C++ Fundamentals for Competitive Programming

CPPoliTO

Why C++?

Why C++?

C++ is commonly favored in competitive programming due to its speed and efficiency compared to Python, making it a preferred choice for time-sensitive tasks.

C++ is 100 times faster than Python because it is statically typed, which leads to a faster compilation of code.

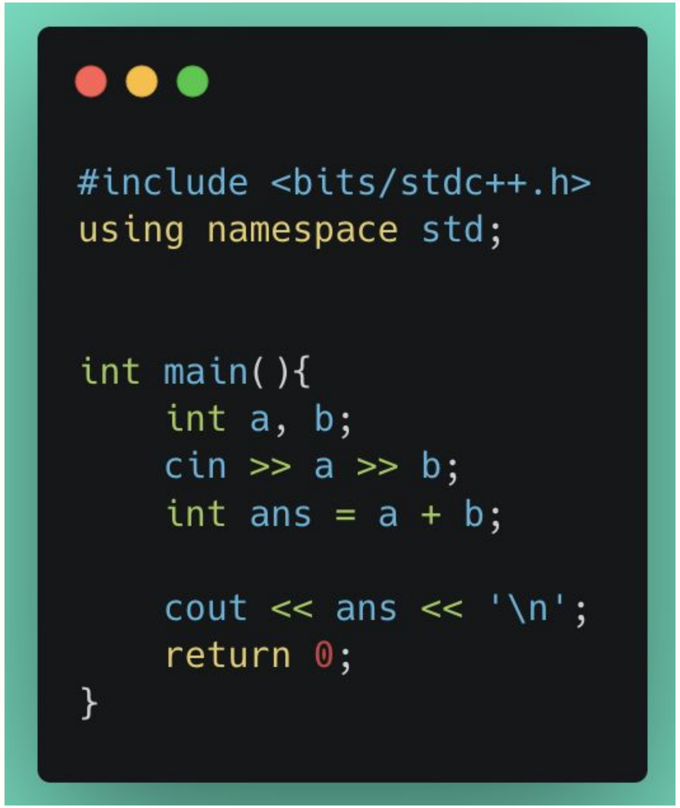
- **C++ is a fast and efficient language.**
- **C++ allows dynamic memory allocation.**
- **Unlike C, C++ is an object-oriented language.**
- **C++ provides a rich library of pre-implemented functions, known as the Standard Template Library (STL)**
- **C++ is a portable language: you can use the same piece of code in different environments.**

Basics of C++

Basics of C++

Only 5 lines of template to start coding with C++.

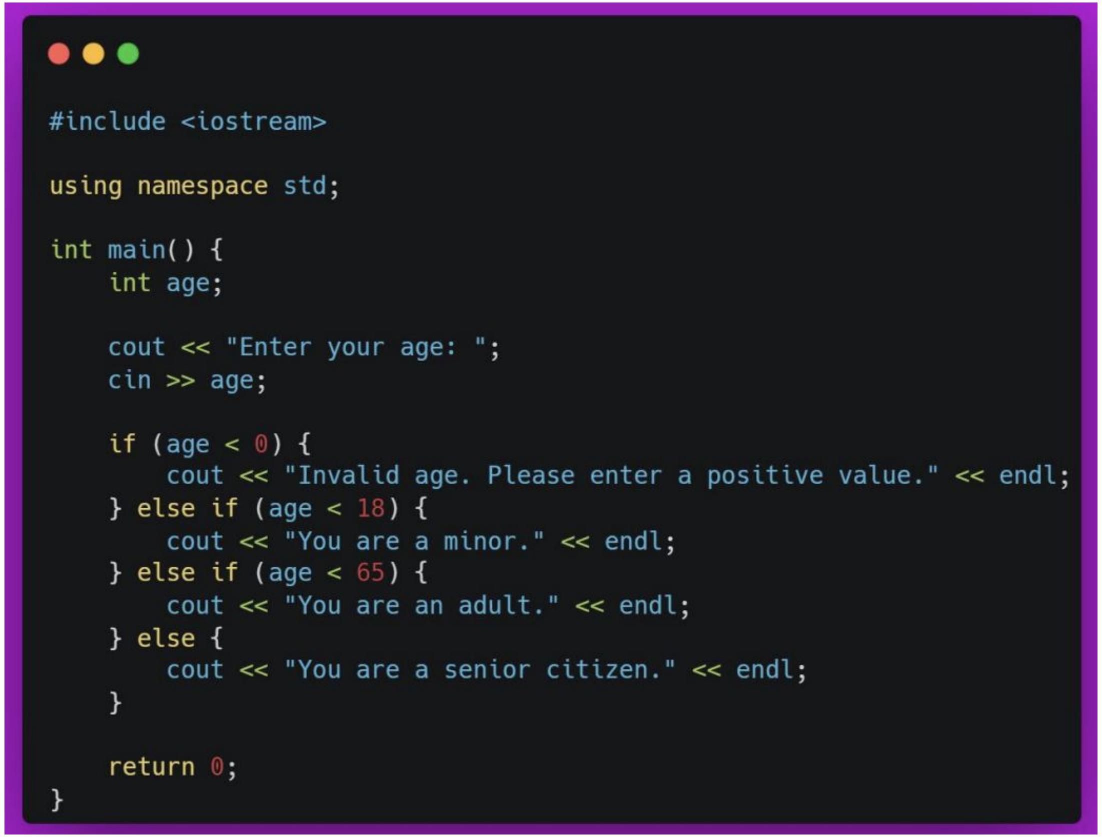
Don’t forget semicolon (‘;’)



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


int main(){
    int a, b;
    cin >> a >> b;
    int ans = a + b;

    cout << ans << '\n';
    return 0;
}
```

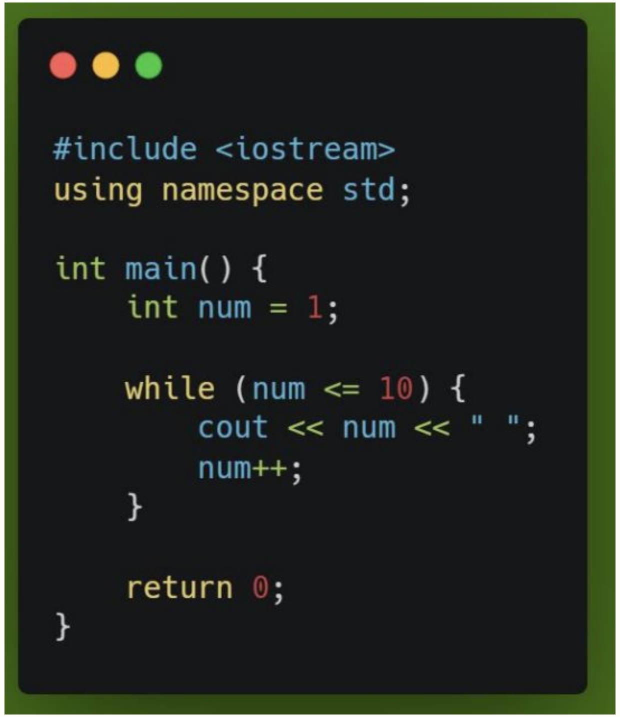
Basic Statements in C++

if & else



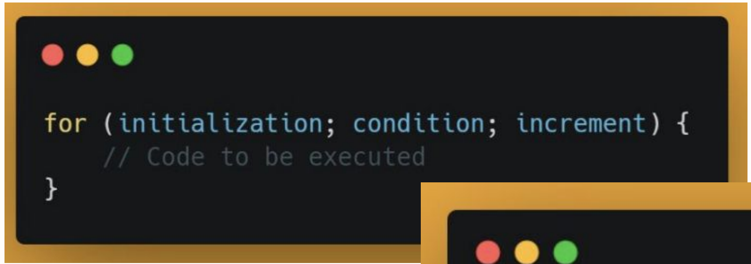
```
#include <iostream>

using namespace std;

int main() {
    int age;

    cout << "Enter your age: ";
    cin >> age;

    if (age < 0) {
        cout << "Invalid age. Please enter a positive value." << endl;
    } else if (age < 18) {
        cout << "You are a minor." << endl;
    } else if (age < 65) {
        cout << "You are an adult." << endl;
    } else {
        cout << "You are a senior citizen." << endl;
    }

    return 0;
}
```

Loops in C++



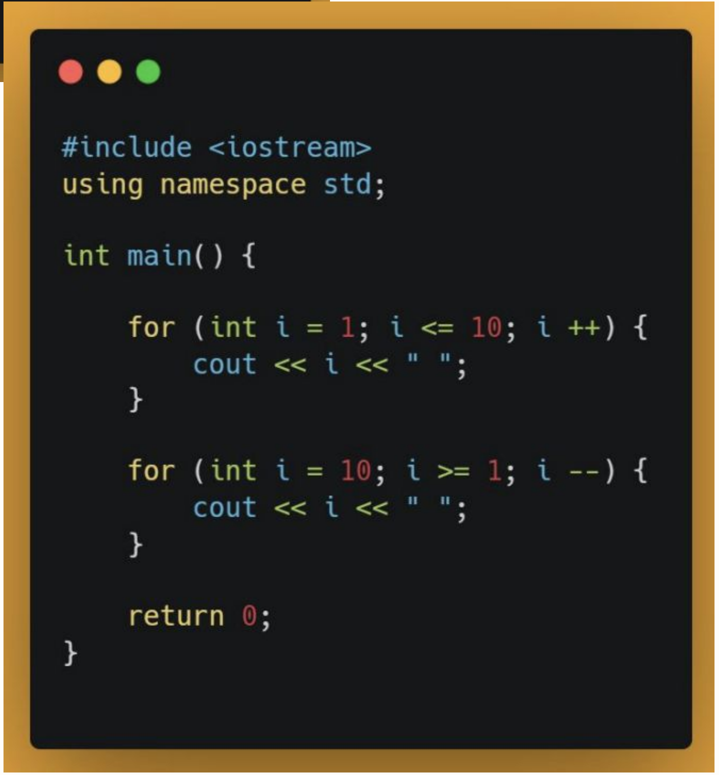
```cpp
#include <iostream>
using namespace std;

int main() {
    int num = 1;

    while (num <= 10) {
        cout << num << " ";
        num++;
    }

    return 0;
}
```

While loop



```cpp
for (initialization; condition; increment) {
    // Code to be executed
}
```

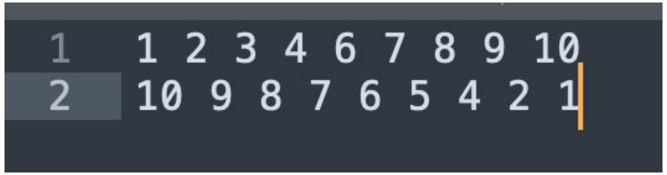
For loop



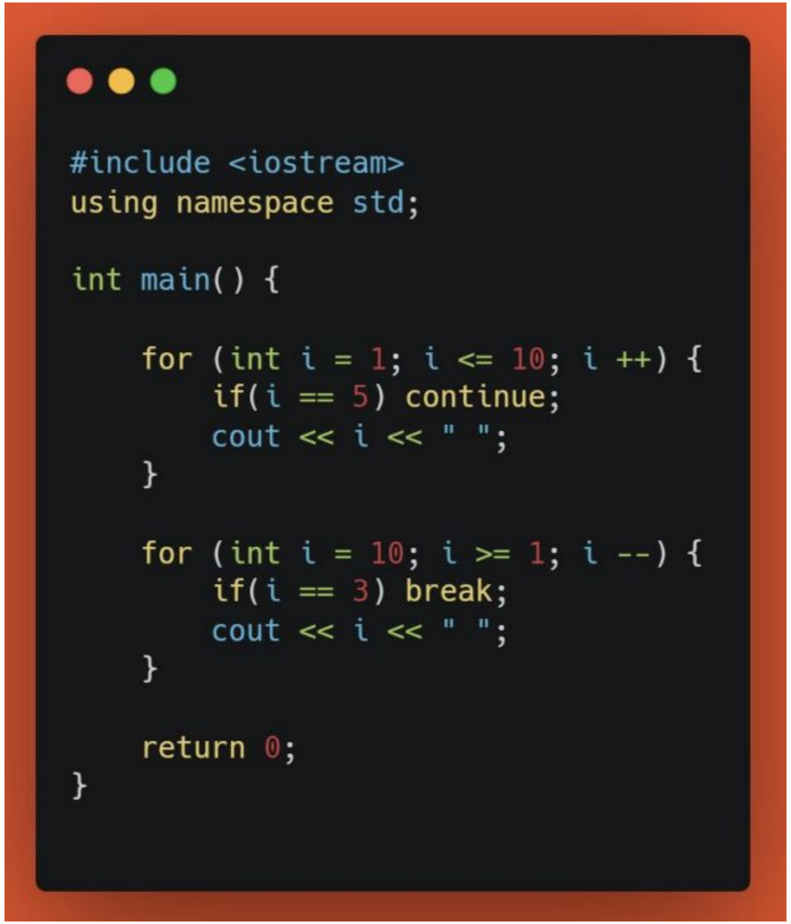
```cpp
#include <iostream>
using namespace std;

int main() {

    for (int i = 1; i <= 10; i ++) {
        cout << i << " ";
    }

    for (int i = 10; i >= 1; i --) {
        cout << i << " ";
    }

    return 0;
}
```

Loops in C++

continue & break in loops



```
1 2 3 4 6 7 8 9 10
10 9 8 7 6 5 4 2 1
```

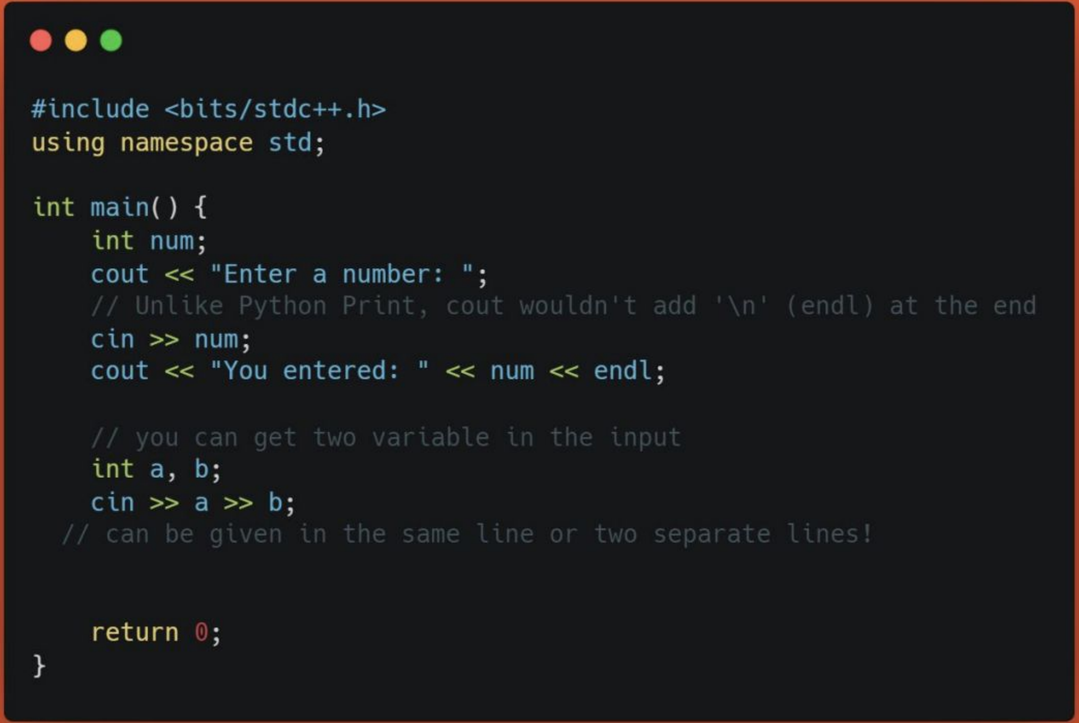


```cpp
#include <iostream>
using namespace std;

int main() {

    for (int i = 1; i <= 10; i ++) {
        if(i == 5) continue;
        cout << i << " ";
    }

    for (int i = 10; i >= 1; i --) {
        if(i == 3) break;
        cout << i << " ";
    }

    return 0;
}
```

Input/Output (IO) in C++

cin & cout

```
cin >> inp;

cout << “Hello!” << endl;
```

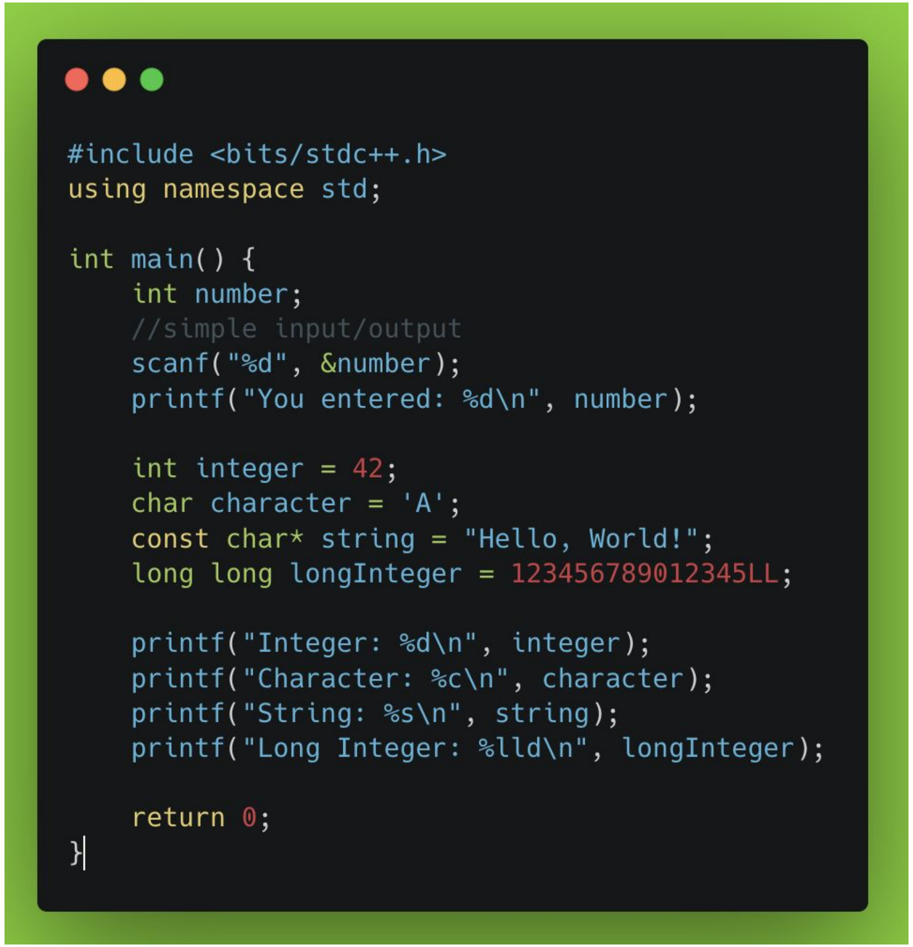


```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int num;
    cout << "Enter a number: ";
    // Unlike Python Print, cout wouldn't add '\n' (endl) at the end
    cin >> num;
    cout << "You entered: " << num << endl;

    // you can get two variable in the input
    int a, b;
    cin >> a >> b;
  // can be given in the same line or two separate lines!


    return 0;
}
```

Input/Output (IO)

scanf & printf

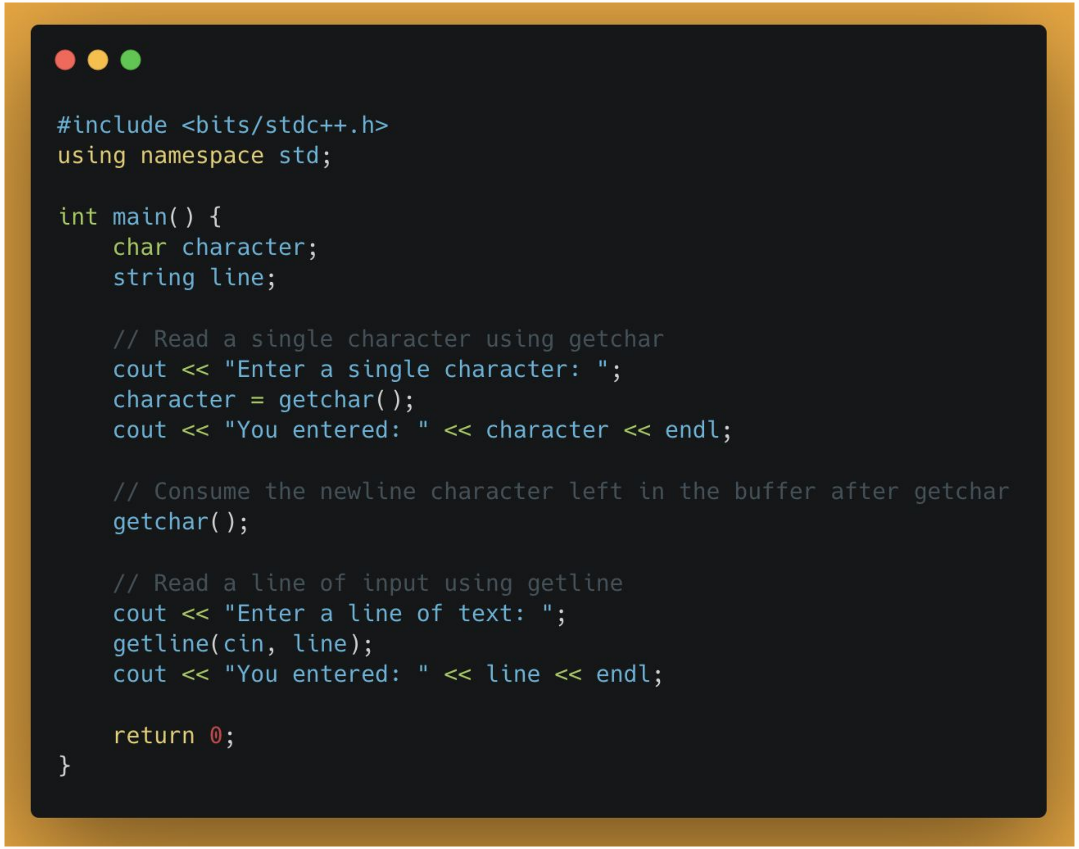
```
'%d' : integer.
'%u' : Unsigned integer.
'%lld' : Long long integer.
'%c' : Single character.
'%s' : Null-terminated string.
'%f' : Floating-point number
'%p' : Pointer address.
```



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

int main() {
    int number;
    //simple input/output
    scanf("%d", &number);
    printf("You entered: %d\n", number);

    int integer = 42;
    char character = 'A';
    const char* string = "Hello, World!";
    long long longInteger = 123456789012345LL;

    printf("Integer: %d\n", integer);
    printf("Character: %c\n", character);
    printf("String: %s\n", string);
    printf("Long Integer: %lld\n", longInteger);

    return 0;
}
```

Input/Output (IO)

getchar & getline



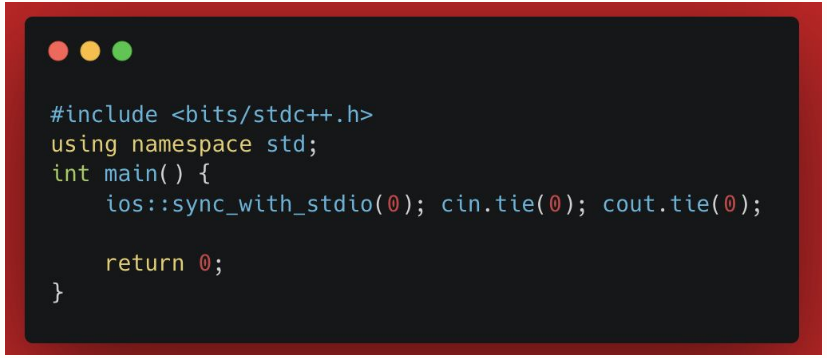
```
#include <bits/stdc++.h>
using namespace std;

int main() {
    char character;
    string line;

    // Read a single character using getchar
    cout << "Enter a single character: ";
    character = getchar();
    cout << "You entered: " << character << endl;

    // Consume the newline character left in the buffer after getchar
    getchar();

    // Read a line of input using getline
    cout << "Enter a line of text: ";
    getline(cin, line);
    cout << "You entered: " << line << endl;

    return 0;
}
```

Fast Input/Output (FastIO)

By default, C++ iostream operations (cin, cout, etc.) are synchronized with the C standard I/O operations (scanf, printf, etc.).
This means that C++ input/output operations are buffered together with C standard I/O operations, which can lead to slower input/output performance.

Setting `ios::sync_with_stdio(0);` disables the synchronization between C++ iostream and C standard I/O, which can improve input/output performance.

Setting `cout.tie(0);` and `cin.tie(0);` unties *cout* from *cin*, meaning that output operations won't automatically flush the input buffer. This can improve output performance by avoiding unnecessary input buffer flushes.

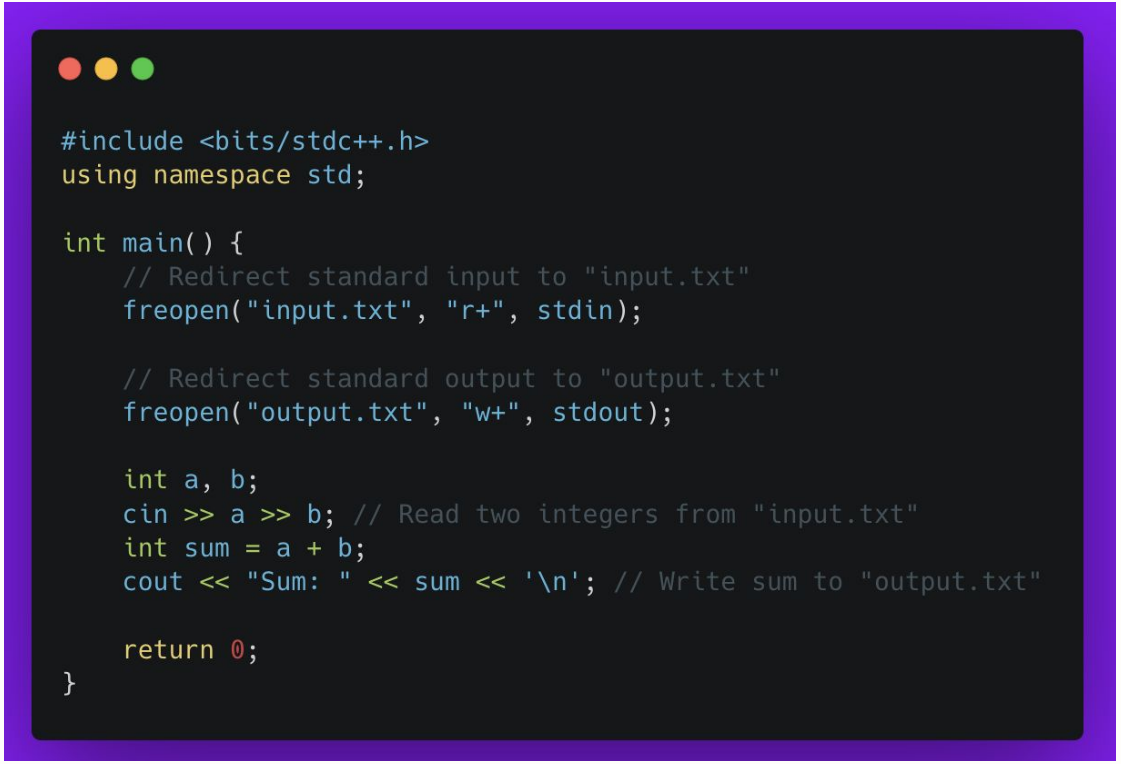


```
#include <bits/stdc++.h>
using namespace std;
int main() {
    ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);

    return 0;
}
```

Reading & Writing in File in C++ IO

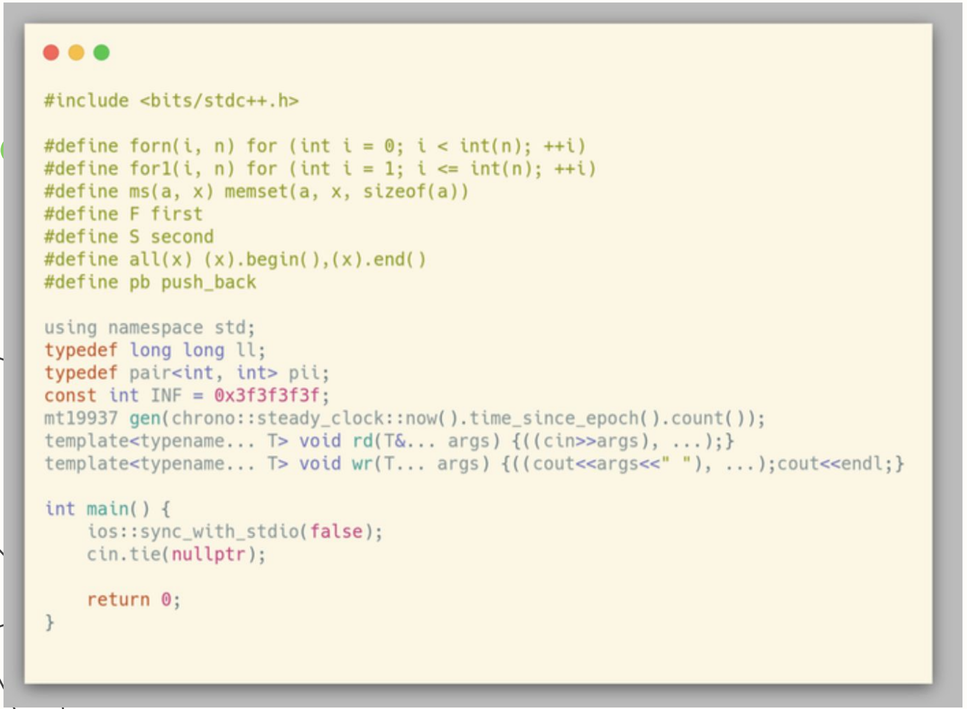
```
freopen(<file_name>,
     "r+" / "w+",
     stdin / stdout);
```

This code snippet redirects the standard input (stdin) to read from the file "*input.txt*" and the standard output (stdout) to write to the file *"output.txt"*.

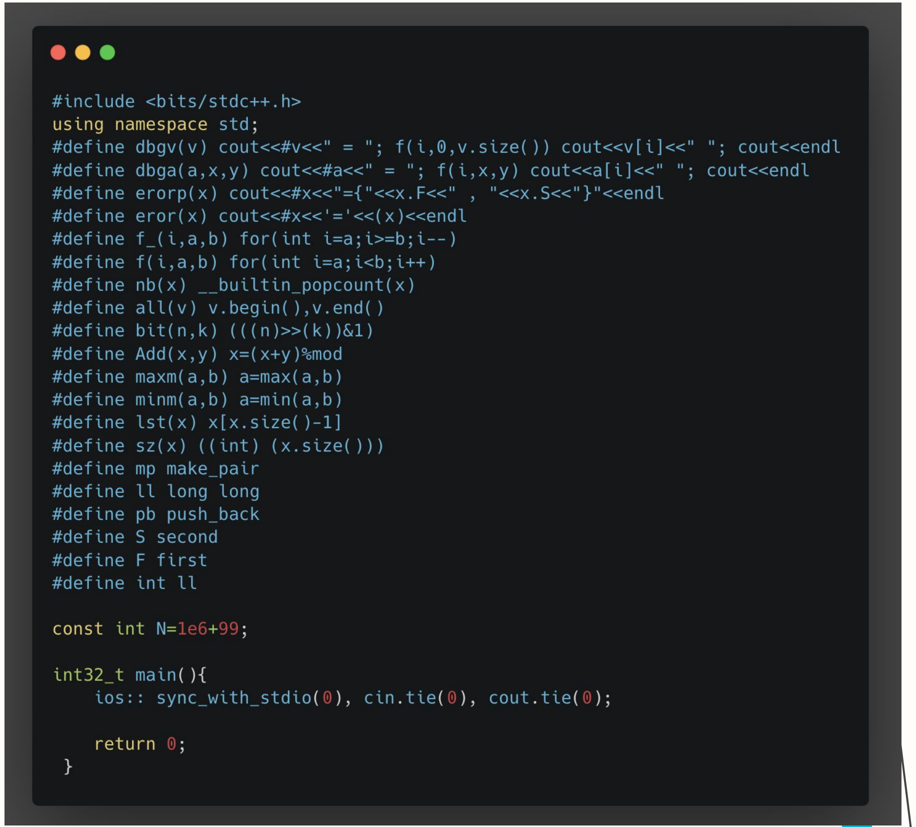


```
#include <bits/stdc++.h>
using namespace std;

int main() {
    // Redirect standard input to "input.txt"
    freopen("input.txt", "r+", stdin);

    // Redirect standard output to "output.txt"
    freopen("output.txt", "w+", stdout);

    int a, b;
    cin >> a >> b; // Read two integers from "input.txt"
    int sum = a + b;
    cout << "Sum: " << sum << '\n'; // Write sum to "output.txt"

    return 0;
}
```

Make your own template!



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

#define forn(i, n) for (int i = 0; i < int(n); ++i)
#define for1(i, n) for (int i = 1; i <= int(n); ++i)
#define ms(a, x) memset(a, x, sizeof(a))
#define F first
#define S second
#define all(x) (x).begin(),(x).end()
#define pb push_back

using namespace std;
typedef long long ll;
typedef pair<int, int> pii;
const int INF = 0x3f3f3f3f;
mt19937 gen(chrono::steady_clock::now().time_since_epoch().count());
template<typename... T> void rd(T&... args) {((cin>>args), ...);}
template<typename... T> void wr(T... args) {((cout<<args<<" "), ...);cout<<endl;}

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

    return 0;
}
```



```cpp
#include <bits/stdc++.h>
using namespace std;
#define dbgv(v) cout<<#v<<" = "; f(i,0,v.size()) cout<<v[i]<<" "; cout<<endl
#define dbga(a,x,y) cout<<#a<<" = "; f(i,x,y) cout<<a[i]<<" "; cout<<endl
#define erorp(x) cout<<#x<<"={"<<x.F<<" , "<<x.S<<"}"<<endl
#define eror(x) cout<<#x<<'='<<(x)<<endl
#define f_(i,a,b) for(int i=a;i>=b;i--)
#define f(i,a,b) for(int i=a;i<b;i++)
#define nb(x) __builtin_popcount(x)
#define all(v) v.begin(),v.end()
#define bit(n,k) (((n)>>(k))&1)
#define Add(x,y) x=(x+y)%mod
#define maxm(a,b) a=max(a,b)
#define minm(a,b) a=min(a,b)
#define lst(x) x[x.size()-1]
#define sz(x) ((int) (x.size()))
#define mp make_pair
#define ll long long
#define pb push_back
#define S second
#define F first
#define int ll

const int N=1e6+99;

int32_t main(){
    ios:: sync_with_stdio(0), cin.tie(0), cout.tie(0);

    return 0;
 }
```

Get ideas from other programmers templates and create the one makes you fastest.
Share your templates! [CodeForcesBlog#77199]

Variables and Data Types

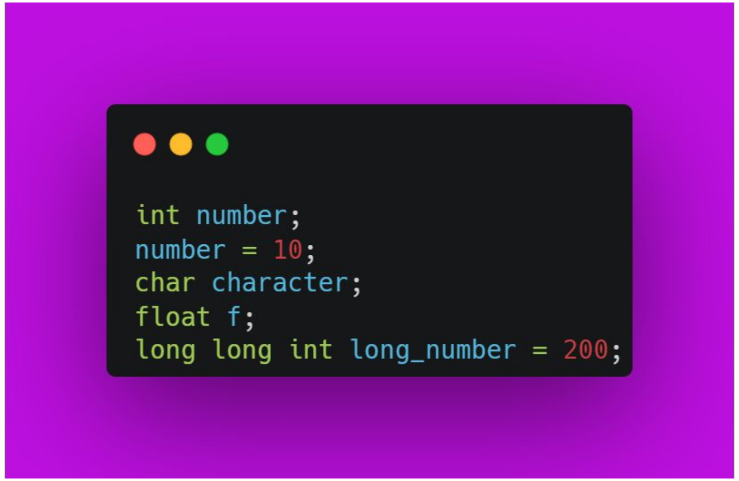
Variable definition and initialization

In C++, we have to **define** variables in order to use them, and we do it using the following expression:

`<data type> <name of the variable>;`

We can initialize a variable together with its definition or after its definition.

In C++, each variable has a data type.



```
int number;
number = 10;
char character;
float f;
long long int long_number = 200;
```

Integers (int)

Integer variables store **whole numbers**. For example, the line of code:

```
int x = 10;
```

defines the integer variable x and assigns to it the value 10.

An `"int"` data type typically uses **4 bytes** (**32 bits**) of memory and can represent integers in the range of *-2,147,483,648* to *2,147,483,647* (i.e., of the order of *-1e9* to *1e9*).

Long long integers (long long int)

Used to store **very big** numbers.
A "long long int" data type typically uses **8 bytes** (**64 bits**) of memory and can represent integers in the range of *-9,223,372,036,854,775,808* to *9,223,372,036,854,775,807* (i.e., of the order of *-1e18* to *1e18*).

There exist also the data types **unsigned int** and **unsigned long long int** which are used to store positive integers.

Floats (float) and doubles (double)

Used to store **floating-point numbers**, which are number with a decimal point. For example, the line of code:

```
double pi = 3.14;
```

defines the variable pi that stores the value *3.14*

A float variable typically uses **4 bytes** of memory, while a double uses **8 bytes** of memory.

In general, a `"double"` can store values in the range of approximately *10^-308* to *10^308*, with a precision of about 15-16 digits.

Long doubles (long double)

The precision of a double can be affected by factors such as the specific values being stored, the operations being performed on those values, and any potential rounding errors!
A long double, on the other hand, typically uses **10** or **12 bytes** of memory (depending on the implementation) and can store values in the range of approximately *10^-4932* to *10^4932*, with a precision of about 19-20 digits.

This provides much greater **precision** and **range** than a double.

Characters (char)

Used to store **single characters**, such as letters or symbols. For example, the line of code:

```
char letter = 'A';
```

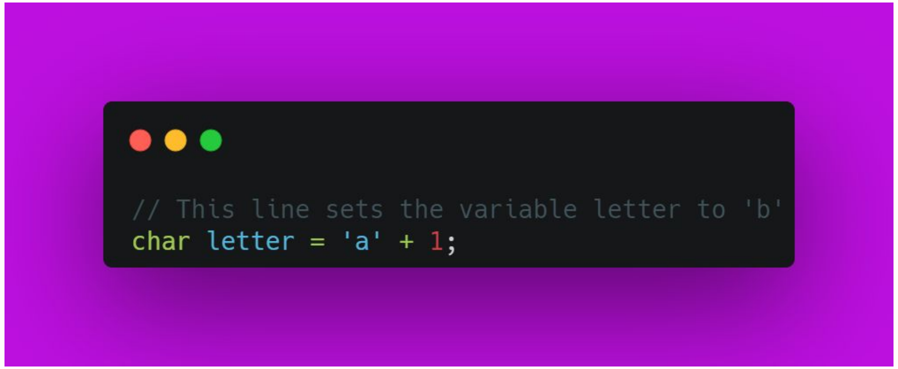
declares a variable called letter that stores the value `'A'`.

Important note:
Characters are indicated with **single quotes** (`‘A’` represents the single character, `“A”` represents a string).

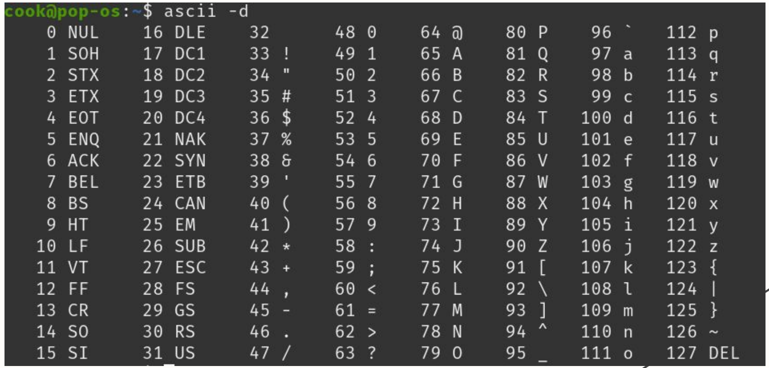
A "char" variable uses **1 byte** (**8 bits**) of memory and can store 256 different values, ranging from 0 to 255.

ASCII encoding of characters

In addition to representing individual characters, `"char”` can also be used to represent integer values using the **ASCII encoding**.
For example, the character `'a'` has an ASCII value of 97 (we can convert
numbers from 0 to 255 to characters and vice versa).



```
// This line sets the variable letter to 'b'
char letter = 'a' + 1;
```



```
cook@pop-os:~$ ascii -d
    0 NUL    16 DLE    32       48 0    64 @    80 P    96 `    112 p
    1 SOH    17 DC1    33 !     49 1    65 A    81 Q    97 a    113 q
    2 STX    18 DC2    34 "     50 2    66 B    82 R    98 b    114 r
    3 ETX    19 DC3    35 #     51 3    67 C    83 S    99 c    115 s
    4 EOT    20 DC4    36 $     52 4    68 D    84 T   100 d    116 t
    5 ENQ    21 NAK    37 %     53 5    69 E    85 U   101 e    117 u
    6 ACK    22 SYN    38 &     54 6    70 F    86 V   102 f    118 v
    7 BEL    23 ETB    39 '     55 7    71 G    87 W   103 g    119 w
    8 BS     24 CAN    40 (     56 8    72 H    88 X   104 h    120 x
    9 HT     25 EM     41 )     57 9    73 I    89 Y   105 i    121 y
   10 LF     26 SUB    42 *     58 :    74 J    90 Z   106 j    122 z
   11 VT     27 ESC    43 +     59 ;    75 K    91 [   107 k    123 {
   12 FF     28 FS     44 ,     60 <    76 L    92 \   108 l    124 |
   13 CR     29 GS     45 -     61 =    77 M    93 ]   109 m    125 }
   14 SO     30 RS     46 .     62 >    78 N    94 ^   110 n    126 ~
   15 SI     31 US     47 /     63 ?    79 O    95 _   111 o    127 DEL
```

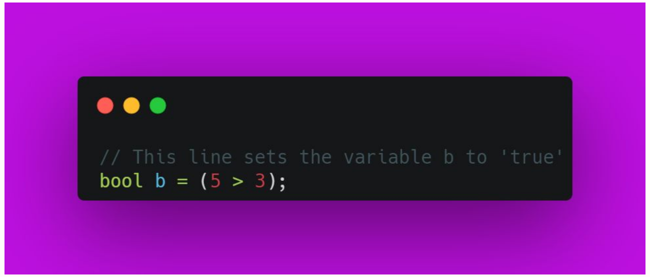
Boolean variables (bool)

Used to store boolean values, which are either *true*(1) or *false*(0). For example, the line of code:

`bool isTrue = true;`

defines a variable called `isTrue` that stores the value *true*.
A "bool" variable uses **1 byte** (**8 bits**, naively 8 bits not only 1!) of memory and can store only two possible values: "*true*" or "*false*".

In C++, "*false*" is represented by the value **0** and "*true*" is represented by **any non-zero value**.



```
// This line sets the variable b to 'true'
bool b = (5 > 3);
```

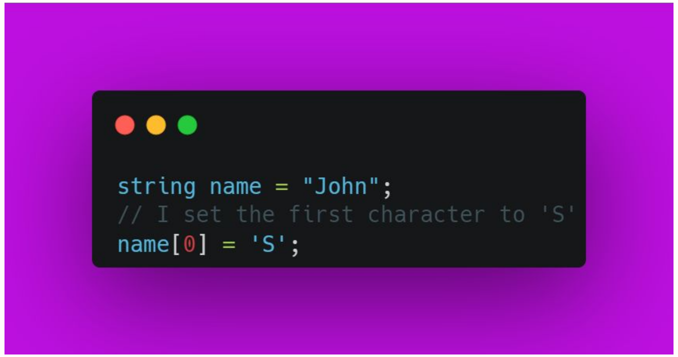
Strings (string)

Used to store a **sequence of characters**, such as a word or sentence. For example, the line of code:

`string name = "John";`

defines a variable called name that stores the value `"John"`.

Unlike python, strings in C++ are **mutable**, which means you can change the *i'th* character of the string!

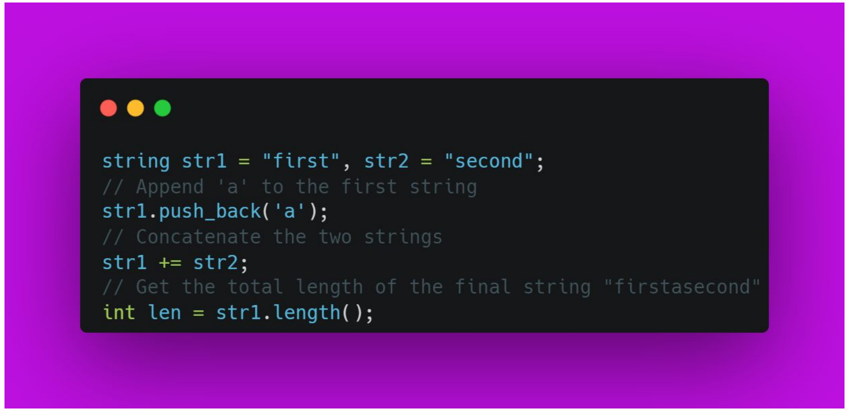


```
string name = "John";
// I set the first character to 'S'
name[0] = 'S';
```

Operations on strings

We can:

- append a character to the string with the function “**push_back()**”;
- concatenate two strings with the **+ operator**;
- retrieve the length of a string with the function “**length()**” (which works in *O(1)*).



```cpp
string str1 = "first", str2 = "second";
// Append 'a' to the first string
str1.push_back('a');
// Concatenate the two strings
str1 += str2;
// Get the total length of the final string "firstasecond"
int len = str1.length();
```

Integer operations

Basic operations

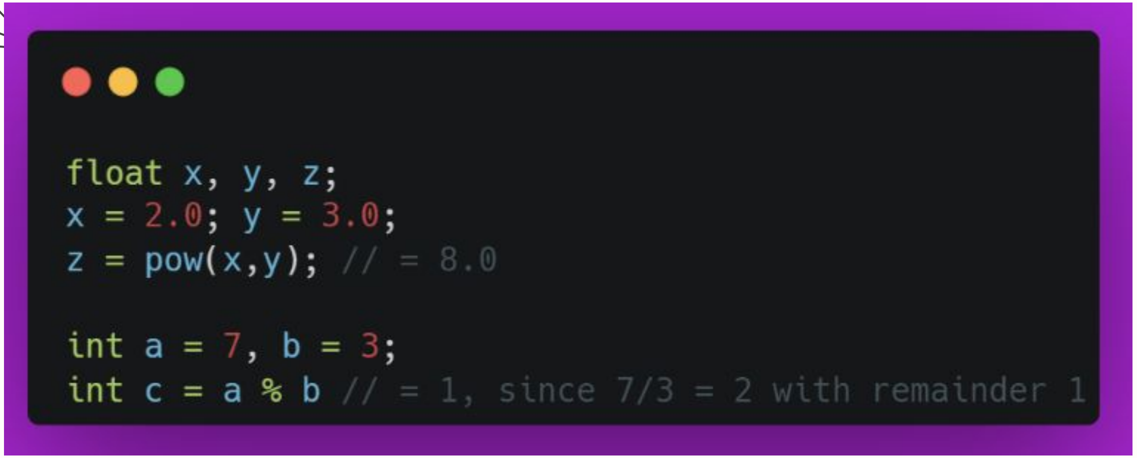
In C++ we can of course perform the basic operations between integers (addition, subtraction, multiplication, division).

When doing division we need to be careful with the `'/'` operator!
if we use it with integers, we will perform an **integer division**. If we want to compute the normal division, we first have to convert our operands to floats!

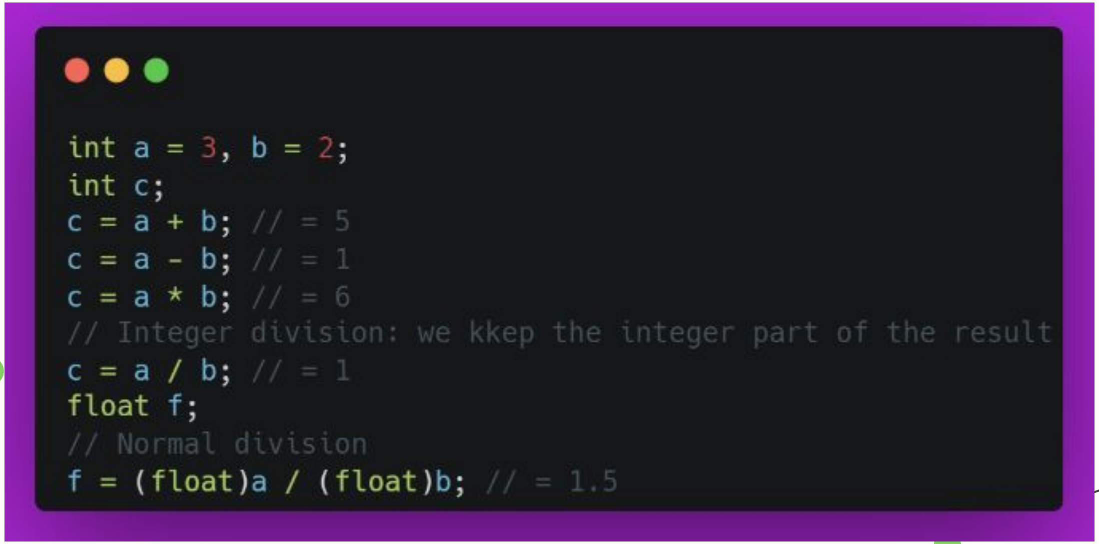
To compute a **power**, we can use the library function `pow(x,y)`, but notice that x, y and the return value of that function will **all** be **floats**.

There is also the **modulus** operator `'%'`, which takes as a result the remainder of a division.

Some examples:



```
float x, y, z;
x = 2.0; y = 3.0;
z = pow(x,y); // = 8.0

int a = 7, b = 3;
int c = a % b // = 1, since 7/3 = 2 with remainder 1
```

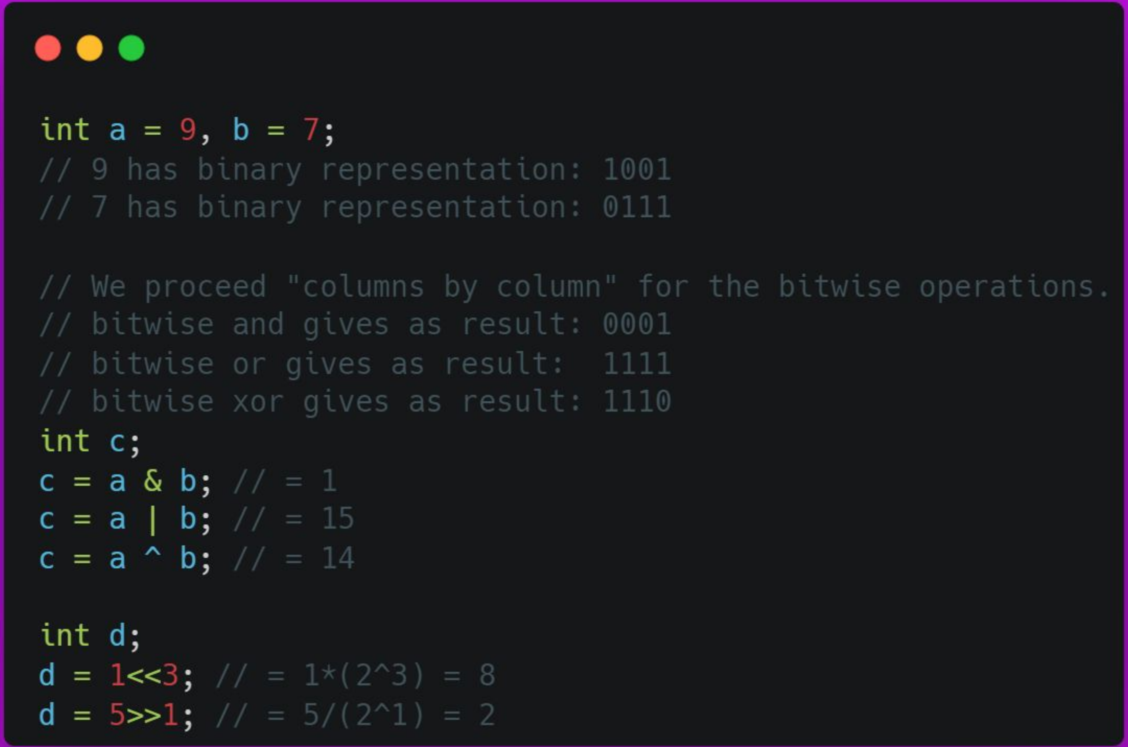


```
int a = 3, b = 2;
int c;
c = a + b; // = 5
c = a - b; // = 1
c = a * b; // = 6
// Integer division: we kkep the integer part of the result
c = a / b; // = 1
float f;
// Normal division
f = (float)a / (float)b; // = 1.5
```

Bitwise operations

Every integer has a binary representation, and we can perform some operations between integers based on those representations:

- `'&'` is the **bitwise AND** operator;
- `'|'` is the **bitwise OR** operator;
- `'^'` is the **bitwise XOR** operator;
- `'<<'` is the **left-shift** and `'>>'` is the **right-shift** operator.
  `x<<y` computes *"x*(2^y)"* and `x>>y` computes *"x/(2^y)"*
  (in this sentence the '^' denotes the **power** operation, not the bitwise xor).



```
int a = 9, b = 7;
// 9 has binary representation: 1001
// 7 has binary representation: 0111

// We proceed "columns by column" for the bitwise operations.
// bitwise and gives as result: 0001
// bitwise or gives as result:  1111
// bitwise xor gives as result: 1110
int c;
c = a & b; // = 1
c = a | b; // = 15
c = a ^ b; // = 14

int d;
d = 1<<3; // = 1*(2^3) = 8
d = 5>>1; // = 5/(2^1) = 2
```

C++ Standard Template Library (STL)

C++ offers a library of various algorithm and data structures which can be easily used. We will go over the basic elements of the main STL algorithms and data structures.

Pairs

In C++, a pair is a container that holds two variables. We can define a pair with the following syntax:

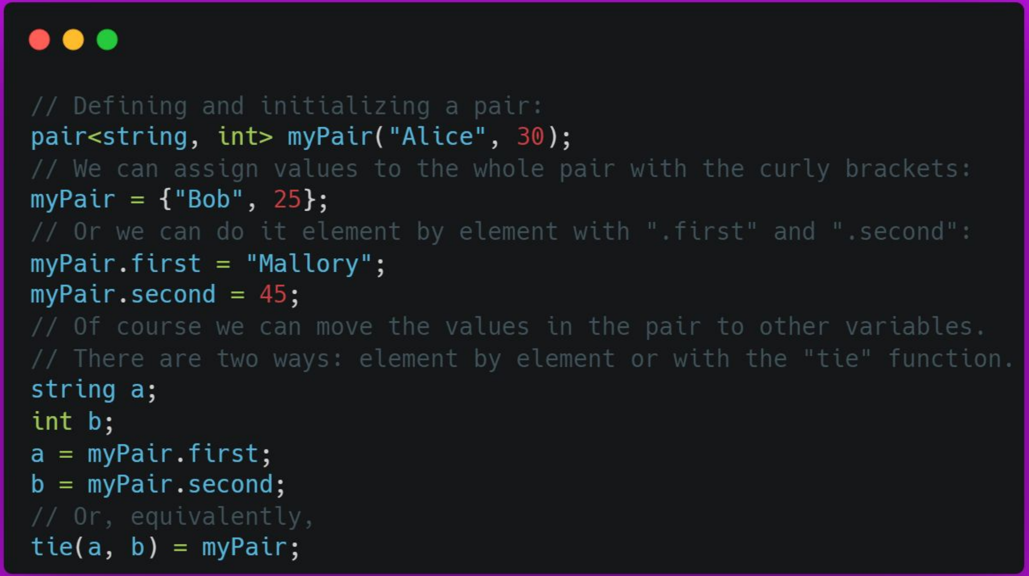
```
pair<”data type 1”, “data type 2”> myPair;
```

With this line, we defined the pair “*myPair*”, which contains a first value of data type “*data type 1*” and a second value of data type “*data type 2*”.

Notice that pairs **do not** contain necessarily two values of the same data type. We can access the first and the second value respectively with the following syntax:

```
firstValue = myPair.first;
secondValue = myPair.second;
```

Here are some examples with pairs:

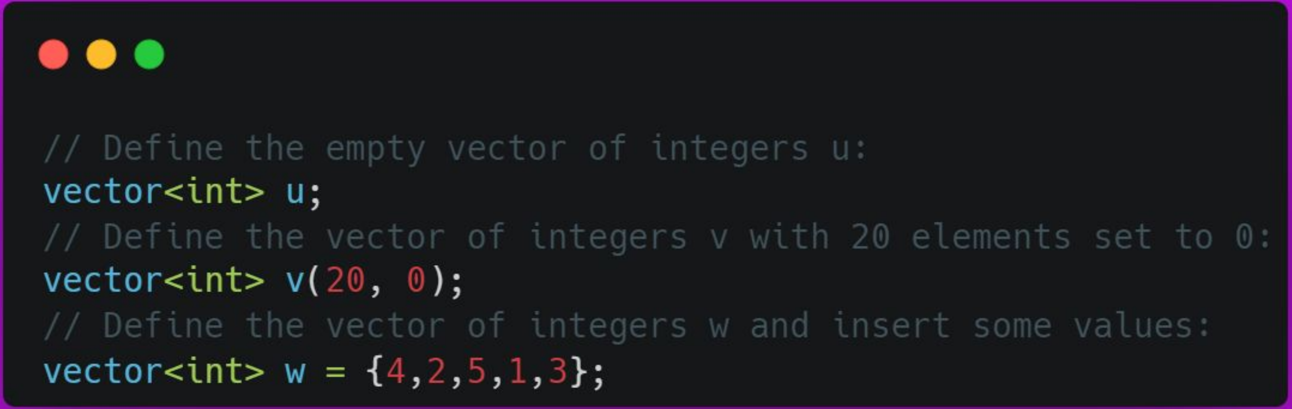


```
// Defining and initializing a pair:
pair<string, int> myPair("Alice", 30);
// We can assign values to the whole pair with the curly brackets:
myPair = {"Bob", 25};
// Or we can do it element by element with ".first" and ".second":
myPair.first = "Mallory";
myPair.second = 45;
// Of course we can move the values in the pair to other variables.
// There are two ways: element by element or with the "tie" function.
string a;
int b;
a = myPair.first;
b = myPair.second;
// Or, equivalently,
tie(a, b) = myPair;
```

Vectors

C++ vectors are dynamic arrays which are resized automatically.
They can be **defined** in the following ways:

- `vector<”data type”> name;`
  creates a new **empty** vector of elements with data type “data type”;
- `vector<”data type”> name(n, value);`
  creates a new vector with “n” elements of data type “data type”. All the elements in the vector are **initialized** with the value “*value*”. Both the “n” and the “value” argument are **not** mandatory;
- `vector<”data type”> name = {el1, el2, el3, ..., eln}`
  creates a vector containing the elements inside the braces (all those elements have data type equal to “data type”).



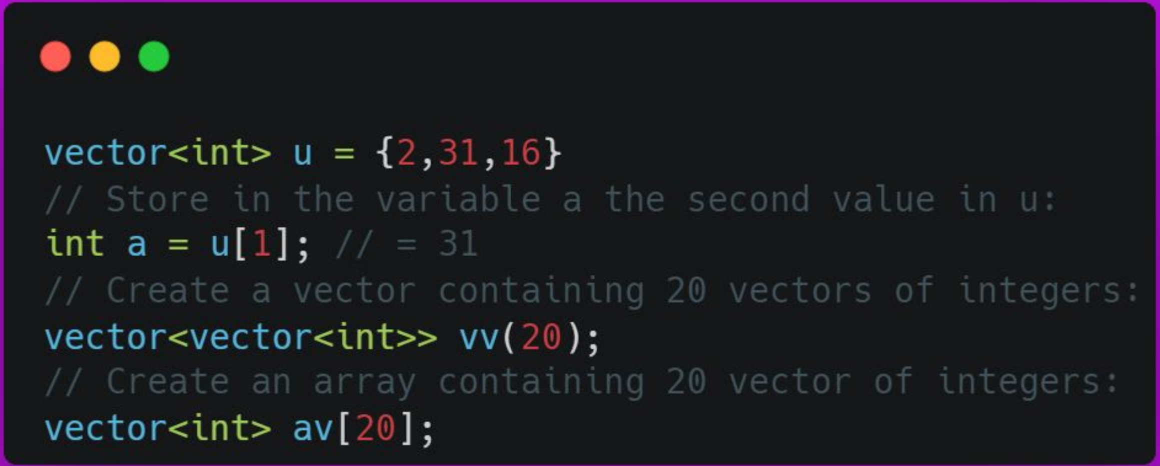
```
// Define the empty vector of integers u:
vector<int> u;
// Define the vector of integers v with 20 elements set to 0:
vector<int> v(20, 0);
// Define the vector of integers w and insert some values:
vector<int> w = {4,2,5,1,3};
```

Vectors are *0-indexed* and it's possible to access each element by its index, like regular C style arrays.

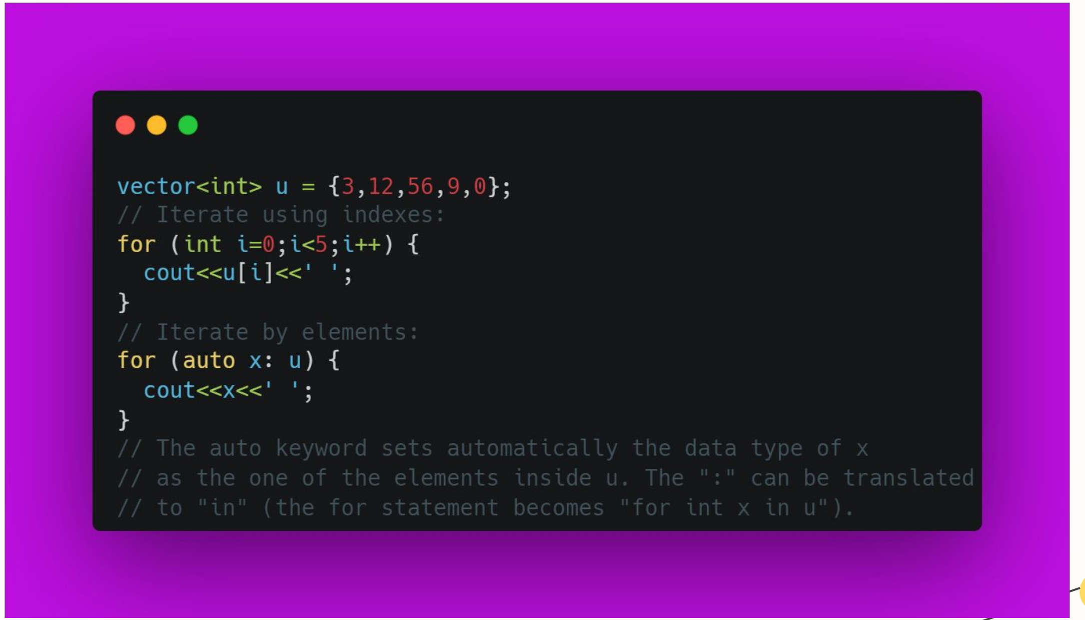
- Pay attention to the fact that, if we don't specify the initial size of the vector during its definition, we will create an empty vector. Thus, if we try to access any element in it (for example, with v[0]), we will get an error.

Vectors and C arrays are interchangeable in some situations, but not in others, because they are **very different** objects: arrays have a fixed size, and all the functions that can be used on vectors that we will see in the next slides cannot be used on arrays.

It is possible however to store C++ data structures in both arrays and vectors. For example, we can create a vector of vectors or an **array of vectors** (but not a vector of arrays).


```cpp
vector<int> u = {2,31,16}
// Store in the variable a the second value in u:
int a = u[1]; // = 31
// Create a vector containing 20 vectors of integers:
vector<vector<int>> vv(20);
// Create an array containing 20 vector of integers:
vector<int> av[20];
```

There are two ways to iterate through a vector. The first one is based on **indexes**, exactly like we do in C for arrays.
The second one iterates on the **elements**.



```
vector<int> u = {3,12,56,9,0};
// Iterate using indexes:
for (int i=0;i<5;i++) {
  cout<<u[i]<<' ';
}
// Iterate by elements:
for (auto x: u) {
  cout<<x<<' ';
}
// The auto keyword sets automatically the data type of x
// as the one of the elements inside u. The ":" can be translated
// to "in" (the for statement becomes "for int x in u").
```

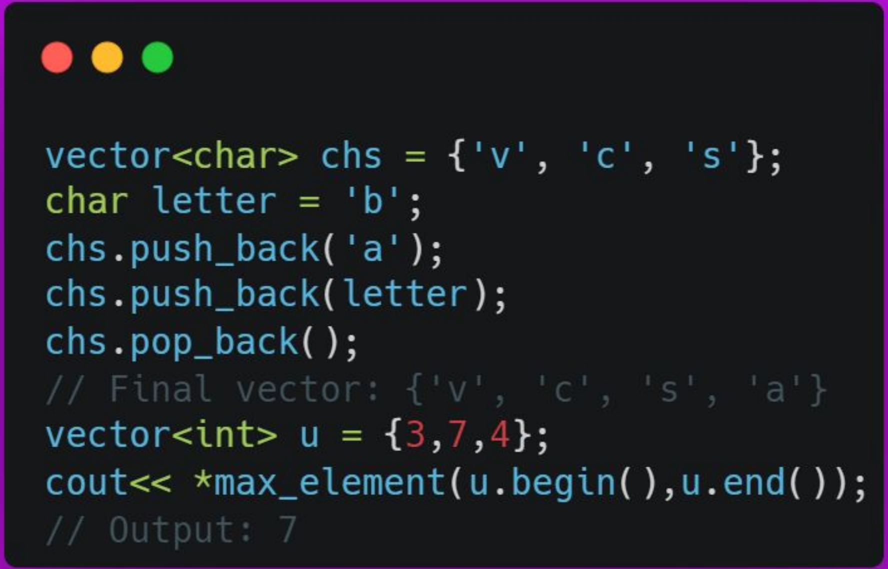
Here are some functions that can be used on vectors:

- `v.push_back(x);` appends the value x (x can be either a “raw” value or an other variable containing the value we want to append) to the vector v. Obviously the data type of x must be the same of the one of the elements in the vector (we’ll see how to add new elements after the initialization).

- `v.pop_back();` removes the last element in the vector v.

- `int a = *max_element(v.begin(), v.end());` puts in a the maximum element in the vector v.

  *Basically, `max_element(v.begin(),v.end())` returns an **iterator** (which is like a pointer) to the maximum element in v, and the * dereferences it.

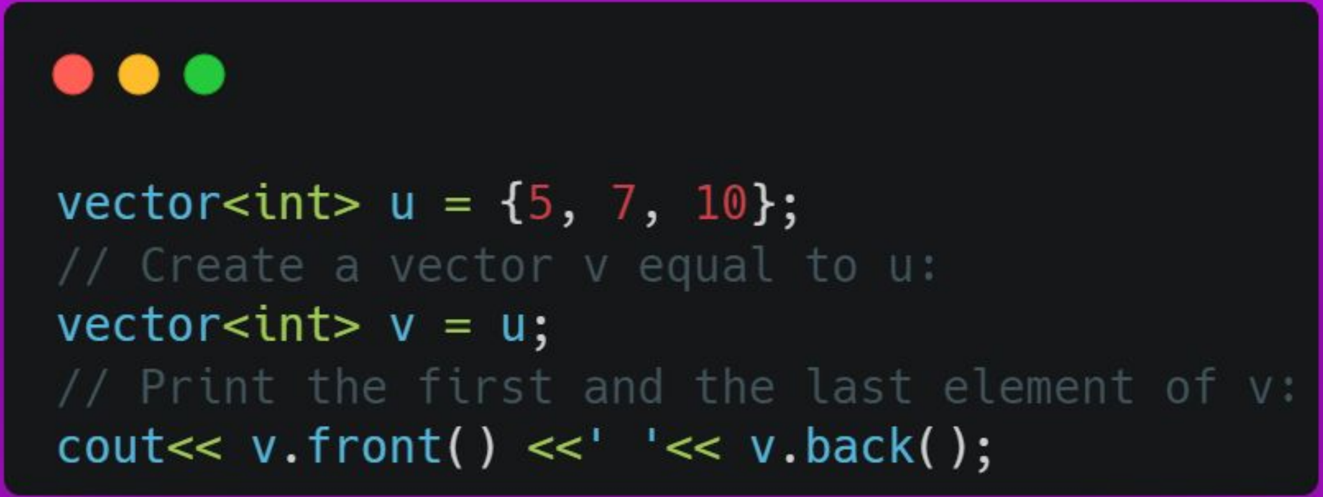


```
vector<char> chs = {'v', 'c', 's'};
char letter = 'b';
chs.push_back('a');
chs.push_back(letter);
chs.pop_back();
// Final vector: {'v', 'c', 's', 'a'}
vector<int> u = {3,7,4};
cout<< *max_element(u.begin(),u.end());
// Output: 7
```

`v.begin()` and `v.end()` are also iterators: the first one points to the **first** element in the vector, the other points to the slot in memory **after** the last element of the vector. So, **never try to access** the element with *v.end()*.

To access the first and the last element of a vector, we can use instead `v.front()` and `v.back()`.

Another thing we can do with vectors is **assigning** to a vector the values of another vector using the '=' operator. This operation is **linear** in the number of elements we want to assign.



```cpp
vector<int> u = {5, 7, 10};
// Create a vector v equal to u:
vector<int> v = u;
// Print the first and the last element of v:
cout<< v.front() <<' '<< v.back();
```

Some more functions:

- `v.clear();` deletes all the elements in v and makes it an empty vector with linear complexity.

- `v.resize(x,y);` changes the size of v to x, assigning to the new elements the value y.

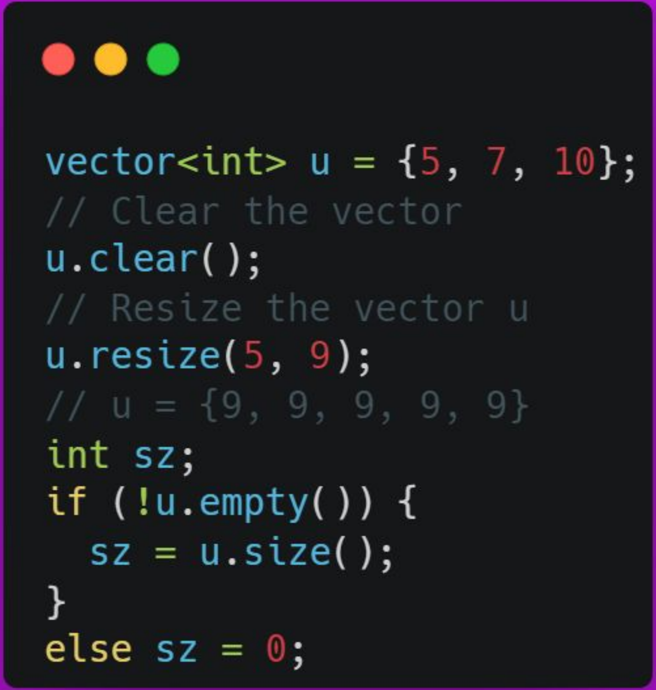
- `v.empty();` checks whether the vector v is empty (returns 1 if it is, 0 otherwise).

- `v.size();` returns the number of elements in v in O(1).

Be careful: the *.size()* function may lead to undefined behaviour if the vector is empty. So, if we want to know the number of elements inside a vector, but that vector can be empty, we first have to check whether the vector is actually empty, and then use the *.size()* function.

Note:

The .clear(), .empty() and . size() functions are also valid for the other data structures that will be presented in the next slides.



```
vector<int> u = {5, 7, 10};
// Clear the vector
u.clear();
// Resize the vector u
u.resize(5, 9);
// u = {9, 9, 9, 9, 9}
int sz;
if (!u.empty()) {
  sz = u.size();
}
else sz = 0;
```

Sorting

Sorting in C

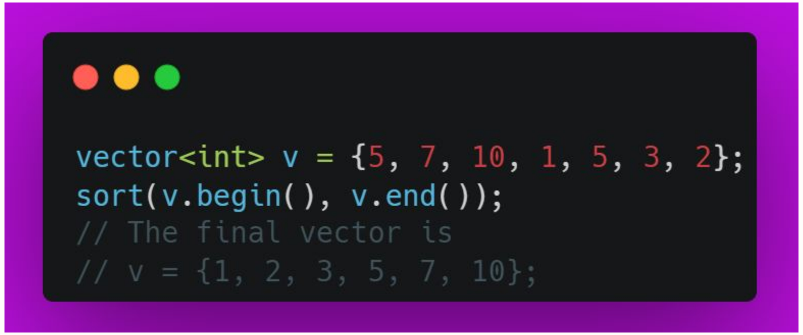
Sorting in C++

C++ like many other languages has a built-in STL function for sorting an array.

- To sort a **vector**, we can use `sort(v.begin(),v.end());` to sort in increasing order, or `sort(v.rbegin(),v.rend());` to sort in reverse order.

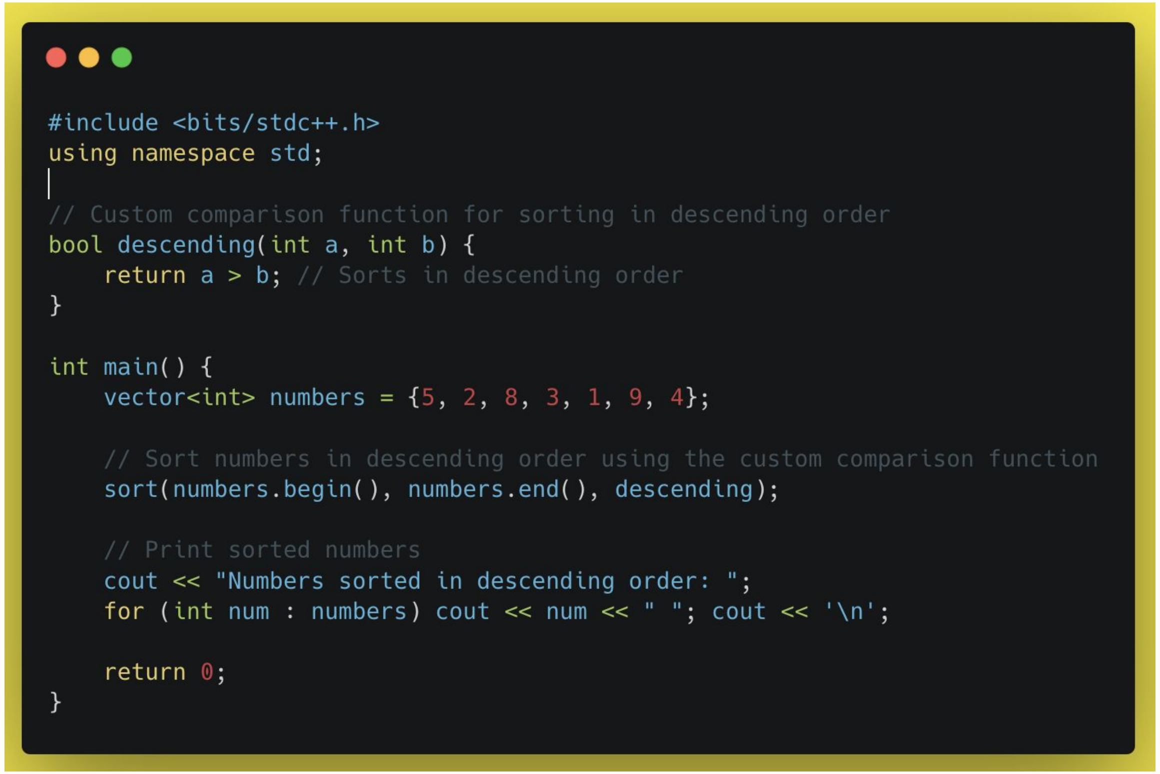
- To sort a C style **array** we can use `sort(a,a+n);` where n is the size of the array.

The sort function has time complexity *O(n*log(n))*, where n is the size of the vector/array.



```
vector<int> v = {5, 7, 10, 1, 5, 3, 2};
sort(v.begin(), v.end());
// The final vector is
// v = {1, 2, 3, 5, 7, 10};
```

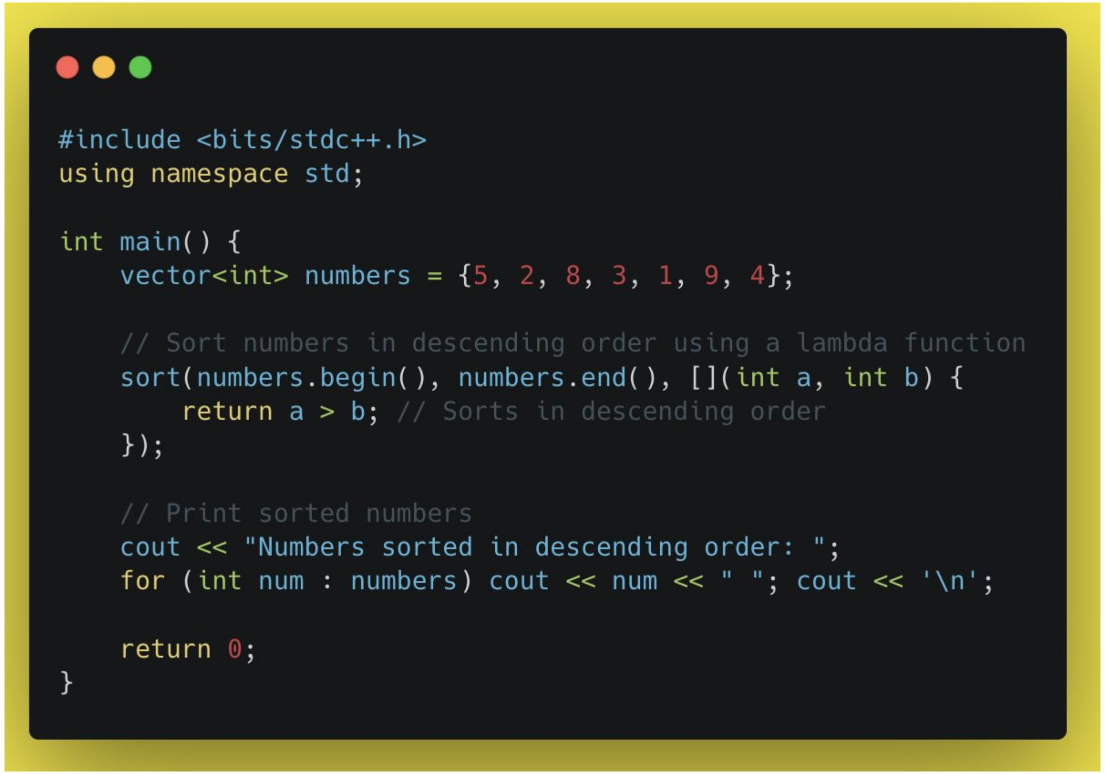
Passing custom comparison function for sorting (CMP)



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

// Custom comparison function for sorting in descending order
bool descending(int a, int b) {
    return a > b; // Sorts in descending order
}

int main() {
    vector<int> numbers = {5, 2, 8, 3, 1, 9, 4};

    // Sort numbers in descending order using the custom comparison function
    sort(numbers.begin(), numbers.end(), descending);

    // Print sorted numbers
    cout << "Numbers sorted in descending order: ";
    for (int num : numbers) cout << num << " "; cout << '\n';

    return 0;
}
```

Passing custom comparison function for sorting (CMP)

Using lambda function, like a pro!



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

int main() {
    vector<int> numbers = {5, 2, 8, 3, 1, 9, 4};

    // Sort numbers in descending order using a lambda function
    sort(numbers.begin(), numbers.end(), [](int a, int b) {
        return a > b; // Sorts in descending order
    });

    // Print sorted numbers
    cout << "Numbers sorted in descending order: ";
    for (int num : numbers) cout << num << " "; cout << '\n';

    return 0;
}
```

When we have a vector and we know that we don’t need to add anymore elements to it, we have seen that we can easily sort it.
But what if we want to keep a data structure that **remains sorted** whenever we insert or remove an element?
Luckily, the STL provides for the **set** data structure.

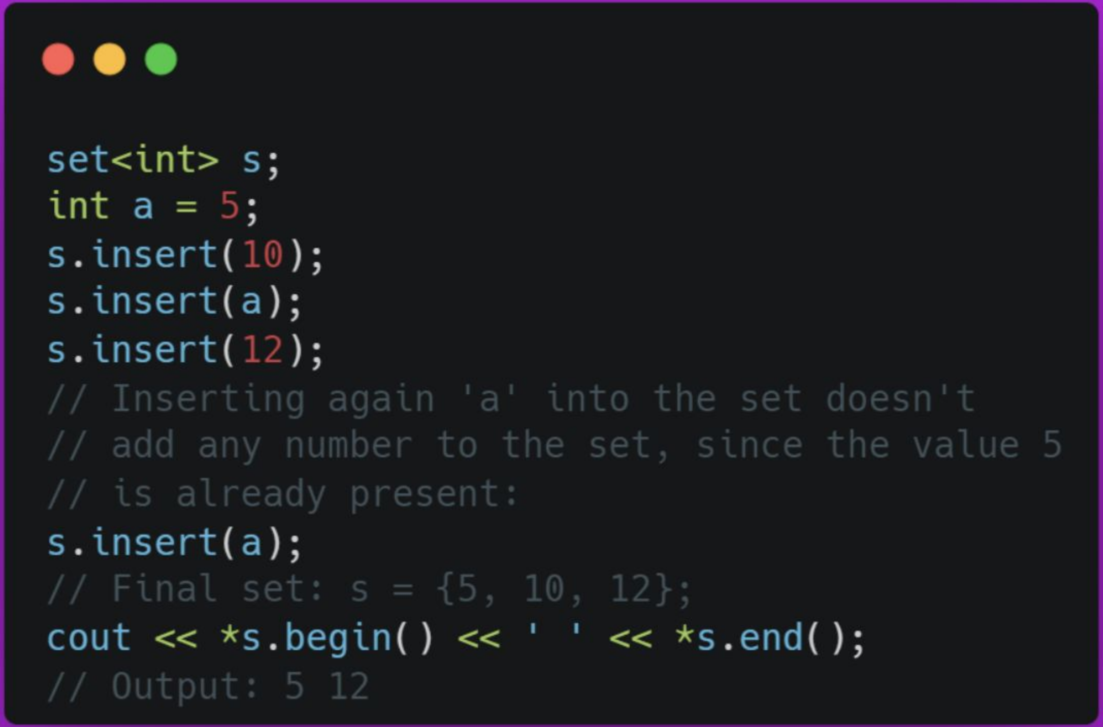
Sets

C++ set is a container which stores unique elements in increasing order. We can define an integer set in the following way:

```
set<int> s;
```

It's possible to insert and erase element, and also accessing the smallest and biggest elements from a set. Let's look at some functions for integer sets:

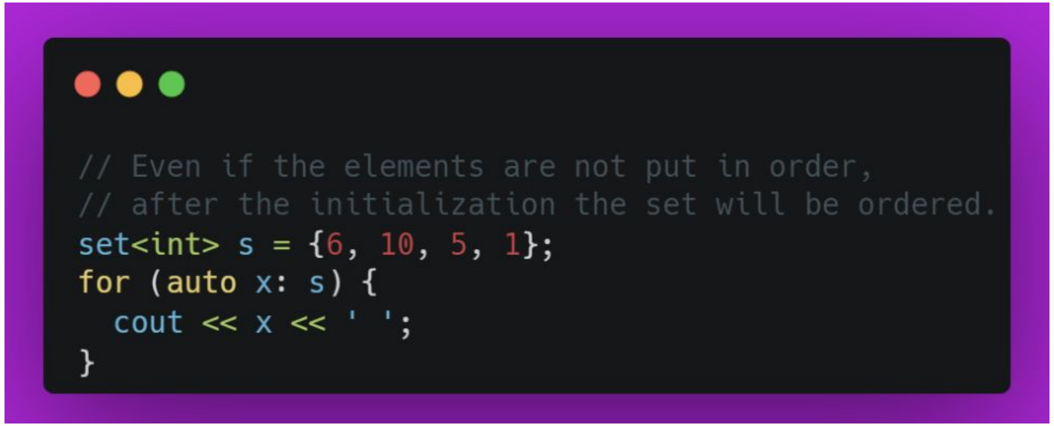
- Insert or delete an element x with `s.insert(x)` and `s.erase(x)` (where x can be either a raw value or another variable)
- Get the iterator pointing to the first or last elements by using `s.begin()` and `s.rbegin()`. If we want to get the actual values, we have to dereference those iterators using `*` .



```
set<int> s;
int a = 5;
s.insert(10);
s.insert(a);
s.insert(12);
// Inserting again 'a' into the set doesn't
// add any number to the set, since the value 5
// is already present:
s.insert(a);
// Final set: s = {5, 10, 12};
cout << *s.begin() << ' ' << *s.end();
// Output: 5 12
```

Sets in C++ are implemented as BSTs, which means that insertion, search and deletion operations for sets work with complexity O(log(n)), where n is the number of elements in the set.

Unlike vectors, elements in a set don’t have indexes. For example, we can not access second element like `s[2]`. But still, we can iterate by element in a set, as we did for vectors.



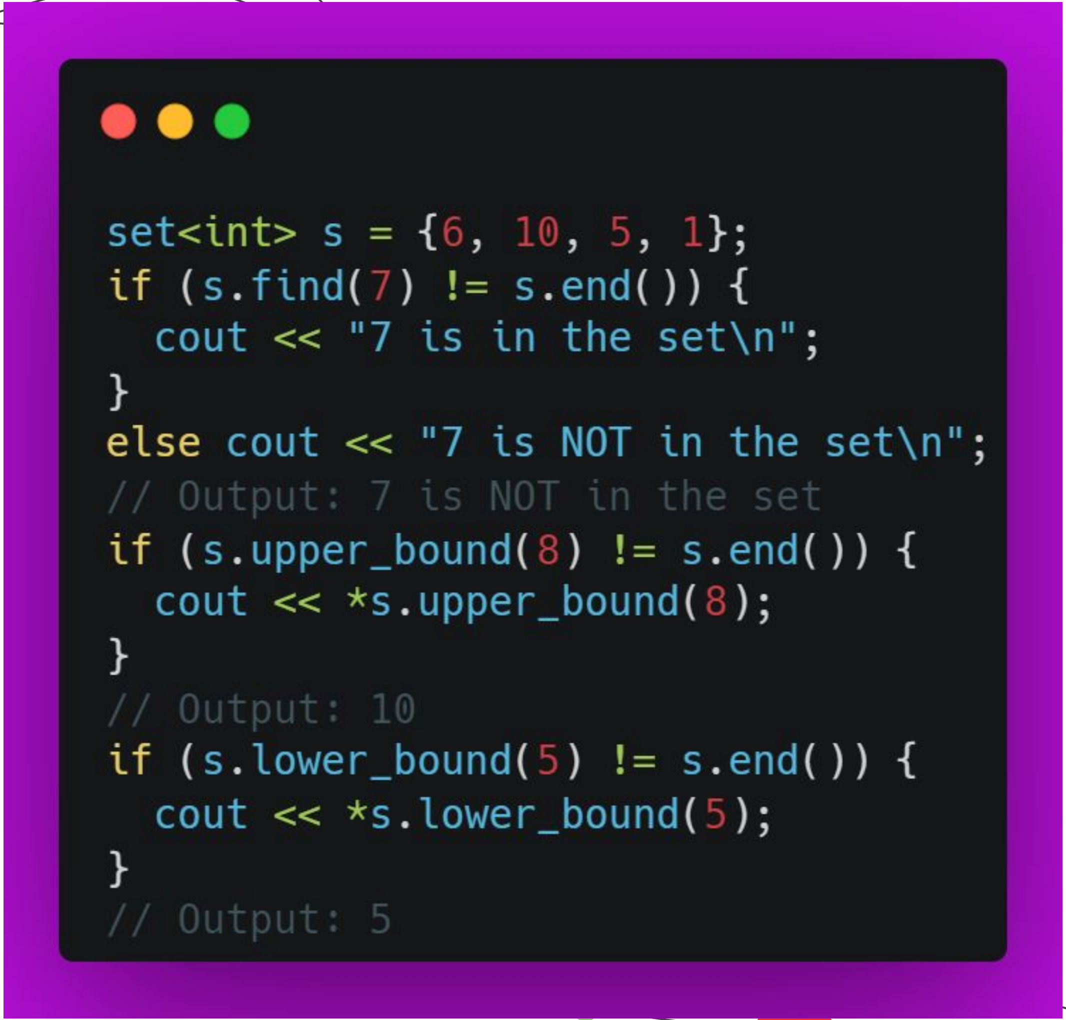
```cpp
// Even if the elements are not put in order,
// after the initialization the set will be ordered.
set<int> s = {6, 10, 5, 1};
for (auto x: s) {
  cout << x << ' ';
}
```

Here are some functions to search for specific elements in a set:

- `s.find(x);` returns an iterator that points to the element in the set with the value x. If x is not present in the set, `s.find(x)` returns `s.end()` (which points **outside** the set)

- `s.upper_bound(x);` returns an iterator to the smallest value in the set which is greater than x. If such a value is not present, it returns `s.end()`.

- `s.lower_bound(x);` returns an iterator to the smallest element in the set which is equal or greater than x (it's almost the same as the `upper_bound()` function, but with >= instead of >). If such a value is not present, it returns `s.end()`.



```
set<int> s = {6, 10, 5, 1};
if (s.find(7) != s.end()) {
  cout << "7 is in the set\n";
}
else cout << "7 is NOT in the set\n";
// Output: 7 is NOT in the set
if (s.upper_bound(8) != s.end()) {
  cout << *s.upper_bound(8);
}
// Output: 10
if (s.lower_bound(5) != s.end()) {
  cout << *s.lower_bound(5);
}
// Output: 5
```

The functions we have just seen are useful if we want to determine whether certain elements are present in the set or not. If we are not sure that the element we are searching for is present in the set, **always** make the check with `s.end()`.
Those functions are implemented using binary search, so their time complexity is O(log(n)).

Binary Search on sorted vector!

`find()`, `upper_bound()` and `lower_bound()` can be also used for **sorted** vectors, even though their syntax is slightly different:

```
upper_bound(vec.begin(), vec.end(), 6);
```

will return an iterator to the upper bound of 6 in the vector v.

Multisets

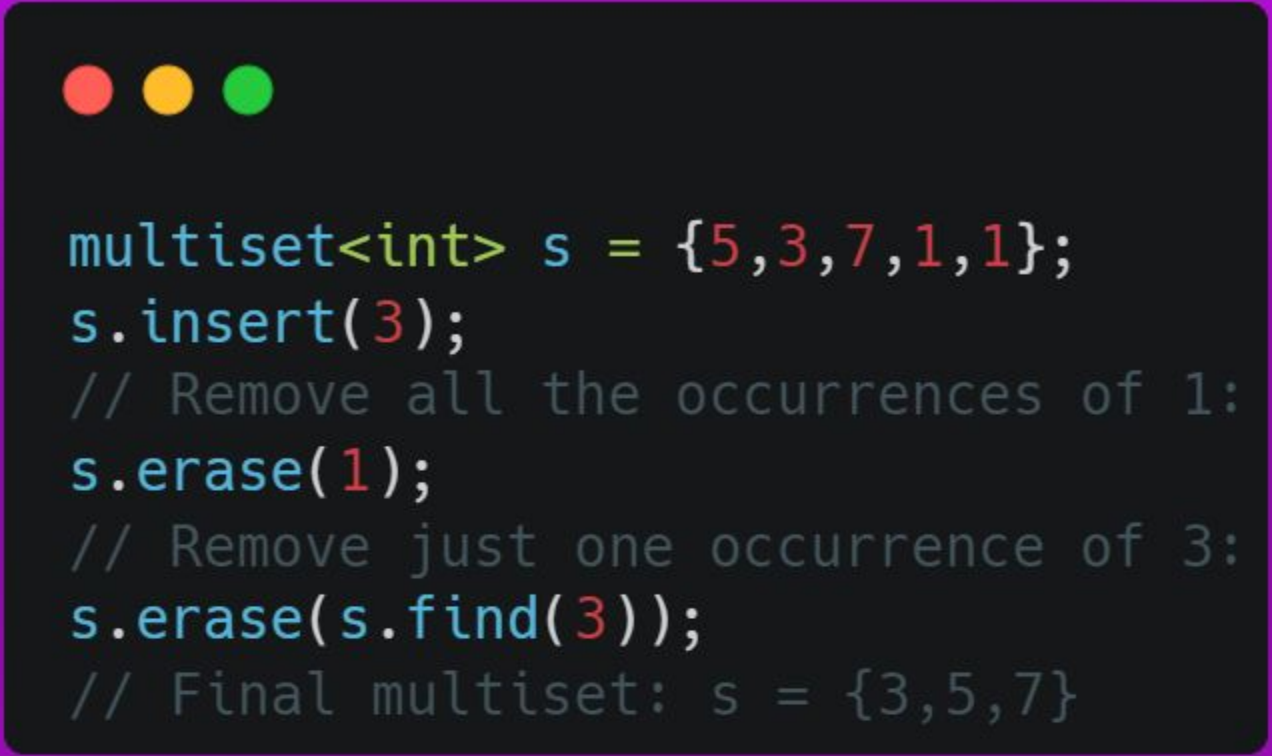
A multiset is just like a set (it is ordered and works in logarithmic time), but the difference is that it allows multiple entries with the same value. We can define one with:

```
multiset<int> s;
```

We need to be careful when we want to remove one occurrence of one value from the container. If we use `s.erase(x)`, where x is the value, we will delete **all** the occurrences of x. Instead, we have to write:

```
s.erase(s.find(x));
```

where `s.find(x)` returns an iterator to one occurrence of x. Once again, we must be sure that `s.find(x)` doesn't return `s.end()`, otherwise we will break everything!



```
multiset<int> s = {5,3,7,1,1};
s.insert(3);
// Remove all the occurrences of 1:
s.erase(1);
// Remove just one occurrence of 3:
s.erase(s.find(3));
// Final multiset: s = {3,5,7}
```

Maps

A map is a container storing **key-value pairs**. We can define a map with the syntax:

`map<”data type 1”, “data type 2”> myMap;`

The keys will have data type *“data type 1”*, while the values will have data type *“data type 2”*.

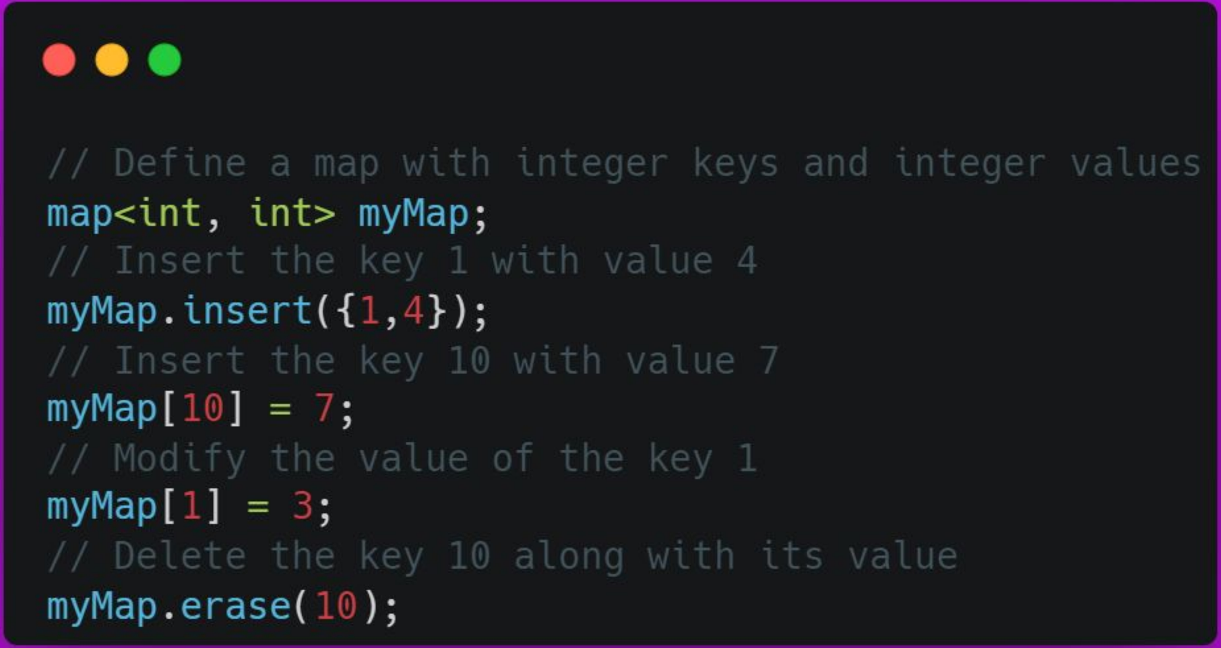
Keys in a map are **unique** and **ordered**. In fact, also maps are implemented as BSTs (like sets) with respect to the keys.

We can insert keys by doing one of these:

1. `myMap.insert({key,value});`
2. `myMap[key] = value;` (This can also modify the value of a present key)

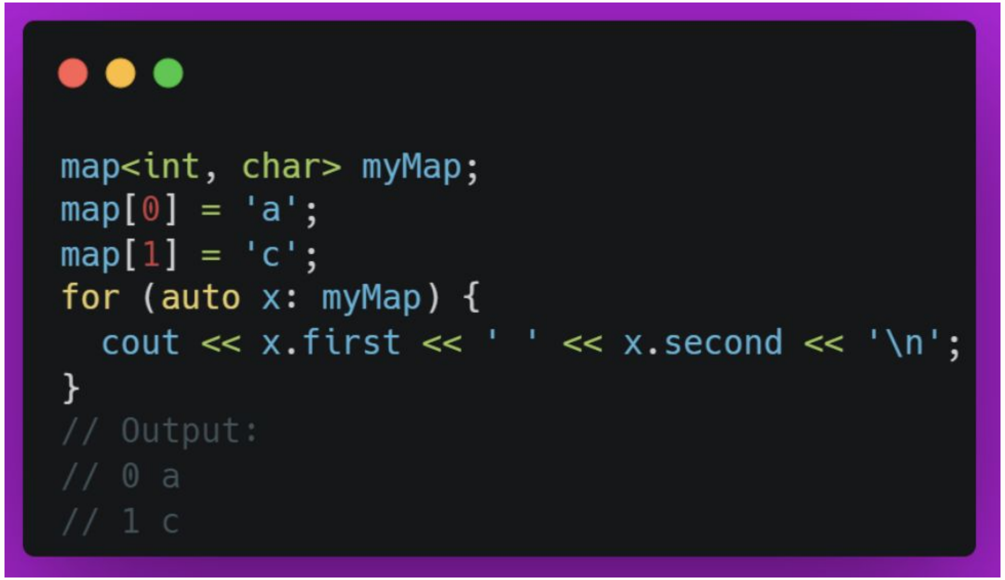
`myMap.erase(key);` To delete a key from the map.

`myMap..find(key);` To determine if a certain key is already present in the map



```
// Define a map with integer keys and integer values
map<int, int> myMap;
// Insert the key 1 with value 4
myMap.insert({1,4});
// Insert the key 10 with value 7
myMap[10] = 7;
// Modify the value of the key 1
myMap[1] = 3;
// Delete the key 10 along with its value
myMap.erase(10);
```

We can iterate through a map by element. This time, however, the variable x is the **pair** containing a key (first element) and its value (second element), not only a single element.



```
map<int, char> myMap;
map[0] = 'a';
map[1] = 'c';
for (auto x: myMap) {
  cout << x.first << ' ' << x.second << '\n';
}
// Output:
// 0 a
// 1 c
```

We remark that maps in C++ are implemented using BSTs. This means that, as it is for sets, inserting, modifying, searching and deleting an element has **logarithmic** time complexity.

In C++ there are also hashmaps, and they can be defined by specifying the keyword "**unordered**", like:

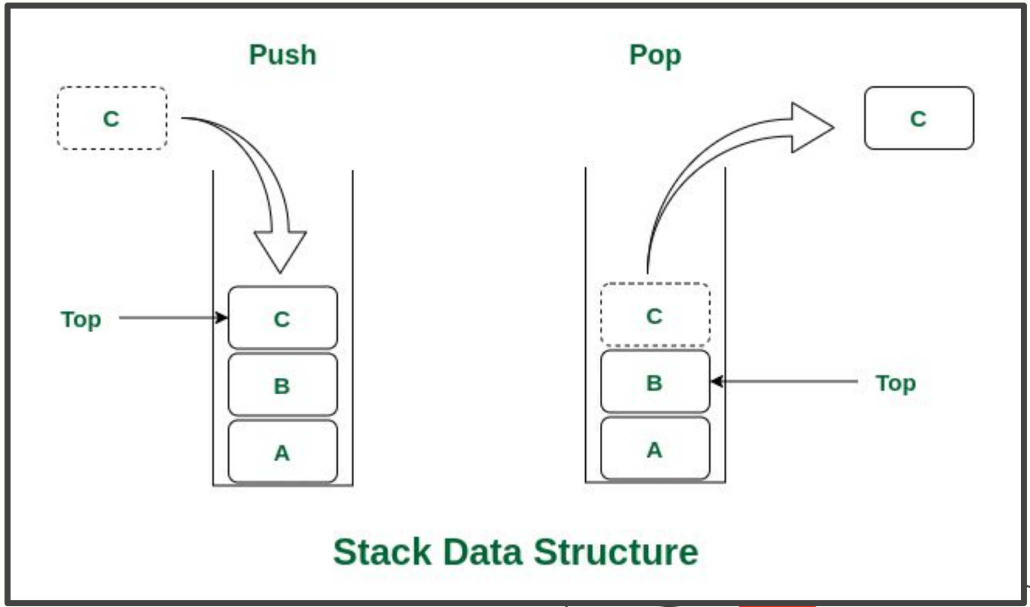
```
unordered_map<int, int> myMap;
```

Using a hashmap instead of an ordered map can have benefits in terms of time costs, but they are more delicate objects and we will not treat them here.

There are some attacks against these hashing algorithms that makes them tricky to use in competitive programming.

Stacks

A C++ stack is a **LIFO** (Last In First Out) data structure. We can think of it as a pile of dishes: when we add a dish to the pile. we add it on the top. When we remove a dish from the pile, we remove it from the top.



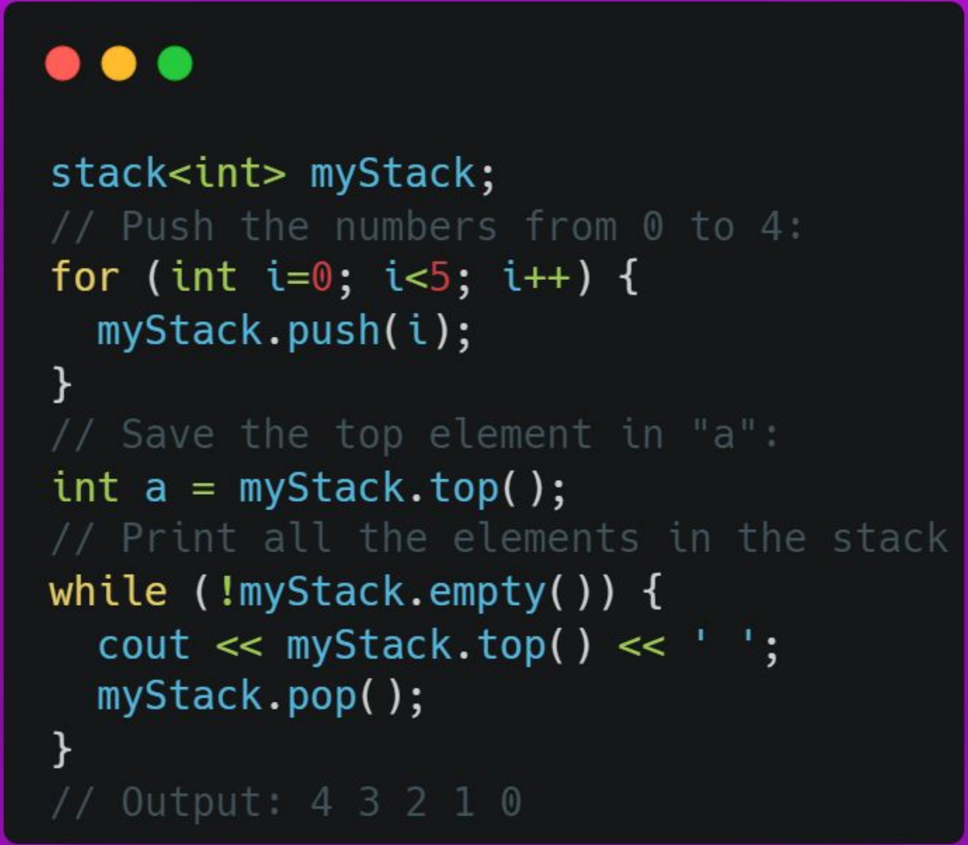
We can define a stack as in the following line:

`stack<”data type”> myStack;`

When using a stack, we only have access to its top, and we can use the following functions:

- `myStack.push(x)` pushes x to the top of the stack.
- `myStack.top()` takes the element at the top of the stack. We can store that element in another variable.
- `myStack.pop()` removes the element at the top of the stack.

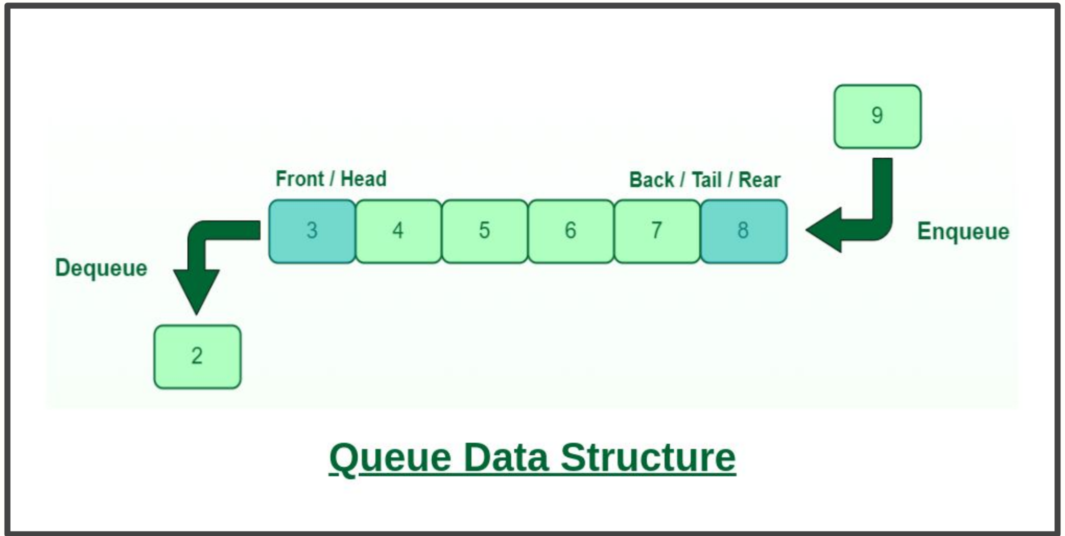
Important: To access a given element in the stack, we first have to pop all the elements which are on top of it.



```
stack<int> myStack;
// Push the numbers from 0 to 4:
for (int i=0; i<5; i++) {
  myStack.push(i);
}
// Save the top element in "a":
int a = myStack.top();
// Print all the elements in the stack
while (!myStack.empty()) {
  cout << myStack.top() << ' ';
  myStack.pop();
}
// Output: 4 3 2 1 0
```

Queues

A C++ queue is a **FIFO** (First In First Out) data structure. We can think of it as an actual queue: the first who enters the queue is the first to go out of the queue.



We can define a queue as in the following line:

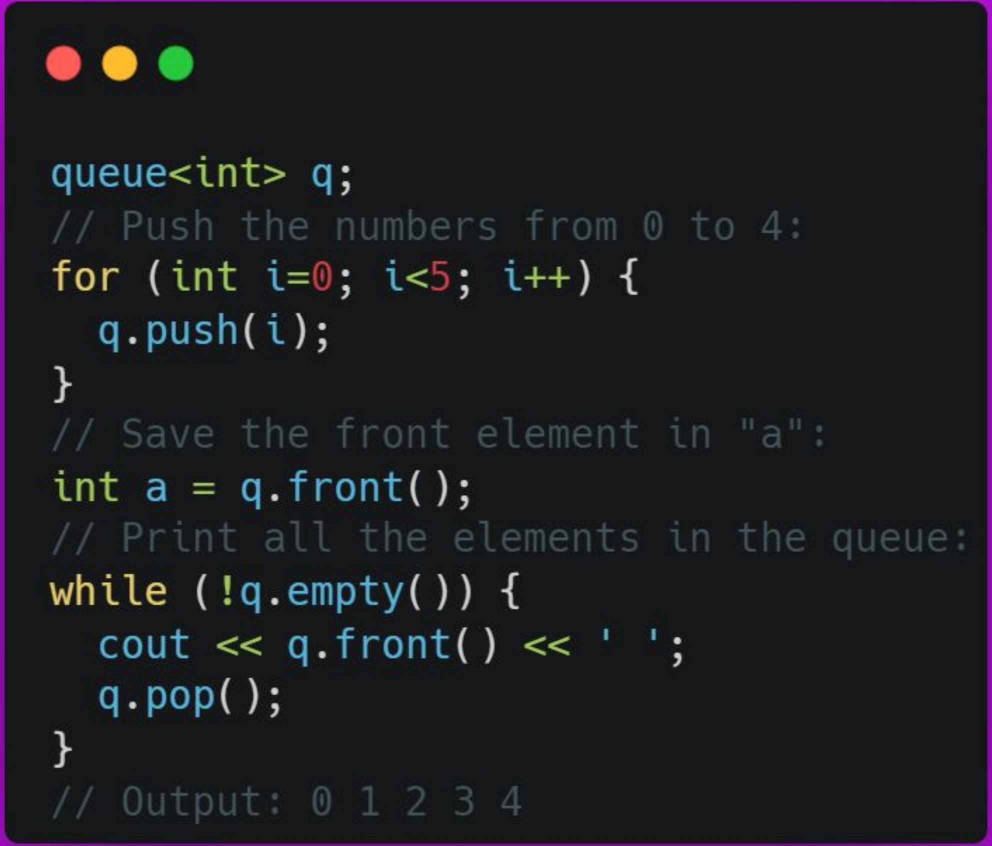
`queue<”data type”> q;`

When using a queue, we only have access to its front, and we can use the following functions:

- `q.push(x)` pushes x to the back of the queue.
- `q.front()`/`q.back()` takes the element at the front/back of the queue.
- `q.pop()` removes the element at the front of the queue. (it also returns it’s value)

Important: To access a given element in the queue, we first have to pop all the elements which are before of it.

Note: for both stacks and queues, push, pop and access operations work in constant time (O(1)).



```
queue<int> q;
// Push the numbers from 0 to 4:
for (int i=0; i<5; i++) {
  q.push(i);
}
// Save the front element in "a":
int a = q.front();
// Print all the elements in the queue:
while (!q.empty()) {
  cout << q.front() << ' ';
  q.pop();
}
// Output: 0 1 2 3 4
```

Priority queues

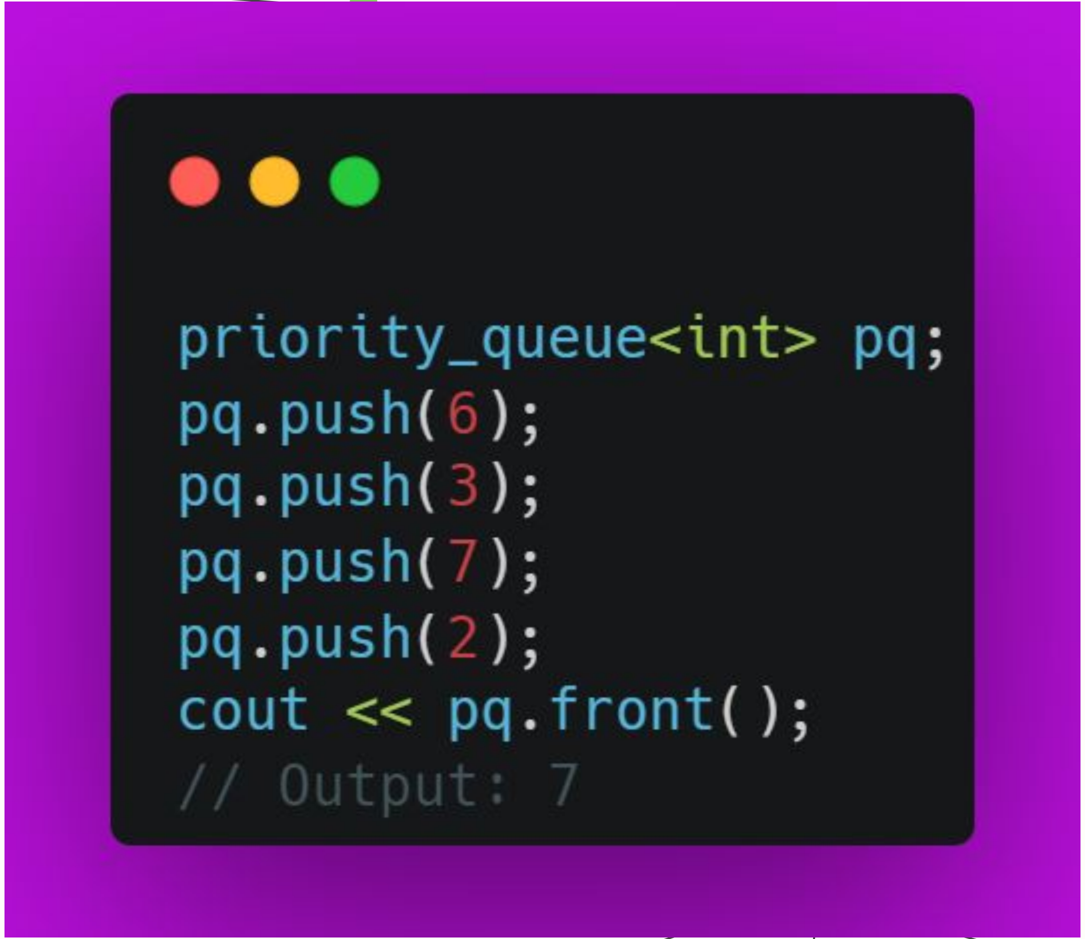
A priority queue is a special type of queue in which the element at the front is always the **maximum** element among all the values present in the queue. We can define a priority queue as in the following line:

```
priority_queue<”data type”> pq;
```

Priority queues are implemented as **heaps**, which means that pushing an element or popping the front are operations that have a **logarithmic** cost O(log(n)).

Notice that a priority queue is **not** an ordered container: it only guarantees that the first element in the queue is the maximum. (We don’t have random access in queues!)

The functions related to priority queues are the same as those related to normal queues. With one exception which is `.top()` instead of `.front()` and there is no `.back()` here.



```
priority_queue<int> pq;
pq.push(6);
pq.push(3);
pq.push(7);
pq.push(2);
cout << pq.front();
// Output: 7
```

Example:

Counting Occurrences

We are given a sequence of n integers. Each integer x in the sequence is in the range 1<=x<=x_max.

$$seq_1, seq_2, .., seq_n (1 \leq seq_i \leq x\_max)$$

For each number from 1 to x_max, we want to count how many times it appears in the input sequence.

For example, if sequence = [1, 5, 7, 8, 1, 1, 7] we will have:
occurrences[1] = 3
occurrences[5] = 1
occurrences[7] = 2
occurrences[8] = 1

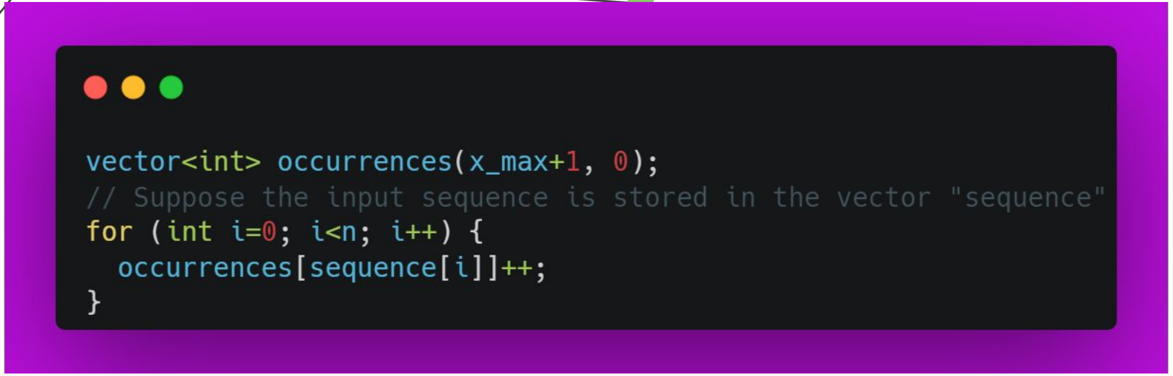
There are two possible options:

- We keep a vector in which, for each position i in the range $[1,x\_max]$, we store the number of occurrences of i.

  Each element in the vector is initialized with the value 0 — the number does not appear in the sequence.

- We keep a map that associates to each integer from 1 to x_max the number of its occurrences. We insert in the map only the values that appear at least one time in the sequence.

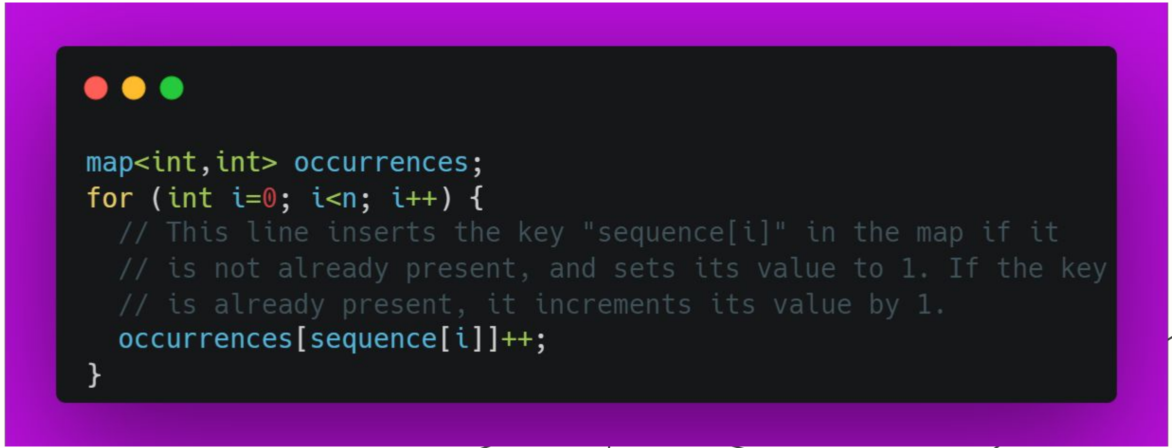
In the next slide there are the implementations for the two approaches.



```
vector<int> occurrences(x_max+1, 0);
// Suppose the input sequence is stored in the vector "sequence"
for (int i=0; i<n; i++) {
  occurrences[sequence[i]]++;
}
```

← Using a vector

Time Complexity: O(n + x_max)
Memory Complexity: O(x_max)



```
map<int,int> occurrences;
for (int i=0; i<n; i++) {
  // This line inserts the key "sequence[i]" in the map if it
  // is not already present, and sets its value to 1. If the key
  // is already present, it increments its value by 1.
  occurrences[sequence[i]]++;
}
```

Using a map →

Time Complexity: O(n * log(n))
Memory Complexity: O(n)

Suppose that x_max can be at most 10^5.

Should we use a vector or a map in order to store the number of occurrences of each number?

Which one of the two approaches is more efficient?

Time Complexity: O(n + x_max)
Memory Complexity: O(x_max)

In this case, it is preferable to use the vector!

In terms of memory, a map could be more efficient but not in the worst case. Consider that if all the n elements are different, both map and vector would have *O(n)=1MB* memory!
So we can move our focus to the time costs.

For the time costs, vectors are more efficient, because updating a value inside them has complexity O(1). Instead, modifying a value in a map works in O(log(n)).

Now, Suppose that x_max can be at most $10^\wedge 9$.

Does the previous approach still work or we have to do something else?

Time Complexity: O(n * log(n))
Memory Complexity: O(n)

Unfortunately, here we are forced to use a map. We cannot afford to store in memory 10^9 integers in a vector, because it would take 4GB!

Instead, with $n = 10^5$, a map will contain 10^5 different keys in the worst case.

This example was made to highlight the advantages and disadvantages of the two data structures and show memory/time trade off.

✚ Actually, there are some algorithms and techniques to handle the case in which x_max = 10^9 in a more optimal way, but this is another story.

Thank you for attending!

CPPoliTO

Competitive Programming Team of Politecnico di Torino

Resources

- Last year presentation on Youtube CPPoliTO: Introduction to C++ for Competitive Programming
- C++ Vs Python: Overview, Uses & Key Differences
- CodeForces blog “Share your template!”
- C++ for Competitive Programming - Tips and Tricks in 10 min
- GeeksforGeeks Blog: C++ tricks for competitive programming
- Programiz: C++ Standard Template Library

Follow us!

Website
LinkedIn
Linktree