**Extended CPP Notes**

**Nathan Warner**

Computer Science
Northern Illinois University
April 17, 2024
United States

# Contents


**1 STL Containers** 10

1.1 STL Vectors . . . 10

1.1.1 Implementation . . . 10

1.1.2 Performance in operations on the end . . . 10

1.1.3 Size and capacity . . . 10

1.1.4 Constructors . . . 11

1.1.5 Note about at() . . . 12

1.1.6 Iterator methods . . . 12

1.1.7 Using vectors as 2d arrays . . . 12

1.1.8 For those interested . . . 14

1.1.8.1 Manual Dynamic Memory Allocation (Using new) . 14

1.1.8.2 Using unique pointer . . . 14

1.1.8.3 Recall: Unique pointer for dynamic arry . . . 15

1.1.8.4 If sizes are truly known at compile time . . . 16

1.2 STL Deque . . . 17

1.2.1 Implementation . . . 17

1.2.2 Abilities, performance, uses . . . 17

1.2.3 When to use deques . . . 17

1.2.4 Constructors . . . 18

1.3 STL Lists . . . 19

1.3.1 Implementation . . . 19

1.3.2 Abilities . . . 19

1.3.3 Differencs in the methods . . . 19

1.3.4 Constructors . . . 20

1.3.5 Element access . . . 20

1.3.6 Iterator functions . . . 20

1.3.7 Splice Functions and Functions to Change the Order of Elements . . . 21

1.4 STL Forward lists . . . 22
1.4.1 Implementation . . . 22
1.4.2 Abilities, limitations . . . 22
1.4.3 No size()? . . . 23
1.4.4 Similarities to list . . . 23
1.4.5 Constructors . . . 24
1.5 STL Sets and multisets . . . 25
1.5.1 Implementation . . . 25
1.5.2 Strict weak ordering . . . 25
1.5.3 Abilities . . . 26
1.5.4 Changing elements directly, no direct element access . . . 26
1.5.5 Constructors . . . 26
1.5.6 Types . . . 27
1.5.7 Constructors . . . 28
1.6 STL Maps and multimaps . . . 29
1.6.1 Implementation . . . 29
1.6.2 Template parameters . . . 29
1.6.3 Abilities . . . 29
1.6.4 Constructors and types . . . 30
1.6.5 Using maps as associative arrays . . . 31
1.6.6 Constructors . . . 31
1.7 Example of bounds and equal range . . . 33
1.8 STL Unordered containers . . . 34
1.8.1 Implementation . . . 34
1.8.2 Abilities . . . 34
1.8.3 Disadvantages . . . 35
1.9 STL Containers: Implementations . . . 36
1.10 STL Containers: Iterator Functions . . . 37
1.11 STL containers: Main concepts, differences, uses . . . 38
1.12 STL Containers: Iterator invalidation . . . 40
1.13 STL Containers: Reallocation . . . 41
1.14 STL Containers: Element access . . . 42
1.15 STL Containers: Uses and advantages . . . 43

1.16 STL Iterators . . . 44

1.17 Complexity of container operations . . . 48

**2 STL Algorithms** 49

2.1 <algorithm> . . . 49

2.2 <numeric> . . . 50

2.2.1 transform_reduce . . . 50

2.2.1.1 Unary transform and reduce . . . 50

2.2.1.2 Binary transform and reduce . . . 51

2.2.1.3 std::plus<> and std::multiplies<> . . . 52

2.2.1.4 Other key function objects . . . 52

**3 Labels and goto** 54

**4 Type traits** 55

4.1 Nonstandard type of types . . . 55

4.2 ::value, ::type, and ::value_type . . . 56

**5 Function Objects** 57

5.0.0.1 Why? . . . 57

5.0.0.2 Predefined function objects . . . 58

**6 Decltype** 58

6.1 Syntax . . . 58

6.2 Example . . . 59

6.3 Things to pair with decltype . . . 59

**7 Constexpr** 60

7.1 Variables . . . 60

7.2 Functions . . . 60

7.3 Object Constructors . . . 61

7.4 constexpr vs const . . . 61

**8 Function pointers and callable parametrs** 62

8.1 Function pointers . . . 62

8.1.1 As types . . . 62

8.1.2 As function paramater . . . 62

8.1.3 Function pointers to member functions . . . 63

8.2 Using std::function . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
8.3 With forwarding references . . . . . . . . . . . . . . . . . . . . . . . . . . 64

**9 Templates** 65
9.1 Template Function . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
9.2 Template Class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
9.3 Class vs typename keyword . . . . . . . . . . . . . . . . . . . . . . . . . . 66
9.4 Handle friend functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
9.4.1 Friendship to a Non-Template Function . . . . . . . . . . . . . . . . . . 66
9.4.2 Friendship to a Template Function . . . . . . . . . . . . . . . . . . . . . 67
9.5 Function Template Specialization . . . . . . . . . . . . . . . . . . . . . . . 69
9.6 Class/Struct Template Specialization . . . . . . . . . . . . . . . . . . . . . 69
9.7 Template Parameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
9.8 Trailing return type . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
9.8.1 Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
9.8.2 Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
9.9 Template functions with mixed types (Trailing return type) . . . . . . . . . 70
9.10 Template functions with mixed types (Deduced return type) . . . . . . . . . 71
9.11 Dependent name resolution . . . . . . . . . . . . . . . . . . . . . . . . . . 71
9.11.1 Dependent names . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
9.11.2 Typename Keyword . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
9.11.3 Nested types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
9.11.4 Prereq: Using aliases defined in classes . . . . . . . . . . . . . . . . . . 72
9.11.5 Type Aliases . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
9.11.6 Return Types in Template Functions . . . . . . . . . . . . . . . . . . . . 74
9.11.7 Base Class Members . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
9.11.8 Dependent Types in Expressions . . . . . . . . . . . . . . . . . . . . . . 74
9.12 Variadic templates with functions . . . . . . . . . . . . . . . . . . . . . . . 74
9.13 Left vs right folds . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
9.14 Parentheses in fold expressions . . . . . . . . . . . . . . . . . . . . . . . . 79
9.15 Pack size . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
9.16 Function calls in fold expressions . . . . . . . . . . . . . . . . . . . . . . . 80
9.17 Variadic templates with classes . . . . . . . . . . . . . . . . . . . . . . . . 81

9.18 std::forward . . . . . . . . 83
9.18.1 Key Differences Between std::forward and std::move . . . . . . . . 83
9.19 Universal reference (forwarding reference) . . . . . . . . 85
9.20 Concepts . . . . . . . . 86

**10 More on the comma operator** 88

**11 More on Lambdas** 89
11.1 Auto in lambda args . . . . . . . . 89
11.2 Template lambdas . . . . . . . . 89
11.3 Recursive lambdas . . . . . . . . 90

**12 When initializer lists are required** 91

**13 Inheritence and Subtype Polymorphism** 92
13.1 OOP Main Concepts . . . . . . . . 92
13.2 Object Relationships . . . . . . . . 92
13.3 Ineritance . . . . . . . . 93
13.4 Inheritance and Member Access . . . . . . . . 93
13.5 Inheritance Syntax . . . . . . . . 94
13.6 Upcasting and Downcasting . . . . . . . . 95
13.7 More on Downcasting . . . . . . . . 96
13.7.1 What Happens Without Virtual Functions . . . . . . . . 97
13.7.2 Downcasting example . . . . . . . . 98
13.7.3 Base class pointer example . . . . . . . . 99
13.8 Object Slicing . . . . . . . . 99
13.9 Multiple Inheritance . . . . . . . . 101
13.9.1 Why Use Multiple Inheritance? . . . . . . . . 101
13.9.2 Example . . . . . . . . 101
13.9.3 Issues with Multiple Inheritance . . . . . . . . 102
13.10 Virtual inheritance . . . . . . . . 103
13.10.1 The Diamond Problem . . . . . . . . 103
13.10.2 Solution with Virtual Inheritance . . . . . . . . 104
13.11 Subtype Polymorphism . . . . . . . . 104

13.12 Declaring Virtual Member Functions . . . 105
13.12.1 The override keyword . . . 105
13.13 Abstract or Pure virtual Member Functions . . . 106
13.14 Abstract Classes . . . 106
13.15 Interface Inheritance . . . 107

**14 <regex.h> Pattern Matching and String Validation** 108
14.1 regcomp . . . 108
14.1.1 Signature . . . 108
14.1.2 Return value . . . 108
14.1.3 Return errors . . . 108
14.1.4 Flags . . . 109
14.2 Regexec . . . 109
14.2.1 Signature . . . 110
14.2.2 Return value . . . 110
14.3 Regerror . . . 110
14.3.1 Signature . . . 110
14.3.2 Return value . . . 111
14.4 Regfree . . . 111
14.4.1 Signature . . . 111
14.5 regmatch_t and pmatch . . . 112
14.5.1 regmatch_t . . . 112
14.5.2 pmatch array . . . 113
14.6 Regex Example . . . 113

**15 Structured bindings** 116
15.1 As unpacks . . . 116
15.2 With returning . . . 116
15.3 With structs and classes . . . 117
15.4 With maps . . . 117

**16 Attributes in c++** 118
16.1 General Attributes . . . 118

**17 Inline functions** 121

**18 Advanced iterator usage** 122

18.1 base() . . . 122

**19 Regular expressions in c++** 123

19.1 Basic components . . . 123

19.1.1 The regex object . . . 123

19.1.2 The smatch array . . . 123

19.1.3 regex_match . . . 124

19.1.4 regex_search . . . 124

19.1.5 regex_replace . . . 126

19.2 match_results . . . 126

19.2.1 Methods . . . 126

19.3 sub_match . . . 127

19.4 Passing string iterators . . . 128

19.4.1 Getting all matches . . . 128

19.5 smatch prefix and suffix . . . 129

**20 Standard namespace** 131

20.1 std::bind . . . 131

20.1.1 std::placeholders . . . 131

20.1.2 Using std::ref with bind . . . 132

20.2 std::invoke . . . 133

20.2.1 Calling a regular function . . . 133

20.2.2 Calling a member function . . . 133

20.2.3 Accessing a member variable . . . 134

20.2.4 Calling a lambda . . . 134

20.3 std::exchange . . . 135

20.4 std::swap . . . 136

20.5 std::get . . . 137

20.6 std::tuple . . . 138

20.6.1 std::tuple and std::make_tuple . . . 138

20.6.2 Modifying elements . . . 138

20.6.3 std::tuple_size . . . 138

20.6.4 Unpacking with std::tie . . . 139

20.6.5 std::tie with std::ignore . . . 139

20.6.6 std::optional . . . 139

20.6.7 std::format . . . 140

**21 Views and Ranges** 141

21.1 Views . . . 141

21.1.1 Filter and transform . . . 141

21.1.2 Iota . . . 142

21.1.3 Take and drop . . . 142

21.2 The ranges library . . . 142

21.3 std::span . . . 142

21.3.1 What does it mean to be ”non-owning” . . . 143

21.3.2 subspans . . . 143

21.3.3 Key member functions . . . 144

**22 constexpr, consteval, and constinit** 145

22.1 constexpr Objects . . . 145

22.2 More on constexpr variables . . . 146

22.3 When const Can Make a Compile-Time Constant . . . 146

22.4 if constexpr . . . 147

22.5 Is constexpr const . . . 148

22.6 consteval . . . 148

22.7 constinit . . . 150

22.8 constexpr and consteval functions are implicitly inline . . . 150

**23 Smart pointers** 151

23.1 Class shared_ptr . . . 152

23.1.1 Using Class shared_ptr . . . 152

23.1.2 Defining a Deleter . . . 154

23.1.3 Dealing with Arrays . . . 155

23.1.4 More on make_shared . . . 156

23.1.5 Understanding the reference count . . . 157

23.1.6 Cyclic references . . . 158
23.1.7 Practical use of owner_before() . . . 159
23.2 Class weak_ptr . . . 160
23.2.1 Using a weak_ptr . . . 161
23.3 Unique_ptr . . . 162
23.3.1 Using a unique_ptr . . . 163
23.3.2 Transfer of Ownership by unique_ptr . . . 164
23.3.3 Source and Sink . . . 165
23.3.4 unique_ptrs as Members . . . 166
23.3.5 Dealing with arrays . . . 167
23.3.6 Double ownership . . . 168
23.4 Smart pointers to stack memory? . . . 169

**24 Three way comparisons (spaceship operator) <compare> 170**
24.1 Custom <=> logic . . . 170
24.2 Return types . . . 170
24.3 Spaceship on primitive types . . . 171

**25 Single Dispatch and Overload Resolution 172**
25.1 Single dispatch . . . 172
25.2 Overload resolution (compile time) . . . 172

**26 Bitfields 173**
26.0.1 Address operator on a bitfield? . . . 174

**27 C++ versions and their additions 175**

# 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.

```cpp
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

```cpp
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

```cpp
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 this 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.~list()`
  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()

```
#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`

## Labels and goto

goto is a statement that allows you to jump directly to a labeled section of code within the same function. The target of a goto statement is called a label, which is defined by writing a name followed by a colon (:).

```
#include <iostream>
using namespace std;

int main() {
    cout << "Start of program" << endl;
    goto skip;  // Jumps to the 'skip' label

    // This line is skipped because of the goto
    cout << "This line will be skipped" << endl;

    skip:
    cout << "Jumped to label 'skip'" << endl;
    return 0;
}
```

# Type traits

Type traits are a collection of template-based utilities that allow you to perform compile-time introspection and manipulation of types. They are part of the Standard Template Library (STL) and are available through the header <type_traits>. These utilities help you gather information about types (e.g., whether a type is an integer or floating-point, whether it is const-qualified, etc.) and perform transformations on them (e.g., remove const, add reference, etc.).

- **Type Properties:** These allow you to query certain properties of a type
- **Type Modifications**: These traits allow you to modify types, like adding or removing qualifiers
- **Conditional Typing:** These help you select types conditionally:
- **Transformation Traits:** These are useful when you need to change a type in a generic way:

## 4.1 Nonstandard type of types

- **Fundamental:** Includes basic types like int, float, char, void, and nullptr_t. They are built-in types in the language.
- **Arithmetic:** Includes all integral types (like int, char) and floating-point types (like float, double).
- **Scalar:** Includes arithmetic types, pointers, and nullptr_t. Essentially, types that hold a single value.
- **Object:** Any type that can hold data in memory, including scalar types, arrays, and classes/structs.
- **Compound:** Any type built from other types, such as arrays, pointers, references, classes, or unions.
- **Trivial:** A type that has a trivial constructor, destructor, and copy/move operations. It means they can be easily copied or constructed without any custom behavior.
- **Trivially copyable:** A type that can be copied using simple memory operations, such as memcpy, without needing any special handling.
- **Polymorphic:** A class type that contains at least one virtual function, enabling runtime polymorphism through inheritance.
- **Abstract:** A class type with at least one pure virtual function, making it impossible to instantiate directly.
- **Final:** A class or virtual function marked as final cannot be derived from or overridden further.
- **Aggregate:** A simple class or struct with no private or protected members, no constructors, and no virtual functions, allowing it to be initialized with a brace-enclosed initializer list.

## 4.2 ::value, ::type, and ::value_type

::value, ::type, and ::value_type are commonly seen in template metaprogramming, especially with traits and type manipulation.

- **::value**: Used in type traits to access a constant value. Typically, this is a static constexpr or enum that holds an integral value.

```
std::is_integral<int>::value  // true (1)
```

- **::type**: Refers to a type alias or typedef within a template. It is commonly used in type traits to represent the result of a type transformation or query.

```
std::remove_const<const int>::type  // int (removes const
↪  qualifier)
```

- **::value_type**: A member type that typically represents the type of elements stored in a container. Found in STL containers like std::vector or custom types.

```
std::vector<int>::value_type  // int
```

# 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
```

#### 5.0.0.1 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.

#### 5.0.0.2 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 >

# Decltype

**Concept 1:** **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.
□

## 6.1 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.

## 6.2 Example

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

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

## 6.3 Things to pair with decltype

- **std::decay_t** is commonly used with decltype to remove references and qualifiers from a type. It transforms types to their "decayed" forms, similar to what happens when passing an argument to a function.

  ```cpp
  auto var = someFunction();
  using DecayedType = std::decay_t<decltype(var)>;
  ```

- **std::remove_reference / std::remove_const / std::remove_cv:**: These traits can be used to strip references, const, and volatile qualifiers from the type deduced by decltype.

  ```cpp
  using NonConstType = std::remove_const_t<decltype(someVar)>;
  ```

- **std::is_same**: Useful when you want to compare the type deduced by decltype with some other type to perform type-checking in templates.

  ```cpp
  static_assert(std::is_same_v<decltype(var), int>, "Type must
  ↪  be int");
  ```

- **std::common_type:** It computes the common type between two or more types, which is useful when you want to get a type that can hold the result of an expression involving multiple types deduced by decltype.

  ```cpp
  using CommonType = std::common_type_t<decltype(a),
  ↪  decltype(b)>;
  ```

- **std::conditional**: This allows for conditional type selection based on a compile-time boolean value, often paired with decltype to determine types based on some logic.

  ```cpp
  using ConditionalType = std::conditional_t<some_condition,
  ↪  decltype(a), decltype(b)>;
  ```

  *a* if true, *b* if false. A compile-time boolean value is a value that is determined during the compilation phase, rather than at runtime. You typically get these compile-time boolean values from type traits or constexpr values

Where $\_t$ is shorthand for $::type$ and $\_v$ is shorthand for $::value$

# Constexpr

In C++, constexpr is a keyword that allows you to declare that a variable or function can be evaluated at compile time if its inputs are known during compilation. It is used to define constant expressions, which are expressions that can be computed at compile time, leading to potential performance improvements and safer code by ensuring certain values are computed early.

## 7.1 Variables

A constexpr variable is guaranteed to have a constant value, and that value is computed at compile time. This is useful for things like array sizes or any situation where you need a compile-time constant.

```
constexpr int size = 10;  // size is computed at compile time
int arr[size];            // Valid because size is constant
```

## 7.2 Functions

A constexpr function can be evaluated at compile time if its arguments are known at compile time. If the arguments are not known until runtime, it will behave like a regular function and be evaluated at runtime.

constexpr functions must have a single return statement, or all branches must return a value.

```
constexpr int square(int x) {
    return x * x;
}

constexpr int result = square(5);  // This will be computed at compile time
```

## 7.3 Object Constructors

You can also mark constructors as constexpr, meaning you can create objects that are initialized at compile time

```
struct Point {
    constexpr Point(int x, int y) : x(x), y(y) {}
    int x, y;
};

constexpr Point p(3, 4);  // Object created at compile time
```

## 7.4 constexpr vs const

const variables are constant at runtime but not necessarily at compile time. You can initialize const with runtime values.

constexpr implies const but guarantees that the value or function result is computed at compile time if possible.

# Function pointers and callable parametrs

## 8.1 Function pointers

Function pointers are variables that store the address of a function. They can be passed as parameters to other functions, allowing the called function to invoke the pointed-to function.

### 8.1.1 As types

```
void fn(unsigned short a) {
    cout << "Arg is: " << a << '\n';
}

void (*f)(unsigned short) = &fn;
void (*f)(unsigned short) = fn;
f(20);
```

Here *f* is a pointer to function that takes an unsigned short and returns void. You may notice that we can omit the address of operator. In C++, function names can implicitly convert to pointers to those functions. This behavior is similar to how array names decay to pointers to their first elements.

### 8.1.2 As function paramater

```
void fn(unsigned short a) {
    cout << "Arg is: " << a << '\n';
}

void fun(void (*f)(unsigned short), int n) {
    f(n);
}

void (*f)(unsigned short) = &fn;
fun(f, 12);
```

Here, fun is a function that has two parameters, a pointer a function that takes an unsigned short and returns void, and an integer n.

We can also pass lambdas.

```
void fun(void (*f)(unsigned short), unsigned short x) {
    f(x);
}

auto lambda = [] (unsigned short n) -> void {
    cout << n << endl;
};

fun(lambda, 12);
```

### 8.1.3 Function pointers to member functions

Function pointers to member functions in C++ are a way to refer to non-static member functions of a class. They differ from regular function pointers because member functions implicitly take a this pointer to operate on a specific instance of the class.

To declare a pointer to a member function of a class ClassName with a signature ReturnType(ClassName::*)(Arguments)

```
ReturnType (ClassName::*pointerName)(Arguments);
```

```
struct foo {
    void print() {
        cout << "Hello world" << endl;
    }
};

void (foo::*f)() = &foo::print;
foo f1;
(f1.*f)();

foo* fptr = &f1; // Through a ptr
(fptr->*f)();
```

## 8.2 Using std::function

We can also define function types with std::function.

```
void fn(unsigned short a) {
    cout << "Arg is: " << a << '\n';
}

void fun(std::function<void(unsigned short)> f, int x) {
    f(x);
}

fun(fn, 12);

std::function<void(unsigned short)> f = [] (unsigned short a) -> void {
    cout << a << endl;
};

fun(f, 13);
```

## 8.3 With forwarding references

F&& in a template context allows you to accept any callable (function, lambda, functor) while preserving its lvalue/rvalue nature. This is a common pattern for writing generic functions that can accept various types of arguments and forward them to other functions without losing efficiency.

```
template<typename F>
void fn(F&& fun) {
    std::invoke(std::forward<F>(fun));
}
```

We used std::forward<F>(fun) instead of just fun. This ensures that if fun is an rvalue, it gets forwarded correctly as an rvalue; if it's an lvalue, it stays an lvalue. This is part of perfect forwarding.

# 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.

## 9.1 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;
}
```

## 9.2 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;
}
```

## 9.3 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

## 9.4 Handle friend functions

### 9.4.1 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>&);
};
```

### 9.4.2 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
};
```

## 9.5 Function Template Specialization

**Concept 2:** **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;
}
```

## 9.6 Class/Struct Template Specialization

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

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

## 9.7 Template Parameters

**Concept 3:** Templates can have more than one parameter, including non-type parameters.
□

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

## 9.8 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.

### 9.8.1 Syntax

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

### 9.8.2 Example

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

## 9.9 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;
}
```

## 9.10 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;
}
```

## 9.11 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.

### 9.11.1 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.

### 9.11.2 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

### 9.11.3 Nested types

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

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

### 9.11.4 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*

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

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

This also applies to typedefs...

```cpp
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

### 9.11.5 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
}
```

### 9.11.6 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();
}
```

### 9.11.7 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;
};
```

### 9.11.8 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.

## 9.12 Variadic templates with functions

Variadic templates in C++ allow you to write functions or classes that can take an arbitrary number of template arguments

A variadic template is defined by using an ellipsis (...) both in the template parameter list and in the function arguments.

```cpp
template<typename... Args>
void print(Args... args) {
    // function body
}
```

- **typename...** Args declares a parameter pack named Args, which can hold zero or more template arguments.
- **Args...** args is another parameter pack that holds the function parameters of arbitrary types and numbers.

A parameter pack in C++ is a feature of variadic templates that represents zero or more template parameters or function arguments. It allows functions or classes to handle an arbitrary number of arguments.

To process the elements of the parameter pack, you can use a technique called pack expansion. A common way to do this is by recursively calling the variadic function.

```cpp
#include <iostream>

void print() {
    std::cout << std::endl;
}

template<typename T, typename... Args>
void print(T first, Args... args) {
    std::cout << first << " ";
    print(args...);  // recursive call with remaining arguments
}

int main() {
    print(1, 2.5, "Hello", 'A');
    return 0;
}
```

In C++17, fold expressions simplify handling variadic templates by eliminating the need for recursion. You can apply an operation over the entire parameter pack in a single expression

```cpp
#include <iostream>

template<typename... Args>
void print(Args... a) {
    ((std::cout << a << std::endl) , ...); // Right fold
     (..., (std::cout << a << endl)); // Left fold
     // Both folds output the same, more on this later
}

int main() {
    print(1, 2.5, "Hello", 'A');
    return 0;
}
```

The ellipsis ... is part of the fold expression syntax, and here it's applied to the entire operation inside the parentheses: (cout « a « endl). This means the expression cout « a « endl will be expanded for each argument a in the parameter pack.

The comma operator (operator,) allows multiple expressions to be evaluated in sequence. It guarantees that for each a, the expression cout « a « endl will be executed. This effectively prints each argument followed by a newline.

The expanded form of ((cout « a « endl), ...) would look something like:

```cpp
(cout << arg1 << endl, cout << arg2 << endl, cout << arg3 << endl, ...);
```

To exapand upon this...

```cpp
template <typename ... args>
void fn(args ... a) {
    ((std::cout << a << endl) , ...);
}

void f(int a, int b, int c) {
    cout << a << endl, cout << b << endl, cout << c << endl;
}

fn(1,2,3);
cout << endl;
f(1,2,3);
```

Both functions have the same output.

In the example above we are using the comma operator, but it could be any operator

```
template<typename T, typename... Args>
T fn(Args ... a) {
    int sum = 0;
    sum = (sum + ... + a); // Expand: sum = (0 + a_1 + a_2 + ... + a_n);
    (sum += ... += a); // Expand: sum += a_1 += a_2 += a_3 += ... += a_n
    sum += (... + a); // Expand: sum += (a-1 + ... + a_n)
    return sum;
}
```

Recall, assignment operators in c++ are right to left

## 9.13 Left vs right folds

A left fold applies the operator from left to right.

```
(... op pack)
```

The operator op is applied between the leftmost elements first, and then successively to the rest of the elements.

A right fold applies the operator from right to left.

```
pack op ...
```

The operator op is applied between the rightmost elements first, and then successively to the remaining elements.

## 9.14 Parentheses in fold expressions

The parentheses in fold expressions in C++ are used to ensure correct grouping and to determine how the fold expression expands over the parameter pack. They help dictate the precedence and associativity of the operation being applied to the elements of the parameter pack, ensuring the operator is applied in the desired order.

Parentheses are necessary to group the fold expression correctly, especially when using operators like +, -, +=, etc., which need to be explicitly applied across the elements of a parameter pack.

The parentheses ensure that the fold applies the operator across the entire parameter pack and handles the result correctly.

```
(sum += ... + args);
```

Here, the parentheses ensure that the += is applied between sum and the result of the left fold ... + args.

Without parentheses, operator precedence could cause the fold expression to be parsed incorrectly by the compiler, leading to syntax errors or unintended behavior.

## 9.15 Pack size

sizeof...(Args) (also called "pack size") is a compile-time operator that gives you the number of template arguments in the parameter pack Args.... It works for both types and non-type template parameter packs. This can be helpful in function calls, especially when you're dealing with variadic templates and need to know the size of the pack for controlling logic or iteration.

```
// Function template using sizeof... to count the number of arguments
template<typename... Args>
void countArguments(Args... args) {
    std::cout << "Number of arguments: " << sizeof...(Args) << std::endl;
}
```

## 9.16 Function calls in fold expressions

```
template<typename T>
void print(T t) {
    cout << t << endl;
}

template<typename... args>
void f(args...a) {
    (print(a),...);
    (..., print(a)); // Does the same
}
```

## 9.17 Variadic templates with classes

A variadic class template is defined in much the same way as a variadic function template, but it applies to classes. You use typename... (or class...) to declare a parameter pack in the class definition, which allows the class to accept multiple template arguments.

```cpp
template<typename... Args>
class MyClass {
    // Use Args... in the class definition
};
```

In this example, Args... is a parameter pack that can hold any number of types.

```cpp
// Variadic class template
template<typename... Args>
class MyClass {};

// Specialization for one type argument
template<typename T>
class MyClass<T> {
    public:
    void display() {
        std::cout << "One template argument: " << typeid(T).name() << std::endl;
    }
};

// Specialization for two type arguments
template<typename T1, typename T2>
class MyClass<T1, T2> {
    public:
    void display() {
        std::cout << "Two template arguments: "
        << typeid(T1).name() << " and " << typeid(T2).name() << std::endl;
    }
};

// Specialization for more than two arguments
template<typename T, typename... Rest>
class MyClass<T, Rest...> {
    public:
    void display() {
        std::cout << "First argument: " << typeid(T).name() << std::endl;
        std::cout << "Number of remaining arguments: " << sizeof...(Rest) << std::endl;
    }
};
```

```cpp
template<typename...Args>
struct foo {

    void printargs(Args...args) {
        ((cout << args << endl), ...);
    }
};

foo<int,string,double> a;
a.printargs(1,"foo",3.14f);
```

## 9.18 std::forward

std::forward is a utility function in C++ that is used primarily in template functions to enable perfect forwarding. It is part of the C++ Standard Library and is typically used to forward arguments while preserving their value category (whether they are lvalues or rvalues).

In generic template code, when you want to pass arguments to another function or callable, you need to preserve whether those arguments were passed as lvalues (references to existing objects) or rvalues (temporary objects). std::forward allows you to forward arguments while ensuring that their value category is preserved.

Without std::forward, arguments passed to template functions would often lose their original value category, especially when working with rvalues

```
template< class T >
T&& forward( typename std::remove_reference<T>::type\& arg ) noexcept;
```

If the argument is an rvalue std::forward will cast it to an rvalue (i.e., T&&), allowing the function to move the argument if appropriate (typically used in move semantics).

If the argument is an lvalue std::forward will cast it to an lvalue reference (T&), ensuring that the original object is used and not moved or copied unnecessarily.

```
template <typename T>
void wrapper(T&& arg) {
    func(std::forward<T>(arg));  // Perfectly forward arg, preserving its value category
}
```

Now, std::forward<T>(arg) ensures that if arg was originally passed as an rvalue, it remains an rvalue when forwarded to func, enabling func to take advantage of move semantics. Similarly, if arg was an lvalue, it is passed as an lvalue.

### 9.18.1 Key Differences Between std::forward and std::move

std::move Unconditionally casts the argument to an rvalue, suggesting that it can be ”moved from” (even if it was originally an lvalue). It doesn’t preserve the original value category but rather forces the argument to be treated as an rvalue.

std::forward Conditionally casts the argument, preserving its value category. It is only used in template code and requires knowledge of whether the argument was an lvalue or rvalue (via the type deduction)

```cpp
#include <iostream>
#include <utility>

void process(const std::string& s) {
    std::cout << "Lvalue: " << s << '\n';
}

void process(std::string&& s) {
    std::cout << "Rvalue: " << s << '\n';
}

template <typename T>
void wrapper(T&& arg) {
    process(std::forward<T>(arg));  // Forward arg to process
}

int main() {
    std::string str = "Hello";

    wrapper(str);            // Lvalue is forwarded, calls process(const std::string&)
    wrapper(std::move(str)); // Rvalue is forwarded, calls process(std::string&&)
}
```

## 9.19 Universal reference (forwarding reference)

Universal references (also called forwarding references) in C++ are template parameters declared as T&& that can bind to both lvalues and rvalues. The behavior of a universal reference depends on the type of the argument passed:

If an lvalue is passed, T&& becomes an lvalue reference (T&). If an rvalue is passed, T&& remains an rvalue reference (T&&).

Universal references are typically used with perfect forwarding, allowing you to forward arguments to other functions while preserving their value category (lvalue or rvalue).

```cpp
template<typename T>
void bar(T&& x) {
    cout << x << '\n';
}

template<typename T>
void foo(T&& x) {
    bar(std::forward<T>(x));
}

template<typename T>
void fn(T&& x) {
    cout << x << endl;
}

int x = 20;
decltype(x)& y = x;
fn(std::forward<decltype(y)>(y));
```

## 9.20 Concepts

Concepts are a feature introduced in C++20 that allow you to specify constraints on template parameters in a clear, concise, and readable way. They improve the expressiveness, safety, and usability of templates by providing a mechanism to enforce specific requirements on the types passed to a template.

A concept is defined using the concept keyword and is essentially a compile-time boolean expression that evaluates to true or false.

```
#include <concepts>

// Define a concept

// This is std::integral
template <typename T>
concept Integral = std::is_integral_v<T>;

// This is std::floating_point
template <typename T>
concept FloatingPoint = std::is_floating_point_v<T>;
```

You can constrain template parameters directly using a concept.

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

add(40,20) // OK
add(40.0f, 20.0f) // ERROR
```

Alternatively, you can use a requires clause:

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

You can combine concepts with auto for a more concise syntax:

```
auto add(Integral auto a, Integral auto b) {
    return a + b;
}
```

Concepts can also be used to constrain class templates.

```cpp
template <Integral T>
class Calculator {
    T value;
};
```

Concepts can be combined using logical operators (&&, ||, !).

```cpp
template <typename T>
concept Numeric = Integral<T>  FloatingPoint<T>;
```

The <concepts> header provides a set of predefined concepts for common use cases:

- std::same_as<T, U> Ensures T is the same type as U.
- std::convertible_to<T, U> Ensures T can be converted to U.
- std::derived_from<T, U> Ensures T is derived from U.
- std::integral Matches integral types (int, char, etc.).
- std::floating_point Matches floating-point types (float, double).
- std::assignable_from<T, U> Ensures T = U is valid.

## More on the comma operator

The comma acts as both a separator and an operator, depending on the context.

As a separator, The comma separates elements in lists, such as function arguments, initializer lists, or variables in declarations. This much is trivial

As an Operator, the comma operator (when used in expressions) evaluates two or more expressions from left to right, and the result of the entire comma expression is the result of the right-most expression.

```
int a = (1, 2);  // a gets the value of 2
```

In this case, 1 is evaluated first, then 2 is evaluated, and the result of the entire expression is 2.

In loops,

```
for (int i = 0, j = 10; i < j; ++i, --j) {
    // Do something
}
```

Here, the comma separates the initialization of $i$ and $j$ as well as the increment and decrement operations in the loop.

# More on Lambdas

## 11.1 Auto in lambda args

In C++, the auto keyword is used in lambda expressions to infer the type of the parameters or the return type. It helps make lambda functions more flexible and convenient to use without explicitly specifying types, especially when types are complex or unknown at compile time.

```
struct foo {
    int x = 0, y = 0;

    foo() = default;
    foo(int a, int b) : x(a), y(b) {}

    friend std::ostream& operator<<(std::ostream& os, foo f);
};

std::ostream& operator<<(std::ostream& os, foo f) {
    os << "x: " << f.x << endl << "y: " << f.y << endl;
    return os;
}

foo f1(1,2), f2(3,4);

auto fn = [] (const auto& a, const auto& b) -> auto {
    cout << a << endl << b;
};

fn(f1,f2);
```

It should be noted that you cannot do this with regular functions

```
// Not allowed!
void fn(const auto& a, const auto& b) {
    cout << a << endl << b;
}
```

## 11.2 Template lambdas

Since C++20, we can make template lambdas.

```
auto fn = []<typename T>(const T& a, const T& b) -> auto{
    cout << a << endl << b;
};
fn(f1,f2); // No need for <> notation, auto deduced
```

## 11.3 Recursive lambdas

In C++, you cannot use auto to declare a recursive lambda directly because lambdas do not have a name. Since they are anonymous, you can't directly reference the lambda within its own body. For recursion, a function needs to call itself, but without a name, the lambda cannot do that.

As a side note, we also must capture the lambda by reference.

```
 auto lambda = [&lambda](int x, int n) { // ERROR
     if (n == 0) return 1;

     return x * lambda(x,n-1);
};
```

This code gives the error "Variable 'lambda' declared with deduced type 'auto' cannot appear in its own initializer".

We can use std::function to explicitly define the type of the lambda. This works because std::function provides a way to store callable objects with a known signature.

```
std::function<int(int,int)> lambda = [&lambda](int x, int n) {
 ↪  // Works
    if (n == 0) return 1;

    return x * lambda(x,n-1);
};
```

## 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

## 13.1 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.

## 13.2 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.

## 13.3 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.

## 13.4 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.

## 13.5 Inheritance Syntax

To declare a derived class:

```
// 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;
}
```

## 13.6 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:

```
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.

```
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:

```
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
}
```

## 13.7 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).

### 13.7.1 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.

### 13.7.2 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.

### 13.7.3 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.

## 13.8 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
```

## 13.9 Multiple Inheritance

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

### 13.9.1 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++.

### 13.9.2 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;
}
```

### 13.9.3 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.

## 13.10 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.

### 13.10.1 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.

### 13.10.2 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.

```cpp
#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;
}
```

## 13.11 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.

## 13.12 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.

### 13.12.1 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
}
```

## 13.13 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.

## 13.14 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.

## 13.15 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

## 14.1 regcomp

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

### 14.1.1 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).

### 14.1.2 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.

### 14.1.3 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.

### 14.1.4 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.

## 14.2 Regexec

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

### 14.2.1 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.

### 14.2.2 Return value

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

## 14.3 Regerror

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

### 14.3.1 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.

### 14.3.2 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.

## 14.4 Regfree

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

### 14.4.1 Signature

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

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

## 14.5 regmatch_t and pmatch

### 14.5.1 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.

### 14.5.2 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.

## 14.6 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);
```

# Structured bindings

Structured bindings in C++ (introduced in C++17) allow you to unpack or "decompose" tuples, pairs, or other structured objects into separate variables in a concise way. This makes it easier to work with multiple return values or complex data structures.

## 15.1 As unpacks

```
int arr[3] = {1,2,3};
auto [x,y,z] = arr;

cout << x << '\n' << y << '\n' << z;
// 1
// 2
// 3
```

This works for a number of containers

- std::Tuple
- std::Pairs
- Fixed-sized arrays
- std::Array
- Structs/Classes

## 15.2 With returning

```
std::pair<int,int> foo() {
    std::pair<int,int> p = {1,2};
    return p;
}

auto [a,b] = foo();
cout << a << endl << b << endl;
// 1
// 2
```

## 15.3 With structs and classes

```cpp
struct foo {
    int x;
    int y;
    int z;
};

foo f1{1,2,3};

auto [a,b,c] = f1;

cout << a << endl << b << endl << c << endl;
// 1
// 2
// 3
```

## 15.4 With maps

Since we can unpack std::pairs, we can use structured bindings in map range-based loops

```cpp
map<int, int> a = {{0,1}, {1,2}};

for (const auto& [key,value] : a) {
    cout << key << " " << value << endl;
}
```

# Attributes in c++

In C++, attributes are a way to provide additional information or hints to the compiler without affecting the actual logic of the program. They can be used to optimize the program, check for potential issues, or control specific aspects of how the compiler processes the code. Attributes are written inside double square brackets ([[ ... ]]) and can be attached to various elements of the code, such as functions, variables, types, and control structures

```
[[attribute-name]]
[[attribute-name(arg1, arg2, ...)]]
```

Attributes can be placed in various locations:

- Before a function or variable declaration
- Before a class or struct declaration
- In the middle of a function, such as on a statement in a switch case

## 16.1 General Attributes

These attributes are part of the standard C++ specification and are supported across most modern compilers.

- [[**nodiscard**]]**:** Ensures that the return value of a function is not ignored.

```
[[nodiscard]] int calculate() { return 42; }
int main() {
    calculate(); // Compiler warning: return value ignored
}
```

- [[**deprecated**]]**:** Marks code elements like functions or variables as deprecated. Using them will generate a warning, helping to phase out old or unsafe functionality.

```
[[deprecated("Use newFunction() instead")]]
void oldFunction() {}
```

- [[**maybe_unused**]]**:** Suppresses compiler warnings for unused variables or parameters, often useful in code that will vary between different builds (e.g., debug vs release builds)

```
void fn([[maybe\_unused]] int a);
[[maybe\_unused]] int x = 5;
```

- [[**fallthrough**]]: Used in switch statements to indicate that a case is intentionally falling through to the next one, preventing warnings from the compiler in such cases.

```cpp
switch (value) {
    case 1:
        // Some code
        [[fallthrough]];  // Intentional fallthrough to the next case
    case 2:
        // More code
        break;
}
```

- **[[likely]] and [[unlikely]]**: (since C++20): Provide hints to the compiler about the likelihood of branches in if or switch statements. These hints allow the compiler to optimize for the most likely or unlikely branches.

```cpp
if ([[likely]] condition) {
    // Optimized assuming this is more likely
}
```

- **[[alignas(n)]]:** Specifies the memory alignment for a variable or type. It ensures that the object is aligned on a boundary of n bytes.

```cpp
alignas(16) int alignedInt;  // Ensures the integer is aligned on a 16-byte boundary
```

- **[[no_unique_address]]**: (since C++20): Allows the compiler to optimize the memory layout of a class or struct by not requiring a unique address for certain members (e.g., empty classes).

```cpp
struct Empty {};
struct S {
    [[no_unique_address]] Empty e;
    int x;
};
```

- **[[noreturn]]:** Marks a function as one that will never return to the caller (e.g., functions that throw exceptions or terminate the program).

```cpp
[[noreturn]] void fatalError() {
    throw std::runtime_error("Critical error!");
}
```

- **[[gnu::always_inline]]:** Forces the compiler to inline the function, even when optimizations are disabled. This attribute is specific to certain compilers like GCC or Clang.

```cpp
[[gnu::always_inline]] void inlineFunction() {
    // The function will always be inlined
}
```

- **[[gnu::pure]]:** Marks a function as "pure," meaning its return value depends only on its parameters and has no side effects, allowing the compiler to perform certain optimizations.

```cpp
[[gnu::pure]] int square(int x) {
    return x * x;
}
```

## Inline functions

In C++, an inline function is a function where the compiler attempts to replace the function call with the actual function code itself (i.e., inline expansion) rather than generating a normal function call. The goal of inlining is to reduce the overhead of function calls, especially for small functions, and potentially improve performance.

However, inline is a request or hint to the compiler, not a command. The compiler may ignore the request to inline the function, particularly if the function is too complex or if it would increase the code size too much.

To declare a function as inline, you use the inline keyword in its declaration or definition:

```cpp
inline int add(int a, int b) {
    return a + b;
}
```

Consider the following example

```cpp
inline int square(int x) {
    return x * x;
}

int main() {
    int result = square(5);
}
```

Without inline, the function call square(5) would generate assembly code that jumps to the function definition, executes it, and returns the result. With inline, the compiler might replace the square(5) call directly with 5 * 5 in the code, eliminating the need for a function call.

# Advanced iterator usage

## 18.1 base()

The function base() is used when working with reverse iterators in C++. It converts a reverse iterator (i.e., one that iterates from the end of a container to the beginning) into a regular iterator that points to the corresponding element in the normal (forward) iteration sequence.

When you reverse iterate through a container (like a std::string or std::vector), the reverse iterator rbegin() points to the last element in the container, and as you increment it, it moves backward through the container. When you reach an element using the reverse iterator and want to switch back to normal (forward) iteration, you use the base() method. It returns a forward iterator that points just past the element that the reverse iterator refers to.

# Regular expressions in c++

To use regex expressions, we include <regex>

```
#include <regex>
```

## 19.1 Basic components

- **std::regex:** Represents the regular expression pattern.
- **std::smatch:** Used for storing results of a match against a std::string.
- **std::regex_match:** Tests whether an entire string matches the regex.
- **std::regex_search:** Searches a string for any sequence matching the regex.
- **std::regex_replace:** Replaces parts of a string that match the regex pattern.

### 19.1.1 The regex object

The regex object is used to store a pattern.

```
std::regex pattern("^\\w+$");
std::regex pattern(R"(^\w+$)"); // With raw strings
```

### 19.1.2 The smatch array

We can store captures in the smatch array

```
string s = "     hello     ";
regex pattern1(R"(\s*((\w)+)\s*)");
regex pattern2(R"(\s*(\w+)\s*)"); // Does the same as pattern one
smatch capture_space;

// Strips leading and trailing spaces
if (regex_match(s, capture_space, pattern1)) {
    cout << capture_space[1] << endl;
}
```

capture_space[0] is the entire string, capture_space[1-n] are the captures 1-n. In the example above capture_space[1] holds the trimmed string. We index with one because it is the first (and only) capture.

### 19.1.3 regex_match

We can use the std::regex_match function to match entire strings against a defined pattern, like we did in the example above. Returns true if the string was matched, false otherwise.

```
bool regex_match(const string& s, smatch& capture_space, const regex& pattern);
```

### 19.1.4 regex_search

Searches a string for any match of a regular expression pattern. Unlike regex_match, it doesn't require the entire string to match the pattern, just a part of it.

```
bool regex_search(const string& s, smatch& capture_space, const regex& pattern);
```

```
std::string s = "abc123";
std::regex pattern("\\d+");
std::smatch match;

if (std::regex_search(s, match, pattern)) {
    std::cout << "Found match: " << match.str() << '\n';
}
```

Note that we can always use regex_match in places where regex_search is required. Consider the pattern from the example above

```
std::string s = "abc123";
std::regex pattern("\\d+"); // Suitable  for regex_search
std::regex pattern(".*\\d+.*"); // Suitable  for regex_match
```

**Note:** Note that regex_search will put only the first matching part in the smatch array. For example

```cpp
string s = "key1=value1 key2=value2 key3=value3";
smatch capture_space;

if (regex_search(s, capture_space, regex(R"(\s*(\w+)=(\w+)\s*)"))) {
    for (const auto& item : capture_space) {
        cout << item << endl;
    }
}
/* Outputs:
    key1=value1
    key1
    value1
*/
```

### 19.1.5 regex_replace

Replaces occurrences of a pattern in a string with a specified replacement string.

```
std::string regex_replace(const string& s, const regex& pattern, const string& replacement)
```

**Note:** This function will not modify the original string, it always returns a new string.

## 19.2 match_results

std::match_results is a template class in C++ that holds the results of a regular expression match operation, such as those produced by std::regex_search or std::regex_match. It is typically used with std::smatch (for std::string matches) or std::cmatch (for C-style string matches).

- **std::smatch:** A typedef for `std::match_results<std::string::const_iterator>`.
- **std::cmatch:** A typedef for `std::match_results<const char*>`.

### 19.2.1 Methods

- **str():** Returns the matched string or a submatch as a `std::string`.
- **size():** Returns the number of matches (main match + submatches).
- **position():** Returns the position in the input string where the match begins.
- **length():** Returns the length of the match.
- **operator[]:** Access individual submatches via index.
- **prefix():** Returns the part of the input string that precedes the match as a `sub_match` object.
- **suffix():** Returns the part of the input string that follows the match as a `sub_match` object.
- **empty():** Checks if the match was successful (returns `true` if no match was found).
- **ready():** Checks if the `match_results` object is ready and valid for use after a regex search.
- **format():** Returns the formatted string based on the matched results using a specified format string.
- **begin():** Returns an iterator to the first submatch (constant iterator).
- **end():** Returns an iterator past the last submatch (constant iterator).

## 19.3 sub_match

`std::sub_match` is a class template that represents a single match or submatch from a regular expression search. It typically stores a reference to the portion of the input string that matched a particular capture group.

It behaves like a `std::pair<BidirectionalIterator, BidirectionalIterator>`, storing the start and end iterators of the matched range.

You can access the matched string using the str() method.

It supports comparison operators (==, !=) and conversion to std::string.

It is often used with `std::match_results` to represent the main match or submatches (captured groups).

```
string s = "key1=value1 key2=value2 key3=value3";

smatch capture_space;
string::const_iterator curr = s.begin();

if (regex_search(curr, s.cend(), capture_space, regex(R"(\s*(\w+)=(\w+)\s*)"))) {
    cout << capture_space[1] << endl; // key1

    std::sub_match<string::const_iterator> sm = capture_space[1];
    cout << *sm.first << endl << *sm.second << endl;
    // k
    // =
}
```

## 19.4 Passing string iterators

We can also pass iterators to the regex functions.

```
string s = "key1=value1 key2=value2 key3=value3";
smatch capture_space;
string::const_iterator curr = s.begin();

if (regex_search(s, capture_space, regex(R"(\s*(\w+)=(\w+)\s*)"))) {
    for (const auto& item : capture_space) {
        cout << item << endl;
    }
}
```

**Note:** The string iterators passed must be constant.

### 19.4.1 Getting all matches

To get all matches, we can search in a loop, updating our start iterator after each match. For example

```
string s = "key1=value1 key2=value2 key3=value3";

smatch capture_space;
string::const_iterator curr = s.begin();
while (regex_search(curr, s.cend(), capture_space, regex(R"(\s*(\w+)=(\w+)\s*)"))) {
    cout << "key: " << capture_space[1] << '\n' << "value: " << capture_space[2] << '\n';

    sub_match<string::const_iterator> sm = capture_space[2];
    curr = sm.second;

    */
        key: key1
        value: value1
        key: key2
        value: value2
        key: key3
        value: value3
    */
}
```

The sub match object we created captures the second match group (the value) into a std::sub_match object. The std::sub_match object provides the iterators (.first and .second) that point to the start and end of the matched portion of the string.

The iterator `sm.second` points to the first character after the matched value. By setting `curr = sm.second`, we move the search position forward, skipping over the current match so that the next iteration of regex_search starts searching after the last key=value pair.

## 19.5 smatch prefix and suffix

Consider the code

```
string s = "key1=value1 key2=value2 key3=value3";

smatch capture_space;
string::const_iterator curr = s.begin();
if (regex_search(curr, s.cend(), capture_space, regex(R"(\s*(\w+)=(\w+)\s*)"))) {
    for (const auto& item : capture_space) {
        cout << "capture space item : " << item << endl;
    }

    cout << "prefix first: " << *capture_space.prefix().first << endl;
    cout << "prefix second: " << *capture_space.prefix().second << endl;

    cout << "Suffix first: " << *capture_space.suffix().first << endl;
    cout << "Suffix second: " << *(capture_space.suffix().second - 1) << endl;

    /*
        capture space item : key1=value1
        capture space item : key1
        capture space item : value1
        prefix first: k
        prefix second: k
        Suffix first: k
        Suffix second:
    */
}
```

The prefix() and suffix() methods in std::smatch provide access to the parts of the string outside of the matched portion:

- **prefix():** Returns a std::sub_match object that represents the part of the string before the match.
- **suffix():** Returns a std::sub_match object that represents the part of the string after the match.

In the code above:

- **prefix().first:** Points to the first character before the match (which is the first character of the string, 'k'), since the match starts at the beginning of the string.
- **prefix().second:** Points to the same location as .first, which is the beginning of the string. This is why both prefix first and prefix second are printing 'k'.

Because there is no actual prefix in this case, both prefix().first and prefix().second are equal to the first character of the string.

- **suffix()** refers to the part of the string after the match. In this case, the match is "key1=value1", so the suffix is everything after that: " key2=value2 key3=value3".
- **suffix().first:** Points to the first character after the match, which is the space between "key1=value1" and "key2=value2". The character it points to is ' ' (a space), but since spaces are sometimes not visible in output, it is displaying the next visible character, which is 'k', from "key2=value2".
- **suffix().second:** Points to the end of the string, so dereferencing it is undefined behavior. In this case, the output doesn't display a value for suffix().second because it points to the end of the string, and dereferencing an iterator at the end is unsafe.

Let's consider the code example above, where we used a while loop and sub_match objects to get all matches in a string. Instead of using

```
sub_match<string::const_iterator> sm = capture_space[2];
curr = sm.second;
```

To update the curr iterator, let's use suffix.

```
string s = "key1=value1 key2=value2 key3=value3";

smatch capture_space;
string::const_iterator curr = s.begin();
while (regex_search(curr, s.cend(), capture_space, regex(R"(\s*(\w+)=(\w+)\s*)"))) {
    cout << "key: " << capture_space[1] << '\n' << "value: " << capture_space[2] << '\n';

    curr = capture_space.suffix().first;
}
```

Generates the same output

```
key: key1
value: value1
key: key2
value: value2
key: key3
value: value3
```

# Standard namespace

## 20.1 std::bind

### 20.1.1 std::placeholders

Before we get into std::bind, we must discuss std::placeholders

std::placeholders is a namespace in C++ that provides placeholder objects like std::placeholders::_1, std::placeholders::_2, and so on, which are used in conjunction with std::bind. These placeholders represent arguments that will be supplied when the resulting callable object (created by std::bind) is invoked.

Each placeholder corresponds to a positional argument that the function object will expect at the time of calling. For example:

- std::placeholders::_1 represents the first argument.
- std::placeholders::_2 represents the second argument, and so on.

std::bind in C++ is used to create a function object (or a callable) by binding specific arguments to a function or member function, essentially "fixing" some of its arguments ahead of time. This allows the resulting function object to be called later with fewer arguments, as some are already provided.

```cpp
int add(int a, int b) {
    return a + b;
}

auto addtwo = std::bind(add, 2, std::placeholders::_1);
cout << addtwo(4) << '\n'; // 6
```

Here, std::bind(add, 2, std::placeholders::_1) binds 2 as the first argument of add, leaving the second argument to be provided later. The placeholder _1 indicates where future arguments should be inserted.

### 20.1.2 Using std::ref with bind

std::ref is a utility in C++ that allows you to create reference wrappers for objects. It's commonly used when you need to pass objects by reference to functions that usually take arguments by value, such as when using function objects or lambdas with standard algorithms (like std::for_each or std::thread).

It provides a way to pass a reference to an object where an argument would otherwise be copied.

It's especially useful when the function you're working with doesn't explicitly accept references but you want to pass one anyway.

```cpp
void add(int& a) {
    ++a;
}

int local = 0;

auto f = std::bind(add, local);
f();
f();
cout << local << endl; // Outputs zero

auto f = std::bind(add, std::ref(local)); // Outputs two
f();
f();
cout << local << endl;
```

**Note:** std::ref makes a std::reference_wrapper<Type>...

```cpp
int local = 0;
std::reference_wrapper<int> rlocal = local;

auto f = std::bind(add, rlocal); // Outputs two
```

## 20.2 std::invoke

std::invoke is a utility function in C++ (introduced in C++17) that allows you to invoke a callable object in a uniform way, regardless of whether the callable is a regular function, a member function, or a function object (like a lambda or std::function). The main benefit of std::invoke is that it simplifies calling different types of callable objects without needing to worry about the specific calling syntax.

```
template< class F, class... Args >
decltype(auto) invoke(F&& f, Args&&... args);
```

### 20.2.1 Calling a regular function

```
int add(int a, int b) {
    return a + b;
}

int result = std::invoke(add, 2, 3);  // Calls add(2, 3)
```

### 20.2.2 Calling a member function

```
struct MyClass {
    int multiply(int x) { return x * 2; }
};

MyClass obj;
int result = std::invoke(&MyClass::multiply, obj, 5);  // Calls
↪  obj.multiply(5)
```

### 20.2.3 Accessing a member variable

```cpp
struct MyClass {
    int value;
};

MyClass obj{42};
int value = std::invoke(&MyClass::value, obj);  // Accesses obj.value
```

### 20.2.4 Calling a lambda

```cpp
auto lambda = [](int a, int b) { return a + b; };
int result = std::invoke(lambda, 2, 3);  // Calls lambda(2, 3)
```

## 20.3 std::exchange

std::exchange is a utility function in the C++ Standard Library that is used to replace the value of an object with a new value and return the old value. This can be very handy for implementing things like move operations, resetting variables, or swapping values efficiently.

```
template< class T, class U = T >
T exchange( T& obj, U&& new_value );
```

It takes a reference to an object obj and replaces it with a new value new_value. It returns the original value of the object before the replacement.

```
 int a = 5;
int b = std::exchange(a, 10); // a becomes 10, b takes the old value of a, which was 5
```

## 20.4 std::swap

Swap exchanges the values of two objects

```cpp
template< class T >
void swap( T& a, T& b );
```

```cpp
int x=5, y=10;
std::swap(x,y);

// Swaps the values of x and y
```

## 20.5 std::get

Allows access to elements from containers like std::tuple, std::pair, std::array, and std::variant either by index or type.

Tuple has no .at() method or subscript operator, so this utility function can be very useful to retrieve things from tuples.

```
std::tuple<int, string> v{1, "abc"};

cout << std::get<string>(v); // "abc"
cout << std::get<1>(v); // "abc"
cout << std::get<int>(v); // 1
cout << std::get<0>(v); // 1
```

## 20.6 std::tuple

In C++, a tuple is a fixed-size collection of elements that can hold objects of different types. Tuples are part of the <tuple> header and can store an arbitrary number of values of varying types in a single object.

### 20.6.1 std::tuple and std::make_tuple

You can create a tuple using std::tuple and std::make_tuple

```cpp
std::tuple<int, float, string> t = std::make_tuple(1,1.0f, "hello");
```

As we saw above, we can retrieve elements in a tuple with std::get

### 20.6.2 Modifying elements

You can modify elements directly if the tuple is non-const

```cpp
std::tuple<int, float, string> t = std::make_tuple(1,1.0f, "hello");
std::get<0> t = 12;
```

### 20.6.3 std::tuple_size

Use std::tuple_size to get the number of elements

```cpp
std::tuple<int, float, string> t = std::make_tuple(1,1.0f, "hello");

constexpr size_t tsize = std::tuple_size<tuple<int, float,string>>::value;
constexpr size_t tsize2 = std::tuple_size<decltype(t)>::value;
```

### 20.6.4 Unpacking with std::tie

We can unpack tuple elements into variables using std::tie:

```
std::tuple<int, float, string> t = std::make_tuple(1,1.0f, "hello");

int x; float y; string z;
std::tie(x,y,z) = t;
```

**Note:** std::tie requires existing variables to tie to the elements of the tuple. std::tie doesn't create new variables; it binds existing ones to the values in the tuple.

### 20.6.5 std::tie with std::ignore

If you don't need all the values from the tuple or pair, you can use std::ignore to ignore specific elements.

```
std::tuple<int, float, string> t = std::make_tuple(1,1.0f, "hello");

int x; [[maybe_unused]] float y; string z;
std::tie(x,std::ignore,z) = t;
```

### 20.6.6 std::optional

std::optional (introduced in C++17) is a utility that represents an optional value—a value that may or may not be present. It provides a safer and clearer alternative to using raw pointers or special values like nullptr or -1 for optional semantics.

Either contains a value or is empty (no value).

Useful Methods:

- std::optional<T>::has_value() – Checks if a value is present.
- std::optional<T>::value() – Accesses the contained value (throws if empty).
- std::optional<T>::value_or(default_value) – Returns the value or a default if empty.

Once set, it behaves like a `T`.

```
std::optional<int> divide(int a, int b) {
    if (b == 0) return std::nullopt; // No value
    return a / b; // Contains a value
}
```

std::nullopt is a constant of type std::nullopt_t. Used to indicate that an std::optional does not contain a value.

### 20.6.7 std::format

std::format (from <format>) provides a Python-like string formatting mechanism

```cpp
int age = 21;
std::string name = "Nate";

std::string message = std::format("Hello, {}! You are {} years old.", name, age);
```

# Views and Ranges

## 21.1 Views

A view is a lightweight, non-owning adaptor in the C++20 Ranges library that lazily transforms, filters, or otherwise manipulates a range without creating a new container. Views enable the efficient and expressive processing of data using pipelines.

- **Non-Owning:**: A view does not own the underlying data. It operates on a range or another view.
- **Lazy Evaluation:** Computations are deferred until the elements are accessed, which improves performance by avoiding unnecessary computations.
- **Composable:** Views can be chained together using the | operator, creating expressive pipelines.
- **Efficient**: Since views are non-owning and lazy, they avoid copying or storing intermediate results, minimizing memory overhead.

Views are built on the concept of range adaptors, which modify or filter ranges. For example:

- **std::ranges::views::filter:** Filters elements based on a predicate.
- **std::ranges::views::transform:** Transforms elements using a function.

### 21.1.1 Filter and transform

```
#include <ranges>
#include <vector>
#include <iostream>

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

    auto evens = nums | std::ranges::views::filter([](int n) { return n % 2 == 0; });

        for (int n : evens) {
            std::cout << n << " "; // Output: 2 4 6
        }
    }
```

```
auto squares = nums | std::ranges::views::transform([](int n) { return n * n; });
```

We can also combine views

```
auto evens_squared = v  filter([](int n)  return !(n % 2); ) transform([](int n) {return n*n;});
```

### 21.1.2 Iota

std::ranges::views::iota Generates a sequence of numbers.

```
auto numbers = std::ranges::views::iota(1, 10); // [1, 2, ..., 9]

for (auto i : std::ranges::views::iota(1,10)) cout << i << " ";
```

### 21.1.3 Take and drop

- **std::ranges::views::take:** Takes the first $n$ elements from a range.
- **std::ranges::views::drop:** Drops the first n elements from a range.

```
auto firstThree = numbers | std::ranges::views::take(3); // [1,
↪  2, 3]
auto afterThree = numbers | std::ranges::views::drop(3); // [4,
↪  5, ...]
```

## 21.2 The ranges library

The ranges library is an extension and generalization of the algorithms and iterator libraries that makes them more powerful by making them composable and less error-prone.

The library creates and manipulates range views, lightweight objects that indirectly represent iterable sequences (ranges).

## 21.3 std::span

std::span is a lightweight, non-owning view of a contiguous sequence of elements. Introduced in C++20, it provides a way to access and manipulate arrays, std::vectors, or other contiguous containers without copying or transferring ownership of the data.

```
int arr[] = {1,2,3,4};
std::span<int> sp(arr);

for (const auto& item : sp) cout << item;

vector<int> v{1,2,3,4};
std::span<int> sp(v.data(),3); // First three elements

for (const auto& item : sp) cout << item;
```

```
vector<int> v{1,2,3,4,5};

std::span<int> sp(v.data(),3);

cout << sp[0] << endl;
sp[0] = 5;

cout << "Observing v:" << endl;
for (const auto& item : v) cout << item << " ";
cout << endl << endl;

cout << "Observing span:" << endl;
for (const auto& item : sp) cout << item << " ";

/*
1
Observing v:
5 10 3 4 5

Observing span:
5 10 3
*/
```

Notice that when we change an element of *sp*, that same element of *v* gets changed, and when we change an element of *v*, that same element of *sp* gets changed.

### 21.3.1 What does it mean to be ”non-owning”

when we say that std::span is ”non-owning,” we mean that it does not manage the lifetime of the objects it refers to. Instead, it provides a lightweight view over a contiguous sequence of elements in memory without copying or taking ownership of the data.

- **No Memory Allocation or Deallocation:** std::span does not allocate or free memory—it merely observes an existing range of elements.
- **References Existing Data:** It simply stores a pointer to the first element and a size, allowing it to access a subset of an array, std::vector, or other contiguous memory.
- **Does Not Extend Lifetime:** If a std::span outlives the data it points to, it becomes dangling (undefined behavior), as it does not keep the underlying data alive.
- **Lightweight & Cheap to Copy:** Since std::span only holds a pointer and a size, copying a std::span is as cheap as copying two integers, unlike std::vector, which involves deep copies of data.

### 21.3.2 subspans

```
int arr[] = {1, 2, 3, 4, 5};
std::span<int> sp(arr, 5);
auto sub = sp.subspan(1, 3); // Elements 1 to 3
```

#### 21.3.3 Key member functions

- **.at()**
- **.size()**
- **.subspan()**
- **.data()**

# constexpr, consteval, and constinit

constexpr is a keyword introduced in C++11 and enhanced in later standards. It is used to indicate that the value of a variable, function, or object can be evaluated at compile time, provided all inputs and operations are also constant expressions. This helps in improving performance by allowing certain computations to be done during compilation rather than runtime.

- **Compile-Time Evaluation:** If possible, constexpr ensures computations are performed at compile time, resulting in faster execution.
- **Constants:** Variables declared as constexpr must be initialized with constant expressions.
- **Functions:** A constexpr function can be evaluated at compile time if called with constant arguments. It can also be called at runtime with non-constant arguments.

```
constexpr int x = 5; // Compile-time constant
constexpr int square(int n) { return n * n; }

constexpr int result = square(4); // Evaluated at compile time
```

```
constexpr int factorial(int n) {
    return (n <= 1) ? 1 : n * factorial(n - 1);
}

constexpr int result = factorial(5); // Computed at compile time
```

If result is not declared as constexpr, it would simply be a regular, non-constant integer variable. The compiler would still attempt to evaluate the factorial(5) call at compile time if the inputs and the function itself meet the requirements for a constexpr evaluation. However, since result is not explicitly marked as constexpr, its value would be treated as a runtime constant.

## 22.1 constexpr Objects

You can use constexpr constructors to define constant objects:

```
struct Point {
    int x, y;
    constexpr Point(int x_val, int y_val) : x(x_val), y(y_val) {}
};

constexpr Point p(3, 4); // p is a constant expression
```

## 22.2 More on constexpr variables

If a variable is declared as constexpr in C++, it means that:

- The value of the variable must be known and evaluated at compile time.
- The initializer for a constexpr variable must be a constant expression (an expression that the compiler can evaluate during compilation).

Consider the code

```
constexpr int x = 42; // Compile-time constant
constexpr int y = x + 10; // Also a compile-time constant
```

Here, both x and y are evaluated at compile time, and their values are embedded into the compiled program as constants.

constexpr variables are read-only after they are initialized. Once assigned, their value cannot be changed

If $x$ is not constexpr, it is treated as a regular variable, and the compiler cannot guarantee that its value is a compile-time constant.

Since the initializer for $y$ depends on $x$, and $x$ is not constexpr, $y$ would fail to meet the requirement of being initialized with a constant expression.

A constexpr variable can be used in contexts where a constant expression is required, such as

- Array sizes
- Template parameters
- static_assert conditions

Because the value of a constexpr variable is computed at compile time, it eliminates the need for runtime computation, which can improve performance.

## 22.3 When const Can Make a Compile-Time Constant

Whether a const variable is a compile-time constant depends on how and where it is initialized.

A const variable is considered a compile-time constant if its value is known at compile time. This typically occurs in the following scenarios

- **Literal Initialization**: If a const variable is initialized with a literal or a constant expression, it is a compile-time constant.

  ```
  const int x = 10;  // Compile-time constant
  const double pi = 3.14159;  // Compile-time constant
  ```

- **Constant Expressions**: If a const variable is initialized with a constant expression (an expression that can be evaluated at compile time), it is a compile-time constant.

```cpp
const int y = x + 5;  // Compile-time constant if `x` is a
    compile-time constant
const int z = sizeof(int) * 2;  // Compile-time constant
```

- **constexpr Variables:** The constexpr keyword explicitly indicates that a variable must be a compile-time constant. If a variable is declared with constexpr, it must be initialized with a constant expression.

```cpp
constexpr int a = 10;  // Compile-time constant
constexpr int b = a * 2;  // Compile-time constant
```

A const variable is not a compile-time constant if its value is determined at runtime. This happens in the following cases

- **Dynamic Initialization:** If a const variable is initialized with a value that is not known until runtime, it is not a compile-time constant.

```cpp
int input;
std::cin >> input;
const int c = input;  // Not a compile-time constant
```

- **Function Return Values:** If a const variable is initialized with the return value of a function (unless the function is constexpr), it is not a compile-time constant.

```cpp
int getValue() { return 42; }
const int d = getValue();  // Not a compile-time constant
```

## 22.4 if constexpr

if constexpr is a feature introduced in C++17 that allows conditional branching at compile-time. Unlike a regular if statement, the condition in if constexpr must be a compile-time constant expression that the compiler can evaluate during compilation. This enables writing more efficient, type-safe, and flexible code, especially in template programming.

- **Compile-Time Condition:** The condition inside if constexpr is evaluated during compilation.

  Only the branch corresponding to the true condition is compiled; other branches are ignored.

- **Dead Code Removal:** Unreachable branches are completely removed by the compiler, so there is no runtime overhead.

```cpp
template <typename T>
void printType(T value) {
    if constexpr (std::is_integral<T>::value) {
        std::cout << "Integral type: " << value << '\n';
    } else if constexpr (std::is_floating_point<T>::value) {
        std::cout << "Floating-point type: " << value << '\n';
    } else {
        std::cout << "Other type\n";
    }
}
```

## 22.5 Is constexpr const

Is constexpr const? Does it make read-only variables? The short answer is yes, constexpr implies const for variables, but not for functions or class members.

When you declare a variable as constexpr, it is implicitly const, meaning it cannot be modified after initialization.

```cpp
constexpr int x = 42;
x = 10;  // ERROR: x is read-only (implicitly `const`)
```

This is equivalent to

```cpp
const int x = 42;
```

## 22.6 consteval

consteval is a keyword introduced in C++20 that specifies a function must be evaluated exclusively at compile time. Unlike constexpr, which allows a function to be evaluated at either compile time or runtime, consteval ensures that the function is always evaluated at compile time. This makes it a stricter form of constexpr.

A consteval function can only be called in a context where the result can be computed at compile time.

If you try to call a consteval function in a runtime context, the code will not compile.

constexpr functions can be called at runtime if their arguments are not compile-time constants.

consteval functions cannot be called at runtime under any circumstances.

Since consteval functions are evaluated at compile time, they introduce no runtime overhead.

The result of the function is "baked into" the compiled code.

Understand that to call a consteval function, the functions must be called with compile time expressions

```
consteval int f(int x) {
    return x;
}
constexpr x = 20;
f(x);
```

Consider the following code

```
consteval int f(int x) {
    int y = 20;
    return x+y;
}
constexpr int x = 20;
cout << f(x) << endl;
```

Why does this work? Isn't `int y = 20` a runtime operation? The key here is understanding the distinction between compile-time evaluation and runtime behavior in the context of consteval functions. Let's clarify why int y = 20 inside a consteval function is not a runtime thing and how the compiler handles it.

A consteval function is required to be evaluated at compile time. This means that everything inside the function (including local variables like int y = 20) is processed and computed during compilation, not at runtime.

The compiler treats the entire body of a consteval function as a compile-time context.

When you declare a local variable inside a consteval function, such as int y = 20, the compiler does not allocate memory for it at runtime.

Instead, the compiler treats y as a compile-time constant because:

- It is initialized with a literal value (20), which is a compile-time constant.
- The function itself is required to be evaluated at compile time.

Therefore, you can essentially create constexpr variables without the use of constexpr or const if the variable is initialized inside a consteval function?

Next, consider

```
consteval void f() {
    int data;
    cout << "Enter: ";
    cin >>  data;
}
f();
```

This code violates the fundamental requirement of consteval functions: they must be evaluated entirely at compile time

A consteval function is required to be evaluated exclusively at compile time. This means that all computations and operations inside the function must be resolvable during compilation.

The compiler must be able to determine the result of the function without executing any runtime code.

## 22.7 constinit

the constinit keyword was introduced to ensure that a variable is initialized at compile-time with a constant expression. This keyword is used to enforce that the initialization of a variable is done in a way that is compatible with constant initialization, which can help catch errors early and improve performance by guaranteeing that the initialization happens at compile-time.

Unlike constexpr, constinit does not imply that the variable is immutable (const). The variable can still be modified at runtime, but its initial value must be a compile-time constant.

constinit is typically used with static or thread$_{l}ocalvariables, asthesearethekindsofvariablesthatbenefitfromcompil$
$timeinitialization.$

A variable cannot be declared with both constinit and constexpr because constexpr already implies compile-time initialization and immutability.

If the initialization expression is not a constant expression, the compiler will generate an error, helping to catch mistakes early.

Note that constinit cannot be used with local variables

## 22.8 constexpr and consteval functions are implicitly inline

All constexpr functions are implicitly treated as inline by the compiler. This is because constexpr functions are often evaluated at compile-time, and their definitions need to be available in every translation unit where they are used.

Like constexpr functions, consteval functions are also implicitly inline.

This ensures that their definitions are available wherever they are used, as they must always be evaluated at compile-time.

# Smart pointers

We know that pointers are important but are a source of trouble. One reason to use pointers is to have reference semantics outside the usual boundaries of scope. However, it can be very tricky to ensure that their lifetime and the lifetime of the objects they refer to match, especially when multiple pointers refer to the same object. For example, to have the same object in multiple collections , you have to pass a pointer into each collection, and ideally there should be no problems when one of the pointers gets destroyed (no "dangling pointers" or multiple deletions of the referenced object) and when the last reference to an object gets destroyed (no "resource leaks").

A usual approach to avoid these kinds of problems is to use "smart pointers." They are "smart" in the sense that they support programmers in avoiding problems such as those just described. For example, a smart pointer can be so smart that it "knows" whether it is the last pointer to an object and uses this knowledge to delete an associated object only when it, as "last owner" of an object, gets destroyed.

Note, however, that it is not sufficient to provide only one smart pointer class. Smart pointers can be smart about different aspects and might fulfill different priorities, because you might pay a price for the smartness. Note that with a specific smart pointer, it's still possible to misuse a pointer or to program erroneous behavior.

Since C++11, the C++ standard library provides two types of smart pointer:

- **Class shared_ptr**: for a pointer that implements the concept of shared ownership. Multiple smart pointers can refer to the same object so that the object and its associated resources get released whenever the last reference to it gets destroyed. To perform this task in more complicated scenarios, helper classes, such as weak_ptr, bad_weak_ptr, and enable_shared_from_this, are provided.
- **Class unique_ptr**: for a pointer that implements the concept of exclusive ownership or strict ownership. This pointer ensures that only one smart pointer can refer to this object at a time. However, you can transfer ownership. This pointer is especially useful for avoiding resource leaks, such as missing calls of delete after or while an object gets created with new and an exception occurred.

**Quick note**: When an object is created using the new operator in C++ and an exception occurs during its construction, C++ ensures that

- **Memory Allocated by new is Automatically Freed:** If the constructor of the object throws an exception, the memory allocated by new is automatically released to prevent a memory leak.

  This behavior is part of the C++ standard, ensuring robust exception safety during dynamic allocation.
- **The Destructor is Not Called:**: Since the object's construction is incomplete, its destructor will not be called. Only fully constructed objects have their destructors called.

Historically, C++98 had only one smart pointer class provided by the C++ standard library, class auto_ptr<>, which was designed to perform the task that unique_ptr now provides. However, due to missing language features, such as move semantics for constructors and assignment operators and other flaws, this class turned out to be difficult to understand and error prone. So, after class shared_ptr was introduced with TR1 and class unique_ptr was introduced with C++11, class auto_ptr officially became deprecated with C++11, which means that you should not use it unless you have old existing code to compile.

All smart pointer classes are defined in the <memory> header file.

## 23.1 Class shared_ptr

Almost every nontrivial program needs the ability to use or deal with objects at multiple places at the same time. Thus, you have to “refer” to an object from multiple places in your program. Although the language provides references and pointers, this is not enough, because you often have to ensure that when the last reference to an object gets deleted, the object itself gets deleted, which might require some cleanup operations, such as freeing memory or releasing a resource

So we need a semantics of “cleanup when the object is nowhere used anymore.” Class shared_ptr provides this semantics of shared ownership. Thus, multiple shared_ptrs are able to share, or “own,” the same object. The last owner of the object is responsible for destroying it and cleaning up all resources associated with it.

By default, the cleanup is a call of delete, assuming that the object was created with new. But you can (and often must) define other ways to clean up objects. You can define your own destruction policy. For example, if your object is an array allocated with new[], you have to define that the cleanup performs a delete[]. Other examples are the deletion of associated resources, such as handles, locks, associated temporary files, and so on

To summarize, the goal of shared_ptrs is to automatically release resources associated with objects when those objects are no longer needed (but not before).

### 23.1.1 Using Class shared_ptr

You can use a shared_ptr just as you would any other pointer. Thus, you can assign, copy, and compare shared pointers, as well as use operators * and ->, to access the object the pointer refers to. Consider the following example

```cpp
shared_ptr<string> pNico(new string("nico"));
shared_ptr<string> pJutta(new string("jutta"));

// capitalize person names
(*pNico)[0] = 'N';
pJutta->replace(0,1,"J");

// put them multiple times in a container
vector<shared_ptr<string>> whoMadeCoffee;
whoMadeCoffee.push_back(pJutta);
whoMadeCoffee.push_back(pJutta);
whoMadeCoffee.push_back(pNico);
whoMadeCoffee.push_back(pJutta);
whoMadeCoffee.push_back(pNico);

// print all elements
for (auto ptr : whoMadeCoffee) {
    cout << *ptr << " ";
}
cout << endl;

// overwrite a name again
*pNico = "Nicolai";

// print all elements again
for (auto ptr : whoMadeCoffee) {
    cout << *ptr << " ";
}
cout << endl;

// print some internal data
cout << "use_count: " << whoMadeCoffee[0].use_count() << endl;
```

After including <memory>, where shared_ptr class is defined, two shared_ptrs representing pointers to strings are declared and initialized:

Note that because the constructor taking a pointer as single argument is explicit, you can't use the assignment notation here because that is considered to be an implicit conversion. However, the new initialization syntax is also possible:

```cpp
shared_ptr<string> pNico = new string("nico"); // ERROR
shared_ptr<string> pNico{new string("nico")}; // OK
```

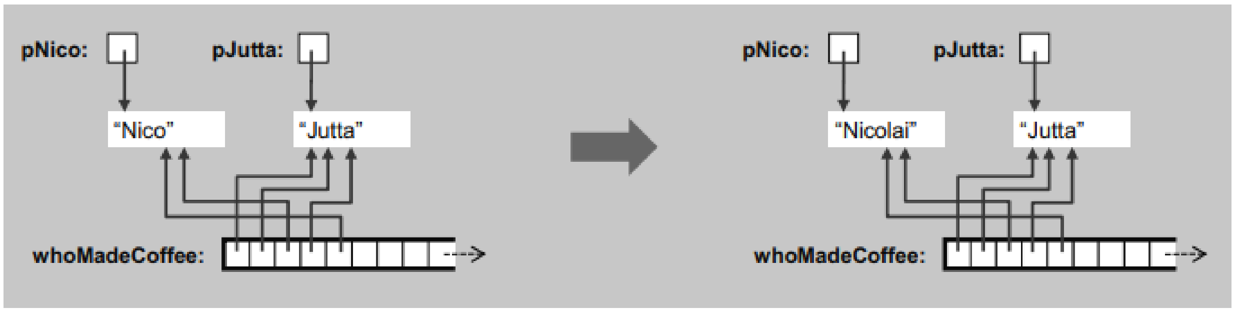
You can also use the convenience function make_shared() here:

```cpp
shared_ptr<string> pNico = make_shared<string>("nico");
shared_ptr<string> pJutta = make_shared<string>("jutta");
```

This way of creation is faster and safer because it uses one instead of two allocations: one for the object and one for the shared data the shared pointer uses to control the object

Alternatively, you can declare the shared pointer first and assign a new pointer later on. However, you can't use the assignment operator; you have to use reset() instead

```
shared_ptr<string> pNico4;
pNico4 = new string("nico"); // ERROR: no assignment for ordinary pointers
pNico4.reset(new string("nico")); // OK
```




We insert both pointers multiple times into a container of type vector<>. The container usually creates its own copy of the elements passed, so we would insert copies of strings if we inserted the strings directly. However, because we pass pointers to the strings, these pointers are copied, so the container now contains multiple references to the same object. This means that if we modify the objects, all occurrences of this object in the container get modified. Thus, after replacing the value of the string pNico, all occurrences of this object now refer to the new value.

The last row of the output is the result of calling use_count() for the first shared pointer in the vector. use_count() yields the current number of owners an object referred to by shared pointers has. As you can see, we have four owners of the object referred to by the first element in the vector: pJutta and the three copies of it inserted into the container

At the end of the program, when the last owner of a string gets destroyed, the shared pointer calls delete for the object it refers to. Such a deletion does not necessarily have to happen at the end of the scope. For example, assigning the nullptr to pNico or resizing the vector so that it contains only the first two elements would delete the last owner of the string initialized with nico.

### 23.1.2 Defining a Deleter

We can declare our own deleter, which, for example, prints a message before it deletes the referenced object:

```cpp
shared_ptr<string> pNico(new string("nico"),
[](string* p) {
    cout << "delete " << *p << endl;
    delete p;
});
...
pNico = nullptr; // pNico does not refer to the string any longer
whoMadeCoffee.resize(2); // all copies of the string in pNico
    ↪ are destroyed
```

The lambda function gets called when the last owner of a string gets destroyed. So the preceding program with this modification would print

```
delete Nicolai
```

when resize() gets called after all statements as discussed before. The effect would be the same if we first changed the size of the vector and then assigned nullptr or another object to pNico.

### 23.1.3 Dealing with Arrays

Note that the default deleter provided by shared_ptr calls delete, not delete[]. This means that the default deleter is appropriate only if a shared pointer owns a single object created with new. Unfortunately, creating a shared_ptr for an array is possible but wrong:

```cpp
std::shared_ptr<int> p(new int[10]); // ERROR, but compiles
```

So, if you use new[] to create an array of objects you have to define your own deleter. You can do that by passing a function, function object, or lambda, which calls delete[ ] for the passed ordinary pointer

```cpp
std::shared_ptr<int> p(new int[10],
[](int* p) {
    delete[] p;
});
```

You can also use a helper officially provided for unique_ptr, which calls delete[] as deleter

```cpp
std::shared_ptr<int> p(new int[10],
    ↪ std::default_delete<int[]>());
```

Note, however, that shared_ptr and unique_ptr deal with deleters in slightly different ways. For example, unique_ptrs provide the ability to own an array simply by passing the corresponding element type as template argument, whereas for shared_ptrs this is not possible:

```
std::unique_ptr<int[]> p(new int[10]); // OK
std::shared_ptr<int[]> p(new int[10]); // ERROR: does not compile
```

std::unique_ptr can handle arrays natively because its design allows for specialization to manage arrays directly. std::unique_ptr has a specialized version for arrays (std::unique_ptr<T[]>), which ensures that the correct delete[] operator is called for arrays.

In addition, for unique_ptrs, you have to specify a second template argument to specify your own deleter:

```
std::unique_ptr<int,void(*)(int*)> p(new int[10],
[](int* p) {
    delete[] p;
});
```

Note also that shared_ptr does not provide an operator [ ]. For unique_ptr, a partial specialization for arrays exists, which provides operator [ ] instead of operators * and ->. The reason for this difference is that unique_ptr is optimized for performance and flexibility

### 23.1.4 More on make_shared

Recall the two methods used above

```
std::shared_ptr<std::string> ptr(new std::string("Hello"));
std::shared_ptr<std::string> ptr = std::make_shared<std::string>("Hello");
```

For the first method, memory gets allocated twice, once for the std::string object, again for the control block. This results in two separate memory allocations, which is less efficient.

If an exception is thrown between the allocation of the object and the creation of the shared_ptr, the memory allocated for the object will leak because the shared_ptr never takes ownership of it. If the constructor of std::string throws an exception, the new operation has already allocated memory, and there's no shared_ptr to clean it up.

For the second method, memory gets allocated once. std::make_shared allocates memory for both the control block and the object in a single allocation. This reduces heap fragmentation and improves performance.

std::make_shared ensures that the object and the shared_ptr's control block are created together in a single step. If any part of the process throws an exception, no memory leak occurs because all cleanup is handled by make_shared.

### 23.1.5 Understanding the reference count

Consider the following code

```
auto a = make_shared<string>("Hello");
```

A std::shared_ptr named *a* is created. It dynamically allocates a std::string with the value "Hello". The shared_ptr internally manages the resource using reference counting. The reference count for the managed resource is 1 because a owns it.

When main ends, *a* goes out of scope. The destructor of *a* is called automatically, which

- Decrements the reference count for the managed resource.
- Since the reference count becomes 0, the std::string object is destroyed, and its memory is deallocated.

Next, consider

```
auto a = std::make_shared<string>("Hello");
shared_ptr<string> b = a;
```

We creates a dynamically allocated std::string with the value "Hello". A std::shared_ptr named *a* is created to manage this resource. The reference count for the resource is now 1.

Then, a new std::shared_ptr named *b* is created by copying *a* Both *a* and *b* now share ownership of the same std::string resource. The reference count for the resource is incremented to 2.

When main ends, both *a* and *b* go out of scope. Their destructors are called in the reverse order of creation:

- *b* is destroyed first, decrementing the reference count to 1.
- Then *a* is destroyed, decrementing the reference count to 0.

Since the reference count reaches 0, the managed resource (the std::string) is destroyed, and its memory is deallocated.

Understand that the reference count is stored in a control block that is managed internally by std::shared_ptr. This control block is allocated when the first std::shared_ptr is created for a given resource (in your case, when a is initialized using std::make_shared).

The control block is a separate structure that contains:

- **The reference count (use_count):** Tracks how many std::shared_ptr instances share ownership of the resource.
- **The weak reference count:** Tracks how many std::weak_ptr instances observe the resource.
- **A pointer to the managed resource**: (e.g., the std::string in your code).

Both *a* and *b* in the above code point to the same control block, which manages the reference count and the std::string resource.

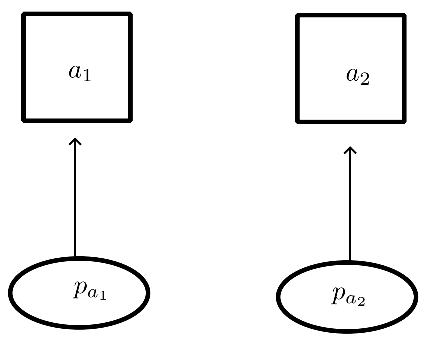
### 23.1.6 Cyclic references

Cyclic references occur when two or more objects reference each other in a way that creates a loop, preventing them from being deallocated properly

Consider the code

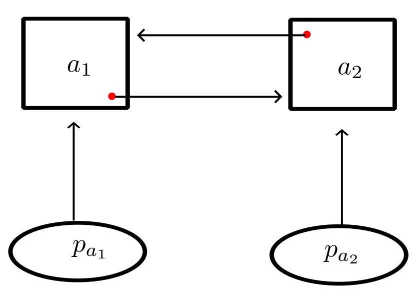
```
struct A {
    shared_ptr<A> adjacent;
};

shared_ptr<A> p_a1(new A());
shared_ptr<A> p_a2(new A());
```

So we have two objects, call them $a_1$, and $a_2$, and pointers $p_{a_1}$, and $p_{a_2}$ to them.




Now, define

```
p_a1->adjacent = p_a2;
p_a2->adjacent = p_a1;
```



When the program ends, $p_{a_2}$ will be destroyed, but the object that it points to still has a reference count of one, because $a_1$ "owns" it, so it will not be destroyed. Similarly, when $p_{a1}$ gets destroyed, the object that it refers to ($a_1$) still has a reference count of one, because $a_2$ still remains and "owns" $a_1$. This is a cyclic reference.

Introduced in the next section, we use weak_ptr to solve this issue. If we adjust the code

```cpp
struct A {
    weak_ptr<A> adjacent;
};

shared_ptr<A> p_a1(new A());
shared_ptr<A> p_a2(new A());

p_a1->adjacent = p_a2;
p_a2->adjacent = p_a1;
```

Our problem is resolved. A std::weak_ptr is a smart pointer in C++ that provides a non-owning reference to an object managed by a std::shared_ptr. Unlike std::shared_ptr, a std::weak_ptr does not increase the reference count of the object it observes, making it useful for breaking cyclic references.

When $p_{a_2}$ gets destroyed, the object that it manages ($a_2$) has no references to it. That is, its reference count will be zero because it has instead a weak_ptr to it. Then, when $p_{a_1}$ gets destroyed, $a_2$ will have already been destroyed, and thus $a_1$ has no references to it, and is therefore also destroyed.

### 23.1.7 Practical use of owner_before()

The owner_before method is used to compare ownership order of std::shared_ptr (or std::weak_ptr). It does not compare the actual pointer values but rather their control blocks (ownership metadata).

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

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

Let's examine a practical use case in std::set

```cpp
std::set<std::shared_ptr<int>, std::owner_less<std::shared_ptr<int>>> spSet;

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

spSet.insert(sp1);
spSet.insert(sp2);

std::cout << "Stored shared_ptr objects in set:\n";
for (const auto& sp : spSet) {
    std::cout << *sp << " ";
}
```

Here, std::owner_less<std::shared_ptr<int» ensures std::set is ordered based on ownership metadata, not the actual pointer values.

## 23.2 Class weak_ptr

The major reason to use shared_ptrs is to avoid taking care of the resources a pointer refers to. As written, shared_ptrs are provided to automatically release resources associated with objects no longer needed.

However, under certain circumstances, this behavior doesn't work or is not what is intended:

- One example is cyclic references. If two objects refer to each other using shared_ptrs, and you want to release the objects and their associated resource if no other references to these objects exist, shared_ptr won't release the data, because the use_count() of each object is still 1. You might want to use ordinary pointers in this situation, but doing so requires explicitly caring for and managing the release of associated resources
- Another example occurs when you explicitly want to share but not own an object. Thus, you have the semantics that the lifetime of a reference to an object outlives the object it refers to. Here, shared_ptrs would never release the object, and ordinary pointers might not notice that the object they refer to is not valid anymore, which introduces the risk of accessing released data

For both cases, class weak_ptr is provided, which allows sharing but not owning an object. This class requires a shared pointer to get created. Whenever the last shared pointer owning the object loses its ownership, any weak pointer automatically becomes empty. Thus, besides default and copy constructors, class weak_ptr provides only a constructor taking a shared_ptr

You can't use operators * and -> to access a referenced object of a weak_ptr directly. Instead, you have to create a shared pointer out of it. This makes sense for two reasons:

- Creating a shared pointer out of a weak pointer checks whether there is (still) an associated object. If not, this operation will throw an exception or create an empty shared pointer (what exactly happens depends on the operation used).
- While dealing with the referenced object, the shared pointer can't get released.

As a consequence, class weak_ptr provides only a small number of operations: Just enough to create, copy, and assign a weak pointer and convert it into a shared pointer or check whether it refers to an object.

### 23.2.1 Using a weak_ptr

std::weak_ptr is used as a companion to std::shared_ptr to prevent circular references, which can lead to memory leaks. It provides a way to observe an object managed by a std::shared_ptr without affecting its reference count

Characteristics:

- **No Ownership:** A std::weak$_ptrdoesnotowntheobjectitobserves.Itdoesnotcontributetothereferencecountofth$ $shared_ptritobserves.$**Temporary Access:**$Youmustconvertastd :: weak_ptrtoastd ::$ $shared_ptrtosafelyaccessthemanagedobject.$

- **Null Safety:** If the managed object has been destroyed, the std::weak$_ptrbecomesexpired.$

You create a std::weak_ptr from a std::shared_ptr:

```cpp
std::shared_ptr<int> sharedPtr = std::make_shared<int>(42);
std::weak_ptr<int> weakPtr = sharedPtr; // weakPtr observes
↪  sharedPtr
```

You must lock a std::weak_ptr to get a std::shared_ptr. This ensures the object is not destroyed while you're using it

```cpp
if (auto locked = weakPtr.lock()) { // lock() returns a
↪  std::shared_ptr
    std::cout << "Value: " << *locked << std::endl;
} else {
    std::cout << "The object has been destroyed." << std::endl;
}
```

You can check whether the object managed by a std::weak_ptr still exists using expired()

```cpp
if (weakPtr.expired()) {
    std::cout << "The object has expired." << std::endl;
}
```

You do not need to convert a std::shared_ptr back to a std::weak_ptr when you're done accessing the object.

- **Temporary Ownership via std::shared_ptr:** When you convert a std::weak_ptr to a std::shared_ptr using lock(), the resulting std::shared_ptr takes temporary ownership of the managed object for as long as it is in scope. Once the std::shared_ptr goes out of scope, it automatically releases this ownership, reducing the reference count.

- **std::weak_ptr Remains Unchanged:** The original std::weak_ptr remains intact and continues observing the managed object, if it still exists. There is no need to explicitly convert a std::shared_ptr back to a std::weak_ptr

## 23.3 Unique_ptr

The unique_ptr type, provided by the C++ standard library since C++11, is a kind of a smart pointer that helps to avoid resource leaks when exceptions are thrown. In general, this smart pointer implements the concept of exclusive ownership, which means that it ensures that an object and its associated resources are "owned" only by one pointer at a time. When this owner gets destroyed or becomes empty or starts to own another object, the object previously owned also gets destroyed, and any associated resources are released.

Class unique_ptr succeeds class auto_ptr, which was originally introduced with C++98 but is deprecated now. Class unique_ptr provides a simple and clearer interface, making it less error prone than auto_pointers have been

If bound to local objects, the resources acquired on entry get freed automatically on function exit because the destructors of those local objects are called. But if resources are acquired explicitly and are not bound to any object, they must be freed explicitly. Resources are typically managed explicitly when pointers are used.

A typical example of using pointers in this way is the use of new and delete to create and destroy an object

```
void f()
{
    ClassA* ptr = new ClassA; // create an object explicitly
    ... // perform some operations
    delete ptr; // clean up (destroy the object explicitly)
}
```

This function is a source of trouble. One obvious problem is that the deletion of the object might be forgotten, especially if you have return statements inside the function. There also is a less obvious danger that an exception might occur. Such an exception would exit the function immediately, without calling the delete statement at the end of the function. The result would be a memory leak or, more generally, a resource leak

Avoiding such a resource leak usually requires that a function catch all exceptions. For example:

```
void f()
{
    ClassA* ptr = new ClassA; // create an object explicitly
    try {
        ... // perform some operations
    }
    catch (...) { // for any exception
        delete ptr; // - clean up
        throw; // - rethrow the exception
    }
    delete ptr; // clean up on normal end
}
```

To handle the deletion of this object properly in the event of an exception, the code gets more complicated and redundant. If a second object is handled in this way, or if more than one catch clause is used, the problem gets worse. This is bad programming style and should be avoided because it is complex and error prone.

A smart pointer can help here. The smart pointer can free the data to which it points whenever the pointer itself gets destroyed. Furthermore, because it is a local variable, the pointer gets destroyed automatically when the function is exited, regardless of whether the exit is normal or is due to an exception. The class unique_ptr was designed to be such a smart pointer

A unique_ptr is a pointer that serves as a unique owner of the object to which it refers. As a result, an object gets destroyed automatically when its unique_ptr gets destroyed. A requirement of a unique_ptr is that its object have only one owner

Here is the previous example rewritten to use a unique_ptr:

```
// header file for unique_ptr
#include <memory>
void f()
{
    // create and initialize an unique_ptr
    std::unique<ClassA> ptr(new ClassA);
    ... // perform some operations
}
```

That's all. The delete statement and the catch clause are no longer necessary.

### 23.3.1 Using a unique_ptr

A unique_ptr has much the same interface as an ordinary pointer; that is, operator * dereferences the object to which it points, whereas operator -> provides access to a member if the object is a class or a structure:

However, no pointer arithmetic, such as ++, is defined (this counts as an advantage because pointer arithmetic is a source of trouble). Pointer arithmetic allows you to move a pointer beyond the bounds of an array or allocated memory. Accessing memory outside valid bounds results in undefined behavior

Note that class unique_ptr<> does not allow you to initialize an object with an ordinary pointer by using the assignment syntax. Thus, you must initialize the unique_ptr directly, by using its value:

```
std::unique_ptr<int> up = new int; // ERROR
std::unique_ptr<int> up(new int); // OK
```

A unique_ptr does not have to own an object, so it can be empty. This is, for example, the case when it is initialized with the default constructor:

```
std::unique_ptr<std::string> up;
```

You can also assign the nullptr or call reset() :

$$up = nullptr; up.reset();$$

n addition, you can call release(), which yields the object a unique_ptr owned, and gives up ownership so that the caller is responsible for its object now

```
std::unique_ptr<std::string> up(new std::string("nico"));
...
std::string* sp = up.release(); // up loses ownership
```

You can check whether a unique pointer owns an object by calling operator bool():

```
if (up) { // if up is not empty
    std::cout << *up << std::endl;
}
```

Instead, you can also compare the unique pointer with nullptr or query the raw pointer inside the unique_ptr, which yields nullptr if the unique_ptr doesn't own any object

```
if (up != nullptr) // if up is not empty
if (up.get() != nullptr) // if up is not empty
```

### 23.3.2 Transfer of Ownership by unique_ptr

A unique_ptr provides the semantics of exclusive ownership. However, it's up to the programmer to ensure that no two unique pointers are initialized by the same pointer:

```
std::string* sp = new std::string("hello");
std::unique_ptr<std::string> up1(sp);
std::unique_ptr<std::string> up2(sp); // ERROR: up1 and up2 own
↪ same data
```

Unfortunately, this is a runtime error, so the programmer has to avoid such a mistake.

This leads to the question of how the copy constructor and the assignment operator of unique_ptrs operate. The answer is simple: You can't copy or assign a unique pointer if you use the ordinary copy semantics. However, you can use the move semantics provided since C++11. In that case, the constructor or assignment operator transfers the ownership to another unique pointer

```cpp
// initialize a unique_ptr with a new object
std::unique_ptr<ClassA> up1(new ClassA);
// copy the unique_ptr
std::unique_ptr<ClassA> up2(up1); // ERROR: not possible
// transfer ownership of the unique_ptr
std::unique_ptr<ClassA> up3(std::move(up1)); // OK
```

After the first statement, up1 owns the object that was created with the new operator. The second, which tries to call the copy constructor, is a compile-time error because up2 can't become another owner of that object. Only one owner at a time is allowed. However, with the third statement, we transfer ownership from up1 to up3. So afterward, up3 owns the object created with new, and up1 no longer owns the object. The object created by new ClassA gets deleted exactly once: when up3 gets destroyed

The assignment operator behaves similarly:

```cpp
// initialize a unique_ptr with a new object
std::unique_ptr<ClassA> up1(new ClassA);
std::unique_ptr<ClassA> up2; // create another unique_ptr
up2 = up1; // ERROR: not possible
up2 = std::move(up1); // assign the unique_ptr
// - transfers ownership from up1 to up2
```

Here, the move assignment transfers ownership from up1 to up2. As a result, up2 owns the object previously owned by up1.

If up2 owned an object before an assignment, delete is called for that object. A unique_ptr that loses the ownership of an object without getting a new ownership refers to no object.

To assign a new value to a unique_ptr, this new value must also be a unique_ptr. You can't assign an ordinary pointer:

```cpp
std::unique_ptr<ClassA> ptr; // create a unique_ptr
ptr = new ClassA; // ERROR
ptr = std::unique_ptr<ClassA>(new ClassA); // OK, delete old object
// and own new
```

Assigning nullptr is also possible, which has the same effect as calling reset():

```cpp
up = nullptr; // deletes the associated object, if any
```

### 23.3.3 Source and Sink

The transfer of ownership implies a special use for unique_ptrs; that is, functions can use them to transfer ownership to other functions. This can occur in two ways:

1. A function can behave as a *sink* of data. This happens if a unique_ptr is passed as an argument to the function by rvalue reference created with std::move(). In this case, the parameter of the called function gets ownership of the unique_ptr. Thus, if the function does not transfer it again, the object gets deleted on function exit:

```
void sink(std::unique_ptr<ClassA> up) // sink() gets ownership
{
    ...
}
std::unique_ptr<ClassA> up(new ClassA);
...
sink(std::move(up)); // up loses ownership
```

2. A function can behave as a *source* of data. When a unique_ptr is returned, ownership of the returned value gets transferred to the calling context. The following example shows this technique

```
std::unique_ptr<ClassA> source()
{
    std::unique_ptr<ClassA> ptr(new ClassA); // ptr owns the new object
    ...
    return ptr; // transfer ownership to calling function
}
void g()
{
    std::unique_ptr<ClassA> p;
    for (int i=0; i<10; ++i) {
        p = source(); // p gets ownership of the returned object
        // (previously returned object of f() gets deleted)
        ...
    }
} // last-owned object of p gets deleted
```

Each time source() is called, it creates an object with new and returns the object, along with its ownership, to the caller. The assignment of the return value to p transfers ownership to p. In the second and additional passes through the loop, the assignment to p deletes the object that p owned previously. Leaving g(), and thus destroying p, results in the destruction of the last object owned by p. In any case, no resource leak is possible. Even if an exception is thrown, any unique_ptr that owns data ensures that this data is deleted.

### 23.3.4 unique_ptrs as Members

By using unique_ptrs within a class, you can also avoid resource leaks. If you use a unique_ptr instead of an ordinary pointer, you no longer need a destructor because the object gets deleted with the deletion of the member. In addition, a unique_ptr helps to avoid resource leaks caused by exceptions thrown during the initialization of an object. Note that destructors are called only if any construction is completed. So, if an exception occurs inside a constructor, destructors are called only for objects that have been fully constructed. This can result in resource leaks for classes with multiple raw pointers if during the construction the first new was successful but the second was not.

```cpp
class ClassB {
private:
    ClassA* ptr1; // pointer members
    ClassA* ptr2;
public:
    // constructor that initializes the pointers
    // - will cause resource leak if second new throws
    ClassB (int val1, int val2)
    : ptr1(new ClassA(val1)), ptr2(new ClassA(val2)) {
    }
    ~ClassB () {
        delete ptr1;
        delete ptr2;
    }
```

To avoid such a possible resource leak, you can simply use unique_ptrs:

```cpp
class ClassB {
private:
    std::unique_ptr<ClassA> ptr1; // unique_ptr members
    std::unique_ptr<ClassA> ptr2;
public:
    // constructor that initializes the unique_ptrs
    // - no resource leak possible
    ClassB (int val1, int val2)
    : ptr1(new ClassA(val1)), ptr2(new ClassA(val2)) {
    }
};
```

Note, first, that you can skip the destructor now because unique_ptr does the job for you. You also have to implement the copy constructor and assignment operator. By default, both would try to copy or assign the members, which isn't possible. If you don't provide them, ClassB also would provide only move semantics.

### 23.3.5 Dealing with arrays

By default, unique_ptrs call delete for an object they own if they lose ownership (because they are destroyed, get a new object assigned, or become empty). Unfortunately, due to the language rules derived from C, C++ can't differentiate between the type of a pointer to one object and an array of objects. However, according to language rules for arrays, operator delete[] rather than delete has to get called. So, the following is possible but wrong:

```cpp
std::unique_ptr<std::string> up(new std::string[10]); // runtime
    ERROR
```

Now, you might assume that as for class shared_ptr, you have to define your own deleter to deal with arrays. But this is not necessary.

Fortunately, the C++ standard library provides a partial specialization of class unique_ptr for arrays, which calls delete[ ] for the referenced object when the pointer loses the ownership to it. So, you simply have to declare:

```cpp
std::unique_ptr<std::string[]> up(new std::string[10]); // OK
```

Note, however, that this partial specialization offers a slightly different interface. Instead of operators * and ->, operator [ ] is provided to access one of the objects inside the referenced array

As usual, it's up to the programmer to ensure that the index is valid. Using an invalid index results in undefined behavior.

### 23.3.6 Double ownership

Consider the code

```cpp
int* ptr = new int(5);

std::unique_ptr<int> up1(ptr);
std::unique_ptr<int> up2(ptr);

/*
fish: Job 1, './bin' terminated by signal SIGABRT (Abort)
free(): double free detected in tcache 2
*/
```

The issue with this code is that it results in double ownership of the dynamically allocated memory, which leads to undefined behavior due to double deletion.

Since std::unique_ptr is designed for exclusive ownership, when up1 and up2 both attempt to delete the same pointer upon going out of scope, you get double deletion, which results in undefined behavior.

C++ does not automatically track raw pointers, and std::unique_ptr's constructor does not check if another std::unique_ptr is already managing the same pointer. Since ptr is just a raw pointer, nothing stops you from using it to initialize multiple unique_ptr instances.

To transfer ownership, we must use the move constructor

```cpp
std::unique_ptr<int> up1(ptr);
std::unique_ptr<int> up2(std::move(up1));
```

The move constructor will ensure that up1 is set to a valid state (nullptr)

## 23.4 Smart pointers to stack memory?

Consider

```
int x=10;
int* ptr = &x;

std::unique_ptr<int> up1(ptr);

/*
free(): invalid pointer
fish: Job 1, './bin' terminated by signal SIGABRT (Abort)
*/
```

The issue is that up1 tries to delete a non-heap-allocated memory when it goes out of scope, leading to undefined behavior (usually a crash).

When up1 is destroyed (out of scope), it tries to call delete ptr, which is invalid because $x$ was never allocated using new.

# Three way comparisons (spaceship operator) <compare>

Suppose we had a Point structure

```cpp
struct Point {
    int x,y;
};
```

Then if we tried to compare say to objects of type Point, C++ would not know how to handle the comparisons unless we overloaded the comparison operators. We would generally need to write overloads for all the comparison operators we need. <, >, <=, >=, ==, etc.

Since c++20, we can get the compiler to generate default comparison overloads with some simple syntax. Observe

```cpp
#include <compare>
struct Point {
    int x,y;

    auto operator <=>(const Point& other) const = default;
}

Point p1{1,2},p2{3,4};

cout << (p1 < p2) <<
    (p1 > p2) <<
    (p1 == p2) << endl;
```

## 24.1 Custom <=> logic

We can also define specific details for the spaceship operator

struct Rect  int width, height;
    auto operator<=>(const Rec other) const  return (width * height) <=> (other.width * other.height);  ;

## 24.2 Return types

- **std::strong_ordering:** For classes where all comparisons (<, <=, >, >=, ==, !=) are meaningful and consistent.
- **std::weak_ordering:** For cases where equivalence (==) is consistent with ordering but doesn't imply substitutability.
- **std::partial_ordering:** For cases where not all values are always comparable (e.g., floating-point numbers).

## 24.3 Spaceship on primitive types

Consider the spaceship operator no integers

```
int a = 5, b = 10;
std::strong_ordering c = a <=> b;

if (c == std::strong_ordering::less) {
    ...
} else if (c == std::strong_ordering::greater) {
    ...
} else if (c == std::strong_ordering::equal)
```

# Single Dispatch and Overload Resolution

## 25.1 Single dispatch

Single dispatch in C++ refers to the mechanism where the function that gets called is determined by the dynamic type of a single object at runtime. This is typically achieved using virtual functions in a class hierarchy.

- A base class defines a virtual function.
- Derived classes override that function.
- A pointer or reference to the base class calls the function.
- The actual function executed depends on the runtime type of the object.

Uses virtual table (vtable) lookup. Only considers one object's dynamic type (hence, single dispatch). C++ lacks multiple dispatch natively (where function resolution depends on multiple object types).

## 25.2 Overload resolution (compile time)

Involves selecting the best-matching function at compile-time based on function arguments. Uses static (compile-time) polymorphism. Does not rely on runtime object types.

"the function is chosen at compile-time" in the context of overload resolution means that the C++ compiler determines which function to call based on the types of arguments provided in the function call, without needing any runtime information.

At compile-time, the compiler

1. Examines the available overloaded functions (functions with the same name but different parameter types).
2. Matches the function call to the best candidate based on:
   - Exact match (best case).
   - Implicit conversions (if needed).
   - Function templates (if applicable).
3. Generates a direct function call to the resolved function.

Once the best match is found, the function call is hardcoded into the compiled binary, meaning there is no runtime lookup.

At runtime, no decision-making occurs—each call directly jumps to the compiled function.

## Bitfields

A bitfield in C++ is a special way of defining and manipulating groups of bits within a structure (struct) or class to optimize memory usage. It allows you to allocate a specific number of bits to a variable, making it useful for situations where you need to store multiple small values compactly.

A bitfield is declared inside a struct or class using an integer type followed by a colon (:) and the number of bits it should occupy.

```
struct Example {
    unsigned int a : 3;  // 3-bit field
    unsigned int b : 5;  // 5-bit field
    int c : 2;  // 2-bit field
};
```

$a$ takes 3 bits, $b$ takes 5 bits, and $c$ takes 2 bits. The type must be an integer type (int, unsigned int, char, etc.), but the exact storage behavior depends on the compiler. The total struct size depends on the compiler's memory alignment rules.

```
struct A{
    unsigned x : 1;
};
A a{0};
A b{1};
A c{10}; // Implicit truncation from 'int' to bit-field changes value from 10 to 0
A d{11}; // Implicit truncation from 'int' to bit-field changes value from 11 to 1
```

```
struct A{
    int x : 1;
};
```

Since $x$ is an signed int, valid values are $0, -1$, 1 changes to $-1$, even integers change to 0, odd integers change to $-1$.

Observe

```
struct A{
    unsigned x : 1;
};
cout << sizeof(A) << endl; // still 4
```

The reason sizeof(A) still returns 4 (or another word-aligned size depending on the system) is due to memory alignment and storage unit limitations in C++.

Even though the struct contains only a 1-bit field, the compiler allocates memory in units of int or a similar word-aligned type, typically 4 bytes (on most 32-bit and 64-bit systems).

So what is efficient about bitfields? Consider the example

```
struct A {
    unsigned a : 3;  // 3 bits
    unsigned b : 5;  // 5 bits
    unsigned c : 8;  // 8 bits
}; // Total: 4 bytes (or slightly more, but much smaller than 12)
```

The bitfield version can store three values within a single int (4 bytes) instead of requiring 12 bytes.

When multiple small values need to be stored, bitfields can significantly reduce memory usage.

### 26.0.1 Address operator on a bitfield?

Consider

```
struct A {
    int x : 3;
    int y;
};
A a;
cout << (&a.x < &a.y) << endl;
```

int var1 : 3; declares a bit-field, and you can not apply operator& to a bit-field.

The address-of operator & shall not be applied to a bit-field, so there are no pointers to bit-fields.

Bitfields in C++ are special because they do not have independent memory addresses. The key reason why you cannot take the address of a bitfield is that bitfields are packed into a single integer storage unit, rather than existing as independent variables in memory.

Bitfields do not have their own memory locations. Instead, multiple bitfields share a single underlying storage unit (typically an int or char).

Unlike normal variables, which reside at specific memory addresses, bitfields exist within a packed storage unit.

If we allowed &a.x, what would it return? There's no separate memory location for x, only an address for the whole storage unit.

Since bitfields are stored efficiently, the compiler manipulates bits directly using bitwise operations instead of treating them as full memory objects.

Allowing & would require breaking this optimization and making bitfields behave like regular variables, which defeats their purpose.

# C++ versions and their additions

- **c++11**:
  - Move Semantics
  - Automatic Type Deduction (auto keyword)
  - Lambdas
  - Range-Based For Loops
  - constexpr
  - Smart Pointers
  - Threading
  - nullptr
  - Uniform initialization with {} braces
  - threading and concurrency
  - std::array
  - std::unordered_map
  - std::tuple
  - <type_traits>
  - Trailing return type
  - Variadic templates
- **c++14**:
  - Relaxed constexpr to allow for control flow (if, for, etc) in constexpr functions.
  - Generic Lambdas,
  - lambdas parameters can be set auto,
  - std::make_unique,
  - Return type deduction for normal functions,
  - make_unique
  - function returns auto without TRT (Trailing return type)
  - decltype(auto)
- **c++17**:
  - Structured Bindings
  - if constexpr (Compile-time conditional branching)
  - std::optional
  - std::variant
  - Fold expressions
  - static constexpr members can be initialized inside the class definition (without inline keyword)
- **c++20**:
  - Ranges library
  - concepts library
  - modules

- coroutines
- spaceship operator
- std::span
- std::format

- **c++23**:
  - **std::expected**: Represents a value or an error, simplifying error handling.

    ```
    std::expected<int, std::string> result = 42;
    ```

  - **Deduction Guides**: For better type deduction in templates.
  - **Extended constexpr**: Nearly all standard library functions are constexpr.
  - **Improved Pattern Matching**: Early steps toward native pattern-matching constructs.