<class T>
ll sum_square_diff(T a, T b, int n, ll m) {
    ll t = 0;
    while (n--) {
        ll v = (*(a++) - *(b++)) % m;
        t = (t + v * v) % m;
    }
    // ensure that t >= 0
    return (t % m + m) % m;
}

template<class T>
bool almost_same(T a, T b, int n) {
    // do the following check two times
    for (int it = 0; it < 2; it++) {
        // choose a random modulus m between 2^29 and 2^30...
        ll m = rnd() % (1LL << 29) + (1LL << 29);
        // ...then check the condition modulo m
        if (sum_square_diff(a, b, n, m) > 1)
            return false;
    }
    return true;
}

vector<ll> take_array() {
    int n;
    cin >> n;
    vector<ll> a(n);
    for (ll& v: a)
        cin >> v;
    return a;
}

int main() {
    vector<ll> b = take_array();
    vector<ll> s = take_array();

    rnd = mt19937(base_seed + b.size() + s.size()); // initialize the RNG

    int ans = 0;
    for (int i = 0; i + s.size() <= b.size(); i++) {
        if (almost_same(s.begin(), b.begin() + i, s.size()))
            ans++;
    }
    cout << ans << '\n';
}
```

Can you find a breaking case for this solution? Note that a case that triggers TLE will not be counted because Franz knows he can optimize this with FFT.

Exercise 10.3. Note: This continues off from the previous exercise.

Franz' solution got hacked!

He sensed that the hack has something to do with the randomization, so just to be extra extra sure, he updated his RNG initialization to this:

```
    ll seed = base_seed + b.size() + s.size();
    for (ll v : b) seed ^= v;
    for (ll v : s) seed ^= v;
    rnd = mt19937((unsigned)seed);
```

Can you now find a breaking case? Again, note that a case that triggers TLE will not be counted.

Exercise 10.4. Note: This continues off from the previous exercise.

Franz' solution got hacked again!

He's now getting desperate. Just to be extra, extra sure, he updated his code to now do the check *three* times!

```
template<class T>
bool almost_same(T a, T b, int n) {
    // do the following check three times
    for (int it = 0; it < 3; it++) {
        ... // same contents as before, so omitted
    }
    return true;
}
```

Can you now find a breaking case? Again, a case that triggers TLE will not be counted.

Exercise 10.5. Note: This continues off from the previous exercise.

Franz' solution got hacked yet again by some clever chap!

He doesn't really know what to do anymore. Not caring about running time anymore, he now does the check 1000 times:

```
    ...
    // do the following check 1000 times
    for (int it = 0; it < 1000; it++) {
    ...
```

Now, Franz realizes that this may not run fast anymore, even with FFT, but in the interest of pure correctness, can you still find a breaking case?

11 Anonymous Functions

Suppose you have a list of numbers, and you want to sort them by increasing absolute value. (For simplicity, suppose you don't need to distinguish between $-x$ and x.) You can use the `sort` function in STL like this:

```
bool abscomp(int a, int b) {
    return abs(a) < abs(b);
}

int main() {
    vector<int> vals = {2, -7, -1, 8, 2, -8, 1, 8, -2, 8};
    sort(vals.begin(), vals.end(), abscomp);
    print_separated(vals);  // print_separated defined as before
}
```

The output is `-1 1 2 2 -2 -7 8 -8 8 8` as expected.

However, there's a subtle issue with this code. The function `abscomp` is really only used as an argument to `sort`, nothing else. So making a full-blown function out of it rather clutters the namespace! Having too many globals is very messy. (See discussion of the `static` keyword below.) And we even had to cook up a name for it, "`abscomp`", which is too much effort, when all you really need is some sort of *throwaway* single-use function that compares by absolute value.

This is where **anonymous functions** come in. You can create a "function" in C++ that doesn't have a name, unlike something like `abscomp` above. The syntax is:

```
[&](ARGUMENTS) -> RETURNTYPE {
    FUNCTION BODY
}
```

Note that nowhere do we write a *name* for this function. It's just a raw function without a name, hence *anonymous*.

Anonymous functions are also called *lambda expressions* for historical reasons.

Note that it's an *expression*, not a *statement*, just like `123` or `(x ? "foo" : "bar")` are expressions, which means they represent objects that can be passed around.

For example, we can rewrite our absolute-value sorting function as follows:

```
int main() {
    vector<int> vals = {2, -7, -1, 8, 2, -8, 1, 8, -2, 8};
    sort(vals.begin(), vals.end(), [&](int a, int b) -> bool {
        return abs(a) < abs(b);
    });
    print_separated(vals);
}
```

We passed the anonymous function directly to `sort`, and didn't need to come up with a name!

You can also make an anonymous function without specifying the return type. The compiler infers the return type for you automatically:

```
sort(vals.begin(), vals.end(), [&](int a, int b) {
    return abs(a) < abs(b);
});
```

This is the more common way people write anonymous functions because it's more convenient.

11.1 Anonymous functions are expressions

Note that anonymous functions are *expressions*; that is, they represent a value. It's just a more complicated value than integers or strings, since it represents an object that can be "called" like a function. But it's still a value, so it behaves like any other value. For example, I can assign them to variables like this:

```
auto diffsquare = [&](int a, int b) {
    return (a - b) * (a - b);
};
cout << diffsquare(20, 30) << '\n'; // 100
```

Note that the expression itself is still *anonymous*, it's just that it has been assigned to a variable named `diffsquare`.

You'll notice that I used the keyword **`auto`**, which is me telling the compiler to infer the type for me because I'm lazy. But what if you want to write it out explicitly? Well, here's how you write it:

```
function<int(int,int)> diffsquare = [&](int a, int b) {
    return (a - b) * (a - b);
};
```

But I suggest sticking with **`auto`**.

You'll also notice that there's a semicolon after the closing bracket. Unlike declaring functions normally, this is required; omitting it is the same as omitting the semicolon at the end of the following line:

```
int x = 11;
```

11.2 Capturing variables

Note that anonymous functions can refer to variables outside of it, as long as they're in the same scope.

As an example, here's a function that takes in a graph from `cin`:

```
pair<int,vector<vector<int>>> get_graph() {
    int n, e; cin >> n >> e;
    vector<vector<int>> adj(n);
    auto add_edge = [&](int i, int j) {
        adj[i].push_back(j);
        adj[j].push_back(i);
    };
    while (e--) {
        int a, b; cin >> a >> b;
        add_edge(--a, --b);
    }
    return {n, adj};
}

int main() {
    auto [n, adj] = get_graph();
    cout << solve(n, adj) << '\n';
}
```

Note that the anonymous function `add_edge` refers to the variable `adj` outside of it, and even modifies it.

Here's another example, a function that can sort numbers to their values mod m (breaking ties according to actual values):

```
using ll = long long;

template<class T>
void sort_mod(vector<T>& vals, ll m) {
    auto mod = [&](T v) {
        return (v % m + m) % m; // ensures the output is in [0, m)
    };
    sort(vals.begin(), vals.end(), [&](T a, T b) {
        T d = mod(a) - mod(b);
        return d != 0 ? d < 0 : a < b;
    });
}

int main() {
    vector<int> vals = {2, -7, -1, 8, 2, -8, 1, 8, -2, 8};
    sort_mod(vals, 4);
    print_separated(vals);
}
```

The output is `-8 8 8 8 -7 1 -2 2 2 -1` as expected. (We sorted the numbers mod 4.) This time, we even referred to another anonymous function inside our anonymous function (the function `mod`).

Exercise 11.1. Given an array A of length n, create an array of indices $[0, 1, \ldots, n-1]$ and sort them according to the corresponding values in A. In other words, create a permutation p of $[0, 1, \ldots, n-1]$ such that $A[p[i]] \leq A[p[i+1]]$ for each $i \in \{0, 1, \ldots, n-2\}$.

You can even have nested anonymous functions, and they'll be able to access variables at different functions, as long as they're in scope!

```
void foo() {
    int n = 10;
    auto bar = [&]() {
        int m = n + 10; // can access n here
        auto baz = [&]() {
            m += n; // can access n and m here
        };
    };
}
```

Note that you can explicitly tell the compiler which variables you want to capture by listing it inside the brackets instead of just writing `[&]`. You can also tell it which variables to capture by *value* or by *reference*. For example,

```
void foo() {
    int n = 10;
    auto bar = [n]() { // n is captured by value
        int m = n + 10;
        auto baz = [n, &m]() { // m is captured by reference
            m += n;
        };
    };
}
```

However, this isn't needed in most cases, so I suggest just always writing `[&]`, which tells the compiler to "capture everything that needs to be captured" (by reference).

11.3 Recursive anonymous functions

By the way, you can have recursive anonymous functions too. You just have to declare their type explicitly. For example, here's a function that returns the nth term in a generalized Fibonacci sequence whose first two terms are a and b:

```
const ll mod = 1'000'000'007;
ll genfib(int n, ll a, ll b) {
    function<ll(int)> fib = [&](int n) {
        return n == 0 ? a : n == 1 ? b : (fib(n - 1) + fib(n - 2)) % mod;
    };
    return fib(n);
}

int main() {
    cout << genfib(4, 2, 5) << '\n';  // 19 (the sequence goes: 2 5 7 12 19 ...)
}
```

If you tried to declare it as **auto** `fib` instead, the compiler will complain. Due to technical reasons, it isn't able to infer the correct types of recursive anonymous functions, so you have to declare it.[12]

Here's another example, a program that computes the number of connected components of a given undirected graph in the input.

```cpp
int comp_count(int n, const vector<vector<int>>& adj) {
    vector<bool> vis(n);
    function<void(int)> dfs = [&](int i) {
        assert(!vis[i]); // sanity check
        vis[i] = true;
        for (int j : adj[i]) {
            if (!vis[j]) dfs(j);
        }
    };
    int compc = 0;
    for (int s = 0; s < n; s++) {
        if (!vis[s]) {
            compc++;
            dfs(s);
        }
    }
    return compc;
}

int main() {
    auto [n, adj] = get_graph();  // defined as before
    cout << comp_count(n, adj) << '\n';
}
```

[12] Other languages have smarter compilers/interpreters that can infer types even for recursive functions.

12 How to debug better

Still unwritten. If you want to learn about this, let us know in Discord.

Notes for now:

- As you probably already know, debugging C++ is a nightmare. The fact that `g++` usually spits out garbage error messages doesn't help things.
- "Binary search debugging."
- Use a debugger.
- Use `assert`s to verify invariants and expectations.
- Stress test. Hence, learn how to generate random input.

13 Compiler flags

Still unwritten. If you want to learn about this, let us know in Discord.

Notes for now:

- Specifying the version: `-std=c++17` or `-std=c++23` (You should already know this.)
- Optimization flags: `-O3`.[13]
- Debugging flags: `-Wall` and `-fsanitize` and related
- Compile-time symbols: `-D` (will be explained later)
- There's a CF tutorial about these flags linked in the Python to C++ transition guide. It has a full compilation command which includes all the recommended compiler flags. It's very long, so put it in a local (bash) script so you don't have to type it every time. You can pass arguments to a bash script, e.g.,

```
set -e

echo "The first argument is $1"
echo "The second argument is $2"

# you can use arguments in commands, like this:
g++ -O3 -std=c++17 -DDEBUG=$2 $1.cpp -o $1.exe
```

Then, if you save this as, say, `compile.sh`, and then run

```
./compile.sh sol 1
```

then the script will print out

```
The first argument is sol
The second argument is 1
```

and run the following `g++` command:

```
g++ -O3 -std=c++17 -DDEBUG=1 sol.cpp -o sol.exe
```

- Learn the exact compile command of the judge.
- IOI has a local compile script (named `compile.sh` in some past years) which (almost always) compiles your program the same way the judge does. Use it instead of running `g++` yourself.
- IOI also has a run script (named `run.sh` in some past years) which usually runs your program the same way the judge does. Use it instead of running your program yourself.
- Learn to use these things during the practice round!

[13] Among other things, this optimizes your code so that, for example, some variables may even disappear completely, e.g., if their completely unneeded, or their roles can be performed by different equivalent code, or be taken on by other variables.

14 The DRY principle

Still unwritten. If you want to learn about this, let us know in Discord.

Notes for now:

- "DRY" is "don't repeat yourself," a useful principle in programming, not just competitive programming but real-world programming too.
- Wikipedia states it as "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." It's not very catchy though.
- If you find yourself copy-pasting code (and potentially modifying each copy slightly), then it's a code smell—an indication that you might be doing something wrong.
- Such duplicated code is usually bad; it's harder and more time-consuming to edit, more error-prone, and bloats your program making it harder to read and debug.
- Your language always strives to provide tools and features to remove the need for copy-pasting, e.g., loops, functions, `const` variables/values, templates, lambdas, abstraction, etc.
- Don't write the modulus every time! Do something like `const ll mod = 1'000'000'007;`
- If you've learned about macros elsewhere, **avoid using them**! They're not safe.

15 The `using` keyword

Still unwritten. If you want to learn about this, let us know in Discord.

Notes for now:

- `using namespace std;` tells the compiler to use the namespace `std` (duh). This is also the namespace used by the STL. Using other namespaces isn't needed much in competitive programming. It's more important for the real world.
- `using ll = long long;` to alias a type. Another important use especially in graph problems: `using edge = tuple<int,int>;`

16 More on STL

Still unwritten. If you want to learn about this, let us know in Discord.
Notes for now:

- `binary_search`, `lower_bound` and `upper_bound`
- erase-remove idiom
- `erase(unique(vec.begin(), vec.end()), vec.end())`
- Related to the above: sort-uniq pattern
- `accumulate` and `partial_sum`
- `fill`, `fill_n`, `iota`
- `copy`
- `swap(x, y)`
- `swap` vs `clear`
- `begin` and `end`, `rbegin` and `rend`
- `find`, `find_if`
- `for_each`
- multisets
- `erase()` vs `erase(find())` (I think the difference is more important in multisets?)
- Iterating through sets and maps
- `shuffle` (and not `random_shuffle`)
- quirks of `vector<bool>`
- `bitset`
- `array`
- `vector`: `.reserve`, `.resize`
- Fast I/O: `ios_base::sync_with_stdio(false);` and `cin.tie(NULL);` and what they do.
- Many more, too many to enumerate. Read the docs sometime!

17 Assertions

Still unwritten. If you want to learn about this, let us know in Discord.

Notes for now:

- `assert(x);` crashes the program unless `x` is true, in which case it simply lets the program continue.
- Useful for debugging.
- Also useful in real-world programming.
- Side note: In the real world, `assert` statements are only meant to catch programmer errors/mistakes. They're not meant to be used to validate user-provided input! "Exceptions" are more appropriate for that.

18 Policy-based data structures

Still unwritten. If you want to learn about this, let us know in Discord.
Notes for now:

- There's a CF post about it.
- Ask Vernon since he has more experience with it.

19 the `static` keyword

Still unwritten. If you want to learn about this, let us know in Discord.

Notes for now:

- Too many globals is bad. It invites more errors and makes code more confusing.
- You can sometimes use **`static`** to move some globals into functions. They turn into variables that are shared across different function calls. For example, in

```
int _temp[1 << 22];
void riffle_shuffle(vector<int>& a, int n) {
    for (int i = 0; i < n; i++)
        _temp[i] = a[i];
    int j = 0;
    for (int b : {0, 1})
        for (int i = b; i < n; i += 2)
            a[j++] = _temp[i];
}
```

the `_temp` array is really only being used by `riffle_shuffle`, but is nonetheless visible to other functions. Instead, you can do this:

```
void riffle_shuffle(vector<int>& a, int n) {
    static int _temp[1 << 22];
    for (int i = 0; i < n; i++)
        _temp[i] = a[i];
    int j = 0;
    for (int b : {0, 1})
        for (int i = b; i < n; i += 2)
            a[j++] = _temp[i];
}
```

The `_temp` variable is still shared across different calls to `riffle_shuffle` so it doesn't get created every time it's called, but now it's not visible outside this function!

- There are other places where **`static`** is used. They don't matter too much, though.

20 Dark arts with `pragma`

Still unwritten. If you want to learn about this, let us know in Discord.
Notes for now:

- We're now doing dark magic, so be careful.
- You can pass things to the compiler through your code with `#pragma`
- I think there's a CF post about this.

21 C++'s "range based" paradigm

Still unwritten. If you want to learn about this, let us know in Discord.

Notes for now:

- Notice that all functions in STL that require a range always ask for "pointers" to the two endpoints of the range.
- Technically, they're *iterators*, which you can think of as fancy pointers.
- This is why you see `x.begin()` and `x.end()` all the time.
- You can do arithmetic with them, e.g., `x.begin() + 5` or `x.end() - 5`.
- You can do it with plain arrays, e.g.,

```
1    int x[n];
2    sort(x, x + n);
3    reverse(x, x + n);
```

  The unifying theme is that the two things you pass must be pointer-like: they can be dereferenced, and you can do pointer arithmetic on them.

- I think you can use `begin(x)` and `end(x)` to deal with plain arrays and STL-based objects uniformly; `begin(x)` and `end(x)` works with both I think. I think it turns into `x.begin()` and `x.end()` for STL-based objects and into `x` and `x + n` (where `n` is the size of `x`) for plain arrays. (I think it gets `n` with `sizeof`.)

22 Preprocessor directives

Still unwritten. If you want to learn about this, let us know in Discord.

Notes for now:

- Preprocessor directives are the lines that begin with `#`.
- They're processed during compilation, before your program is even parsed like a C++ program. In other words, during the phase when your program is still mostly seen as just a sequence of symbols, and before the phase when your program's syntax tree is created.
- `#include` is a preprocessor directive. Roughly, it copy-pastes code from another file.
- You can pass/define symbols in the compilation command and check that they're defined via `#ifdef SOMETHING` then closed by `#endif`. The code inside is treated as text and is only included in your program if the `SOMETHING` symbol is passed.
- `#ifndef` to check if something is *not* defined.
- Useful for debugging locally, e.g., you can put debugging code between `#ifdef DEBUGGING` and `#endif` and compile your code with `-DDEBUGGING`. Since the judge (probably) won't add the symbol `DEBUGGING`, the debugging code won't be compiled in the judge's machine. In other words, you don't need to manually delete/comment out debugging code before submitting; you can just submit it as is. Since it's totally ignored, there are no performance penalties.
- You can also pass values, e.g., `-DVAL=5`. Then you can use `VAL` in your code and the compiler will treat it as if you typed `5` instead.
- The above is useful if you want to "parameterize" your code, that is, change some part or parts of it without editing the file every time. For example, this is useful when optimizing the parameter of your parameterized algorithm, such as the chunk size of your sqrt decomposition. This makes it easier to ternary search/binary search for the optimal parameter. You can combine this with bash scripting to automate more of this process. (It's possible to completely automate it, but it's technical.)
- The `#define` directive lets you create macros. **Avoid macros.**
- IOI's local compiler and/or judge compiler usually pass some `-D` symbol; read the official announcements and/or read the compiler script to determine the exact symbol names. This is included for your convenience, so that you don't have to do it yourself, and you can just use `#ifdef` or `#ifndef` on these symbols.
- Many other contests and online judges also pass their own symbol, e.g., `-DONLINE_JUDGE`.

23 Basics of object-oriented programming

Still unwritten. If you want to learn about this, let us know in Discord.

Notes for now:

- The basic terms of object-oriented programming (OOP) are "classes" and "objects". Classes correspond to structs, while objects correspond to instances of structs.
- Classes have attributes and methods. Attributes are like struct attributes. Methods are like functions but attached/bound to struct instances. (We've already seen these used above.)
- A constructor is called when an object is created. A destructor is called when an object is destroyed/deallocated. (E.g., via **`delete`**, or when a local variable is destroyed when a function call returns.)
- Classes have a feature called *visibility*: the programmer can choose which attributes and methods are visible to outside code, so it may hide attributes and methods that are merely "implementation details" and are not meant to be changed/used. This prevents them from potentially destroying the internal consistency of objects.
- The following two things can be moved to an appendix section:
  - There's a thing called inheritance; classes can be "subclasses" of other classes, and they inherit attributes and methods of the superclass. More useful in the real world, though sometimes convenient in contest code.
  - There's also a thing called polymorphism.
- Static methods are a slightly different deal. You don't need to worry about them much.

23.1 C++ classes

Still unwritten. If you want to learn about this, let us know in Discord.

Notes for now:

- There is a keyword called **`class`** more suited for OOP. You can think of it as a fancy **`struct`**.

Some sample code for now:

```
class Something {
private:
    int a, b;
    int *v;

public:
    // constructor
    Something(int a = 0, int b = 0): a(a), b(b) {
        v = new int[10];
        for (int i = 0; i < 10; i++)
            v[i] = i * i;
    }

    // destructor
```

```
15      ~Something() {
16          delete x;
17      }
18
19      // this allows access to v. but there's no way to edit it.
20      int get(int i) {
21          return v[i];
22      }
23
24      // public method that calls a private method
25      int x(int c, int d) {
26          _x(min(c, d), max(c, d));
27      }
28
29  private:
30      // private method. can't be called by outsiders
31      void _x(int _a, int _b) {
32          a = _a;
33          b = _b;
34      }
35  };
36
37  // too lazy to add code demonstrating inheritance and polymorphism
38  // we'll probably want to move those in the same appendix section too
```