C++ STL

Standard library (Functions etc.)

Nathan Warner

Northern Illinois University

Computer Science
Northern Illinois University
United States

Contents

1 C++ Strings (<string>)

Interlude. Before we begin with the C++ string methods, there are two things to know.

- **size_t**: an unsigned integral type, and it's designed to be able to represent the size of any object in bytes
- **npos**: (Constant) It's the largest possible value representable by the size_type of std::string

Note:-

size_t is used as the return value for methods such as "find" to indicate unsuccessful

1.1 Element Access

- **at(r:size_t pos):↦** `char&` — Returns a reference to the character at the specified position. Store return value in char&
- **str.front():↦** `char&` — returns reference to the first character.
- **str.back():↦** `char&` — returns a reference to the last character.
- **str.c_str():→** const char* pointing to the null-terminated character array.
- **str.data():→** const char* pointing to the underlying character array.

1.2 Capacity

- **str.length():↦** `size_t` — Returns the number of characters in the string
- **str.size():↦** `size_t` — Returns the number of characters in the string
- **str.empty():↦** `bool` — Returns true if the string is empty, false otherwise
- **resize(r:size_t n, o:char c):↦** `void` — Resizes the string to contain n characters.
- **capacity():↦** `size_t` — Returns the size of the storage space currently allocated
- **reserve(size_t new_cap):↦** `void` — Reserves storage (increases capacity).
- **max_size():↦** `size_t` — Returns the maximum number of characters the string can hold
- **shrink_to_fit():↦** `void` — Reduces memory usage by freeing unused memory

1.3 Modifiers

- **append(string str):**↦ `std`::string& mutated_string — String to append. (Has other overloads.) (Doesnt need to be saved in T&)
- **push_back(char c):**↦ `void` — Character to append.
- **assign(string str, o:start,o:stop):**↦ `std`::string& mutated_string — used to replace the current content of the string with a new set of characters. (Has other overloads.)
- **insert(size_t pos, string str):**↦ `std`::string& mutated_str — Position and string to insert. (Has other overloads.)
- **replace(size_t pos, size_t len, string str):**↦ `mutated`_string — Position, length, and string for replacement. (Has other overloads.)
- **swap(string str):**↦ `void` — String to swap with.
- **pop_back():** ↦ `void` — Removes the last character in the string

1.4 String Operations

- **substr(pos, o:len):**↦ `string` — Generate a substr from a string
- **copy(char[] dest, o:len, pos):**↦ `size_t` (number of characters that were copied) send contents of string to some character array
- **find(substr, o:pos):**↦ `size_t`=npos find a substr from within a string
- **rfind(substr, o:pos):**↦ `size_t`=npos find a substr form withing as string, starting search from the end of the string
- **find_first_of(substr (or char), o:pos)**↦ `size_t`=npos Find character in string (public member function)
- **find_last_of(substr (or char), o:pos)**↦ `size_t`=npos Find character in string from the end (public member function)
- **find_first_not_of(substr (or char), o:pos)**↦ `size_t`=npos Find absence of character in string (public member function)
- **find_last_not_of(substr (or char), o:pos)**↦ `size_t`=npos Find non-matching character in string from the end (public member function)

1.5 Comparison

- **compare(o:pos, o:len, str):**↦ `unsigned` integral
  - 0: they compare equal
  - <0: Either the value of the first character that does not match is lower in the compared string, or all compared characters match but the compared string is shorter.
  - >0: Either the value of the first character that does not match is greater in the compared string, or all compared characters match but the compared string is longer.

1.6 Conversions (These are functions)

- **stoi(str, o:idx, o:base):**↦ `int` (idx is a pointer to a size_t object)
- **stol(str, o:idx, o:base):**↦ `long`
- **stoul(str, o:idx, o:base):**↦ `unsigned` long
- **stoll(str, o:idx, o:base):**↦ `long` long
- **stoull(str, o:idx, o:base):**↦ `unsigned` long long
- **stof(str, o:idx):**↦ `float`
- **stod(str, o:idx):**↦ `double`
- **stold(str, o:idx):**↦ `long` double

2 C-strings (<cstring>)

2.1 Manipulation

- **strncpy(char[] dest, char[] src, size_t n):**↦ `const` char* — Copy up to n characters from the string src to dest.
- **strcpy(char[] dest, char[] src)**↦ `const` char* —- Copies the C string pointed by source into the array pointed by destination
- **strncat(char[] dest, char[] src, size_t n):**↦ `const` char* — Append up to n characters from the string src onto the end of dest.
- **strcat(char[] dest, char[] src)**↦ `const` char* — Appends a copy of the source string to the destination string.

  Note: cpy functions return a const char* comprised of the characters from the src array that were used.

2.2 Examination

- **strnlen(char[] src, size_t maxlen)**↦ `size_t` — Return the length of the string (not including the null terminator).
- **strlen(char[] src):**↦ `size_t` — Return the length of the string (not including the null terminator).
- **strncmp(char[] src1, char[] src2, size_t n):**↦ `uint` (value varies) — Compare up to n characters of two strings.
- **strcmp(char[] src1, char[] src2)**↦ `size_t` (value varires) — compare two strings

2.3 Searching

- **strchr(char[] src, char c):**↦ `char*` —- get pointer to the first occurrence of character c in the string s, or nullptr if c is not found.
- **strrchr(char[] src, char c):**↦ `char*` — get pointer to the last occurrence of character c in the string s.
- **strstr(char[] src, char[] str):**↦ `const` char* — Return a pointer to the first occurrence of the substring
- **strspn(char[] src, char[] charset):**↦ `size_t` — Returns the length of the initial portion of str1 which consists only of characters that are part of str2.
- **strcspn(char[] src, char[] charset):**↦ `size_t` — Get position of first character found from charset
- **strpbrk(char[] src, char[] charset):**↦ `const` char* — Return string consisting of first match form character set in string onward
- **strtok(char[] src, const char* delims):**↦ `char*`

2.4 Error

- **strerror(errno)**↦ `const` char* — Get pointer to error message string (we are literally passing in the defined variable *errno*)

```
std::ifstream inf("file.txt");
if (inf.fail()) { cout << strerrror(errno); } // No such file in directory
(Because that is what the error string is set to after inf.fail())
```

we also have control over the error we can pass in, errors that are defined in <cerrno>. Rather than leaving it up to errno. We have:

E2BIG (C++11) Argument list too long (macro constant)
EACCES (C++11) Permission denied (macro constant)
EADDRINUSE (C++11) Address in use (macro constant)
EADDRNOTAVAIL (C++11) Address not available (macro constant)
EAFNOSUPPORT (C++11) Address family not supported (macro constant)
EAGAIN (C++11) Resource unavailable, try again (macro constant)
EALREADY (C++11) Connection already in progress (macro constant)
EBADF (C++11) Bad file descriptor (macro constant)
EBADMSG (C++11) Bad message (macro constant)
EBUSY (C++11) Device or resource busy (macro constant)
ECANCELED (C++11) Operation canceled (macro constant)
ECHILD (C++11) No child processes (macro constant)
ECONNABORTED (C++11) Connection aborted (macro constant)
ECONNREFUSED (C++11) Connection refused (macro constant)
ECONNRESET (C++11) Connection reset (macro constant)
EDEADLK (C++11) Resource deadlock would occur (macro constant)
EDESTADDRREQ (C++11) Destination address required (macro constant)
EDOM Mathematics argument out of domain of function (macro constant)
EEXIST (C++11) File exists (macro constant)
EFAULT (C++11) Bad address (macro constant)
EFBIG (C++11) File too large (macro constant)
EHOSTUNREACH (C++11) Host is unreachable (macro constant)
EIDRM (C++11) Identifier removed (macro constant)
EILSEQ Illegal byte sequence (macro constant)
EINPROGRESS (C++11) Operation in progress (macro constant)
EINTR (C++11) Interrupted function (macro constant)
EINVAL (C++11) Invalid argument (macro constant)
EIO (C++11) I/O error (macro constant)
EISCONN (C++11) Socket is connected (macro constant)
EISDIR (C++11) Is a directory (macro constant)
ELOOP (C++11) Too many levels of symbolic links (macro constant)

EMFILE (C++11) File descriptor value too large (macro constant)

EMLINK (C++11) Too many links (macro constant)

EMSGSIZE (C++11) Message too large (macro constant)

ENAMETOOLONG (C++11) Filename too long (macro constant)

ENETDOWN (C++11) Network is down (macro constant)

ENETRESET (C++11) Connection aborted by network (macro constant)

ENETUNREACH (C++11) Network unreachable (macro constant)

ENFILE (C++11) Too many files open in system (macro constant)

ENOBUFS (C++11) No buffer space available (macro constant)

ENODATA (C++11)(deprecated in C++23) No message is available on the STREAM head read queue (macro constant)

ENODEV (C++11) No such device (macro constant)

ENOENT (C++11) No such file or directory (macro constant)

ENOEXEC (C++11) Executable file format error (macro constant)

ENOLCK (C++11) No locks available (macro constant)

ENOLINK (C++11) Link has been severed (macro constant)

ENOMEM (C++11) Not enough space (macro constant)

ENOMSG (C++11) No message of the desired type (macro constant)

ENOPROTOOPT (C++11) Protocol not available (macro constant)

ENOSPC (C++11) No space left on device (macro constant)

ENOSR (C++11)(deprecated in C++23) No STREAM resources (macro constant)

ENOSTR (C++11)(deprecated in C++23) Not a STREAM (macro constant)

ENOSYS (C++11) Function not supported (macro constant)

ENOTCONN (C++11) The socket is not connected (macro constant)

ENOTDIR (C++11) Not a directory (macro constant)

ENOTEMPTY (C++11) Directory not empty (macro constant)

ENOTRECOVERABLE (C++11) State not recoverable (macro constant)

ENOTSOCK (C++11) Not a socket (macro constant)

ENOTSUP (C++11) Not supported (macro constant)

ENOTTY (C++11) Inappropriate I/O control operation (macro constant)

ENXIO (C++11) No such device or address (macro constant)

EOPNOTSUPP (C++11) Operation not supported on socket (macro constant)

EOVERFLOW (C++11) Value too large to be stored in data type (macro constant)

EOWNERDEAD (C++11) Previous owner died (macro constant)

EPERM (C++11) Operation not permitted (macro constant)

EPIPE (C++11) Broken pipe (macro constant)

EPROTO (C++11) Protocol error (macro constant)

EPROTONOSUPPORT (C++11) Protocol not supported (macro constant)

EPROTOTYPE (C++11) Protocol wrong type for socket (macro constant)

ERANGE Result too large (macro constant)

EROFS (C++11) Read-only file system (macro constant)

ESPIPE (C++11) Invalid seek (macro constant)

ESRCH (C++11) No such process (macro constant)

ETIME (C++11)(deprecated in C++23) Stream ioctl() timeout (macro constant)

ETIMEDOUT (C++11) Connection timed out (macro constant)

ETXTBSY (C++11) Text file busy (macro constant)

EWOULDBLOCK (C++11) Operation would block (macro constant)

EXDEV (C++11) Cross-device link (macro constant)

2.5 Conversion Found in <cstdlib>

- **atoi(char[])** - Converts a C-string to an int.
- **atol(char[])** - Converts a C-string to a long.
- **atoll(char[])** - Converts a C-string to a long long.
- **atof(char[])** - Converts a C-string to a double.
- **strtol(char[], endptr, base)** - Converts a C-string to a long int, with error checking and more flexibility with base representations.
- **strtoul(char[], endptr, base)** - Converts a C-string to an unsigned long int.
- **strtoll(char[], endptr, base)** - Converts a C-string to a long long int.
- **strtoull(char[], endptr, base)** - Converts a C-string to an unsigned long long int.
- **strtof(char[], endptr)** - Converts a C-string to a float.
- **strtod(char[], endptr)** - Converts a C-string to a double.
- **strtold(char[], endptr)** - Converts a C-string to a long double.

Note: Consider the codeblock

```
const char* a = "12.322string";
char* ptr;

double val = strtod(a, &ptr); // ptr will hold the value "string"
```

In this example, you can see that we are making use of the endptr optional parameter, *endptr* is the address to a char pointer. It allows us to send all the string data that is not able to be converted to the requested data type . It allows us to send all the string data that is not able to be converted to the requested data type to this pointer. Only data AFTER the converted data will be sent.

3 Numeric to string conversions <string>

- **to_string(n):**↦ `string` — converts numeric type to c++ string

4 Characters (<cctype>)

4.1 Character Classification

- **isalpha(char c):** Checks if the character is an alphabet (either uppercase or lowercase).
- **isdigit(char c):** Checks if the character is a digit (0-9).
- **isalnum(char c):** Checks if the character is either an alphabet or a digit.
- **isspace(char c):** Checks if the character is a whitespace character (like space, tab, newline, etc.).
- **isupper(char c):** Checks if the character is uppercase.
- **islower(char c):** Checks if the character is lowercase.
- **ispunct(char c):** Checks if the character is a punctuation character.
- **isprint(char c):** Checks if the character is printable.
- **iscntrl(char c):** Checks if the character is a control character.

4.2 Character Conversion:

- **toupper(char c):** Converts the character to uppercase (if it's lowercase).
- **tolower(char c):** Converts the character to lowercase (if it's uppercase).

5 C++ Arrays <array>

Note: C++ Arrays are defined

```
std::array<T,N> name;
```

- **at(size_t)**↦ `T&`: Accesses the element at the specified *position* with bounds checking.
- **front()**↦ `T&`: Returns a reference to the first element.
- **back()**↦ `T&`: Returns a reference to the last element.
- **data()**↦ `T*`: Returns a direct pointer to the first element in the array, usefull if we want to use our std::array as if it were a c-style array
- **empty()**↦ `bool`: Checks if the container has no elements.
- **size()**↦ `size_t`: Returns the number of elements in the container.
- **max_size()**↦ `size_t`: Returns the maximum number of elements the container can hold (same as size).
- **fill(value)**↦ `void`: Fills the array with the specified *value*.
- **swap(other)**↦ `void`: Swaps the contents of the array with those of *other*.
- **begin()**↦ `Iterator`: Returns an iterator pointing to the first element.
- **cbegin()**↦ `Const_Iterator`: Returns a const iterator pointing to the first element.
- **end()**↦ `Iterator`: Returns an iterator pointing to one-past-the-last element.
- **cend()**↦ `Const_Iterator`: Returns a const iterator pointing to one-past-the-last element.
- **rbegin()**↦ `Reverse_Iterator`: Returns a reverse iterator pointing to the last element.
- **crbegin()**↦ `Const_Reverse_Iterator`: Returns a const reverse iterator pointing to the last element.
- **rend()**↦ `Reverse_Iterator`: Returns a reverse iterator pointing to one-past-the-first element.
- **crend()**↦ `Const_Reverse_Iterator`: Returns a const reverse iterator pointing to one-past-the-first element.

6 Algorithms <algorithm>

Prereq: Vocab

- An **InputIterator** is a type of iterator that can be used to read data from a sequence of elements. It supports operations like incrementing (to move to the next element in the sequence), dereferencing (to access the value of the element it currently points to), and comparing with other iterators (to check for the end of the sequence). Input iterators are the least powerful, but most widely applicable kind of iterator, as they only require single-pass, read-only access.
- An **ForwardIterator** is a type of iterator that has all the capabilities of an InputIterator but with some additional properties:
  - It can be incremented multiple times and will always give the same sequence of results (multi-pass guarantee).
  - It supports both it++ and ++it operations with the same effect.
  - Two dereferenced copies of a ForwardIterator are guaranteed to reference the same element (if neither is modified).

  ForwardIterator is used in contexts where an algorithm might need to pass over a range of elements multiple times.
- A **UnaryPredicate** is a function or a function object that takes a single argument and returns a bool. It is used to test whether a certain condition is true for the elements in a sequence. The predicate can be a regular function, a lambda expression, or an object of a class that overloads the function call operator.
- A **UnaryOperator** refers to a function or a function object that takes a single argument and returns some value.
- An **OutputIterator** is a type of iterator that can be used to write to a sequence of elements. It's a concept from the C++ Standard Library that defines the requirements for an iterator that can be used to output or write data to a container.

  Note: A standard iterator can be used in place of a outputiterator, however, here is how we can use a outputiterator to write contents of an array to the output buffer 1br

```
0 int arr[5] = {1,2,3,4,5};
1 std::copy(std::begin(arr), std::end(arr),
  ↪  std::ostream_iterator<int>(cout, " "));
```

std::pair type: std::pair is a class template that provides a way to store two heterogeneous objects as a single unit. The way we access the elements is with .first and .second

6.1 Non-Modifying

- **all_of(InputIterator, InputIterator, UnaryPredicate)** ↦ `bool`: Returns true if the predicate is true for all elements in the given range.
- **any_of(InputIterator, InputIterator, UnaryPredicate)** ↦ `bool`: Returns true if the predicate is true for any element in the range.
- **none_of(InputIterator, InputIterator, UnaryPredicate)** ↦ `bool`: Returns true if the predicate is false for all elements in the range.

- **for_each(InputIterator, InputIterator, Function)** ↦ `Function`: Applies a function to each element in the range.
- **count(InputIterator, InputIterator, const T&)** ↦ `size_t`: Counts elements equal to the specified value.
- **count_if(InputIterator, InputIterator, UnaryPredicate)** ↦ `size_t`: Counts elements for which the predicate is true.
- **mismatch(InputIterator1, InputIterator1, InputIterator2)** ↦ `pair<InputIterator1, InputIterator2>`: Finds the first position where two ranges differ.
- **find(InputIterator, InputIterator, const T&)** ↦ `InputIterator`: Finds the first element equal to the specified value.
- **find_if(InputIterator, InputIterator, UnaryPredicate)** ↦ `InputIterator`: Finds the first element for which the predicate is true.
- **find_if_not(InputIterator, InputIterator, UnaryPredicate)** ↦ `InputIterator`: Finds the first element for which the predicate is false.
- **find_end(InputIterator, InputIterator, InputIterator, InputIterator)** ↦ `InputIterator`: Finds the last occurrence of a subsequence.
- **find_first_of(InputIterator, InputIterator, InputIterator, InputIterator)** ↦ `InputIterator`: Finds the first element that matches any element in another range.
- **adjacent_find(InputIterator, InputIterator, o:UnaryPredicate)** ↦ `InputIterator`: Finds the first two adjacent items that are equal (or satisfy a given predicate).
- **search(InputIterator, InputIterator, InputIterator, InputIterator)** ↦ `InputIterator`: Searches for the first occurrence of a subsequence.
- **search_n(InputIterator, InputIterator, size_t, const T&)** ↦ `InputIterator`: Searches for a sequence of repeated elements.

> **Note:**
>
> The for_each function returns the function that was used to map each element.

- **copy(InputIterator, InputIterator, OutputIterator)** ↦ `OutputIterator`: Copies a range of elements to a new location.
- **copy_if(InputIterator, InputIterator, OutputIterator, UnaryPredicate)** ↦ `OutputIterator`: Copies elements satisfying a condition to a new location.
- **copy_n(InputIterator, Size, OutputIterator)** ↦ `OutputIterator`: Copies a number of elements to a new location.
- **copy_backward(BidirectionalIterator1, BidirectionalIterator1, BidirectionalIterator2)** ↦ `BidirectionalIterator2`: Copies elements in reverse order.
- **move(InputIterator, InputIterator, OutputIterator)** ↦ `OutputIterator`: Moves a range of elements to a new location.
- **move_backward(BidirectionalIterator1, BidirectionalIterator1, BidirectionalIterator2)** ↦ `BidirectionalIterator2`: Moves elements in reverse order.
- **fill(ForwardIterator, ForwardIterator, const T&)** ↦ `void`: Assigns a value to all elements in a range.

- **fill_n(OutputIterator, Size, const T&)** ↦ `OutputIterator`: Assigns a value to a number of elements.
- **transform(InputIterator, InputIterator, OutputIterator, UnaryOperation)** ↦ `OutputIterator`: Applies a function to a range of elements.
- **generate(ForwardIterator, ForwardIterator, Generator)** ↦ `void`: Fills a range with the results of successive function calls.
- **generate_n(OutputIterator, Size, Generator)** ↦ `OutputIterator`: Fills a range with N results of successive function calls.
- **remove(InputIterator, InputIterator, const T&)** ↦ `InputIterator`: Removes elements equal to a value.
- **remove_if(InputIterator, InputIterator, UnaryPredicate)** ↦ `InputIterator`: Removes elements satisfying a condition.
- **remove_copy(InputIterator, InputIterator, OutputIterator, const T&)** ↦ `OutputIterator`: Copies elements not equal to a value.
- **remove_copy_if(InputIterator, InputIterator, OutputIterator, UnaryPredicate)** ↦ `OutputIterator`: Copies elements not satisfying a condition.
- **replace(InputIterator, InputIterator, const T&, const T&)** ↦ `void`: Replaces all values equal to a specified value.
- **replace_if(InputIterator, InputIterator, UnaryPredicate, const T&)** ↦ `void`: Replaces values satisfying a condition.
- **replace_copy(InputIterator, InputIterator, OutputIterator, const T&, const T&)** ↦ `OutputIterator`: Copies and replaces values.
- **replace_copy_if(InputIterator, InputIterator, OutputIterator, UnaryPredicate, const T&)** ↦ `OutputIterator`: Copies and replaces values based on a condition.
- **swap(T&, T&)** ↦ `void`: Swaps the values of two objects.
- **swap_ranges(ForwardIterator1, ForwardIterator1, ForwardIterator2)** ↦ `ForwardIterator2`: Swaps two ranges of elements.
- **iter_swap(ForwardIterator1, ForwardIterator2)** ↦ `void`: Swaps elements pointed to by two iterators.
- **reverse(BidirectionalIterator, BidirectionalIterator)** ↦ `void`: Reverses the order of elements in a range.
- **reverse_copy(BidirectionalIterator, BidirectionalIterator, OutputIterator)** ↦ `OutputIterator`: Creates a reversed copy of a range.
- **rotate(ForwardIterator, ForwardIterator, ForwardIterator)** ↦ `ForwardIterator`: Rotates the order of elements in a range.
- **rotate_copy(ForwardIterator, ForwardIterator, ForwardIterator, OutputIterator)** ↦ `OutputIterator`: Copies and rotates a range of elements.
- **shift_left(ForwardIterator, ForwardIterator, Size)** ↦ `ForwardIterator`: Shifts elements in a range to the left.
- **shift_right(ForwardIterator, ForwardIterator, Size)** ↦ `ForwardIterator`: Shifts elements in a range to the right.

- **random_shuffle(RandomAccessIterator, RandomAccessIterator)** ↦ `void`: Randomly re-orders elements in a range.
- **shuffle(RandomAccessIterator, RandomAccessIterator, URNG&)** ↦ `void`: Randomly re-orders elements in a range using a generator.
- **sample(InputIterator, InputIterator, OutputIterator, Size, URNG&)** ↦ `OutputIterator`: Selects N random elements from a sequence.
- **unique(ForwardIterator, ForwardIterator)** ↦ `ForwardIterator`: Removes consecutive duplicate elements in a range.
- **unique_copy(InputIterator, InputIterator, OutputIterator)** ↦ `OutputIterator`: Creates a copy of a range without consecutive duplicates.
- **partition(BidirectionalIterator, BidirectionalIterator, UnaryPredicate)**↦ `Iterator`: Partition range in two
- **partition_copy(BidirectionalIterator, BidirectionalIterator, UutputIterator)**↦ `OutputIterator`: Partition range in two
- **stable_partition(BidirectionalIterator, BidirectionalIterator, UnaryPredicate)**↦ `Iterator`: Partition range in two - stable ordering

6.2 Using lambdas for these functions

```
0  int arr[]  = {1,2,3};
1  std::for_each(arr, arr+3, [](int& a) -> int { return ++x; });
```

6.3 Getting position of element from returned iterator

Concept 1: If we subtract the iterator (which is the address of the array element) from the beginning of the array, we get the distance between the two pointers (so the index position of the iterator value)

```
0  idx = it - std::begin(array);
```

□

6.4 Using shuffle and sample

For these functions we need two components, a seed and a random engine

```
0  #include <chrono>
1  #include <random>
2
3  unsigned seed
   ↪  std::chrono::system_clock::now().time_since_epoch().count();
4  std::default_random_engine engine(seed);
```

We then use the engine variable in place of URNG&

6.5 Sorting operations

- **is_sorted(ForwardIterator, ForwardIterator)** ↦ `bool`: Checks whether a range is sorted into ascending order.
- **is_sorted_until(ForwardIterator, ForwardIterator)** ↦ `ForwardIterator`: Finds the largest sorted subrange.
- **sort(RandomAccessIterator, RandomAccessIterator)** ↦ `void`: Sorts a range into ascending order.
- **partial_sort(RandomAccessIterator, RandomAccessIterator, RandomAccessIterator, comp)** ↦ `void`: Sorts the first N elements of a range.
- **partial_sort_copy(InputIterator, InputIterator, RandomAccessIterator, RandomAccessIterator, comp)** ↦ `RandomAccessIterator`: Copies and partially sorts a range of elements.
- **stable_sort(RandomAccessIterator, RandomAccessIterator)** ↦ `void`: Sorts a range of elements while preserving order between equal elements.
- **nth_element(RandomAccessIterator, RandomAccessIterator, RandomAccessIterator)** ↦ `void`: Partially sorts the given range making sure that it is partitioned by the given element.

6.6 Binary search operations (on sorted ranges)

- **lower_bound(ForwardIterator, ForwardIterator, const T&)** ↦ `ForwardIterator`: Returns an iterator to the first element in the range [first, last) that is not less than (i.e., greater or equal to) the given value.
- **upper_bound(ForwardIterator, ForwardIterator, const T&)** ↦ `ForwardIterator`: Returns an iterator to the first element in the range [first, last) that is greater than the given value.
- **binary_search(ForwardIterator, ForwardIterator, const T&)** ↦ `bool`: Determines if an element equal to the given value exists within the range [first, last).
- **equal_range(ForwardIterator, ForwardIterator, const T&)** ↦ `pair<ForwardIterator, ForwardIterator>`: Returns a range containing all elements equivalent to the given value in the range [first, last).

6.7 Minimum/maximum operations

- **max(const T&, const T&)** ↦ `T`: Returns the greater of the two given values.
- **max_element(ForwardIterator, ForwardIterator)** ↦ `ForwardIterator`: Returns an iterator to the largest element in the range [first, last).
- **min(const T&, const T&)** ↦ `T`: Returns the smaller of the two given values.
- **min_element(ForwardIterator, ForwardIterator)** ↦ `ForwardIterator`: Returns an iterator to the smallest element in the range [first, last).

- **minmax(const T&, const T&)** ↦ `pair<T, T>`: Returns a pair consisting of the smaller and larger of the two elements.
- **minmax_element(ForwardIterator, ForwardIterator)** ↦ `pair<ForwardIterator, ForwardIterator>`: Returns a pair of iterators to the smallest and the largest elements in the range [first, last).
- **clamp(const T&, const T&, const T&)** ↦ `T`: Clamps a value between a pair of boundary values. If the value is less than the lower bound, it returns the lower bound. If it's greater than the upper bound, it returns the upper bound. Otherwise, it returns the value itself.

6.8 Other

- **merge(InputIterator1, InputIterator1, InputIterator2, InputIterator2, OutputIterator)** ↦ `OutputIterator`: Merges two sorted ranges [first1, last1) and [first2, last2). The resulting range is also sorted.
- **inplace_merge(BidirectionalIterator, BidirectionalIterator, BidirectionalIterator)** ↦ `void`: Merges two consecutive ordered ranges [first, middle) and [middle, last) into a single ordered range [first, last).
- **equal(InputIterator1, InputIterator1, InputIterator2)** ↦ `bool`: Determines if the range [InputIterator1, InputIterator1) is equal to the range starting at Inputiterator2.
- **is_permutation(ForwardIterator1, ForwardIterator1, ForwardIterator2)** ↦ `bool` (C++11): Determines if the range [first1, last1) is a permutation of the range starting at first2.

6.9 Set Operations

- **includes(InputIterator1, InputIterator1, InputIterator2, InputIterator2)** ↦ `bool`: Returns true if the range [first2, last2) is a subsequence of the range [first1, last1).
- **set_difference(InputIterator1, InputIterator1, InputIterator2, InputIterator2, OutputIterator)** ↦ `OutputIterator`: Computes the difference of two sets. The result contains elements that are in the first set but not in the second.
- **set_intersection(InputIterator1, InputIterator1, InputIterator2, InputIterator2, OutputIterator)** ↦ `OutputIterator`: Computes the intersection of two sets. The result contains elements that are present in both sets.
- **set_symmetric_difference(InputIterator1, InputIterator1, InputIterator2, InputIterator2, OutputIterator)** ↦ `OutputIterator`: Computes the symmetric difference between two sets. The result contains elements that are in either of the sets but not in their intersection.
- **set_union(InputIterator1, InputIterator1, InputIterator2, InputIterator2, OutputIterator)** ↦ `OutputIterator`: Computes the union of two sets. The result contains all elements that are present in either of the sets, without duplicates.

7 Iterators <iterator>

- **advance(Iterator&, Distance)**↦ `void`: Advances the iterator by the given distance.
- **distance(InputIterator, InputIterator)** ↦ `typename iterator_traits<InputIterator>::difference_t` Returns the number of steps between two iterators.
- **begin(Container&)** ↦ `Iterator`: Returns an iterator pointing to the first element of the container.
- **end(Container&)** ↦ `Iterator`: Returns an iterator pointing to the past-the-end element of the container.
- **prev(BidirectionalIterator)** ↦ `BidirectionalIterator`: Returns an iterator pointing to the element immediately before the given iterator.
- **next(ForwardIterator)** ↦ `ForwardIterator`: Returns an iterator pointing to the element immediately after the given iterator.

7.1 Iterator generators:

- **back_inserter(Container&)** ↦ `BackInsertIterator`: Constructs a back-insert iterator that appends new elements to the end of the specified container.
- **front_inserter(Container&)** ↦ `FrontInsertIterator`: Constructs a front-insert iterator that prepends new elements to the beginning of the specified container.
- **inserter(Container&, Iterator)** ↦ `InsertIterator`: Constructs an insert iterator for inserting new elements into the container at the position specified by the given iterator.
- **make_move_iterator(Iterator)** ↦ `MoveIterator`: Converts a regular iterator into a move iterator, causing the elements to be moved rather than copied.

8 cstdarg (For variadic functions)

8.1 Types

- **va_list** Type to hold information about variable arguments

8.2 Functions

- **va_start** Initialize a variable argument list
- **va_arg** Retrieve next argument
- **va_end** End using variable argument list
- **va_copy** Copy variable argument list

9 Functional

Memory stuff from Cstdlib and <cstring>

- **calloc(size_t num, size_t size)** ↦ `void*`: Allocate and zero-initialize an array. Returns a pointer to the allocated memory, or NULL if the allocation fails.
- **free(void* ptr)** ↦ `void`: Deallocate memory block. Frees the memory space pointed to by `ptr`, which must have been returned by a previous call to `malloc`, `calloc`, or `realloc`. Otherwise, or if `free(ptr)` is called more than once, undefined behavior occurs.
- **malloc(size_t size)** ↦ `void*`: Allocate a memory block. Returns a pointer to the allocated memory, or NULL if the allocation fails.
- **realloc(void* ptr, size_t new_size)** ↦ `void*`: Reallocate a memory block. Changes the size of the memory block pointed to by `ptr` to `new_size`. The function may move the memory block to a new location, in which case the new location is returned. Returns NULL if the allocation fails.
- **memchr(const void* str, int c, size_t n)** ↦ `void*`: Searches for the first occurrence of the character `c` (an unsigned char) in the first `n` bytes of the string pointed to by `str`. Returns a pointer to the matching byte or NULL if the character does not occur in the given memory area.
- **memcmp(const void* str1, const void* str2, size_t n)** ↦ `int`: Compares the first `n` bytes of the memory areas `str1` and `str2`. Returns an integer less than, equal to, or greater than zero if `str1` is found, respectively, to be less than, to match, or be greater than `str2`.
- **memset(void* str, int c, size_t n)** ↦ `void*`: Fills the first `n` bytes of the memory area pointed to by `str` with the constant byte `c`. Returns a pointer to the memory area `str`.
- **memcpy(void* dest, const void* src, size_t n)** ↦ `void*`: Copies `n` bytes from memory area `src` to memory area `dest`. The memory areas must not overlap. Returns a pointer to `dest`.
- **memmove(void* dest, const void* src, size_t n)** ↦ `void*`: Moves `n` bytes from memory area `src` to memory area `dest`. The memory areas may overlap. Returns a pointer to `dest`.

Exception Classes <std::except>

11.1 Objects

- **logic_error:** exception class to indicate violations of logical preconditions or class invariants
- **invalid_argument:** exception class to report invalid arguments
- **domain_error:** exception class to report domain errors
- **length_error:** exception class to report attempts to exceed maximum allowed size
- **out_of_range:** exception class to report arguments outside of expected range
- **runtime_error:** exception class to indicate conditions only detectable at run time
- **range_error:** exception class to report range errors in internal computations
- **overflow_error:** exception class to report arithmetic overflows
- **underflow_error:** exception class to report arithmetic underflows

Note:-

All of these constructors have four overloads

1. **std::string**
2. **C-String**
3. **Other of same type**
4. **Other of same type (noexcept version)**

11.2 Methods

- **what()**

c-style IO operations <cstdio>

12.1 Operations on files

- **remove(const char* filename)** ↦ `int`: Removes the file specified by `filename`. Returns zero on success, or nonzero if an error occurs.
- **rename(const char* oldname, const char* newname)** ↦ `int`: Renames the file or directory specified by `oldname` to `newname`. Returns zero on success, or nonzero if an error occurs.
- **tmpfile(void)** ↦ `FILE*`: Creates a temporary file in binary update mode (`wb+`) that is automatically deleted when it is closed or the program terminates. Returns a pointer to the `FILE` object that represents the temporary file.
- **tmpnam(char* str)** ↦ `char*`: Generates and returns a unique temporary filename. If `str` is a null pointer, the temporary filename is stored in a static internal array and returned. If `str` is not a null pointer, the temporary filename is stored in `str`. Returns a pointer to the temporary filename.

12.2 File Access

- **fclose(FILE* stream)** ↦ `int`: Closes the file associated with the stream. Returns zero on success, or EOF if an error occurs.
- **fflush(FILE* stream)** ↦ `int`: Flushes the output buffer of a stream. If `stream` is NULL, flushes all output buffers. Returns zero on success, or EOF if an error occurs.
- **fopen(const char* filename, const char* mode)** ↦ `FILE*`: Opens the file specified by `filename` using the mode specified by `mode`. Returns a pointer to the `FILE` object associated with the opened file, or NULL if the file cannot be opened.
- **freopen(const char* filename, const char* mode, FILE* stream)** ↦ `FILE*`: Frees the stream and reopens the file specified by `filename` using the mode specified by `mode`. Returns a pointer to the `FILE` object associated with the reopened file, or NULL if the file cannot be opened.
- **setbuf(FILE* stream, char* buf)** ↦ `void`: Sets the buffer to be used by the stream for I/O operations to `buf`. If `buf` is NULL, buffering is disabled for the stream.
- **setvbuf(FILE* stream, char* buf, int mode, size_t size)** ↦ `int`: Sets the buffer to be used by the stream for I/O operations to `buf` and specifies the mode of buffering. The `size` parameter specifies the size of the buffer. Returns zero on success, or nonzero if an error occurs.

12.3 Formatted input/output

- **fprintf(FILE* stream, const char* format, ...)** ↦ `int`: Writes formatted data to the given output stream. Returns the number of characters written, or a negative value if an error occurs.
- **fscanf(FILE* stream, const char* format, ...)** ↦ `int`: Reads formatted input from a stream. Returns the number of input items successfully matched and assigned, which can be fewer than provided for, or even zero in the event of an early matching failure.

- **printf(const char* format, ...)** ↦ `int`: Prints formatted data to stdout. Returns the number of characters transmitted, or a negative value if an error occurs.
- **scanf(const char* format, ...)** ↦ `int`: Reads formatted input from stdin. Returns the number of items successfully read and assigned, which may be less than the number of format specifiers.
- **snprintf(char* str, size_t size, const char* format, ...)** ↦ `int`: Writes formatted output to a string of up to `size` - 1 characters, including a terminating null byte. Returns the number of characters that would have been written if `size` had been sufficiently large, not counting the terminating null byte.
- **sprintf(char* str, const char* format, ...)** ↦ `int`: Writes formatted data to a string. Returns the number of characters written, excluding the terminating null byte.
- **sscanf(const char* str, const char* format, ...)** ↦ `int`: Reads formatted input from a string. Returns the number of fields successfully converted and assigned; the return value does not include fields that were read but not assigned.
- **vfprintf(FILE* stream, const char* format, va_list arg)** ↦ `int`: Writes formatted data from a variable argument list to the given output stream. Returns the number of characters written, or a negative value if an error occurs.
- **vfscanf(FILE* stream, const char* format, va_list arg)** ↦ `int`: Reads formatted data from a stream into a variable argument list. Returns the number of input items successfully matched and assigned.
- **vprintf(const char* format, va_list arg)** ↦ `int`: Prints formatted data from a variable argument list to stdout. Returns the number of characters transmitted, or a negative value if an error occurs.
- **vscanf(const char* format, va_list arg)** ↦ `int`: Reads formatted data into a variable argument list from stdin. Returns the number of items successfully read and assigned.
- **vsnprintf(char* str, size_t size, const char* format, va_list arg)** ↦ `int`: Writes formatted data from a variable argument list to a sized buffer. Returns the number of characters that would have been written if `size` had been sufficiently large, not counting the terminating null byte.
- **vsprintf(char* str, const char* format, va_list arg)** ↦ `int`: Writes formatted data from a variable argument list to a string. Returns the number of characters written, excluding the terminating null byte.
- **vsscanf(const char* str, const char* format, va_list arg)** ↦ `int`: Reads formatted data from a string into a variable argument list. Returns the number of fields successfully converted and assigned.

12.4 Character input/output

- **fgetc(FILE* stream)** ↦ `int`: Gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream. Returns the character read as an unsigned char cast to an int or EOF on end of file or error.
- **fgets(char* str, int n, FILE* stream)** ↦ `char*`: Reads in at most one less than `n` characters from stream and stores them into the buffer pointed to by `str`. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A

terminating null byte ('\0') is stored after the last character in the buffer. Returns `str` on success, and NULL on error or when end of file occurs while no characters have been read.

- **fputc(int char, FILE* stream)** ↦ `int`: Writes the character `char` (an unsigned char) to the specified stream and advances the position indicator for the stream. Returns the character written as an unsigned char cast to an int or EOF on error.
- **fputs(const char* str, FILE* stream)** ↦ `int`: Writes a string to the specified stream up to but not including the null character. Returns a nonnegative number on success, or EOF on error.
- **getc(FILE* stream)** ↦ `int`: Gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream. This function may be implemented as a macro. Returns the character read as an unsigned char cast to an int or EOF on end of file or error.
- **getchar(void)** ↦ `int`: Gets the next character (an unsigned char) from stdin and advances the position indicator for the stream. Returns the character read as an unsigned char cast to an int or EOF on end of file or error.
- **gets(char* str)** ↦ `char*` (**Deprecated and dangerous**): Reads a line from stdin into the buffer pointed to by `str` until either a terminating newline or EOF, which it replaces with a null byte ('\0'). No check for buffer overrun is performed (please use `fgets` instead). Returns `str` on success, and NULL on error or when end of file occurs while no characters have been read.
- **putc(int char, FILE* stream)** ↦ `int`: Writes the character `char` (an unsigned char) to the specified stream and advances the position indicator for the stream. This function may be implemented as a macro. Returns the character written as an unsigned char cast to an int or EOF on error.
- **putchar(int char)** ↦ `int`: Writes the character `char` (an unsigned char) to stdout and advances the position indicator for the stream. Returns the character written as an unsigned char cast to an int or EOF on error.
- **puts(const char* str)** ↦ `int`: Writes the string `str` and a trailing newline to stdout. Returns a nonnegative number on success, or EOF on error.
- **ungetc(int char, FILE* stream)** ↦ `int`: Pushes the character `char` (an unsigned char) back onto the input buffer of the specified stream. The character will be returned by the next read on that stream. Returns the character pushed back after cast to unsigned char and then to int, or EOF on error.

12.5 Direct input/output

- **fread(void* ptr, size_t size, size_t nmemb, FILE* stream)** ↦ `size_t`: Reads `nmemb` items of data, each `size` bytes long, from the stream pointed to by `stream`, storing them at the location given by `ptr`. Returns the number of items (not bytes) successfully read which may be less than `nmemb` if an error occurs or if the end of file is reached before reaching `nmemb`.
- **fwrite(const void* ptr, size_t size, size_t nmemb, FILE* stream)** ↦ `size_t`: Writes `nmemb` items of data, each `size` bytes long, to the stream pointed to by `stream`, obtaining them from the location given by `ptr`. Returns the number of items successfully written, which may be less than `nmemb` if an error occurs.

12.6 File positioning

- **fgetpos(FILE* stream, fpos_t* pos)** ↦ `int`: Gets the current file position of the stream and stores it in the object pointed to by `pos`. Returns zero on success, or nonzero on error.
- **fseek(FILE* stream, long int offset, int whence)** ↦ `int`: Sets the file position of the stream to the given offset. The new position, measured in bytes, is obtained by adding `offset` bytes to the position specified by `whence`. If `whence` is set to `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, the offset is relative to the start of the file, the current file position, or the end of the file, respectively. Returns zero on success, or nonzero on error.
- **fsetpos(FILE* stream, const fpos_t* pos)** ↦ `int`: Sets the file position of the stream to the position specified by `pos`, which is a position given by a previous call to `fgetpos`. Returns zero on success, or nonzero on error.
- **ftell(FILE* stream)** ↦ `long int`: Returns the current file position of the stream, or -1 on an error.
- **rewind(FILE* stream)** ↦ `void`: Sets the file position to the beginning of the file of the given stream. Also clears the error and the end-of-file indicators for the stream.

12.7 Error-handling

- **clearerr(FILE* stream)** ↦ `void`: Clears both the end-of-file and the error indicators for the stream. This function does not return a value.
- **feof(FILE* stream)** ↦ `int`: Checks the end-of-file indicator for the given stream. Returns a nonzero value if the end-of-file indicator is set for `stream`, otherwise zero.
- **ferror(FILE* stream)** ↦ `int`: Checks the error indicator for the given stream. Returns a nonzero value if the error indicator is set for `stream`, otherwise zero.
- **perror(const char* s)** ↦ `void`: Prints a descriptive error message to stderr. First, if the string `s` is not NULL and does not point to an empty string, this string is printed followed by a colon and a space. Then an error message corresponding to the current value of the global variable `errno` is printed, followed by a newline. This function does not return a value.

12.8 Macros

- **BUFSIZ**: This macro constant expands to an integral expression with the size of the buffer used by the setbuf function.
- **EOF**: It is a macro definition of type int that expands into a negative integral constant expression (generally, -1).
- **FILENAME_MAX**: This macro constant expands to an integral expression corresponding to the size needed for an array of char elements to hold the longest file name string allowed by the library.
- **FOPEN_MAX**: This macro constant expands to an integral expression that represents the maximum number of files that can be opened simultaneously.

- **L_tmpna**: This macro constant expands to an integral expression corresponding to the size needed for an array of char elements to hold the longest file name string possibly generated by tmpnam.
- **NULL**: This macro expands to a null pointer constant.
- **TMP_MAX**: This macro expands to the minimum number of unique temporary file names that are guaranteed to be possible to generate using tmpnam.

12.9 Types

- **FILE** Object containing information to control a stream (type)
- **fpos_t** Object containing information to specify a position within a file (type)
- **size_t** Unsigned integral type (type)

Miscellaneous utilities: <cstdlib>

13.1 Process control

- **abort(void)** ↦ `void`: Causes abnormal program termination without cleaning up. The program terminates as if by calling 'std::terminate', and the termination status is implementation-defined. This function does not return.
- **exit(int status)** ↦ `void`: Causes normal program termination with cleaning up. Performs standard cleanup for program termination (like flushing buffers) and calls the functions registered by 'std::atexit', then terminates the program with the given status. This function does not return.
- **quick_exit(int status)** ↦ `void` (**C++11**): Causes quick program termination without completely cleaning up. Calls the functions registered by 'std::at_quick_exit' and terminates the program with the given status, without calling destructors for objects of automatic, thread, or static storage duration. This function does not return.
- **_Exit(int status)** ↦ `void` (**C++11**): Causes normal program termination without cleaning up, similar to 'abort'. It terminates the program without calling destructors or any functions registered with 'std::atexit' or 'std::at_quick_exit'. This function does not return.
- **atexit(void (*func)(void))** ↦ `int`: Registers a function to be called on 'std::exit()' invocation. The specified function is called when the program terminates normally. Returns zero on success, non-zero if the registration fails.
- **at_quick_exit(void (*func)(void))** ↦ `int` (**C++11**): Registers a function to be called on 'std::quick_exit' invocation. The specified function is called when the program terminates via 'std::quick_exit'. Returns zero on success, non-zero if the registration fails.
- **system(const char* command)** ↦ `int`: Calls the host environment's command processor ('/bin/sh' on Unix, 'CMD.EXE' on Windows) with the specified command. Returns an implementation-defined value if successful, and a negative value if an error occurs or if the command processor is not available.
- **getenv(const char* name)** ↦ `char*`: Provides access to the list of environment variables. Returns a pointer to the value associated with the environment variable 'name', or 'NULL' if the variable is not found. The returned pointer points to static data whose content should not be modified.

Posix regex API <regex.h>

Note:-

This is not the standard regex library

- **regcomp(regex_t *preg, const char *regex, int cflags)** ↦ `int`: Compiles a regular expression into a format that can be used for pattern matching. It stores the result in the object pointed to by `preg`. The `cflags` argument is used to customize the compilation process. Returns zero on success, and a non-zero error code on failure.
- **regexec(const regex_t *preg, const char *string, size_t nmatch, regmatch_t pmatch[], int eflags)** ↦ `int`: Matches a null-terminated string against the precompiled pattern specified by `preg`. `nmatch` and `pmatch` are used to specify how many matches to find and where to store them. The `eflags` argument can be used to modify the execution behavior. Returns zero if the regular expression matches; otherwise, returns a non-zero value indicating the error or mismatch.
- **regerror(int errcode, const regex_t *preg, char *errbuf, size_t errbuf_size)** ↦ `size_t`: Translates an error code returned by ‘regcomp()‘ or ‘regexec()‘ into a human-readable error message. The error message is stored in ‘errbuf‘, which is a buffer provided by the caller, with a maximum size specified by ‘errbuf_size‘. Returns the size of the error message (including the terminating null byte). If ‘errbuf_size‘ is too small to hold the error message, the message is truncated but the return value still reflects the full length of the message.
- **regfree(regex_t *preg)** ↦ `void`: Frees any memory that was allocated by ‘regcomp()‘ for the compiled regular expression pointed to by ‘preg‘. After calling ‘regfree()‘, the ‘regex_t‘ object should not be used again until it has been reinitialized by another call to ‘regcomp()‘.

Directory Input/Output <dirent.h>

- **chdir(const char *path)** ↦ `int`: Changes the current working directory of the calling process to the directory specified in `path`. Returns zero on success, and -1 on failure, setting `errno` to indicate the error.
- **getcwd(char *buf, size_t size)** ↦ `char*`: Copies an absolute pathname of the current working directory to the array pointed to by `buf`, which is of length `size`. If `size` is large enough, returns `buf`; if `size` is too small, NULL is returned, and `errno` is set to `ERANGE`; on other errors, NULL is returned, and `errno` is set appropriately.
- **opendir(const char *name)** ↦ `DIR*`: Opens a directory stream corresponding to the directory name, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory. Returns NULL if an error occurs, setting `errno` to indicate the error.
- **readdir(DIR *dirp)** ↦ `struct dirent*`: Reads the next directory entry from the directory stream pointed to by `dirp`. Returns a pointer to a `struct dirent` representing the next directory entry, or NULL when reaching the end of the directory stream or if an error occurs.

15.1 The DIR type

The DIR type is an opaque data type that represents a directory stream. A directory stream is an ordered sequence of all the directory entries in a particular directory. Directory entries include files, subdirectories, and other types of file system objects contained in the directory.

You cannot see the structure of DIR directly, as it is hidden (opaque) to provide abstraction and portability across different operating systems and filesystems. This means you interact with directory streams using pointer variables of type DIR *, and you manipulate these streams through the functions provided by <dirent.h>, such as opendir(), readdir(), and closedir().

15.2 dirent Structure

The struct dirent structure represents an individual directory entry, which could be a file, a subdirectory, or another type of file system object. The exact contents of this structure can vary between different operating systems, but it generally includes the following members:

- **d_ino:** The inode number of the directory entry. The inode number is a unique identifier within a filesystem.
- **d_name:** A character array (string) containing the name of the directory entry. This name is not the full path, but just the filename or directory name.

15.2.1 Example

```
DIR* d;
struct dirent* dir;
d = opendir("."); // Open the current directory

if (d) {
    while ((dir = readdir(d)) != NULL) {
        printf("%s\n", dir->d_name); // Print the name of each
↪  directory entry
    }
    closedir(d); // Close the directory stream
}

return 0;
```

Type traits <type_traits>

16.1 Primary type categories

- `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 a 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)
- `is_lvalue_reference` (C++11): checks if a type is an lvalue reference (class template)
- `is_rvalue_reference` (C++11): checks if a type is an rvalue reference (class template)
- `is_member_object_pointer` (C++11): checks if a type is a pointer to a non-static member object (class template)
- `is_member_function_pointer` (C++11): checks if a type is a pointer to a non-static member function (class template)

16.2 Composite type categories

- `is_fundamental`: checks if a type is a fundamental type (class template)
- `is_arithmetic`: checks if a type is an arithmetic type (class template)
- `is_scalar`: checks if a type is a scalar type (class template)
- `is_object`: checks if a type is an object type (class template)
- `is_compound`: checks if a type is a compound type (class template)
- `is_reference`: checks if a type is either an lvalue reference or rvalue reference (class template)
- `is_member_pointer`: checks if a type is a pointer to a non-static member function or object (class template)

16.3 Type properties

- `is_const`: checks if a type is const-qualified (class template)
- `is_volatile`: checks if a type is volatile-qualified (class template)
- `is_trivial`: checks if a type is trivial (class template)
- `is_trivially_copyable`: checks if a type is trivially copyable (class template)
- `is_standard_layout`: checks if a type is a standard-layout type (class template)
- `is_pod` (deprecated in C++20): checks if a type is a plain-old data (POD) type (class template)
- `has_unique_object_representations`: checks if every bit in the type's object representation contributes to its value (class template)
- `is_empty`: checks if a type is a class (but not union) type and has no non-static data members (class template)
- `is_polymorphic`: checks if a type is a polymorphic class type (class template)
- `is_abstract`: checks if a type is an abstract class type (class template)
- `is_final`: checks if a type is a final class type (class template)
- `is_aggregate`: checks if a type is an aggregate type (class template)
- `is_implicit_lifetime`: checks if a type is an implicit-lifetime type (class template)
- `is_signed`: checks if a type is a signed arithmetic type (class template)
- `is_unsigned`: checks if a type is an unsigned arithmetic type (class template)
- `is_bounded_array`: checks if a type is an array type of known bound (class template)
- `is_unbounded_array`: checks if a type is an array type of unknown bound (class template)
- `is_scoped_enum`: checks if a type is a scoped enumeration type (class template)

16.4 Type relationships

- `is_same`: checks if two types are the same (class template)
- `is_base_of`: checks if a type is a base of the other type (class template)
- `is_virtual_base_of`: checks if a type is a virtual base of the other type (class template)
- `is_convertible`: checks if a type can be converted to the other type (class template)
- `is_nothrow_convertible`: checks if a type can be converted to the other type without throwing exceptions (class template)
- `is_layout_compatible`: checks if two types are layout-compatible (class template)
- `is_pointer_interconvertible_base_of`: checks if a type is a pointer-interconvertible (initial) base of another type (class template)
- `is_invocable`: checks if a type can be invoked (as if by `std::invoke`) with the given argument types (class template)
- `is_invocable_r`: checks if a type can be invoked with the given argument types, returning a specific result type (class template)
- `is_nothrow_invocable`: checks if a type can be invoked without throwing exceptions (class template)
- `is_nothrow_invocable_r`: checks if a type can be invoked with the given argument types, returning a specific result type without throwing exceptions (class template)

16.5 Const-volatility specifiers

- `remove_cv`: removes both const and volatile specifiers from the given type (class template)
- `remove_const`: removes the const specifier from the given type (class template)
- `remove_volatile`: removes the volatile specifier from the given type (class template)
- `add_cv`: adds both const and volatile specifiers to the given type (class template)
- `add_const`: adds the const specifier to the given type (class template)
- `add_volatile`: adds the volatile specifier to the given type (class template)

16.6 References

- `remove_reference`: removes a reference from the given type (class template)
- `add_lvalue_reference`: adds an lvalue reference to the given type (class template)
- `add_rvalue_reference`: adds an rvalue reference to the given type (class template)

16.7 Pointers

- `remove_pointer`: removes a pointer from the given type (class template)
- `add_pointer`: adds a pointer to the given type (class template)

16.8 Sign modifiers

- `make_signed`: obtains the corresponding signed type for the given integral type (class template)
- `make_unsigned`: obtains the corresponding unsigned type for the given integral type (class template)

16.9 Miscellaneous transformations

- `aligned_storage` (deprecated in C++23): defines the type suitable for use as uninitialized storage for types of given size (class template)
- `aligned_union` (deprecated in C++23): defines the type suitable for use as uninitialized storage for all given types (class template)
- `decay`: applies type transformations as when passing a function argument by value (class template)
- `remove_cvref`: combines `std::remove_cv` and `std::remove_reference` (class template)
- `enable_if`: conditionally removes a function overload or template specialization from overload resolution (class template)
- `conditional`: chooses one type or another based on compile-time boolean (class template)
- `common_type`: determines the common type of a group of types (class template)
- `common_reference` and `basic_common_reference`: determines the common reference type of a group of types (class template)
- `underlying_type`: obtains the underlying integer type for a given enumeration type (class template)
- `result_of` (removed in C++20) and `invoke_result`: deduces the result type of invoking a callable object with a set of arguments (class template)
- `void_t`: void variadic alias template (alias template)
- `type_identity`: returns the type argument unchanged (class template)
- `unwrap_reference` and `unwrap_ref_decay`: get the reference type wrapped in `std::reference_wrapper` (class template)

16.10 Most common

- `std::is_integral`: Checks if a type is an integral type (e.g., `int`, `bool`).
- `std::is_floating_point`: Checks if a type is a floating-point type (e.g., `float`, `double`).
- `std::is_pointer`: Checks if a type is a pointer type.
- `std::is_const`: Checks if a type is `const`-qualified.
- `std::is_reference`: Checks if a type is either an lvalue or rvalue reference.
- `std::is_same`: Checks if two types are the same.
- `std::is_class`: Checks if a type is a class (excluding union).
- `std::is_array`: Checks if a type is an array type.
- `std::enable_if`: Conditionally includes/excludes function overloads or template specializations based on a compile-time condition.
- `std::remove_cv`: Removes `const` and `volatile` qualifiers from a type.
- `std::remove_reference`: Removes lvalue or rvalue reference from a type.
- `std::decay`: Transforms a type similarly to how it would be passed to a function by value (removes references, `const`, etc.).
- `std::underlying_type`: Retrieves the underlying type of an enumeration.

Smart pointers

17.1 Unique_ptr

17.1.1 Constructors

- **std::unique_ptr();** Default constructor (holds nullptr).
- **explicit std::unique_ptr(T* ptr);** Takes ownership of a raw pointer.
- **explicit std::unique_ptr(T* ptr, Deleter d);** Takes ownership and uses a custom deleter.
- **std::unique_ptr(std::unique_ptr&& other) noexcept;** Move constructor (transfers ownership).
- **explicit std::unique_ptr(T[] ptr);** Specialized constructor for arrays.
- **explicit std::unique_ptr(T[] ptr, Deleter d);** Specialized constructor for arrays with a custom deleter.

The explicit keyword prevents implicit conversions, ensuring that std::unique_ptr is only created in a deliberate and controlled way. This helps avoid unintended behavior, especially when working with raw pointers and preventing accidental ownership transfer.

```
0 std::unique_ptr<int> up1 = new int(10);  // ERROR: Cannot
  ↪ implicitly convert
1 std::unique_ptr<int> up2(new int(10));   // OK: Explicitly
  ↪ calling the constructor
```

17.1.2 Modifer methods

- **pointer release()**: returns a pointer to the managed object and releases the ownership
- **void reset(ptr)**: replaces the managed object
- **void swap(other)**: swaps the managed objects

17.1.3 Observers

- **pointer get()**: returns a pointer to the managed object
- **Deleter& get_deleter()**: returns the deleter that is used for destruction of the managed object
- **bool operator bool**: checks if there is an associated managed object, true if *this owns an object, false otherwise.

17.1.4 Operators

We have operator bool, dereference (*), the arrow operator (->), index operator ([]), output operator («), and the comparison operators. Comparison operators compare the pointer not the value

17.1.5 Non member functions

- **unique_ptr<T> make_unique(args, size)**: creates a unique pointer that manages a new object
- **unique_ptr<T> make_unique_for_overwrite(args, size)** creates a unique pointer that manages a new object
- **void swap(lhs,rhs)**: specializes the std::swap algorithm

17.2 shared_ptr

17.2.1 Constructors

- **std::shared_ptr();** Default constructor (holds nullptr).
- **explicit std::shared_ptr(T* ptr);** Takes ownership of a raw pointer and enables shared ownership.
- **std::shared_ptr(T* ptr, Deleter d);** Takes ownership with a custom deleter.
- **std::shared_ptr(T* ptr, Deleter d, Allocator a);** Takes ownership with a custom deleter and allocator.
- **std::shared_ptr(const std::shared_ptr& other);** Copy constructor (increments reference count).
- **std::shared_ptr(std::shared_ptr&& other) noexcept;** Move constructor (transfers ownership, does not increase reference count).
- **template<class Y> std::shared_ptr(const std::shared_ptr<Y>& other);** Copy constructor for related types (e.g., shared_ptr<Derived> to shared_ptr<Base>).
- **template<class Y> std::shared_ptr(std::shared_ptr<Y>&& other);** Move constructor for related types.
- **template<class Y> explicit std::shared_ptr(const std::weak_ptr<Y>& other);** Creates a shared_ptr from a weak_ptr, if the resource is still available.
- **explicit std::shared_ptr(std::nullptr_t);** Constructs a shared_ptr that holds nullptr.

17.2.2 Modifiers

- **void reset(ptr, deleter(optional), allocator(optional))**: replaces the managed object
- **void swap(other)**: swaps the managed objects

17.2.3 Observers

- **pointer get()**: returns the stored pointer
- **long use_count()**: returns the number of shared_ptr objects referring to the same managed object
- **bool unique()**: checks whether the managed object is managed only by the current shared_ptr object
- **operator bool:**
- **bool owner_before(other):** provides owner-based ordering of shared pointers. used to compare ownership order of std::shared_ptr (or std::weak_ptr). It does not compare the actual pointer values but rather their control blocks (ownership metadata)

```
std::shared_ptr<int> sp1 = std::make_shared<int>(10);
std::shared_ptr<int> sp2 = std::make_shared<int>(20);

if (sp1.owner_before(sp2)) {
    std::cout << "sp1's ownership is before sp2\n";
} else if (sp2.owner_before(sp1)) {
    std::cout << "sp2's ownership is before sp1\n";
} else {
    std::cout << "sp1 and sp2 have the same ownership\n";
}
```

- **bool owner_equal(other):** provides owner-based equal comparison of shared pointers

  Checks whether this shared_ptr and other share ownership or are both empty. The comparison is such that two smart pointers compare equivalent only if they are both empty or if they both own the same object

- **bool operator bool()**: Checks if *this stores a non-null pointer

17.2.4 Overloads

We have operator bool, dereference, arrow, index operator, operator «, and all comparision operators

Note that the comparison operators do not compare the value that the pointer points to, but rather the object itself.

```
shared_ptr<int> ptr1 = std::make_shared<int>(20);
shared_ptr<int> ptr2 = std::make_shared<int>(20);

cout << (ptr1 == ptr2) << endl; // False

shared_ptr<int> ptr3 = std::make_shared<int>(20);
shared_ptr<int> ptr4 = ptr3;

cout << (ptr3 == ptr4) << endl; // True
```

Further note that the output stream operator («) outputs the value of the ptr (the address that it holds)

```
shared_ptr<int> ptr1 = std::make_shared<int>(20);
cout << ptr1; // 0x55e55983b2c0
```

17.2.5 Non member functions

- **make_shared:** creates a shared pointer that manages a new object
- **make_shared_for_overwrite:** creates a shared pointer that manages a new object
- **allocate_shared** creates a shared pointer that manages a new object allocated using an allocator
- **allocate_shared_for_overwrite** creates a shared pointer that manages a new object allocated using an allocator
- **static_pointer_cast** applies static_cast, dynamic_cast, const_cast, or reinterpret_cast to the stored pointer
- **dynamic_pointer_cast** applies static_cast, dynamic_cast, const_cast, or reinterpret_cast to the stored pointer
- **const_pointer_cast** applies static_cast, dynamic_cast, const_cast, or reinterpret_cast to the stored pointer
- **reinterpret_pointer_cast** applies static_cast, dynamic_cast, const_cast, or reinterpret_cast to the stored pointer
- **std::swap(std::shared_ptr)**: specializes the std::swap algorithm

17.3 weak_ptr

17.3.1 Constructors

- **std::weak_ptr();** Default constructor (creates an empty weak pointer).
- **std::weak_ptr(const std::weak_ptr& other);** Copy constructor (creates a new weak pointer that shares the ownership control block).
- **std::weak_ptr(std::weak_ptr&& other) noexcept;** Move constructor (transfers ownership of the weak reference).
- **template<class Y> std::weak_ptr(const std::weak_ptr<Y>& other);** Copy constructor for related types.
- **template<class Y> std::weak_ptr(std::weak_ptr<Y>&& other);** Move constructor for related types.
- **template<class Y> std::weak_ptr(const std::shared_ptr<Y>& other);** Constructs a weak_ptr from a shared_ptr, sharing ownership control but not ownership of the resource.

17.3.2 Modifiers

- **void reset()**: Releases the reference to the managed object
- **void swap(other)** swaps the managed objects

17.3.3 Observers

- **long use_count()**: returns the number of shared_ptr objects that manage the object
- **bool expired()**: checks whether the referenced object was already deleted
- **shared_ptr<T> lock()**: creates a shared_ptr that manages the referenced object
- **bool owner_before(other)**: provides owner-based ordering of weak pointers
- **bool owner_equal(other)**: provides owner-based equal comparison of weak pointers

Note: No operator overloads

17.3.4 Non member functions

- **void std::swap(lhs,rhs)**: specializes the std::swap algorithm