C++

Last Update: Jan 2022.

1. Stages:
   1.1 Preprocessing- #
   1.2 Compilation
   1.3 Linking
2. Types:
   int – 4b
   char- 1b
   float – 4b
   long – 4b
   long float – 8b
   long long int – 8b
   int a[];

   2.1 Data types:
   primitive:
   int, char, double, float, long, long long

   2.2 common functions for containers:
   begin(),
   end()
   size(),
   clear(),

   2.3 sequence containers:
   vector
   array
   list: 2 way linked list
   forward_list:
   deque: double ended queue, takes O(1) time to add/remove element from the ends.
   queue: FIFO
   stack: LIFO
   string: uses char for each element
   wstring: uses wchar_t for each element, wchar_t is wide character and holds a much greater number of values than a char . prefix l<string>.
   bitset
   priority_queue

   2.4 Bitset: The array that takes the least amount of space in memory as each element can be either 1 or 0 hence they only take 1 bit per element.
   To declare a bitset,
   bitset<10> s(string("0010011010")); // from right to left
   cout << s[4] << "\n"; // 1
   cout << s[5] << "\n"; // 0

   or create like a normal array.
   2.4.1 They can be used for bit operations

```
bitset<10> a(string("0010110110"));
bitset<10> b(string("1011011000"));
cout << (a&b) << "\n"; // 0010010000
cout << (a|b) << "\n"; // 1011111110
cout << (a^b) << "\n"; // 1001101110
```

2.4.2 <bit array var>.count() returns the number of 1s.

2.5 Priority Queue: By default a PQ sorts the elements by value, so the largest element gets removed first. It is like a set but it is faster.
To declare one,
priority_queue<int> q;

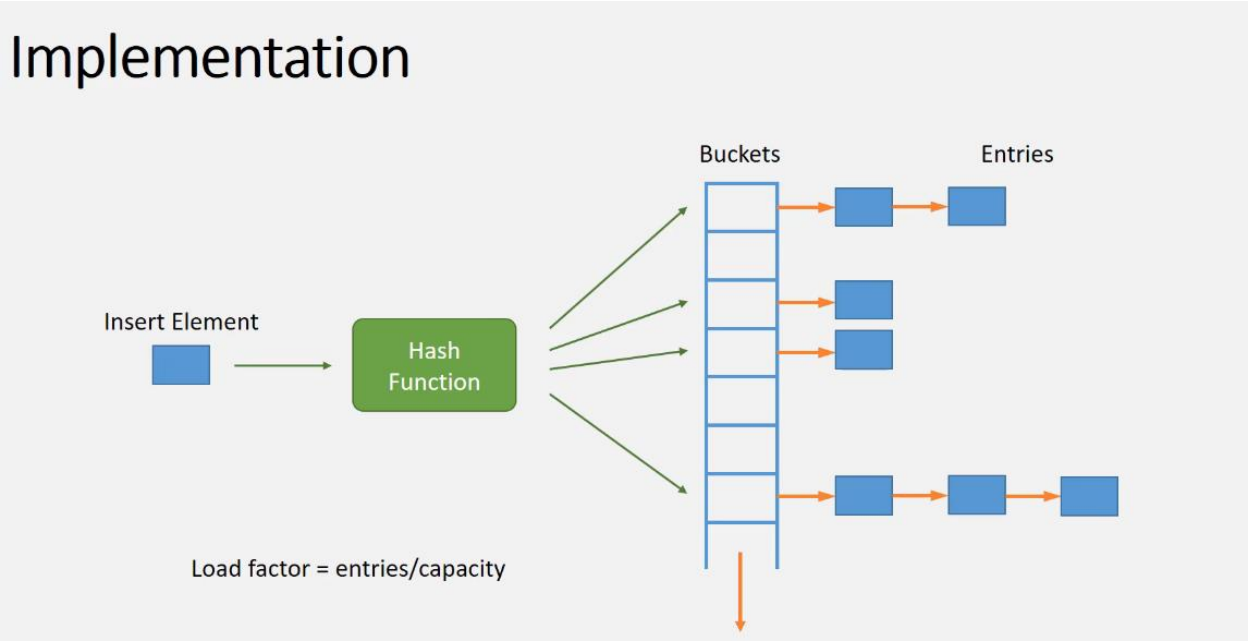
Declare a PQ which pop/top the smallest value.
priority_queue<int,vector<int>,greater<int>> q;

2.6 Stack, queue and PQ support pop(), meaning remove the element at the front. PQ and Stack also have top method to get the element at front without removing it, the same in Queue can be done with front.

2.7 char8_t utf8str[]= u8"\u0123"; standard type to hold utf8 strings.

2.8 associative containers:
set
multiset
map : <map obj>.erase(<key>) to remove a key. doesn't allow duplicate keys, new keys are discarded.
multimap

2.9 unordered containers: hashes the value and stores it, also generates a key that is basically used as its index in the container. doesn't sort like the associative containers.



each entry actually stores a pointer to a linked list which will hold the values, if theres collision (i.e 2 different values have same hash they are added at the end of the list. bucket size is not dynamic , it does increase but only if the no. of elements are more than the bucket size. when the bucket size increases, all values will be rehashed , this time the key limit will be bigger so lesser chance of collision.

unordered_set: bucket_count() -> returns the size of the bucket, size()-> returns the no. of elements, load_factor() -> returns the load factor, bucket(<key>) -> returns the bucket # where key is the retrieved from the iterator
unordered_multiset
unordered_map: the map's key is used for hash.
unordered_multimap

2.10

Complexity of Operations

Container	[ ]	push_back	pop_back	insert	erase	find	sort
array	O(1)					O(n)	O(n log n)
vector	O(1)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n log n)
deque	O(1)	O(1)*	O(1)*	O(n)	O(n)	O(n)	O(n log n)
list	NA	O(1)	O(1)	O(1)	O(1)	O(n)	O(n log n)
forward_list	NA	NA*	NA*	O(1)	O(1)	O(n)	NA
set/multiset	NA	NA	NA	O(log n)	O(log n)	O(log n)	NA
map/multimap	NA	NA	NA	O(log n)	O(log n)	O(log n)	NA
unordered_set/ unordered_multiset	NA	NA	NA	O(1)	O(1)	O(1)	NA
unordered_map/ unordered_multimap	NA	NA	NA	O(1)	O(1)	O(1)	NA

2.11 std::hash<t> <varname>; use <varname>(<value>) or std::hash<t>{}(<value>) to return the hash of the value

2.12 Policy based data structures: Apart from the data structures in the STL, there are some added by G++, they are called by this name.
To use them in a g++ env.,
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;

typedef tree<int,null_type,less<int>,rb_tree_tag,
tree_order_statistics_node_update> indexed_set;

and now we can use indexed_set defined by g++. This is a set that can be accessed with indices as well. Time taken in logn

2.13

3. std::begin(a); returns the beginning address of an array

std::end(a); returns an address after the end of the array. This can be used to iterate over an array till the address is equal to end. The other functions like lower bound, find etc. return an iterator and if they don't find an element the iterator returned is end.

3.1 The iterator can then be used to access the element, *<it object> to get the element's value.

3.2

4. Ref:
   4.1 std::ref(<object>); creates a reference and returns the reference.
   4.2 std::cref(<object>); creates a constant reference and returns the constant reference.

5. atexit(<function object>) – runs stuff after main end, if we send a function pointer or a lambda(empty capture list, otherwise it doesn't decay to a function pointer) , it will be run at exit.
6. Initialization:
   scalar types
   int a= 0 ; copy initialization;
   int a(5); direct initialization;
   int a(); most vexing parse
   int a{2}; aggregate initialization
   int a{}; value initialization
   vector types
   int a[];
7. &variable – returns the address
8. Designated Initializers: A way to initialize an object.

```
struct A {
  int x;
  int y;
  int z = 123;
};

A a {.x = 1, .z = 2}; // a.x == 1, a.y == 0, a.z == 2
```

9. for(auto &x: <container that supports iterators>){ }; runs a loop with x having the value of each element in each iteration. we can use it for user-defined types or primitive data types too.
   like:
   int a[5]{1,2,3,4,5};
   auto beg=begin(a);
   auto e=end(a);
   for a(;beg!=e;++beg)
    auto v=*beg;

   will work for any primitive data type as the begin and end function return their iterators which we can use in our own for loop.

10. for(auto &[key,value]: <container with pair objects>){} ; useful for iterating over pair object vectors,maps etc. as the key is the pairs first and value is the pairs second object here.
11. <data type> (&<varname>)[sizeofarray]=<arrayname>; array reference.

12. int a=2;
    a – lvalue
    2- rvalue

int &&value = 2 ; rvalue reference , only takes rvalues.

13. .h header files
    .cpp cpp files.

14. inline <function declaration> - causes the func to replace the call wherever it is called

15. <return type> (*<name>) (<parameter data types>) = <funcname> ; function pointer

16. namespace – like a class minus oops
    namespace <name of namespace>
    {
    stuff
    }

17. Using <namespace name>;
    open either the type of stuff inside or the namespace .
    namespace without the name means it is automatically open and the stuff can only be accessed inside the same file.

    namespace can have nested namespaces and can be accessed like,
    <namespace 1>::<namespace 2>::<some function>();

18. using can be used to open class enums in C++2a.
    enum class rgba_color_channel { red, green, blue, alpha };

    ```
    std::string_view to_string(rgba_color_channel my_channel) {
      switch (my_channel) {
        using enum rgba_color_channel;
        case red:   return "red";
        case green: return "green";
        case blue:  return "blue";
        case alpha: return "alpha";
      }
    }
    ```

19. class and structure are same thing except struct have access modifier as public by default.
    initialize stuff inside class by normal initialization , if theres a constructor defined then that replaces the value normally initialized.

    19.1 all class member functions pass a this pointer by default, use this-> to access a member variable
    19.2 <function signature()> const makes the function unable to modify the state of the object
    19.3 static variables depend on class not on object of class and have to be initialized in the cpp file not in the header or where class is declared unless you use inline before static.
    19.4 ~<classname()> - destructor
    19.5 <class name> <class name>(const <class name> &<var name>) – copy constructor
    19.6 <class name><class name>(<class name> &&<varname>) – move constructor
    19.7 In cpp files we can delegate a constructor call using : , <constructor()>: <some other constructor(any values to pass>
    19.8 when we declare parameterized constructor the normal constructor is deleted.
    use <constructor()> = default to have a default implementation of the function.
    or =delete to disable a function call.

    19.9 <return value> operator <operator type> (arguments) – operator overloading
    argument number should be same as operands.
    void operator <return type>() – type conversion operator overloading, declared inside the class and returns any value specified.

19.10 <class name> var= 1 is equal to <class name> var = <class name> var(1) through implicit type conversion
which can be blocked through explicit keyword on constructor.

19.11 If a ctor uses explicit keyword then it must be invoked explicitly.
In C++2a, explicit can be applied to a ctor based on a condition using templates.

```
struct foo {
  // Specify non-integral types (strings, floats, etc.) require explicit construction.
  template <typename T>
  explicit(!std::is_integral_v<T>) foo(T) {}
};

foo a = 123; // OK
foo b = "123"; // ERROR: explicit constructor is not a candidate (explicit specifier evaluates to true)
foo c {"123"}; // OK
```

19.12 class <class name> : <access modifier> <parent class>
19.13 private attributes are hidden from child class. protected attributes are only accessible by child classes and public can be accessed by anything .
19.14 the inheritance access modifier changes the access modifiers for the attributes only for the inheriting class, except for private attributes which remain unaffected and inaccessible.
public – changes nothing.
protected – makes public and protected attributes protected
private – changes every attribute to private

19.15 Create a class:
create a header file then put signature of functions and attributes, attributes can be initialized here.
put the constructor in public .
then create its .cpp file and include “header name.h”
<return type> classname::funcname(parameters){}; to initialize the different fields of the class including the constructor.
<child class constructor>:parent class constructor call ; in cpp file, to initialize the parent class as well.
19.16 class name::function name , from the child class to call the base class’ function.
in header file, using classname::classname we can inherit all constructors
19.17 virtual <function signature> to allow the functions to be overridden by child classes.
polymorphism only works through pointer or reference.
base class object pointer can point to any of its child classes’ object.
if theres inheritance, base class destructor should always be virtual.
19.18 class name final{}; to disallow a class from being inherited.
<func signature> override{} ; to allow the compiler to make sure a virtual function is being overridden.
<func signature> override final{} ; to disallow any more child classes from inheriting the function.
base class obj pointer = child class obj address; upcasting . if we don’t use pointers or references the child class features will be sliced off .
child class obj pointer = base class obj address ; downcasting , needs manual casting.
19.19 rtti: determine type of objects in runtime , works only for objects with atleast 1 virtual function.
typeid(object or class).name() to see it.

19.20 virtual <func signature> =0 ; is pure virtual declaration, makes the class an abstract class , and that means the child classes will need to override the pure virtual function . and the class cant be instantiated, can be used through a ptr or ref though.

19.21 use virtual inheritance to disallow creation of multiple instances of a single base class in child classes. This is a solution to diamond inheritance problem.

```
class A {
virtual void pp();
}
class B: virtual A {
    virtual void bc(){...}
}
class C: virtual A {
    virtual void dd() {..}
}
class D: B, C {...}
```

Without virtual inheritance B and C would point to 2 different A's and the ambiguity would require explicit casting of a D object to B or C to use A's methods, the diamond problem. With virtual inheritance, B and C point to the same A and hence this issue is solved.

19.22 try {}
catch(<error type> &x){} use reference to not create a copy.
throw <exception object>(string to pass to catch blocks or to compiler).
ex.what() holds a string given to the exception when it was rasied

20. friend <function signature> inside class to make the function able to access stuff inside the class or structure

20.1 friend <func sig> or func <classname>: allows this method/class to access private props of a class. The function itself can have body but it can't be used. The main purpose of friend is to allow a func/class see the private props from outside the class.

```
#include <iostream>

class ABC {
private:
        int b{ 2 };
public:
        friend void abcs(int);
};

void abcs(int a)
{
        std::cout << "yae \n" << ABC().b;
}
int main()
{
        auto a{ ABC() };
        abcs(2);
        std::cout << "Hello World!\n";
    }
```

21. (data type) (var name) – c style caste
    static_cast<data type>(var name) – c++ style casting
    reinterpret_cast<data type>(var name) – c++ style casting which forces conversion .

    static_assert(<the check>, <string message to pass to compiler>); checks an expression at compile time

22. unique_ptr -
    unique_ptr<type> varname{make_unique<type>(value)};
    or unique_ptr<type> varname{ new int{value}};
    but in this case the value can only be accessed by *varname.get()

    or auto varname= make_unique<type>(value);

    for array
    unique_ptr<type> varname{new int[size of array] };
    auto varname=make_unique<type[]>(size of array);

    and put values as varname[0]=2; ....

    22.1 same for shared_ptr except it allows copying but unique doesn't.
    shared_ptr varname.use_count() returns no of copies.
    22.2 weak_ptr<data type> varname{shared_ptr};
    weak_ptr varname.expired() returns bool if shared ptr is null

    22.3 shared_ptr<data type> varname=weak_ptr varname2.lock() returns 1 shared ptr to make it not go null.
    22.4 In C++2a, make shared accepts int array.
    auto p = std::make_shared<int[]>(5); // pointer to `int[5]`
    // OR
    auto p = std::make_shared<int[5]>(); // pointer to `int[5]`
    22.5

23. enum color{red,blue,green};
    this enum is open to all items in cpp.
    int a=red; will be 0

    23.1 enum class color: <data type>{red=<data type value>,blue,green};
    blue is next ascii value for non-numeric types and same for green automatically.
    int a=color::red; is 1
    color pp{static_cast<color>(<data type value>)} ; is green

24. strings
    24.1 getline(cin,string object); to get input into a string
    24.2 cin.getline(char array obj, size) ; to get input into a char array
    24.3 while(getline(string stream obj(string),string obj, char obj)); to split a string

string obj.insert(position, const char *) ;
string obj.clear() to clear a string.
string obj.erase(start index, stop index) ; to remove part of a string
string obj.find(const char*); returns the position of substring , or string::npos if not found.

24.4 using std::string_literals;
string b="hello"s ; s here indicates it is a string and not a declaration of a char* which it normally would be, and then converted to a string to be stored in the string container.

string b=r"hello \s \f \t"; will be a rawstring and any escape sequence inside will not be processed.

string b=r"asb(<the message>)asb" ; asb is a delimiter and (marks the start of the string while) marks the end, delimiters are not included in the string but act as a starting and ending sequence.

24.5 \ is an escape character meaning the next character is applied as is, like \ itself wont be printed but \\ prints a \ .

24.6 \n or \t and the like are called escape sequence and convey special meaning to the compiler .

24.7 for(auto &x: string obj) {} works. puts each character in x.

24.8 literals: We can define custom literals in c++ by overloading the operator"". For example, if we have a custom class DistanceInKm which stores and returns a float which is distance in km, we can provide literals to convert other Distances, such as in miles.
Syntax: <return type> operator"" _<literal>(<argument>){...}
Like:
class DistanceInKm{};
DistanceInKm operator"" _miles(long double val){
Return DistanceInKm{val*1.6};
}
And we can use it like DistanceInKm dis{24_miles}; which will store 24*1.6 in dis object.

We can only provide literals for the following base classes, int, long, char, char* , float and double. And the argument must be of unsigned long long for int, long double for float/double, char, char16_t, char32_t for characters and const char * for strings. This operator can only be overloaded in the global scope and must have _ as the no underscore literals are reserved by std namespace

24.9 stringstream
ss obj{string obj}; to initialize a stringstream to a string.
obj<<""<<a+b<<1 ; works like cin
or obj.str(value); to clear the stream and put the value in it.
obj>>a; puts a value of its data type in the object a.
string var=obj.str(); works
to_string(obj) ; converts any data type into string and returns it.
stoi(string); retruns a string converted into int.
obj.fail() is set when the stream is empty.
char ch{};
while(obj.get(ch)) {...} gets each character in the stringstream and puts it in ch .

25. constexpr <func signature>() to make the func a constant expression, i.e it will be processed at compile time, given its return is a literal (void, scalar types, references etc.)

26. even constexpr func (int a, int b) return a+b will be compile time. (as it only accepts 1 line , i.e the return line to be a constexpr, accepts if else and switch statements).
    if the constexpr func returns a variable or is called with a variable it will be not be processed at runtime , may not even work.

    constexpr variables need to be initialized at compile time and are processed right then.

27. constexpr can be used with virtual functions from C++2a.
28. consteval: Used to create Immediate functions, these are just like constexpr functions but they must return constants.

```
consteval int sqr(int n) {
  return n * n;
}

constexpr int r = sqr(100); // OK
int x = 100;
int r2 = sqr(x); // ERROR: the value of 'x' is not usable in a constant expression
            // OK if `sqr` were a `constexpr` function
```

29. {2,3,4} is an initializer list as it is under {}
    std::initializer_list<t> x ; holds such an initializer list and can be traversed using iterators..
    can only be initialized during compile time .

30. union <name>{} public by default
    is the same size as largest of its members taken automatically during compile time, can initialize only 1 member at any time.
    accepts constructor and destructor.
    we can manually assign a value to any member and that will be the active member , all others will be lost.
    if a union has a user defined object with custom constructor or destructor then union needs a custom constructor and destructor. and the destructor needs to be called manually .

31. new – allocates and initializes the memory
    placement new – only initializes the memory

    new(obj address) <data type>{value}; placement new
32. std::exception is the parent class of all exceptions.
33. catch(...) to catch all exceptions , even the ones not defined in exception .
34. Random:
    #include<random>
    std::default_random_engine obj(<seed>);
    std::uniform_int_distribution<t> obj2(0,6);
    for seed use time() , time(0) is the time in seconds since epoch.
    obj2(obj) to get a random number.
    a random number in cpp11 is generated using a combination of a generator and a distribution type.
    a generator takes a seed , that's the first value its gonna be and then a distribution uses the generator to generate the next value using the generator. distribution can take range of values.

35. <func signature>noexcept(true) or just noexcept informs the compiler that the function will not throw an error , that means any try catch blocks are skipped over the call of the function. and if the function does throw it will cause a crash.

36. noexcept(noexcept(function)) takes the noexcept status of the function and applies the same to this function.
37. destructors should not throw and are implicitly marked with noexcept.
38. std::cout<<std::boolalpha<<<any boolean> ; will print the boolean in english instead of 0 or 1
39. move constructors and assignment should have noexcept , as otherwise they are skipped by some objects such as vector and instead copy constructors are invoked.
40. File I/O:
    #include<fstream>
    ifstream obj{"filelocation"}; obj will open a file, and the contents of it will be inside it in a string manner, that is how seekg,tellg etc. work.
    string a;
    getline(obj,a); puts the data of obj file in 'a' string
    works like stringstream;
    we can use obj.is_open(); which returns a bool if the file exists.
    obj.get(ch) works here as well.
    obj.tellg(); returns the current location of the get pointer. -1 if it fails.
    obj.seekg(<int>); seeks the pointer to the int location;
    obj.seekg(<int>,std::ios::<begin/cur/end>) ; seeks the pointer to int + begin, current or end . so if you want to go at -5 from end it will be , obj.seekg(-5,std::ios::end).

    instead of using the constructor , obj.open("location") ; also works the same
    the boolean operator for the obj is overloaded to return the state of the file, returns true if file exists and is open , false otherwise.

    40.1 ifstream obk{<filename>, std::ios::binary | std::ios::out) ; opens file as binary and in write mode.
    40.2 obk.read(<char *>&<address of variable>,<size to be allocated for reading , usually size of the container>): reads the file and puts the data in the variable.
    40.3 ofstream obj{"filelocation"};
    obj<<any data;
    obj.close() to close the output stream object
    obj here has obj.put(ch) instead of get() in ifstream. puts ch at current position.
    obj.seekp(<int>). works the same as seekg but for put pointer.
    ofstream obk{<filename>, std::ios::binary | std::ios::out) ; opens file as binary and in write mode.
    obk.write(<const char *>,<size to be allocated for writing>):
    40.4 hex takes lesser space than both decimal and binary, hence writing hex to files is better to save space.

    40.5 For fstream obj, both get and put pointer are moved with any i/o operation so manually seek them.

    40.6 modes:

    app : append
    binary: open in binary mode
    in: open for reading
    out: open for writing
    trunc: discards file content and writes from beginning
    ate: opens the file and puts the write from the end of the file.

40.7 bits:

good() : no error

bad(): irrecoverable stream error

failbit(): i/o failed

eof(): eof reached

good bit is set at start, the rest bits are set if an input or output failed . and when that happens, good bit is set to false.

obj.clear(); clears all bits and sets true for goodbit.

obj.setstate(std::ios::<bitname>); sets the bit to true.

41. Filesystem: Another way to work with file i/o, this is a more verbose way of working with files .

#include<filesystem>

41.1 filesystem::current_path(<new path>,<std::error_code object>); returns the current directory's path, if the new path is defined changes the current path to that. if there was an error the error code object holds it and we can check if there was an error by calling if else directly on the object as the error code object bool is overloaded.

object.message() on the error code object returns the string of the error .

41.2 filesystem::path obj(<takes location>); stores a path object or a location for a file

obj /= "sss.xxx"; to append /sss.xxx at the end of the location in path object. or use the obj.append(<value>) .

obj.string() ; returns the path in string without double \;

41.3 filesystem::exists(<path object>); returns true if the path exists else false.

if(!filesystem::create_directory(path)){failed to create directory do stuff}; create directory creates a directory and returns a boolean if it succeeded or failed in doing so.

filesystem::remove(path) removes the directory at the path and also returns a boolean.

path obj can be broken down into its subsequent types , relative_path(), root_directory(),root_name(),parent_path(),stem(),root_directory(),filename() and extension(). use obj.has_ before these to get boolean if that part exists in the path or use .string() after the methods to get that part in string or use obj.remove_ before the methods to remove that part.

41.4 the ifstream and ofstream objects accepts path objects.

while(!getline(ifstream object, string object).eof()) {}; puts each line of the file in the string object which can then be used wherever needed.

41.5 directory entry and directory iterator:

a directory iterator iterates over the contents of the directory.

path p{<some path>};

directory_iterator d1{p};

directory_iterator d2{};

while(d1!=d2){auto d= *d1++; d.path() ;} works, here d1 will iterate over the contents of the directory until there isn't any which will be equal to an empty iterator object. we can dereference it and call its path() method to get the full path. we can even add any of the subsequent types of a path object and that will return only that part in the path. we can also do

for(directory_entry &x: directory_iterator{path}) and x.path() similar but lesser bulky.

41.6 x.status().type() returns either filesystem::file_type::directory or regular, if it's the latter then we can call x.file_size() which returns the file size in bytes in int container.

filesystem::status(x.path()).type() is the same as above.

41.7 std::partition(<vector holding directory entries>.begin,...end(),[](auto de){ return de.is_directory();) here, partition sorts the vector by having directories at the top of the vector using the is_directory boolean method .

41.8 filesystem::status(<path>).permissions(); returns a perms object that holds the file permissions.

here are the file perms,

```
void demo_perms(fs::perms p)
{
    std::cout << ((p & fs::perms::owner_read) != fs::perms::none ? "r" : "-")
              << ((p & fs::perms::owner_write) != fs::perms::none ? "w" : "-")
              << ((p & fs::perms::owner_exec) != fs::perms::none ? "x" : "-")
              << ((p & fs::perms::group_read) != fs::perms::none ? "r" : "-")
              << ((p & fs::perms::group_write) != fs::perms::none ? "w" : "-")
              << ((p & fs::perms::group_exec) != fs::perms::none ? "x" : "-")
              << ((p & fs::perms::others_read) != fs::perms::none ? "r" : "-")
              << ((p & fs::perms::others_write) != fs::perms::none ? "w" : "-")
              << ((p & fs::perms::others_exec) != fs::perms::none ? "x" : "-")
              << '\n';
}
```

filesystem::permissions(<path>,fs::perms::<permission to change>,fs::perm_options::add/remove); adds or removes the chosen permission on the path object.

42. templates

template<typename t>

t func(t a, t b) {} ; when the func is called, the type of t will be inferred from whatever it was called with.

we can override the argument deduction by passing the type manually, like func<int>(2,3);

42.1 template<typename t>

class pp{};

pp(data type) -> pp<another data type>; explicit specialization for classes, i.e if an object of pp has the data type that matches this one then it is assumed as the other data type in compile time argument deduction.

42.2 templates are opened in compile time.

<data type> func (<data type> a, <data type> b) ; to instantiate a template, this function will be available in the compiled file even though theres no call to its template with this data type.

42.3 template<> <data type> func(<data type> a, <data type> b){} explicit specialization, meaning if the call to the func matches this data type , this function will be invoked instead of the template function.

42.4 template<int size>

void print()

{

int a= size;

}

will work, this is called non type template argument.

to call the function give it a constant value , print<3>();

42.5 using reference of an array and a non type template we can,

template<typename t,int size>
t print(t (&arr)[size]){}; called with just print(arr); yes, arr must have its size declared at compile time.

42.6 Non-type template parameters: In C++2a, even classes can be used here.
template<classname T>
...

42.7 templates inside a class are called member templates.
useful for constructors, like
class a{
template<typename t>
int b;
a(t &&a)
{
b=a;
}

};

now the class can be initialized with not only rvalue but also lvalue

42.8 we can use std::forward<type>(obj); to return the value as well as its type (lvalue or rvalue) . so if the obj is an rvalue, forward will preserve it. otherwise it is passed as an lvalue. this is called perfect forwarding.

42.9 template<typename t>
<return type> <function name>(t[] a) {}; to declare an array template.
to override the call , funcname<<datatype>[]>(<parameters>) ;

42.10 template<typename... params>
void print(params... args){}; variadic template, takes any number of arguments , only at compile time however.
however, args will have all the args and we wont be able to access any single element.
the solution is to use a typename with it ,
template<typename t, typename... params>
void print(t a, params... b)
{
do stuff with a,
print(b...); to pass a parameter pack

}

and define a print(t a) before to overload print , to accept only 1 argument and this solves recursion as well as allow individual item access.

42.11 sizeof...(b); to know size of a parameter pack
for reference of parameter pack , use <func>(params &... a);
<func(std::forward<params>(b)...); to forward a parameter pack
use rvalues instead of lvalues for template parameters, with perfect forwarding it will always be a move operation instead of a copy operation class member functions are defined using the template type along with the class name.

42.12 <return type> <classname><t,size> ::<function definition>{} ;for a class using a typename t and int size .

42.13 we can skip the template argument list only if we are declaring the function body inside the class.
template<>
class <classname><data type>{//redeclare entire class} ; explicit specialization for class template

42.14 template<typename t>
<classname> <function name><t,80>{};
we can also partially specialize a class template, meaning specify only a select few template arguments and leave the rest to ctad.

42.15 To specialize only a single member function
template<>
<return type> <classname><data type> :: <function declaration>(){}

42.16 type traits:

is_void (C++11)	checks if a type is `void` (class template)
is_null_pointer (C++14)	checks if a type is `std::nullptr_t` (class template)
is_integral (C++11)	checks if a type is an integral type (class template)
is_floating_point (C++11)	checks if a type is a floating-point type (class template)
is_array (C++11)	checks if a type is an array type (class template)
is_enum (C++11)	checks if a type is an enumeration type (class template)
is_union (C++11)	checks if a type is an union type (class template)
is_class (C++11)	checks if a type is a non-union class type (class template)
is_function (C++11)	checks if a type is a function type (class template)
is_pointer (C++11)	checks if a type is a pointer type (class template)

then there are the newer type traits:

is_default_constructible_v<t> ; here v means it returns a boolean and if a trait has _t at the end it means it's a transformation and returns the transformed object.

is_same_v<t,another t> ; returns true if both types are same.

42.17 std::decay_t<t>; returns the basic type of t, kinda like remove reference but also works for other types such as pointers.

42.18 template<typename t>
is_integral<t>::value ; will return a boolean
remove_reference<t>::type ; returns the value which is not a reference.

42.19 Folds:
template<typename... params>
<func>(params&... a)

(a+...) ; unary right fold . returns the sum of all values by adding from the rightmost value.
(...+a); unary left fold, sums from left.
(<value> + ...+a); binary left fold, this also adds the value in it.
(a+...+<value>); binary right fold.

allowed operations: used instead of '+'

```
 + - * / % ^ & | = < > << >> += -
= *= /= %= ^= &= |= <<= >>= == != <= >= && || , .* ->*

&&          - true
||          - false
,           - void()
Others      - ill-formed
```

bools work like this

(... || (a%2==0)) , here a will be each element and it will be ored with each other element in a fold.

42.20 Template Constraints: C++2a introduces 'concept's which can constraint a type that a template can have.

```
template<typename T>
concept <varname>= <bool>;
```

if this bool is true for a type, then T can be that type else not.

```
template <typename T>
concept myIntegral = std::is_integral_v<T>;
```

42.21 To enforce concepts, there are multiple syntaxes:

```
template < myIntegral T>
void f(T v);

template <typename T>
  requires myIntegral <T>
void f(T v);

template <typename T>
void f(T v) requires myIntegral <T>;

void f(myIntegral auto v);

template < myIntegral auto v>
void g();


myIntegral auto foo = ...;

auto f = []<myIntegral T> (T v) {
  // ...
};
```

and so on for lambdas.

42.22 'requires' keyword: It's used to run an expression or value which returns a bool.
requires myIntegral<T> expects myIntegral to return true for T.

concept pp= requires (T f) { f(); } //can be used to create an expr

<template T>
requires requires (T x) { x+x;}

42.23 Usecase for requires and concept:

```
struct foo {
 int foo;
};

struct bar {
 using value = int;
 value data;
};

struct baz {
 using value = int;
 value data;
};

// Using SFINAE, enable if `T` is a `baz`.
template <typename T, typename = std::enable_if_t<std::is_same_v<T, baz>>>
struct S {};

template <typename T>
using Ref = T&;

template <typename T>
concept C = requires {
                // Requirements on type `T`:
 typename T::value; // A) has an inner member named `value`
 typename S<T>;     // B) must have a valid class template specialization for `S`
 typename Ref<T>;   // C) must be a valid alias template substitution
};

template <C T>
void g(T a);

g(foo{}); // ERROR: Fails requirement A.
g(bar{}); // ERROR: Fails requirement B.
g(baz{}); // PASS.
```

SFINAE, substitution failure is not an error, is ensured through std enable if, this means if the T isn't satisfied then it would not throw an error.

42.24 Compound requirements:

```
template <typename T>
concept C = requires(T x) {
```

```
  {*x} -> typename T::inner; // the type of the expression `*x` is convertible to `T::inner`
  {x + 1} -> std::same_as<int>; // the expression `x + 1` satisfies `std::same_as<decltype((x + 1))>`
  {x * 1} -> T; // the type of the expression `x * 1` is convertible to `T`
};
```

42.25 Nested requires: requires can have more requires nested within them.

43. Typedef
    typedef <type> <varname>;
    like, typedef const char* (*afunction)(int,int);

44. alias type
    using <identifier> = <type> ;
    using afunction = const char*(*)(int,int);

    same as typedef but it supports templates too
    so
    template<typename t>
    using hello= std::vector<t>;
45. void print(int a, char b, hello c){}
    bool hello(int a , char b){}
    c is a predicate here, it is a function that returns a boolean.

46. struct bb{
    bool operator()(int a, int b){}
    }
    bb b;
    b(2,3) is equal to b.operator()(2,3); here the b is a function object. works only for user defined overloads of ().
47. system(<command>) ; to issue commands to the system, for example: “pause” will pause the program wherever it is called.

48. lambda functions:
    [<capture list>](<arguments>)<mutable><exception specification> -> <return type> {<body>}; the return type is called trailing return type .
    48.1 a lambda func is an immutable func by default, mutable allows internal variable value changes inside the function.
    48.2 to run this lambda function where it is created , use (<args>) before the ; at the end.
    48.3 to store the lambda func use,
    <return type> fn= <lambda func> ; we can use auto here.
    lambda arguments can be auto, this is called a generic lambda
    48.4 To capture an external variable, i.e to allow function to work on external variables , we use capture list.
    can only capture variables declared before it.
    for ex.
    int sum=1;
    int op=1;
    [&sum](<args>)... ; capture sum variable by reference , now we don’t need the func to be mutable.

[&sum,op](<args>)...; captures 2 variables.
[=](<args>)...; captures all variable by value. Until C++2a it also captures 'this' but from c++2a, it needs explicit capture using [=,*this].

[&](<args>)...; captures all variable by reference.
[&,var](<args>)...; capture var by value and all other by reference.
[this](<args>)...; captures this (all member variables)[x=sum](<args>)...; a generalized lambda capture, i.e x is a variable whose lifetime is only limited to the scope of the function . its type is the same as the value it copies or references.

```
template <typename... args>
auto f(args&&... arg)
{
    return [...arg=std::forward<args>(arg)]{...}
    //or return [&...arg=///] for reference
}
```
This captures parameter pack.

48.5 Inside classes if a lambda function is used , it should capture *this,this copies the this pointer into the lambda and yes it invokes the copy constructor , as capturing this normally would mean it relying on the memory allocated to the class and if it is destroyed and the lambda function is invoked it will throw.

48.6 a function inside the class can return a lambda function, that is how this can become a problem if *this is not used.

48.7 constexpr auto var= <lambda function>(<constant value arguments>) works and the return value is evaluated at compile time.

48.8 Lambdas can use templates as well:

```
auto f = []<typename T>(std::vector<T> v) {
  // ...
};
```

48.9

49. Algorithm Library: provided using STL, named after it's creator. Standard Library aka STL is also the initials for its creator.
#include<algorithm>
std::sort(<begin iterator>,<end iterator>,<predicate>) ; predicate is the function returning bool for 2 values.

49.1 std::count (<begin>,<end>,<value to search for>); returns the count of the value in the container use count_if to allow a predicate instead of a value.

49.2 std::find_if(<begin>,<end>,<predicate>) ; returns an iterator to the value in the predicate.

49.3 std::for_each(<begin>,<end>,<predicate>); gives each element to the predicate where we can do stuff with the element.

49.4 std::accumulate(...,<identity value, i.e the most basic value of the data type, 0 under most cases>); performs left fold on the elements.

49.5 std::reduce(...) works the same way but it uses a tree approach and is parallelized while accumulate is not.

49.6 most methods under algorithm support execution policy which is std::execution::seq by default meaning the non-parallelized version is used which is faster for small algos , par is faster for bigger algos.

Theres also the par_unseq policy.

49.7 std::midpoint: to find midpoint between 2 ints.
std::midpoint(1, 3); // == 2

50. #include<numeric>
std::numeric_limits<t>::max() ; returns the maximum value that can be stored in the t data type .

51. Concurrency:
#include<thread>
51.1 std::thread obj(<funcname>,<arguments>); this allows the function to be ran on another thread, obj.join() is necessary at the end unless the thread is a detached thread otherwise it crashes the program. the second thread is started at its initialization and is asked to wrap it up and join the main thread on join().
51.2 obj.joinable(); returns true if the thread isn't a detached thread , otherwise false.
51.3 #include<mutex>
std::mutex obj2; when obj2.lock() has been called , if the mutex object is locked then pauses the execution of the thread until the object isn't unlocked by the same thread that locked it , if its not locked then it allows the execution of the thread and puts other threads that call the same mutex object on wait . this synchronizes the execution so that no race conditions are created.
51.4 std::lock_guard<t> obj3(<mutex obj>); is a better version of mutex, as it locks the mutex on the mutex obj when it is called but it will automatically unlock it at the end of the scope . if the function terminates before the unlock of a traditional mutex then that mutex is locked forever, however with lock guard that is not the case .
51.5 auto id= obj.get_id(); returns an id object that represents the thread and is unique to each thread. it looks like an int, and can be directly outputted.
51.6 std::this_thread::get_id(); is the same but it only returns the id of the thread that calls it.
handle obj4(obj.native_handle()); a handle is an object that provides platform specific features for a thread, native handle returns the handle object of a thread.

51.7 #include<windows.h>
setthreaddescription(<handle obj>, <widestring>) ; sets a thread description on windows , that can be used to identify the thread.
int cores {std::thread::hardware_concurrency()); returns the number of threads available on a system .
51.8 std::this_thread::sleep_for(std::chrono::seconds(<int>)); sleeps the thread for given seconds, chrono namespace holds the time objects that the sleep for needs.
51.9 using namespace std::chrono_literals; allows for chrono objects to be used without the chrono::seconds/hours etc. '1s' is the same as std::chrono::seconds(1);

51.10 #include<future>

std::future<return type of function> obj{std::async(<launch policy>,<function>,<args>)}; creates a future object that is initialized with an async object which takes a function and processes the function asynchronously. just like thread but in a more higher-level way. if async object is defined without a future object taking it then the function will work synchronously. the future object runs the thread at the end of the scope or whenever obj.get() is called . we can check if the future object has not been run before with valid() which will return true for first time and false afterwards.

51.11 std::launch::deferred for synchronous execution, async for async exec.
obj.get(); returns whatever value was returned by the function. put it under obj.valid(); which will return true if the function has been executed completely .
obj.wait(); waits for the future object to complete the function, its like a get but doesn't return any value.
obj.wait_for(<duration object>); waits for the specified duration if the future has an async otherwise doesn't wait and then returns an enum.
std::future_status::ready , deferred and timeout ; ready means the future completed within the wait and timeout means it didn't . deferred means it didn't wait and the future had a deferred object.
obj.wait_until(<timepoint object>); waits until the time specified.

51.12 auto timepoint = std::chrono::system_clock::now(); returns a timepoint object that has the current time, we can add more time to that timepoint using duration objects.
so timepoint +=1s; works under the opened chrono literals namespace.
...now().time_since_epoch().count(); returns a long long int with time since epoch.

51.13 std::promise<data type> obk; this is a promise object, it is used in places where we need to wait for the data to come before we can use it.
for example:
void afunc(std::promise<int> &a)
{
auto f=a.get_future();
int val=f.get();
...
}
here the promise object will wait for the value to arrive with the get() method (meaning the thread will pause and wait until the value arrives), however get is a method of the future object and we need to take its future object with the get future method.

to call it we use
std::promise<int> data;
std::future<void> f1{std::async(afunc,std::ref(data))}; now until data.set_value(<value of data type>) is called the thread with afunc will wait wherever get is called in it. the promise object has to be passed by reference.

51.14 promise object also provides exception propagation through exception pointers.
basically,
data.set_exception(make_exception_ptr(<exception object>)); will pass the exception object to the promise waiting for the value to arrive and there the exception can be handled.

51.15 promise only works once however, i.e data can be set and retrieved only once.

52. common attributes: these are static asserts that check for a certain thing
[[noreturn]] indicates function does not retrun
[[nodiscard]] the functions return value must be assigned to a variable and not discarded
[[maybe_unused]] prevents compiler for issuing warning on non-used entities.
[[fall_through]] indicates deliberate fall through in a switch case statement.
[[deprecated("<msg>")]] issues an error with the msg on the function it is used.

53. #ifdef _has_include
if _has_include(<<header file>>)

```
            # include<<header file>>
            <do something extra if you want>
        #else
            # include<<some other header file>>
        #endif
#endif
```

this is a feature test macro that checks if a header file is available and includes it if it is else includes some other header file defined in the else
we can have stuff between the #if and the #else and it will be run when the if block evaluates to true, same for else block.
each if else has an endif at the end, and the ifdef has its own endif as well.

54. c++17 if else
    if(<initialize a new variable>;<the check for if >){
    use the new variable, the scope of this variable is valid for all the else blocks of this if
    }
    else if works the same
    switch(<initialization>;<variable>):
    case....
    works the same way and the initialized variable has the same scope as in if else.

55. if constexpr (<check>){}
    else if constexpr (<check>){} ;
    else {}
    these if else are checked at compile time and the block that evaluates to true is kept and all other are discarded in compilation.

56. Range based for loop with initializer: Introduced in C++2a.

```
for (auto v = std::vector{1, 2, 3}; auto& e : v) {
  std::cout << e;
}
// prints "123"
```

57. Likely and Unlikely: C++2a brings this feature, it hints the optimizer if a branch of code is more likely to be executed than the others. Unlikely is the opposite.

```
switch (n) {
case 1:
  // ...
  break;

[[likely]] case 2:  // n == 2 is considered to be arbitrarily more
  // ...           // likely than any other value of n
  break;
}

if (random > 0) [[likely]] {
  // body of if statement
  // ...
}

while (unlikely_truthy_condition) [[unlikely]] {
```

```
  // body of while statement
  // ...
}
```

58. extern and inline variables:
    extern <data type> varname; normally when we declare a variable, i.e int a, the compiler allocates memory for the variable. this is called its definition. for functions however, they may or may not be used so an extern keyword is applied to any function by default, this is why we can declare a function somewhere and define it somewhere else, but for variables that is not the case they are defined right as they are created.

    extern makes sure they are only declared, so no memory is allocated. this allows for multiple definitions of the variable to exist and is useful for global variables across multiple files.

    inline <data type> varname; makes sure the variable is declared and defined only once and can be used anywhere.
    inline is really useful in classes where header files want to initialize and use the variable then and there.

    constexpr static variables are inline by default;

59. Structured Bindings:
    class p1{
    int a{};
    char b{};
    };
    p1 p;
    auto [c,d]=p; will work, as c will have a and d will have b .
    auto &[c,d]=p; will create reference to the anonymous object that references the variables. an anonyumous object automatically references the values from the container however if we don't use & then the new objects will copy the value from this anonymous object.

    int a[3]{1,2,3};
    auto [b,c,d]=a; will work .
    auto arr=a; will work as well but it will be an int* and not int[];
    however,
    if a[3]{1,2,3] is part of a class or struct , and we create a structured binding to its object, then that will be an int[].

60. Optional:
    optional<t> obj; an object to store a value or an absence of one.
    we can use it to store a std::nullopt ; this means there is an absence of a value, and if we call boolean on obj it will be false. if the value is nullopt it doesn't allocate any memory.
    so (obj==0 && obj==false && obj==nullptr) will be true if it has a nullopt value.

    we can check if it has been initialized with obj.has_value() or just (obj) .
    to get the value inside we can use either *obj , or obj.value() or obj.value_or(<value of the same type>) . if obj doesn't have a value value or will return the value given to it.
    obj.reset(); resets the state of the optional obj.

61. Variant:

just like a union but a type-safe replacement. only 1 data type can be active at any time and the object takes as much space as the largest data type.
std::variant<<types>> obj{}; if it is default constructed like this , the first data type is initialized.

for example:
std::variant<int,string> obj{"h"}; will initialize to string type automatically.
auto v= std::get<string>(obj); will pass the string value to v.
auto v =std::get<1>(obj); will pass value of type at index 1.
bad variant access is thrown for index or data type that doesn't exist or is not the current active element.
obj.index(); returns the index of the current active type.
auto v=std::get_if<1>(&obj); v is a pointer pointing to the value at index 1 of the variant. access the value in v by *v. here, if the type is not the same as active member it returns nullptr to v.

61.1 To initialize obj:
we can use std::get<<active type>>(obj) = <value of active type>;
obj=<value of any of the types in the variant>;
obj.emplace<<any type in variant>>(<value of the type>);

62. Visitor and Visit:
std::visit(<function>,<variant object>); passes the current active value to the function.
we can use lambda functions or struct or classes with overloaded () operators.
so for obj the struct could look like
struct visitor{
void operator()(string &a){do stuff with a}
void operator()(int &a){do stuff}
}
here the corresponding to the active element operator will be automatically invoked.
for lambda we can use
[](auto &x){ using t=decay_t<decltype(x)> ;
if(is_same<t,int> {}...
}

decltype(x) returns the declaration type of x and decay transforms it to a basic type.

63. #include<any> ; this object can hold value of any type.
std::any obj{2}; will create any of type int.
we can access the value by using , std::any_cast<int>(obj); if the value isn't of the type given to any cast then it throws bad any cast exception.

std::any_cast<t&>(obj); returns a reference to the obj.
obj.has_value(); returns true if obj has a value.
obj.type(); returns the type and obj.type().name(); returns a string of the name of the type.
obj.reset() ; resets the container.
auto obj2= std::make_any<t>(<value>); creates an any object of type t with value.
auto obj2=std::any_cast<t>(&obj); returns the address and obj2 is a pointer of type t now. meaning *std::any_cast<t*>(obj2) will return the value. yes obj2 isnt an any container but any cast still works.

64. #include<string_view>

a cheaper alternative to string if modification on the string is not needed, doesn't allocate any memory.
string_view obj{"hello"}; works and we can also use
auto obj{"hello"sv} ; sv is the string view literal under the string_view_literals namespace.
obj ; returns the string as is.
obj.data(); returns a c style string meaning every character is processed and null characters,newlines etc. are processed too.
obj[<index>]; returns the character at index .
obj.at(<index>); is the same however it throws an exception if the index is invalid unlike the raw [];
obj.length(); returns the size.
obj.find("<string>"); finds the first occurrence of the substring. std::string::npos is returned if the value isn't found otherwise it returns the index of the substring's first character in the obj.
obj.remove_prefix(<value>); returns the string with the value number of characters removed from the start, note it doesn't modify the string view object.
obj.remove_suffix(<value>); does the same but for the end.
obj.substr(<index1>,<value>); returns a string_view from index1 to index+value index.

65. Class to measure running time of a program:

```
class timer {
std::chrono::steady_clock::time_point m_start ;
public:
        timer():m_start{std::chrono::steady_clock::now()} {}

void showresult(std::string_view message = "") {

        auto end = std::chrono::steady_clock::now() ;

        auto difference = end - m_start ;

        std::cout << message

                << ':'

                << std::chrono::duration_cast<std::chrono::nanoseconds>(difference).count()

                << '\n' ;

}

};
```

66. Forward declaration: In C++ we can declare objects before defining them, that's why we can do this:
Void lol(int x);
Or
Struct A;
Then later define them, i.e give them bodies. Since in C++ the compiler goes top-down, if we call a function at a part of file where it hasn't been declared yet then the compiler gives an error, even if the function is fully defined below later in the file.

67. Speed tips!
    67.1 \n is faster than endl because endl causes a flush operation.

67.2 ios_base::sync_with_stdio(0); decouples c and cpp streams and now c++ streams aren't thread safe.

67.3 Cin.tie(0); decouples cin from cout, cout is buffered by default meaning it only displays stuff when the buffer is full or demanded by flushing(pushing stuff to console) it manually.

67.4 cin.ignore(cin.rdbuf()->in_avail()); , alternative to the numeric limits being used to clear the cin buffer.

67.5 #include<iostream>

Cout.precision(<int>); sets the precision for float objects

Cout.setf(<flag's condition>,<flagname>) sets flagname with flagscondition.

For example:

Cout.setf(std::ios::hex,std::ios::basefield) sets hex as the basefield meaning couts will be in hex.

Cout.setf(std::ios::showbase) activates showbase meaning cout will append base to each output.

Cout.setf(std::ios::fixed,std::ios::floatfield) sets floatfield as fixed size, meaning if theres a precision given , output float or double object will match the given precision .

Cout.unsetf(<flagname>) disables the format flag.

For example:

Cout.unsetf(std::ios::showbase); disables showbase

Cout.unsetf(std::ios::floatfield) sets the precision to default , with lower bound unfixed and upper bound given by precision meaning if the precision is 10 and there are only 5 digits it will display all 5 digits , but if it is set to fixed 10 digits will be displayed despite the object size.

68. <string> + <int> : returns string after int index.
    Like,
    "abc" + 1
    Returns "bc".

69. Coroutines: Just like in Kotlin, another abstraction over async programming.
    For example:

```
generator<int> range(int start, int end) {
  while (start < end) {
    co_yield start;
    start++;
  }

  // Implicit co_return at the end of this function:
  // co_return;
}

for (int n : range(0, 10)) {
  std::cout << n << std::endl;
```

```
}
```

This is a generator function. co_yield implicitly uses co_await so the values are ready to be processed once requested.

```
task<void> echo(socket s) {
  for (;;) {
    auto data = co_await s.async_read();
    co_await async_write(s, data);
  }

  // Implicit co_return at the end of this function:
  // co_return;
}
```

co_await suspends execution until the expression returns a value;

```
task<int> calculate_meaning_of_life() {
  co_return 42;
}

auto meaning_of_life = calculate_meaning_of_life();
// ...
co_await meaning_of_life; // == 42
```

Here we lazily use co_await.

Coroutines aren't in standard library yet.

70. Synchronized Buffered Outputstream: Buffers output stream which ensures the output will appear synchronized.
    ```
    std::osyncstream{std::cout} << "The value of x is:" << x << std::endl;
    ```
71. span: Just like in other langs, a span is a view, which basically means they hold references to the values of a sequence container rather than copying/moving them.
    ```
    void f(std::span<int> ints) {
      std::for_each(ints.begin(), ints.end(), [](auto i) {
        // ...
      });
    }

    std::vector<int> v = {1, 2, 3};
    f(v);
    std::array<int, 3> a = {1, 2, 3};
    f(a);
    ```
72. Math constants:
    ```
    std::numbers::pi; // 3.14159...
    std::numbers::e; // 2.71828...
    ```
73. Compile time evaluation:
    ```
    constexpr bool is_compile_time() {
      return std::is_constant_evaluated();
    }

    constexpr bool a = is_compile_time(); // true
    ```

bool b = is_compile_time(); // false

74. to array: C++2a feature.
    std::to_array("foo"); // returns `std::array<char, 4>`
    std::to_array<int>({1, 2, 3}); // returns `std::array<int, 3>`

    int a[] = {1, 2, 3};
    std::to_array(a); // returns `std::array<int, 3>`
75.