C++ Workbook

Mikael J. Gonsalves

July 11, 2023

Contents

Chapter 1

C++

1.1 Variables

```
    int: integers                   // 4 bytes
    double: floating-point numbers  // double 8 bytes
    char: individual characters     // 1 byte
    float:                          // 4 bytes
    long:                           // 4 or 8 bytes (platform dependent)
    long long:                      // 8 bytes
    bool: true/false                // 1 byte

    // use sizeof for size
bool flag;
std::cout << "Size of int: " << sizeof(numInt);
```

1.2 Enums

```
enum class Day {
    Monday,
    Tuesday,
    Wednesday,
};

int main() {
    Day today = Day::Tuesday;

    if (today == Day::Saturday || today == Day::Wednesday) {
    } else {}

    return 0;
}

enum Color {
```

```
    Red,
    Green,
    Blue
};

void printColor(Color color) {
    switch (color) {
        case Red:
            break;
        case Green: // ...
            break;
        case Blue: // ...
            break;
    }
}

int main() {
    Color favoriteColor = Color::Green;
    printColor(favoriteColor);

    return 0;
}
```

1.2.1 Enum with Array Mapping

```
enum class Fruit {
    Apple,
    Banana,
    Orange
};

const std::array<std::string, 3> fruitNames = {
    "Apple",
    "Banana",
    "Orange"
}

const std::string fruitNames[] = { // c-style array
                                   // size by initializer
    "Apple",
    "Banana",
    "Orange"
};

int main() {
    Fruit selectedFruit = Fruit::Banana;
    int fruitIndex = static_cast<int>(selectedFruit);

    std::cout << "Selected fruit: " << fruitNames[fruitIndex] << std::endl;

    return 0;
```

```
}
```

1.2.2 Enum with Vector Mapping

```
enum class Month {
    January,
    February,
    March // ...
};

const std::vector<std::string> monthNames = {
    "January",
    "February",
    "March" // ...
};

int main() {
    Month currentMonth = Month::May;
    int monthIndex = static_cast<int>(currentMonth);

    std::cout << "Current month: " << monthNames[monthIndex] << std::endl;

    return 0;
}
```

1.3 Arrays

A fixed-size stack-based container. Having the size type information gives more optimization oppotunities.

```
#include <array> // c++ 11
    std::array<char, 128> second = {'H', 'e', 'l'} // from library
                            // fixed size of 128
                            // has .begin(), .end(), .at(), .size()

    sint arr[] = {1, 2, 3}; // c-style array
                            // size determined by initializer's list
                            // fixed at compile-time

std::array<Type, Size> data

#include <numeric>
#include <array>

template<typename Value_Type>
std::array<Value_Type, 3> get_data(const Value_Type &v1, const Value_type &v2,
                                   const Value_type &v3)
{
    std::array<Value_Type 3> data;
    data[0] = v1;
```

```
    data[1] = v2;
    data[2] = v3;
    return data;
}

// no dynamic allocation,
// win-win scenario with knowing the size of the data struture at compile time.

template<typename> VT> // takes 3 parameters
std::array<VT, 3> get_data(const VT &v1, const VT &v2, const VT &v3)
{
    return {v1, v2, v3};
}

template<typename> VT> // takes 4 parameters
std::array<VT, 4> get_data(const VT &v1, const VT &v2, const VT &v3, const VT &v4))
{
    return {v1, v2, v3, v4};
}

...
// If only there was a way to avoid all this code duplication !!!
```

1.3.1 Dynamic Array Allocation

Achieved using pointers and dynamic memory allocation operators, such as ‘new‘ and ‘delete‘. While arrays are considered static containers, dynamic arrays allow you to allocate memory at runtime.

```
int size = 5; // desired size of the array
int* dynamicArray = new int[size]; // allocate memory for the array

// Access and modify elements of the dynamic array
dynamicArray[0] = 10;
dynamicArray[1] = 20;
// ...

// Deallocate the memory when it's no longer needed
delete[] dynamicArray;

#include <array>

int main() {
    std::array<int, 3> ar{1,2,3};

    int* dyn_ar = new int[4];

    dyn_ar[0] = 10;
    dyn_ar[1] = 20;
    dyn_ar[2] = 30;
    dyn_ar[3] = 40;
    dyn_ar[4] = 50;
```

```
    dyn_ar[5] = 60;
    dyn_ar[6] = 70;

    for (int i = 0; i < 7; i++) {
        std::cout << dyn_ar[i] << " ";
    }
    std::cout << std::endl; // prints 10, 20, 30, 40.. 70.

    delete[] dyn_ar;

    return 1;
}
```

‘new int[size]‘ dynamically allocates memory. ‘delete[] dynamicArray‘ deallocates the memory to avoid memory leaks.

Alternatively, using smart pointers or container classes like ‘std::vector‘ can help automate memory management and provide safer alternatives for dynamic arrays.

1.4 Vectors

Vectors are **dynamic array-like container that can grow or shrink.**

```
  std::vector<double> subway_adult; // value is 0.0 is default
  std::vector<double> location(2); // initialize two elements!
}

std::vector<char> vowels = {'a', 'e', 'i', 'o', 'u'};
std::vector vec{1,2,3}; // now possible!

int main(int argc, char* argv[]) {
    std::vector<std::string> arguments(argv + 1, argv + argc);

}

#include <vector>

template<typename Value_Type>
std::vector<Value_Type> get_data(const Value_Type &v1, const Value_Type &v2,
                                 const Value_Type &v3)

{
    std::vector<Value_Type> data;
    data.push_back(v1);
    data.push_back(v2);
    data.push_back(v3);
    return data;
}
```

1.5 Container Functions

```
begin() - returns an iterator on the first element
end() - returns an iterator after the last element
cbegin() - version constant of begin() and end()
cend() -
rbegin() - returns a reverse iterator pointing to the last element of a container
rend() - returns an iterator pointing to one position before the beginning
before_begin() - Only for forward_list!
                 returns an iterator pointing to the position
                 before the first element in a container.
cbefore_begin() - Only for forward_list!
                  Constant version of before_begin()
```

```
#include <vector>

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

    // Using rbegin() to iterate over elements in reverse order
    for (auto it = numbers.rbegin(); it != numbers.rend(); ++it) {
        std::cout << *it << " ";
    }

        // 5 4 3 2 1

    return 0;
}
```

```
#include <forward_list>

int main() {
    std::forward_list<int> numbers{2, 3, 4};

    // Inserting an element at the beginning using before_begin()
    numbers.insert_after(numbers.before_begin(), 1);

    // Printing the elements
    for (auto it = numbers.begin(); it != numbers.end(); ++it) {
        std::cout << *it << " ";
    }

        // 1 2 3 4

    return 0;
}
```

```
Forward_list doesn't have support for rbegin() / rend() or
crbegin() / crend()
```

```
size()     - returns the number of elements
             not available for forward_lists
max_size() - returns the maximum amount of elements that can be
             stocked in a container
resize()   - resize the container, to a new capacity
             not available for arrays
empty()    - returns true if container is empty, else false
```

1.5.1 Brace Initialization on the Return

```
#include <array>

template<typename> VT> // takes 1 parameter
std::array<VT, 1> get_data(const VT &v1)
{
    return {v1};
}
template<typename> VT> // takes 2 parameters
std::array<VT, 2> get_data(const VT &v1, const VT &v2)
{
    return {v1, v2};
}

template<typename> VT> // takes 3 parameters
std::array<VT, 3> get_data(const VT &v1, const VT &v2, const VT &v3)
{
    return {v1, v2, v3};
}
```

1.5.2 List Initialization

C++ initialization comes in many, many, many different flavors. But there's no denying that list-initialization (and in this case direct intitialization) has changed the way we use containers.

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

Standard c++11
```

1.5.3 insert()

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

    // Inserting at beginning
    numbers.insert(numbers.begin(), 0);

    // Inserting at index 3
```

```
    numbers.insert(numbers.begin() + 3, 10);

    // Inserting multiple elements at index 2
    numbers.insert(numbers.begin() + 2, {7, 8, 9});

    // Printing the modified vector
    for (const auto& num : numbers) {
        std::cout << num << " ";
    }

    return 0;
}

#include <list>
#include <algorithm>

struct Person {
    std::string name;
    int age;

    Person(const std::string& n, int a) : name(n), age(a) {}
};

int main() {
    std::list<Person> people;

    // Inserting elements in sorted order based on age
    people.insert(people.begin(), {"Alice", 25});
    people.insert(people.begin(), {"Bob", 30});
    people.insert(people.begin(), {"Charlie", 20});

    // Inserting an element after finding a specific person
    auto it = std::find_if(people.begin(), people.end(), [](const Person& p) {
        return p.name == "Bob";
    });
    if (it != people.end()) {
        people.insert(std::next(it), {"David", 27});
    }

    // Printing the list of people
    for (const auto& person : people) {
        std::cout << "Name: " << person.name << ", Age: " << person.age << std::endl;
    }

    return 0;
}
```

1.5.4 size()

```
std::cout << grocery.size() << "\n";
```

1.5.5 push_back() pop.back()

```
std::vector<std::string> dna = {"ATG", "ACG"};
dna.push_back("GTG"); // GTG added
std::cout << dna[2] << "\n"; // GTG

dna.pop_back(); // GTG has been removed

int main() {
    // int arr[5] = {1, 2, 3, 4, 5}; // c-styled array
    std::array <int, 5> arr = {1, 2, 3, 4, 5};
    std::array <int, 5> arr{1, 2, 3, 4, 5}; // all compiling

    int size = sizeof(arr) / sizeof(arr[0]);

    // Example with vector
    std::vector<int> vec = {1, 2, 3, 4, 5};
    int newValue = 6;

    // Push the new value to the vector
    vec.push_back(newValue);

    // Display the updated vector
    std::cout << "Updated vector: ";
    for (int num : vec) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

1.5.6 static_cast

Used with enums to convert between different enum types or to convert an enum value to an integer type. It allows for explicit and controlled type conversions, providing better type safety

```
int main() {
    Fruit selectedFruit = Fruit::Banana;
    int fruitIndex = static_cast<int>(selectedFruit);

    std::cout << "Selected fruit: " << fruitNames[fruitIndex] << std::endl;

    return 0;
}
```

1.5.7 Iterating over a Container

For Jason Turner, this construct, was one of the most difficult things to learn!

```
void print_map(const std::map<int, std::string> &map,
               const std::string &key_desc = "key",
               const std::string &value_desc = "value")
{
    for (std::map<int, std::string>::const_iterator data_itr = map.begin(); // this right here
         data_itr != map.end();
         ++data_itr)
    {
        std::cout << key_desc << ": '" << data_itr->first << "' "
                  << value_desc << ": '" << data_itr->second << "'\n";
    }
}

void print_map(const std::map<int, std::string> &map,
               const std::string &key_desc = "key",
               const std::string &value_desc = "value")
{
    for (auto data_itr = map.begin(); // auto, from c++11
         data_itr != map.end();
         ++data_itr)
    {
        std::cout << key_desc << ": '" << data_itr->first << "' "
                  << value_desc << ": '" << data_itr->second << "'\n";
    }
}

// If only there was some simple way to iterate over all the values in a container.....
```

1.5.8 Mismatched containers while looping

```
for (auto itr = container.begin();
    itr != container2.end();
    ++itr) {
    // oops! most of us have done this at some point
}
```

1.6 Size_t

```
template<typename Value_Type>
struct Data {
    Data(const std::size_t size)
      : data(new Value_Type[size]) // constructor
    {
    }

    ~Data() { delete [] data; }

    Value_Type *data;
};
```

Size of the dynamically allocated array ‘data‘ in the ‘Data‘ struct. std::size_t is an unsigned integer type. specifically designed to represent the size of objects or arrays.

In the ‘Data‘ struct, the constructor takes a ‘std::size_t‘ parameter named ‘size‘, which specifies the desired size of the ‘data‘ array. By using ‘std::size_t‘ as the parameter type, it ensures that the value provided for ‘size‘ is appropriate for representing the size of the array.

Inside the constructor, the ‘data‘ member is **allocated dynamically using ‘new‘**.

The size of the array is specified as the value of ‘size‘, which is of type ‘std::size_t‘. This ensures that the correct amount of memory is allocated for the array based on the given size.

In the destructor, ‘std::size_t‘ is not directly used, but it is common to use ‘std::size_t‘ for deallocating memory when deleting dynamically allocated arrays, as shown with ‘delete [] data;‘.

This ensures that the memory occupied by the ‘data‘ array is properly deallocated.

1.6.1 Type mismatch (int vs std::size_t)

Size() of containers returns a value of type size_t, an unsigned integral type specifically designed for representing sizes of objects.

Comparing a ‘size_t‘ value with an ‘int‘, causes a mismatch. Comparing s signed integer with an unsigned value causes UB.

To solve it, either change the type of the loop variable ‘i‘ to ‘size_t‘ or you can cast the container’s size to ‘int‘ for comparison.

```
Option 1: Change the loop variable type to ‘size_t‘:
for (size_t i = 0; i < container.size(); ++i) {
    // ...
}

Option 2: Cast the container’s size to ‘int‘:
for (int i = 0; i < static_cast<int>(container.size()); ++i) {
    // ...
}

for (int i = 0; i < container.size(); ++i) {
    // oops mismatched types
}
```

1.6.2 User Input

```
std::cout << "Enter your password: ";
std::cin >> password;

#include <iostream>
#include <string>
#include <cstdlib>

int main() {
    std::string answer;
    std::cout << "Place the output in" << output_dir << "? [y/yes, n/no]: ";
```

```
    std::cin >> answer;

    if (answer == "y" || answer == "yes") {
    } else if (answer == "n" || answer == "no") {
    } else {
        std::cout << "Invalid response, exiting";
        std::exit(1) // (failure)
    }
    return 0;
}
```

1.7 Iterators

1.7.1 Conditionals

```
  if (coin == 0) {
  } else {}
}
```

1.7.2 Switch Statements

```
int main() {
  int number = 9;
  switch(number) {
    case 1 : // ...
      std::cout << "case one";
      break;
    case 2 :
      break;
    default : // ...
      break;
  }
}
```

1.7.3 Loops

```
while (guess != 8) {
  std::cout << "Wrong guess, try again: ";
  std::cin >> guess;
}

for (int i = 0; i < 20; i++)  // incrementing
{}
for (int i = 20; i > 0; i--) // decrementing
{}
```

1.8 Ranges

```
#include <format>
#include <string_view>

void print_map(const auto &map, const std::string_view &key_desc = "key",
                                const std::string_view &value_desc = "value")
{
    const auto print_key_value = [&](const auto &data) {
        const auto &[key, value] = data;
        std::puts(std::format("{}: '{}' {}: '{}'",
                          key_desc, key, value_desc, value).c_str());
    };

    for_each(map, print_key_value);
}

#include <vector>
#include <ranges>

int main()
{
    std::vector<int> ints{1, 2, 3, 4, 5};
    auto even = [](int i){ return 0 == i % 2; };
    auto square = [](int i){ return i * i; }; // this is a lambda!
                                              // it defines an anonymous function
                                              // on the fly.

    for (int i : ints | std::view::filter(even) | std::view::transform(square)) {
        std::cout << i << ' ';
    }
}
```

1.8.1 Pipes for Ranges

‘—‘ are operators for composing ranges in C++20.

They chain range adaptors, transforming or filtering operations. Pipes take the output of one range and passes it as the input to the next range adaptor, allowing you to compose multiple operations on a range in a concise and readable way.

```
auto even = [](int i){ return 0 == i % 2; };
auto square = [](int i){ return i * i; }; // this is a lambda!

for (int i : ints | std::view::filter(even) | std::view::transform(square)) {
    std::cout << i << ' ';
}
```

‘ints‘: The input range of integers.
‘std::view::filter(even)‘: Filters the ‘ints‘ range, keeping only the even numbers.
‘std::view::transform(square)‘: Transforms the filtered range by squaring each element.

‘int i : ...‘: Iterates over the resulting transformed range and assigns each element to ‘i‘.
‘std::cout << i << ’ ’;‘: Prints each element ‘i‘ separated by a space.

Note that this code snippet utilizes C++20’s Ranges library, which requires a compatible compiler with proper language and library support.

1.8.2 Range-Based for Loops

Iteratore over elements of a container, such as an array, std::vector, or std::list. It doesn’t work for forward_list.

Works with anything that has begin() and end() members/functions, C-Style arrays and initializer lists.

```
for (const auto &element : container) {
    // eliminates both other problems
}

for (int element : first_three_multiples(8)) {
 std::cout << element << "\n";
}
std::string str = "Hello";
for (char character : str) {
    std::cout << character << std::endl;
}

template<typename Map>
void print_map(const Map &map, const std::string &key_desc = "key",
                               const std::string &value_desc = "value")
{
    for (const auto &data : map)
    {
        std::cout << key_desc << ": ’" << data_itr->first << "’ "
                  << value_desc << ": ’" << data_itr->second << "’\n";
    }
}

for (const auto &value : container) {} // for each element in the container

standard c++11

Never mutate the container itself while iterating inside of a ranged-for loop.
Use clang-tidy’s modernize-loop-convert check. Not using auto eases silent mistakes in your code.

template<typename Map>
void print_map(const Map &map, const std::string &key_desc = "key",
                               const std::string &value_desc = "value")
{
    for (const auto &data : map)
    {
        std::cout << key_desc << ": ’" << data.first << "’ "
```

```
                  << value_desc << ": '" << data.second << "'\n";
    }
}

// if only there was some way to make this data.first, data.second nonsense more readable!!
```

1.8.3 Accidental conversions

```
for (const int value : container_of_double) {
    // accidental conversion, possible warning
}
```

1.8.4 Accidental slicing

```
for (const base value : container_of_derived) {
    // accidental silent slicing
}

If container_of_derived holds objects of a derived class.
Base is the base class.
The loop is iterating over the container and assigning each
derived object to a base object (value) due to object slicing.

Object slicing occurs because the base object can only store
the base class's attributes and behavior. Additional defined
class attributes will be lost during the assignment or copy.

To avoid accidental slicing, you use pointers or references.

// no problem
for (const auto &value : container) {
    // no possible accidental conversion
}

Using pointers or references ensures that
the derived objects retain their specific attributes and behavior.

const auto & for non-mutating loops
auto & for mutating loops
auto &&
// only when you have to with weird types like std::vector<bool>,
// or if moving elements out of the container
```

1.8.5 Enable Ranged-Loop Warnings

```
GCC/ Clang?

-Wrange-loop-construct
-Wall // included in -Wall
```

1.9 Functions

```
void eat() {
  std::cout << "nom nom\n";
}

bool even(int num) {
  return ( num % 2 == 0 ? true : false );
  // this should be tested
}
```

1.9.1 Inline Functions (2 meanings)

The compiler inserts the function's body on the function call

```
inline
void eat() {
  std::cout << "nom nom\n";
}

// or

// function defined and declared in a single line in a header file
void Cookie::eat() {std::cout << "nom nom\n";}

Without inline in header is slower.

// source
std::string goodnight1(std::string thing1) {
  return "Goodnight, " + thing1 + ".\n";
}
// header
std::string goodnight1(std::string thing1);

With inline in header is faster. 0,004 milliseconds faster.

//inline in header, night.hpp
inline
std::string goodnight1(std::string thing1) {
  return "Goodnight, " + thing1 + ".\n";
}
```

1.9.2 Member Functions

Functions inside of classes.

```
class Musician {
private:
    int instruments;
```

```
public:
    // Getter function
    int getMyVariable() const {
        return myVariable;
    }

    // Setter function
    void setMyVariable(int newValue) {
        myVariable = newValue;
    }
};

int main() {
    MyClass obj;
    obj.setMyVariable(42);

    int value = obj.getMyVariable();
    std::cout << "MyVariable value: " << value << std::endl;

    return 0;
}
```

1.9.3 Public

```
class City {
  int population;

public: // accessible outside of the class
  void add_resident() {
    population++;
  }

private: // private to this class
  bool is_capital;
};

#include <string>

class Song { // header song.hpp
  std::string title;
  std::string artist;

public:
  void add_title(std::string new_title);
  std::string get_title();

  void add_artist(std::string new_artist);
  std::string get_artist();
};

#include "song.hpp"
```

```
void Song::add_title(std::string new_title) { // song.cpp
  title = new_title; // setters
}

std::string Song::get_title() { // getter
  return title;
}

void Song::add_artist(std::string new_artist) { // setters
  artist = new_artist;
}

std::string Song::get_artist() { // getters
  return artist;
}

#include "city.hpp"

class City {
  std::string name; // In header, city.hpp
  int population;

public:
  City(std::string new_name, int new_pop);

};

City::City(std::string new_name, int new_pop) // city.cpp
  : name(new_name), population(new_pop) {} // member initialized to passed values.

City ankara("Ankara", 5445000); // works, in main()

// in Header, in song.hpp
class Song {
  std::string title;
  std::string artist;

public:
  Song(std::string new_title, std::string new_artist); // constructor

  void add_title(std::string new_title);
  std::string get_title();

  void add_artist(std::string new_artist);
  std::string get_artist();
};
                ***
#include "song.hpp"

Song::Song(std::string new_title, std::string new_artist) in main, song.cpp
```

```
  : title(new_title), artist(new_artist) {}

void Song::add_title(std::string new_title) {
  title = new_title;
}
std::string Song::get_title() {
  return title;
}

void Song::add_artist(std::string new_artist) {
  artist = new_artist;
}

std::string Song::get_artist() {
  return artist;
}

#include <string> // in header file, song.hpp

class Song {
  std::string title;
  std::string artist;

public:
  Song(std::string new_title, std::string new_artist);

  std::string get_title();
  std::string get_artist();
};
                    ***
#include "song.hpp"

Song::Song(std::string new_title, std::string new_artist) // in main song.cpp
  : title(new_title), artist(new_artist) {}

std::string Song::get_title() {
  return title;
}

std::string Song::get_artist() {
  return artist;
}

#include <iostream> // in main.cpp
#include "song.hpp"

int main() {
  Song back_to_black("Back to Black", "Amy Winehouse");

  std::cout << back_to_black.get_title() << "\n";
  std::cout << back_to_black.get_artist() << "\n";
```

```
}
```

1.9.4 Overloading

Accepts many types as parameters.

Change behavior based on parameter's type.

```
// one must be true
    Each has different type parameters.
    Each has a different number of parameters.

// in num.cpp
// defining
int fancy_number(int num1, int num2) {
  return num1 - num2 + (num1 * num2);
}

int fancy_number(int num1, int num2, int num3) {
  return num1 - num2 - num3 + (num1 * num2 * num3);
} // different number of params.

int fancy_number(double num1, double num2) {
  return num1 - num2 + (num1 * num2);
} // different type of params.
// in num.hpp
// declaring
int fancy_number(int num1, int num2);
int fancy_number(int num1, int num2, int num3);
int fancy_number(double num1, double num2);
```

1.9.5 Parameters

```
void get_emergency_number(std::string emergency_number) {}

#include <iostream>
#include <vector>

struct ComplexType {
    int value;
    std::vector<int> data;
};

void processComplexType(const ComplexType& complexParam) {
    std::cout << "Value: " << complexParam.value << std::endl;
    std::cout << "Data:";
    for (int num : complexParam.data) {
        std::cout << " " << num;
    }
    std::cout << std::endl;
```

```
}

int main() {
    ComplexType complexObj;
    complexObj.value = 42;
    complexObj.data = {1, 2, 3, 4, 5};

    processComplexType(complexObj);

    return 0;
}

Many Parameters

double get_tip(double price, double tip, bool total_included) {
    get_tip(0.25, true, 45.50); // will not work. Order matters.
}

#include <iostream>

void name_x_times(std::string name, int x){
  while (x > 0) {
    std::cout << name << "\n";
    x--;
  }
}
```

1.9.6 Pass-by-value (in a function's params)

```
Yet, we can't modify the value of the arguments.
The variable passed are out of scope.

#include <iostream>

struct ComplexStruct {
    int value1;
    int value2;
};

void modifyValue(int val) {
    val = 100;
}

void modifyStruct(ComplexStruct obj) {
    obj.value1 = 200;
    obj.value2 = 300;
}

int main() {
    int x = 10;
```

```
    std::cout << "Before modif: x = " << x << std::endl;
    // Output: Before modif: x = 10

    modifyValue(x);  // Pass x by value to the modifyValue function

    std::cout << "After modif: x = " << x << std::endl;
    // Output: After modif: x = 10

    ComplexStruct obj;
    obj.value1 = 50;
    obj.value2 = 60;

    std::cout << "Before modif: obj.value1 = " << obj.value1 << std::endl;
    // Output: Before modif: obj.value1 = 50
    std::cout << "Before modif: obj.value2 = " << obj.value2 << std::endl;
    // Output: Before modif: obj.value2 = 60

    modifyStruct(obj);  // Pass obj by value to the modifyStruct function

    std::cout << "After modif: obj.value1 = " << obj.value1 << std::endl;
    // Output: After modif: obj.value1 = 50
    std::cout << "After modif: obj.value2 = " << obj.value2 << std::endl;
    // Output: After modif: obj.value2 = 60

    return 0;
}
```

1.9.7 Default Values

```
struct Chapters {
    bool IsToPrint = true;
};

struct Chapters {
    bool IsToPrint;

    Chapters() : IsToPrint(true) {} // with default constructor
};

int main() {
    Chapters c1;
    Chapters c2 = {false}; //  list initialization
                           // works when only one bool in struct

    // c1.IsToPrint will be true
    // c2.IsToPrint will be false

    return 0;
}
```

1.9.8 Default Arguments

```
// Declaration
void intro(std::string name, std::string lang = "C++");

// Definition
void intro(std::string name, std::string lang) {
    std::cout << "my" << name << "is" << lang << '\n';
}
```

1.9.9 Ls Function

```
// C++17 // g++ -std=c++17 myfile.cpp -o output-name

#include <filesystem>
namespace fs = std::filesystem;

int main()
{
    std::string path = "./foo";
    for (const auto &entry : fs::directory_iterator(path))
        std::cout << entry.path() << std::endl; // valid single statement synthax
                                                // If no optional block
                                                // only the statement following
                                                // the loop construct is executed.
}

&entry : fs::directory_iterator(path)
& creates a reference to the elements of the fs::directory_iterator object.
```

1.9.10 Standard Library Functions

```
C++ considers system architecture.
The libraries are differents as well?

Listing files in a directory is different in both OS.
```

1.9.11 Return Type Deduction for Normal Functions

Not to be underestimated, this allows for the creation of powerful higher order function constructs, among other things.

```
auto get_thing() {
    struct Thing {}; // creating my own type inside the function

    return Thing{};
}
```

1.10 Reference

1.10.1 References Variables

A second name for an existing variable

```
int &sonny = songqiao; // Sonny, a reference to songqiao;

\\ Changes to the reference happens to the original.
\\ Aliases cannot be changed to alias something else.

  int soda = 99;
  int &pop = soda;
  pop++; // soda and pop equal at 100

Using a reference (&) instead of making a copy of the elements is more efficient,
especially when dealing with large objects or containers.

By using a reference, the loop avoids creating a new copy of each element
, reducing unnecessary memory usage and improving performance.

#include <iostream>

struct ComplexStruct {
    int value1;
    int value2;
};

void modifyStruct(ComplexStruct& ref) {
    ref.value1 = 100;
    ref.value2 = 200;
}

int main() {
    int x = 10;
    int& ref1 = x;  // Reference variable ref1 refers to x

    std::cout << "x: " << x << std::endl;        // Output: x: 10
    std::cout << "ref1: " << ref1 << std::endl;  // Output: ref1: 10

    ref1 = 20;  // Modifying ref1 will also modify x

    std::cout << "x: " << x << std::endl;        // Output: x: 20
    std::cout << "ref1: " << ref1 << std::endl;  // Output: ref1: 20

    ComplexStruct obj;
    obj.value1 = 50;
    obj.value2 = 60;
    ComplexStruct& ref2 = obj;  // Reference variable ref2 refers to obj

    std::cout << "obj.value1: " << obj.value1 << std::endl;      // Output: obj.value1: 50
```

```
    std::cout << "obj.value2: " << obj.value2 << std::endl;      // Output: obj.value2: 60
    std::cout << "ref2.value1: " << ref2.value1 << std::endl;    // Output: ref2.value1: 50
    std::cout << "ref2.value2: " << ref2.value2 << std::endl;    // Output: ref2.value2: 60

    modifyStruct(ref2);  // Modifying ref2 will also modify obj

    std::cout << "obj.value1: " << obj.value1 << std::endl;      // Output: obj.value1: 100
    std::cout << "obj.value2: " << obj.value2 << std::endl;      // Output: obj.value2: 200
    std::cout << "ref2.value1: " << ref2.value1 << std::endl;    // Output: ref2.value1: 100
    std::cout << "ref2.value2: " << ref2.value2 << std::endl;    // Output: ref2.value2: 200

    return 0;
}
```

1.10.2 Dereference

Obtain the pointer's pointed value.

```
Declaration? * creates a pointer.
Not a declaration? * is a dereference operator.

int main() {
  int power = 9000;
  int* ptr = &power;
  std::cout << *ptr; // prints 9000
}

int main() {
    int x = 10;
    int* ptr = &x;

    std::cout << "Value of x: " << x << std::endl;
    // Output: Value of x: 10

    std::cout << "Address of x: " << &x << std::endl;
    // Output: Address of x: 0x7ffeebdbcbe4

    std::cout << "Value of ptr: " << ptr << std::endl;
    // Output: Value of ptr: 0x7ffeebdbcbe4

    std::cout << "Dereferenced value of ptr: " << *ptr << std::endl;
    // Output: Dereferenced value of ptr: 10

    *ptr = 20;  // Dereference ptr and assign a new value

    std::cout << "Modified value of x: " << x << std::endl;    // Output: Modified value of x: 20

    return 0;
}
```

1.10.3 Pass-by-reference (in a funtion's params)

Now, when called, the function can modify the argument's value. Avoid making copies of a variable/object for performance reasons.

```
//Calling swap_num(), variables a and b's value will be modified
//because they are passed by reference

void swap_num(int &i, int &j) {
  int temp = i;
  i = j;
  j = temp;
}

int main() {
  int a = 100;
  int b = 200;

  swap_num(a, b);

  std::cout << "A is " << a << "\n";
  std::cout << "B is " << b << "\n";
}

WHEN? To MODIFY the arguments' values.

int triple(int &i) {
  i = i * 3;

  return i;
}

int main() {
  int num = 1;
  std::cout << triple(num) << "\n";
}

template<typename T>
void modifyVector(std::vector<T>& vec) {
    for (auto& element : vec) {
        element *= 2;
    }
}

template<typename T>
void printVector(const std::vector<T>& vec) {
    for (const auto& element : vec) {
        std::cout << element << " ";
    }
    std::cout << std::endl;
}
```

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

    std::cout << "Original vector: ";
    printVector(numbers);  // Output: Original vector: 1 2 3 4 5

    modifyVector(numbers);

    std::cout << "Modified vector: ";
    printVector(numbers);  // Output: Modified vector: 2 4 6 8 10

    return 0;
}
```

1.10.4 Const Reference

```
int triple(int const &i) { // save computational cost
  return i * 3; // don't make a copy of the argument
}

int square(int const &i) {
  return i * i;
}

int main() {
  int side = 5;
  std::cout << square(side) << "\n";
}
```

1.10.5 Reference Operator

```
int soda = 99; // in a declaration
int &pop = soda; // reference operator
pop++;
```

1.10.6 Address Operator

Not a declaration? It is an address operator.

```
int porcupine = 3;
std::cout << &porcupine << "\n";

// 0x7ffd7caa5b54
```

1.10.7 Boarding References

```
auto get_data(auto && ... params)
{
    return std::array{std::forward<decltype(params)>(params)...};
```

```
}

// no unnecessary copies of params, guaranteed copy/move elisions of return value.
```

This does not make c++ easier to teach, but it allows for high level efficiency. At this point, is get_data() even necessary?

1.10.8 Rvalue References

Difficult to teach, but allows to more accurately reason about the lifetime of objects. C++11.

1.10.9 Guaranteed Copy Elision

Compilers have "always done copy elision on return values but c++17 guarantees it in some situations:

```
struct S {
    S() = default;
    S(S&&) = delete;
    S(const S &) = delete;
};

auto s_factory(){
    return S{}; // compiles in C++17, neither a copy nor a move.
}
```

1.11 Pointers

```
// Pointers store a memory address. If possible, avoid pointers!

int* number; // Declaring
double* decimal;
char* character;

int gum = 8;
int* ptr = &gum; // int* [declare a pointer]
                 // ptr [pointer's name]
                 // &gum [store gum's memory address in ptr]

int* number; // Declaration standard
int *number;
int * number; // All Synthaxically valid

Certainly! Here are two complex examples showcasing the use of pointers in C++:

Example 1: Pointer to Pointer
‘‘‘cpp
#include <iostream>

int main() {
```

```
    int value = 5;
    int* ptr = &value;
    int** ptrToPtr = &ptr;

    std::cout << "Value: " << value << std::endl;
    std::cout << "Value via pointer: " << *ptr << std::endl;
    std::cout << "Value via pointer to pointer: " << **ptrToPtr << std::endl;

    return 0;
}
```
In this example, we declare a variable `value` and initialize it with the value 5. We then create a p

```
// Dynamic Memory Allocation

int main() {
    int size = 5;
    int* dynamicArray = new int[size];

    for (int i = 0; i < size; ++i) {
        dynamicArray[i] = i + 1;
    }

    std::cout << "Dynamic Array: ";
    for (int i = 0; i < size; ++i) {
        std::cout << dynamicArray[i] << " ";
    }
    std::cout << std::endl;

    delete[] dynamicArray;

    return 0;
}
```

We allocate dynamic memory for an integer array of size 5 using the `new` keyword.
We then populate the array with values from 1 to 5.
Finally, we print and release the allocated memory using `delete[]` to avoid memory leaks.

1.11.1 Null pointers

```
// Ptr to a yet unknown memory address. Dangerous stuff!

int* ptr; // declared, not yet initialized
```

1.11.2 Nullptr - null pointers since C++11

```
int* ptr = nullptr; // type safe pointer with nullptr, replacement for NULL
```

1.12 Classes

```
A C++ class is a user-defined type.

class City {
}; // needs semicolon
```

1.12.1 Class Components

```
Class components are called class members
Attributes and methods are class members

class City {
  int population; // attribute

public:
  void add_resident() { // method
    population++;
  }
};
```

1.13 Constructor

```
City montreal;
montreal.population = 20000;

montreal.get_population();

class Song { // header file, song.hpp

  std::string title;

public:
  void add_title(std::string new_title);
  std::string get_title();
};

void Song::add_title(std::string new_title) {
  title = new_title; // main file, song.cpp
}

std::string Song::get_title() {
  return title;
}
int main() {
  Song electric_relaxation;
  electric_relaxation.add_title("Electric Relaxation");
  std::cout << electric_relaxation.get_title();
}
```

1.14 Destructors - prevent memory leaks

```
class City { // In Header, city.hpp
  std::string name;
  int population;

public:
  City(std::string new_name, int new_pop);
  ~City(); // destructor
};

City::~City() { // in main, city.cpp
  // any final cleanup
}

class Song { // in header, .hpp
  std::string title;
  std::string artist;

public:
  Song(std::string new_title, std::string new_artist);
  ~Song();

  std::string get_title(); // getters
  std::string get_artist();
};

Song::Song(std::string new_title, std::string new_artist) // .cpp
  : title(new_title), artist(new_artist) {}

Song::~Song() { // aded destructor
  std::cout << "Goodbye " << title << "\n";
}

struct Double_Data {

    Double_Data(const std::size_t size)
      : data(new double[size] /// allocate
    {
    }

    ~Double_Data() { /// destructor
      delete [] data; // free
    }

    double *data;

};
```

1.14.1 Automatic Destructors (no necessary calls)

```
The object moves out of scope.
The object is explicitly deleted.
When the program ends.
```

1.15 Object lifetime

1.15.1 Automatic

```
For format see p.44 filetime puzzler book 1

S object_1("a","t");
```

1.15.2 Thread_local

```
thread_local object_2("a","t");
```

1.15.3 Static

```
static object_3("a","t");
```

1.15.4 Dynamic

```
delete new S *object_4("a","t");
```

1.15.5 Deterministic Object Lifetime and Destruction

Constructor / Destruct pairs (RAII) combined with scoped values give us determinism that removes the need for things like 'finally'

```
#include <string>

void some_func() {
    std::string s{"Hello"}; // allocate a string
} // automatically free it when scope exits

Standard: c++ 98
```

1.15.6 For doubles, ints and floats

```
// Handy dandy tool for doubles
struct Double_Data {
    Double_Data(const std::size_t size) : data(new double[size] { }
    ~Double_Data() { delete [] data; }
```

```
    double *data;
   };

// I want one for ints!
struct Int_Data {
    Int_Data(const std::size_t size) : data(new int[size] { }
    ~Int_Data() { delete [] data; }
    int *data;
   };

// I want one for floats!
struct Float_Data {
    Float_Data(const std::size_t size) : data(new float[size] { }
    ~Float_Data() { delete [] data; }
    float *data;
   };

// If only there was some way to avoid repeating ourselves here!!
```

1.15.7 Sum

```
// I want to sum the values

double sum_data(const Double_Data &d) {
    return d.data[0] + d.data[1] + d.data[2];
}

// If only ther was some utility to automatically add these values up....
```

1.16 Span

A non-owning, lightweight and flexible view over a contiguous sequence of elements. It allows you to work with a range of elements without owning or managing the underlying memory. It provides a way to represent a view into an array or any other contiguous sequence of elements.

`The ‘span‘ type was introduced in C++20 as part of the ‘<span>‘ header.`

The elements can be of any type, such as fundamental types (e.g., ‘int‘, ‘double‘), user-defined types, or even pointers.

A ‘span‘ is a pointer to the first element and the number of elements in the sequence. It provides various member functions to access and manipulate the elements, similar to other standard containers like ‘std::vector‘ or ‘std::array‘.

1.16.1 Span Advantage

Work with existing arrays without making copies.

Pass and manipulate data ranges, such as sub-arrays or portions of containers, without memory allocation overhead or ownership.

It allows you to write generic functions that operate on different containers without requiring specific container types. Since ‘span‘ is a non-owning view, **it can be used to work with the array without copying its elements**.

```
#include <span>

void printSpan(std::span<int> sp) {
    for (int element : sp) {}
}

int main() {
    std::array<int,5> arr{1, 2, 3, 4, 5};
    std::span<int> span(arr, 5);

    printSpan(span); // takes span and prints elements
    return 0;
}
```

‘Span‘ represents a view of the array with 5 elements. The ‘printSpan()‘ function takes a ‘span‘ as a parameter and prints its elements.

1.17 Templates

The ultimate in the DRY principle. Write a template that has types and values filled in at compile-time.

```
// in Header
template <typename meaningful_Type>
meaningful_Type get_smallest(meaningful_Type num1, meaningful_Type num2) {
  return num2 < num1? num2 : num1;
}

int main() {
  std::cout << get_smallest(100, 60) << "\n";
  std::cout << get_smallest(2543.2, 3254.3) << "\n";
}

template types are generated by the compiler at compile time
Do not need any kind of type-erasure (like Java generics do)

Highly efficient runtime code possible,
as good as (or better than) hand writing the various options
template system is Turing complete (not necessarily a good thing)

    template<typename SomeType>
    struct S { // struct can do anything it wants with this type
    };

// declare a class template that can hold anything we want
template<typename Value_Type>
```

```
struct Data {
    Data(const std::size_t size)
      : data(new Value_Type[size])
    {
    }

    ~Data() { delete [] data; }

    Value_Type *data;
};

// declare a function template that take 3 params of the same type
// and passes that type on to the 'Data' template

template<typename Value_Type>
Data<Value_Type> get_data(const Value_Type &v1, const Value_Type &v2,
                          const Value_type &v3)
{
    {Data<Value_Type> d(3);
    d.data[0] = v1; d.data[1] = v2; d.data[2] = v3;
    return d;
}
```

1.17.1 Variadic Templates

Drastic simplification of code needing to match a variable number of parameters. Absolutely critical for maintainable implementations of things like std::function!

```
#include <array>

// require at least one parameter and it sets the type
template<typename VT, typename ... Params>
std::array<VT, sizeof...(Params)+1> get_data(const VT &v1, const Params& ...params)
{
    return {v1, params...};
}

                            ***

#include <array>

template<typename VT, Typename ... P> // variadic template
std::array<VT, sizeof...(P)> get_data(const P & ... params) // param expansion
{
    return {params...}; // pack expansion
}

standard C++11

#include <array>
```

```
template<typename> VT>
std::array<VT, 3> get_data(const VT &v1, const VT &v2, const VT &v3)
{
    std::array<VT, 3> data;
    data[0] = v1; data[1] = v2; data[2] = v3;
    return data;
}
// But I would like a version of this that takes only 1 parameter

#include <array>

template<typename> VT>
std::array<VT, 2> get_data(const VT &v1, const VT &v2)
{
    std::array<VT, 2> data;
    data[0] = v1; data[1] = v2;
    return data;
}

// Only 1 parameter

#include <array>

template<typename> VT>
std::array<VT, 1> get_data(const VT &v1)
{
    std::array<VT, 1> data;
    data[0] = v1;
    return data;
}

#include <array>

template<typename> VT> // takes 1 parameter
std::array<VT, 1> get_data(const VT &v1)
{
    std::array<VT, 1> data;
    data[0] = v1;
    return data;
}
template<typename> VT> // takes 2 parameters
std::array<VT, 2> get_data(const VT &v1, const VT &v2)
{
    std::array<VT, 2> data;
    data[0] = v1; data[1] = v2;
    return data;
}

template<typename> VT> // takes 3 parameters
std::array<VT, 3> get_data(const VT &v1, const VT &v2, const VT &v3)
{
```

```
    std::array<VT, 3> data;
    data[0] = v1; data[1] = v2; data[2] = v3;
    return data;
}

// if only there was a way to initialize the array values in one step...

// This bothers.
template<typename Value_Type>
struct Data {
    Data(const std::size_t size)
      : data(new Value_Type[size])
    {
    }

    ~Data() { delete [] data; }

    Value_Type *data;
};

// declare a function template that take 3 params of the same type
// and passes that type on to the 'Data' template

template<typename Value_Type>
Data<Value_Type> get_data(const Value_Type &v1, const Value_Type &v2,
                          const Value_type &v3)
{
    {Data<Value_Type> d(3);
    d.data[0] = v1; d.data[1] = v2; d.data[2] = v3;
    return d; // This only works because of the copy elision that compilers have 'always' implemented

    // If only there was some way to contain a set of values
    that has already taken care of these issues...
    // * Heap overflow problems I believe
}

#include <array>

// require at least one parameter and it sets the type
template<typename VT, typename ... Params>
std::array<VT, sizeof...(Params)+1> get_data(const VT &v1, const Params& ...params)
{
    return {v1, params...};
}

// auto doesn't work well here
// if only there was a way to deduce the type of the array being returned......
```

1.17.2 CTAD - Class Template Argument Deduction

Just how function template arguments have always been deduces class templates can be as of c++17.

```
#include <array>

template<typename VT, typename ... Params>
auto get_data(const VT &v1, const Params & ... params)
{
    return std::array{v1, params...}; // auto deduced size/type
}

std::vector vec{1,2,3}; // now possible!
and now we can simplify the template arguments

/include <array>

template<typename ... Params>
auto get_data(const Params & ... params)
{
    return std::array{params...};
}
```

1.17.3 Same Type Templates Issue

```
// This forces same type use, for both the numberator and denominator.
// not good, weird compile errors here!

template<typename Numerator, typename Denominator>
    /*what's the return type*/
    divide(Numerator numerator, Denominator denominator) {
    return numerator / denominator;
}
```

1.17.4 C++14 auto return types

```
template<typename Numerator, typename Denominator>
auto divide(Numerator numerator, Denominator denominator)
{
    return numerator / denominator;
}
```

1.17.5 Templates and Don't Repeat Yourself (DRY)

Principle aimed at reducing repetition of code and reusability. We have more tools today, concepts, generic lambdas, etc. to help with templates.

```
// divide doubles
double divide(double numerator, double denominator) {
    return numerator / denominator;
}
```

```
//But you don't want all of your divisions to be promoted to double.

// Divide floats
float divide(float numerator, float denominator) {
    return numerator / denominator;
}

// You want to handle some kind of int values.

//divide ints
int divide(int numerator, int denominator) {
    return numerator / denominator;
}

// Templates were designed for just this scenario.
```

1.17.6 Basic template usage

```
template<typename T>
T divide(T numerator, T denominator) {
    return numerator / denominator;
}
```

1.17.7 Example use T. don't do that. Name your Type meaningfully

```
template Parameters with actual names.

template<typename Arithmetic>
<Arithmetic divide(Arithmetic numerator, Arithmetic denominator) {
    return numerator / denominator;
}
```

1.18 Structs

```
The only difference between a struct and a class is
With structs, all properties are public by default
```

1.18.1 Use structs instead of classes

```
The only difference between them is that struct has all members by default public.
Using struct makes examples shorter and easier to read.
```

1.19 Lambdas

Lambdas allow us to create unnamed function objects which may or may not have captures. Standard c++11 We are not allowed to know the name of the type of a lambda. We are not allowed to call its name.

```
auto lambda = [/*captures*/](int param1){ return param1 * 10; };

std::vector<int> ints{1, 2, 3, 4, 5};
auto even = [](int i){ return 0 == i % 2; };
auto square = [](int i){ return i * i; }; // this is a lambda!
                                          // it defines an anonymous function
                                          // on the fly.

for (int i : ints | std::view::filter(even) | std::view::transform(square)) {
    std::cout << i << ' ';
}

#include <string>

template<typename Map>
void print_map(const Map &map, const std::string key_desc = "key",
                               const std::string value_desc = "value")
{
    for_each(begin(map), end(map),
        [&](const typename Map::value_type &data) { /// lambda! This is too wordy
            std::cout << key_desc << ": '" << data.first << "' "
                      << value_desc << ": '" << data.second << "'\n";
            }
    );

/// If only there was some way to automatically deduce the types of lambda parameter!!!
}
```

1.19.1 Capturing

Mechanism through which a lambda function access variables from its surrounding scope. Use outside variable within their body.

```
auto lambda = [/*captures*/](int param1){ return param1 * 10; };

The captures s(ection) can be left empty if the lambda does not need to access any external variables

By value
auto lambda = [param](int param1){ return param1 * 10; };

A copy of the variable is made at the lambda's creation time.
Then, it uses that copy within its body.

Reference Capture
auto lambda = [&param](int param1){ return param1 * 10; };
```

```
The lambda function refers to the original variable in the surrounding scope directly,
without making a copy.

Combine value and reference captures in the same lambda expression,
capture multiple variables,
or capture all variables in the surrounding scope by using ‘[=]‘ for value capture or ‘[&]‘ for refer

int x = 42;
double y = 3.14;

auto lambda = [x, &y](int param1){ return x * param1 + y; };
```

In this example, ‘x‘ is captured by value, creating a copy, while ‘y‘ is captured by reference, allowing direct access to the original variable. The lambda can then use these captured variables (‘x‘ and ‘y‘) within its body to perform calculations.

By capturing variables, lambda functions gain access to the state of the surrounding scope, providing a powerful and flexible way to operate on external data within the lambda's body.

1.19.2 Lambda with auto

```
#include <iostream>
#include <string>

template<typename Map>
void print_map(const Map &map, const std::string key_desc = "key",
                               const std::string value_desc = "value")
{
    for_each(begin(map), end(map),
        [&](auto &data) { /// Sick !!
                          /// This makes a generic lambda.
            std::cout << key_desc << ": '" << data.first << "' "
                      << value_desc << ": '" << data.second << "'\n";
            }
    );
}
```

1.19.3 Generic and Variadic Lambdas

Create implicit templates by simply using the auto keyword.

```
auto lambda = [/*captures*/](auto ... params){
    return std::vector<int>{params...};
}

standard c++14
```

1.19.4 A lambda to initialize a const object

```
const auto data = [](){ // no parameters
```

```
    std::vector,int. result;
    // fill result with things.
    return result;
}(); // immediately invoked

Because of RVO, using a lambda will not add overhead and may increase performance.
What is RVO?

Yet, you don't want to make class members constl.
It can break things silently.
```

1.20 Collections

```
double *get_data() {
    double *data = new double[3];

    data[0] = 1.1;
    data[1] = 2.2;
    data[2] = 3.3;

    return data;
}

struct Double_Data {
    Double_Data(const std::size_t size) : data(new double[size] {} // allocate
    ~Double_Data() { delete [] data; }
    double *data;
   };

Double_Data get_data() {
    Double_Data data(3);
    data.data[0] = 1.1; data.data[1] = 2.2; data.data[2] = 3.3;
    return data;
   };

double sum_data(const Double_Data &d) {
    return d.data[0] + d.data[1] + d.data[2];
}

int main() {
    return sum_data(get_data()); // no leak, but we'll come back to it
}

#include <vector>
#include <map>

struct Person {
    std::string name;
    int age;
    std::vector<std::string> hobbies;
```

```
};

int main() {
    std::vector<Person> people = {
        {"Alice", 25, {"Reading", "Painting"}},
        {"Bob", 30, {"Gaming", "Hiking"}},
        {"Charlie", 20, {"Cooking", "Photography"}}
    };

    std::map<std::string, Person> personMap;
    for (const auto& person : people) {
        personMap[person.name] = person;
    }

    for (const auto& pair : personMap) {
        std::cout << "Name: " << pair.first << std::endl;
        std::cout << "Age: " << pair.second.age << std::endl;
        std::cout << "Hobbies: ";
        for (const auto& hobby : pair.second.hobbies) {
            std::cout << hobby << " ";
        }
        std::cout << std::endl;
    }

    return 0;
}
```

1.21 Concepts (c++20)

I want to have two versions of a function, one takes a Floating Point, the other an Integral value.

Allows us to specify the requirements for a type, implicitly creating a template that constrains how a function can be used. Standard: C++20

```
old c++ implementation, using type\_traits.

#include <type_traits>

template<typename T,
         typename std::enable_if<std::is_floating_point<T>::value, int>::type = 0>
auto func(T f) -> decltype(f * 3) { return f * 3; }

template<typename T,
         typename std::enable_if<std::is_integral<T>::value, int>::type = 0>
auto func(T i) -> decltype(i + 3) { return i + 3; }

c++17 implementation, using type\_traits still.
#include <type_traits>

template<typename T,
         std::enable_if_t<std::is_floating_point_v<T>, int> = 0>
```

```
auto func(T f) { return f * 3; }

template<typename T,
         std::enable_if_t<std::is_integral_v<T>, int> = 0>
auto func(T i) { return i + 3; }

C++20 implementation using concepts,

#include <concepts>

auto func(std::floating_point auto f) { return f * 3; }
auto func(std::integral auto i) { return i + 3; }
```

1.21.1 Auto Concept

```
# include <string>

// C++20's auto concept or further constrained to something that
// has values that can be destructured into 2 elements.

void print_map(const auto &map, const std::string &key_desc = "key",
                                const std::string &value_desc = "value")
{
    for (const auto &[key, value] : map) /// strucuted binding
    {
        std::cout << key_desc << ": '" << key << "' "
                  << value_desc << ": '" << value << "'\n";
    }
}

// implicitely created a template for us. Just like our lamdas did!
```

1.22 Constants

The most important tool to write clean code. An object declared const or accessed via a const referece or const pointer cannot be modified. It forces us to think about initialization and lifetime of objects, which affects performance.

Plus, it communicates meaning to readers. If a variable is not const, ask why not? Would using a lambda or adding a named function allow you to make the value const?

```
int triple(int const i) { // we know parameters won't change
  return i * 3;
}

const double pi = 3.141593; // tells the compiler the value can't change

int main()
{
    const double radius = 1.5;
```

```
    const double area = pi * radius * radius;
    std::cout << area;

    // east const or west const, the same
    const int i = 5;
    int const j = 6;
}
```

Const everything that's not constexpr

Is this a good const?

```
double *get_data() {
    double *data = new double[3];
    data[0] = 1.1; data[1] = 2.2; data[2] = 3.3;
    return data;
}

double *sum_data(double *data) {
    return data[0] + data[1] + data[2]; // uncaught leak
}

int main() {
    return sum_data(get_data());
}

// If only there was some way to automatically delete things when they are no longer needed...
```

1.23 Auto

```
constexpr double calculate_pi() {
    return 22/7;
}

constexpr auto pi = calculate_pi();

int main()
{
    const auto radius = 1.5;
    const auto area = pi * radius * radius;
    std::cout << area;

}
```

Automatic deduction of value types. Standard c++11

```
constexpr auto calculate_pi() { // Return type deduction
                                // for normal functions.
    return 22/7;
}
```

```
constexpr auto pi = calculate_pi();

int main()
{
    const auto radius = 1.5;
    const auto area = pi * radius * radius;
    std::cout << area;
}

// const auto
const auto result = std::count( /*stuff */);

or, if you prefer:
// auto const

auto const result = std::count( /*stuff*/ );
```

1.23.1 Familiarize Yourself with Auto Deduction

```
What is the type of val in all these situation?

const int *get();

int main() {
    const auto val = get();
}

const int &get();

int main() {
    const auto val = get();
}

The function get() returns a const int\& (a reference to a constant integer),
and the auto keyword deduces the type of val as the same as the type of the
expression on the right-hand side of the assignment.
Since get() returns a const int&, the type of val is also const int&.
The const qualifier in front of auto ensures that the type deduced for
val is also a const reference.

Const int *get();

int main() {
    const auto *val = get();
}

According to GPT

The function get() returns a const int* (a pointer to a constant integer),
```

```
and the auto keyword deduces the type of val as the same as the type of the
expression on the right-hand side of the assignment.
Since get() returns a const int*, the type of val is also const int*.

The const qualifier applies to the integer pointed to by val,
not to the pointer val itself. Therefore, val is a non-constant pointer to a constant integer.
If you wanted a constant pointer to a constant integer,
you would need to declare val as const int * const val = get();

const int &get();

int main() {
    const auto &val = get();
}

const int *get();

int main() {
    const auto &val = get();
}

const int &get();

int main() {
    const auto &&val = get();
}
```

1.23.2 possible expensive conversion

Avoid potential expensive conversions. Same as ranged-for loops.

Auto requires initialization, same as const.

```
const std::string value = get_string_value();

// What is the return type of get_string_value()?

If its std::string_view or const char *,
we'll get costly conversion on all compilers with no diagnostic.

// avoids conversion
const auto value = get_string_value();

Plus, auto return types simplifies code

template<typename Arithmetic>
Arithmetic divide(Arithmetic numerator, Arithmetic denominator) {
    return numerator / denominator;
}
```

1.24 Constexp

Compile-time generation of code and data.

```
#include <iostream>

const double pi = 3.141593;

int main()
{
   const double radius = 1.5;
   const double area = pi * radius * radius;
   std::cout << area;
}
    // is pi known at compile time??

// If only there was some way to make a compile-time constant....

#include <iostream>

constexpr double pi = 3.141593;

int main()
{
   const double radius = 1.5;
   const double area = pi * radius * radius;
   std::cout << area;
}

// OR EVEN, Generate it at compile-time.

#include <iostream>

constexpr double calculate_pi() {
    return 22/7;
}

constexpr double pi = calculate_pi();

int main()
{
   const double radius = 1.5;
   const double area = pi * radius * radius;
   std::cout << area;
}

// This function can be executed at compile-time
constexpr double calculate_pi() {
    return 22/7;
}

// This value will be available at compile-time
```

```
constexpr double pi = calculate_pi();

Standard c++11, relaxed in c++14, relaxed a ton c++20

// But what is the type of pi?? (double, float, long double?)
I don't care?
It depends?

// If only there was a way to mention "I don't care or It depends" for what a type is ......
‘constexpr‘ is used for compile-time evaluation.
It ensures that an expression or function can be computed at compile time.
It reduces runtime overhead.
The program avoids the need to perform those computations at runtime.

The computations are already done during the compilation process,
The program doesn't incur the additional time and resources required
to perform them during runtime execution.
```

1.24.1 Overcomplicated Constexpr

```
// static const data known at compile time.
static const std::vector<int> angles{-90,-45,0,45,90};
```

1.24.2 Useful Constexpr

```
// Moving static const to static constexpr.
static constexpr std::array<int, 5> angles{-90,45,0,45,90};

Here static constexpr makes sure the object
is not reinitialized each time the function is encountered.

With static (see object lifetimes), the variable lasts for the lifetime of the program.
It will be initialized only once.

    the size of the array is now known at compile time
    We've removed dynamic allocations
    We no longer pay the cost of accessing a static

    " when you see a const, always ask yourself: "is this value known at compile time?
    If it is, what would it take to make the value constexpr?"
```

1.24.3 What is a const char?

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;
```

```
int main() {
    fs::path path = "your_directory_path";

    for (const auto &entry : fs::directory_iterator(path)) {
        const char *cstr = entry.path().c_str();
        std::cout << cstr << '\n';
    }
    return 0;
}
```

1.25 Memory Allocation

```
// In declaration & is a reference

Memory Address (&)

int porcupine = 3;
std::cout << &porcupine << "\n"; // Memory address
                                 // 0x7ffd7caa5b54

Dynamic Memory Allocation

int main() {
    int size = 5;
    int* dynamicArray = new int[size];

    for (int i = 0; i < size; ++i) {
        dynamicArray[i] = i + 1;
    }

    std::cout << "Dynamic Array: ";
    for (int i = 0; i < size; ++i) {
        std::cout << dynamicArray[i] << " ";
    }
    std::cout << std::endl;

    delete[] dynamicArray;

    return 0;
}

We allocate dynamic memory for an integer array of size 5 using the ‘new‘ keyword.
We then populate the array with values from 1 to 5.
Finally, we print and release the allocated memory using ‘delete[]‘ to avoid memory leaks.
```

1.26 Factories

Design pattern.

```
struct S {
    S() = default;
    S(S&&) = delete;
    S(const S &) = delete;
};

auto s_factory(){
    return S{}; // compiles in C++17, neither a copy nor a move.
}

#include <memory>

class Product {
public:
    virtual void use() const = 0;
};

class ConcreteProductA : public Product {
public:
    void use() const override {
        std::cout << "Using ConcreteProductA" << std::endl;
    }
};

class ConcreteProductB : public Product {
public:
    void use() const override {
        std::cout << "Using ConcreteProductB" << std::endl;
    }
};

class Factory {
public:
    std::unique_ptr<Product> createProduct(const std::string& productType) {
        if (productType == "A") {
            return std::make_unique<ConcreteProductA>();
        } else if (productType == "B") {
            return std::make_unique<ConcreteProductB>();
        } else {
            return nullptr;
        }
    }
};

int main() {
    Factory factory;

    std::unique_ptr<Product> productA = factory.createProduct("A");
    if (productA) {
        productA->use(); // Output: Using ConcreteProductA
    }
```

```
    std::unique_ptr<Product> productB = factory.createProduct("B");
    if (productB) {
        productB->use(); // Output: Using ConcreteProductB
    }

    return 0;
}
```

Here, a factory class (‘Factory‘) that creates different types of products (‘ConcreteProductA‘ and ‘ConcreteProductB‘) derived from an abstract base class ‘Product‘. The factory’s ‘createProduct‘ method takes a product type as input and returns a unique pointer to the created product. The main function creates instances of ‘ConcreteProductA‘ and ‘ConcreteProductB‘ through the factory.

1.26.1 Virtual Factories

A virtual factory is a design pattern that uses virtual functions and polymorphism to create objects of different types through a common interface. Polymorphism is the ability of an object to take on many forms and behave differently based on the context or the type of object it is being accessed through.

```
#include <memory>

class Product {
public:
    virtual void use() const = 0;
};

class ConcreteProductA : public Product {
public:
    void use() const override {
        std::cout << "Using ConcreteProductA" << std::endl;
    }
};

class ConcreteProductB : public Product {
public:
    void use() const override {
        std::cout << "Using ConcreteProductB" << std::endl;
    }
};

class AbstractFactory {
public:
    virtual std::unique_ptr<Product> createProduct() const = 0;
};

class ConcreteFactoryA : public AbstractFactory {
public:
    std::unique_ptr<Product> createProduct() const override {
        return std::make_unique<ConcreteProductA>();
```

```
    }
};

class ConcreteFactoryB : public AbstractFactory {
public:
    std::unique_ptr<Product> createProduct() const override {
        return std::make_unique<ConcreteProductB>();
    }
};

int main() {
    std::unique_ptr<AbstractFactory> factoryA = std::make_unique<ConcreteFactoryA>();
    std::unique_ptr<Product> productA = factoryA->createProduct();
    if (productA) {
        productA->use(); // Output: Using ConcreteProductA
    }

    std::unique_ptr<AbstractFactory> factoryB = std::make_unique<ConcreteFactoryB>();
    std::unique_ptr<Product> productB = factoryB->createProduct();
    if (productB) {
        productB->use(); // Output: Using ConcreteProductB
    }

    return 0;
}
```

Here, an abstract factory class (‘AbstractFactory‘) that defines the interface for creating products. The concrete factory classes (‘ConcreteFactoryA‘ and ‘ConcreteFactoryB‘) implement implement the createProduct method to create specific products (ConcreteProductA and ConcreteProductB, respectively). The main function demonstrates the usage of the virtual factory by creating instances of the concrete factories and using them to create products.

1.27 System()

```
#include <string>

int main() {
    std::string cmd = "ls -l";
    system(cmd.c_str()); // change to c-string
    return 0;
}
```

1.27.1 Error Handeling - if cmd succeeded

```
int main() {
    std::string cmd = "ls -l";
    int result = system(cmd.c_str());  // Pass C-string to system()

    if (result == 0) {
        std::cout << "Command executed successfully." << std::endl;
```

```
    } else {
        std::cout << "Command execution failed." << std::endl;
    }
    return 0;
}
```

1.28 OpCode

```
mov
```

1.29 Structured Bindings

Used to decompose a strcure or array into a set of identifiers. You must use auto, and the number of elements must match. There's no way to skip an element.

```
const auto &[elem1, elem2] = some_thing;

Standard c++17

# include <iostream>
# include <string>

template<typename Map>
void print_map(const Map &map, const std::string &key_desc = "key",
                               const std::string &value_desc = "value")
{
    for (const auto &[key, value] : map) /// strucuted binding
    {
        std::cout << key_desc << ": '" << key << "' "
                  << value_desc << ": '" << value << "'\n";
    }
}

Standard c++17
```

1.29.1 Bulky Template Synthax

```
#include <string>

template<typename Map>
void print_map(const Map &map, const std::string &key_desc = "key",
                               const std::string &value_desc = "value")
{
    for (const auto &[key, value] : map) /// strucuted binding
    {
        std::cout << key_desc << ": '" << key << "' "
                  << value_desc << ": '" << value << "'\n";
    }
```

```
}

// If only there was a way to simplify this code......

Potential Inefficiency Hiding

# include <iostream>
# include <string>

void print_map(const auto &map, const std::string &key_desc = "key",
                                const std::string &value_desc = "value")
{
    for (const auto &[key, value] : map) /// strucuted binding
    {
        std::cout << key_desc << ": '" << key << "' "
                  << value_desc << ": '" << value << "'\n";
    }
}

int main()
{
    print_map(get_some_map(), "index", "location");
}

/// Std::String vs Const Char *

We are constructing a std::string from a const char * for no particular reason.

If Only there was a way to observe string-like things without actually constructing a std::string!!..
```

1.30 fmt - format library

```
#include <format>
#include <string_view>

void print_map(const auto &map, const std::string_view &key_desc = "key",
                                const std::string_view &value_desc = "value")
{
    for (const auto &[key, value] : map) /// strucuted binding
    {
        std::puts(std::format("{}: '{}' {}: '{}'",
                        key_desc, key, value_desc, value).c_str());

        // this is genious spacing for readability
    }
}

Standard c++20
```

1.30.1 String_View

A non-owning "view" of a string like structure.

```
#include <string_view>
std::string_view sv{some_string_like_thing}; // no copy

Standard c++17

These are passed-by-value on purpose.
String_view are cheap to copy. It is recommended to pass them by value.
The following code doesn't create a string anymore, if doesn't have to.

# include <iostream>
# include <string>

void print_map(const auto &map, const std::string_view &key_desc = "key",
                                const std::string_view &value_desc = "value")
{
    for (const auto &[key, value] : map) /// strucuted binding
    {
        std::cout << key_desc << ": '" << key << "' "
                  << value_desc << ": '" << value << "'\n";
    }
}

int main()
{
    print_map(get_some_map(), "index", "location");
}
```

1.30.2 Cout

Std::Cout is quite verbose, relatively slow and difficult to reason about.

```
If only there was some easier way of formatting our output...

Puts (algorithm)
```

1.30.3 Text Formatting

A subset of the excellent fmt library, allowing for formatting of strings with positional, named and python/printf style formatting options.

```
#include <format>
std::string s = fmt::format("I'd rathe be {1} than {0}.", "right", "happy");
// "I'd rather be happy than right."

Standard C++20
```

1.30.4 All Combined (algorithms, text formatting, concepts, etc.)

```
#include <format>
#include <string_view>

void print_map(const auto &map, const std::string_view &key_desc = "key",
                                const std::string_view &value_desc = "value")
{
    const auto print_key_value = [&](const auto &data) {
        const auto &[key, value] = data;
        std::puts(std::format("{}: '{}' {}: '{}'",
                        key_desc, key, value_desc, value).c_str());
    };

    for_each(begin(map), end(map), print_key_value);
}

// if only there was some way to not have to do this begin(map) and end(map)!!!

Auto Concept in c++20

#include <array>

auto get_data(const auto & ... params)
{
    return std::array{params...};
}

// The only problem now is that this code requires the types to be copyable.
// If only there was a way to forward arguments and avoid copies......
```

1.31 System Design

1.31.1 Header Files

1.31.2 Source Files

1.32 Arguments

```
#include <iostream>

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

    return 0;
}
```

1.32.1 Flags

```
bool hasFlag(const std::vector<std::string>& arguments, const std::string& flag) {
    return std::find(arguments.begin(), arguments.end(), flag) != arguments.end();
}

int main(int argc, char* argv[]) {
    std::vector<std::string> arguments(argv + 1, argv + argc);

    if (arguments.size() == 0) {
        arguments.push_back("-help");
    }

    if (hasFlag(arguments, "-o")) {
        std::cout << "Flag -o for -omit is present!" << std::endl;
    }
```

1.33 Synthax

1.33.1 Ternary Operator

```
std::cout << (coin == 0 ? "Heads" : "Tails") << "\n";
```

1.33.2 Relational Operators

```
    ==              equal to
    !=              not equal to
    >               greater than
    <               less than
    >=              greater than or equal to
    <=              less than or equal to
```

1.33.3 Logical Operators

```
    &&              and
    ||              or
    !               not

    if (hunger && anger){}

    ( !true )       not false
    ( !(10 < 11) )  not condition
```

1.33.4 String Manipulation

```
reversed_text += text
```

```
std::string chapOnePath = parentDirectory + "/chap_1";
```

1.33.5 Scope

```
run() {
    {
        Is this in scope? //This variable is out of scope.?
    {

}
```

1.33.6 Chaining

```
int age = 28;
std::cout << age << "years old.\n";

template <typename T>
struct MathOperation {
    T value;

    MathOperation(T val) : value(val) {}

    template <typename U>
    MathOperation<U> add(U val) {
        return MathOperation<U>(value + val);
    }

    template <typename U>
    MathOperation<U> multiply(U val) {
        return MathOperation<U>(value * val);
    }

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

int main() {
    MathOperation<int> operation(5);
    operation.add(3).multiply(2).print();

    return 0;
}
```

1.33.7 Sstream

‘std::stringstream‘ is a C++ class. It handling string-based input and output operations, allowing you to read from and write to strings as if they were input/output streams.

```
#include <map>
#include <sstream>
#include <string>

int main() {
    std::string text = "lorem ipsum dolor sit amet, consectetur adipiscing elit.";

    // convert text to lowercase
    for (char& c : text) {
        c = std::tolower(c);
    }

    std::map<std::string, int> wordfreq;
    std::istringstream iss(text);
    std::string word;

    // count the frequency of each word in the text
    while (iss >> word) {
        wordfreq[word]++;
    }

    // display the word frequencies
    for (const auto& pair : wordfreq) {
        std::cout << pair.first << ": " << pair.second << std::endl;
    }

    return 0;
}
```

1.34 Type

1.34.1 Return Types

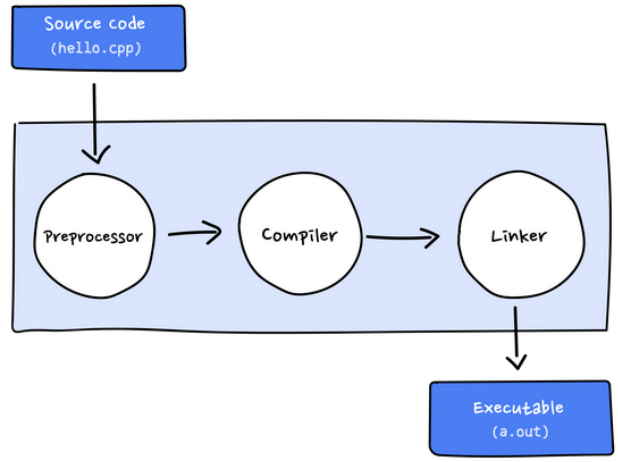
```
  std::exit(0) (success)
  std::exit(1) (failure)

  int isPrime(int x) {
    if (x % 2 == 1) {
        return 0; // is considered as false
                  // Yet, is it not the same as the boolean false
                  // In a boolean context,
                  // 0 is implicitly converted to false,
                  // However, not equivalent in all contexts.
    }
  }

Void, Also refered as subroutine.

void oscar_wilde_quote() {
  std::cout << "The highest";
```

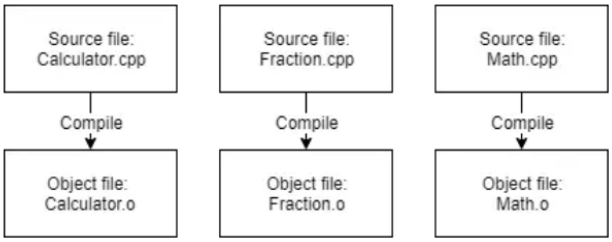
```
}

#include <vector>

template <typename T>
struct ComplexType {
    T value;
    std::vector<T> elements;

    ComplexType(T val) : value(val) {} // constructor

    void print() {
        std::cout << "Value: " << value << std::endl;
        std::cout << "Elements:";
        for (const auto& elem : elements) {
            std::cout << " " << elem;
        }
        std::cout << std::endl;
    }
};

template <typename T>
ComplexType<T> createComplexType(T value, const std::vector<T>& elements) {
    ComplexType<T> complex(value);
    complex.elements = elements;
    return complex;
}

int main() {
    std::vector<int> elements{1, 2, 3, 4, 5};
    ComplexType<int> result = createComplexType(10, elements);
    result.print();

    return 0;
}
```

A template struct ‘ComplexType‘ represents a complex type with a value and a vector of elements. The ‘createComplexType‘ function creates an instance of ‘ComplexType‘ by taking a value and a vector of elements as parameters. The function initializes a ‘ComplexType‘ object, sets its elements, and returns it. Finally, in the ‘main‘ function, we call ‘createComplexType‘ and print the resulting complex type.

1.35 Compiler

If your program has 3 .cpp files, the compiler would generate 3 object files. After all object files were compiled, the linker starts.

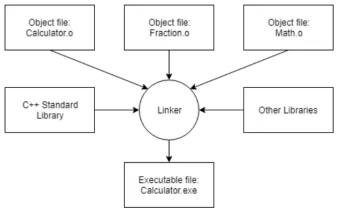





```
MVSC // Microsoft Compiler
GCC // Gnu C Compiler
LLVM Clang // LLVM Compiler
```

1.36 Linker

```
It combined all object files in one executable.
the linker links library files.
resolves cross-file dependencies.
```



1.37 Libraries

1.37.1 Library files

```
a collection of precompiled code that has been "packaged up" for reuse in other programs.
C++ comes with a standard library providing additional functionality.
```

```
commonly used in the iostream library
```

1.37.2 External Libraries

A collection of pre-compiled code, providing functionality.

Functions, classes, and data structures that are not part of the standard C++ library.

These libraries are often provided by third-party developers or organizations

They speed up the development process by providing ready-to-use functionality.

```
Popular C++ external libraries:
```

1.37.3 Boost

1.37.4 OpenCV

1.37.5 Qt

1.38 Builds

1.38.1 Build Configuration (build target)

```
Project settings determining how your project will be built.
Build configuration includes executable name, project arch and library files.
It specifies keepings or strippings of debugging info, compiler optimization details
```

1.38.2 Debug Config

```
Debug configuations turns off all optimizations, includes debugging information,
Jason Turner would say "with as much information as possible"
Such configs makes your programs larger and slower, but much easier to debug.
```

1.38.3 Release Config

```
Optimized for size and performance, no debugging information.
With all optimization, now testing for code performance.

When the Hello World program (from lesson 0.7) was built using Visual Studio,
Executable produced in the debug configuration was 65kb,
Executable built in the release version was 12kb.
The difference is largely due to the extra debugging information kept in the debug build.
```

1.38.4 G++ Builds

```
GCC / Clang?
    -ggdb  // cmd line debugging
           // This is the GNU Debugger !?
    -ggdb O2 -DNDEBUG for release builds. ??
    -g++ O2 -DNDEBUG for release builds. ??
```

1.39 Compiler Extensions (compiler-specific behavior)

many compilers implement their own changes to the language,

```
Using compiler extension allows you to write programs that are incompatible
with the C++ standard.
```

Programs using non-standard extensions generally will not compile on other compilers (lacking same extensions support) , or if they do, they may not run correctly.

Compiler extensions are often enabled by default. Overpermissive compilers.

Compiler extensions are optional, cause programs non-compliancy with C++ standards, Turn compiler extensions off.

```
GCC
         -pedantic-errors // Disable extensions
```

1.40 Max Warnings

```
GCC
    -Wall -Weffc++ -Wextra -Wsign-conversion
    -Werror

Turner does it with cmake?
```

1.41 Standard Set-Up

```
GCC pre-8
          -std=c++11 // set c++ standard
          -std=c++14
          -std=c++17
          -std=c++20
GCC 8 or 9
          -std=c++2a for C++20 support

g++ -std=c++17 myfile.cpp -o output // cmd line
```

1.42 Dear ImGui

1.43 PCH - Precompiling Headers

Some compilers' feature to speed compilation of large projects. It generates a precompiled version of commonly included header files,

This version is known as the PCH file. It is used during subsequent compilations. Avoids reparse and re-process of header files.

Great for large projects: numerous headers, included in multiple source files.

1.43.1 Enable PCH

PCH usage is compiler-specific, the enabling and configuring method varies.

It typically involves specifying which headers should be precompiled and how the precompiled information should be stored.

1.43.2 Include Header Files

```
// Main.cpp
// #include "my_function.hpp"

//then
// g++ main.cpp my_function.cpp -o program
```

1.43.3 Header Files 2.0

```
// Declare in header, fun.hpp or fun.h
double average(double num1, double num2);

// Define in fun.cpp
double average(double num1, double num2) {
  return (num1 + num2) / 2;
}
```

1.43.4 Standard Library Headers

```
Some standard library headers are included by others.

<cstdlib> is included by <iostream>, since it relies on its functionalities,
such as the declaration of the system() function.
```

1.44 Command Line Linking and Compiling

```
g++ main.cpp my_functions.cpp // link both files
```

1.44.1 Source Code Files Suffix

```
.cpp (ex: hello.cpp) or
.h (ex: std_lib_facilities.h).
```

1.44.2 Compile

```
g++ hello.cpp -o hello
```

```
Compiling translates C++ programs into machine code.
It is stored on disk as a file with the .o extension (hello.o).
```

A linker then links the object code with standard library routines that the program may use and creates an executable image which is also saved on disk,

usually as a file with the file name without any extension (e.g. hello).

1.45 Debugg and Error Type

1.45.1 Compile Time Errors

1.45.2 Synthax Errors

1.45.3 Type errors

Forgetting to declare a variable

Storing a value in a different type.

1.45.4 Link-Time Errors

link-time errors are based on unfindable needed function or library.

When the linker tries to combine object files into an executable.

1.45.5 Run-Time Errors

Errors which happen during program execution (run-time) after successful compilation.

Division by zero

Open an non-existing file

1.45.6 Logical Errors

Flawed programming's logical thinking.

No errors, but output is wrong.

1.45.7 Testing Framework

1.46 Benchmarking strategy

1.46.1 Chrono, clock time

```
#include <chrono>
int main() {

  // Measure time taken for goodnight1():
  std::chrono::high_resolution_clock::time_point start = std::chrono::high_resolution_clock::now();

  std::cout << goodnight1("tulip");

  std::chrono::high_resolution_clock::time_point end = std::chrono::high_resolution_clock::now();
  std::chrono::duration<double, std::milli> time_span = end - start;

  // Print time taken for goodnight1():
  std::cout << "Time taken for goodnight1(): " << time_span.count() << " milliseconds.\n\n";
```

1.47 Good practices

1.47.1 Proper Design

```
    if a component is hard to test, it is not properly designed.
    if a component is easy to test, it indicates proper design.
    Approval tests ressource : https://cppcast.com/clare-macrae/
```

1.47.2 Warnings

Enable as many compiler warnings as you can. Fix new warning generated.

It will feel tedious and meaningfless But this is the c++ way to catch real bugs.

1.47.3 Slow Down!

```
Copy and pasting is easy.
Forging ahead in comfort is too easy.
Plan ahead, don't get caught off guard.
```

1.47.4 Ponder for solutions

```
If the solution seems large or complex, stop.
Walk and ponder for the solution.
discuss the design with a rubber duck.
spend less time programming, more thinking.
```

1.47.5 C++ is not magic nor Object-Oriented

```
It's not magic, construct to test your doubts.
It is multi-disciplinary, supports all programming paradigms.

    Procedural
    Functional
    Object-Oriented
    Generic
    Compile-Time(contexpr and template metaprogramming)

Knowing when paradims are needed is the key to good C++.
Using appropriate techniques takes time and appropriate technique.
```

1.47.6 Learn a different language

```
    Lisp // Diverge from the C-family languages. Learn,
    Haskell
    Erlang
```

Chapter 2

Data Structures

2.1 Maps

Dynamically resizing containers.

Unlike arrays, fixed sizes, std::map has an arbitrary number of key-value pairs. ‘std::map‘ is a balanced binary search tree, it keeps elements sorted based on the keys, Insertions, deletions and searches have logarithmic time complexity. Plus, keys must be unique within the map.

C++’s equivalent to Ruby’s hash object. Both ‘std::map‘ and Ruby’s hash are associative containers that store key-value pairs. They allow efficient lookup and retrieval of values based on a given key. The key-value pairs are stored in an unordered, unspecific manner.

```
#include <map>

int main() {
    std::string str = "hello";
    std::map<char, int> charFreq;

    // Count the frequency of each character in the string
    for (char c : str) {
        charFreq[c]++;
    }

    // Display the character frequencies
    for (const auto& pair : charFreq) {
        std::cout << pair.first << ": " << pair.second << std::endl;
    }

    return 0;
}

#include <map>
#include <string>

int main() {
    std::map<std::string, int> studentGrades;
```

```
    // Add student names and their grades
    studentGrades["Alice"] = 90;
    studentGrades["Bob"] = 85;
    studentGrades["Charlie"] = 95;

    // Access and display individual student grades
    std::cout << "Charlie's grade: " << studentGrades["Charlie"] << std::endl;

    // Iterate over all student grades
    for (const auto& pair : studentGrades) {
        std::cout << pair.first << ": " << pair.second << std::endl;
    }

    return 0;
}

std::map<int, Product> map1 {
    {1, Product(1, "Item 1")},
    {2, Product(10, "Item 2")},
};

map1[3] = Product(30, "Item 3");
for (std::pair<int, Product> elt : map1)
{
    cout << elt.first << " ";
    cout << elt.second.name << '\n';

}


#include <map>
#include <sstream>
#include <string>

int main() {
    std::string text = "lorem ipsum dolor sit amet, consectetur adipiscing elit.";

    // convert text to lowercase
    for (char& c : text) {
        c = std::tolower(c);
    }

    std::map<std::string, int> wordfreq;
    std::istringstream iss(text);
    std::string word;

    // count the frequency of each word in the text
    while (iss >> word) {
        wordfreq[word]++;
    }
```

```
    // display the word frequencies
    for (const auto& pair : wordfreq) {
        std::cout << pair.first << ": " << pair.second << std::endl;
    }

    return 0;
}

// Understand how this code is working
// Accidental copy?

std::map<std::string, int> get_map();

using element_type = std::pair<std::string, int>;

for (const element_type & : get_map())
{}
```

2.1.1 Maps CRUD

```
    Create (Insert Key-Value Pair):
        map[key] = value
        map.insert(std::make_pair(key, value))

    Read (Retrieve a value by key):
        Value = map[key]
        Checking if a key exists: map.count(key)

    Update (Modify the value of a key):
        map[key] = new_value

    Delete (Remove a key-value pair):
        map.erase(key)
        Removing all elements: map.clear()

#include <map>

int main() {
    std::map<int, std::string> map;

    // Create (Insert)
    map[1] = "Apple";
    map.insert(std::make_pair(2, "Banana"));

    // Read (Retrieve)
    std::cout << "Value at key 1: " << map[1];
    std::cout << "Key 2 exists? " << (map.count(2) ? "Yes" : "No");

    // Update
    map[1] = "Apricot";
```

```
    // Delete
    map.erase(2);

    // Display all key-value pairs
    for (const auto& pair : map) {
        std::cout << "Key: " << pair.first << ", Value: " << pair.second << std::endl;
    }

    return 0;
}
```

2.1.2 Multimaps

A multimap allows multiple values to be associated with the same key.

```
#include <map>

int main() {
    std::multimap<int, std::string> myMultimap;

    // Insert key-value pairs into the multimap
    myMultimap.insert(std::make_pair(1, "Apple"));
    myMultimap.insert(std::make_pair(2, "Banana"));
    myMultimap.insert(std::make_pair(1, "Apricot"));

    for (const auto& pair : myMultimap) {
        std::cout << "Key: " << pair.first << ", Value: " << pair.second << std::endl;
    }

        // Key: 1, Value: Apple
        // Key: 1, Value: Apricot
        // Key: 2, Value: Banana

    return 0;
}
```

2.2 Sets

Std::set represents a sorted set of unique elements. The set maintains a sorted order, the elements are automatically sorted. Plus, it makes sure they are unique.

```
#include <set>

int main() {
    std::set<int> mySet;
    mySet.insert(5);
    mySet.insert(2);
    mySet.insert(8);
```

```
    for (const auto& element : mySet) {
        std::cout << element << " ";
    }
    std::cout << std::endl;

    // Check if an element exists in the set
    int searchElement = 2;
    if (mySet.count(searchElement) > 0) {
        std::cout << searchElement << " exists in the set" << std::endl;
    } else {
        std::cout << searchElement << " does not exist in the set" << std::endl;
    }

    // Remove an element from the set
    int removeElement = 5;
    mySet.erase(removeElement);

    for (const auto& element : mySet) {
        std::cout << element << " ";
    }
    std::cout << std::endl;

    // 2 5 8
    // 2 exists in the set
    // 2 8

    return 0;
}
```

2.2.1 Multisets

Std::multiset represents a sorted set of elements (duplicates are possible). The set maintains a sorted order, the elements are automatically sorted.

```
#include <iostream>
#include <set>

int main() {
    std::multiset<int> myMul;
    myMul.insert(5);
    myMul.insert(2);
    myMul.insert(2); // Duplicates are allowed in a multiset

    for (const auto& element : myMul) {
        std::cout << element << " ";
    }
    std::cout << std::endl;

    // Check the count of a specific element
    int countE = 2;
    std::cout << countE << " appears " << myMul.count(countE) << " times" << '\n';
```

```
    // Remove specific elements
    int removeElement = 2;
    myMul.erase(removeElement);

    for (const auto& element : myMul) {
        std::cout << element << " ";
    }
    std::cout << std::endl;

        // 2 2 5
        // 2 appears 2 times
        // 2 5

    return 0;
}
```

2.3 Maps and Sets Functions

2.3.1 Size

```
empty() - returns true if empty
max_size() - returns max number of elements to be stocked
size() - returns the number of elements
```

2.3.2 Access and Research

```
at()            - Only for Maps.
                - returns an reference to the element. Can list std::out_of_range exception, I think?
operator[]      - Only for Maps.
                - returns a reference to the element
count()         - returns number of elements matching the given key
find()          - finds an element with given key
lower_bound()   - return an iterator to the first element < (less than) the given key
upper_bound()   - return an iterator to the first element > (less than) the given key
equal_range()   - find a range of elements that are equivalent to a given value in a sorted container
```

```
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> numbers = {2, 4, 6, 8, 10};

    auto it = std::find(numbers.begin(), numbers.end(), 6);

    if (it != numbers.end()) {
        std::cout << "Value 6 found at index: " << std::distance(numbers.begin(), it) << std::endl;
    } else {
        std::cout << "Value 6 not found" << std::endl;
```

```
    }

    return 0;
}

#include <set>
#include <algorithm>

int main() {
    std::multiset<int> numbers = {1, 2, 3, 3, 3};

    // Find the range of elements equal to 3 in the multiset
    auto range = numbers.equal_range(3);

    // Print the elements in the range
    std::cout << "Elements equal to 3: ";
    for (auto it = range.first; it != range.second; ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    // Elements equal to 3: 3 3 3

    return 0;
}
```

2.3.3 Modify Maps and Sets

```
clear()         - empties the container
emplace()       -
emplace_hint()  -
erase()         - delete one element, a range or all elements at given key
extract()       - extract a knot (a node? my translation)
insert()        - insert elements or knots (a node?) from other containers
merge()         - merge a container in a container
swap()          - swap the content of two containers

insert_or_assign() - insert a new element or deletes the value
try_emplace()      - builds a new key-value pair at given idx
```

2.4 Lists

List (std::list) is a doubly-linked list container, storing in a linear sequence. It allows efficient insertion and removal of elements at any position, but direct access to elements by index is slower, compared to random-access containers like vector.

std::list container does not support random access using the subscript operator [], like an array or std::vector does. Use an iterator

```
std::list mylist{100, 20 , 300, 40, 500, 20, 100};
```

```
mylist.sort();
mylist.unique();

for (size_t i = 0; i < love.size(); i++)
    std::cout << love[i]
                        // impossible,
                        // no support for subscript
                        // operator []

#include <list>

int main() {
    std::list<int> love{20, 30, 40};

    for (const auto& element : love) {
        std::cout << element << '\n';
    }

    return 0;
}
```

2.4.1 Forward Lists

std::forward_list store elements as a singly linked list.

It does not provide the necessary iterator support for the range-based for loop syntax. Unlike other standard containers like std::vector or std::list, std::forward_list does not have a begin() and end() member function that returns iterators.

```
#include <forward_list>

int main() {
    std::forward_list<int> mylist{20, 30, 40};

    for (auto it = mylist.begin(); it != mylist.end(); ++it) {
        std::cout << *it << '\n';
    }

    return 0;
}
```

2.4.2 Lists Functions

```
merge() - merges two sorted lists. The merged list is emptied.
remove() - delete list elements for a value.
remove_if() - delete list element for a boolean (to verify).
reverse() - reverses list content.
sort() - sort elements.
splice() - moves elements, before given index. (not valid for std::forward_list).
splice_after() - moves elements, after given index. (not valid for std::list).
unique() - replaces consecutive duplicates by one value.
```

2.5 Hash Maps

Hash maps are the answer to every interview. They are containers with fast elements inserts.

2.5.1 Unordored Map

A map storing key-value pairs with constant-time average complexity for insertion, deletion, and access operations based on the key. The elements in the map are not ordered by their keys.

```
#include <unordered_map>

int main() {
    std::unordered_map<std::string, int> ages = {
        {"Alice", 25},
        {"Bob", 32},
        {"Charlie", 42}
    };

    // Insert a new key-value pair
    ages["Dave"] = 55;

    // Access elements using the [] operator
    std::cout << "Age of Alice: " << ages["Alice"] << std::endl;

    // Check if a key exists
    if (ages.count("Charlie") > 0) {
        std::cout << "Charlie's age: " << ages["Charlie"] << std::endl;
    }

    // Iterate over the map
    for (const auto& entry : ages) {
        std::cout << "Name: " << entry.first << ", Age: " << entry.second << std::endl;
    }

    // Erase an element
    ages.erase("Bob");

    // Check if a key exists after erasing
    if (ages.count("Bob") > 0) {
        std::cout << "Bob's age: " << ages["Bob"] << std::endl;
    } else {
        std::cout << "Bob is not found in the map." << std::endl;
    }

    return 0;
}
```

2.5.2 Unordored MultiMap

A map that allows for multiple values associated with the same key.

```
#include <unordered_map>

int main() {
    std::unordered_multimap<int, std::string> students;

    // Insert multiple key-value pairs with the same key
    students.insert({101, "Alice"});
    students.insert({102, "Bob"});
    students.insert({101, "Charlie"});
    students.insert({103, "Dave"});
    students.insert({101, "Eve"});

    // Iterate over the multimap
    for (const auto& entry : students) {
        std::cout << "ID: " << entry.first << ", Name: " << entry.second << std::endl;
    }

    // Find all students with ID 101
    int targetId = 101;
    auto range = students.equal_range(targetId);
    std::cout << "Students with ID " << targetId << ":" << std::endl;
    for (auto it = range.first; it != range.second; ++it) {
        std::cout << it->second << std::endl;
    }

        // ID: 101, Name: Alice
        // ID: 103, Name: Dave
        // ID: 101, Name: Charlie
        // ID: 102, Name: Bob
        // ID: 101, Name: Eve
        // Students with ID 101:
        // Alice
        // Charlie
        // Eve

    return 0;
}
```

2.5.3 Unordored Lists

'std::unordered_map' is a hash table. Unspecific order, with constant-time complexity,for average-case insertions, deletions, and searches. Keys must be unique as well.

```
Need the elements to be sorted by keys or require efficient range-based operations?
'std::map' is a good choice.

Prioritize constant-time lookups and insertions? Don't need a specific order?
'std::unordered_map' can provide better performance.

constant time (O(1))
linear time (O(n))
```

2.5.4 Unordered Sets

A container storing unique elements with unordered access and efficient search operations.

```
#include <unordered_set>

int main() {
    std::unordered_set<int> numbers;
    numbers.insert(5);
    numbers.insert(2);
    numbers.insert(10);
    numbers.insert(7);
    numbers.insert(3);

    for (const auto& number : numbers) {
        std::cout << number << " ";
    }
    std::cout << std::endl;

    // Check if a value exists in the set
    int target = 7;
    if (numbers.find(target) != numbers.end()) {
        std::cout << target << " is found in the set." << std::endl;
    } else {
        std::cout << target << " is not found in the set." << std::endl;
    }

    // Remove an element from the set
    int toRemove = 2;
    numbers.erase(toRemove);

    for (const auto& number : numbers) {
        std::cout << number << " ";
    }
    std::cout << std::endl;

        // 7 5 10 3 2
        // 7 is found in the set.
        // 7 5 10 3

    return 0;
}
```

2.5.5 Unordered Multisets

A container storing elements with unordered access and efficient search operations.

```
#include <unordered_set>

int main() {
    std::unordered_multiset<int> numbers;
    numbers.insert(5);
```

```
    numbers.insert(2);
    numbers.insert(5); // Duplicate element

    for (const auto& number : numbers) {
        std::cout << number << " ";
    }
    std::cout << std::endl;

    // Count occurrences of a value in the multiset
    int target = 5;
    int count = numbers.count(target);
    std::cout << "Occurrences of " << target << ": " << count << std::endl;

    // Remove specific occurrences of a value from the multiset
    int toRemove = 5;
    numbers.erase(numbers.find(toRemove));

        // 2 5 5
        // Occurrences of 5: 2
        // 2 5

    return 0;
}
```

2.5.6 Hash Functions

```
hash_function()  - returns a function, unclear
key_eq()         - returns a function, unclear

begin(int)
end(int)
cbegin(int)
cend(int)
bucket(key)
bucket_count(key)
bucket_size(key)
max_bucket_count(key)


load_factor()
max_load_factor()
rehash()
reserve()

I need definition and example for these.
```

2.6 Queue

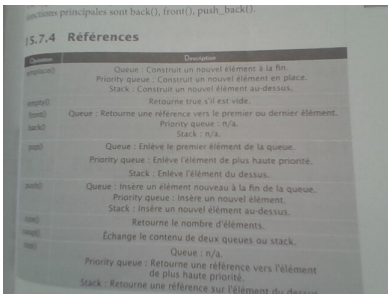
std::queue is a FIFO (first in, first out). Important functions are back(), front(), push_back(), pop_front().

2.7 Priority Queue

std::priority_queue can only be modified from the back, not from the front. Important functions are front(), push_back(), and pop_back().

2.8 Stack

std::stack is a LIFO (last in, first out). Important functions are back(), front(), push_back().



2.9 Tries - Retrieval Trees - AutoComplete

'std::queue' and 'std::stack' containers as part of the standard library, but tries (retrieval trees) are not implemented.

Here, and example with 'std::unordered_map' or 'std::map' for the children nodes, and custom classes or structs to represent the nodes.

Yet, third-party libraries have implementions of trie data structures.

```
#include <unordered_map>

class TrieNode {
public:
    bool isEndOfWord;
    std::unordered_map<char, TrieNode*> children;

    TrieNode() : isEndOfWord(false) {}
};

class Trie {
private:
    TrieNode* root;
```

```
public:
    Trie() {
        root = new TrieNode();
    }

    void insert(const std::string& word) {
        TrieNode* curr = root;
        for (char c : word) {
            if (curr->children.find(c) == curr->children.end()) {
                curr->children[c] = new TrieNode();
            }
            curr = curr->children[c];
        }
        curr->isEndOfWord = true;
    }

    bool search(const std::string& word) {
        TrieNode* curr = root;
        for (char c : word) {
            if (curr->children.find(c) == curr->children.end()) {
                return false;
            }
            curr = curr->children[c];
        }
        return curr->isEndOfWord;
    }
};

int main() {
    Trie trie;

    // Insert words into the trie
    trie.insert("apple");
    trie.insert("banana");
    trie.insert("cat");
    trie.insert("dog");

    // Search for words in the trie
    std::cout << trie.search("apple") << std::endl;  // Output: 1 (true)
    std::cout << trie.search("banana") << std::endl; // Output: 1 (true)
    std::cout << trie.search("cat") << std::endl;    // Output: 1 (true)
    std::cout << trie.search("dog") << std::endl;    // Output: 1 (true)
    std::cout << trie.search("car") << std::endl;    // Output: 0 (false)

    return 0;
}
```

2.9.1 Serialize a Trie with JSON or XML

To save a retrieval tree (trie) and use it later without recreating the tree every time, you can serialize the tree data structure to a file and then deserialize it when needed. This allows you to persist the

tree structure to disk and load it back into memory when required.

Here are the general steps to achieve this:

1. Serialize the trie: Traverse the trie and convert its nodes and data into a serialized representation that can be written to a file. This typically involves converting the tree nodes and their contents into a suitable format, such as JSON, XML, or a custom binary format.

2. Write the serialized data to a file: Open a file in write mode and write the serialized data to the file. You can use file I/O operations provided by the programming language or libraries to accomplish this.

3. Save the file: Close the file and make sure it is saved to a location of your choice, such as a specific directory.

Use the saved trie later:

1. Read the serialized data from the file: Open the saved file in read mode and read the serialized data from it.

2. Deserialize the data: Convert the serialized data back into the original trie data structure. This involves parsing the serialized format and reconstructing the trie nodes and their relationships.

3. Use the trie: Once the trie is deserialized, you can use it in your program for retrieval or any other operations as needed.

By saving and loading the serialized trie data, you avoid the need to recreate the entire trie every time your program runs, improving efficiency and performance.

The exact implementation details will depend on the programming language you're using and the specific serialization and deserialization mechanisms available. Many programming languages provide libraries or built-in features for serialization, such as JSON or XML parsers, binary serialization libraries, or custom serialization methods.

2.10 Algorithms

2.10.1 Accumulate

```
// the algorithm everybody knows. (For_each and accumulate)

#include <numeric>

template<typename T>
T sum_data(const std::vector<T> &d) {
    return std::accumulate(d.begin(), d.end(), T());
}
```

2.10.2 Std::Puts

Generating a null-terminated string, a sequence of characters stored in an array where the end of the string is marked by a null character (''). The null character serves as a sentinel value to indicate the end of the string

```
#include <format>
#include <string_view>

void print_map(const auto &map, const std::string_view &key_desc = "key",
                                const std::string_view &value_desc = "value")
{
    for (const auto &[key, value] : map) /// structured binding
    {
        std::puts(std::format("{}: '{}' {}: '{}'",
                        key_desc, key, value_desc, value).c_str());

        // this is genious spacing for readability
    }
}

Standard c++20
```

2.10.3 Algorithms and Standard Template Library

```
set<>
vector<>
for_each<>
any_of<>
etc.

A generic set of composable tools

// now, here we are

#include <numeric>
#include <vector>

template<typename Value_Type>
std::vector<Value_Type> get_data(const Value_Type &v1, const Value_type &v2,
                                 const Value_type &v3)
{
    std::vector<Value_Type> data;
    data.push_back(v1);
    data.push_back(v2);
    data.push_back(v3);
    return data;
}

template<typename T>
T sum_data(const std::vector<T> &d) {
    return std::accumulate(d.begin(), d.end(), T());
}

int main() {
    return sum)data(get_data(1,2,3));
}
```

```
// But we know the amount of data at compile-time.
// If only there was some fixed-size container available!
```

2.10.4 Prefer Algorithms Over Loops

Algorithms communicate meaning and help "const all the things".

Taking a functional approach and using algorithms, we can write cleaner c++.

```
// Algorithms end game (before C++20)

const auto has_value
    = std::any_of(begin(container), end(container),
            greater_than(12));

// Algorithm end game (c++20)

const auto has_value
    = std::any_of(container, greater_than(12));

Next time you are reading through a loop in your codebase,
cross-reference it with the C++ <algorithm> header[2]
and try to find an algorithm that applies instead.

https://en.cppreference.com/w/cpp/algorithm
```

Chapter 3

Cmake

3.1 Starter Pack - Jason Turner's Template

```
lefticus/cmake_template // Jason Turner 2023 cmake starter pack

rename "myproject" in the cmake files to use it.
```

3.1.1 Lefticus Defaults - ProjectOptions.cmake

```
Address sanitizer
Undefined behavior sanitizer
Fuzzing example built
Procedural optimization IPO (link time optimization)
Warnings as errors
Clang-tidy enabled
CPPcheck enabled
Options for precompiled headers
```

3.1.2 Hardening - Hardening.cmake

```
Hardened compilation // make code safer
More compilation options / securities.

-fstack-protector
-fcf-protection
-fsanitize=undefined // undefined behavior sanitizer
-fno-sanitize-recover=undefined
-fsanitizise-minimal-runtime

+ debug information
```

3.2 Simple Cmake (Modern)

3.2.1 Context

Cmake is "a generator of make files", it abstracts away makefile complexity.

```
First, cmake // generate make files
Second, make // run make files
Last, ./hello // run the created executable
```

To create an executable of hello.cpp. We usually:

```
:wq // quit vim
g++ main.cpp -o hello // compile
./hello // run the executable
```

3.2.2 CMakeLists.txt

With Cmake, we can have:

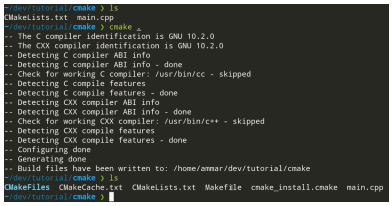
```
// have cmake installed

cmake_minimum_required(VERSION 3.10)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

project(hello VERSION 1.0)
add_executable(hello main.cpp)
```

Run CMake from the command line, specify a directory.

```
cmake . && make && ./hello
```

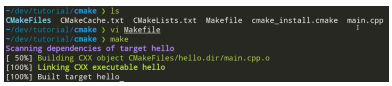
3.2.3 Cmake .

Generates all needed make files (Artifacts).

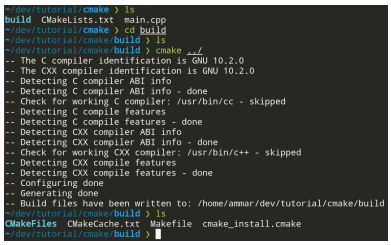


3.2.4 Make

With the MakeFile generated by cmake. Build your binary (the executable) with:



3.2.5 Build folder



3.2.6 Sick CMake Vim plugins combos

See codevion/cpp2.md

3.2.7 COC - for code completion in nvim

https://github.com/neoclide/coc.nvim
Jason Turner has this too

3.2.8 Include Header File - CMake Continued

```
target_include_directories(hello PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include)

    // Standard is having header files in /include directory!

hello // our target, where to add the stuff from headers
PUBLIC // gives the scope of added stuff from headers.
       // Public, Private or Interface
       // Usage: when you have cmake library, make sure it is seen by #include in files

cmake_minimum_required(VERSION 3.10)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

project(hello VERSION 1.0)
add_executable(hello main.cpp)
target_include_directories(hello PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include)
```

In header file

```
#pragma once
#include <iostream>

class Blah {
    public:
        inline void boo() {
            std::cout << "Boo!\n";
        }
```

```
};
```

3.2.9 Pragma Once

```
#pragma once is a non-standard directive that serves as an include guard.
It ensures that a header file is included only once during the compilation process,
regardless of how many times it is referenced.

Placed at the beginning of a header file, it acts as a compiler directive to prevent multiple inclusi
Supported by most compilers, including GCC, Clang, and MSVC.
```

3.2.10 Glob - Include Many files with CMake

You have at least two options. First, include every files one-by-one in the CMakeList.txt.

```
cmake_minimum_required(VERSION 3.10)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

project(hello VERSION 1.0) // traditional way to include files
add_executable(hello main.cpp Blah.cpp) // added here
target_include_directories(hello PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include)
```

Cmake discourages this glob method, but it is a more sane options for large projects.

```
file(GLOB_RECURSE SRC_FILES src/*.cpp) // glob everything in src/
add_executable(hello ${SRC_FILES})
```

3.2.11 /src Directory - source files

Declaration in the header file **blah.h**.

```
#pragma once

class Blah {
    public:
        void boo(); // declaring function boo in header
}
```

Definition (implementation) of class Blah in the source files **blah.cpp**.

```
#include "blah.h"
#include <iostream>

void Blah::boo() {
    std::cout << "Boo!\n"; // defining function boo in src file
}
```

In CMake - Traditionally Added

```
add_executable(hello main.cpp src/Blah.cpp) // added here
```

In CMake - Globing

```
file(GLOB_RECURSE SRC_FILES src/*.cpp)
add_executable(hello main.cpp ${SRC_FILES})
```

3.2.12 CMake Custom Libraries

Create a lib from some source files:

```
Replace add_executable with add_library

add_library(mylib STATIC lib/blah.cpp) // create a library
                                       // Staticly linked or dynamicly linked
```

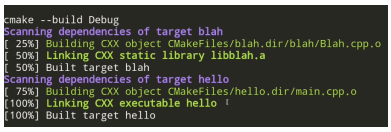
Then, include it in your main executable

```
target_link_libraries(hello Public mylib)
```

3.2.13 Custom Library Implementation - Blah example


```
cmake_minimum_required(VERSION 3.10)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

project(hello VERSION 1.0)

add_library(blah STATIC blah/Blah.cpp)
target_include_directories(blah PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/blah/include)

// file(GLOB_RECURSE SRC_FILES src/*.cpp)
// now useless, since main.cpp has #include added library

// We are linking our library with our executable directly, with
// target_include_libraries

add_executable(hello main.cpp)
target_link_libraries(hello PUBLIC blah)
```

The target link generates a libblah.a



3.3 Jason Turner's CMake Template - Options

```
include(cmake/SystemLink.cmake)
include(cmake/LibFuzzer.cmake)
include(CMakeDependentOption)
include(CheckCXXCompilerFlag)
```

```
macro(myproject_supports_sanitizers)
  if((CMAKE_CXX_COMPILER_ID MATCHES ".*Clang.*" OR CMAKE_CXX_COMPILER_ID MATCHES ".*GNU.*") AND NOT W
    set(SUPPORTS_UBSAN ON)
  else()
    set(SUPPORTS_UBSAN OFF)
  endif()

  if((CMAKE_CXX_COMPILER_ID MATCHES ".*Clang.*" OR CMAKE_CXX_COMPILER_ID MATCHES ".*GNU.*") AND WIN32
    set(SUPPORTS_ASAN OFF)
  else()
    set(SUPPORTS_ASAN ON)
  endif()
endmacro()

macro(myproject_setup_options)
  option(myproject_ENABLE_HARDENING "Enable hardening" ON)
  option(myproject_ENABLE_COVERAGE "Enable coverage reporting" OFF)
  cmake_dependent_option(
    myproject_ENABLE_GLOBAL_HARDENING
    "Attempt to push hardening options to built dependencies"
    ON
    myproject_ENABLE_HARDENING
    OFF)

  myproject_supports_sanitizers()

  if(NOT PROJECT_IS_TOP_LEVEL OR myproject_PACKAGING_MAINTAINER_MODE)
    option(myproject_ENABLE_IPO "Enable IPO/LTO" OFF)
    option(myproject_WARNINGS_AS_ERRORS "Treat Warnings As Errors" OFF)
    option(myproject_ENABLE_USER_LINKER "Enable user-selected linker" OFF)
    option(myproject_ENABLE_SANITIZER_ADDRESS "Enable address sanitizer" OFF)
    option(myproject_ENABLE_SANITIZER_LEAK "Enable leak sanitizer" OFF)
    option(myproject_ENABLE_SANITIZER_UNDEFINED "Enable undefined sanitizer" OFF)
    option(myproject_ENABLE_SANITIZER_THREAD "Enable thread sanitizer" OFF)
    option(myproject_ENABLE_SANITIZER_MEMORY "Enable memory sanitizer" OFF)
    option(myproject_ENABLE_UNITY_BUILD "Enable unity builds" OFF)
    option(myproject_ENABLE_CLANG_TIDY "Enable clang-tidy" OFF)
    option(myproject_ENABLE_CPPCHECK "Enable cpp-check analysis" OFF)
    option(myproject_ENABLE_PCH "Enable precompiled headers" OFF)
    option(myproject_ENABLE_CACHE "Enable ccache" OFF)
  else()
    option(myproject_ENABLE_IPO "Enable IPO/LTO" ON)
    option(myproject_WARNINGS_AS_ERRORS "Treat Warnings As Errors" ON)
    option(myproject_ENABLE_USER_LINKER "Enable user-selected linker" OFF)
    option(myproject_ENABLE_SANITIZER_ADDRESS "Enable address sanitizer" ${SUPPORTS_ASAN})
    option(myproject_ENABLE_SANITIZER_LEAK "Enable leak sanitizer" OFF)
    option(myproject_ENABLE_SANITIZER_UNDEFINED "Enable undefined sanitizer" ${SUPPORTS_UBSAN})
    option(myproject_ENABLE_SANITIZER_THREAD "Enable thread sanitizer" OFF)
    option(myproject_ENABLE_SANITIZER_MEMORY "Enable memory sanitizer" OFF)
    option(myproject_ENABLE_UNITY_BUILD "Enable unity builds" OFF)
```

```
    option(myproject_ENABLE_CLANG_TIDY "Enable clang-tidy" ON)
    option(myproject_ENABLE_CPPCHECK "Enable cpp-check analysis" ON)
    option(myproject_ENABLE_PCH "Enable precompiled headers" OFF)
    option(myproject_ENABLE_CACHE "Enable ccache" ON)
  endif()

  if(NOT PROJECT_IS_TOP_LEVEL)
    mark_as_advanced(
      myproject_ENABLE_IPO
      myproject_WARNINGS_AS_ERRORS
      myproject_ENABLE_USER_LINKER
      myproject_ENABLE_SANITIZER_ADDRESS
      myproject_ENABLE_SANITIZER_LEAK
      myproject_ENABLE_SANITIZER_UNDEFINED
      myproject_ENABLE_SANITIZER_THREAD
      myproject_ENABLE_SANITIZER_MEMORY
      myproject_ENABLE_UNITY_BUILD
      myproject_ENABLE_CLANG_TIDY
      myproject_ENABLE_CPPCHECK
      myproject_ENABLE_COVERAGE
      myproject_ENABLE_PCH
      myproject_ENABLE_CACHE)
  endif()

  myproject_check_libfuzzer_support(LIBFUZZER_SUPPORTED)
  if(LIBFUZZER_SUPPORTED AND (myproject_ENABLE_SANITIZER_ADDRESS OR myproject_ENABLE_SANITIZER_THREAD
    set(DEFAULT_FUZZER ON)
  else()
    set(DEFAULT_FUZZER OFF)
  endif()

  option(myproject_BUILD_FUZZ_TESTS "Enable fuzz testing executable" ${DEFAULT_FUZZER})

endmacro()

macro(myproject_global_options)
  if(myproject_ENABLE_IPO)
    include(cmake/InterproceduralOptimization.cmake)
    myproject_enable_ipo()
  endif()

  myproject_supports_sanitizers()

  if(myproject_ENABLE_HARDENING AND myproject_ENABLE_GLOBAL_HARDENING)
    include(cmake/Hardening.cmake)
    if(NOT SUPPORTS_UBSAN
       OR myproject_ENABLE_SANITIZER_UNDEFINED
       OR myproject_ENABLE_SANITIZER_ADDRESS
       OR myproject_ENABLE_SANITIZER_THREAD
       OR myproject_ENABLE_SANITIZER_LEAK)
      set(ENABLE_UBSAN_MINIMAL_RUNTIME FALSE)
```

```
    else()
      set(ENABLE_UBSAN_MINIMAL_RUNTIME TRUE)
    endif()
    message("${myproject_ENABLE_HARDENING} ${ENABLE_UBSAN_MINIMAL_RUNTIME} ${myproject_ENABLE_SANITIZ
    myproject_enable_hardening(myproject_options ON ${ENABLE_UBSAN_MINIMAL_RUNTIME})
  endif()
endmacro()

macro(myproject_local_options)
  if(PROJECT_IS_TOP_LEVEL)
    include(cmake/StandardProjectSettings.cmake)
  endif()

  add_library(myproject_warnings INTERFACE)
  add_library(myproject_options INTERFACE)

  include(cmake/CompilerWarnings.cmake)
  myproject_set_project_warnings(
    myproject_warnings
    ${myproject_WARNINGS_AS_ERRORS}
    ""
    ""
    ""
    "")

  if(myproject_ENABLE_USER_LINKER)
    include(cmake/Linker.cmake)
    configure_linker(myproject_options)
  endif()

  include(cmake/Sanitizers.cmake)
  myproject_enable_sanitizers(
    myproject_options
    ${myproject_ENABLE_SANITIZER_ADDRESS}
    ${myproject_ENABLE_SANITIZER_LEAK}
    ${myproject_ENABLE_SANITIZER_UNDEFINED}
    ${myproject_ENABLE_SANITIZER_THREAD}
    ${myproject_ENABLE_SANITIZER_MEMORY})

  set_target_properties(myproject_options PROPERTIES UNITY_BUILD ${myproject_ENABLE_UNITY_BUILD})

  if(myproject_ENABLE_PCH)
    target_precompile_headers(
      myproject_options
      INTERFACE
      <vector>
      <string>
      <utility>)
  endif()

  if(myproject_ENABLE_CACHE)
```

```
    include(cmake/Cache.cmake)
    myproject_enable_cache()
  endif()

  include(cmake/StaticAnalyzers.cmake)
  if(myproject_ENABLE_CLANG_TIDY)
    myproject_enable_clang_tidy(myproject_options ${myproject_WARNINGS_AS_ERRORS})
  endif()

  if(myproject_ENABLE_CPPCHECK)
    myproject_enable_cppcheck(${myproject_WARNINGS_AS_ERRORS} "" # override cppcheck options
    )
  endif()

  if(myproject_ENABLE_COVERAGE)
    include(cmake/Tests.cmake)
    myproject_enable_coverage(myproject_options)
  endif()

  if(myproject_WARNINGS_AS_ERRORS)
    check_cxx_compiler_flag("-Wl,--fatal-warnings" LINKER_FATAL_WARNINGS)
    if(LINKER_FATAL_WARNINGS)
      # This is not working consistently, so disabling for now
      # target_link_options(myproject_options INTERFACE -Wl,--fatal-warnings)
    endif()
  endif()

  if(myproject_ENABLE_HARDENING AND NOT myproject_ENABLE_GLOBAL_HARDENING)
    include(cmake/Hardening.cmake)
    if(NOT SUPPORTS_UBSAN
       OR myproject_ENABLE_SANITIZER_UNDEFINED
       OR myproject_ENABLE_SANITIZER_ADDRESS
       OR myproject_ENABLE_SANITIZER_THREAD
       OR myproject_ENABLE_SANITIZER_LEAK)
      set(ENABLE_UBSAN_MINIMAL_RUNTIME FALSE)
    else()
      set(ENABLE_UBSAN_MINIMAL_RUNTIME TRUE)
    endif()
    myproject_enable_hardening(myproject_options OFF ${ENABLE_UBSAN_MINIMAL_RUNTIME})
  endif()

endmacro()
```

3.3.1 CPM (C++ package manager)

A nice section to expand on later.

Chapter 4

GDB - GNU Debugger

4.1 Keywords

```
Seen commands:
info gdb  //Manual
info locals          //Vars in local scope
info variables //Vars declared outside current scope
info functions //Names and datatypes of all defined functions
info b  //List all breakpoints
break funcName //Set breakpoint at function funcName (short: b funcName)
break file::line         //Set breakpoint at line in file
layout next //Cycle through the layouts of gdb
p var  //Print the value of variable var
p var = value           //Force set value to a var
run          //Start the program
start  //Synonymous to (b main && run). Puts temporary b at main
next  //Execute the current line in source (short: n)
step  //Step into function call at current line (short: s)
finish //Finish the execution of current function (short: fin)
continue //Resume execution (After a breakpoint) (short: c)
refresh          //Repaint the interface (To fix corrupted interface)
shell cmd  //Run shell command cmd from gdb prompt
python gdb.execute(cmd) //Run a gdb command cmd from python prompt
set print pretty on //Enable pretty printing
  (Put in ~/.gdbinit)
$ gdb -c core.num //Examine the dumped core file from a SIGSEGV(shell command)
bt //Print backtrace
break _exit          //Breakpoint at exit of program
whatis expr //Print datatype of expr
ptype expr //Detailed print of datatype of expr
watch var  //Stop when var is modified
watch -l foo         //Watch foo loaction
rwatch foo //Stop when foo is read
watch foo if foo>10 //Watch foo conditionally
delete //Delete all breakpoints
```

```
delete breakpoint_no          //Delete breakpoint breakpoint_no
command breakpoint_no //Run user listed commands when breakpoint is hit
  (End list of commands with 'end')
file executable           //Load the executable for debugging from inside gdb
quit          //Quit (short: q)

Feel free to correct/add any useful command you know.
```