NOI.PH-YTP Junior Track: C++ Programming 2

Dynamic types and control structures

Aldrich Ellis Asuncion

Contents

1 References, pointers, and dynamic types

Remark 1.1. What follows is a simplified version of learncpp.com – lvalue references and learncpp.com – introduction to pointers.

This handout follows a simplified presentation of references which is sufficient for competitive programming, but is not a fully accurate description of how C++ works. Don't worry, it is normal to learn C++ by slowly refining your understanding of the language. It's near impossible to learn C++ by seeing all the details all at once on your first go.

But for the sake of accuracy, I'll be adding footnotes. **You can ignore all footnotes.**

1.1 References

1.1.1 The basic idea

In C++, a **reference**[1] is an alias for an existing object. Once a reference has been defined, any operation on the reference is applied to the object being referenced. To denote a reference type, we use an ampersand (`&`) in the type declaration:

```
int      // a normal int type
int&     // an reference to an int object
double&  // an reference to a double object
```

```
#include <iostream>
int main() {
    int x { 5 };    // x is a normal integer variable
    int& y { x };   // y is an alias for variable x
    std::cout << x << " " << y << std::endl;  // output: 5 5
    x = 6;
    std::cout << x << " " << y << std::endl;  // output: 6 6
    y = 7;
    std::cout << x << " " << y << std::endl;  // output: 7 7
}
```

Notice that `x` and `y` are now essentially two "variable names" which refer to the same variable. Compare what happened above to if the second variable wasn't a reference.

```
#include <iostream>
int main() {
    int x { 5 };
    int y { x }; // y is a normal integer variable, note the lack of &
    std::cout << x << " " << y << std::endl;  // output: 5 5
    x = 6;
    std::cout << x << " " << y << std::endl;  // output: 6 5
    y = 7;
    std::cout << x << " " << y << std::endl;  // output: 6 7
}
```

[1] Actually, an lvalue reference (pronounced ell-value)

1.1.2 Declaring references

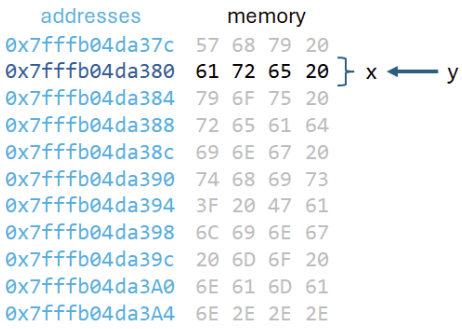
When a reference is initialized with an object, we say it is **bound** to that object. The process by which such a reference is bound is called **reference binding**. The object (or function) being referenced is sometimes called the **referent**.

```
int x { 5 };
int& y { x };
```

Given the above code, we can say the following:

- y is bound to x.
- y is a reference to x.
- x is the referent of y.

Perhaps surprisingly, references are not objects in C++. A reference is not required to exist or occupy storage. If possible, the compiler will optimize references away by replacing all occurrences of a reference with the referent. However, this isn't always possible, and in such cases, references may require storage. Your mental picture of what it means when we say "y is a reference to x" can be something like this:



When you declare a reference, it *must* be initialized to an alias for an existing modifiable variable[2]. In most cases[3], the type of the reference must match the type of the referent. Once initialized, a reference in C++ cannot be *reseated*, meaning it cannot be changed to reference another object.

```
const int x {6};
int& a;     // error: reference not initialized
int& b {5}; // error: 5 is not a variable
int& c {x}; // error: x is constant

double y {3.14};
int z {7};
int& d { y }; // error: reference to int cannot bind to double variable
double& e { z }; // error: reference to double cannot bind to int variable
```

That's all we'll mention about references for now. They might not seem that useful, but are crucial for understanding some later concepts in this handout.

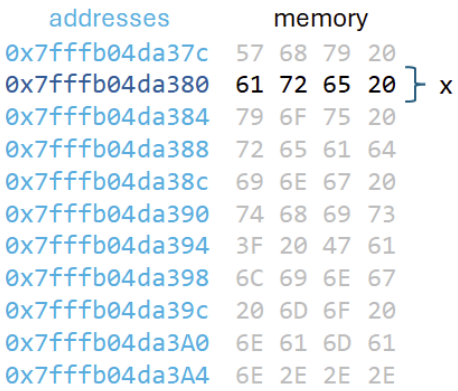
[2] Actually, the rule is "Lvalue references must be bound to a modifiable lvalue." `int& b{5}` fails since `5` is an rvalue, while `int& c {x};` fails since `x` is `const`.

[3] Like pointers, the type of the reference can be a superclass of the type of the referent.

1.2 Pointers

1.2.1 The basic idea

Remember this picture?



When we are working with a variable like `x`, we don't usually care about the *memory address* that `x` is located at. But we do have access to this information.

The **address-of operator** (`&`) returns the memory address of its operand. In other words, `&x` obtains the address of the variable `x`. Try the following code, and you'll get a memory address.

```
int x{ 5 };
std::cout << x << '\n';  // print the value of variable x
std::cout << &x << '\n'; // print the memory address of variable x
```

A **pointer** is a variable which holds a memory address. To denote a pointer type, we use an asterisk (`*`) in the type declaration:

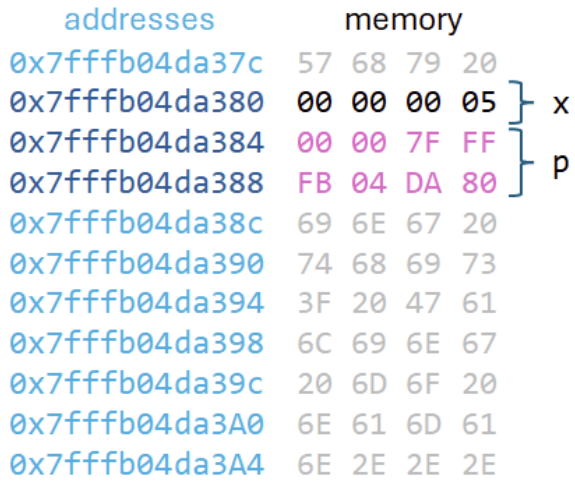
```
int      // a normal int type
int*     // an pointer to an int object (a memory address to an int)
double*  // an pointer to a double object (a memory address to a double)
```

Here is an example.

```
int x {5};
int* p {&x};
```

Notice in the picture[4] on the right that `p` is like a regular variable. It just holds the memory address of `x`.

In this case, we say `p` **points** to `x`.



[4] Picture assumes `int` is 32 bits and pointers are 64 bits. Pointers are basically `unsigned long long`.

1.2.2 The dereference operator

Getting the address of a variable isn't very useful by itself.

The most useful thing we can do with an address is access the value stored at that address. The **dereference operator** (*) (also occasionally called the **indirection operator**) returns the (modifiable) variable[5] at a given memory address.

Try running the following code and following along with what is happening.

```
1  #include <iostream>
2  int main() {
3      int x {5};
4      int y {7};
5
6      // p points to x
7      int* p {&x};
8
9      // print x directly
10     std::cout << x << std::endl;
11
12     // print the address stored in p
13     std::cout << p << std::endl;
14
15     // print the variable at the address stored in p
16     std::cout << *p << std::endl;
17
18     // assign 2 to the variable at the address stored in p
19     *p = 2;
20     std::cout << x << std::endl;
21
22     // p now points to y
23     p = &y;
24
25     *p = 3;
26     std::cout << x << std::endl;
27     std::cout << y << std::endl;
28 }
```

Roughly speaking, here's how to understand the new syntax we've seen:

- `&x` means the location of `x`.
- `*x` means the variable pointed to by `x`.

Notice that pointers are more flexible than references, since we can *change what variable a pointer points to.* (line 23) Additionally, unlike references, pointers do not have to be initialized. Pointers do not actually have to point to a variable. A pointer that does not point to a variable is called a **null pointer**. We can assign the value `nullptr` to a pointer to indicate that it doesn't point to anything.

```
1  int* p {nullptr};
```

[5] The dereference operator returns an lvalue.

Remark 1.2. A single ampersand symbol `&` can mean one of three things in C++:

- When following a type name, `&` denotes a reference: `int& ref`.
- When used before a single value, `&` obtains an address: `&x`.
- When used between two values, `&` is the bitwise AND operator, discussed later in this handout.

Remark 1.3. `nullptr` is a value, just like `true` and `false`. It was introduced in C++11. Before C++11, values such as `0` and `NULL` were used to denote null pointers. These are not recommended for use now.

For this handout, we will not be discussing pointers further, since a lot of the complicated work involving pointers will be handled behind the scenes. Understanding what a pointer is and what the `&` and `*` operations do to a pointer is enough for now.

Remark 1.4. I am leaving out a bunch of introductory C++ stuff, like `new` and `delete`, because they aren't super important to learn early on when you're learning C++ for competitive programming.

If you're looking to learn C++ to develop apps, games, etc. or for a job, you'll want to actually go through the *entirety* of learncpp.com.

1.3 Dynamic types: `std::vector` and `std::string`

1.3.1 The basic idea behind how `std::vector` works

A `std::vector`, defined in the `<vector>` header, is basically a resizable array. Like an array, all the elements of a vector have to be the same type.

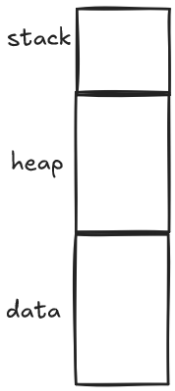
```
// a vector containing 4 integers
std::vector<int> v {8, 4, 5, 9};

// a vector containing 3 doubles
std::vector<double> v {8.0, 4.2, 5.1};
```

We can use the `push_back` function to add elements to the end of a vector, and the `pop_back` function to remove elements.

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v {8, 4, 5, 9};
    // v is now [8, 4, 5, 9]

    v.push_back(6);
    v.push_back(2);

    // Add two more integers to vector
    v.pop_back();
    // v is now [8, 4, 5, 9, 6]
}
```

Normally, when you declare variables and C-style arrays, you basically cannot change their size or move where they are in memory anymore. So how does a `std::vector` work? **What black magic does a vector do?**

Actually, behind the scenes, a `std::vector` creates a C-style array in memory as the program is running. Since it is not known in advance exactly how big the `std::vector` should be before the program is run, we say that the memory is **dynamically allocated.** Dynamically allocated memory goes on the heap segment of your program memory. Remember this diagram?



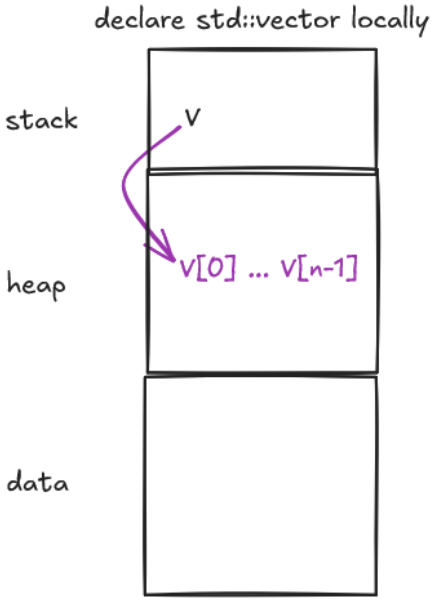
Consider the following code.

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v {8, 4, 5, 9};
}
```

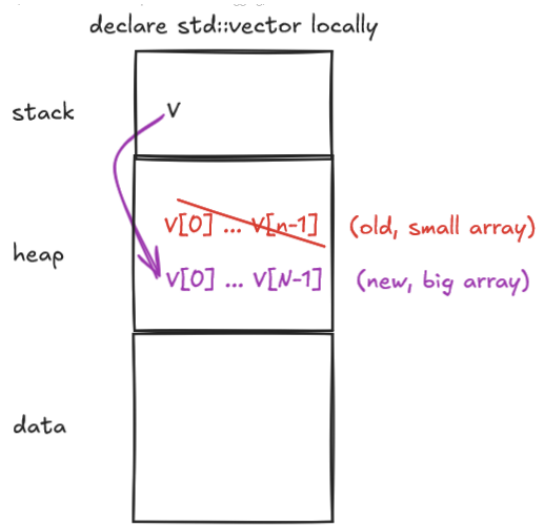
The variable `v` itself is on the stack, since it's declared inside a function. When the vector is created, it creates an array on the heap. The variable `v` just has a pointer to the array on the heap.

The array that is managed by `v` has *fixed-size*, just like a C-style array. Nothing magical there.

You don't have direct access to the array that is being managed by `v`. Even if we think we have 4 elements, it's possible that the array managed by the vector is actually, say, 8 elements long. This gives the vector "room to grow."



Most of the time, when you "add" elements to a vector, you're just making more use of the internal array that the vector manages. But what happens when we add so many elements that the array *runs out of space*? Well, the vector just does the following:



1. Make a new *larger* array on the heap.
2. Copy all the contents of the old array into the new array.
3. Make the vector point to the new array.
4. Discard the old array.

(Friendship ended with small array, now big array is my friend.)

So, a `std::vector` **doesn't actually "grow" like how you think it might.** It just manipulates memory behind the scenes to give the illusion that it can grow and shrink. By the way, this is basically also how lists in Python work.

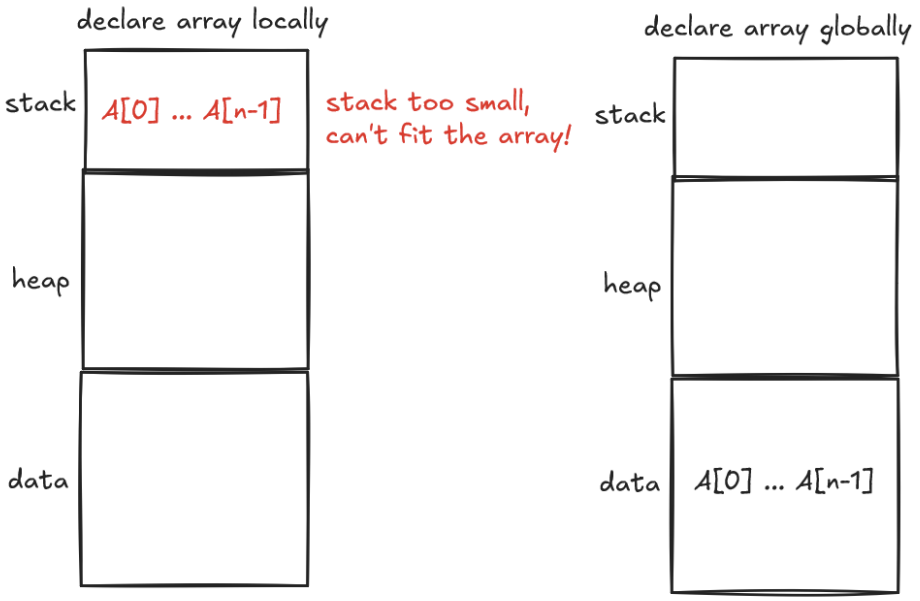
One practical application of this knowledge is as follows. Remember these two programs?

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

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

The first one crashes because local variables are placed on the stack, which has a smaller capacity compared to the heap and data segments.

The second program does not crash because global variables end up on the data segment, which has a large capacity.



To declare a vector with n value-initialized elements, we do something like the following. Note the use of parentheses instead of braces.

```cpp
// declare a vector containing 1000 integers, all set to 0
std::vector<int> V(1000);
```

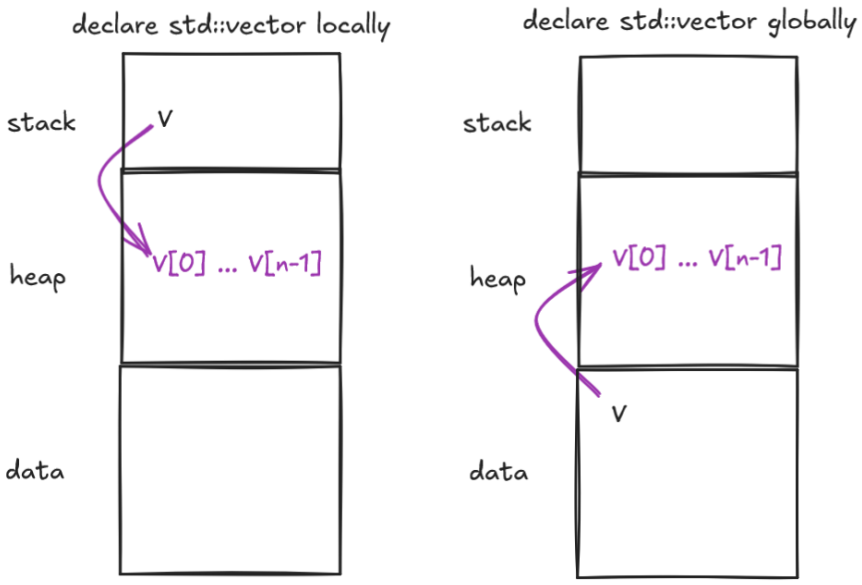
Now, which of the following programs do you think will crash?

```cpp
#include <iostream>
#include <vector>

std::vector<int> V(10000000);

int main() {
    cout << V[0] << endl;
}
```

```cpp
#include <iostream>
#include <vector>


int main() {
    std::vector<int> V(10000000);
    cout << V[0] << endl;
}
```

It turns out, neither of the two programs using vectors crash!

While the vector object would be on the stack or data segment, the dynamically-allocated array that it points to lives on the heap. And in either case, the heap is large enough to carry a large number of elements.



1.3.2 Using `std::vector`

Because a `std::vector` is effectively just a C-style array that is extended on demand, it has the following characteristics.

1. A `std::vector` is a homogenous container, meaning all of its elements must have the same type.
2. A `std::vector` allows fast access to individual elements, just like a C-style array.
3. Elements can only be added to and removed from the end of a `std::vector`, not the beginning of the vector.

Here, we lay out all the general syntax you need to know to use `std::vector` for competitive programming. Full details are of course available in the documentation.

Initialization. We can use braces to initialize the vector with existing elements.

```
std::vector<int> primes { 2, 3, 5, 7, 11 };

// an empty vector
std::vector<int> V {};
```

We can use parentheses to initialize a vector with a certain number of elements.

```
// this vector contains one integer, which is 10
std::vector<int> V1 {10};

// this vector contains ten integers, all 0
std::vector<int> V2 (10);
```

Size queries. A `std::vector` has functions `size()` (returns number of elements) and `empty()` (checks if container has no elements).

```
std::vector<int> primes { 2, 3, 5, 7, 11 };
std::vector<int> V {};

std::cout << primes.size() << std::endl; // output: 5
std::cout << V.size() << std::endl; // output: 0

// The empty() function returns a bool.
// In C++, when a bool is printed using std::cout, true displays as 1 and
↪  false displays as 0.
// (Unlike Python, which prints the words "True" and "False"
std::cout << primes.empty() << std::endl; // output: 0 (false)
std::cout << V.empty() << std::endl; // output: 1 (true)
```

Printing out a vector. We can use the following *range for loop* to print out the elements of a vector. (Range for loops will be discussed in part 2 of this handout.)

```
std::vector<int> v {2, 3, 5, 7, 11};

for (auto& x : v)
{
    std::cout << x << std::endl;
}
```

Accessing elements. The [] operator[6] can be used to access elements of a vector.

```
std::vector<int> primes {2, 3, 5, 7, 11};
std::cout << primes[3] << std::endl; // output: 7
```

The `front()` and `back()` functions can also be used to directly access the first and last elements of a vector. In some cases, writing `back()` can denote the intent of your code clearer than say, `V[V.size() - 1]`.

```
std::vector<int> primes {2, 3, 5, 7, 11};
std::cout << primes.front() << std::endl; // output: 2
std::cout << primes.back() << std::endl;  // output: 11
```

Assigning elements. The [] operator[7] returns a *reference* to an element. So you can assign to it and it will modify the element of the vector.

```
std::vector<int> primes {2, 3, 5, 7, 11};
primes[2] = 42; // primes: {2, 3, 42, 7, 11}
```

The `front()` and `back()` functions *also* return references to the first and last elements respectively. So you can assign to them.

```
std::vector<int> primes {2, 3, 5, 7, 11};
primes.front() = 0;
primes.back() = -1;
// primes: {0, 3, 5, 7, -1}
```

Appending. The `push_back()` function adds an element to the end of the vector. The `pop_back()` function removes an element from the end of a vector.

```
std::vector<int> primes {2, 3, 5, 7, 11};
primes.push_back(12); // primes: {2, 3, 5, 7, 11, 12}
primes.pop_back();    // primes: {2, 3, 5, 7, 11}
primes.push_back(13); // primes: {2, 3, 5, 7, 11, 13}
```

[6] There's also an `at()` function, which does error checking, and has a `const` overload. But you probably won't need that for competitive programming. You shouldn't be writing code which results in an out-of-bounds vector access. And you'll definitely not be specifying functions as `const` in competitive programming.

[7] The `at()` function also returns a reference, so you can write `v.at(2) = 3;` for example.

Resetting contents. The `clear()` function removes all elements from a vector.

```
std::vector<int> V {2, 1, 3, 4, 0};
V.clear(); // V is now empty
```

The `assign()` function replaces the contents of a vector with copies of a given value.

```
std::vector<int> V {2, 1, 3, 4, 0};
// set V to 10 elements, each -1.
V.assign(10, -1);
// V: {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}
```

Unlike C-style arrays, you can assign a braced list to a vector using the = operator and it will replace the contents, even if its not during initialization of the vector.

```
std::vector<int> V {2, 1, 3, 4, 0};
V = {1,2,3,4,5,6};
```

Using <algorithm>. Similar to C-style arrays, we can use the `std::begin()` and `std::end()` functions from the <iterator> header.

```
std::vector<int> V {2, 1, 3, 4, 0};
std::sort(std::begin(V), std::end(V));
// V: {0, 1, 2, 3, 4}
```

However, a more natural syntax for C++ containers like `std::vector` is to use the `begin()` and `end()` functions of the vector itself.

```
std::vector<int> V {2, 1, 3, 4, 0};
std::sort(V.begin(), V.end());
// V: {0, 1, 2, 3, 4}
```

Comparing two vectors. The == and != operators can be used to check if two vectors are equal (==) or not equal (!=). Two vectors are equal if they have the same number of elements and their corresponding elements are equal.

```
std::vector<int> a{1, 2, 3};
std::vector<int> b{1, 2, 3};
std::vector<int> c{7, 8, 9, 10};
std::vector<int> d{7, 9, 8, 10};

std::cout << (a == b) << std::endl; // output: 1 (true)
std::cout << (a != c) << std::endl; // output: 1 (true)
std::cout << (c == d) << std::endl; // output: 0 (false)
```

1.3.3 Using `std::string`

Those of you who already know C++ might be wondering why I'm only mentioning strings now. Well, `std::string` is basically an optimized and specialized[8] version of `std::vector<char>`, meant to represent and work with character arrays.

`std::string` is defined in the `<string>` header (which is often already included by `<iostream>`, but is not guaranteed to be included by `<iostream>`).

In fact, *all of the functions described in the previous section also work with strings.* This is unlike some other languages where strings are "immutable"[9]. Here's another list of common functions. Full details are of course available in the documentation.

Initialization and size.

```
// initializing an empty string
std::string s1;
// initializing a string
std::string s2 {"hello"};
// initializing a string using a list of characters
// yes you can do this i guess but why not just assign to "abc"
std::string s3 {'a', 'b', 'c'};
// strings can be printed out directly without a loop
std::cout << s1 << std::endl;
std::cout << s2 << std::endl;
std::cout << s3 << std::endl;
std::cout << s1.empty() << std::endl;   // output: 1 (true)
std::cout << s2.size() << std::endl;    // output: 5
```

Accessing and assigning elements.

```
std::string s {"boat"};
std::cout << s[1] << std::endl; // output: o
std::cout << s.front() << std::endl; // output: b
std::cout << s.back() << std::endl;  // output: t
s[2] = 'o';
std::cout << s << std::endl;         // output: boot
s.back() = 'r';
std::cout << s << std::endl;         // output: boor
s.front() = 'm';
std::cout << s << std::endl;         // output: moor
```

Appending. For strings, the += operator can be used for appends.

```
std::string s {"rail"};
s.pop_back();     // s: "rai"
s.push_back('n'); // s: "rain"
// can use += to append characters
s += 's';         // s: "rains"
// can use += to append strings
s += "torm"       // s: "rainstorm"
```

[8] `std::string` is not a subclass of `std::vector<char>`. It just functions similarly, storing a contiguous character array (as of C++11), and has a similar interface.

[9] Technically, string literals are immutable in C++, but string variables are not immutable

Concatenation. Two strings can be concatenated (chained) using the `+` operator. Whenever possible, prefer using `s += t` instead of `s = s + t`, which is slower.

```
std::string s {"know"};
std::string t {"ledge"};
std::cout << s + t << std::endl; // output: knowledge
```

Substrings. The substring function in C++ works differently from languages like Java and Python! The `substr(pos, count)` function returns a `std::string` which is the substring starting at `pos`, and containing `count` characters.

```
std::string s {"hello world"};
std::cout << s.substr(6, 3) << std::endl; // output: wor
std::cout << s.substr(0, 2) << std::endl; // output: he
```

Omitting the second argument takes the suffix starting at index `pos`.

```
std::string s {"hello world"};
std::cout << s.substr(6) << std::endl; // output: world
std::cout << s.substr(8) << std::endl; // output: rld
```

Using <algorithm>. Similar to `std::vector`, use `begin()` and `end()` as arguments to functions in `<algorithm>`.

```
std::string s {"binarysearchtree"};
std::sort(s.begin(), s.end());
std::cout << s << std::endl;        // output: aabceeehinrrrsty

std::string t {"tACoCaT"};
std::reverse(t.begin(), t.end());
std::cout << t << std::endl;        // output: TaCoCAt
```

Finding characters and substrings. The `find()` function returns the index of the first occurrence of the given substring or character. The `rfind()` function returns the index of the last occurrence of the given substring or character. For both functions, if the desired substring or character is not found, the special value `string::npos` is returned.

```
std::string s {"This is a string!"};
// technically unsigned int, but just using int is ok for comp prog
int i = s.find("is");
if (i != string::npos)
{
    std::cout << i << std::endl;
}
else
{
    std::cout << "not found" << std::endl;
}
```

Type conversion.[10] Use `std::stoi` (string to int) to convert a string to an `int`, `std::stoll` to convert a string to a `long long`, and `std::to_string` to convert a numeric type to a string. These are all part of the <`string`> header.

```
int a {std::stoi("125")};
long long b {std::stoll("1250")};
std::string s {std::to_string(232)};
std::string t {std::to_string(232LL)};
```

[10] The functions in the <`cstdlib`>, which includes `atoi`, `strtoll`, and similar functions, are meant for C-style strings, which you shouldn't be using anyway.

2 Control structures

2.1 The basics you should know

Programming languages generally share common control structures, such as conditionals and loops. Read up on the following so you are familiar with the basic syntax for control structures in C++.

- learncpp.com - 8.1 - Control flow introduction
- learncpp.com - 8.2 - If statements and blocks
- learncpp.com - 8.3 - Common if statement problems
- learncpp.com - 8.5 - Switch statement basics
- learncpp.com - 8.6 - Switch fallthrough and scoping
- learncpp.com - 8.8 - Introduction to loops and while statements
- learncpp.com - 8.9 - Do while statements
- learncpp.com - 8.10 - For statements
- learncpp.com - 8.11 - Break and continue
- learncpp.com - 16.8 - Range-based for loops (for-each)

2.2 Some common patterns

Here, we'll show some examples of very common competitive programming tasks. These are basically meant for you to adapt into your own programming style.

2.2.1 Reading a list of n integers

A range for loop works with both vectors and C-style arrays.

```cpp
#include <iostream>
#include <vector>
int main()
{
    int n; std::cin >> n;
    std::vector<int> V(n);
    for (auto& x : V) std::cin >> x;
}
```

```cpp
#include <iostream>
int main()
{
    int n; std::cin >> n;
    int A[n] {};
    for (auto& x : A) std::cin >> x;
}
```

Remarks:

- We use `auto&` instead of just `auto` so that we obtain a *reference* to each vector/array element. That way, when we modify `x` (using `std::cin`), we also modify the respective vector/array element.
- Using a C-style array like this is less common, because of the limits on how big of an array you can declare on the stack.

Another common style is to declare a global array which is large enough to handle all input sizes. This may make it easier for you to have an idea of how much memory your program is using.

```cpp
#include <iostream>
int A[200000];
int main()
{
    int n; std::cin >> n;
    for (int i = 0; i < n; i++) std::cin >> A[i];
}
```

2.2.2 Reading an $R \times C$ grid

An $R \times C$ grid has R rows and C columns. You can think of a grid as a "list of rows".

```cpp
#include <iostream>
int main()
{
    // create an empty grid
    int R, C; std::cin >> R >> C;
    std::vector<std::string> grid (R);       // there are R rows
    for (auto& row : grid) row.assign(C, 0); // each row has C columns

    // read into the grid
    for (auto& row : grid)
    {
        for (auto& el : grid)
        {
            std::cin >> el;
        }
    }

    // alternate method: can also manually index into the grid
    for (int i = 0; i < R; i++)
    {
        for (int j = 0; j < C; j++)
        {
            std::cin >> grid[i][j];
        }
    }
}
```

If you are dealing with a grid of characters, you can represent each row with a string instead.

```cpp
#include <iostream>
#include <string>
#include <vector>

int main()
{
    // create an empty grid
    int R, C; std::cin >> R >> C;
    std::vector<std::string> grid (R);
    for (auto& row : grid) std::cin >> row;
}
```