

## 14.x — Chapter 14 summary and quiz

👤 [ALEX](#)<sup>1</sup> ⌚ AUGUST 17, 2024

In this chapter, we explored the meat of C++ -- classes! This is the most important chapter in the tutorial series, as it sets the stage for much of what's left to come.

### Chapter Review

In **procedural programming**, the focus is on creating “procedures” (which in C++ are called functions) that implement our program logic. We pass data objects to these functions, those functions perform operations on the data, and then potentially return a result to be used by the caller.

With **Object-oriented programming** (often abbreviated as OOP), the focus is on creating program-defined data types that contain both properties and a set of well-defined behaviors.

A **class invariant** is a condition that must be true throughout the lifetime of an object in order for the object to remain in a valid state. An object that has a violated class invariant is said to be in an **invalid state**, and unexpected or undefined behavior may result from further use of that object.

A **class** is a program-defined compound type that bundles both data and functions that work on that data.

Functions that belong to a class type are called **member functions**. The object that a member function is called on is often called the **implicit object**. Functions that are not member functions are called **non-member functions** to distinguish them from member functions. If your class type has no data members, prefer using a namespace instead.

A **const member function** is a member function that guarantees it will not modify the object or call any non-const member functions (as they may modify the object). A member function that does not (and will not ever) modify the state of the object should be made `const`, so that it can be called on both non-const and const objects.

Each member of a class type has a property called an **access level** that determines who can access that member. The access level system is sometimes informally called **access controls**. Access levels are defined on a per-class basis, not on a per-object basis.

**Public members** are members of a class type that do not have any restrictions on how they can be accessed. Public members can be accessed by anyone (as long as they are in scope). This includes other members of the same class. Public members can also be accessed by **the public**, which is what we call code that exists outside the members of a given class type. Examples of the public include non-member functions, as well as the members of other class types.

By default, all members of a struct are public members.

**Private members** are members of a class type that can only be accessed by other members of the same class.

By default, the members of a class are private. A class with private members is no longer an aggregate, and therefore can no longer use aggregate initialization. Consider naming your private members starting with

an “m\_” prefix to help distinguish them from the names of local variables, function parameters, and member functions.

We can explicitly set the access level of our members by using an **access specifier**. Structs should generally avoid using access specifiers so all members default to public.

An **access function** is a trivial public member function whose job is to retrieve or change the value of a private member variable. Access functions come in two flavors: getters and setters. **Getters** (also sometimes called **accessors**) are public member functions that return the value of a private member variable. **Setters** (also sometimes called **mutators**) are public member functions that set the value of a private member variable.

The **interface** of a class type defines how a user of the class type will interact with objects of the class type. Because only public members can be accessed from outside of the class type, the public members of a class type form its interface. For this reason, an interface composed of public members is sometimes called a **public interface**.

The **implementation** of a class type consists of the code that actually makes the class behave as intended. This includes both the member variables that store data, and the bodies of the member functions that contain the program logic and manipulate the member variables.

In programming, **data hiding** (also called **information hiding** or **data abstraction**) is a technique used to enforce the separation of interface and implementation by hiding the implementation of a program-defined data type from users.

The term **encapsulation** is also sometimes used to refer to data hiding. However, this term is also used to refer to the bundling of data and functions together (without regard for access controls), so its use can be ambiguous.

When defining a class, prefer to declare your public members first and your private members last. This spotlights the public interface and de-emphasizes implementation details.

A **constructor** is a special member function that is used to initialize class type objects. A matching constructor must be found in order to create a non-aggregate class type object.

A **Member initializer list** allows you to initialize your member variables from within a constructor. Member variables in a member initializer list should be listed in order that they are defined in the class. Prefer using the member initializer list to initialize your members over assigning values in the body of the constructor.

A constructor that takes no parameters (or has all default parameters) is called a **default constructor**. The default constructor is used if no initialization values are provided by the user. If a non-aggregate class type object has no user-declared constructors, the compiler will generate a default constructor (so that the class can be value or default initialized). This constructor is called an **implicit default constructor**.

Constructors are allowed to delegate initialization to another constructor from the same class type. This process is sometimes called **constructor chaining** and such constructors are called **delegating constructors**. Constructors can delegate or initialize, but not both.

A **temporary object** (sometimes called an **anonymous object** or an **unnamed object**) is an object that has no name and exists only for the duration of a single expression.

A **copy constructor** is a constructor that is used to initialize an object with an existing object of the same type. If you do not provide a copy constructor for your classes, C++ will create a public **implicit copy constructor** for you that does memberwise initialization.

The **as-if rule** says that the compiler can modify a program however it likes in order to produce more optimized code, so long as those modifications do not affect a program's "observable behavior". One exception to the as-if rule is copy elision. **Copy elision** is a compiler optimization technique that allows the compiler to remove unnecessary copying of objects. When the compiler optimizes away a call to the copy constructor, we say the constructor has been **elided**.

A function that we've written to convert a value to or from a program-defined type is called a **user-defined conversion**. A constructor that can be used to perform an implicit conversion is called a **converting constructor**. By default, all constructors are converting constructors.

We can use the **explicit** keyword to tell the compiler that a constructor should not be used as a converting constructor. Such a constructor can not be used to do copy initialization or copy list initialization, nor can it be used to do implicit conversions.

Make any constructor that accepts a single argument explicit by default. If an implicit conversion between types is both semantically equivalent and performant (such as a conversion from `std::string` to `std::string_view`), you can consider making the constructor non-explicit. Do not make copy or move constructors explicit, as these do not perform conversions.

Member functions (including constructors) may be constexpr. As of C++14, constexpr member functions are not implicitly const.

## Quiz time

### Author's note

The blackjack quiz that used to be part of this lesson has been moved to lesson [17.x -- Chapter 17 summary and quiz](https://www.learncpp.com/cpp-tutorial/chapter-17-summary-and-quiz/) (<https://www.learncpp.com/cpp-tutorial/chapter-17-summary-and-quiz/>)<sup>2</sup>.

### Question #1

a) Write a class named `Point2d`. `Point2d` should contain two member variables of type `double`: `m_x`, and `m_y`, both defaulted to `0.0`.

Provide a constructor and a `print()` function.

The following program should run:

```
1  #include <iostream>
2
3  int main()
4  {
5      Point2d first{};
6      Point2d second{ 3.0, 4.0 };
7
8      // Point2d third{ 4.0 }; // should error if uncommented
9
10     first.print();
11     second.print();
12
13     return 0;
14 }
```

This should print:

```
Point2d(0, 0)
Point2d(3, 4)
```

[Show Solution](#) (javascript:void(0))<sup>3</sup>

b) Now add a member function named `distanceTo()` that takes another `Point2d` as a parameter, and calculates the distance between them. Given two points  $(x_1, y_1)$  and  $(x_2, y_2)$ , the distance between them can be calculated using the formula  $\text{std::sqrt}((x_1 - x_2)*(x_1 - x_2) + (y_1 - y_2)*(y_1 - y_2))$ . The `std::sqrt` function lives in header `cmath`.

The following program should run:

```
1  #include <cmath>
2  #include <iostream>
3
4  int main()
5  {
6      Point2d first{};
7      Point2d second{ 3.0, 4.0 };
8
9      first.print();
10     second.print();
11
12     std::cout << "Distance between two points: " << first.distanceTo(second) << '\n';
13
14     return 0;
15 }
```

This should print:

```
Point2d(0, 0)
Point2d(3, 4)
Distance between two points: 5
```

[Show Solution](#) (javascript:void(0))<sup>3</sup>

## Question #2

In lesson [13.10 -- Passing and returning structs](https://www.learncpp.com/cpp-tutorial/passing-and-returning-structs/) (https://www.learncpp.com/cpp-tutorial/passing-and-returning-structs/)<sup>4</sup>, we wrote a short program using a `Fraction` struct. The reference solution looks like this:

```

1  #include <iostream>
2
3  struct Fraction
4  {
5      int numerator{ 0 };
6      int denominator{ 1 };
7  };
8
9  Fraction getFraction()
10 {
11     Fraction temp{};
12     std::cout << "Enter a value for numerator: ";
13     std::cin >> temp.numerator;
14     std::cout << "Enter a value for denominator: ";
15     std::cin >> temp.denominator;
16     std::cout << '\n';
17
18     return temp;
19 }
20
21 Fraction multiply(const Fraction& f1, const Fraction& f2)
22 {
23     return { f1.numerator * f2.numerator, f1.denominator * f2.denominator };
24 }
25
26 void printFraction(const Fraction& f)
27 {
28     std::cout << f.numerator << '/' << f.denominator << '\n';
29 }
30
31 int main()
32 {
33     Fraction f1{ getFraction() };
34     Fraction f2{ getFraction() };
35
36     std::cout << "Your fractions multiplied together: ";
37
38     printFraction(multiply(f1, f2));
39
40     return 0;
41 }

```

Convert `Fraction` from a struct to a class. Convert all of the functions to (non-static) member functions.

### Author's note

Note: This quiz does not comply with best practices for when you should use a non-member or member function. The goal is to test whether you understand how to convert non-member functions to member functions.

[Show Solution](#).(javascript:void(0))<sup>3</sup>

### Question #3

In the prior quiz solution, why was the `Fraction` constructor made `explicit`?

[Show Solution](#).(javascript:void(0))<sup>3</sup>

### Question #4

In the prior quiz question, why might it be better to leave `getFraction()` and `print()` as non-members?



## Next lesson

15.1 [The hidden "this" pointer and member function chaining](#)

5



[Back to table of contents](#)

6



## Previous lesson

14.17 [Constexpr aggregates and classes](#)

7

8



B

U

URL

INLINE CODE

C++ CODE BLOCK

HELP!

Leave a comment...



Name\*



Email\*



Notify me about replies:



POST COMMENT

🚩 Find a mistake? Leave a comment above!?

👤 Avatars from <https://gravatar.com/><sup>11</sup> are connected to your provided email address.

868 COMMENTS

Newest ▼



Copernicus

🕒 June 1, 2025 1:04 am PDT

Question #3

To stop implicit conversations by the compiler.

Question #4

Abstraction and data hiding.



0



Reply



**victorxu**

Reply to [Copernicus](#)<sup>12</sup> ⌚ June 18, 2025 9:07 am PDT

yeah we don't want the compiler to talk behind our back

👍 0

➡ Reply



**Copernicus**

⌚ June 1, 2025 1:01 am PDT

Question #2

```

1  #include <iostream>
2
3  class Fraction
4  {
5  public:
6      Fraction(int numerator, int denominator)
7          : m_numerator{ numerator }, m_denominator{ denominator }
8      {
9
10     }
11
12     Fraction()
13     {
14         std::cout << "Enter a value for numerator: ";
15         std::cin >> m_numerator;
16         std::cout << "Enter a value for denominator: ";
17         std::cin >> m_denominator;
18     }
19
20     void print() const
21     {
22         std::cout << m_numerator << "/" << m_denominator << '\n';
23     }
24
25     int getNumerator() const
26     {
27         return m_numerator;
28     }
29     int getDenominator() const
30     {
31         return m_denominator;
32     }
33
34     void multiple(const Fraction& other) const
35     {
36         std::cout << "Your fractions multiplied together: " << m_numerator *
other.getNumerator() << "/" << m_denominator * other.getDenominator() << '\n';
37     }
38
39 private:
40     int m_numerator{ 0 };
41     int m_denominator{ 1 };
42 };
43
44 int main()
45 {
46     Fraction first{};
47     Fraction second{};
48     first.print();
49     second.print();
50     first.multiple(second);
51
52     return 0;
53 }

```



0



Reply



Copernicus

🕒 June 1, 2025 12:35 am PDT

Question #1



```

1  #include <iostream>
2  #include <cmath>
3
4  class Point2d
5  {
6  public:
7      Point2d() = default;
8      Point2d(double x, double y)
9          : m_x{ x }, m_y{ y }
10     {
11     }
12
13     void print() const
14     {
15         std::cout << "Point(" << m_x << ", " << m_y << ")\n";
16     }
17
18     double disatnceTo(const Point2d& other) const
19     {
20         return std::sqrt((m_x - other.m_x) * ((m_x - other.m_x)) + (m_y - other.m_y) *
21 (m_y - other.m_y));
22     }
23
24 private:
25     double m_x{ 0.0 };
26     double m_y{ 0.0 };
27 };
28
29 int main()
30 {
31     Point2d first{};
32     Point2d second{ 3.0, 4.0 };
33
34     //Point2d third{ 4.0 }; // should error if uncommented
35
36     first.print();
37     second.print();
38
39     std::cout << first.disatnceTo(second);
40
41     return 0;
42 }

```



0



Reply



AJAY.CHALLA

🕒 May 23, 2025 11:34 pm PDT

QUESTION#1

a)

```

1  #include <iostream>
2  using namespace std;
3
4  class Point2d
5  {
6      private:
7          double m_x {0.0};
8          double m_y {0.0};
9
10     public:
11         Point2d() = default;
12
13         Point2d(double x, double y)
14             :m_x{x}, m_y{y}
15         {
16         }
17
18         void print() const
19         {
20             cout << "Point2d(" << m_x << "," << m_y << ")\n";
21         }
22     };
23     int main()
24     {
25         Point2d fir{};
26         Point2d sec{3.0,4.0};
27
28         fir.print();
29         sec.print();
30
31         return 0;
32     }
33 }

```



0

Reply



**X\_Taste**

🕒 May 15, 2025 11:24 am PDT

I already feel that I'm going to come back, it's slowly becoming a law of nature



0

Reply



**smart**

🕒 May 8, 2025 3:27 pm PDT

I decided to try Visual Studio Code instead of Visual Studio 2022... it was so exhausting that I'll probably stick with my usual IDE.

**Question #1**

```

1  #include <cmath>
2  #include <iostream>
3
4  class Point2d
5  {
6  public:
7      constexpr explicit Point2d(double x = defaultPoint,
8                                double y = defaultPoint)
9          : m_x{ x }, m_y{ y }
10         {
11         }
12
13     void print() const
14     {
15         std::cout << "Point2d(" << m_x << ", "
16                   << m_y << ")\n";
17     }
18
19     constexpr double distanceTo(const Point2d& s) const
20     {
21         // g++ -std=gnu++2c (Or implement custom constexpr sqrt)
22         return std::sqrt((m_x - s.m_x) * (m_x - s.m_x)
23                        +
24                        (m_y - s.m_y) * (m_y - s.m_y));
25     }
26
27 private:
28     static constexpr double defaultPoint{ 0.0 };
29
30     double m_x{ defaultPoint };
31     double m_y{ defaultPoint };
32 };
33
34 constexpr auto CONSTEVAL(auto value)
35 {
36     return value;
37 }
38
39 int main()
40 {
41     constexpr Point2d first{};
42     constexpr Point2d second{ 3.0, 4.0 };
43
44     first.print();
45     second.print();
46
47     std::cout << "Distance between two points: "
48               << CONSTEVAL(first.distanceTo(second))
49               << '\n';
50
51     return 0;
52 }

```

**Question #2** Tried to make minimal changes to the reference solution

```

1  #include <iostream>
2
3  class Fraction
4  {
5  public:
6      explicit Fraction(int x = 0, int y = 1)
7          : numerator{ x }, denominator{ y }
8      {
9      }
10
11     Fraction getFraction()
12     {
13         Fraction temp{};
14         std::cout << "Enter a value for numerator: ";
15         std::cin >> temp.numerator;
16         std::cout << "Enter a value for denominator: ";
17         std::cin >> temp.denominator;
18         std::cout << '\n';
19
20         return temp;
21     }
22
23     Fraction multiply(const Fraction& f1, const Fraction& f2)
24     {
25         return Fraction{ f1.numerator * f2.numerator,
26                           f1.denominator * f2.denominator };
27     }
28
29     void printFraction(const Fraction& f)
30     {
31         std::cout << f.numerator << '/' << f.denominator << '\n';
32     }
33
34 private:
35     int numerator{ 0 };
36     int denominator{ 1 };
37 };
38
39 int main()
40 {
41     Fraction f1{ f1.getFraction() };
42     Fraction f2{ f2.getFraction() };
43
44     std::cout << "Your fractions multiplied together: ";
45
46     f1.printFraction(f1.multiply(f1, f2));
47
48     return 0;
49 }

```



0



Reply



AJAY.CHALLA

Reply to [smart](#)<sup>13</sup> · May 23, 2025 11:37 pm PDT

Well, I actually think vs code is better than Visual studio because it does not take nearly 20 to 30 GB like Visual studio it takes less memory like around 300mb to 500mb and it is super fast while opening to!!! What is your opinion??



0



Reply



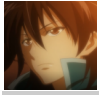
smart

Reply to [AJAY.CHALLA](#)<sup>14</sup> May 25, 2025 7:56 am PDT

When I use Visual Studio 2022, I can just sit down and start coding because everything I need is already built in. With VS Code, I spent about three days just trying to set it up properly. It was so exhausting that I don't even want to look at it anymore.

👍 0

➡ Reply



Zeca

May 1, 2025 4:46 pm PDT

## Question #1

```
1  #include <iostream>
2  #include <cmath>
3
4  class Point2d
5  {
6  private:
7      double m_x{0.0};
8      double m_y{0.0};
9
10 public:
11     Point2d(double x = 0.0, double y = 0.0)
12         : m_x{x}, m_y{y}
13     {
14     }
15
16     void print(void)
17     {
18         std::cout << "Point2d" << '(' << m_x << ',' << ' ' << m_y << ")\n";
19     }
20
21     double distanceTo(Point2d& point1)
22     {
23         return std::sqrt((point1.m_x - m_x)*(point1.m_x - m_x) + (point1.m_y - m_y)*
24 (point1.m_y - m_y));
25     }
26 };
27
28 int main()
29 {
30     Point2d first{};
31     Point2d second{ 3.0, 4.0 };
32
33     first.print();
34     second.print();
35
36     std::cout << "Distance between two points: " << first.distanceTo(second) << '\n';
37
38     return 0;
39 }
```

## Question #2

```

1  #include <iostream>
2
3  class Fraction
4  {
5  private:
6      int numerator{0};
7      int denominator{1};
8
9  public:
10     Fraction(int num = 0, int den = 1)
11         : numerator{num}, denominator{den}
12     {
13     }
14
15     void getFraction(void)
16     {
17         std::cout << "Enter a value for numerator: ";
18         std::cin >> numerator;
19         std::cout << "Enter a value for denominator: ";
20         std::cin >> denominator;
21         std::cout << '\n';
22     }
23
24     Fraction multiply(const Fraction& f2)
25     {
26         return {numerator * f2.numerator, denominator * f2.denominator};
27     }
28
29     void printFraction(void)
30     {
31         std::cout << numerator << '/' << denominator << '\n';
32     }
33
34
35 };
36
37
38 int main()
39 {
40     Fraction f1{};
41     Fraction f2{};
42
43     f1.getFraction();
44     f2.getFraction();
45
46     std::cout << "Your fractions multiplied together: ";
47
48     Fraction f3{f1.multiply(f2)};
49
50     f3.printFraction();
51
52     return 0;
53 }

```



1



Reply



**LittleBabyPigeon**

🕒 April 15, 2025 11:41 am PDT

Made the Fraction class mostly constexpr for fun. Also overloading std::cout << operator.



```

1 #include <iostream>
2 #include <numeric>          //for std::gcd
3
4 class Fraction
5 {
6 public:
7     //constructors
8     explicit constexpr Fraction()    //default constructor
9         : m_num{ 0 }, m_den{ 1 }
10    {};
11     explicit constexpr Fraction(int num, int den)
12         : m_num { num }, m_den { den }
13    {};
14
15     void getFraction()
16     {
17         std::cout << "Enter the numerator of the fraction: ";
18         std::cin >> m_num;
19         while(true)
20         {
21             std::cout << "Enter the denominator of the fraction: ";
22             std::cin >> m_den;
23             if(m_den == 0)
24             {
25                 std::cout << "The denominator cannot be zero. Please try again.\n";
26                 continue;
27             }
28             //if denominator is (-) swap for (-) numerator instead for easier math
29             if(m_den < 0)
30             {
31                 m_den *= -1;
32                 m_num *= -1;
33             }
34             break;
35         }
36
37         return;
38     }
39
40     constexpr Fraction simplify() const
41     {
42         int num{ m_num };
43         int den{ m_den };
44         while (std::gcd(num, den) != 1)
45         {
46             int gcd {std::gcd(num, den)};
47
48             num /= gcd;
49             den /= gcd;
50         }
51
52         return Fraction { num, den };
53     }
54
55     constexpr Fraction add(const Fraction& f2) const
56     {
57         return Fraction {m_num*f2.m_den + f2.m_num*m_den, m_den*f2.m_den}.simplify();
58     }
59
60     constexpr Fraction multiply(const Fraction& f2) const
61     {
62         return Fraction{m_num*f2.m_num, m_den*f2.m_den}.simplify();
63     }
64
65     constexpr Fraction divide(const Fraction& f2) const
66     {
67         return Fraction{ m_num*f2.m_den, m_den*f2.m_num }.simplify();
68     }
69
70     constexpr Fraction subtract(const Fraction& f2) const
71     {

```





```

71     {
72         return Fraction{m_num*f2.m_den - f2.m_num*m_den, m_den*f2.m_den}.simplify();
73     }
74
75     friend std::ostream& operator <<(std::ostream& out, Fraction f);
76
77 private:
78     int m_num {0};
79     int m_den {1};
80 };
81
82 std::ostream& operator <<(std::ostream& out, Fraction f)    //Fraction friend function
83 {
84     if(f.m_den == 0)
85         out << "NaN";
86     else if(f.m_num == 0)
87         out << 0;
88     else if(f.m_num == f.m_den)
89         out << 1;
90     else
91         out << f.m_num << '/' << f.m_den;
92
93     return out;
94 }
95
96 int main()
97 {
98     constexpr Fraction f1{ 1, 3 };
99     constexpr Fraction f2{2, 3};
100
101     std::cout << f1.subtract(f2) << '\n';
102     return 0;
103 }

```

 Last edited 2 months ago by LittleBabyPigeon

 1  Reply

 **LuluHuyen**  
 March 28, 2025 7:24 am PDT

I think for question 1, in order that `Point2d third{ 4.0 };` errors, we also have to add in `explicit` keyword for the user-defined constructor (perhaps the authors forget to put that in the statement of the question so people haven't noticed that and it hasn't existed in the solution either). Here is my answer:

```

1  #include <iostream>
2  #include <cmath>
3
4  class Point2d
5  {
6  public:
7      constexpr Point2d() = default;
8      constexpr explicit Point2d(double x, double y)
9          : m_x{x}, m_y{y} {}
10     void print() const
11     {
12         std::cout << "Point2d(" << m_x << ", " << m_y << ")\n";
13     };
14     constexpr double distanceTo(const Point2d &p) const
15     {
16         return std::sqrt((m_x - p.m_x) * (m_x - p.m_x) + (m_y - p.m_y) * (m_y -
17 p.m_y));
18     }
19
20 private:
21     double m_x{0.0};
22     double m_y{0.0};
23 };
24 int main()
25 {
26     constexpr Point2d first{};
27     constexpr Point2d second{3.0, 4.0};
28
29     Point2d third{4.0}; // should error if uncommented
30
31     first.print();
32     second.print();
33
34     std::cout << "Distance between two points: " << first.distanceTo(second) << '\n';
35
36     return 0;
37 }

```

 Last edited 2 months ago by LuluHuyen

 1  Reply



**Kania**

 March 12, 2025 11:41 pm PDT

I will just copy the solution and paste it here so people will think I completed the quiz :D

```

1  #include <iostream>
2
3  struct Fraction
4  {
5      int numerator{ 0 };
6      int denominator{ 1 };
7  };
8
9  Fraction getFraction()
10 {
11     Fraction temp{};
12     std::cout << "Enter a value for numerator: ";
13     std::cin >> temp.numerator;
14     std::cout << "Enter a value for denominator: ";
15     std::cin >> temp.denominator;
16     std::cout << '\n';
17
18     return temp;
19 }
20
21 Fraction multiply(const Fraction& f1, const Fraction& f2)
22 {
23     return { f1.numerator * f2.numerator, f1.denominator * f2.denominator };
24 }
25
26 void printFraction(const Fraction& f)
27 {
28     std::cout << f.numerator << '/' << f.denominator << '\n';
29 }
30
31 int main()
32 {
33     Fraction f1{ getFraction() };
34     Fraction f2{ getFraction() };
35
36     std::cout << "Your fractions multiplied together: ";
37
38     printFraction(multiply(f1, f2));
39
40     return 0;
41 }

```

 Last edited 3 months ago by Kania

 3  Reply

## Links

1. <https://www.learncpp.com/author/Alex/>
2. <https://www.learncpp.com/cpp-tutorial/chapter-17-summary-and-quiz/>
3. `javascript:void(0)`
4. <https://www.learncpp.com/cpp-tutorial/passing-and-returning-structs/>
5. <https://www.learncpp.com/cpp-tutorial/the-hidden-this-pointer-and-member-function-chaining/>

6. <https://www.learncpp.com/>
7. <https://www.learncpp.com/cpp-tutorial/constexpr-aggregates-and-classes/>
8. <https://www.learncpp.com/chapter-14-summary-and-quiz/>
9. <https://www.learncpp.com/cpp-tutorial/non-static-member-initialization/>
10. <https://www.learncpp.com/cpp-tutorial/header-guards/>
11. <https://gravatar.com/>
12. <https://www.learncpp.com/cpp-tutorial/chapter-14-summary-and-quiz/#comment-610612>
13. <https://www.learncpp.com/cpp-tutorial/chapter-14-summary-and-quiz/#comment-609899>
14. <https://www.learncpp.com/cpp-tutorial/chapter-14-summary-and-quiz/#comment-610355>
15. <https://g.ezoic.net/privacy/learncpp.com>