12.x — Chapter 12 summary and quiz

ALEX[1] **JANUARY 4, 2025**

Quick review

Compound data types (also called **composite data type**) are data types that can be constructed from fundamental data types (or other compound data types).

The **value category** of an expression indicates whether an expression resolves to a value, a function, or an object of some kind.

An **lvalue** is an expression that evaluates to a function or an object that has an identity. An object or function with an **identity** has an identifier or an identifiable memory address. Lvalues come in two subtypes: **modifiable lvalues** are lvalues that can be modified, and **non-modifiable lvalues** are lvalues whose values can't be modified (typically because they are const or constexpr).

An **rvalue** is an expression that is not an l-value. This includes literals (except string literals) and the return values of functions or operators (when returned by value).

A **reference** is an alias for an existing object. Once a reference has been defined, any operation on the reference is applied to the object being referenced. C++ contains two types of references: lvalue references and rvalue references. An **lvalue reference** (commonly just called a **reference**) acts as an alias for an existing lvalue (such as a variable). An **lvalue reference variable** is a variable that acts as a reference to an lvalue (usually another variable).

When a reference is initialized with an object (or function), we say it is **bound** to that object (or function). The object (or function) being referenced is sometimes called the **referent**.

Lvalue references can't be bound to non-modifiable lvalues or rvalues (otherwise you'd be able to change those values through the reference, which would be a violation of their const-ness). For this reason, lvalue references are occasionally called **lvalue references to non-const** (sometimes shortened to **non-const reference**).

Once initialized, a reference in C++ cannot be **reseated**, meaning it can not be changed to reference another object.

When an object being referenced is destroyed before a reference to it, the reference is left referencing an object that no longer exists. Such a reference is called a **dangling reference**. Accessing a dangling reference leads to undefined behavior.

By using the `const` keyword when declaring an lvalue reference, we tell an lvalue reference to treat the object it is referencing as const. Such a reference is called an **lvalue reference to a const value** (sometimes called a **reference to const** or a **const reference**). Const references can bind to modifiable lvalues, non-modifiable lvalues, and rvalues.

A **temporary object** (also sometimes called an **unnamed object** or **anonymous object**) is an object that is created for temporary use (and then destroyed) within a single expression.

When using **pass by reference**, we declare a function parameter as a reference (or const reference) rather than as a normal variable. When the function is called, each reference parameter is bound to the appropriate argument. Because the reference acts as an alias for the argument, no copy of the argument is made.

The **address-of operator** (&) returns the memory address of its operand. The **dereference operator** (*) returns the value at a given memory address as an lvalue.

A **pointer** is an object that holds a *memory address* (typically of another variable) as its value. This allows us to store the address of some other object to use later. Like normal variables, pointers are not initialized by default. A pointer that has not been initialized is sometimes called a **wild pointer**. A **dangling pointer** is a pointer that is holding the address of an object that is no longer valid (e.g. because it has been destroyed).

Besides a memory address, there is one additional value that a pointer can hold: a null value. A **null value** (often shortened to **null**) is a special value that means something has no value. When a pointer is holding a null value, it means the pointer is not pointing at anything. Such a pointer is called a **null pointer**. The **nullptr** keyword represents a null pointer literal. We can use `nullptr` to explicitly initialize or assign a pointer a null value.

A pointer should either hold the address of a valid object, or be set to `nullptr`. That way we only need to test pointers for null, and can assume any non-null pointer is valid.

A **pointer to a const value** (sometimes called a **pointer to const** for short) is a (non-const) pointer that points to a constant value.

A **const pointer** is a pointer whose address can not be changed after initialization.

A **const pointer to a const value** can not have its address changed, nor can the value it is pointing to be changed through the pointer.

With **pass by address**, instead of providing an object as an argument, the caller provides an object's address (via a pointer). This pointer (holding the address of the object) is copied into a pointer parameter of the called function (which now also holds the address of the object). The function can then dereference that pointer to access the object whose address was passed.

Return by reference returns a reference that is bound to the object being returned, which avoids making a copy of the return value. Using return by reference has one major caveat: the programmer must be sure that the object being referenced outlives the function returning the reference. Otherwise, the reference being returned will be left dangling (referencing an object that has been destroyed), and use of that reference will result in undefined behavior. If a parameter is passed into a function by reference, it's safe to return that parameter by reference.

If a function returns a reference, and that reference is used to initialize or assign to a non-reference variable, the return value will be copied (as if it had been returned by value).

Type deduction for variables (via the `auto` keyword) will drop any reference or top-level const qualifiers from the deduced type. These can be reapplied as part of the variable declaration if desired.

Return by address works almost identically to return by reference, except a pointer to an object is returned instead of a reference to an object.

Quiz time

Question #1

For each of the following expressions on the right side of operator <<, indicate whether the expression is an lvalue or rvalue:

a)

```
1  std::cout << 5;
```

Show Solution (javascript:void(0))[2]

b)

```
1  int x { 5 };
2  std::cout << x;
```

Show Solution (javascript:void(0))[2]

c)

```
1  int x { 5 };
2  std::cout << x + 1;
```

Show Solution (javascript:void(0))[2]

d)

```
1  int foo() { return 5; }
2  std::cout << foo();
```

Show Solution (javascript:void(0))[2]

e)

```
1  int& max(int &x, int &y) { return x > y ? x : y; }
2  int x { 5 };
3  int y { 6 };
4  std::cout << max(x, y);
```

Show Solution (javascript:void(0))[2]

Question #2

What is the output of this program?

```cpp
#include <iostream>

int main()
{
    int x{ 4 };
    int y{ 6 };

    int& ref{ x };
    ++ref;
    ref = y;
    ++ref;

    std::cout << x << ' ' << y;

    return 0;
}
```

Show Solution (javascript:void(0))[2]

Question #3

Name two reasons why we prefer to pass arguments by const reference instead of by non-const reference whenever possible.

Show Solution (javascript:void(0))[2]

Question #4

What is the difference between a const pointer and a pointer-to-const?

Show Solution (javascript:void(0))[2]

Question #5

Write a function named `sort2` which allows the caller to pass 2 int variables as arguments. When the function returns, the first argument should hold the lesser of the two values, and the second argument should hold the greater of the two values.

Hint: The `std::swap()` function (in the <algorithm> header) can be used to swap the values of two variables. e.g. `std::swap(x, y)` swaps the values of the variables `x` and `y`.

The following code should run and print the values noted in the comments:

```cpp
#include <iostream>

int main()
{
    int x { 7 };
    int y { 5 };

    std::cout << x << ' ' << y << '\n'; // should print 7 5

    sort2(x, y); // make sure sort works when values need to be swapped
    std::cout << x << ' ' << y << '\n'; // should print 5 7

    sort2(x, y); // make sure sort works when values don't need to be swapped
    std::cout << x << ' ' << y << '\n'; // should print 5 7

    return 0;
}
```

> ## Next lesson
> **13.1** Introduction to program-defined (user-defined) types

3

> ## 🏠 Back to table of contents

4

> ## Previous lesson
> **12.15** std::optional

5

6

| B | U | URL | INLINE CODE | C++ CODE BLOCK | HELP! |

Leave a comment...

👤 Name*

@ Email* | ⊘ | t

🐞 Find a mistake? Leave a comment above! ⊘

👤 Avatars from https://gravatar.com/[9] are connected to your provided email address.

Notify me about replies: 🔔

POST COMMENT

119 COMMENTS

Newest ▼

victorxu

🕐 June 12, 2025 10:41 am PDT

...

✍ *Last edited 19 days ago by victorxu*

👍 0   ↪ Reply

victorxu

```cpp
void sort2(int& x1, int&y1){

if(x1>y1){
const int x{x1};
const int y{y1};
y1=x;
x1=y;
}
}

int main()
{
int x { 5 };
int y { 6 };

std::cout << x << ' ' << y << '\n'; // should print 7 5

sort2(x, y); // make sure sort works when values need to be swapped
std::cout << x << ' ' << y << '\n'; // should print 5 7

sort2(x, y); // make sure sort works when values don't need to be swapped
std::cout << x << ' ' << y << '\n'; // should print 5 7

return 0;
}
```

👍 0　　➥ Reply

---

vb101

Well I believe, you can write your own swap method, but here's a neat one liner to swap two numbers

```cpp
void swap(int& x, int&y)
{
    x = (x + y) - (y = x);
}
```

So our swap2 function could be

```
1  void swap2(int& firstParam, int& secondParam)
2  {
3      if(firstParam > secondParam)
4      {
5          firstParam = (firstParam + secondParam) - (secondParam = firstParam);
6      }
7  }
```

✎ *Last edited 1 month ago by vb101*

👍 0   ➥ Reply

> ### Ivan
> 💬 Reply to vb101 [10]   ⊙ July 2, 2025 1:28 am PDT
>
> Is it a guarantee that `(x + y)` is evaluated before `(y = x)` ?
>
> 👍 0   ➥ Reply

Copernicus
⊙ May 21, 2025 6:06 am PDT

Question #5

```
1  #include <iostream>
2
3  static void sort2(int& smallerNumber, int& largerNumber)
4  {
5      int temporary{ smallerNumber };
6      if (smallerNumber > largerNumber)
7      {
8          smallerNumber = largerNumber;
9          largerNumber = temporary;
10     }
11 }
12
13 int main()
14 {
15     int x{ 7 };
16     int y{ 5 };
17
18     std::cout << x << ' ' << y << '\n'; // should print 7 5
19
20     sort2(x, y); // make sure sort works when values need to be swapped
21     std::cout << x << ' ' << y << '\n'; // should print 5 7
22
23     sort2(x, y); // make sure sort works when values don't need to be swapped
24     std::cout << x << ' ' << y << '\n'; // should print 5 7
25
26     return 0;
27 }
```

I made my own swap logic, should I use the already made std::swap function? (don't try and reinvent the wheel)

👍 0   ➥ Reply

Copernicus
🕒 May 21, 2025 5:55 am PDT

Question #2
7 6

Question #3
1; so the function can't modify the argument
2; const reference can accept non-modifiable lvalues, modifiable lvalues, rvalues arguments. non-const references only accept modifiable lvalue arguments.

Question #4
const pointer cannot be reseated, pointer to const, the variable its pointing to cannot be modified.

👍 0   ↪ Reply

Copernicus
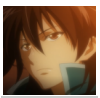🕒 May 21, 2025 5:47 am PDT

Question #1
rvalue
lvalue
rvalue
rvalue
rvalue (I didn't see that the function was returning by reference, so it makes it an lvalue).

👍 0   ↪ Reply

Zeca
🕒 April 22, 2025 7:47 pm PDT

Well, for my code i needed to create a copy of the x, but works.

```cpp
#include <iostream>

void sort2(int& x, int& y)
{
    int a{x};

    x = x < y ? x : y;

    y = a < y ? y : a;
}


int main()
{
    int x { 7 };
    int y { 5 };

    std::cout << x << ' ' << y << '\n'; // should print 7 5

    sort2(x, y); // make sure sort works when values need to be swapped
    std::cout << x << ' ' << y << '\n'; // should print 5 7

    sort2(x, y); // make sure sort works when values don't need to be swapped
    std::cout << x << ' ' << y << '\n'; // should print 5 7

    return 0;
}
```
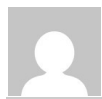
👍 1   ➥ Reply

---

smart
🕐 April 10, 2025 12:17 am PDT

I was mentally prepared to spend about 10 hours on the quiz...

👍 9   ➥ Reply

---

yzdpw
🕐 February 18, 2025 5:45 pm PST

```cpp
// The sort2 of Q5
namespace {
    void sort2(int& num1, int& num2) {
        if (num1 > num2) {
            int temp{ num1 };
            num1 = num2;
            num2 = temp;
        }
    }
}
```

👍 1   ➥ Reply

---

ice-shroom

in the fifth question, could we use an int function with a return value instead of void?

👍 0   ➦ Reply

Alex `Author`

💬 Reply to ice-shroom [11]   🕐 February 13, 2025 11:15 am PST

You're asking if you can return `int` instead of `void`? If so, yes, you can, but I don't think it makes sense to do so semantically. What would the return value mean?

👍 0   ➦ Reply

Bentley Spur

💬 Reply to Alex [12]   🕐 February 16, 2025 11:03 am PST

```cpp
1   #include <iostream>
2   #include <utility>
3
4   void sort2(int& x, int& y)
5   {
6       if (x > y)
7       {
8           std::swap(x, y);
9
10      }
11  }
12
13  int main()
14  {
15      int x{ 7 };
16      int y{ 5 };
17
18      std::cout << x << ' ' << y << '\n'; // should print 7 5
19
20      sort2(x, y); // make sure sort works when values need to be swapped
21      std::cout << x << ' ' << y << '\n'; // should print 5 7
22
23      sort2(x, y); // make sure sort works when values don't need to be
24  swapped
25      std::cout << x << ' ' << y << '\n'; // should print 5 7
26
27      return 0;
    }
```

Is this fine?

✎ *Last edited 4 months ago by Bentley Spur*

👍 0   ➦ Reply

Anatoliy

💬 Reply to Bentley Spur [13]   🕐 February 23, 2025 10:10 am PST

You can test your solution at any online compiler.

✎ *Last edited 4 months ago by Anatoliy*

Links

1. https://www.learncpp.com/author/Alex/
2. javascript:void(0)
3. https://www.learncpp.com/cpp-tutorial/introduction-to-program-defined-user-defined-types/
4. https://www.learncpp.com/
5. https://www.learncpp.com/cpp-tutorial/stdoptional/
6. https://www.learncpp.com/chapter-12-summary-and-quiz/
7. https://www.learncpp.com/cpp-tutorial/cpp-faq/
8. https://www.learncpp.com/cpp-tutorial/void-functions-non-value-returning-functions/
9. https://gravatar.com/
10. https://www.learncpp.com/cpp-tutorial/chapter-12-summary-and-quiz/#comment-610608
11. https://www.learncpp.com/cpp-tutorial/chapter-12-summary-and-quiz/#comment-607681
12. https://www.learncpp.com/cpp-tutorial/chapter-12-summary-and-quiz/#comment-607703
13. https://www.learncpp.com/cpp-tutorial/chapter-12-summary-and-quiz/#comment-607810
14. https://g.ezoic.net/privacy/learncpp.com