6.4 — Introduction to global variables

BY ALEX ON JUNE 19TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 8TH, 2020

In lesson **6.3 -- Local variables**, we covered that local variables are variables defined inside a function (or function parameters). 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 and naming global variables

By convention, global variables are declared at the top of a file, below the includes, but above any code. 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
```

By convention, many developers prefix global variable identifiers with “g” or “g_” to indicate that they are global.

Best practice

Consider using a g” or “g_” prefix for global variables to help differentiate them from local variables.

Global variables have file scope and static duration

Global variables have **file scope** (also informally called **global scope** or **global namespace 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()`.

Because they are defined outside of a function, global variables are considered to be part of the global namespace (hence the term “global namespace scope”).

Global variables are created when the program starts, and destroyed when it ends. This is called **static duration**. Variables with *static duration* are sometimes called **static variables**.

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

Global variable initialization

Non-constant global variables can be optionally initialized:

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

Constant global variables

Just like local variables, global variables can be 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: constant 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
```

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 a couple of lessons.

Quick Summary

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

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

6.5 -- Variable shadowing (name hiding)

Index

6.3 -- Local variables

C++ TUTORIAL | PRINT THIS POST

252 comments to 6.4 — Introduction to global variables

« Older Comments 1 2 3 4

kavin
January 7, 2020 at 3:57 am · Reply

In "introduction-to-global-variables" Unlike local const/constexpr variables where const need not be initialized but constexpr need to be initialized, could you explain when we should use global const and global constexpr since both need to be initialized ?

Alex
January 8, 2020 at 12:09 pm · Reply

Constant variables (whether const or constexpr) must be initialized, otherwise there is no way to give them a value.

You should favor constexpr whenever possible.

koe
December 21, 2019 at 6:23 pm · Reply

The beginning paragraphs are confusing, since it seems the words 'declaration' and 'definition' are used too loosely.

" A variable’s linkage determines whether multiple instances of an identifier refer to the same variable or not."

This seems problematic since in section 2.7 we learn that a definition instantiates a variable. I would think this makes more sense (even this may not be correct since it uses the word 'refer', which may conflict with proper meaning of 'reference' in C++):

"A variable's linkage determines whether appearances of its identifier in other files refers to it."

Also, this note should be moved higher, since it's confusing from the beginning of this section which declarations are definitions and which are pure declarations.

"Note that if you want to define an uninitialized non-const global variable, do not use the extern keyword, otherwise C++ will think you’re trying to make a forward declaration for the variable."

My understanding is that global variables with no const, constexpr, or intern or extern, are extern (default). If it is intended to be an uninitialized definition, then it must be a declaration of the variable that does not have 'extern', and all other declarations of it must have 'extern'. Is this a rule or recommendation?

Since const/constexpr must be initialized, the definition/declaration distinctions between intern and extern linkages are more clear.

Also, here the word 'declared' seems improper, since forward declarations only make sense for definitions.

"Similarly, in order to use an external global variable that has been declared in another file, you must use a variable forward declaration."

koe
December 21, 2019 at 7:48 pm · Reply

Note: my compiler errors out with [-Werror, -Wextern-initializer] from

```
extern int g_x{1};
```

Update: after getting through this section I think this is a better definition of linkage
"A variable's linkage determines whether the linker will associate appearances of its identifier in other files to that variable."

nascardriver
December 22, 2019 at 3:39 am · Reply

Old: "A variable's linkage determines whether multiple instances of an identifier refer to the same variable or not."
New: "A variable's linkage determines whether or not the variable can be used in a file other than the one it is defined in."

Old: "in order to use an external global variable that has been declared in another file"
New: "in order to use an external global variable that has been defined in another file"

> Also, this note should be moved higher
I added comments to the globals up to that point, stating whether they are definitions or declarations.

> Note: my compiler errors out with
I noted this after the first example that uses `extern`.

> My understanding is that global variables with no const, constexpr, or intern or extern, are extern (default)
Correct (I'm assuming you meant `static` when you said intern).

> If it is intended to be an uninitialized definition, then it must be a declaration of the variable that does not have 'extern'
Every definition is a declaration.

```
int i;
```

Saying that this is a declaration is correct, but very confusing. It's a definition (And therefor a declaration).

> and all other declarations of it must have 'extern'.
Correct. Without `extern`, they'd be definitions, violating the one-definition-rule.

> Is this a rule or recommendation?
That's just how it is, you can't do it otherwise.

Thank you very much for your feedback, it will help future readers to understand the topic better. If there's anything else or you don't agree with one of my updates, please let me know!

koe
December 23, 2019 at 1:57 am · Reply

> Saying that this is a declaration is correct, but very confusing. It's a definition (And therefor a declaration).

You're right, I should have just said "For uninitialized variables that are externally linked by default, their definitions cannot have 'extern' since this is the only way to distinguish them from their forward declarations which must have 'extern'."

Charan
December 14, 2019 at 1:57 am · Reply

How can we forward declare a constexpr variable in an external file? I think the chapter says we could not. However there is a provision to initialise a constexpr variable with external linkage ,in a file. What is the purpose of this i.e to initialise with external linkage?

Charan
December 14, 2019 at 2:01 am · Reply

Also, the final summary mentioned that constexpr must be initialised. But isn't it against the philosophy of constexpr ,to be initialised(and stay constant) by the user.?

nascardriver
December 14, 2019 at 4:32 am · Reply

`constexpr` variables cannot be forward declared. They have to be initialized at their declaration.
`extern constexpr` doesn't have any benefits code-wise. The linker will expose the name of the variable, which makes it easily accessible to other programs and libraries. There's probably a niche case where this is useful.

`constexpr` is used for compile-time constants. They can't be initialized with a user-provided value.

koe
December 21, 2019 at 6:40 pm · Reply

How would other programs and libraries use an 'extern constexpr' variable if it can't be forward declared? Why can you forward define an 'extern const' variable even when its definition is a compile-time constant, but not 'extern constexpr'?

nascardriver
December 22, 2019 at 2:08 am · Reply

> How would other programs and libraries use an 'extern constexpr' variable if it can't be forward declared?
`dlsym` on unix, `GetProcAddress` in Windows. The other program can't declare the variable `constexpr`, but they can get the value of our `constexpr` variable.

> Why can you forward define an 'extern const' variable even when its definition is a compile-time constant, but not 'extern constexpr'?
The value of a `const` doesn't matter, because it's allowed to be a run-time or compile-time value. `constexpr` has to be initialized with a compile-time value. `constexpr` variables can be used for computations at compile-time. I suppose allowing separation of declaration and definition would make the compiler's job more difficult.

mooseymouse
December 7, 2019 at 8:50 pm · Reply

hi i dont unerstand something how can the compiler differentiate between a forward declaration of an external variable in an other file from defining a local variable like what if i wanted to define an uninitialised external variable in main you said it wouldn't link as the linker interprets it as a forward declaration for variable in another file. i am confused by this thanks,

Alex
December 10, 2019 at 10:45 pm · Reply

A variable forward declaration and a local variable have different syntaxes.

If you want an uninitialized external global variable, use "int g_x", which is external by default (not "extern int g_x", which will be treated as a forward declaration).

Chandler
December 3, 2019 at 8:25 pm · Reply

All of your examples on this page initialize the variables using parenthesis.

```
// Variables declared outside of a function are global variables
int g_x; // global variable g_x
const int g_y(2); // global variable g_y

static const double g_gravity(9.8);
```

I thought this was done with the curly braces: { }

nascardriver
December 4, 2019 at 4:16 am · Reply

Some lessons still use direct- or copy-initialization, thanks for pointing this one out! I updated it to use brace-initialization instead.

Tushar
November 1, 2019 at 9:32 am · Reply

https://www.youtube.com/watch?v=hDfDnsvEodQ

This video explains this concept very clearly. Had a hard time understanding this so save yourself and go through this tutorial

Fan
October 20, 2019 at 7:40 am · Reply

Is it true that the order of initialization of global variables is undefined, even inside a single file? For example, the code

```
#include <iostream>

extern int j;        //forward declaration of j
extern int i=j+1;
extern int j=i+2;

int main()
{
    std::cout<<i<<' '<<j;

    return 0;
}
```

produces
1 3
as if j=0 first, then i is initialized with 0+1=1, and finally j is initialized with 1+2=3.
But then the code

```
#include <iostream>

extern int j;        //forward declaration of j
extern int i=j+1;
extern int j=2;

int main()
{
    std::cout<<i<<' '<<j;

    return 0;
}
```

produces
3 2
as if j is initialized with 2 first, and then i is initialized with 2+1=3.

nascardriver
October 21, 2019 at 4:49 am · Reply

There are several initialization steps for non-local variables. First, variables that are initialized by a constant expression (eg. a number) are initialized. The rest of the variables is zero-initialized. This happens before your program starts running. The values are stored in the binary. Then, when your program starts running, the other variables in the current file are initialized in-order (This might also happen before your program starts running, but that's just an optimization).

Example 1:
\`j\` is zero-initialized, because it isn't initialized by a constant expression.
\`i\` is zero-initialized, same reason as \`j\`.
Now your program starts.
\`i\` is initialized (0 + 1)
\`j\` is initialized (1 + 2)

Example 2:
\`j\` is initialized to 2, because 2 is a constant expression.
Now your program starts
\`i\` is initialized (2 + 1)

This order is well-defined.

To take it a little further

```
#include <iostream>

extern double j;

double i{ ++j };
double k{ 9 };
double j{ k + 3 };

int main()
{
    std::cout << i << ' ' << j << ' ' << k << '\n';

    return 0;
}
```

Output: 1 12 9

\`k\` is initialized to 9, because 9 is a constant expression.
\`i\` and \`j\` are 0.
Now the program starts.
\`i\` is initialized (0 + 1)
\`j\` is initialized (9 + 3)

There are exceptions, but none you'd have to worry about just yet.

hellmet
October 13, 2019 at 11:53 am · Reply

I have a bunch of questions, some stupid perhaps...
1) Why is it that a const has the exception that it has only internal linkage even when it is declared at the file scope (i.e. outside main)? (The explanation on StackOverflow didn't make sense to me)
2) How would constexpr change things with each of the declarations shown in the summary? From what I recall (and I hope I understood this well) const is for values that do not change at runtime, once initialized; constexpr need to be initialized by compile time, so the compiler is free to replace all occurrences of the variable with the value it holds. So, does this mean whenever and wherever I include the header with something defined as constexpr, the compiler will replace them with that variable's values?
3) Clearly, static and extern are opposing concepts. Is there any way I can include my 'secret' variable only select files and not in any other? Say I have a case where I am keeping track of number of students in a file, and to calculate the average expenditure per student, I want to get the number of students. This is a case where the finance.cpp and students.cpp are different, but need access to a single 'secret' variable. From what I've read so far, accomplishing that with static doesn't seem possible.

nascardriver
October 14, 2019 at 4:50 am · Reply

1)
It's likely that you need a `const` variable in multiple files. If you need something in multiple files, you put it in a header. A constant never changes, so you initialize it at its declaration. If you initialize a variable with external linkage at its declaration in a header, you'll run into trouble, because it's defined multiple times. If the variable has internal linkage, you can include it multiple times, because the other files don't see it.

2)

```
// Uninitialized definition:
constexpr int g_x;  // not allowed: const variables must be initialized

// Forward declaration via extern keyword:
extern constexpr int g_z; // not allowed: constexpr variables must be initialized at

// Initialized definition:
constexpr int g_y{ 1 };  // defines initialized constexpr variable (internal linkage)

// Initialized definition w/extern keyword:
extern constexpr int g_w{ 1 }; // Legal, but doesn't make sense, as g_z is illegal.
```

`g_w` makes me wonder why it's legal, maybe someone else can help.

`constexpr` tells the compiler that the variable/function can be evaluated at compile-time. What the compiler does with this information is up to it. It might replace the occurrences of the variable with its value, but it doesn't have to.
`constexpr` variables can be used in combination with `constexpr` functions to make sure that your code _could_ be evaluated at compile-time.

3)
`class` and `friend` (Covered later) will help you with keeping your secret. C++20's modules might also help, the future will tell.

hellmet
October 16, 2019 at 7:00 am · Reply

Okay 3 makes sense!
But I'm still worried I don't fully understand 1, 2.

I'm trying to rationalize the meanings of const, constexpr, extern, internal vs external linkage, static and connect the dots so that I better understand and reason with their usage.

This is how I understood them, please correct me if I'm wrong.

1) A variable of the form 'int p' declared at the file level has external scope by default. Its forward declared as 'extern int p' in a header file that can be included in files that require it. This assures that any file #include-ing the relevant header file has the same copy.

2) const, static give a variable internal linkage by default and can be declared+initialized as extern and forward declared in a header file if they are to be used outside the file. Consequently, any files including the header file will get the 'same copy' instead of being assigned new memory each time.

3) The advantage (over C) of 'const' having internal linkage is that it allows for both possibilities, having a local copy (by default) or by using the global unique copy allowing for customization of code.

4) All variables or functions declared as 'constexpr' has a single memory by default, so it's redundant to use extern on its declaration. They can just be used as is, as long they are defined+declared in a header file and included, as they can't be forward declared

Have I got this right?

Sidenote, how do I find all my comments? It would be nice to have the links so that I can refer back to them later

nascardriver
October 16, 2019 at 7:28 am · Reply

1)
Correct

2)
`static` and `extern` can't be mixed. `static` has internal linkage.

3)
"global unique" might be misleading. Every file that includes a header in which a `const` variable was declared gets a copy of the variable. All of the variables will have the same value, but they're distinct objects.
This is not the case for `constexpr` variables. They're the same in every file.

4)
Right. `const` variables can also be made to have a single memory location by declaring them `inline`. All `constexpr` variables are `inline` by default, that's why they do this.

> how do I find all my comments?
There's no friendly interface. You can use the wordpress api and run the result through a json beautifier to get a somewhat readable overview (https://jsonbeautifier.org/?url=https%3A%2F%2Fwww.learncpp.com%2Fwp-json%2Fwp%2Fv2%2Fcomments%3Fper_page%3D100%26search%3Dhellmet) (click on the little arrows on the right the expand the comments).

hellmet
October 16, 2019 at 8:14 am · Reply

Thank you very much! That makes so much stuff clear!
I really appreciate your patience and dedication in maintaining this site!
I'll comeback someday, after I get a job and express my gratitude!

murat yilmaz
August 12, 2019 at 1:52 am · Reply

In find this topic hard to understand.

What is the difference between scope and linkage?

What is the difference between global variables and external variables? (They have automatically the extern keyword)

What is the difference between intern variables and global variables or functions with the keyword "static"?

Why would you want to make variables only visible to that specific file by placing the static keyword?

potterman28wxcv
August 13, 2019 at 8:08 am · Reply

> What is the difference between scope and linkage?

The scope defines where you can use a given variable in the C++ code.

Linkage is the process of linking several object (binary) files into one executable. In this process, each object file has a number of symbols that are defined - if you create a constant in your C++ code, your constant will have a symbol in the object file. However, if your constant is static, then that symbol will not be able to be used by the other object files - the "scope" of the symbol will be local to the object file corresponding to your C++ source file.

I'm not sure how exactly does a symbol stay local to an object file. But the result is that the other object files won't be able to see your symbol. So if you use your constant (that was defined static) in another C++ source file, the corresponding object will use it as well, but when the linker tries to form one executable, it won't be able to link the symbol to its location.

I don't know if it makes it more clear - you might want to re-read the chapter about the compilation process.

> What is the difference between intern variables and global variables or functions with the keyword "static"?

Global variables and functions (with or without the static keyword), will have a symbol attached to them by the compiler, in the object file. If you compile with the -S option (which stops at the assembly), and then read the generated assembly, you will see that your static identifiers have all been translated into a symbol (something that ends with a ':').

On the other hand, intern variables won't have any symbol attached to them. And the compiler will decide whether the intern variable will reside in a processor register, or somewhere on the stack (in memory), or a combination of both. By the way, it is much more efficient to manipulate registers than it is to manipulate memory, so intern variables will always be more efficient than global variables.

> Why would you want to make variables only visible to that specific file by placing the static keyword?

It's a bit like a principle of encapsulation. You want to know where this or that variable is used. If there is no static keyword, then that variable could be used anywhere in your code - and if a bug happens related to that specific variable, then it will potentially be a lot harder for you to pinpoint where something went wrong.

In the examples of these tutorials it's always easy because you never have more than 100 lines of code. On real life projects (even those that you do by yourself), your codebase can reach easily 1000+ lines of code, sometimes even 10000. Good luck trying to find what went wrong if you only use global variables - moreover if none of these global variables have the "static" keyword.

Global variables have their uses - but you should avoid them whenever you can. Because it is much simpler to reason in terms of local variables that reside in 40-50 lines of code, rather than having to deal with variables shared by all the program, and that could be modified anywhere.

I'm not sure about the 2nd question.

Alex
August 13, 2019 at 11:44 am · Reply

> What is the difference between global variables and external variables? (They have automatically the extern keyword)

Nothing, they are the same thing.

Abhinav
July 9, 2019 at 8:12 pm · Reply

I have a question regarding linkage and duration with respect to your example above. Global variables have static duration right? So the variables inside constants.cpp are destroyed when we leave that program right? So how come if I #include constants.h in my main.cpp file, I can access constants defined in constants.cpp, even though constants.cpp has ended executing (Or its not?) ?

edit: Is my following understanding correct:
1)global variables (file scope) have static duration, i.e
destroyed when file finishes execution.
2)extern variables (true global scope) do not have static
duration but persist even after file execution completes.

nascardriver
July 10, 2019 at 4:43 am · Reply

Files don't execute, files only matter to your compiler, but not a runtime. Static variables die when the program ends.

> global variables (file scope) have static duration
Yes

> destroyed when file finishes execution
No

> extern variables (true global scope) do not have static duration
They have static duration

> persist even after file execution completes
There is no "file execution", they die when the program ends.

Vir1oN
June 14, 2019 at 6:40 am · Reply

Hey @Alex
I've just read the chapter for several times, and one thing remains unclear: what happens with

external variables' scope, when they are exported to another file? You wrote:

"Global variables have global scope (aka. file scope), which means they can be accessed from the point of declaration to the end of the file in which they are declared."

As well as:

"Global variables have global scope, and can be used anywhere in the program"

Thus I make the conclusion that scope must be renewed or something, or how the variable can be accessed in the other files otherwise?

Also it's still a bit hard for me to understand the difference between the terms of "linkage" and "scope". In the quiz solution you wrote:

"Scope determines where a variable is accessible"
"Linkage determines whether the variable can be exported to another file or not"

But couldn't the second sentence be rephrased as " linkage determines whether the variable can be accessed from another file or not"? If so, linkage seems to be scope in the broader sense. Or I just don't understand it correctly?

Alex
June 19, 2019 at 11:49 am · Reply

When a global variable is defined, in the file for which it is defined, it can then be accessed from the point of definition to the end of the file.
However, via a forward declaration, the variable can be used from other files (assuming external linkage), from the point of the forward declaration to the end of the file containing the forward declaration. The compiler doesn't know this is the same variable as in another file.

Scope works file by file and is enforced by the compiler.
Linkage works across files, and is enforced by the linker.
Thus, in order to be accessed from another file, the variable must have external linkage (to satisfy the linker) AND it must be in scope (to satisfy the compiler).

Vir1oN
June 19, 2019 at 12:26 pm · Reply

Thank you for your comprehensive answer, seems like everything fell into place in my head

Sweta
June 5, 2019 at 11:11 am · Reply

It was my understanding from one of the previous topics that constexpr must be used for constants that are initialised during compile-time, and const must be used for constants that are initialised during run-time, yet the above example uses "const" for compile-time initialisation. Is there a particular reason for this?

Alex
June 12, 2019 at 12:26 pm · Reply

Const can be used for either runtime or compile-time constants.

It's more precise to use constexpr if you can, but it's not necessary to update older code/examples that already work unless there's a specific reason to.

Louis Cloete
April 29, 2019 at 1:19 pm · Reply

@Alex, since you suggest that one namespace non-const global variables, maybe you should explain how to forward declare a namespaced global variable. It has not been obvious to me when I wanted to use a global Mersenne twister over multiple files for the game in chapter 11.x's quiz. I found out by trial and error after a lot of different errors about duplicate definition and undefined symbols that it is like this:

foo.cpp:

```
namespace MyNamespace
{
    int g_maxPoints { 100 }; // define the global object
}
```

bar.cpp:

```
namespace MyNamespace
{
    extern int g_maxPoints; // forward declare the global object to use it later in the file
}
```

Alex
May 6, 2019 at 12:12 pm · Reply

Thanks for the feedback. There's an example in this very lesson already on how to do this -- perhaps it wasn't obvious enough. I've updated the comments in the example to make it more explicit that the forward declarations need to go in a namespace.

Louis Cloete
May 7, 2019 at 3:52 am · Reply

Indeed! I missed it. Anyway, it is better to be more explicit for idiots like me ;)

Alireza
April 16, 2019 at 1:47 am · Reply

Hi and greeting,
I'm using Qt Creator 4.7.
I have written this code:

```
// mathematical.h
namespace constants
{
    extern const double g_pi ;
    extern const double g_gravity ;
}                                   // there's no error
```

however in mathematical.cpp file:

```
// mathematical.cpp
namespace constants
{
    extern const double g_pi { 3.14'15'9 } ;
    extern const double g_gravity { 9.8'06'6 } ;
}
```

it gives me this warning for both variables @g_pi and g_gravity :

```
..\mathematical.cpp:10: warning: no previous extern declaration for non-static variable 'g_p
```

What do you suggest for it ?

nascardriver
April 16, 2019 at 4:33 am · Reply

Include mathematical.h in mathematical.cpp so it sees the declaration, or use inline variables.

```
// mathematical.hpp

namespace Constants
{
  inline constexpr double pi{ 3.14159 };
  inline constexpr double avogadro{ 6.0221413e23 };
  inline constexpr double my_gravity{ 9.2 };
}
```

By using inline variable you don't need another source file, because inline variables are allowed to be defined multiple times.

Alireza
April 24, 2019 at 4:03 am · Reply

thank you so much, it works

_weedfox
April 2, 2019 at 2:59 am · Reply

So if im working with only 1 file is it better to use static since it will be only used for this file anyway or let it be external since its easier and wont be used external anyway?

nascardriver
April 2, 2019 at 3:05 am · Reply

It doesn't matter. You can omit both static and extern.

« Older Comments 1 2 3 4