perfect—here are programs **13 → 30** with **C++98-friendly code** and detailed, step-by-step explanations. You can compile any of them with:

g++ -std=c++98 -O2 -Wall -Wextra -pedantic file.cpp -o file.exe

# 13) std::bitset — fixed-size bit container

#include <iostream>  
#include <bitset>  
#include <string>  
int main() {  
 std::bitset<8> b(std::string("00010110")); // 8 bits: 0b00010110  
 std::cout << "bits set = " << b.count() << "\n"; // number of 1s  
 std::cout << "b[1] = " << b[1] << "\n"; // index from LSB at 0  
 b.flip(2); // toggle bit #2  
 std::cout << b << "\n"; // print as 8 chars 0/1  
}

**What/why:** bitset<N> stores exactly N bits, supports bitwise ops (& | ^ ~), indexing, count().  
**Use cases:** flags, masks, puzzles, compact booleans.  
**Notes:** Size N is compile-time constant.

# 14) Iterators — generic loop with begin()/end()

#include <iostream>  
#include <vector>  
int main() {  
 int a[] = {3, 1, 4};  
 std::vector<int> v(a, a + 3);  
 for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {  
 std::cout << \*it << " "; // dereference to get element  
 }  
 std::cout << "\n";  
}

**What/why:** Iterators generalize pointers; algorithms operate on iterator **ranges**.  
**Key:** begin() points to first element; end() is one-past-last (sentinel).

# 15) Output iterator adapter — std::back\_inserter + std::copy

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <iterator>  
int main() {  
 int a[] = {1, 2, 3};  
 std::vector<int> v; // empty; size grows during copy  
 std::copy(a, a + 3, std::back\_inserter(v)); // internally calls v.push\_back(...)  
 for (std::size\_t i = 0; i < v.size(); ++i) std::cout << v[i] << " ";  
 std::cout << "\n";  
}

**Why:** Without back\_inserter, copy requires the destination to be pre-sized. With it, algorithms can **append** into containers.

# 16) std::find + std::count — linear search/count

#include <iostream>  
#include <vector>  
#include <algorithm>  
int main() {  
 int a[] = {1, 3, 3, 7};  
 std::vector<int> v(a, a + 4);  
 std::vector<int>::iterator it = std::find(v.begin(), v.end(), 3);  
 if (it != v.end()) {  
 std::cout << "first 3 at index " << (it - v.begin()) << "\n";  
 }  
 std::cout << "count(3) = " << std::count(v.begin(), v.end(), 3) << "\n";  
}

**Why:** Simple searches on any forward sequence. For sorted data, prefer binary-search helpers (see #19).

# 17) Erase–remove idiom — physically remove from a vector

#include <iostream>  
#include <vector>  
#include <algorithm>  
int main() {  
 int a[] = {1, 2, 3, 2, 4};  
 std::vector<int> v(a, a + 5);  
 // remove returns new logical end after moving survivors forward  
 v.erase(std::remove(v.begin(), v.end(), 2), v.end());  
 for (std::size\_t i = 0; i < v.size(); ++i) std::cout << v[i] << " ";  
 std::cout << "\n";  
}

**What:** remove compacts elements ≠ value to the front (doesn’t change size). erase trims the tail.  
**Why:** Core STL idiom for sequence containers (vector, deque, string).

# 18) Deduplicate with std::unique (after sorting)

#include <iostream>  
#include <vector>  
#include <algorithm>  
int main() {  
 int a[] = {3, 1, 3, 2, 2, 1};  
 std::vector<int> v(a, a + 6);  
 std::sort(v.begin(), v.end()); // 1 1 2 2 3 3  
 v.erase(std::unique(v.begin(), v.end()), v.end()); // 1 2 3  
 for (std::size\_t i = 0; i < v.size(); ++i) std::cout << v[i] << " ";  
 std::cout << "\n";  
}

**Why:** unique removes **consecutive** duplicates; to remove **all** duplicates, sort first.  
**Note:** unique returns new logical end; then you erase.

# 19) Binary search helpers — lower\_bound / upper\_bound / equal\_range

#include <iostream>  
#include <vector>  
#include <algorithm>  
int main() {  
 int a[] = {1, 2, 2, 2, 5};  
 std::vector<int> v(a, a + 5); // sorted!  
 std::pair<std::vector<int>::iterator, std::vector<int>::iterator> r =  
 std::equal\_range(v.begin(), v.end(), 2);  
 std::cout << "all 2s in index range ["  
 << (r.first - v.begin()) << ", "  
 << (r.second - v.begin()) << ")\n";  
}

**Why:** Fast O(log N) searching on **sorted** ranges.

* lower\_bound: first position ≥ value
* upper\_bound: first position > value
* equal\_range: both at once (subrange of equals)

# 20) std::stable\_sort with custom comparator (sort by length, keep order)

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <string>  
struct ByLen {  
 bool operator()(const std::string& a, const std::string& b) const {  
 if (a.size() != b.size()) return a.size() < b.size();  
 return a < b; // tiebreak (required for strict weak ordering)  
 }  
};  
int main() {  
 std::string a[] = {"bb","a","ccc","aa"};  
 std::vector<std::string> v(a, a + 4);  
 std::stable\_sort(v.begin(), v.end(), ByLen());  
 for (std::size\_t i = 0; i < v.size(); ++i) std::cout << v[i] << " ";  
 std::cout << "\n";  
}

**Why:** Custom ordering; **stable** keeps relative order of equal-length strings.  
**Rule:** Comparator must define a **strict weak ordering** (no contradictions).

# 21) std::transform + toupper — element-wise mapping

#include <iostream>  
#include <algorithm>  
#include <string>  
#include <cctype>  
int up(int c) { return std::toupper((unsigned char)c); }  
int main() {  
 std::string s = "MiT";  
 std::transform(s.begin(), s.end(), s.begin(), up); // in-place upper  
 std::cout << s << "\n";  
}

**Why:** Apply a function to each element of a range; write to same or different range.  
**Gotcha:** Cast to unsigned char before toupper to avoid UB for negative chars.

# 22) <numeric> — std::accumulate for sum/product

#include <iostream>  
#include <vector>  
#include <numeric>  
#include <functional>  
int main() {  
 int a[] = {1,2,3,4};  
 std::vector<int> v(a,a+4);  
 int sum = std::accumulate(v.begin(), v.end(), 0); // 0 is init  
 int prod = std::accumulate(v.begin(), v.end(), 1, std::multiplies<int>()); // 1 is init  
 std::cout << sum << " " << prod << "\n"; // 10 24  
}

**Why:** Generic reduction. Init value matters (identity of operation).

# 23) Set algorithms — union & intersection on **sorted** ranges

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <iterator>  
int main() {  
 int A[] = {1,2,4}, B[] = {2,3,4};  
 std::vector<int> a(A,A+3), b(B,B+3), out;  
 std::set\_union(a.begin(), a.end(), b.begin(), b.end(), std::back\_inserter(out));  
 for (std::size\_t i=0;i<out.size();++i) std::cout<<out[i]<<" "; std::cout<<"\n";  
 out.clear();  
 std::set\_intersection(a.begin(), a.end(), b.begin(), b.end(), std::back\_inserter(out));  
 for (std::size\_t i=0;i<out.size();++i) std::cout<<out[i]<<" "; std::cout<<"\n";  
}

**Why:** Classical set ops in **sorted** sequences.  
**Notes:** Inputs must be sorted; output will be sorted.

# 24) std::partition — split by predicate (fast but not stable)

#include <iostream>  
#include <vector>  
#include <algorithm>  
struct IsEven { bool operator()(int x) const { return x % 2 == 0; } };  
int main() {  
 int A[] = {1,2,3,4,5,6};  
 std::vector<int> v(A,A+6);  
 std::partition(v.begin(), v.end(), IsEven()); // evens first, odds after  
 for (std::size\_t i=0;i<v.size();++i) std::cout<<v[i]<<" ";  
 std::cout << "\n";  
}

**Why:** Group elements by a condition in linear time.  
**Caveat:** **Not stable** (relative order changes). See #27 for stable version.

# 25) std::remove\_if + erase — remove by predicate

#include <iostream>  
#include <vector>  
#include <algorithm>  
struct LessThan3 { bool operator()(int x) const { return x < 3; } };  
int main() {  
 int A[] = {1,2,3,4};  
 std::vector<int> v(A,A+4);  
 v.erase(std::remove\_if(v.begin(), v.end(), LessThan3()), v.end());  
 for (std::size\_t i=0;i<v.size();++i) std::cout<<v[i]<<" ";  
 std::cout << "\n";  
}

**Why:** Same erase–remove pattern but with a predicate instead of a value.

# 26) std::map with custom comparator (descending keys)

#include <iostream>  
#include <map>  
#include <string>  
#include <functional>  
int main() {  
 std::map<int, std::string, std::greater<int> > m; // sort keys high→low  
 m[2] = "two";  
 m[1] = "one";  
 m[3] = "three";  
 for (std::map<int,std::string, std::greater<int> >::iterator it = m.begin(); it != m.end(); ++it)  
 std::cout << it->first << " ";  
 std::cout << "\n";  
}

**Why:** Associative containers accept a comparator type as a template parameter (default std::less<Key>).  
**Result:** Iteration shows 3 2 1.

# 27) std::stable\_partition — split by predicate, **keep order**

#include <iostream>  
#include <algorithm>  
#include <string>  
struct IsUpper { bool operator()(char c) const { return c >= 'A' && c <= 'Z'; } };  
int main() {  
 std::string s = "aAbBcC";  
 std::stable\_partition(s.begin(), s.end(), IsUpper());  
 std::cout << s << "\n"; // "ABCabc" — uppercase chunk keeps original order (A,B,C)  
}

**Why:** Like partition, but **stability** preserved (relative order among trues and among falses).

# 28) std::rotate — cyclic shift of a range

#include <iostream>  
#include <vector>  
#include <algorithm>  
int main() {  
 int A[] = {1,2,3,4,5};  
 std::vector<int> v(A,A+5);  
 // move [begin, begin+2) to the end: [1,2 | 3,4,5] -> [3,4,5,1,2]  
 std::rotate(v.begin(), v.begin() + 2, v.end());  
 for (std::size\_t i=0;i<v.size();++i) std::cout<<v[i]<<" ";  
 std::cout << "\n";  
}

**Why:** Efficiently bring an element (or block) to the front without extra storage.

# 29) std::replace / std::replace\_if — in-place value change

#include <iostream>  
#include <vector>  
#include <algorithm>  
struct IsOdd { bool operator()(int x) const { return x % 2 == 1; } };  
int main() {  
 int A[] = {1,2,3,4,5};  
 std::vector<int> v(A,A+5);  
 std::replace(v.begin(), v.end(), 2, 20); // replace 2 -> 20  
 std::replace\_if(v.begin(), v.end(), IsOdd(), 99); // replace odd -> 99  
 for (std::size\_t i=0;i<v.size();++i) std::cout<<v[i]<<" ";  
 std::cout << "\n";  
}

**Why:** Quick in-place transformations without changing size.

# 30) Stream iterators + std::map — word frequency counter

#include <iostream>  
#include <iterator>  
#include <string>  
#include <map>  
int main() {  
 std::istream\_iterator<std::string> in(std::cin); // read words from stdin  
 std::istream\_iterator<std::string> end;  
 std::map<std::string, int> freq;  
 while (in != end) {  
 ++freq[\*in]; // increment count for the read word  
 ++in; // advance the input iterator  
 }  
 for (std::map<std::string,int>::iterator it = freq.begin(); it != freq.end(); ++it)  
 std::cout << it->first << ": " << it->second << "\n";  
}

**Why:** Demonstrates that *streams are iterators*, so algorithms/containers integrate with I/O.  
**Use:** Pipe a file into the program and it prints a sorted frequency table.  
Example run:

$ ./a.exe < words.txt  
Apple: 2  
MIT: 1  
banana: 3

## Recap of core STL ideas these illustrate

* **Containers**: sequence (vector, deque, list), associative (set, map, and multi- variants), adaptors (stack, queue, priority\_queue), and utility (bitset).
* **Algorithms**: generic operations such as sort, find, count, copy, transform, accumulate, unique, remove, partition, stable\_partition, rotate, replace, set algorithms, binary search helpers.
* **Iterators**: glue that connects algorithms to containers (and streams!). Different iterator categories imply which algorithms are allowed.
* **Adapters**: back\_inserter (iterator adaptor) and stack/queue/priority\_queue (container adaptors).

If any specific program above is still fuzzy, tell me which number, and I’ll expand with a step-by-step trace, complexity notes, and typical pitfalls.