Extended CPP Notes

Nathan Warner

Computer Science
Northern Illinois University
April 17, 2024
United States

Contents

STL Containers

1.1 STL Vectors

1.1.1 Implementation

A vector models a dynamic array. Thus, a vector is an abstraction that manages its elements with a dynamic C-style arra

A vector copies its elements into its internal dynamic array. The elements always have a certain order. Thus, a vector is a kind of ordered collection. A vector provides random access. Thus, you can access every element directly in constant time, provided that you know its position. The iterators are random-access iterators, so you can use any algorithm of the STL.

1.1.2 Performance in operations on the end

Vectors provide good performance if you append or delete elements at the end. If you insert or delete in the middle or at the beginning, performance gets worse. This is because every element behind has to be moved to another position. In fact, the assignment operator would be called for every following element.

1.1.3 Size and capacity

Part of the way in which vectors give good performance is by allocating more memory than they need to contain all their elements. To use vectors effectively and correctly, you should understand how size and capacity cooperate in a vector.

Vectors provide the usual size operations size(), empty(), and max_size(). An additional "size" operation is the capacity() function, which returns the number of elements a vector could contain in its actual memory. If you exceed the capacity(), the vector has to reallocate its internal memory.

The capacity of a vector is important for two reasons:

1. Reallocation invalidates all references, pointers, and iterators for elements of the vector
2. Reallocation takes time.

Thus, if a program manages pointers, references, or iterators into a vector, or if speed is a goal, it is important to take the capacity into account

To avoid reallocation, you can use reserve() to ensure a certain capacity before you really need it. In this way, you can ensure that references remain valid as long as the capacity is not exceeded:

Another way to avoid reallocation is to initialize a vector with enough elements by passing additional arguments to the constructor. For example, if you pass a numeric value as parameter, it is taken as the starting size of the vector:

```
std::vector<T> v(5);
```

Note: If the only reason for initialization is to reserve memory, you should use reserve()

Unlike for strings, it is not possible to call reserve() for vectors to shrink the capacity. Calling reserve() with an argument that is less than the current capacity is a no-op

Because the capacity of vectors never shrinks, it is guaranteed that references, pointers, and iterators remain valid even when elements are deleted, provided that they refer to a position before the manipulated elements. However, insertions invalidate all references, pointers, and iterators when the capacity gets exceeded

1.1.4 Constructors

- `vector<Elem> c`
  Default constructor; creates an empty vector without any elements.
- `vector<Elem> c(c2)`
  Copy constructor; creates a new vector as a copy of `c2` (all elements are copied).
- `vector<Elem> c = c2`
  Copy constructor; creates a new vector as a copy of `c2` (all elements are copied).
- `vector<Elem> c(rv)`
  Move constructor; creates a new vector, taking the contents of the rvalue `rv` (since C++11).
- `vector<Elem> c = rv`
  Move constructor; creates a new vector, taking the contents of the rvalue `rv` (since C++11).
- `vector<Elem> c(n)`
  Creates a vector with `n` elements created by the default constructor.
- `vector<Elem> c(n, elem)`
  Creates a vector initialized with `n` copies of element `elem`.
- `vector<Elem> c(beg, end)`
  Creates a vector initialized with the elements of the range [`beg`, `end`].
- `vector<Elem> c{initlist}`
  Creates a vector initialized with the elements of the initializer list `initlist` (since C++11).
- `vector<Elem> c = {initlist}`
  Creates a vector initialized with the elements of the initializer list `initlist` (since C++11).
- `c. vector()`
  Destroys all elements and frees the memory.

1.1.5 Note about at()

Out of all the element access operators and methods: [], at(), front(), back(), only at() performs range checking. If the index is out of range, at() throws an out_of_range

All other functions do not check. A range error results in undefined behavior. Calling operator [ ], front(), and back() for an empty container always results in undefined behavior:

1.1.6 Iterator methods

We have

- begin()
- end()
- rbegin()
- cbegin()
- cend()
- crbegin()
- crend()

1.1.7 Using vectors as 2d arrays

There is an issue that comes up when trying to use 2D arrays, when the size of the matrix is not known at compile time.

```
void f(int nrows, int ncols) {
    int arr[nrows][ncols]
}

int main() {
    f(5,6);
    return 0;
}
```

The problem with this code is that the array arr inside the function f is declared with dimensions nrows and ncols, which are non-constant variables. In C++, standard arrays require their sizes to be constant at compile time, but in your code, the array dimensions are determined by function parameters, which are only known at runtime.

This means int arr[nrows][ncols] is not valid because the array size is determined at runtime, not compile time.

The most common and modern way to handle dynamic arrays in C++ is by using std::vector.

```
void f(int r, int c) {
    vector<vector<int>> m;
    m.resize(r);

    for (int i=0; i<c; ++i) {
        m[i].resize(c);
    }
}
```

Or simply

```
void f(int nrows, int ncols) {
    // Create a vector of vectors (2D array)
    std::vector<std::vector<int>> arr(nrows, std::vector<int>(ncols));

    // Access elements like arr[i][j]
    for (int i = 0; i < nrows; ++i) {
        for (int j = 0; j < ncols; ++j) {
            arr[i][j] = i * ncols + j;  // Example of initializing elements
        }
    }
}

int main() {
    f(5, 6);
    return 0;
}
```

the constructor of std::vector is designed to accept an element initializer. In this case, the std::vector<int>(ncols) argument is used to initialize each element of the outer std::vector<std::vector<int». The constructor in question is

```
vector(size_type count, const T& value);
```

Where size_type in this context is a typedef for size_t.

1.1.8 For those interested

There are some other ways to solve this issue.

1.1.8.1 Manual Dynamic Memory Allocation (Using new)

If for some reason you cannot use std::vector and need to manually allocate dynamic memory, you can use new to create a 2D array. This approach gives you control over memory allocation, but you must manually free the memory to avoid memory leaks.

```cpp
void f(int r, int c) {
    int** arr = new int*[r];

    for (int i=0; i<r; ++i) {
        arr[i] = new int[c];
    }

    // Access elements like arr[i][j]
    int count=0;
    for (int i = 0; i < r; ++i) {
        for (int j = 0; j < c; ++j) {
            arr[i][j] = count++;
            cout << arr[i][j] << endl;
        }
    }

    // Free the memory when done
    for (int i = 0; i < r; ++i) {
        delete[] arr[i];  // Free each row
    }
    delete[] arr;  // Free the array of pointers
}
```

Note:-

heap allocation using new in C++ happens at runtime.

1.1.8.2 Using unique pointer

If you need more control over memory allocation but want to avoid the risk of memory leaks, you can use std::unique_ptr to automatically manage memory

```cpp
void f(int nrows, int ncols) {
    // Allocate memory for a 2D array using std::unique_ptr
    std::unique_ptr<std::unique_ptr<int[]>[]> arr = std::make_unique<std::unique_ptr<int[]>[]>(nrows);
    for (int i = 0; i < nrows; ++i) {
        arr[i] = std::make_unique<int[]>(ncols);  // Allocate each row
    }

    // Access elements like arr[i][j]
    for (int i = 0; i < nrows; ++i) {
        for (int j = 0; j < ncols; ++j) {
            arr[i][j] = i * ncols + j;  // Example of initializing elements
        }
    }
    // No need to manually free memory; std::unique_ptr handles it automatically
}
```

How it works:

1. **std::unique_ptr<std::unique_ptr<int[]>[]>**: is a unique pointer to an array of unique pointers, where each unique pointer in the array manages a dynamically allocated array of integers.

2. **std::make_unique<std::unique_ptr<int[]>[]>(r)**: allocates an array of r unique pointers, each of which will eventually point to a row of integers.

1.1.8.3 Recall: Unique pointer for dynamic arry

```cpp
    #include <memory>
    using std::unique_ptr;
    using std::make_unique;
    int main() {
        unique_ptr<int[]> arr = make_unique<int[]>(size) // Which is the same as int* arr = new int[size]

        return 0;
}
```

1.1.8.4 If sizes are truly known at compile time

If the dimensions are truly known at compile time, you can pass them as template arguments:

```cpp
template <int r, int c>
void f() {
    int arr[r][c];  // Valid because r and c are compile-time constants
    // Initialize and print the array (for demonstration)
    for (int i = 0; i < r; ++i) {
        for (int j = 0; j < c; ++j) {
            arr[i][j] = i * c + j;
            std::cout << arr[i][j] << " ";
        }
        std::cout << std::endl;
    }
}

int main() {
    constexpr int r = 5;
    constexpr int c = 6;
    f<r, c>();  // Call function with compile-time constant dimensions
    return 0;
}
```

The reason your code works when using template parameters is that the template parameters r and c are compile-time constants. In C++, when you use template parameters like this, the values of r and c are determined at compile time, allowing the array sizes to be known by the compiler ahead of time

1.2 STL Deque

1.2.1 Implementation

A deque (pronounced “deck”) is very similar to a vector. It manages its elements with a dynamic array, provides random access, and has almost the same interface as a vector. The difference is that with a deque, the dynamic array is open at both ends. Thus, a deque is fast for insertions and deletions at both the end and the beginning

To provide this ability, the deque is typically implemented as a bunch of individual blocks, with the first block growing in one direction and the last block growing in the opposite direction

1.2.2 Abilities, performance, uses

The abilities of deques differ from those of vectors as follows:

- Inserting and removing elements is fast at both the beginning and the end (for vectors, it is fast only at the end). These operations are done in amortized constant time.
- The internal structure has one more indirection to access the elements, so with deques, element access and iterator movement are usually a bit slower.
- Iterators must be smart pointers of a special type rather than ordinary pointers because they must jump between different blocks.
- In systems that have size limitations for blocks of memory (for example, some PC systems), a deque might contain more elements because it uses more than one block of memory. Thus, max_size() might be larger for deques.
- Deques provide no support to control the capacity and the moment of reallocation. In particular, any insertion or deletion of elements other than at the beginning or end invalidates all pointers, references, and iterators that refer to elements of the deque. However, reallocation may perform better than for vectors because according to their typical internal structure, deques don’t have to copy all elements on reallocation.
- Blocks of memory might get freed when they are no longer used, so the memory size of a deque might shrink (however, whether and how this happens is implementation specific).

1.2.3 When to use deques

- You insert and remove elements at both ends (this is the classic case for a queue).
- You don’t refer to elements of the container.
- It is important that the container frees memory when it is no longer used (however, the standard does not guarantee that this happens).

1.2.4 Constructors

- `deque<Elem> c`
  Default constructor; creates an empty deque without any elements.
- `deque<Elem> c(c2)`
  Copy constructor; creates a new deque as a copy of `c2` (all elements are copied).
- `deque<Elem> c = c2`
  Copy assignment operator; creates a new deque as a copy of `c2` (all elements are copied).
- `deque<Elem> c(rv)`
  Move constructor; creates a new deque, taking the contents of the rvalue `rv` (since C++11).
- `deque<Elem> c = rv`
  Move assignment operator; creates a new deque, taking the contents of the rvalue `rv` (since C++11).
- `deque<Elem> c(n)`
  Creates a deque with `n` elements created by the default constructor.
- `deque<Elem> c(n, elem)`
  Creates a deque initialized with `n` copies of element `elem`.
- `deque<Elem> c(beg, end)`
  Creates a deque initialized with the elements of the range `[beg, end]`.
- `deque<Elem> c {inilist}`
  Creates a deque initialized with the elements of initializer list `inilist` (since C++11).
- `deque<Elem> c = {inilist}`
  Creates a deque initialized with the elements of initializer list `inilist` (since C++11).
- `c.~deque()`
  Destroys all elements and frees the memory.

Deque operations differ from vector operations in only two ways:

1. Deques do not provide the functions for capacity (capacity() and reserve()).
2. Deques do provide direct functions to insert and to delete the first element (push_front() and pop_front()).

1.3 STL Lists

1.3.1 Implementation

Manages its elements as a doubly linked list. As usual, the C++ standard library does not specify the kind of the implementation, but it follows from the list's name, constraints, and specifications.

1.3.2 Abilities

The internal structure of a list is totally different from that of an array, a vector, or a deque. The list object itself provides two pointers, the so-called anchors, which refer to the first and last elements. Each element has pointers to the previous and next elements (or back to the anchor). To insert a new element, you just manipulate the corresponding pointers

Thus, a list differs in several major ways from arrays, vectors, and deques:

- A list does not provide random access. For example, to access the fifth element, you must navigate the first four elements, following the chain of links. Thus, accessing an arbitrary element using a list is slow. However, you can navigate through the list from both end. So accessing both the first and the last elements is fast.
- Inserting and removing elements is fast at each position (provided you are there), and not only at one or both ends. You can always insert and delete an element in constant time, because no other elements have to be moved. Internally, only some pointer values are manipulated.
- Inserting and deleting elements does not invalidate pointers, references, and iterators to other elements.
- A list supports exception handling in such a way that almost every operation succeeds or is a no-op. Thus, you can't get into an intermediate state in which only half of the operation is complete.

1.3.3 Differencs in the methods

The member functions provided for lists reflect these differences from arrays, vectors, and deques as follows:

- Lists provide front(), push_front(), and pop_front(), as well as back(), push_back(), and pop_back().
- Lists provide neither a subscript operator nor at(), because no random access is provided.
- Lists don't provide operations for capacity or reallocation, because neither is needed. Each element has its own memory that stays valid until the element is deleted.
- Lists provide many special member functions for moving and removing elements. These member functions are faster versions of general algorithms that have the same names. They are faster because they only redirect pointers rather than copy and move the values.

1.3.4 Constructors

- list<Elem> c
  Default constructor; creates an empty list without any elements.
- list<Elem> c(c2)
  Copy constructor; creates a new list as a copy of c2 (all elements are copied).
- list<Elem> c = c2
  Copy assignment operator; creates a new list as a copy of c2 (all elements are copied).
- list<Elem> c(rv)
  Move constructor; creates a new list, taking the contents of the rvalue rv (since C++11).
- list<Elem> c = rv
  Move assignment operator; creates a new list, taking the contents of the rvalue rv (since C++11).
- list<Elem> c(n)
  Creates a list with n elements created by the default constructor.
- list<Elem> c(n, elem)
  Creates a list initialized with n copies of element elem.
- list<Elem> c(beg, end)
  Creates a list initialized with the elements of the range [beg, end].
- list<Elem> c{inilist}
  Creates a list initialized with the elements of initializer list inilist (since C++11).
- list<Elem> c = {inilist}
  Creates a list initialized with the elements of initializer list inilist (since C++11).
- c.Iist()
  Destroys all elements and frees the memory.

1.3.5 Element access

With lists, we only have front and back methods. However, these methods do not check for existence. Calling these methods on empty containers results in undefined behavior

Thus, the caller must ensure that the container contains at least one element

1.3.6 Iterator functions

To access all elements of a list, you must use iterators. Lists provide the usual iterator functions. However, because a list has no random access, these iterators are only bidirectional. Thus, you can't call algorithms that require random-access iterators. All algorithms that manipulate the order of elements a lot, especially sorting algorithms, are in this category. However, for sorting the elements, lists provide the special member function sort()

1.3.7 Splice Functions and Functions to Change the Order of Elements

Linked lists have the advantage that you can remove and insert elements at any position in constant time. If you move elements from one container to another, this advantage doubles in that you need only redirect some internal pointers

To support this ability, lists provide not only remove() but also additional modifying member functions to change the order of and relink elements and ranges.

1.4 STL Forward lists

1.4.1 Implementation

A forward list (an instance of the container class forward_list<>), which was introduced with C++11, manages its elements as a singly linked list

Conceptionally, a forward list is a list (object of class list<>) restricted such that it is not able to iterate backward. It provides no functionality that is not also provided by lists. As benefits, it uses less memory and provides slightly better runtime behavior. The standard states: "It is intended that forward_list have zero space or time overhead relative to a hand-written C-style singly linked list. Features that would conflict with that goal have been omitted.

1.4.2 Abilities, limitations

Forward lists have the following limitations compared to lists:

- A forward list provides only forward iterators, not bidirectional iterators. As a consequence, no reverse iterator support is provided, which means that types, such as reverse_iterator, and member functions, such as rbegin(), rend(), crbegin(), and crend(), are not provided.
- A forward list does not provide a size() member function. This is a consequence of omitting features that create time or space overhead relative to a handwritten singly linked list.
- The anchor of a forward list has no pointer to the last element. For this reason, a forward list does not provide the special member functions to deal with the last element, back(), push_back(), and pop_back().
- For all member functions that modify forward lists in a way that elements are inserted or deleted at a specific position, special versions for forward lists are provided. The reason is that you have to pass the position of the element before the first element that gets manipulated, because there you have to assign a new successor element. Because you can't navigate backwards (at least not in constant time), for all these member functions you have to pass the position of the preceding element. Because of this difference, these member functions have a _after suffix in their name. For example, instead of insert(), insert_after() is provided, which inserts new elements after the element passed as first argument; that is, it appends an element at that position.
- For this reason, forward lists provide before_begin() and cbefore_begin(), which yield the position of a virtual element before the first element (technically speaking, the anchor of the linked list), which can be used to let built-in algorithms ending with _after exchange even the first element.

1.4.3 No size()?

The decision not to provide size() might be especially surprising because size() is one of the operations required for all STL containers. Here, you can see the consequences of the design goal to have “zero space or time overhead relative to a hand-written Cstyle singly linked list.” The alternative would have been either to compute the size each time size() is called, which would have linear complexity, or to provide an additional field in the forward_list object for the size, which is updated with each and every operation that changes the number of elements. As the design paper for the forward list, “It’s a cost that all users would have to pay for, whether they need this feature or not.” So, if you need the size, either track it outside the forward_list or use a list instead.

If you have to compute the number of elements, you can use distance()

```cpp
#include <forward_list>
#include <iterator>

std::forward_list<int> l;
std::cout << "Size: " << std::distance(l.begin(), l.end()) <<
    std::endl;
```

1.4.4 Similarities to list

- A forward list does not provide random access. For example, to access the fifth element, you
- must navigate the first four elements, following the chain of links. Thus, using a forward list to access an arbitrary element is slow.
- Inserting and removing elements is fast at each position, if you are there. You can always insert and delete an element in constant time, because no other elements have to be moved. Internally, only some pointer values are manipulated.
- Inserting and deleting elements does not invalidate iterators, references, and pointers to other elements.
- A forward list supports exception handling in such a way that almost every operation succeeds or is a no-op. Thus, you can’t get into an intermediate state in which only half of the operation is complete.
- Forward lists provide many special member functions for moving and removing elements. These member functions are faster versions of general algorithms, because they only redirect pointers rather than copy and move the values. However, when element positions are involved, you have to pass the preceding position, and the member function has the suffix _after in its name.

1.4.5 Constructors

- `forward_list<Elem> c`
  Default constructor; creates an empty forward list without any elements.
- `forward_list<Elem> c(c2)`
  Copy constructor; creates a new forward list as a copy of `c2` (all elements are copied).
- `forward_list<Elem> c = c2`
  Copy assignment operator; creates a new forward list as a copy of `c2` (all elements are copied).
- `forward_list<Elem> c(rv)`
  Move constructor; creates a new forward list, taking the contents of the rvalue `rv` (since C++11).
- `forward_list<Elem> c = rv`
  Move assignment operator; creates a new forward list, taking the contents of the rvalue `rv` (since C++11).
- `forward_list<Elem> c(n)`
  Creates a forward list with `n` elements created by the default constructor.
- `forward_list<Elem> c(n, elem)`
  Creates a forward list initialized with `n` copies of element `elem`.
- `forward_list<Elem> c(beg, end)`
  Creates a forward list initialized with the elements of the range `[beg, end]`.
- `forward_list<Elem> c{inilist}`
  Creates a forward list initialized with the elements of initializer list `inilist` (since C++11).
- `forward_list<Elem> c = {inilist}`
  Creates a forward list initialized with the elements of initializer list `inilist` (since C++11).
- `c.~forward_list()`
  Destroys all elements and frees the memory.

1.5 STL Sets and multisets

1.5.1 Implementation

Sets and multisets are implemented as height balanced binary search trees. (red-black trees)

Set and multiset containers sort their elements automatically according to a certain sorting criterion. The difference between the two types of containers is that multisets allow duplicates, whereas sets do not

The elements of a set or a multiset may have any type T that is comparable according to the sorting criterion. The optional second template argument defines the sorting criterion. If a special sorting criterion is not passed, the default criterion less is used. The function object less sorts the elements by comparing them with operator $<$

The optional third template parameter defines the memory model. The default memory model is the model allocator, which is provided by the C++ standard library.

1.5.2 Strict weak ordering

The sorting criterion must define strict weak ordering, which is defined by the following four properties:

1. It has to be **antisymmetric**.
   - This means that for operator `<`: If $x < y$ is true, then $y < x$ is false.
   - This means that for a predicate `op()`: If `op(x, y)` is true, then `op(y, x)` is false.
2. It has to be **transitive**.
   - This means that for operator `<`: If $x < y$ is true and $y < z$ is true, then $x < z$ is true.
   - This means that for a predicate `op()`: If `op(x, y)` is true and `op(y, z)` is true, then `op(x, z)` is true.
3. It has to be **irreflexive**.
   - This means that for operator `<`: $x < x$ is always false.
   - This means that for a predicate `op()`: `op(x, x)` is always false.
4. It has to have **transitivity of equivalence**, which means roughly: If a is equivalent to b and b is equivalent to c, then a is equivalent to c.
   - This means that for operator `<`: If $!(a < b)$ && $!(b < a)$ is true and $!(b < c)$ && $!(c < b)$ is true, then $!(a < c)$ && $!(c < a)$ is true.
   - This means that for a predicate `op()`: If `op(a, b)`, `op(b, a)`, `op(b, c)`, and `op(c, b)` all yield false, then `op(a, c)` and `op(c, a)` yield false.

Note: Note that this means that you have to distinguish between less and equal. A criterion such as operator `<=` does not fulfill this requirement.

Based on these properties, the sorting criterion is also used to check equivalence. That is, two elements are considered to be duplicates if neither is less than the other (or if both `op(x, y)` and `op(y, x)` are false).

For multisets, the order of equivalent elements is random but stable. Thus, insertions and erasures preserve the relative ordering of equivalent elements (guaranteed since C++11).

1.5.3 Abilities

Like all standardized associative container classes, sets and multisets are usually implemented as balanced binary trees

The major advantage of automatic sorting is that a binary tree performs well when elements with a certain value are searched. In fact, search functions have logarithmic complexity. For example, to search for an element in a set or a multiset of 1,000 elements, a tree search performed by a member function needs, on average, one-fiftieth of the comparisons of a linear search

1.5.4 Changing elements directly, no direct element access

Automatic sorting also imposes an important constraint on sets and multisets: You may not change the value of an element directly

Therefore, to modify the value of an element, you must remove the element having the old value and insert a new element that has the new value. The interface reflects this behavior:

- Sets and multisets don't provide operations for direct element access.
- Indirect access via iterators has the constraint that, from the iterator's point of view, the element value is constant.

1.5.5 Constructors

- `set c`
  Default constructor; creates an empty set/multiset without any elements.
- `set c(op)`
  Creates an empty set/multiset that uses `op` as the sorting criterion.
- `set c(c2)`
  Copy constructor; creates a copy of another set/multiset of the same type (all elements are copied).
- `set c = c2`
  Copy assignment operator; creates a copy of another set/multiset of the same type (all elements are copied).
- `set c(rv)`
  Move constructor; creates a new set/multiset of the same type, taking the contents of the rvalue `rv` (since C++11).

- set c = rv
  Move assignment operator; creates a new set/multiset of the same type, taking the contents of the rvalue rv (since C++11).

- set c(beg, end)
  Creates a set/multiset initialized by the elements of the range [beg, end].

- set c(beg, end, op)
  Creates a set/multiset with the sorting criterion op initialized by the elements of the range [beg, end].

- set c{inilist}
  Creates a set/multiset initialized with the elements of initializer list inilist (since C++11).

- set c = {inilist}
  Creates a set/multiset initialized with the elements of initializer list inilist (since C++11).

- c.s̃et()
  Destroys all elements and frees the memory.

1.5.6 Types

- set<Elem>
  A set that by default sorts with less<> (operator <).

- set<Elem, Op>
  A set that by default sorts with Op.

- multiset<Elem>
  A multiset that by default sorts with less<> (operator <).

- multiset<Elem, Op>
  A multiset that by default sorts with Op.

1.5.7 Constructors

- `set c` Default constructor; creates an empty set/multiset without any elements
- `set c(op)` Creates an empty set/multiset that uses `op` as the sorting criterion
- `set c(c2)` Copy constructor; creates a copy of another set/multiset of the same type (all elements are copied)
- `set c = c2` Copy constructor; creates a copy of another set/multiset of the same type (all elements are copied)
- `set c(rv)` Move constructor; creates a new set/multiset of the same type, taking the contents of the rvalue `rv` (since C++11)
- `set c = rv` Move constructor; creates a new set/multiset of the same type, taking the contents of the rvalue `rv` (since C++11)
- `set c(beg,end)` Creates a set/multiset initialized by the elements of the range `[beg,end)`
- `set c(beg,end,op)` Creates a set/multiset with the sorting criterion `op`, initialized by the elements of the range `[beg,end)`
- `set c(initlist)` Creates a set/multiset initialized with the elements of initializer list `initlist` (since C++11)
- `set c = initlist` Creates a set/multiset initialized with the elements of initializer list `initlist` (since C++11)
- `c.~set()` Destroys all elements and frees the memory

- `set<Elem>` A set that by default sorts with `less<>` (`operator <`)
- `set<Elem,Op>` A set that by default sorts with `Op`
- `multiset<Elem>` A multiset that by default sorts with `less<>` (`operator <`)
- `multiset<Elem,Op>` A multiset that by default sorts with `Op`

1.6 STL Maps and multimaps

Maps and multimaps are containers that manage key/value pairs as elements. These containers sort their elements automatically, according to a certain sorting criterion that is used for the key. The difference between the two is that multimaps allow duplicates, whereas maps do not

1.6.1 Implementation

Maps and multimaps are implemented the same as sets and multisets, height balanced binary search trees (red-black trees).

1.6.2 Template parameters

The first template parameter is the type of the element's key, and the second template parameter is the type of the element's associated value. The elements of a map or a multimap may have any types Key and T that meet the following two requirements:

1. Both key and value must be copyable or movable.
2. The key must be comparable with the sorting criterion.

The optional third template parameter defines the sorting criterion. As for sets, this sorting criterion must define a "strict weak ordering" The elements are sorted according to their keys, so the value doesn't matter for the order of the elements. The sorting criterion is also used to check for equivalence; that is, two elements are equal if neither key is less than the other.

If a special sorting criterion is not passed, the default criterion less<> is used. The function object less<> sorts the elements by comparing them with operator <

1.6.3 Abilities

Sets, multisets, maps, and multimaps typically use the same internal data type. So, you could consider sets and multisets as special maps and multimaps, respectively, for which the value and the key of the elements are the same objects. Thus, maps and multimaps have all the abilities and operations of sets and multisets. Some minor differences exist, however. First, their elements are key/value pairs. In addition, maps can be used as associative arrays.

Maps and multimaps sort their elements automatically, according to the element's keys, and so have good performance when searching for elements that have a certain key. Searching for elements that have a certain value promotes bad performance. Automatic sorting imposes an important constraint on maps and multimaps: You may not change the key of an element directly, because doing so might compromise the correct order. To modify the key of an element, you must remove the element that has the old key and insert a new element that has the new key and the old value. As a consequence, from the iterator's point of view, the element's key is constant. However, a direct modification of the value of the element is still possible, provided that the type of the value is not constant.

1.6.4 Constructors and types

- `map c`
  Default constructor; creates an empty map/multimap without any elements.
- `map c(op)`
  Creates an empty map/multimap that uses `op` as the sorting criterion.
- `map c(c2)`
  Copy constructor; creates a copy of another map/multimap of the same type (all elements are copied).
- `map c = c2`
  Copy assignment operator; creates a copy of another map/multimap of the same type (all elements are copied).
- `map c(rv)`
  Move constructor; creates a new map/multimap of the same type, taking the contents of the rvalue `rv` (since C++11).
- `map c = rv`
  Move assignment operator; creates a new map/multimap of the same type, taking the contents of the rvalue `rv` (since C++11).
- `map c(beg, end)`
  Creates a map/multimap initialized by the elements of the range `[beg, end]`.
- `map c(beg, end, op)`
  Creates a map/multimap with the sorting criterion `op` initialized by the elements of the range `[beg, end]`.
- `map c{inilist}`
  Creates a map/multimap initialized with the elements of initializer list `inilist` (since C++11).
- `map c = {inilist}`
  Creates a map/multimap initialized with the elements of initializer list `inilist` (since C++11).
- `c.m̃ap()`
  Destroys all elements and frees the memory.

Here, map may be one of the following types:

- `map<Key, Val>`
  A map that by default sorts keys with `less<>` (operator `<`).
- `map<Key, Val, Op>`
  A map that by default sorts keys with `Op`.
- `multimap<Key, Val>`
  A multimap that by default sorts keys with `less<>` (operator `<`).
- `multimap<Key, Val, Op>`
  A multimap that by default sorts keys with `Op`.

1.6.5 Using maps as associative arrays

Associative containers don't typically provide abilities for direct element access. Instead, you must use iterators. For maps, as well as for unordered maps, however, there is an exception to this rule. Nonconstant maps provide a subscript operator for direct element access. In addition, since C++11, a corresponding member function at() is provided for constant and nonconstant maps

at() yields the value of the element with the passed key and throws an exception object of type out_of_range if no such element is present

For operator [ ], the index also is the key that is used to identify the element. This means that for operator [ ], the index may have any type rather than only an integral type. Such an interface is the interface of a so-called associative array.

For operator [ ], the type of the index is not the only difference from ordinary arrays. In addition, you can't have a wrong index. If you use a key as the index for which no element yet exists, a new element gets inserted into the map automatically. The value of the new element is initialized by the default constructor of its type. Thus, to use this feature, you can't use a value type that has no default constructor. Note that the fundamental data types provide a default constructor that initializes their values to zero

1.6.6 Constructors

- `map c` Default constructor; creates an empty map/multimap without any elements
- `map c(op)` Creates an empty map/multimap that uses `op` as the sorting criterion
- `map c(c2)` Copy constructor; creates a copy of another map/multimap of the same type (all elements are copied)
- `map c = c2` Copy constructor; creates a copy of another map/multimap of the same type (all elements are copied)
- `map c(rv)` Move constructor; creates a new map/multimap of the same type, taking the contents of the rvalue `rv` (since C++11)
- `map c = rv` Move constructor; creates a new map/multimap of the same type, taking the contents of the rvalue `rv` (since C++11)
- `map c(beg,end)` Creates a map/multimap initialized by the elements of the range `[beg,end)`
- `map c(beg,end,op)` Creates a map/multimap with the sorting criterion `op`, initialized by the elements of the range `[beg,end)`
- `map c(initlist)` Creates a map/multimap initialized with the elements of initializer list `initlist` (since C++11)
- `map c = initlist` Creates a map/multimap initialized with the elements of initializer list `initlist` (since C++11)
- `c.~map()` Destroys all elements and frees the memory

- `map<Key,Val>` A map that by default sorts keys with `less<>` (`operator <`)
- `map<Key,Val,Op>` A map that by default sorts keys with `Op`
- `multimap<Key,Val>` A multimap that by default sorts keys with `less<>` (`operator <`)
- `multimap<Key,Val,Op>` A multimap that by default sorts keys with `Op`

1.7 Example of bounds and equal range

1.8 STL Unordered containers

Strictly speaking, the C++ standard library calls unordered containers "unordered associative containers." However, I will just use "unordered containers" when I refer to them. With "associative containers," I still refer to the "old" associative containers, which are provided since C++98 and implemented as binary trees (set, multiset, map, and multimap).

Conceptionally, unordered containers contain all the elements you insert in an arbitrary order . That is, you can consider the container to be a bag: you can put in elements, but when you open the bag to do something with all the elements, you access them in a random order. So, in contrast with (multi)sets and (multi)maps, there is no sorting criterion; in contrast with sequence containers, you have no semantics to put an element into a specific position.

1.8.1 Implementation

All standardized unordered container classes are implemented as hash tables, which nonetheless still have a variety of implementation options.

1.8.2 Abilities

1. The hash tables use the "chaining" approach, whereby a hash code is associated with a linked list. (This technique, also called "open hashing" or "closed addressing," should not be confused with "open addressing" or "closed hashing.")
2. Whether these linked lists are singly or doubly linked is open to the implementers. For this reason, the standard guarantees only that the iterators are "at least" forward iterators.
3. Various implementation strategies are possible for rehashing:
   - With the traditional approach, a complete reorganization of the internal data happens from time to time as a result of a single insert or erase operation.
   - With incremental hashing, a resizing of the number of bucket or slots is performed gradually, which is especially useful in real-time environments, where the price of enlarging a hash table all at once can be too high.
4. Unordered containers allow both strategies and give no guarantee that conflicts with either of them.

y. For each value to store, the hash function maps it to a bucket (slot) in the hash table. Each bucket manages a singly linked list containing all the elements for which the hash function yields the same value.

The major advantage of using a hash table internally is its incredible running-time behavior. Assuming that the hashing strategy is well chosen and well implemented, you can guarantee amortized constant time for insertions, deletions, and element search ("amortized" because the occasional rehashing happens that occurs can be a large operation with a linear complexity).

The expected behavior of nearly all the operations on unordered containers, including copy construction and assignment, element insertion and lookup, and equivalence comparison, depends on the quality of the hash function. If the hash function generates equal values for different elements, which also happens if an unordered container that allows duplicates is populated with equivalent values or keys, any hash table operation results in poor runtime performance. This is a fault not so much of the data structure itself but rather of its use by unenlightened clients

1.8.3 Disadvantages

- Unordered containers don't provide operators <, >, <=, and >= to order multiple instances of these containers. However, == and != are provided (since C++11).
- lower_bound() and upper_bound() are not provided.
- Because the iterators are guaranteed only to be forward iterators, reverse iterators, including rbegin(), rend(), crbegin(), and crend(), are not supported, and you can't use algorithms that require bidirectional iterators, or at least this is not portable

Because the (key) value of an element specifies its position — in this case, its bucket entry — you are not allowed to modify the (key) value of an element directly. Therefore, much as with associative containers, to modify the value of an element, you must remove the element that has the old value and insert a new element that has the new value

- Unordered containers don't provide operations for direct element access.
- Indirect access via iterators has the constraint that, from the iterator's point of view, the element's (key) value is constant.

1.9 STL Containers: Implementations

- **std::vector**
  - Implemented as a dynamically resizable array with contiguous memory.
- **std::deque**
  - Implemented as a sequence of dynamically allocated arrays (blocks) for efficient insertion/removal at both ends.
- **std::list**
  - Implemented as a doubly linked list, where each node contains pointers to the previous and next nodes.
- **std::forward_list**
  - Implemented as a singly linked list, where each node contains a pointer to the next node.
- **std::set / std::multiset**
  - Implemented as a self-balancing binary search tree (typically Red-Black Tree).
- **std::unordered_set / std::unordered_multiset**
  - Implemented as a hash table with separate chaining or open addressing for collision resolution.
- **std::map / std::multimap**
  - Implemented as a self-balancing binary search tree (typically Red-Black Tree) for sorted key-value pairs.
- **std::unordered_map / std::unordered_multimap**
  - Implemented as a hash table with separate chaining or open addressing for key-value pairs.

1.10 STL Containers: Iterator Functions

- **Containers with all the iterator functions (begin(), end(), cbegin(), cend(), rbegin(), rend(), crbegin(), crend())**:

  1. Vector
  2. Deque
  3. List
  4. Set
  5. Muliset
  6. Map
  7. Mulimap
  8. Unordered set
  9. unordered multiset
  10. unordered map
  11. unordered multimap

- **Containers with limited iterator support:**

  1. **Forward_list**: Only supports forward iterators (begin(), end(), cbegin(), cend()).

1.11 STL containers: Main concepts, differences, uses

- **Vectors**:
  - Dynamic array, automatic resizing.
  - We have access to capacity and reserve methods.
  - Fast at end operations.
  - Contiguous memory, random access.
  - at() method to index with error checking.
- **Deque**:
  - Multiple blocks / Dynamic arrays to give access to both ends.
  - Fast at both ends.
  - Front and back operations.
  - Slower iterator access compared to vectors.
  - Iterators are smart pointers.
  - No capacity access
- **List:**
  - Doubly-linked list
  - Insertion and removing is fast
  - Access at any element thats not the first or last is slow.
  - NO random access
  - Member method to sort
  - Splice
  - Unique
  - Merge
- **Forward_list**
  - Singly linked list
  - No size method
  - No reverse iterators
  - No pointer to last element, no back(), push_back(), or pop_back() methods
  - For all member functions that modify forward lists in a way that elements are inserted or deleted at a specific position, special versions for forward lists are provided. The reason is that you have to pass the position of the element before the first element that gets manipulated, because there you have to assign a new successor element. Because you can't navigate backwards (at least not in constant time), for all these member functions you have to pass the position of the preceding element. Because of this difference, these member functions have a after suffix in their name. For example, instead of insert(), insert_after() is provided, which inserts new elements after the element passed as first argument; that is, it appends an element at that position.

For this reason, forward lists provide before_begin() and cbefore_begin(), which yield the position of a virtual element before the first element (technically speaking, the anchor of the linked list), which can be used to let built-in algorithms ending with after exchange even the first element

- **Sets and multisets**
  - Height balanced bst
  - No duplicates in set, can have duplicates in multiset
  - logarithmic searching
  - logarithmic insertion and deletion
  - Automatic sorting
  - Can't change elements directly
  - No direct element access
  - Constant iterators

1.12 STL Containers: Iterator invalidation

- **Vectors:**
  - **Insertion:** All iterators are invalidated if a reallocation occurs; otherwise, only iterators at or after the point of insertion are invalidated.
  - **Deletion:** Iterators at or after the point of deletion are invalidated.
- **Deque**:
  - **Insertion/Deletion:** At beginning or end, no invalidation unless reallocation occurs. Inserting or deleting in the middle invalidates all iterators.
- **List**:
  - **Insertion:** No invalidation.
  - **Deletion:** Only the iterator to the erased element is invalidated.
- **Forward list**:
  - **Insertion:** No invalidation.
  - **Deletion:** Only the iterator to the erased element is invalidated.
- **Set/multiset**:
  - **Insertion:** No invalidation.
  - **Deletion:** Only the iterator to the erased element is invalidated.
- **unordered set/unordered multiset**:
  - **Insertion:** No invalidation unless rehashing occurs.
  - **Deletion:** Only the iterator to the erased element is invalidated.
  - **Rehashing:** All iterators are invalidated.
- **Map/Multimap**:
  - **Insertion:** No invalidation.
  - **Deletion:** Only the iterator to the erased element is invalidated.
- **Unordered map/unordered multimap**:
  - **Insertion:** No invalidation unless rehashing occurs.
  - **Deletion:** Only the iterator to the erased element is invalidated.
  - **Rehashing:** All iterators are invalidated.

1.13 STL Containers: Reallocation

- **Vectors**: Reallocation occurs when inserting elements exceeds the current capacity.
- **Deque**: Reallocation occurs when inserting elements requires more blocks (typically at both ends, but can happen internally).
- **List, forward list**: No reallocation occurs, as they allocate nodes dynamically and do not store elements contiguously.
- **set, multiset, map, multimap**: No reallocation occurs, as they use balanced trees, and elements are not stored contiguously.
- **unordered set, unordered multiset, unordered map, unordered multimap**: Reallocation occurs when the load factor exceeds a threshold, triggering a rehash to a larger bucket array.

1.14 STL Containers: Element access

- **std::vector**
  - Direct access via index: `v[i]`, `v.at(i)`
  - Front element: `v.front()`
  - Back element: `v.back()`
- **std::deque**
  - Direct access via index: `d[i]`, `d.at(i)`
  - Front element: `d.front()`
  - Back element: `d.back()`
- **std::list**
  - No direct access via index.
  - Front element: `l.front()`
  - Back element: `l.back()`
- **std::forward_list**
  - No direct access via index.
  - Front element: `fl.front()`
- **std::set** / **std::multiset**
  - No direct access via index.
  - Access via iterator or functions like `find()`, `lower_bound()`, `upper_bound()`.
- **std::unordered_set** / **std::unordered_multiset**
  - No direct access via index.
  - Access via iterator or `find()`.
- **std::map** / **std::multimap**
  - Access by key: `m[key]` (for `std::map` only, not `std::multimap`).
  - Access via iterator or functions like `find()`, `lower_bound()`, `upper_bound()`.
- **std::unordered_map** / **std::unordered_multimap**
  - Access by key: `um[key]` (for `std::unordered_map` only, not `std::unordered_multimap`).
  - Access via iterator or `find()`.

1.15 STL Containers: Uses and advantages

- **std::vector**
  - Advantages: Fast random access, contiguous memory, efficient for dynamic arrays.
  - Uses: When frequent random access and dynamic resizing are needed.
- **std::deque**
  - Advantages: Fast insertion/removal at both ends, efficient dynamic array.
  - Uses: Double-ended queue operations, efficient at both front and back.
- **std::list**
  - Advantages: Constant time insertion/removal anywhere, no reallocation.
  - Uses: When frequent insertions/removals in the middle are needed.
- **std::forward_list**
  - Advantages: Singly linked list, smaller memory overhead, constant time insertion/removal.
  - Uses: Memory-constrained environments, where only forward traversal is needed.
- **std::set / std::multiset**
  - Advantages: Sorted elements, fast lookup (logarithmic time).
  - Uses: When you need a sorted collection with unique or non-unique elements.
- **std::unordered_set / std::unordered_multiset**
  - Advantages: Fast average-time lookup (constant time), no sorting.
  - Uses: When fast lookup is needed without element ordering.
- **std::map / std::multimap**
  - Advantages: Sorted key-value pairs, fast lookup (logarithmic time).
  - Uses: Key-value pairs where keys must remain sorted.
- **std::unordered_map / std::unordered_multimap**
  - Advantages: Fast average-time lookup (constant time), no sorting.
  - Uses: Key-value pairs where fast lookup is needed without ordering.

1.16 STL Iterators

An object that iterates/navigates over elements in the container. They are essentially an abstraction of pointer

- **Some notes about iterators**:
  1. each container provides its own iterator
  2. interfaces of iterators of different containers are largely the same
  3. internal behaviors depend on the data structure of the container
- **Operations**:
  1. operator * returns the element of the current positions
  2. operator -> access a member of the element
  3. operator ++ step forward
  4. operator – step backward
  5. operator == and != whether two iterators represent the same position
  6. operator = assign an iterator
- **Important iterators**:
  1. begin() gets you the beginning of a container
  2. end() gets you just past the end
- **Iterator types**
  - **iterator**
  - **reverse_iterator**
  - **const_iterator**
  - **const_reverse_iterator**
- **Iterator categories**
  1. **Input Iterator:**
     - **Purpose:** Read-only access to elements in a single-pass manner.
     - **Operations:** Can be incremented (++), compared for equality (==), and dereferenced (*) to access elements.
  2. **Output Iterator:**
     - **Purpose:** Write-only access to elements in a single-pass manner.
     - **Operations:** Can be incremented (++) and dereferenced (*) to assign values.
  3. **Forward Iterator:**
     - **Purpose:** Read and write access to elements; can traverse the container in a single direction.
     - **Operations:** Can be incremented (++), compared for equality (==), and dereferenced (*).
  4. **Bidirectional iterator**:
     - **Purpose:** Read and write access to elements; can traverse the container in both directions.
     - **Operations:** Supports both increment (++) and decrement (–) operations.

5. **Random Access Iterator:**
   - **Purpose**: Read and write access with the ability to jump to any element in constant time.
   - **Operations:** Supports all operations of bidirectional iterators plus direct arithmetic operations like addition (+), subtraction (-), and subscript ([]).

- **Containers and their iterators**:

  1. **Vector**: Random access iterator
  2. **Deque**: Random access iterator
  3. **List**: Bidirectional iterator
  4. **Forward_list**: Forward iterator
  5. **Set**: Bidirectional iterator
  6. **Multiset**: Bidirectional iterator
  7. **Map**: Bidirectional iterator
  8. **Multimap**: Bidirectional iterator
  9. **Unordered_map**: Forward iterator

- **Insert iterators**: Insert iterators in C++ are special types of iterators that allow you to insert elements into a container at specific positions rather than overwriting existing elements. There are three primary types of insert iterators provided by the C++ Standard Library:

  if a container has an insert method, you can and often should use it directly when inserting elements, especially if you want to insert a single element or a specific range of elements into the container.

  Containers that have an insert method are: vector, deque, list, forward list, set, multiset, unordered set, unordered multiset,map, multimap, unordered map, unordered multi map.

  Insert iterators (std::back_inserter, std::front_inserter, and std::inserter) should be used when working with algorithms or situations where automatic insertion logic simplifies your code.

  Some things in <algorithm> require these inserters

  1. **std::front_inserter**: Inserts elements at the front of a container. Calls the container's push_front method to add elements to the front. Used with containers that support push_front

```cpp
#include <list>
#include <algorithm>
#include <iterator>

int main() {
    std::list<int> lst = {1, 2, 3};
    std::list<int> to_add = {4, 5, 6};

    // Insert elements at the front of lst
    std::copy(to_add.begin(), to_add.end(), std::front_inserter(lst));

    // lst now contains: 6, 5, 4, 1, 2, 3
}
```

2. **std::back_inserter**: Inserts elements at the end of a container. Inserts elements at the end of a container. Used with containers that support push_back

```cpp
#include <vector>
#include <algorithm>
#include <iterator>

int main() {
    std::vector<int> vec = {1, 2, 3};
    std::vector<int> to_add = {4, 5, 6};

    // Insert elements at the end of vec
    std::copy(to_add.begin(), to_add.end(), std::back_inserter(vec));

    // vec now contains: 1, 2, 3, 4, 5, 6
}
```

3. **std::inserter**: Inserts elements at a specific position in a container. Takes an iterator indicating the insertion position and calls the container's insert method. Used with containers that support insertion at arbitrary positions

```cpp
#include <vector>
#include <algorithm>
#include <iterator>

int main() {
    std::vector<int> vec = {1, 2, 3};
    std::vector<int> to_add = {4, 5, 6};

    // Insert elements starting at the second position (before 2)
    std::copy(to_add.begin(), to_add.end(), std::inserter(vec, vec.begin() + 1));

    // vec now contains: 1, 4, 5, 6, 2, 3
}
```

1.17 Complexity of container operations

Container	Access	Search	Insertion	Deletion
`vector`	O(1)	O(n)	O(n) (amortized O(1) at end)	O(n)
`deque`	O(1)	O(n)	O(n) (amortized O(1) at ends)	O(n)
`list`	O(n)	O(n)	O(1) (if position known), O(n) (worst case)	O(1) (if position known), O(n) (worst case)
`forward_list`	O(n)	O(n)	O(1) (if position known), O(n) (worst case)	O(1) (if position known), O(n) (worst case)
`set`	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
`multiset`	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
`map`	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
`multimap`	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
`unordered_set`	O(n)	O(n)	O(n)	O(n)
`unordered_multiset`	O(n)	O(n)	O(n)	O(n)
`unordered_map`	O(n)	O(n)	O(n)	O(n)
`unordered_multimap`	O(n)	O(n)	O(n)	O(n)

Unordered container operations are for the most part constant time operations, but worst case they are linear.

STL Algorithms

2.1 <algorithm>

2.2 <numeric>

2.2.1 transform_reduce

std::transform_reduce is a function introduced in C++17 as part of the <numeric> header. It combines the functionality of both std::transform and std::reduce (also known as std::accumulate in earlier C++ standards). This algorithm applies a transformation to elements from one or two ranges and then reduces (aggregates) the results of that transformation using a specified binary operation.

The signature of std::transform_reduce has several overloads, but it can be summarized in two forms:

2.2.1.1 Unary transform and reduce

```
template <typename InputIt, typename T, typename BinaryOp1, typename UnaryOp>
T transform_reduce(InputIt first, InputIt last, T init, BinaryOp1 binary_op, UnaryOp unary_op);
```

- **first, last:** Input range [first, last).
- **init:** Initial value for the reduction.
- **binary_op:** Binary operation to reduce the transformed values (e.g., std::plus<>() for summation).
- **unary_op:** Unary operation to transform each element before reducing

This version is for a single input range, where each element is transformed and then the transformed results are reduced.

Example

```
#include <numeric>
#include <vector>
#include <iostream>

int main() {
    std::vector<int> v = {1, 2, 3, 4, 5};
    int result = std::transform_reduce(v.begin(), v.end(), 0, std::plus<>(), [](int x) { return x * x; });
    std::cout << "Sum of squares: " << result << std::endl;
}

```

- The unary_op is the lambda function, which squares each element.
- The binary_op is std::plus<>, which adds the squares together.
- The result will be $1^2 + 2^2 + 3^2 + 4^2 + 5^2 = 55$.

2.2.1.2 Binary transform and reduce

```cpp
template <typename InputIt1, typename InputIt2, typename T, typename BinaryOp1, typename BinaryOp2>
T transform_reduce(InputIt1 first1, InputIt1 last1, InputIt2 first2, T init, BinaryOp1 binary_op, BinaryOp2 binary_transform_op);
```

- **first1, last1:** First input range [first1, last1).
- **first2**: Second input range, assumed to have at least the same length as the first.
- **init:** Initial value for the reduction.
- **binary_op:** Binary operation to reduce the transformed values.
- **binary_transform_op:** Binary operation to transform elements from both ranges before reducing.

This version is for two input ranges, where pairs of elements from both ranges are transformed and then the transformed results are reduced.

Example

```cpp
#include <numeric>
#include <vector>
#include <iostream>

int main() {
    std::vector<int> v1 = {1, 2, 3};
    std::vector<int> v2 = {4, 5, 6};
    int result = std::transform_reduce(v1.begin(), v1.end(), v2.begin(), 0, std::plus<>(), std::multiplies<>());
    std::cout << "Dot product: " << result << std::endl;
}
```

2.2.1.3 std::plus<> and std::multiplies<>

std::plus<> and std::multiplies<> are implemented as function objects

```
template<typename T=void>
struct plus {
    constexpr operator()(const T& lhs, const T& rhs) const {
        return lhs + rhs;
    }
};

template<typename T=void>
struct multiplies {
    constexpr operator()(const T& lhs, const T& rhs) const {
        return lhs * rhs;
    }
};
```

The `T = void` in function objects like std::plus and std::multiplies allows for more flexibility and generic usage.

The T = void part enables a more generic specialization for void, which allows the function object to work with mixed types (as long as the types support the required operation) or automatically deduce the argument types

By specializing for void, the compiler can automatically deduce the types of lhs and rhs. This means that you don't have to explicitly specify a type when you use std::multiplies<>. For example, you can apply std::multiplies<> to two arguments of different types (e.g., int and float), and it will work as long as the multiplication operator (*) is defined for them.

2.2.1.4 Other key function objects

- `std::plus`
- `std::minus`
- `std::multiplies`
- `std::divides`
- `std::modulus`
- `std::negate`
- `std::equal_to`
- `std::not_equal_to`
- `std::greater`
- `std::less`
- `std::greater_equal`

- `std::less_equal`
- `std::logical_and`
- `std::logical_or`
- `std::logical_not`

Function Objects

Functional arguments for algorithms don't have to be functions. As seen with lambdas, functional arguments can be objects that behave like functions. Such an object is called a function object, or functor. Instead of using a lambda, you can define a function object as an object of a class that provides a function call operator

Function objects are another example of the power of generic programming and the concept of pure abstraction. You could say that anything that behaves like a function is a function. So, if you define an object that behaves as a function, it can be used like a function.

So, what is the behavior of a function? A functional behavior is something that you can call by using parentheses and passing arguments

All you have to do is define operator () with the appropriate parameter types:

```
class c {
    public:
    void operator() (int x) {
        cout << "The value is: " << x << endl;
    }
};

c c1;
c1(2); // Call the object as a function

vector<int> v({1,2,3});
std::for_each(b(v), e(v), c()); // Use function object for algorithms
```

3.0.0.1 Why?

You may be wondering what all this is good for. You might even think that function objects look strange, nasty, or nonsensical. It is true that they do complicate code. However, function objects are more than functions, and they have some advantages:

1. Function objects are "functions with state." Objects that behave like pointers are smart pointers. This is similarly true for objects that behave like functions: They can be "smart functions" because they may have abilities beyond operator (). Function objects may have other member functions and attributes. This means that function objects have a state. In fact, the same functionality, represented by two different function objects of the same type, may have different states at the same time. This is not possible for ordinary functions. Another advantage of function objects is that you can initialize them at runtime before you use/call them.

2. Each function object has its own type. Ordinary functions have different types only when their signatures differ. However, function objects can have different types even when their signatures are the same. In fact, each functional behavior defined by a function object has its own type. This is a significant improvement for generic programming using templates because you can pass functional behavior as a template

parameter. Doing so enables containers of different types to use the same kind of function object as a sorting criterion, ensuring that you don't assign, combine, or compare collections that have different sorting criteria. You can even design hierarchies of function objects so that you can, for example, have different, special kinds of one general criterion.

3. Function objects are usually faster than ordinary functions. The concept of templates usually allows better optimization because more details are defined at compile time. Thus, passing function objects instead of ordinary functions often results in better performance.

3.0.0.2 Predefined function objects

- LESS<T>
- GREATER<T>

We can use these for example, in constructor of set:

```
set<ELEM> C // A SET THAT SORTS WITH LESS<>
 set<ELEM, OPERATION> C // A SET THAT SORTS WITH OP
```

Examples:

- SET<INT> S1; // INTEGERS ARE SORTED BY <
- SET<INT, GREATER<INT> > S2; // INTEGERS ARE SORTED BY >

Templates

Templates in C++ are a powerful feature that allows writing generic and reusable code. They enable functions and classes to operate with different data types without being rewritten for each specific type.

4.1 Template Function

A template function defines a family of functions that work with different data types. Here's how to write and use a template function:

```cpp
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    std::cout << add<int>(3, 4) << std::endl; // Instantiates add<int>
    std::cout << add<double>(2.5, 3.1) << std::endl; // Instantiates add<double>
    return 0;
}
```

4.2 Template Class

```cpp
template <typename T>
class Stack {
    std::vector<T> data;

public:
    void push(T value) {
        data.push_back(value);
    }
    void pop() {
        data.pop_back();
    }
    T top() const {
        return data.back();
    }
    bool empty() const {
        return data.empty();
    }
};

int main() {
    Stack<int> intStack; // Instantiates Stack<int>
    intStack.push(10);
    intStack.push(20);
    std::cout << intStack.top() << std::endl;

    Stack<double> doubleStack; // Instantiates Stack<double>
    doubleStack.push(1.1);
    doubleStack.push(2.2);
    std::cout << doubleStack.top() << std::endl;
    return 0;
}
```

4.3 Class vs typename keyword

The choice between using **class** and **typename** in template declarations in C++ is largely a matter of style and historical context, as both keywords serve the same purpose

4.4 Handle friend functions

4.4.1 Friendship to a Non-Template Function

This is straightforward. You directly declare a non-template function as a friend inside your template class. This grants that specific function access to all instances of the template class, regardless of the type parameter.

```cpp
template <typename T>
class MyClass {
    friend void someFunction(MyClass<T>&);
};
```

4.4.2 Friendship to a Template Function

More commonly, you want a template function to be a friend to a template class. This allows each instantiation of the function template to access the corresponding instantiation of the class template.To achieve this, you need to forward declare the function template and then declare it as a friend inside your class template. The tricky part is that the syntax for declaring a template function as a friend inside a template class can vary based on what you're trying to achieve:

How to forward declare:

```cpp
    template<typename T>
    class myclass;

    template<typename T>
    void foo(myclass<T>&);


    template<typename T>
    class myclass {

    public:
        friend void foo <T>(const myclass<T>& obj);
    };

template <typename T>
void foo(myclass<T>& obj) {
    // Define
}
```

Different types:

- More general form used when you want the friendship to apply to all instantiations of the function template

```
template <typename T>
class MyClass {
    friend void someFunction<>(MyClass<T>&); // Specific instantiation
};
```

- All instantiations of the function template are friends:

```
template <typename T>
class MyClass {
    template <typename U>
    friend void someFunction(MyClass<U>&); // All instantiations
};
```

- This form ties the friendship to the specific template instantiation of both MyClass and someFunction using the same template argument T. The function template that takes the same template parameters:

```
template <typename T>
class MyClass {
    friend void someFunction<T>(MyClass<T>&); // Matched instantiation
};
```

4.5 Function Template Specialization

Concept 1: **Template specialization** allows you to define a different implementation for a particular data type.
□

```
template <typename T>
T min(T x, T y) {
    return (x < y) ? x : y;
}

template <>
const char* min(const char* x, const char* y) {
    return (strcmp(x, y) < 0) ? x : y;
}
```

4.6 Class/Struct Template Specialization

```
template<typename T>
struct foo {
    T x = 20;
};

template<>
struct foo<char> {
    char x = 'z';
};
```

4.7 Template Parameters

Concept 2: Templates can have more than one parameter, including non-type parameters.
□

```
template <typename T, int size>
class FixedArray {
    private:
    T arr[size];
    // implementation
};
```

4.8 Trailing return type

In traditional C++, the return type of a function is declared at the beginning of the function declaration. However, C++11 introduced a new syntax that allows the return type to be specified after the parameter list, using auto at the beginning and -> Type after the parameter list.

4.8.1 Syntax

```
auto functionName(parameters) -> returnType {
    // function body
}
```

4.8.2 Example

```
auto foo(int a, int b) -> int {
    return a + b;
}
```

4.9 decltype

Concept 3: **decltype** is a keyword in C++ introduced in C++11, which stands for "declared type". It is used to query the type of an expression without actually evaluating that expression. This can be particularly useful in template programming and type deduction, where the type of an expression might not be known until compile time.
□

4.9.1 Syntax

```
decltype(expression) variable_name;
```

Here, **variable_name** will have the same type as the type of **expression**. It's important to note that **expression** is not evaluated; **decltype** only deduces its type.

4.9.2 Example

```
int a = 5;
decltype(a) b = 5;

cout << typeid(b).name() << endl; // Output: i
```

4.10 Template functions with mixed types (Trailing return type)

Concept 4: To address the challenge of determining the return type for a template function that accepts two different types, we can utilize a strategy involving **auto** and a **trailing return type** with **decltype**. This approach effectively resolves the ambiguity of the return type in such template functions.
□

```
template<typename T, typename U>
auto add(T t, U u) -> decltype(t + u) {
    return t + u;
}
```

4.11 Template functions with mixed types (Deduced return type)

Alternatively, C++14 introduced the concept **deduced return type**. Which provides a simpler way to handle the situation described above

```
template<typename T, typename U>
decltype(auto) foo(T a, U b)  {
    return a + b;
}
```

4.12 Dependent name resolution

Suppose we have the code

```
template <typename T>
void showcont(const T& cont) {
    typename T::iterator it;
    for (it = cont.begin(); it! = cont.end(); ++it)
        cout << *it << endl;
}
```

Why do we need the word ”`typename`” before the iterator `it` is declared?

This concept is called ”**dependent name resolution**” in C++. Specifically, it falls under the broader topic of dependent types and dependent names in template programming.

4.12.1 Dependent names

In templates, a dependent name is any name that depends on a template parameter. For example, `T::iterator` in the code above depends on the template parameter `T`. The compiler cannot determine the exact meaning of T::iterator until the template is instantiated with a concrete type.

4.12.2 Typename Keyword

In C++, the typename keyword is used to indicate that a dependent name refers to a type. Without typename, the compiler might interpret T::iterator as something else, such as a static member variable or a constant. Using typename helps disambiguate and ensures the compiler treats T::iterator as a type.

Dependent name resolution is the process by which the compiler determines the meaning of dependent names during template instantiation. When the template is instantiated with a specific type, the compiler resolves T::iterator to the actual nested type within T.

C++ uses a two-phase name lookup process for templates. In the first phase, the compiler parses the template without knowing the actual template arguments. In the second phase, the compiler instantiates the template with the provided arguments. Since T::iterator cannot be resolved in the first phase (before knowing what T actually is), typename is required to instruct the compiler that iterator is indeed a type

4.12.3 Nested types

Any nested type within a template parameter requires the typename keyword. For example:

```
template <typename T>
void foo() {
    typename T::value_type val; // T::value_type is a nested
    ↪ type within T
}
```

4.12.4 Prereq: Using aliases defined in classes

We can define aliases inside classes, but then we need to reference them outside the class by using the *scope resolution operator*

```
class foo {
public:
    alias ll = long long;
    ll b = 10;
 };

 int main() {
     foo::ll a = 5;
     return 0;
 }
```

This also applies to typedefs...

```
class foo {
public:
    typedef long long ll;
}

int main() {
    foo::ll a = 10;
    return 0;
}
```

Note:-

We can define aliases inside functions, but we can't use them outside the function they are defined in

4.12.5 Type Aliases

If a class template has a type alias defined inside it, you need to use typename when referring to that alias in another template. For example:

```cpp
template <typename T>
class Wrapper {
    public:
    using PointerType = T*; // A type alias within Wrapper
};

template <typename T>
void bar() {
    typename Wrapper<T>::PointerType ptr; // typename is required here
}
```

4.12.6 Return Types in Template Functions

When using a dependent type as a return type for a function within a template, typename is necessary:

```
template <typename T>
typename T::value_type getValue(const T& container) {
    return *container.begin();
}
```

4.12.7 Base Class Members

When accessing a member of a base class that is a dependent type, typename is required.

```
template <typename T>
class foo {
public:
    typedef T* tp;
};

template <typename T>
class bar : public foo<T> {
public:
    typename foo<T>::tp a;
};
```

4.12.8 Dependent Types in Expressions

If you use a dependent type within an expression, you must indicate it with typename:

```
template <typename T>
void example(const T& cont) {
    typename T::size_type size = cont.size(); // typename
    ↪ required here
}
```

Where size_type is a type alias defined inside the class or container type T. In this context, size_type typically represents an unsigned integer type used to express sizes and counts of elements in the container.

Lambdas

5.1 Auto in lambda args

5.2 Template lambdas

When initializer lists are required

Using initialization lists to initialize data members in a constructor can be convenient if you don't need to do any error-checking on the constructor arguments. There are also several instances in C++ where the use of an initializer list to initialize a data member is actually required:

- Data members that are const but not static must be initialized using an initialization list.
- Data members that are references must be initialized using an initialization list.
- An initialization list can be used to explicitly call a constructor that takes arguments for a data member that is an object of another class (see the employee constructor example above).
- In a derived class constructor, an initialization list can be used to explicitly call a base class constructor that takes arguments.

Inheritence and Subtype Polymorphism

7.1 OOP Main Concepts

An **object** is a software bundle of related state (data members or properties) and behavior (member functions or methods). Software objects are often used to model the real-world objects that you find in everyday life.

A **class** is a blueprint or prototype from which objects are created. A class is an abstract definition that is made concrete at run-time when objects based upon the class are created.

Encapsulation, also known as data hiding, is the act of concealing the functionality of a class so that the internal operations are hidden, and irrelevant, to the programmer. With correct encapsulation, the developer does not need to understand how the class actually operates in order to communicate with it via its publicly available member functions and data members, known as its public interface. Encapsulation is essential to creating maintainable object-oriented programs. When the interaction with an object uses only the publicly available interface of member functions and properties, the class of the object becomes a correctly isolated unit. This unit can then be replaced independently to fix bugs, to change internal behavior or to improve functionality or performance. Encapsulation also promotes data integrity by allowing public ”set” member functions to validate new values that are to be assigned to private data members.

Message passing, also known as interfacing, describes the communication between objects using their public interfaces. The primary way of passing a message to an object in C++ is to call a member function for that object.

Abstraction is the process of representing simplified versions of real-world objects in your classes and objects. A car class does not describe every possible detail of a car, only the relevant parts for the system being developed. Modeling software around real-world objects can vastly reduce the time required to understand a solution and be able to develop and maintain it.

7.2 Object Relationships

Objects can work together in many ways within a system. In some situations, classes and objects can be tightly coupled together to provide more complex functionality. This ”has-a” relationship is known as composition. For example, modeling a car might involve creating individual classes such as wheel, engine, and transmission. The car class could then contain objects of these classes as data members, since a car ”has” an engine, wheels, and a transmission. The internal workings of each class are not important due to encapsulation as the communication between the objects is still via passing messages to their public interfaces.

Other types of relationships may be modeled. A class may simply ”use” an object of another class (perhaps creating the object as a local variable in one of its member functions). A class may also ”know” about an object of another class without owning it (in C++, this association relationship might be modeled using a pointer or reference to the object).

Inheritance is an object-oriented programming concept used to model an ”is-a” relationship between two classes. It allows one class (the derived class or subclass) to be based upon another (the base class or superclass) and inherit all of its functionality automatically. Additional code may then be added to create a more specialized version of the base class.

7.3 Ineritance

A **derived class** is more specific than its base class and represents a smaller group of objects.

A **direct base class** is the base class from which a derived class explicitly inherits. An indirect base class is inherited from two or more levels up the class hierarchy.

In the case of single inheritance, a class is derived from one base class. C++ also supports multiple inheritance, in which a derived class inherits from multiple (possibly unrelated) classes. Single inheritance is straightforward. Multiple inheritance can be complex and error prone.

Single-inheritance relationships form tree-like hierarchical structures - a base class exists in a hierarchical relationship with its derived classes.

C++ offers three kinds of inheritance - public, protected, and private. public inheritance in C++ is used to model "is a" relationships. Every object of a derived class is also an object of that derived class's base class. However, base-class objects are not objects of their derived classes. For example, all car objects are also vehicle objects, but not all vehicle objects are car objects.

With public inheritance, a derived class may

- add new data members
- add new member functions
- override member functions defined in the base class

private and protected inheritance do not model "is-a" relationships and are not used as frequently.

7.4 Inheritance and Member Access

Base class modifier	public Inheritance	protected Inheritance	private Inheritance
public	public	protected	Hidden
protected	protected	protected	Hidden
private	Hidden	Hidden	Hidden

A base class's public members are accessible anywhere that the program has a "handle" to an object (an object name or a pointer or reference to an object) of the base class or to an object of one of that base class's derived classes. Derived class member functions can access public base class data members directly.

A base class's private members are "hidden" - they are accessible only within the definition of that base class or from a friend of that class. A derived class cannot access the private members of its base class directly; allowing this would violate the encapsulation of the base class. A derived class can only access private base-class members through non-private member functions defined in the base class and inherited by the derived class.

A base class's protected members have an intermediate level of protection between public and private access. A base class's protected members can be directly accessed by member functions of that base class, by a friend of that base class, by member functions of a class derived from that base class, and by a friend of a class derived from that base class.

The use of protected data members allows for a slight increase in performance, because we avoid incurring the overhead of a call to a "set" or "get" member function. Unfortunately, protected data members often yield two major problems. First, the derived class object does not have to use a "set" member function to change the value of the base class's protected data. A derived class object can easily assign an illegal value to a protected data member. Second, derived class member functions are more likely to depend on base class implementation details. Changes to the base class may require changes to some or all of the derived classes of that base class.

Declaring data members private, while providing non-private member functions to manipulate and perform validation checking on this data, enforces good software engineering. The programmer should be able to change the base class implementation freely, while still providing the same services to the derived class. The performance increases gained by using protected data members are often negligible compared to the optimizations that compilers can perform. It is appropriate to use the protected access modifier when a base class should provide a service (i.e., a member function) only to its derived classes and should not provide the service to other clients.

When a base class member function is inappropriate for a derived class, that member function can be redefined in the derived class with an appropriate implementation. This is called overriding the base class member function.

When a derived class member function overrides a base class member function, the base class member function can still be accessed from the derived class by preceding the base class member function name with the base class name and the scope resolution operator (::).

When an object of a derived class is created, the base class's constructor is called immediately (either explicitly or implicitly) to initialize the base class data members in the derived class object (before the derived class data members are initialized). Explicitly calling a base class constructor requires using the same special "member initialization list syntax" used with composition and const data members.

When a derived class object is destroyed, the destructors are called in the reverse order of the constructors - first the derived class destructor is called, then the base class destructor is called.

7.5 Inheritance Syntax

To declare a derived class:

```cpp
// car is a derived class of vehicle.
class car : public vehicle
{
    // Car data members and member functions
};
```

A constructor initialization list can be used to pass arguments from a derived class constructor to a base class constructor:

```
// Pass the string color to the base class vehicle constructor.
car::car(const string& color, int num_doors) : vehicle(color)
{
    this->num_doors = num_doors;
}
```

A derived class member function that overrides a base class member function can call the base class version of the function to do part of its work:

```
void car::print() const
{
    vehicle::print();   // Call the vehicle version of print() to print the car's color.
    cout << num_doors;
}
```

7.6 Upcasting and Downcasting

Upcasting is converting a derived class pointer (or reference) to a pointer (or reference) of the derived class's base class. In other words, upcasting allows us to treat a derived type as though it were its base type. It is always allowed for public inheritance, without an explicit type cast. This is a result of the "is-a" relationship between the base and derived classes. For example, if car is a class derived from vehicle, the following code is legal:

```
vehicle* vptr = new car();
```

The car object does not actually become a vehicle object as a result of this type cast (in a sense, it already is one). However, the vehicle pointer can only be used to access parts of the car object that are defined in the vehicle class. For example, you can only call member functions that are defined in the vehicle class. The car object is treated like any other vehicle, and its car-specific data members and member functions are unavailable.

```
vehicle* vptr = (vehicle*) new car();
vehicle* vptr = dynamic_cast<vehicle*>(new car()); // Using c++ casting
vehicle* vptr = static_cast<vehicle*>(new car()); // Using c++ casting
```

The opposite process, converting a base class pointer (or reference) to a derived class pointer (or reference) is called **downcasting**. Downcasting is not allowed without an explicit type cast. The reason for this restriction is that the "is-a" relationship is not always symmetric. A car is a vehicle, but a vehicle may or may not be a car. For example:

```cpp
car c1;

vehicle* vptr = &c1;          // Upcast - no type cast required.

car* car_ptr = (car*) vptr;   // Downcast - type cast required.
```

The code shown above works, because the object pointed to by vptr actually is a car object. If it wasn't, the results could lead to an unsafe operation.

```cpp
bus b1;                       // Assume bus is also a derived class of vehicle.

vehicle* vptr = &b1;          // Upcast - no type cast required.

car* car_ptr = (car*) vptr;   // Downcast - type cast required. Fails because vptr
                                // points to a bus, not a car.
```

C++ provides a special explicit cast called **dynamic_cast** that allows for safe downcasting. If the type cast fails, it will return nullptr rather than crashing your program:

```cpp
car* carptr = dynamic_cast<car*>(vptr);
if (carptr != nullptr)
{
    // Type cast succeeded, vptr was pointing to a car object
    // Can now safely call car-specific member functions using carptr
}
```

7.7 More on Downcasting

Downcasting is the process of converting a base class pointer or reference to a derived class pointer or reference

- Downcasting is potentially unsafe, so it requires an explicit cast.
- dynamic_cast should be used for safe downcasting to check if the cast is valid at runtime.
- Downcasting typically requires polymorphic classes with virtual functions to enable dynamic_cast.

In C++, the dynamic_cast operator, used for safe downcasting, requires that the base class has at least one virtual function. This is because dynamic_cast relies on runtime type information (RTTI), which is only available for polymorphic classes.

- **RTTI Availability:**
  - RTTI is used to store type information at runtime, which dynamic_cast uses to determine the exact type of the object being cast.
  - RTTI is only generated by the compiler for polymorphic classes, which are classes that have at least one virtual function.
- **Polymorphic Behavior:** Virtual functions enable polymorphic behavior, allowing derived classes to override base class functions. This is the essence of polymorphism, which dynamic_cast utilizes to ensure safe downcasting.
- **Checking Actual Type:** The type information stored in RTTI allows dynamic_cast to check if the object being cast is indeed of the target derived type. If not, it returns nullptr (for pointers) or throws a std::bad_cast exception (for references).

7.7.1 What Happens Without Virtual Functions

- **No RTTI:** If the base class has no virtual functions, it is not polymorphic, and the compiler doesn't generate RTTI for it. Without RTTI, dynamic_cast cannot validate types at runtime.
- **Compile-Time Error:** Attempting to use dynamic_cast on a class without virtual functions will lead to a compile-time error indicating that the class type is not polymorphic.

7.7.2 Downcasting example

```cpp
#include <iostream>

class Base {
    public:
    virtual ~Base() = default; // Make the class polymorphic
};

class Derived1 : public Base {
    public:
    void func1() {
        std::cout << "Derived1 Function\n";
    }
};

class Derived2 : public Base {
    public:
    void func2() {
        std::cout << "Derived2 Function\n";
    }
};

int main() {
    // Case 1: Valid downcast, will not get nullptr
    Base* basePtr1 = new Derived1(); // Base pointer to Derived1
    Derived1* derived1Ptr = dynamic_cast<Derived1*>(basePtr1);
    if (derived1Ptr) {
        derived1Ptr->func1(); // This will execute correctly
    } else {
        std::cout << "Downcast to Derived1 failed\n";
    }

    // Case 2: Invalid downcast, will get nullptr
    Base* basePtr2 = new Derived2(); // Base pointer to Derived2
    Derived1* derived1PtrInvalid = dynamic_cast<Derived1*>(basePtr2);
    if (derived1PtrInvalid) {
        derived1PtrInvalid->func1(); // This will not execute
    } else {
        std::cout << "Downcast to Derived1 failed\n"; // This will be printed
    }

    // Clean up
    delete basePtr1;
    delete basePtr2;

    return 0;
}
```

Note:-

When a Base* points to a Base object and we attempt to downcast it to a derived type using dynamic_cast, the cast will fail and return nullptr. This is because the actual type of the object being pointed to is Base, not the derived type.

7.7.3 Base class pointer example

```
class base {

public:
    virtual void print() const {
        cout << "Base class" << endl;
    }

};


class derived : public base {
    void print() const override {
        cout << "Child class" << endl;
    }
};


int main(int argc, char* argv[]) {

    base* bptr = new derived();

    bptr->print(); // Child class
    bptr->base::print(); // Base class

    return EXIT_SUCCESS;
}
```

Note:-

Notice we are able to call the private method, this is because the virtual function mechanism directs the call to the most derived method.

7.8 Object Slicing

Object slicing occurs when a derived class object is assigned or copied to a base class object, causing the derived class's specific members to be "sliced off."

```
base b1 = base();
derived d1 = derived();

b1 = d1; // Slicing
d1 = b1; // Does not work
```

7.9 Multiple Inheritance

Multiple inheritance is a feature in C++ that allows a derived class to inherit from more than one base class.

7.9.1 Why Use Multiple Inheritance?

- **Combining Functionality:** When a derived class needs to combine the functionalities of multiple base classes.
- **Mixins:** Allows implementing mixins, which are small base classes that provide specific functionalities to derived classes.
- **Interface Implementation:** Multiple inheritance can also be used to implement interfaces (abstract base classes) in C++.

7.9.2 Example

```cpp
#include <iostream>

class Base1 {
public:
    void func1() {
        std::cout << "Base1 Function" << std::endl;
    }
};

class Base2 {
public:
    void func2() {
        std::cout << "Base2 Function" << std::endl;
    }
};

class Derived : public Base1, public Base2 {
    // Derived class inherits from both Base1 and Base2
public:
    void func3() {
        std::cout << "Derived Function" << std::endl;
    }
};

int main() {
    Derived d;
    d.func1(); // Inherited from Base1
    d.func2(); // Inherited from Base2
    d.func3(); // Own function

    return 0;
}
```

7.9.3 Issues with Multiple Inheritance

1. **Ambiguity:**

   - If two base classes have the same method or attribute name, calling it from the derived class can cause ambiguity.
   - This can be resolved using the scope resolution operator :: to specify which base class method to call.

2. **Diamond Problem:**

   - If two base classes inherit from the same class and a derived class inherits from both base classes, it leads to ambiguity in the derived class about which base class's implementation to use.
   - This can be addressed using virtual inheritance to ensure only one copy of the shared base class exists.

7.10 Virtual inheritance

Virtual inheritance in C++ is a mechanism to prevent multiple "copies" of a base class when using multiple inheritance. It ensures that the derived class has only one shared instance of the base class, thus preventing ambiguity and redundant base class objects.

7.10.1 The Diamond Problem

The diamond problem occurs when two classes inherit from the same base class and another class inherits from those two classes. This results in ambiguity and multiple copies of the shared base class.

```cpp
#include <iostream>

class Base {
    public:
    int data;
    Base() : data(0) {}
};

class Derived1 : public Base {};

class Derived2 : public Base {};

class FinalDerived : public Derived1, public Derived2 {};

int main() {
    FinalDerived obj;

    // Error: Ambiguity, which Base::data to access?
    // obj.data = 10;
    // Explicit resolution:
    obj.Derived1::data = 10;
    obj.Derived2::data = 20;

    std::cout << obj.Derived1::data << ", " <<
    ↪ obj.Derived2::data << std::endl;

    return 0;
}
```

There are two separate instances of the Base class, one inherited through Derived1 and one through Derived2 This leads to ambiguity and multiple instances.

7.10.2 Solution with Virtual Inheritance

Virtual inheritance addresses this problem by ensuring that the shared base class is inherited virtually. This makes the derived class have a single, shared instance of the base class.

```
#include <iostream>
class Base {
public:
    int data;
    Base() : data(0) {}
};

class Derived1 : virtual public Base {};

class Derived2 : virtual public Base {};

class FinalDerived : public Derived1, public Derived2 {};

int main() {
    FinalDerived obj;

    // No ambiguity, only one instance of Base exists
    obj.data = 10;

    std::cout << obj.data << std::endl;

    return 0;
}
```

7.11 Subtype Polymorphism

The term binding means matching a function or member function call to a function or member function definition.

In C++, binding normally takes place when the program is compiled and linked. This is referred to as early binding or static binding.

In object-oriented programming, subtype polymorphism refers to the ability of objects belonging to different types to respond to member function calls of the same name, each one according to an appropriate type-specific behavior. The calling code does not have to know the exact type of the called object; which member function definition is called is determined at run-time (this is called late binding or dynamic binding).

In order for dynamic binding to take place in C++, several conditions must be met:

1. The call must be to a member function, not a standalone function. Function calls in C++ always use static binding.
2. The member function must have been declared using the keyword virtual. Calls to non-virtual member functions always use static binding.

3. The member function must be called through a pointer or reference to an object, not an object name. All calls to member functions (including those to virtual member functions) through object names use static binding.

With dynamic binding, C++ distinguishes between a static type and a dynamic type of a variable. The static type is determined at compile time. It's the type specified in the pointer declaration. For example, the static type of vptr is vehicle*. However, the dynamic type of the pointer is determined by the type of object to which it actually points: car* in this case. When a virtual member function is called using vptr, C++ resolves the dynamic type of vptr and ensures that the appropriate version of the member function is invoked, a process referred to as virtual dispatch.

Dynamic binding exacts a toll. Resolving the dynamic type of an object takes place at runtime and therefore incurs performance overhead. However, this penalty is negligible in most cases.

One of the most common runtime techniques for implementing virtual dispatch is a virtual member function table, or v-table. A v-table is simply an array of pointers to member functions. Each class that contains virtual member functions has a v-table. Each object that is an instance of a class with virtual member functions contains, as a hidden field, a pointer to the class's v-table. The compiler encodes a member function call as an offset into a v-table, and the appropriate v-table is used with that offset at runtime to access the correct member function.

7.12 Declaring Virtual Member Functions

Declaring Virtual Member Functions

```
virtual void print() const;
```

Note:-

A member function in a derived class that overrides a virtual member function in a base class is automatically virtual as well.

Destructors may also be virtual. You should make the destructor for your class virtual if it contains any virtual member functions.

In C++, when you overload a virtual function from a base class in a derived class, you do not necessarily need to mark the function in the derived class as virtual again for it to behave as a virtual function. It will still be virtual in any further derived classes. However, explicitly marking it as virtual in the derived class can improve code readability and make the class's design intentions clearer.

7.12.1 The override keyword

Using override: Instead of (or in addition to) marking functions as virtual in derived classes, C++11 introduced the override specifier. This ensures that the function is intended to override a virtual function in a base class. Using override helps catch errors at compile-time where the function signature does not match any virtual function in the base class, thus preventing unintended behavior.

```cpp
class Base {
    public:
    virtual void foo() { /* implementation */ }
};

class Derived : public Base {
    public:
    virtual void foo() override { /* new implementation */ }  // Using 'virtual' and 'override' for clarity
    void foo() override; // Better approach... no need for another virtual keyword
};
void Base::foo() {
    // Base class definition
}
void Derived::foo() {
    // Derived class definition
}
```

7.13 Abstract or Pure virtual Member Functions

An abstract member function is a member function that has a special prototype, but no definition. C++ refers to abstract member functions as pure virtual member functions. The prototype for a pure virtual member function ends with = 0, like this:

```cpp
virtual void earnings() const = 0;
```

Note:-

Since a pure virtual member function has no definition, you can't really call it. However, if a base class contains a pure virtual member function, a derived class is allowed to override the member function and provide a definition.

7.14 Abstract Classes

A class that contains one or more pure virtual member functions is called an abstract class (as opposed to a concrete class that provides definitions for all of its member functions).

You cannot create an object of an abstract class. However, an abstract class can be used as a base class for inheritance purposes. A class derived from an abstract class must provide definitions for any pure virtual member functions that it inherits, or it is also an abstract class.

You can also declare a pointer (or a reference) of an abstract class type. Such a pointer (or reference) would typically be used to point to a derived class object.

7.15 Interface Inheritance

Interface inheritance allows a derived class to inherit a base class's data type (which can be useful for subtype polymorphism) without actually inheriting any of the base class's implementation (member function definitions, etc.).

An interface can be defined in C++ as an abstract class that contains only pure virtual member functions and symbolic constants (public data members that are static and const).

<regex.h> Pattern Matching and String Validation

8.1 regcomp

Concept 5: Compiles a regular expression into a format that the **regexec()** function can use to perform pattern matching.
□

8.1.1 Signature

```c
int regcomp(regex_t *preg, const char *regex, int cflags)
```

- **preg:** A pointer to a regex_t structure that will store the compiled regular expression.
- **regex:** The regular expression to compile.
- **cflags:** Compilation flags that modify the behavior of the compilation. Common flags include REG_EXTENDED (use extended regular expression syntax), REG_ICASE (ignore case in match), REG_NOSUB (don't report the match), and REG_NEWLINE (newline-sensitive matching).

8.1.2 Return value

Upon successful completion, the regcomp() function shall return 0. Otherwise, it shall return an integer value indicating an error as described in <regex.h>, and the content of preg is undefined.

8.1.3 Return errors

- **REG_NOMATCH**: regexec() failed to match.
- **REG_BADPAT**: Invalid regular expression.
- **REG_ECOLLATE**: Invalid collating element referenced.
- **REG_ECTYPE**: Invalid character class type referenced.
- **REG_EESCAPE**: Trailing '\' in pattern.
- **REG_ESUBREG**: Number in "\digit" invalid or in error.
- **REG_EBRACK**: "[]" imbalance.
- **REG_EPAREN**: "\(\)" or "()" imbalance.
- **REG_EBRACE**: "\{\}" imbalance.

- **REG_BADBR**: Content of "\{\}" invalid: not a number, number too large, more than two numbers, first larger than second.
- **REG_ERANGE**: Invalid endpoint in range expression.
- **REG_ESPACE**: Out of memory.
- **REG_BADRPT**: '?', '*', or '+' not preceded by valid regular expression.

8.1.4 Flags

- **REG_EXTENDED:** Enables the use of Extended Regular Expressions (ERE) rather than Basic Regular Expressions (BRE). EREs allow a broader set of regex features, such as more flexible quantifiers and additional metacharacters without needing to escape them.
- **REG_ICASE:** Makes the pattern matching case-insensitive. This means that characters will match regardless of being in upper or lower case. For example, using REG_ICASE, the pattern "a" would match both 'a' and 'A'.
- **REG_NOSUB:** Disables the reporting of the position of matches. This flag is useful when you only need to know if a match occurred but not where it occurred. It can lead to performance improvements because the regex engine does not need to track and store the match positions.
- **REG_NEWLINE:** Alters the handling of newline characters in the text. Specifically, it:
  - Prevents the match from spanning multiple lines. The $\wedge$ and $ metacharacters will match the start and end of the input string but not the start or end of a line within the string.
  - Causes the dot . metacharacter to stop matching at a newline, which it normally would match.
  - Treats newline characters in the input as a boundary that cannot be crossed by the quantifiers $*$, $+$, ?, and $\{n\}$ unless explicitly included in a character class.
- **REG_NOTBOL and REG_NOTEOL:**
  - **REG_NOTBOL (Not Beginning Of Line):** Tells the regex engine that the beginning of the provided string should not be treated as the beginning of the line. This affects how the $\wedge$ anchor (which normally matches the start of the string) behaves. Use this if the string is a substring that does not start at the beginning of a new line.
  - **REG_NOTEOL (Not End Of Line):** Indicates that the end of the provided string should not be treated as the end of the line. This affects how the $ anchor (which normally matches the end of the string) behaves. Use this if the string is a substring that does not end at the end of a line.

8.2 Regexec

Concept 6: After compiling a regular expression, we can use regexec to match against strings.
□

8.2.1 Signature

```
int regexec(const regex\_t *preg, const char *string, size\_t
    ↪ nmatch, regmatch\_t pmatch[], int eflags)
```

- **preg:** The compiled regular expression.
- **string:** The string to match against the regular expression.
- **nmatch:** The maximum number of matches and submatches to find.
- **pmatch:** An array of regmatch_t structures that will hold the offsets of matches and submatches.
- **eflags:** Execution flags that modify the behavior of the match. A common flag is REG_NOTBOL which indicates that the beginning of the specified string is not the beginning of a line.

8.2.2 Return value

Upon successful completion, the regexec() function shall return 0. Otherwise, it shall return REG_NOMATCH to indicate no match.

8.3 Regerror

Concept 7: This function translates error codes from regcomp() and regexec() into human-readable messages.
□

8.3.1 Signature

```
size_t regerror(int errcode, const regex\_t *preg, char *errbuf,
    ↪ size\_t errbuf\_size)
```

- **errcode:** The error code returned by regcomp() or regexec().
- **preg:** The compiled regular expression (if the error is related to regexec()).
- **errbuf:** The buffer where the error message will be stored.
- **errbuf_size:** The size of the buffer.

8.3.2 Return value

Upon successful completion, the regerror() function shall return the number of bytes needed to hold the entire generated string, including the null termination. If the return value is greater than errbuf_size, the string returned in the buffer pointed to by errbuf has been truncated.

8.4 Regfree

Concept 8: Frees the memory allocated to the compiled regular expression.
□

8.4.1 Signature

```
void regfree(regex_t *preg)
```

- **preg:** The compiled regular expression to free.

8.5 regmatch_t and pmatch

8.5.1 regmatch$_t$

regmatch_t is a structure used to describe a single match (or submatch) found by regexec(). It contains at least the following two fields:

- **rm_eo:** This is the end offset of the match, which is one more than the index of the last character of the match. In other words, rm_eo - rm_so gives the length of the match.
- **rm_so:** This is the start offset of the match, relative to the beginning of the string passed to regexec(). If the match is successful, rm_so will be the index of the first character of the match.

8.5.2 pmatch array

When you call regexec(), you can pass it an array of regmatch_t structures as the pmatch argument. This array is where regexec() will store information about the matches (and submatches) it finds. The size of this array (nmatch) determines how many matches regexec() will look for and fill in. The zeroth element of this array corresponds to the entire pattern's match, and the subsequent elements correspond to parenthesized subexpressions (submatches) within the regular expression, in the order they appear.

8.6 Regex Example

```c
    regex_t regex;
    int reti;
    char msgbuf[100];
    regmatch_t pmatch[1]; // Array to store the match positions
    const char* search = "abc";

    // Compile regular expression
    reti = regcomp(&regex, "^a[[:alnum:]]", REG_EXTENDED);
    if (reti) {
        fprintf(stderr, "Could not compile regex\n");
        exit(EXIT_FAILURE);
    }

    // Execute regular expression
    // Note: Changed the third argument to 1 to indicate we want to capture up to 1 match
    // and the fourth argument to pmatch to store the match position.
    reti = regexec(&regex, search, 1, pmatch, 0);
    if (!reti) {
        printf("Match\n");
        // If you want to use the match information, you can do so here.
        // For example, to print the start and end positions of the match:
        printf("Match at position %d to %d\n", (int)pmatch[0].rm_so, (int)pmatch[0].rm_eo - 1);
    }
    else if (reti == REG_NOMATCH) {
        printf("No match\n");
    }
    else {
        regerror(reti, &regex, msgbuf, sizeof(msgbuf));
        fprintf(stderr, "Regex match failed: %s\n", msgbuf);
        exit(EXIT_FAILURE);
    }regex_t regex;
int reti;
char msgbuf[100];
regmatch_t pmatch[1]; // Array to store the match positions

// Compile regular expression
reti = regcomp(&regex, "^a[[:alnum:]]", REG_EXTENDED);
if (reti) {
    fprintf(stderr, "Could not compile regex\n");
    exit(EXIT_FAILURE);
}
```

```
// Execute regular expression
// Note: Changed the third argument to 1 to indicate we want to capture up to 1 match
// and the fourth argument to pmatch to store the match position.
reti = regexec(&regex, "abc", 1, pmatch, 0);
if (!reti) {
    printf("Match\n");
    // If you want to use the match information, you can do so here.
    // For example, to print the start and end positions of the match:
    printf("Match at position %d to %d\n", (int)pmatch[0].rm_so, (int)pmatch[0].rm_eo - 1);
}
else if (reti == REG_NOMATCH) {
    printf("No match\n");
}
else {
    regerror(reti, &regex, msgbuf, sizeof(msgbuf));
    fprintf(stderr, "Regex match failed: %s\n", msgbuf);
    exit(EXIT_FAILURE);
}

// Free the compiled regular expression
regfree(&regex);

    for (int i=(int)pmatch[0].rm_so; i<=(int)pmatch[0].rm_eo; ++i) {
        cout << search[i];
    }
    cout << endl;

    regfree(&regex);
```