7.4 — Introduction to global variables

In lesson 7.3 -- Local variables, we covered that local variables are variables defined inside a function body. Local variables have block scope (are only visible within the block they are declared in), and have automatic duration (they are created at the point of definition and destroyed when the block is exited).

In C++, variables can also be declared *outside* of a function. Such variables are called **global variables**.

Declaring global variables

By convention, global variables are declared at the top of a file, below the includes, in the global namespace. Here’s an example of a global variable being defined:

```
#include <iostream>

// Variables declared outside of a function are global variables
int g_x {}; // global variable g_x

void doSomething()
{
    // global variables can be seen and used everywhere in the file
    g_x = 3;
    std::cout << g_x << '\n';
}

int main()
{
    doSomething();
    std::cout << g_x << '\n';

    // global variables can be seen and used everywhere in the file
    g_x = 5;
    std::cout << g_x << '\n';

    return 0;
}
// g_x goes out of scope here
```

The above example prints:

```
3
3
5
```

The scope of global variables

Identifiers declared in the global namespace have **global namespace scope** (commonly called **global scope**, and sometimes informally called **file scope**), which means they are visible from the point of declaration until the end of the *file* in which they are declared.

Once declared, a global variable can be used anywhere in the file from that point onward! In the above example, global variable `g_x` is used in both functions `doSomething()` and `main()`.

Global variables can also be defined inside a user-defined namespace. Here is the same example as above, but `g_x` has been moved from the global scope into user-defined namespace `Foo`:

```
#include <iostream>

namespace Foo // Foo is defined in the global scope
{
    int g_x {}; // g_x is now inside the Foo namespace, but is still a global
variable
}

void doSomething()
{
    // global variables can be seen and used everywhere in the file
    Foo::g_x = 3;
    std::cout << Foo::g_x << '\n';
}

int main()
{
    doSomething();
    std::cout << Foo::g_x << '\n';

    // global variables can be seen and used everywhere in the file
    Foo::g_x = 5;
    std::cout << Foo::g_x << '\n';

    return 0;
}
```

Although the identifier `g_x` is now limited to the scope of `namespace Foo`, that name is still globally accessible (via `Foo::g_x`), and `g_x` is still a global variable.

Key insight

Variables declared inside a namespace are also global variables.

Global variables have static duration

Global variables are created when the program starts (before `main()` begins execution), and destroyed when it ends. This is called **static duration**. Variables with *static duration* are sometimes called **static variables**.

Naming global variables

By convention, some developers prefix non-const global variable identifiers with “g” or “g_” to indicate that they are global. This prefix serves several purposes:

- It helps avoid naming collisions with other identifiers in the global namespace.
- It helps prevent inadvertent name shadowing (we discuss this point further in lesson 7.5 -- Variable shadowing (name hiding)).
- It helps indicate that the prefixed variables persist beyond the scope of the function, and thus any changes we make to them will also persist.

Global variables defined inside a user-defined namespace often omit the prefix (since the first two points in the list above are not an issue in this case, and we can infer that a variable is a global when we see a prepended namespace name). However, it doesn’t hurt if you want to keep the prefix as a more visible reminder of the third point.

Best practice

Consider using a “g” or “g_” prefix when naming non-const global variables, to help differentiate them from local variables and function parameters.

Author’s note

We sometimes get feedback from readers asking whether prefixes such as `g_` are okay because they’ve been told that prefixes are a form of Hungarian notation and “Hungarian notation is bad”.

The objection to Hungarian notation comes mainly from the use of Hungarian notation to encode the *type* of the variable in the variable’s name. e.g. `nAge`, where `n` means `int`. That’s not that useful in modern C++.

However, using prefixes (typically `g`/`g_`, `s`/`s_`, and `m`/`m_`) to represent the *scope* or *duration* of a variable does add value, for the reasons noted in this section.

Global variable initialization

Unlike local variables, which are uninitialized by default, variables with static duration are zero-initialized by default.

Non-constant global variables can be optionally initialized:

```
int g_x;       // no explicit initializer (zero-initialized by default)
int g_y {};    // value initialized (resulting in zero-initialization)
int g_z { 1 }; // list initialized with specific value
```

Constant global variables

Just like local variables, global variables can be constant. As with all constants, constant global variables must be initialized.

```
#include <iostream>

const int g_x;     // error: constant variables must be initialized
constexpr int g_w; // error: constexpr variables must be initialized

const int g_y { 1 };     // const global variable g_y, initialized with a value
constexpr int g_z { 2 }; // constexpr global variable g_z, initialized with a value

void doSomething()
{
    // global variables can be seen and used everywhere in the file
    std::cout << g_y << '\n';
    std::cout << g_z << '\n';
}

int main()
{
    doSomething();

    // global variables can be seen and used everywhere in the file
    std::cout << g_y << '\n';
    std::cout << g_z << '\n';

    return 0;
}
// g_y and g_z goes out of scope here
```

Related content

We discuss global constants in more detail in lesson 7.9 -- Sharing global constants across multiple files (using inline variables).

A word of caution about (non-constant) global variables

New programmers are often tempted to use lots of global variables, because they can be used without having to explicitly pass them to every function that needs them. However, use of non-constant global variables should generally be avoided altogether! We'll discuss why in upcoming lesson 7.8 -- Why (non-const) global variables are evil.

Quick Summary

```
// Non-constant global variables
int g_x;                 // defines non-initialized global variable (zero initialized
by default)
int g_x {};              // defines explicitly value-initialized global variable
int g_x { 1 };           // defines explicitly initialized global variable

// Const global variables
const int g_y;           // error: const variables must be initialized
const int g_y { 2 };     // defines initialized global const

// Constexpr global variables
constexpr int g_y;       // error: constexpr variables must be initialized
constexpr int g_y { 3 }; // defines initialized global constexpr
```