12.x — Chapter 12 summary and quiz

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) 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)

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

Show Solution

b)

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

Show Solution

c)

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

Show Solution

d)

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

Show Solution

e)

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

Show Solution

Question #2

What is the output of this program?

```
#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

Question #3

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

Show Solution

Question #4

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

Show Solution

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.

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

```
#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;
}
```

Show Solution