1 Introduction, and Why C++?

This guide assumes that you have some experience with competitive programming, and that you are already familiar with another programming language such as Python, Java, JavaScript, etc. We will cover the basics of C++ and how to use it in USACO (and other contests).

First of all, why C++? The fundamental reason is that competitive programmers like to go fast. C++ runs substantially quicker even than its fellow compiled language Java, and *much* quicker than interpreted languages like JavaScript and Python.[1]

The practical reason is that many USACO problems at the Gold level and above simply can't be fully solved in any other language, regardless of what the USACO platform supports. (And as of very recently, the IOI now *only* supports C++!) Although the USACO has recently taken steps to make Silver problems solvable in Python, it's unlikely that the USACO/IOI organizers will make serious efforts to make the hardest problems solvable in slower languages.[2]

Learning and using C++ can be frustrating at times – one of the authors of this guide prefers to use Python in everyday life. But it's even more frustrating to know the best answer for a problem and still not be able to pass it! You need to be able to use the right tool for the job, and once you're far enough up the USACO ladder, that tool is almost always C++.

Even if you eventually end up working at, say, a software engineering job where you're mostly using some other language, it's still well worth knowing C++. It teaches you to think "close to the metal", so to speak, writing in a way that's closer to machine code at the cost of some of the handholding of higher-level languages. Get ready for (figurative) sparks to fly... and you may also hear some horrible scraping and shrieking sounds from time to time!

We're on this journey together. If you find that the document is missing information on an aspect of C++ that you're curious about, please let Ian know, and we will (eventually) add it!

2 Basics

2.1 Program Structure

A very basic C++ program looks like this:

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

using namespace std;

int main() {
  // code goes here
  return 0;
}
```

The first line imports every standard library, including many useful functions and data structures. This saves you the trouble of directly importing individual libraries such as `algorithm`, `vector`, etc. We would not recommend using such a file in daily programming life or on a job; it adds to compilation time and obscures what your program actually depends on. It's like wearing everything in your closet at once so you're ready for anything. But in the

[1] There *is* a version of Python called PyPy that narrows the gap a lot, essentially by rewriting the code into C. It's available on Codeforces and other contest platforms, but not for USACO.

[2] There is often no way to set time limits that allow a clever Python solution to pass while making a brute-force C++ solution time out. Giving Python some extra wiggle room (like USACO's extra $2\times$ time factor) just isn't enough and doesn't work consistently, and coming up with separate time limits for every language is, frankly, an unreasonable burden for teams that are often composed of volunteers.

© 2024 AoPS Incorporated

context of competitive programming, this file is a big time-saver. See our setup guide for advice on getting this working on your machine.

The `using namespace std;` line allows you to use functions and data structures from the standard library without having to prefix them with `std::`. Again, the less you have to think about besides the contest problem itself, the better!

The `//` lines denote a comment. To get a comment spanning multiple lines, enclose the text in `/*` at the start and `*/` at the end. Of course, you don't need to write comments for USACO, but the comment syntax can be great for temporarily disabling lines or chunks of your program if needed as you debug.

The `main()` function is where your code goes.

The `main` function is required to return an integer. By convention, a return value of 0 indicates that the program ran successfully, while any other value indicates an error.

2.2 Input and Output

The standard library provides several functions for input and output. The `cin` and `cout` functions are used for input and output, respectively. [3] For example, the following code reads two integers from standard input and prints their sum to standard output. (We'll drop the `#include <bits/stdc++.h>` and `using namespace std;` lines from here on out just for brevity.)

```
int main() {
  int a, b;
  cin >> a >> b;
  cout << a + b << "\n";
  return 0;
}
```

The `cin` function reads from standard input, and the `cout` function writes to standard output. The >> and << operators are used to read and write data, respectively.[4] If you mix up the directions, it can help to think of << (output) as being two megaphones blaring the output.

Reading input with >> is one of the rare examples of something being *easier* in C++ than in other languages. `cin` knows the data types of the things you want to read in, and so it will consume whitespace (including newline characters!) from standard output until it finds something it can treat as that datatype. You don't need to explicitly tell it to read newlines or spaces.

You may have seen the `endl` keyword, which is used to print a newline. However, this is actually less efficient than printing a newline character `\n` directly, as the `endl` keyword also flushes the output buffer, which is unnecessary (and slow) in most cases. It's like fully cleaning a bathroom after each time you use it!

C++ has another way to read input from (and write output to) text files rather than standard input and output. For USACO, you will only need to do this if you practice with problems from before the 2020-2021 season, back in the day when USACO expected you to read from one "`.in`" text file and write to another "`.out`" text file. On those problems, `freopen` is super helpful and painless:

```
freopen("bessie.in", "r", stdin);
freopen("bessie.out", "w", stdout);
```

Once you've done this, C++ will know that your input is coming from `bessie.in` (rather than standard input) and your output is going to `bessie.out` (rather than standard output). After putting in just those two lines, you can use `cin` and `cout` exactly as we discussed earlier!

[3] There are other, older, C-style ways of reading and writing, but they are a little less elegant.

[4] These operators are actually just overridden versions of the bitwise shift operators – more on those later – but they are used for input and output in C++.

© 2024 AoPS Incorporated

2.3 Variables

Variables in C++ are declared using the `type name` syntax. For example, the following code declares an integer variable named `a`:

```
int a;
```

Unlike in some other languages, C++ will *not* automatically initialize this variable to a default value such as 0. In fact, `a` could be holding just about anything after being declared in this way – whatever old garbage happened to be in the memory address that C++ picked for it. If you accidentally *use* this garbage instead of initializing the variable yourself, this can cause very frustrating behavior where your program happens to run correctly sometimes, and fails other times, since the garbage in memory may be different on different runs of the program!

The `int` type is used to store 32-bit integers, with values roughly in the range $[-2 \cdot 10^9, 2 \cdot 10^9]$.[5] Other common types include `long long` for 64-bit integers[6], `double` for double-precision floating point numbers, and `string` for strings.

In competitive programming, you should be on the lookout for cases in which an arithmetic operation on two `int`s can produce a value with (positive or negative) magnitude that is too large to fit in an `int`. In this case, instead of throwing an error, C++ will silently *overflow* the result, giving you an answer you don't want. Don't let it do this! Use `long long`s if there is any doubt.

We'll have more to say about overflow later, but it's so important that it's worth mentioning twice – if your code is working on test cases with small numbers and mysteriously failing on cases with large numbers, this may be the culprit.

You can use the `auto` keyword to have C++ automatically infer the type of a variable. For example, the following code declares a variable named `a` with type `int`:

```
auto a = 10;
```

The `auto` keyword is especially useful when dealing with complex types such as iterators, pointers, and lambda functions (not mentioned in this guide).

2.4 Vectors, Pairs, and Tuples

The standard library provides several useful data structures. The `vector` type is used to store a list of elements. For example, the following code declares a vector named `v` that stores integers:

```
vector<int> v; // v = {}
```

You can use the `push_back` function to add an element to the end of a vector. For example, the following code adds the integers 10 and 20 to the end of `v`:

```
v.push_back(10); // v = {10}
v.push_back(20); // v = {10, 20}
```

The `pop_back` function removes the last element of a vector:

```
v.pop_back(); // v = {10}
v.pop_back(); // v = {}
```

It is actually possible to erase elements from anywhere in a `vector`, not just the back end... but be forewarned that this can be very slow. See Handout 1 for more details.

The `size` function gets the number of elements in a `vector`:

[5] An `unsigned int`, on the other hand, can represent only positive values, but can go as high as $\approx 4 \cdot 10^9$.

[6] You may wonder what happened to `long`. The answer has to do with changes in computer storage space over the years. Nowadays `long`s are the same size as `int`s, and there is no reason to use just `long`s. Java programmers, don't forget to type that second `long`!

© 2024 AoPS Incorporated

```
cout << v.size() << "\n";
```

You can also initialize a `vector` with a given size and value. For example, the following code declares a `vector` named `v` with 10 elements, each initialized to -1:

```
vector<int> v(10, -1); // v = {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}
```

You can use the `vector` type to store any single type of data, including other `vector`s. For example, the following code declares a `vector` named `v` that stores `vector`s of integers; you can think of it as representing a two-dimensional grid (although it would be in a more linear shape in memory).

```
vector<vector<int>> v; // v = {}
```

Be forewarned that you need to actually build up those inner `vector`s before trying to store something in them. That is, you can't say, e.g., `v[0][0] = 5;` right after the above code, because there is no 0-th inner `vector` yet – all C++ knows is that the outer `vector` will hold `vector`s.

If you know how many inner `vector`s you want in advance – say, 10 – you can build up the `vector` of `vector`s as follows:

```
vector<vector<int>> v; // v = {}
for (int i = 0; i < 10; i++) {
  vector<int> vv;
  v.push_back(vv);
}
```

The `pair` type is used to store a pair of elements, not necessarily of the same type. For example, the following code declares a `pair` named `p` that stores an `int` and a `string`, in that order:

```
pair<int, string> p;
```

You can use the `make_pair` function to create a pair. For example, the following code creates a `pair<int, string>` named `p` with the first element equal to 10 and the second element equal to `"twenty"`:

```
auto p = make_pair(10, "twenty");
```

The `first` and `second` fields access the first and second elements of a `pair`:

```
cout << p.first << " " << p.second << "\n";
```

Otherwise, you can use the `auto` keyword to deconstruct a `pair` into its two component pieces. For example, the following code deconstructs `p` into two variables named `a` and `b`:

```
auto [a, b] = p;
```

The `tuple` type extends the notion of a `pair` to multiple elements, and saves you from having to make `pair`s with `pair`s in them (unless you want to). For example, the following code declares a `tuple` named `t` that stores two `int`s and a `double`, in that order:

```
tuple<int, int, double> t;
```

`tuple`s can be created and deconstructed in the same way as `pair`s.

2.5 Ordered vs. Unordered Sets and Maps

The standard library provides two types of maps and sets: ordered and unordered, according to how they store their values internally.

Ordered maps and sets are implemented using self-balancing binary search trees, which are efficient and complex data structures that would be too difficult to implement by hand during a contest – thank goodness for the

© 2024 AoPS Incorporated

standard C++ libraries! You can iterate through the members of an unordered map or set while being confident that they are in the standard sorted order.

Unordered maps and sets, on the other hand, are implemented using hash tables. These allow for faster access of elements, but you cannot rely on the contents being in any particular order. You may also find that these unordered structures are less convenient in some contexts – if for some reason you want your code to compare two unordered maps, you have to tell C++ how to do it, whereas it already knows how to compare ordered maps because there is an obvious way to compare ordered things.

Now, what *are* `set`s and `map`s? If you're used to Java, you already know more or less how these work. If you're used to Python, you've probably seen its sets, and Python's dictionaries are like unordered C++ `map`s.

The `set` type is used to store a set of *unique* elements – that is, there is at most one copy of any element in the set. For example, the following code declares an ordered set named `s` that stores integers:

```
set<int> s; // s = {}
```

You can use the `insert` function to add an element to a set. For example, the following code adds the integers 10 and 20 to `s`:

```
s.insert(10); // s = {10}
s.insert(20); // s = {10, 20}
```

If we tried another `s.insert(10)`, it would *not* add an additional 10 to the `set`.[7]

The `erase` function removes an element from a `set`, if that element is present. (If you try to `erase` an element that is not in the `set`, nothing happens.) For example, the following code removes the integer 10 from `s`:

```
s.erase(10); // s = {20}
```

One way to check if an element is in a set is to use `count`. For example, the following code prints `1` if the integer 10 is in `s` and `0` otherwise. (Since elements of a set are unique, the only possible results of `count` are 0 and 1, which may feel a bit strange!)

```
cout << s.count(10) << "\n";
```

You can use the `size` function to get the number of elements in a set:

```
cout << s.size() << "\n";
```

Now, on to maps. These have keys, each of which is associated with one stored value. (You may have run into the term "associative array" for a map.) Think of keys as being like words in a dictionary and values as being like definitions; all the keys have to be distinct words, but there could be two or more words with the same definition.[8]

For example, the following code declares an ordered map named `m` that maps strings to integers:

```
map<string, int> m; // m = {}
```

Getting and setting values in a map is similar to getting and setting values in a vector:

```
m["abc"] = 10; // m = {"abc": 10}
cout << m["abc"] << "\n"; // prints 10
```

If you ask a map for a key it does not have, it will return the default value for its data type.

As with `set`s, you can use the `count` function to check if a key is in a map. The following code prints `1` if the key `"abc"` is in `m` and `0` otherwise:

[7] If you do want to be able to have multiple instances of the same value, there is a `multiset` library, but this may not actually be the most efficient tool for your job.

[8] Another way to view a set is as a map with just keys and no values.

© 2024 AoPS Incorporated

```
cout << m.count("abc") << "\n";
```

An unordered set is declared using the `unordered_set` type, and an unordered map is declared using the `unordered_map` type. While `unordered_set`s and `unordered_map`s can do lookups in constant ($O(1)$) time, the $O(\log n)$ time of lookups in ordered `set`s and `map`s may be faster in practice.[9] We recommend using ordered `set`s and `map`s by default unless you have a compelling reason.

2.6 Functions

Functions in C++ are declared using the `return_type name(parameters)` syntax. For example, the following code declares a function named `f` that takes in two `int`s as inputs, and returns another `int`.

```
int f(int x, int y) {
  return x + y;
}
```

The `return_type` is the type of the value returned by the function. For functions that don't return a value, the `void` type is used.

```
void f(int x, int y) {
  int z = x + y;
}
```

A function like the above would be useless, since the result of the calculation would not be stored anywhere, and the value `z` would go away at the end of the function call (more on this later). So you may wonder what the point of `void` is! But we'll see examples where we pass in data structures that can be permanently modified in-place by the function, without needing to return anything.

2.7 References

References are used to create aliases for variables. For example, the following code declares a variable named `a` and a reference named `b`:

```
int a = 10;
int& b = a;
```

The value of `b` is always equal to the value of `a`, including when `a` is modified.

```
cout << a << " " << b << "\n";
a = 20;
cout << a << " " << b << "\n";

// prints:
// 10 10
// 20 20
```

Suppose we had a problem that required us to swap two variables. We could implement a swap function that takes two references as parameters:

```
void custom_swap(int& a, int& b) {
  int temp = a;
  a = b;
  b = temp;
}
```

[9] See Handout 1 if this big-O notation is unfamiliar.

© 2024 AoPS Incorporated

This function can be then used to swap the first two elements of a `vector`:

```
vector<int> v = {10, 20, 30};
custom_swap(v[0], v[1]);
// v = {20, 10, 30}
```

This function `custom_swap` is identical to the C++ standard library function `swap` that swaps two values.

References are also useful when deconstructing pairs and tuples:

```
auto [a, b] = p; // copies the values of p into a and b
auto& [a, b] = p; // creates references to the values of p
```

Creating references is more efficient than copying values, because references don't create new variables in memory. One cause of slow code can be that you're silently making a bunch of copies that you don't intend to make!

2.8 Range-Based `for` Loops

The syntax `for(auto x : a)` is called a *range-based for loop*, and it iterates through the elements of a collection `a`. Since a `map` is a collection of key-value pairs, we can iterate through a `map` using a `for` loop, destructing each pair into a key and a value:

```
map<string, int> m = {{"abc", 10}, {"def", 20}};
for (auto& [key, value] : m) {
  cout << key << " " << value << "\n";
}
```

The syntax `auto& [key, value]` creates references to the keys and values of the map. If we used `auto [key, value]` instead, the keys and values would be copied instead of referenced, which could be very slow!

It's also possible to access the pair directly: `map<string, int> m = {{"abc", 10}, {"def", 20}};`
```
for (auto& p: m) {
  cout << p.first << " " << p.second << "\n";
}
```

A range-based `for` loop can also be used to iterate through `vector`s and `set`s, among other data structures.

```
vector<int> v = {10, 20, 30};
for (auto x : v) {
  cout << x << " ";
}
cout << "\n";
```

This prints: `10 20 30`

If you need the indices for something, though, a standard loop works fine.

```
vector<int> v = {10, 20, 30};
for (int i = 0; x < v.size(); ++i) {
  cout << Element number << i << " is: " v[i] << " ";
}
cout << "\n";
```

See the end of this document for a comparison of various data structure operations in C++, Java, and Python.

© 2024 AoPS Incorporated

3 C++ Quirks And Other Advice

3.1 Fast I/O

Input and output can be slow, and some USACO problems have you read and/or write hundreds of thousands of values! We can use the following code to speed up input and output:

```
ios_base::sync_with_stdio(0);
cin.tie(0);
```

These two lines untie the `cin` and `cout` streams from each other, which cuts down on a lot of extraneous buffer flushing. Most competitive programmers include these lines in every C++ program as the first two lines of their `main` function.

3.2 Initializing Arrays

If you want to initialize an array to all zeroes, you can use the following syntax:

```
int a[10] = {};
```

Unless the array is global (see below) or static (not covered in this document), you must include the $\{\}$. Doing just `int a[10];` will leave the array uninitialized, and the values will be garbage.

Alternatively, you can use a vector instead of an array, which is automatically initialized to zero if you specify a size:

```
vector<int> a(10);
```

3.3 Overflow

Since C++ represents integers using a fixed number of bits, it is possible to *overflow* an `int`, i.e., represent it as a value that is too large to fit in the number of bits allocated to it. For C++, the type `int` is 32 bits, and the type `long long` is 64 bits.

Since these types are also signed, the first bit is used to represent the sign of the number. This means that the maximum value of an `int` is $2^{31}-1$, and the maximum value of a `long long` is $2^{63}-1$. If you try to exceed any of these numbers, the value will overflow and it'll flip the bit representing the sign of the number, which will result in a negative number. This is never what you want (it's hard to imagine a case in which you would *rely* on overflow behavior!)

Overflow also happens when values are sufficiently *negative* (in absolute magnitude) to be represented.

You can cast `int`s to `long long`s, e.g.

```
int a = 2000000000;
int b = 2000000000;
cout << a + b << '\n';
cout << (long long)a + (long long)b << '\n';
```

The first `cout` prints the value -294967296, the result of overflow. The second `cout` prints the value we actually want: 4000000000.

What if we need even bigger values? Java has `BigInteger` for this, and Python handles huge integers with no problems[10] C++ has no library support for this, and – barring *very* unlikely exceptions – USACO problems will not expect you to whip up your own BigInteger class. Instead, you will be asked to use modular arithmetic, which we will handle later in the course.

[10] unless you consider slowness a problem!

© 2024 AoPS Incorporated

3.4 `typedef`s and Macros (`#define`)

`typedef`s and macros are used to make code more readable and to make it easier to change the type of a variable. A very common use case for `typedef`s is to designate `ll` as a shorthand for `long long`, but you can also use any other type or data structure.

Both `typedef`s and macros should be placed at the top of your code, before the `main` function.

```
typedef long long ll;
typedef vector<int> vi;
typedef pair<int, int> pi;
typedef vector<pair<int, int>> vpi;

// ... later ...

ll a = 10;

vpi b;
b.push_back({1, 2});
```

Macros are compiler directives that are replaced with the code that they represent. For example, the following code defines a macro named `MOD`; when the code is compiled, the compiler will replace all instances of `MOD` (anywhere in the code) with the value `1000000007`:

```
#define MOD 1000000007
```

(This specific example is useful when problems ask you to do modular arithmetic, since you would otherwise need to type out `1000000007` many times throughout your program. Again, we will discuss this later in the course.)

Macros are useful for defining constants, but they can also be used to define functions. For example, the following macro defines a shorthand for the `push_back` method of a vector:

```
#define pb push_back
```

The macros can be used as follows:

```
vector<int> a;
a.pb(10);
```

Most competitive programmers will come up with their own set of macros that they use to save on typing. For example, the following code defines a macro named `all` that returns an iterator to the beginning of a vector and an iterator to the end of a vector:

```
#define all(x) x.begin(), x.end()
```

This macro can be used as follows:

```
vector<int> a = {1, 2, 3};
sort(all(a));
```

Be careful with macros – the find/replace way in which they are used can have some strange consequences. For example, can you figure out why C++ will *not* print 32 here (as the user probably expects)?

```
#define MY_CONSTANT 5 + 3

int main() {
  cout << MY_CONSTANT * 4 << '\n';
}
```

© 2024 AoPS Incorporated

Also, make sure you don't put a semicolon at the end of a `#define` line, unless you are absolutely sure that's what you want!

3.5 Bitwise Operations

Some competitive programming problems involve operating on, e.g., a small array of boolean values. If you imagine representing these values as 0s and 1s, then an `int` or `long long` can be used to represent a string of them!

If you haven't encountered binary (base 2) representations before, here's an example. Suppose we want to write the "normal" (decimal) number 25 in binary. Unlike decimal, which has a ones place, a tens place, a hundreds place, etc., binary has a ones place, a twos place, a fours place, and so on. To figure out what 25 is in binary, we can use an algorithm like the following:

- start with an empty string
- while our original decimal number is still greater than 0:
  - if the number is odd, write 1 at the leftmost position of our string; otherwise, write 0
  - divide the decimal number by 2, rounding down

If we try this on 25, we get 11001, which represents $1 \times 16 + 1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1$.

We can go from binary to decimal by doing something like the reverse of the above. One way is as follows:

- start with the (decimal) total 0, and the (decimal) multiplier 1
- go right to left through the binary string, and at each position, do the following:
  - if there is a 1 at the position, add the current multiplier to the total
  - in either case, now multiply the multiplier by 2

So if we have a grid of boolean values like

```
TFTF
FFFT
TTTF
FFFF
```

we can interpret the `T`s as 1s and the `F`s as 0s, getting the binary numbers

```
1010
0001
1110
0000
```

which are the following in decimal:

```
10
1
14
0
```

Each of the above numbers can be stored in one `int`.[11] A 32×32 grid could be stored in just 32 `int`s (though you may consider using `unsigned int`s in this case, to deal only with positive numbers).[12] This is much more efficient than, say, a 2-dimensional grid of `char`s, each of which is `T` or `F`.

[11] In this specific case it seems like it would make more sense to use a smaller data type like `short`. However, it's probably better to just stick with `int`s – in practice, using `short`s may not actually help, since modern architectures probably mostly stuff them into `int`-sized registers anyway.

[12] The following examples all work for `int`s as well, though.

The other reason to do this is that C++ has very fast "bitwise" operations for adding two binary numbers, etc., plus some other useful operations that don't have a direct analogue in decimal numbers. All of these take two binary values of the same storage type (such as `int` or `long long`):

- Bitwise and (`&`): For each position in the two bit strings, put `1` if **both** bit strings have a `1` there, or `0` otherwise.[13]
- Bitwise or (`|`): For each position in the two bit strings, put `1` if **at least one** of the two bit strings has a `1` there, or `0` otherwise.
- Bitwise xor (`^`): For each position in the two bit strings, put `1` if **exactly one** of the two bit strings has a `1` there, or `0` otherwise. ("xor" stands for "exclusive or".)

There is also the bitwise not (`~`), which takes a bit string and replaces every `1` with a `0`, and vice versa.

These operations pair nicely with the bit shift operations: >> (shift right) and << (shift left). (Remember the directions in which the arrows point!) Bit shifting right is like dividing by 2, and bit shifting left is like multiplying by 2. For example, if you have the `unsigned int` `00000000000000000000000000000110` (which is the decimal 6), then shifting right one step produces `00000000000000000000000000000011` (decimal 3), and shifting left two steps produces `00000000000000000000000000011000` (decimal 24). You can represent the latter by:

```
a <<= 2;
```

In fact, you can even write `a >>= 1` to represent dividing by 2, but this just makes your code a little less readable. The compiler will probably turn a line like `a /= 2;` into this bit shift operation anyway!

With combinations of bitwise operations, you can do things like "select the 3rd, 4th, and 5th bits from the right in an `unsigned int`" as follows. Let's try this on the `unsigned int` value `v = 25`, which is `00000000000000000000000000011001`.

- create a *bitmask* of three 1s, e.g., by declaring `unsigned int a = 7;`, which creates the value `00000000000000000000000000000111`.
- shift them to the right place, e.g., via `a <<= 2`. Now we have `00000000000000000000000000011100`.
- bitwise-and this with your value, via `a &= v`. (here `00000000000000000000000000011001`), getting `00000000000000000000000000011000`. Notice that
- shift the bits back to the end so they are easier to examine, via `a >>= 2` (undoing the previous operation).
- Now `a` stores `00000000000000000000000000000110`, telling us that the three bits of interest are `110`.

Because CPUs operate on binary values rather than decimal ones, they are very good at doing these operations fast, so working on the bitwise version of a problem can speed things up a lot. Some contest problems (e.g., those involving dynamic programming) will even require you to use bitwise operations for maximum speed.

3.6 Errors and Debugging

C++ is notorious for gigantic error messages, but when your compiler spews out hundreds of lines of error text, don't despair.

Usually, the bulk of these messages are C++ complaining that you're trying to use one data type where it expects another. To be thorough, it will list all of the interpretations it thought you *might* have meant, and then justify, in painful detail, why none of them work. You can generally ignore most of this and look at the top of the error, where you'll get a sense of where you accidentally used a `pair(int, string)` where you wanted a `pair(string, int)`, or whatever.

[13] Do not confuse this single & with the double &&, which is like Python's `and`.

© 2024 AoPS Incorporated

The best way to deal with more arcane error messages is to Google them. Find a part of the error text that doesn't mention specific variable names etc., and search for it. You'll likely run into tales from other adventurers of the past, sometimes with helpful insights like, "oh, this error message appears to be about X, but it's really about Y".

Even worse is when your code compiles and runs, but crashes with a "Segmentation fault", aka segfault. What this means is that the code tried to access a portion of memory that it shouldn't have – e.g., perhaps you have a `vector` of size 4, but tried to read the fifth element. But knowing this tells you almost nothing about what is going wrong – lots of bugs eventually cause something to go out of bounds.

Your first instinct may be to use "print-based debugging" – have the code output what values are stored in your variables, etc., and hope to spot what is going wrong. This is a surprisingly good debugging strategy in general, but it won't work well in the case of segfaults, since all of this output essentially gets eaten by the segfault and you'll never see it!

One way to get around this is to use the line `exit(0);`, which stops the program immediately. If you insert this somewhere in your program, and the program makes it that far and exits rather than segfaulting, then the segfault is "later on". (But you may not be able to pinpoint a particular line of code – a segfault might not get triggered until your code has run a for loop many times, for instance.) Using `exit(0);`s can let you do print-based debugging up until the point of a successful exit.

Once again, this is not the best advice for C++ programming in general. There are tools such as `gdb` that enable more principled debugging (setting breakpoints, inspecting which variables have which values at which points in execution, etc.). It's up to you to develop a style that works well for you during the heat of a contest.

3.7 Pointers (Optional)

All variables in C++ are stored in memory (RAM), and each variable has a unique memory address. This address is used refer to the specific bits in memory that store the value of the variable.

A pointer is used to store the memory address of a variable. For example, the following code declares a variable named `a` and a pointer named `b`:

```
int a = 10;
int* b = &a;
```

The `&` operator returns the memory address of a variable, and the type `int*` means that `b` is a pointer to an integer.

The `*` operator is used to access the value of the variable that a pointer points to. For example, the following code prints `10`:

```
cout << *b << "\n";
```

You will rarely – if ever – need to explicitly consider the exact memory locations of data. In fact, there is no reason to expect that these will be the same on different runs of the program!

Remember when we mentioned functions that return `void`? We can have a function do permanent work on, e.g,, a vector by passing a reference to it into the function:

```
void set_first_to_zero(vector<int> *v) {
  v->at(0) = 0;
}
int main() {
   vector<int> myvec;
   myvec.push_back(2);
   myvec.push_back(1);
   set_first_to_zero(&myvec);
```

© 2024 AoPS Incorporated

```
    cout << myvec[0] << " " << myvec[1] << '\n';
}
```

This code prints `0 1`. What's going on here?

First, note that `set_first_to_zero` expects to take a pointer (reference) to a `vector`, rather than a `vector`. Accordingly, in `main`, we pass in a reference (`&myvec`) rather than just `myvec`.

Second, what's up with the arrow? Because `v` is a reference to a vector and not a vector itself, we have to "dereference" it before we can use it. That is, if we want to alter a room of a house, we can't do it with just a mailing address written on a piece of paper! We have to follow the address and go to the house and then do our construction work.

The usual way to dereference a reference is `*v`, and here we want to dereference and then access the 0th element. So we might try `*v[0]`, but this is wrong because of the order in which C++ evaluates the `*` versus the `[0]`. We actually need `(*v)[0]` – dereference and then access. But that's a bit cumbersome...

The `->` is shorthand for "dereference a reference and *then* access the method/field of that object." We can't use `->` in conjunction with the access operator `[]`, but there is a method, `.at`, which does the same thing as the `[]` operator. Hence `v->at(0)`, which is the same as `(*v).at(0)`.

This may seem like a lot of work just to make things look a little nicer... but it's very common to dereference a reference to something that is itself a reference to something, etc., so you get nice chains of arrows, without all the stars and parentheses to keep track of.

3.8 Arrays (Optional)

C++ supports arrays that are like `vector`s without the utility methods. Suppose you know in advance that you will need to read in an $N \times N$ grid of integers where $N \leq 500$. You can use the following:

```
#define N 500
int grid[N][N];
int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int n;
  cin >> n;
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      cin >> grid[i][j];
    }
  }
}
```

Notice that although the array was created full of whatever garbage was in memory, we didn't need to initialize the elements because we're immediately overwriting all the garbage with the data we read in from `cin`.

Arrays are holdovers from C (the predecessor to C++), and they can be a little confusing to work with. When you declare an array, it's actually secretly a pointer (!) and has to be treated as such. The array in our example above is actually an `int **` – a pointer to a pointer to an `int`.

`vector`s are basically arrays with some convenient methods added, so they can stand in for arrays if you don't want to deal with arrays. See Handout 1 for a bit more on the limitations of `vector`s.

© 2024 AoPS Incorporated

3.9 Global Variables (Optional)

In the example above, you may have noticed that the array was declared outside the `main()` function. Why would we do that?

Data structures declared outside of `main()` live in a persistent region of memory called the heap (which has nothing to do with the "heap" data structure we'll study in Lesson 1). Something declared this way is in "global scope" – *all* function calls made anywhere within the program can see it.

Unless you use the `new` keyword – which is a critical feature of "real-world" C++ but which you can pretty much ignore for competitive programming – data structures declared inside of `main()` – or within any function call – live in a temporary region of memory called the stack. When the function finishes executing, all the things declared within it go away!

In competitive programming, it is convenient to declare a global data structure that is as big as it could possibly need to be, and then have `main()` and other function calls only work with whatever portion of it is actually needed. (Think about making a restaurant reservation for a million people so that you're fine if any number of friends between 1 and a million friends show up – a jerk move in real life, but fine in competitive programming!)

Another key benefit of this is that if you have helper functions, you don't need to have `main()` pass them references to giant data structures, as described earlier. Everything can just point to the huge object in global scope – the elephant in the room – and it's fine.

You may have heard that global variables should be avoided, and for good reason – they can be a *very* bad idea in a large project with multiple contributors, for example. What if you define a variable called `grid` in global scope, and someone else also does in their code? You can even see hints of this problem sometimes – if you try to define a macro called HUGE, for instance, you'll find that C++ already has such a name, living in global scope. But for competitive programming, in which your program runs alone and is gone in 2 seconds, this is fine.

3.10 Classes

If you're used to object-oriented programming, you're in for a surprise: competitive programmers rarely have a need to define classes. Sometimes you want an object that has behavior, and in this case object-oriented programming is the way to go. You can usually get by with `struct`s, which are ordered packages of fields of data. However, tuples (as mentioned earlier) are even easier to work with, so we won't say more about `struct`s here just yet.

3.11 Iterators (Optional)

If you need to step through the elements of a data structure (ordered or unordered!), iterators provide an alternative to the two kinds of `for` loop we saw above.

For example, the following code prints the elements of a vector:

```
vector<int> v = {10, 20, 30};
for (auto it = v.begin(); it != v.end(); it++) {
  cout << *it << "\n";
}
```

The `begin` and `end` functions return iterators (you can think of them as being like references) to the first and last elements of a data structure. The `++` operator moves an iterator to the next element, and the `*` operator returns the value of the element pointed to by an iterator.

A way to search a map or set with an iterator is:

```
if (mp.find('x') != mp.end()) {
}
```

© 2024 AoPS Incorporated

That is, we run an iterator across our map looking for the key 'x'. If the iterator reaches the `.end()` of the map, then the key is not there.

3.12 The -O2 compiler flags (Optional)

Both our judge and USACO's use the `-O2` optimization flag when compiling. If you compile your code locally to test it, you should make sure to include this flag, since it makes a very noticeable difference in runtime.

3.13 Miscellaneous Gotchas (Optional)

* If you're used to being able to write, e.g., `if a <= x <= b` in Python to check whether x is in between a and b, be aware that this will not work as you expect in C++.

* What is the difference between `x++` and `++x`? The correct answer is that the order determines whether the variable is accessed and then incremented, or incremented and then accessed. Perhaps a better answer is that you should not write dense code – like `(a++) = (++b);` that depends on this ordering, unless/until you're completely comfortable with it and it genuinely helps you code faster.[14] If you emulate the syntax of very advanced competitive programmers without thinking it through, you may be lacking some extra context needed to use it properly. Start by coding in a way that is readable to you – because you will be doing a lot of debugging of your own code! – and use shorthand like an expert once it feels natural.

4 Credits

This document is by Samyok Nepal and Ian Tullis.

5 Comparison of C++/Java/Python Data Structures

[14] In some contests – not USACO or IOI! – your code can be "hacked" by other competitors, and so being more opaque can be useful defensively, although deliberate code obfuscation is not allowed.

© 2024 AoPS Incorporated

Art of Problem Solving is an ACS WASC Accredited School.

Data Struct.	C++	Java	Python
Array	`int a[10];`	`int[] a = new int[10];`	`a = [0] * 10`

Vector	`vector<int> v;`
	`v.push_back(10);`
	`v.pop_back();`
	`v.insert(v.begin()+1, 2);`
		`ArrayList<Integer> v = new ArrayList<>();`
		`v.add(10);`
		`v.remove(v.size() - 1);`
		`v.add(1, 2);`
			`v = []`
			`v.append(10)`
			`v.pop()`
			`v.insert(1, 2)`

Queue	`queue<int> q;`
	`q.push(10);`
	`q.pop();`
	`q.front();`
		`Queue<Integer> q = new LinkedList<>();`
		`q.add(10);`
		`q.remove();`
		`q.peek();`
			`q = deque()`
			`q.append(10)`
			`q.popleft()`
			`q[0]`

Stack	`stack<int> s;`
	`s.push(10);`
	`s.pop();`
	`s.top();`
		`Stack<Integer> s = new Stack<>();`
		`s.push(10);`
		`s.pop();`
		`s.peek();`
			`s = []`
			`s.append(10)`
			`s.pop()`
			`s[-1]`

Deque	`deque<int> d;`
	`d.push_front(10);`
	`d.push_back(20);`
	`d.pop_front();`
	`d.pop_back();`
	`d.front();`
	`d.back();`
		`Deque<Integer> d = new ArrayDeque<>();`
		`d.addFirst(10);`
		`d.addLast(20);`
		`d.removeFirst();`
		`d.removeLast();`
		`d.peekFirst();`
		`d.peekLast();`
			`d = deque()`
			`d.appendleft(10)`
			`d.append(20)`
			`d.popleft()`
			`d.pop()`
			`d[0]`
			`d[-1]`

Map	`map<string,int> m;`
	`m["apple"] = 10;`
	`m.erase("apple");`
	`m.size();`
	`m.count("apple");`
		`Map<String,Integer> m = new HashMap<>();`
		`m.put("apple", 10);`
		`m.remove("apple");`
		`m.size();`
		`m.containsKey("apple");`
			`m = {}`
			`m["apple"] = 10`
			`del m["apple"]`
			`len(m)`
			`"apple" in m`

Set	`set<int> s;`
	`s.insert(10);`
	`s.erase(10);`
	`s.size();`
	`s.count(10);`
		`Set<Integer> s = new HashSet<>();`
		`s.add(10);`
		`s.remove(10);`
		`s.size();`
		`s.contains(10);`
			`s = set()`
			`s.add(10)`
			`s.remove(10)`
			`len(s)`
			`10 in s`

Priority Queue	`priority_queue<int> pq;`
	
	`pq.push(10);`
	`pq.pop();`
	`pq.top();`
		`PriorityQueue<Integer> pq =`
		`new PriorityQueue<>();`
		`pq.add(10);`
		`pq.remove();`
		`pq.peek();`
			`pq = []`
			
			`heapq.heappush(pq, 10)`
			`heapq.heappop(pq)`
			`pq[0]`

Table 1: Comparison of C++, Java, and Python data structures and methods

© 2024 AoPS Incorporated

Thanks to our sponsors: