7.x — Chapter 7 summary and quiz

ALEX DECEMBER 28, 2023

Chapter Review

We covered a lot of material in this chapter. Good job, you're doing great!

A **compound statement** or **block** is a group of zero or more statements that is treated by the compiler as if it were a single statement. Blocks begin with a `{` symbol, end with a `}` symbol, with the statements to be executed placed in between. Blocks can be used anywhere a single statement is allowed. No semicolon is needed at the end of a block. Blocks are often used in conjunction with `if statements` to execute multiple statements.

User-defined namespaces are namespaces that are defined by you for your own declarations. Namespaces provided by C++ (such as the `global namespace`) or by libraries (such as `namespace std`) are not considered user-defined namespaces.

You can access a declaration in a namespace via the **scope resolution operator (::)**. The scope resolution operator tells the compiler that the identifier specified by the right-hand operand should be looked for in the scope of the left-hand operand. If no left-hand operand is provided, the global namespace is assumed.

Local variables are variables defined within a function (including function parameters). Local variables have **block scope**, meaning they are in-scope from their point of definition to the end of the block they are defined within. Local variables have **automatic storage duration**, meaning they are created at the point of definition and destroyed at the end of the block they are defined in.

A name declared in a nested block can **shadow** or **name hide** an identically named variable in an outer block. This should be avoided.

Global variables are variables defined outside of a function. Global variables have **file scope**, which means they are visible from the point of declaration until the end of the file in which they are declared. Global variables have **static duration**, which means they are created when the program starts, and destroyed when it ends. Avoid dynamic initialization of static variables whenever possible.

An identifier's **linkage** determines whether other declarations of that name refer to the same object or not. Local variables have no linkage. Identifiers with **internal linkage** can be seen and used within a single file, but are not accessible from other files. Identifiers with **external linkage** can be seen and used both from the file in which they are defined, and from other code files (via a forward declaration).

Avoid non-const global variables whenever possible. Const globals are generally seen as acceptable. Use **inline variables** for global constants if your compiler is C++17 capable.

Local variables can be given static duration via the **static** keyword.

A **qualified name** is a name that includes an associated scope (e.g. `std::string`). An **unqualified name** is a name that does not include a scoping qualifier (e.g. `string`).

Using statements (including **using declarations** and **using directives**) can be used to avoid having to qualify identifiers with an explicit namespace. A **using declaration** allows us to use an unqualified name (with no scope) as an alias for a qualified name. A **using directive** imports all of the identifiers from a namespace into the scope of the using directive. Both of these should generally be avoided.

Inline functions were originally designed as a way to request that the compiler replace your function call with inline expansion of the function code. You should not need to use the inline keyword for this purpose because the compiler will generally determine this for you. In modern C++, the `inline` keyword is used to exempt a function from the one-definition rule, allowing its definition to be imported into multiple code files. Inline functions are typically defined in header files so they can be #included into any code files that needs them.

A **constexpr function** is a function whose return value may be computed at compile-time. To make a function a constexpr function, we simply use the `constexpr` keyword in front of the return type. A constexpr function that is eligible for compile-time evaluation must be evaluated at compile-time if the return value is used in a context that requires a constexpr value. Otherwise, the compiler is free to evaluate the function at either compile-time or runtime.

C++20 introduces the keyword `consteval`, which is used to indicate that a function must evaluate at compile-time, otherwise a compile error will result. Such functions are called **immediate functions**.

Finally, C++ supports **unnamed namespaces**, which implicitly treat all contents of the namespace as if it had internal linkage. C++ also supports **inline namespaces**, which provide some primitive versioning capabilities for namespaces.

Quiz time

Question #1

Fix the following program:

```
#include <iostream>

int main()
{
    std::cout << "Enter a positive number: ";
    int num{};
    std::cin >> num;


    if (num < 0)
        std::cout << "Negative number entered.  Making positive.\n";
        num = -num;

    std::cout << "You entered: " << num;

    return 0;
}
```

Show Solution (javascript:void(0))

Question #2

Write a file named constants.h that makes the following program run. If your compiler is C++17 capable, use inline constexpr variables. Otherwise, use normal constexpr variables. `max_class_size` should be `35`.

main.cpp:

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

int main()
{
    std::cout << "How many students are in your class? ";
    int students{};
    std::cin >> students;


    if (students > constants::max_class_size)
        std::cout << "There are too many students in this class";
    else
        std::cout << "This class isn't too large";

    return 0;
}
```

Show Solution (javascript:void(0))

Question #3

Write a function `int accumulate(int x)`. This function should return the sum of all of the values of `x` that have been passed to this function.

Show Hint (javascript:void(0))

The following program should run and produce the output noted in comments:

```
#include <iostream>

int main()
{
    std::cout << accumulate(4) << '\n'; // prints 4
    std::cout << accumulate(3) << '\n'; // prints 7
    std::cout << accumulate(2) << '\n'; // prints 9
    std::cout << accumulate(1) << '\n'; // prints 10

    return 0;
}
```

Show Solution (javascript:void(0))

3b) Extra credit: What are two shortcomings of the `accumulate()` function above?

Show Solution (javascript:void(0))

Next lesson
8.1 Control flow introduction

Back to table of contents

Previous lesson
7.13 Unnamed and inline namespaces