Interview cheatsheet

dabljues

Contents

1 C++ language

1.1 Bitwise operations

List of bitwise operations:

- `~` (NOT)
- `&` (AND)
- `|` (OR)
- `^` (XOR)
- `<<` left shift
- `>>` right shift

```cpp
int a = 5;  // a = 0101
int b = 7;  // b = 0111
std::cout << (a & b) << '\n';          // a & b = 0101
std::cout << (a | b) << '\n';          // a | b = 0111
std::cout << (~a) << '\n';             // ~a    = 1010
std::cout << (~b) << '\n';             // ~b    = 1000
std::cout << (a << 2) << '\n';         // a << 2 = 010100 = 20
std::cout << (b << 2) << '\n';         // b << 2 = 011100 = 28
std::cout << (a >> 2) << '\n';         // a >> 2 = 01
std::cout << (b >> 2) << '\n';         // b >> 2 = 01
std::cout << (a << 31) << '\n';        // a << 31 = -INT_MAX
std::cout << (b << 31) << '\n';        // b << 31 = -INT_MAX
std::cout << (a >> 31) << '\n';        // a >> 31 = 0
std::cout << (b >> 31) << '\n';        // b >> 31 = 0
std::cout << (a << 32) << '\n';        // a << 32 = a (UNDEFINED BEHAVIOR)
std::cout << (b << 32) << '\n';        // b << 32 = b (UNDEFINED BEHAVIOR)
std::cout << (a >> 32) << '\n';        // a >> 32 = a (UNDEFINED BEVAHIOR)
std::cout << (b >> 32) << '\n';        // b >> 32 = b (UNDEFINED BEHAVIOR)
std::cout << (1 << 31) << '\n';        // 1 << 31 = -INT_MAX
std::cout << ((1 << 31) - 1) << '\n';  // (1 << 31) - 1 = +INT_MAX
std::cout << (1 >> 31) << '\n';        // 1 >> 31 = 0
```

Left shifting `int` by `31` results in `-INT_MAX`

Right shifting `int` by `31` results int `0`.

Right shifting `int` by `31` and subtracting `1` results int `+INT_MAX`.

Left or right shifting anything by `32` results in undefined `behavior`

1.2 Functions

Functions are defined by syntax: (and they can use `auto` return type deduction)

```cpp
return_type function_name(list_of_arguments) {}
```

In C++17

```cpp
auto function_name(list_of_arguments) -> return_type {}
```

Functions may have special attributes (specified before function return type and name):

- `inline` - very important if in header file which is included in multiple translation units
- `[[nodiscard]]` - encourages the compiler to issue a warning if the return value is discarded
- `[[noreturn]]` - indicates that the function does not return
- `[[deprecated]]` - indicates that the use of the name or entity declared with this attribute is allowed, but discouraged for some reason
- `[[maybe_unused]]` - suppresses compiler warnings on unused entities, if any
- `[[likely]]` and `[[unlikely]]` indicates that the compiler should optimize for the case where a path of execution through a statement is more or less likely than any other path of execution

1.3 Const and constexpr

Const means: **I promise I won't change this value**

```cpp
const int foo = 5;
foo = 10; // compile error
```

Pointer to const vs. const pointer:

```cpp
int a = 0;
int b = 1;

const int* foo = &a; // pointer to const int
foo = &b;  // OK
*foo = 5;  // compile error: cannot assign to variable that is const

int* const bar = &a; // const pointer to int
bar = &b;  // compile error: cannot assign to variable that is const
*bar = 5;  // OK

const int* const baz = &a; // const pointer to const int
baz = &b;  // compile error: cannot assign to variable that is const
*baz = 5;  // compile error: cannot assign to variable that is const
```

Constexpr roughly means: **To be evaluated at compile time**

```cpp
int pow_3(int num) { return num * num * num; }

constexpr int pow_2(const int num) { return num * num; }

int main()
{
    const int a = 2.0;
    constexpr int PI = 3.14;
    int p1 = pow_2(PI);        // evaluated at compile time
    int p2 = pow_2(a);         // evaluated at compile time
    int p3 = pow_2(pow_3(a));  // evaluated at runtime - pow_3() is not
        constexpr, p3 cannot be declared constexpr
}
```

1.4 new and delete

`new`

`delete`

Usage:

```cpp
int* foo = new int(5);
delete foo; // free memory occupied by int foo

int* bar = new int[1000000];
delete[] bar; // free memory occupied by 1 million elements array

struct baz
{
    int* x;
    baz() : x(new int[1000000]) {}
    ~baz() { delete[] x; }
};

// Freeing memory of array of objects with custom destructor:
baz** baz_array = new baz*[100];
for (auto i = 0; i < 100; i++)
{
    delete baz_array[i];
}
delete[] baz_array;
```

Deleting a nullptr:

```cpp
int* foo = nullptr;
delete foo; // no destructors are called, and the deallocation function is not
    called
```

1.5 Constructors (without copy and move)

```cpp
class foo
{
public:
    int x = 42;
    foo() { std::cout << "Default constructor called!\n"; }
    foo(int z) : x(z) { std::cout << "One-argument constructor called!\n"; }
    foo(std::initializer_list<int> l) { std::cout << "Initializer list
        constructor called!\n"; }
};
```

Invoking constructors:

```cpp
foo f;            // default constructor called
foo f1 = foo();   // default constructor called
foo f2 = 1;       // one-argument constructor called
foo f3 = (1);     // one-argument constructor called
foo f4 = {1};     // initializer list constructor called
foo f5(1);        // one-argument constructor called
foo f6(1);        // one-argument constructor called
foo f7{1};        // initializer list constructor called
```

However, if we define one or more argument constructor explicit:

```cpp
class foo
{
public:
    int x = 42;
    foo() { std::cout << "Default constructor called!\n"; }
```

```cpp
    explicit foo(int z) : x(z) { std::cout << "One-argument constructor
        called!\n"; }
    foo(std::initializer_list<int> l) { std::cout << "Initializer list
        constructor called!\n"; }
};
```

Then we end up with:

```cpp
foo f;              // default constructor called
foo f1 = foo();     // default constructor called
foo f2 = 1;         // error: cannot convert from 'int' to 'foo'
foo f3 = (1);       // error: cannot convert from 'int' to 'foo'
foo f4 = {1};       // initializer list constructor called
foo f5(1);          // one-argument constructor called
foo f6(1);          // one-argument constructor called
foo f7{1};          // initializer list constructor called
```

1.6 Destructor

```cpp
class foo
{
private:
    int x*;
public:
    foo() { x = new int[1000000]; }
    ~foo() { delete[] x; }
}
```

Destructor should never throw an exception, because it may happen during `stack unwinding`!

1.7 Copy

```cpp
class foo
{
public:
    int x;
    foo() {}
    foo(const foo& f) : x(f.x) { std::cout << "Copy constructor called!\n"; }
    foo& operator=(const foo& f)
    {
        std::cout << "Copy assignment operator called!\n";
        x = f.x;
        return *this;
    }
};
```

Invoking copy constructor:

```cpp
foo f1;

foo f2(f1);     // copy constructor called
foo f3 = f1;    // copy constructor called
foo f4 = (f1);  // copy constructor called
foo f5 = {f1};  // copy constructor called
```

Invoking copy assignment operator:

```cpp
foo f1;

foo f2;
```

```
f2 = f1;  // copy assignment operator called
foo f3;
f3 = (f1);  // copy assignment operator called
foo f4;
f4 = {f1};  // copy assignment operator called
```

1.8 Move

`std::exchange` is basically an std::move for non-class types (it replaces the old value with new value and returns the old value)

```
#include <utility>
// for std::move and std::exchange

class foo
{
public:
    int x;
    foo() {}
    foo(foo&& f) : x(std::exchange(f.x, 0)) { std::cout << "Move constructor
        called!\n"; }
    foo& operator=(foo&& f)
    {
        x = std::exchange(f.x, 0);
        std::cout << "Move assignment operator called!\n";
        return *this;
    }
};
```

Invoking move constructor:

```
foo f1;
foo f2;
foo f3;
foo f4;

foo f_move1(std::move(f1));     // move constructor called
foo f_move2 = std::move(f2);    // move constructor called
foo f_move3 = (std::move(f3));  // move constructor called
foo f_move4 = {std::move(f4)};  // move constructor called
```

Invoking move assignment operator:

```
foo f1;
foo f2;
foo f3;

foo f_move1;
f_move1 = std::move(f3);  // move assignment operator called
foo f_move2;
f_move2 = (std::move(f2));  // move assignment operator called
foo f_move3;
f_move3 = {std::move(f3)};  // move assignment operator called
```

1.9 Methods and friend functions/classes

Class/struct methods have to be defined inside of them:

```cpp
class foo
{
private:
    int x;

public:
    foo();
    void bar();  // here
};
```

They can be declared inside of them:

```cpp
class foo
{
private:
    int x;

public:
    foo();
    void bar() { std::cout << x << '\n'; }  // here
};
```

or outside of them:

```cpp
class foo
{
private:
    int x;

public:
    foo();
    void bar();
};

void foo::bar() { std::cout << x << '\n'; }  // here
```

Methods can be declared: (it is written after function arguments list)

- `const` - calling that method will not change objects internal state
- `override` - just an indicator that method is overriding it's base class equivalent, see `Virtual inheritance`
- `noexcept` - functions can have it as well, indicates that function cannot throw

Friend functions are functions which have access to object's protected/private members:

```cpp
class foo
{
private:
    int x = 0;
public:
    foo();
    friend void print_x(foo& f);
};

void print_x(foo& f)
{
    std::cout << f.x << '\n';
}
```

Friend classes:

```
class foo
{
private:
    int x = 0;

public:
    friend class bar;
};

class bar
{
public:
    void print_foo(foo& f) { std::cout << "foo::x = " << f.x; }
};

int main()
{
    foo f;
    bar b;
    b.print_foo(f);
}
```

1.10 Inheritance

```
class foo
{
public:
    foo() : x{0}, y{1}, z{2} {}
    int x;

private:
    int y;

protected:
    int z;
};

class bar : public foo
{
public:
    void baz()
    {
        std::cout << x << '\n';
        std::cout << z << '\n';
    }
};

auto main() -> int
{
    bar b;
    std::cout << b.x << '\n';
    std::cout << b.y << '\n'; // error: cannot acces private member of foo
```

```
    std::cout << b.z << '\n'; // error: cannot acces protected member of foo
}
```

Inheritance always goes down, so:

- `public` inheritance doesn't change anything
- `protected` inheritance makes public members protected
- `private` inheritance makes public and protected members private

1.11 Virtual inheritance

A class in an abstract `class` if one or more of it's methods are declared pure `virtual` (= 0)!

```
class foo
{
public:
    foo() { std::cout << "foo() called!\n"; }
    virtual void baz() { std::cout << "foo's baz() called!\n"; } // virtual
        keyword does not matter here
    ~foo() { std::cout << "~foo() called!\n"; }
};

class bar : public foo // or here
{
public:
    bar() { std::cout << "bar() called!\n"; }
    void baz() { std::cout << "bar's baz() called!\n"; }
    ~bar() { std::cout << "~bar() called!\n"; }
};

int main()
{
    foo f;    // foo() called
    f.baz();  // foo::baz() called
    bar b;    // foo() called, bar() called
    b.baz();  // bar::baz() called
}  // ~bar() called, ~foo() called, ~foo() called
```

Problems start with multiple inheritance:

```
class D
{
public:
    D() { std::cout << "D() called!\n"; }
    virtual void foo() { std::cout << "D's foo() called!\n"; }
    ~D() { std::cout << "~D() called!\n"; }
};

class C : public D
{
public:
    C() { std::cout << "C() called!\n"; }
    void foo() override { std::cout << "C's foo() called!\n"; }
    ~C() { std::cout << "~C() called!\n"; }
};
```

```cpp
class B : public D
{
public:
    B() { std::cout << "B() called!\n"; }
    ~B() { std::cout << "~B() called!\n"; }
};

class A : public C, public B
{
public:
    A() { std::cout << "A() called!\n"; }
    ~A() { std::cout << "~A() called!\n"; }
};

auto main() -> int
{
    A a;
    a.foo(); // error: ambiguous access of 'foo', could be the 'foo' in base 'C',
        or could be the 'foo' in base 'D'
}
```

To solve this, one can use virtual inheritance:

```cpp
class D
{
public:
    D() { std::cout << "D() called!\n"; }
    virtual void foo() { std::cout << "D's foo() called!\n"; }
    virtual ~D() { std::cout << "~D() called!\n"; }
};

class C : public virtual D
{
public:
    C() { std::cout << "C() called!\n"; }
    virtual void foo() { std::cout << "C's foo() called!\n"; }
    virtual ~C() { std::cout << "~C() called!\n"; }
};

class B : public virtual D // B also has to do virtual inheritance
{
public:
    B() { std::cout << "B() called!\n"; }
    virtual ~B() { std::cout << "~B() called!\n"; }
};

class A : public C, public B
{
public:
    A() { std::cout << "A() called!\n"; }
    ~A() { std::cout << "~A() called!\n"; }
};

auto main() -> int
```

```
{
    A a;
    a.foo(); // outputs: C's foo() called!
}
```

1.12 Polymorphism

Coming back to Virtual inheritance a little bit:

```
class foo
{
public:
    foo() { std::cout << "foo() called!\n"; }
    void baz() { std::cout << "foo's baz() called!\n"; }
    ~foo() { std::cout << "~foo() called!\n"; }
};

class bar : public foo
{
public:
    bar() { std::cout << "bar() called!\n"; }
    void baz() { std::cout << "bar's baz() called!\n"; }
    ~bar() { std::cout << "~bar() called!\n"; }
};

int main()
{
    foo* f = new bar();
    f->baz();  // calls foo::baz() - we didn't want it
    delete f;
}
```

We can fix this by declaring foo:baz() virtual:

```
class foo
{
public:
    foo() { std::cout << "foo() called!\n"; }
    virtual void baz() { std::cout << "foo's baz() called!\n"; } // declared
        virtual
    ~foo() { std::cout << "~foo() called!\n"; }
};

class bar : public foo
{
public:
    bar() { std::cout << "bar() called!\n"; }
    void baz() override { std::cout << "bar's baz() called!\n"; }
    ~bar() { std::cout << "~bar() called!\n"; }
};

int main()
{
    foo* f = new bar();
    f->baz();  // calls bar::baz() - OK!
```

```
    delete f;
}
```

The two above examples have one problem: wrong destruction!

The problem lies in manipulating `bar` object via pointer to it's base class: `foo`:

```
int main()
{
    foo* f = new bar();  // foo() called, bar() called
    f->baz();
    delete f;  // only ~foo() called - WRONG!
}
```

The rule is: **Always define base classes' destructors `virtual` when they're meant to be manipulated polymorphically!**

We fix it by declaring `foo` destructor `virtual`:

```
class foo
{
public:
    foo() { std::cout << "foo() called!\n"; }
    virtual void baz() { std::cout << "foo's baz() called!\n"; }
    virtual ~foo() { std::cout << "~foo() called!\n"; }
};

class bar : public foo
{
public:
    bar() { std::cout << "bar() called!\n"; }
    void baz() override { std::cout << "bar's baz() called!\n"; }
    ~bar() { std::cout << "~bar() called!\n"; }
};

int main()
{
    foo* f = new bar();  // foo() called, bar() called
    f->baz();
    delete f;  // ~bar() called, ~foo() called - OK!
}
```

1.13 Casts

There are several types of casts in C++:

- plain old C cast `(type_name)`
- `static_cast<>`
- `dynamic_cast<>`
- `const_cast<>`
- `reinterpret_cast<>`

`static_cast<>`

Simple cast, performs no runtime checks:

```
double a = 5.0;
double b = 3.0;
double c = static_cast<int>(a / b);
```

dynamic_cast<>

Useful when you don't know the type of the object. Returns `nullptr` if cast didn't suceed:

```
if (foo *f = dynamic_cast<foo*>(&baz)) {
  ...
} else if (bar *b = dynamic_cast<bar*>(&baz)) {
  ...
}
```

const_cast<>

Used to "cast away" the `const`, but still changing the value after casting can result in runtime error. It is primarly used to match the function prototype which sometimes wants non-const parameter, although it doesn't change it.

```
const int const_foo = 5;
int *nonconst_foo = const_cast<int*>(&foo); // removes const
*nonconst_foo = 10; // potential run-time error
```

reinterpret_cast<>

It handles conversions between certain unrelated types, such as from one pointer type to another incompatible pointer type. It will simply perform a binary copy of the data without altering the underlying bit pattern:

```
char c = 10;       // 1 byte
int *q = static_cast<int*>(&c); // compile-time error (int is 4 bytes)
int *r = reinterpret_cast<int*>(&c); // forced conversion - works!
```

1.14 Templates

1.15 Exceptions and error handling

```
try
{
    auto result = foo();
}
catch (std::exception& e)
{
    std::cout << e.what() << '\n'
}
```

User-defined exception:

```
#include <exception>
class foo : public std::exception
{
public:
    const char* what() const throw ()
    {
        return "C++ Exception";
    }
}
```

Other methods:

- return codes, usually based on `enums`
- `errno`

errno

We check a global variable `errno` for errors and print them out using `std::strerror()`:

```
#include <cmath>
#include <cerrno>
#include <cstring>
#include <clocale>

int main()
{
    double not_a_number = std::log(-1.0);
    if (errno == EDOM) {
        std::cout << "log(-1) failed: " << std::strerror(errno) << '\n';
        std::setlocale(LC_MESSAGES, "de_DE.utf8");
        std::cout << "Or, in German, " << std::strerror(errno) << '\n';
    }
}
```

2 STL containers

2.1 `std::tuple`

Defined in `<tuple>`

```
std::tuple t1{1, 3.14, "Hello"}; // since C++17
auto t2 = std::make_tuple(1, 3.14, "Hello"); // before
```

Getting the values:

```
std::tuple<int, double, std::string> get_tuple();
auto [id, value, comment] = get_tuple(); // C++17 structured bindings
```

2.2 `std::pair`

Defined in `<utility>`

```
std::pair p1{1, "word"};
auto p2 = std::make_pair(1, "word");
```

Gettin values:

```
auto [val, comment] = p1;
```

2.3 `std::array`

Defined in `<array>`

```
std::array<int> a = {1, 2, 3, 4};
```

Iterating:

```
for (auto& val: a)
{
    std::cout << val << '\n';
}
```

Getting the values:

```
auto first = a[0];
auto last = a[v.size() - 1];
auto first = *a.begin();
auto last = *(a.end() - 1);
auto first = a.at(0); // throws std::out_of_range
```

Inserting the values:

```
a[0] = 42; // changing first value
```

2.4 std::vector

Defined in <vector>

```
std::vector<int> v{1, 2, 3, 4};
```

Iterating:

```
for (auto& val: v)
{
    std::cout << val << '\n';
}
```

Getting the values:

```
auto first = v[0];
auto last = v[v.size() - 1];
auto first = *v.begin();
auto last = *(v.end() - 1);
auto first = v.at(0); // throws std::out_of_range
```

Inserting the values:

```
v.push_back(42);
v.emplace_back(42); // in-place - faster if non-trivial
v[0] = 42 // changing first value
```

2.5 std::list

Defined in <list>

```
std::list<int> l1 {1, 2, 3, 4};
std::list l2{v.begin(), v.end()}; // deduction guides
```

Iterating:

```
for (auto& val: l1)
{
    std::cout << val << '\n';
}
```

Getting the values: **you shouldn't!** `std::list` is meant for iterating it!

```
auto first_element = *l1.begin();
```

Inserting the values (front, back):

```
l1.push_front(42);
l1.emplace_front(42);
l1.push_back(42);
l1.emplace_back(42);
```

Inserting the values anywhere:

```
#include <algorithm>
auto it = std::find(l1.begin(), l1.end(), 3); // insert a value before 3 by
    finding it
if (it != l1.end())
{
    l1.insert(it, 42);
    l1.emplace(it, 42); // in-place - faster if non-trivial
}
```

2.6 std::unordered_set, std::set

Defined in <unordered_set>, <set>

std::unordered_set is generally a first choice because of performance, if you don't need values to be ordered.

```
std::unordered_set<std::string> u_set{"dog", "turtle", "cat"};
std::set<std::string> set{"dog", "turtle", "cat"};
std::unordered_set u_set1{1, 2, 3, 4}; // deduction guides
```

Iterating:

```
for(auto& val: u_set)
{
    std::cout << val << '\n' ;
}
```

Getting the values: **you shouldn't!** std::unordered_set, std::set are meant for iterating them!

```
auto it = u_set.find("dog"); // returns iterator
```

Inserting the values:

```
u_set.insert("elephant");
u_set.emplace("elephant"); // in-place - faster if non-trivial
```

2.7 std::unordered_map, std::map

Defined in <unordered_map>, <map>

std::unordered_map is generally a first choice because of performance, if you don't need keys to be ordered.

```
std::unordered_map<std::string, int> u_map = {{"key1", 7}, {"key2", 42}};
std::map<std::string, int> map = {{"key1", 7}, {"key2", 42}};
```

Iterating:

```
for(auto const& [key, val]: u_map)
{
    std::cout << key << ':' << val << '\n' ;
}
```

Getting the values:

```
auto v1 = u_map["key1"]; // doesn't throw
auto v2 = u_map.at("key1"); // can throw if no key
```

Inserting the values:

```
u_map.insert({"key3", 100}); // no need to create an std::pair, compiler will do
    it for us
u_map["key4"] = 1000;
```

or a faster, constructing in place, emplace():

```
u_map.emplace("key3", 100);
```

2.8 std::queue, std::deque

Defined in <queue>, <deque>

```
std::queue<int> q1 {1, 2};
std::deque<int> d1 {3, 4};
```

Inserting the values:

```cpp
// Queue
q1.push(5);
q1.emplace(5); // in-place - faster if non-trivial
// Dequeue
q2.push_front(4);
q2.emplace_front(4);
q2.push_back(5);
q2.emplace_back(5);
```

Getting the values:

```cpp
auto first = q1.front();
auto last = q1.back();
```

Deque stands for `double-ended queue` and is able to add and remove elements from the front it

Removing with `.pop()`, `.pop_front()`, `.pop_back()` is `void`, they don't return the element, they just remove it!!!

2.9 `std::priority_queue`

Defined in `<queue>`

```cpp
std::priority_queue<int> pq1;
std::priority_queue pq2{v.begin(), v.end()}; // deduction guides

std::priority_queue<type, underlaying_container, compare_method> pq3; // full
    construction syntax, i.e.:
std::priority_queue<int, std::vector<int>, std::greater<int>> pq3;
```

The same methods as simple `std::queue`

3 STL algorithms

3.1 `std::copy`

Defined in `<algorithm>`

On STL types:

```cpp
std::vector<int> v1{1, 2, 3, 4, 5, 6};
std::vector<int> v2;
#include <iterator>
// for std::back_inserter
std::copy(v1.begin(), v1.end(),
          std::back_inserter(v2));
```

Requirements for user-defined types: `.begin()`, `.end()`

3.2 `std::iota`

Defined in `<numeric>`

Fills a container with incrementing values

On STL types:

```cpp
std::vector<int> v(10);
std::iota(v.begin(), v.end(), 0);
```

Requirements for user-defined types: `.begin()`, `.end()`

3.3 std::sort

Defined in <algorithm>

On STL types:

```
std::sort(c.begin(), c.end());
struct {
    bool operator()(int a, int b) const
    {
        return a < b;
    }
} customLess;
std::sort(c.begin(), c.end(), customLess); // sort using a custom function object
std::sort(c.begin(), c.end(), [](int a, int b) { // sort using a lambda expression
    return a > b;
});
```

Requirements for user-defined types:

```
bool foo::operator<(const foo& l, const foo& r) const;
```

3.4 std::find, std::find_if, std::find_if_not

Defined in <algorithm>

On STL types:

```
auto result = std::find(c.begin(), c.end(), 42);
auto result1 = std::find_if(c.begin(), c.end(), someFunc);
auto result2 = std::find_if_not(c.begin(), c.end(), someFunc);
```

They search only for first occurence!!!

You can write a function that takes the iterator they return and proceed further

Requirements for user-defined types:

```
bool foo::operator==(const foo& r) const;
```

3.5 std::swap

Defined in <algorithm>

On STL types:

```
int a = 3;
int b = 5;
std::swap(a, b);
std::vector<int> v{1, 2, 3, 4, 5};
std::swap(v.begin(), v.begin() + 1); // swap 1 with 2
```

Requirements for user-defined types: must be move assignable and move constructible!

3.6 std::accumulate

Defined in <numeric>

On STL types:

```
auto sum = std::accumulate(c.begin(), c.end(), 0);
#include <functional>
auto multiply = std::accumulate(c.begin(), c.end(), 1, std::multiplies<int>());
```

Requirements for user-defined types:

```
// Define a lambda!
auto result = std::accumulate(v.begin(), v.end(), 0, [](const int sum, const S& s)
                                                      {
                                                        return sum + s.x;
                                                      });
```

3.7 std::for_each

Defined in <algorithm>

On STL types:

```
std::vector<int> v{3, 4, 2, 8, 15, 267};
auto print = [](const int& n) { std::cout << " " << n; };
std::for_each(v.begin(), v.end(), print); // using lambda

struct Sum
{
    Sum(): sum{0} { }
    void operator()(int n) { sum += n; }
    int sum;
};
std::for_each(v.begin(), v.end(), Sum); // // calls Sum::operator() for each
    number - not constructor!!!
```

Requirements for user-defined types:

```
void foo::operator()(type t);
```

3.8 std::random_shuffle

Defined in <algorithm>

On STL types:

```
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::random_device rd;
std::mt19937 g(rd());

std::random_shuffle(v.begin(), v.end(), g);
```

Requirements for user-defined types: `.begin()`, `.end()`

3.9 std::lower_bound, std::upper_bound

Defined in <algorithm>

`std::lower_bound` returns iterator to first value comparing **not less than**

`std::upper_bound` returns iterator to first value comparing **greater than**

```
std::vector<int> v{10, 10, 10, 20, 20, 20, 30, 30, 30};
std::lower_bound(v.begin(), v.end(), 20); // returns -> 3 (first position of 20)
std::upper_bound(v.begin(), v.end(), 20); // returns -> 6 (first position of 30)
```

4 C++ misc

4.1 File I/O

4.2 Smart pointers

There are are few types of smart pointers in C++:

- `unique_ptr` - basically a normal pointer with automatic deletion, no overhead
- `shared_ptr` - reference-counted smart pointer, there's overhead
- `weak_ptr` - holds a non-owning ("weak") reference to an object that is managed by `shared_ptr`
- `auto_ptr` - deprecated, first attempt to smart pointer

They are all defined in `<memory>` header!

unique_ptr

Creation:

```
std::unique_ptr<int> foo1(new int(47));                    // constructor
std::unique_ptr<int> foo2 = std::make_unique<int>(47);  // make_unique
std::unique_ptr<int> foo3(std::move(foo2));  // unique_ptr cannot be copied, it
    has to be moved
```

Getting the value, accesing the fields:

```
int* ptr = foo1.get();
std::cout << *foo1 << '\n';
std::cout << foo1->bar << '\n';
```

Chaning the pointer or releasing it:

```
foo1.reset(new int(100)); // deletes the previous object and assigns new one
foo1.release(); // returns pointer to managed object and releases ownership -
    DOESN'T DELETE OBJECT
```

shared_ptr

Creation:

```
std::shared_ptr<int> foo1(new int(47));                    // constructor
std::shared_ptr<int> foo2 = std::make_shared<int>(47);  // make_shared
std::shared_ptr<int> foo3(foo2);                          // shared_ptr can be
    copied
```

Getting the value, accesing the fields:

```
int* ptr = foo1.get();
std::cout << *foo1 << '\n';
std::cout << foo1->bar << '\n';
std::cout << foo1.use_count() << '\n'; // returns number of references to managed
    object
std::cout << foo1.unique() << '\n'; // checks whether the managed object is
    managed only by the current shared_ptr instance
```

Chaning the pointer or releasing it:

```
foo1.reset(new int(100)); // deletes the previous object and assigns new one
```

weak_ptr

Complete example:

```
std::weak_ptr<int> gw;

void observe()
{
```

```cpp
    std::cout << "use_count == " << gw.use_count() << ": ";
    if (auto spt = gw.lock()) // Has to be copied into a shared_ptr before usage,
        lock() creates shared_ptr
    {
        std::cout << *spt << "\n";
    }
    else
    {
        std::cout << "gw is expired\n";
    }
}

int main()
{
    {
        auto sp = std::make_shared<int>(42);
        gw = sp;
        observe();
    }
    observe();
}
```

Output:

```
use_count == 1: 42
use_count == 0: gw is expired
```

4.3 Atomics

4.4 Generating random numbers

4.5 Measuring the time

4.6 `<filesystem>`