6.8 — Global constants and inline variables

BY ALEX ON JANUARY 3RD, 2020 | LAST MODIFIED BY NASCARDRIVER ON JANUARY 7TH, 2020

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

There are multiple ways to facilitate this within C++. Pre-C++17, the following is probably the easiest and most common:

1) Create a header file to hold these constants
2) Inside this header file, define a namespace (discussed in lesson **6.2 -- User-defined namespaces**)
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 my_gravity { 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

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

    std::cout << "The circumference is: " << 2 * radius * constants::pi;
}
```

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.

constants.cpp:

```
namespace constants
{
    // actual global variables
    extern const double pi { 3.14159 };
    extern const double avogadro { 6.0221413e23 };
    extern const double my_gravity { 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
    extern const double pi;
    extern const double avogadro;
    extern const double my_gravity;
}

#endif
```

Use in the code file stays the same:

main.cpp:

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

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

    std::cout << "The circumference is: " << 2 * radius * constants::pi;
}
```

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 once every time `constants.h` is #included, and the other uses will simply refer to the version 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`), not anywhere else they are used. This means that outside of `constants.cpp`, they can't be used anywhere that requires a compile-time constant. Second, the compiler may not be able to optimize these as much.

Given the above downsides, prefer defining your constants in the header file. If you find that for some reason those constants are causing trouble, you can move some or all of them into a .cpp file as needed.

Global constants as inline variables

C++17 introduced a new concept called `inline variables`. In C++, the term `inline` has evolved to mean "multiple definitions are allowed". Thus, an **inline variable** is one that is allowed to be defined in multiple files without violating the one definition rule. Inline global variables have external linkage by default.

Inline variables have two primary restrictions that must be obeyed:
1) All definitions of the inline variable must be identical (otherwise, undefined behavior will result).
2) The inline variable definition (not a forward declaration) must be present in any file that uses the variable.

The compiler will consolidate all inline definitions into a single variable definition. This allows us to define variables in a header file and have them treated as if there was only one definition in a .cpp file somewhere. These variables also retain their constexpr-ness in all files in which they are included.

With this, we can go back to defining our globals in a header file without the downside of duplicated variables:

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 my_gravity { 9.2 }; // m/s^2 -- gravity is light on this planet
    // ... other related constants
}
#endif
```

main.cpp:

```
#include "constants.h"

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

    std::cout << "The circumference is: " << 2 * radius * constants::pi;
}
```

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.

Best practice

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

6.9 -- Why global variables are evil

Index

6.7 -- External linkage

C++ TUTORIAL | PRINT THIS POST

8 comments to 6.8 — Global constants and inline variables

Sirdavos
January 13, 2020 at 1:25 pm · Reply

if we update header file(constants.h) in Global constants as inline variables, do we need to compile all the files which included this header file ?

nascardriver
January 14, 2020 at 12:55 am · Reply

Yes, all files that include a header have to be recompiled after you modify the header.

Sirdavos
January 16, 2020 at 3:50 am · Reply

so what is the advantage to use inline variables compare to "Global constants as internal variables" usage ?

nascardriver
January 16, 2020 at 4:02 am · Reply

If you have a non-inline constant in a header and you include that header 100 times, you get 100 copies of that variable.
With inline variables, you have only 1 variable, no matter how often the header gets included.

Fan
January 12, 2020 at 2:39 pm · Reply

Can we inline non-const variables?
Suppose getnum() is a function that asks the user for input and returns an int, and we have

```
1 inline int a{getnum()};
```

#include'd in multiple files. Does it ask for user input multiple times?

nascardriver
January 13, 2020 at 2:02 am · Reply

First off, avoid calling functions during construction of global variables. The function you're calling might depend on some other global object, which hasn't been initialized yet.

You can inline non-const variables, but they'll all have the same value, because there is only 1 variable (despite there being multiple definitions).

Prasad
January 5, 2020 at 8:47 pm · Reply

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 my_gravity;
}

#endif
```

should the namespace match the name mentioned in constants.cpp file, Its capitalized in .h file and not in cpp file.

nascardriver
January 7, 2020 at 4:25 am · Reply

Yep, they have to be an exact match. Lesson updated, thanks!