7.9 — Sharing global constants across multiple files (using inline variables)

In some applications, certain symbolic constants may need to be used throughout your code (not just in one location). These can include physics or mathematical constants that don't change (e.g. pi or Avogadro's number), or application-specific "tuning" values (e.g. friction or gravity coefficients). Instead of redefining these constants in every file that needs them (a violation of the "Don't Repeat Yourself" rule), it's better to declare them once in a central location and use them wherever needed. That way, if you ever need to change them, you only need to change them in one place, and those changes can be propagated out.

This lesson discusses the most common ways to do this.

Global constants as internal variables

Prior to C++17, the following is the easiest and most common solution:

1. Create a header file to hold these constants
2. Inside this header file, define a namespace (discussed in lesson 7.2 -- User-defined namespaces and the scope resolution operator)
3. Add all your constants inside the namespace (make sure they're *constexpr*)
4. #include the header file wherever you need it

For example:

constants.h:

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

// define your own namespace to hold constants
namespace constants
{
    // constants have internal linkage by default
    constexpr double pi { 3.14159 };
    constexpr double avogadro { 6.0221413e23 };
    constexpr double myGravity { 9.2 }; // m/s^2 -- gravity is light on this planet
    // ... other related constants
}
#endif
```

Then use the scope resolution operator (::) with the namespace name to the left, and your variable name to the right in order to access your constants in .cpp files:

main.cpp:

```
#include "constants.h" // include a copy of each constant in this file

#include <iostream>

int main()
{
    std::cout << "Enter a radius: ";
    double radius{};
    std::cin >> radius;

    std::cout << "The circumference is: " << 2 * radius * constants::pi << '\n';

    return 0;
}
```

When this header gets #included into a .cpp file, each of these variables defined in the header will be copied into that code file at the point of inclusion. Because these variables live outside of a function, they're treated as global variables within the file they are included into, which is why you can use them anywhere in that file.

Because const globals have internal linkage, each .cpp file gets an independent version of the global variable that the linker can't see. In most cases, because these are const, the compiler will simply optimize the variables away.

As an aside…

The term "optimizing away" refers to any process where the compiler optimizes the performance of your program by removing things in a way that doesn't affect the output of your program. For example, lets say you have some const variable `x` that's initialized to value `4`. Wherever your code references variable `x`, the compiler can just replace `x` with `4` (since `x` is const, we know it won't ever change to a different value) and avoid having to create and initialize a variable altogether.

Global constants as external variables

The above method has a few potential downsides.

While this is simple (and fine for smaller programs), every time constants.h gets #included into a different code file, each of these variables is copied into the including code file. Therefore, if constants.h gets included into 20 different code files, each of these variables is duplicated 20 times. Header guards won't stop this from happening, as they only prevent a header from being included more than once into a single including file, not from being included one time into multiple different code files. This introduces two challenges:

1. Changing a single constant value would require recompiling every file that includes the constants header, which can lead to lengthy rebuild times for larger projects.
2. If the constants are large in size and can't be optimized away, this can use a lot of memory.

One way to avoid these problems is by turning these constants into external variables, since we can then have a single variable (initialized once) that is shared across all files. In this method, we'll define the constants in a .cpp file (to ensure the definitions only exist in one place), and put forward declarations in the header (which will be included by other files).

Author's note

We use const instead of constexpr in this method because constexpr variables can't be forward declared, even if they have external linkage. This is because the compiler needs to know the value of the variable at compile time, and a forward declaration does not provide this information.

constants.cpp:

```
#include "constants.h"

namespace constants
{
    // actual global variables
    extern const double pi { 3.14159 };
    extern const double avogadro { 6.0221413e23 };
    extern const double myGravity { 9.2 }; // m/s^2 -- gravity is light on this
planet
}
```

constants.h:

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

namespace constants
{
    // since the actual variables are inside a namespace, the forward declarations
need to be inside a namespace as well
    extern const double pi;
    extern const double avogadro;
    extern const double myGravity;
}

#endif
```

Use in the code file stays the same:

main.cpp:

```
#include "constants.h" // include all the forward declarations

#include <iostream>

int main()
{
    std::cout << "Enter a radius: ";
    double radius{};
    std::cin >> radius;

    std::cout << "The circumference is: " << 2 * radius * constants::pi << '\n';

    return 0;
}
```

Because global symbolic constants should be namespaced (to avoid naming conflicts with other identifiers in the global namespace), the use of a “g_” naming prefix is not necessary.

Now the symbolic constants will get instantiated only once (in `constants.cpp`) instead of in each code file where `constants.h` is #included, and all uses of these constants will be linked to the version instantiated in `constants.cpp`. Any changes made to `constants.cpp` will require recompiling only `constants.cpp`.

However, there are a couple of downsides to this method. First, these constants are now considered compile-time constants only within the file they are actually defined in (`constants.cpp`). In other files, the compiler will only see the forward declaration, which doesn’t define a constant value (and must be resolved by the linker). This means in other files, these are treated as runtime constant values, not compile-time constants. Thus outside of `constants.cpp`, these variables can’t be used anywhere that requires a compile-time constant. Second, because compile-time constants can typically be optimized more than runtime constants, the compiler may not be able to optimize these as much.

Key insight

In order for variables to be usable in compile-time contexts, such as array sizes, the compiler has to see the variable’s definition (not just a forward declaration).

Because the compiler compiles each source file individually, it can only see variable definitions that appear in the source file being compiled (which includes any included headers). For example, variable definitions in `constants.cpp` are not visible when the compiler compiles `main.cpp`. For this reason, `constexpr` variables cannot be separated into header and source file, they have to be defined in the header file.

Given the above downsides, prefer defining your constants in a header file (either per the prior section, or per the next section). If you find that the values for your constants are changing a lot (e.g. because you are tuning the program) and this is leading to long

compilation times, you can move just the offending constants into a .cpp file as needed.

Global constants as inline variables C++17

In lesson 5.7 -- Inline functions and variables, we covered inline variables, which are variables that can have more than one definition, so long as those definitions are identical. By making our constexpr variables inline, we can define them in a header file and then #include them into any .cpp file that requires them. This avoids both ODR violations and the downside of duplicated variables.

A reminder

Constexpr functions are implicitly inline, but constexpr variables are not implicitly inline.

constants.h:

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

// define your own namespace to hold constants
namespace constants
{
    inline constexpr double pi { 3.14159 }; // note: now inline constexpr
    inline constexpr double avogadro { 6.0221413e23 };
    inline constexpr double myGravity { 9.2 }; // m/s^2 -- gravity is light on this
planet
    // ... other related constants
}
#endif
```

main.cpp:

```
#include "constants.h"

#include <iostream>

int main()
{
    std::cout << "Enter a radius: ";
    double radius{};
    std::cin >> radius;

    std::cout << "The circumference is: " << 2 * radius * constants::pi << '\n';

    return 0;
}
```

We can include `constants.h` into as many code files as we want, but these variables will only be instantiated once and shared across all code files.

This method does retain the downside of requiring every file that includes the constants header be recompiled if any constant value is changed.

Best practice

If you need global constants and your compiler is C++17 capable, prefer defining inline constexpr global variables in a header file.

A reminder

Use `std::string_view` for `constexpr` strings. We cover this in lesson 5.10 -- Introduction to std::string_view.