restrict, static & inline Keywords in C

— Seminararbeit —

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Markus Fasselt
E-Mail-Adresse:	1fasselt@informatik.uni-hamburg.de
Studiengang:	Informatik
Betreuer:	Konstantinos Chasapis

Hamburg, den March 27, 2014

Abstract

This paper describes the effects on performance and behavior of three programming keywords in the C programming language: restrict, static and inline.
Restrict is a type qualifier on pointer variables and gives a hint to the compiler. It says, a variable has no alias in the current scope. With this knowledge, the compiler can save some unnecessary instruction calls.
The static keyword has two different functions. The first one helps to encapsulate private functions to provide an access control. The second function aims to keep the state of a non-global variable between multiple invocations in an arbitrary scope.
The inline keyword can be used on any function and gives a hint to the compiler, that this function can be embedded where the function call occurs. By using this keyword, the compiler can save an extra function call, which can lead to better performance.

Contents

1 Introduction

For scientific studies it is often necessary to analyze huge amounts of data, which nowadays is done by computers. It is important to get the results in an appropriate time, which requires an efficiently running application. The C programming language has many possible optimizations, some of which are enabled automatically by the compiler, others by the application developer. One factor, the developer can make use of, is the usage of three programming keywords implemented in C, which are restrict, static and inline. This paper provides an overview of those.

2 The restrict-Keyword

This chapter explains the functionality and effects of the restrict-keyword in the C-language. The restrict keyword is included in the C-standard since C99 and is a type qualifier, which can be assigned to pointer variables. The keyword says, that there is no alias[1] for such a variable in the current scope.

2.1 Using the restrict keyword

The restrict-Keyword is a type qualifier on pointer variables. It can be added to any definition of a pointer variable or function parameter. The restrict keyword must be inserted after the asterisk and before the variable name. An example definition can be found in listing 2.1.

Listing 2.1: Example of using the restrict-Keyword

```
int *aPtr;
int *restrict aPtr;
```

It is important, that the restrict keyword is only used, when there is no alias for this variable. Otherwise, the usage of restrict can lead to an undefined behavior. An example of what is happening in such case, will be given in chapter 2.3.

[1]An alias of a variable is another variable, which points to the same address.

2.2 Functional Principle

With the knowledge that a variable is restricted and has no alias, the compiler doesn't need to take care of background changes due to an alias. This helps the compiler to add further optimization into the program.

Hereinafter, the effect of the restrict keyword will be explained on the example of the following `update()`-function. This function takes three integer pointers as parameters, named `a`, `b` and `c`.

Listing 2.2: `update`-Function

```c
void update(int *a, int *b, int *c)
{
    *a += *c;
    *b += *c;
}
```

This function adds the value of `c` to `a` and `b`. When you call this function, for example, with `a = 1, b = 2` and `c = 3` the new values of the variables are `a = 4, b = 5` and `c = 3`.

To understand the effect of the restrict keyword, it is necessary to take a closer look at the assembly level. The (simplified) assembly code of the `update`-function above can be seen in listing 2.3. It was generated by compiling the code with `gcc -Wall -std=c99 main.c -O1 -S`.

Listing 2.3: The assembler-code to listing 2.2

```
update:
    movl    (%rdx), %eax
    addl    %eax, (%rdi)
    movl    (%rdx), %eax
    addl    %eax, (%rsi)
    ret
```

On the assembly level, the three variable names `a`, `b` and `c` are no longer present. Their values are now stored in the CPU registers `%rdi`, `%rsi` and `%rdx`.
The initial values in the registers can be seen in table 2.1. Since all variables are pointers, their values are references to other locations in memory. The $*$val columns shows the actual value behind that pointer.
The additional `%rax` register buffers a temporary value and has no initial value. This buffer is needed, as the compiler can't directly execute an addition between two pointers.
To calculate the sum of `a` and `c`, the CPU has to perform two operations. First, it will copy the referenced value in `%rdx` into the register `%rax`. In the next step, the CPU will calculate the sum of `%rax` and `(%rdi)`[2]. Then the result will be written back into

[2]When a register is written within brackets, the pointer is dereferenced and its underlying value will

Variable	Register	Value	∗val
a	%rdi	0x123	1
b	%rsi	0x124	2
c	%rdx	0x125	3
-	%rax	-	-

Table 2.1: The initial values in the registers

the referenced location in memory.

To calculate the second addition, the CPU runs the same steps again, just with another variable: first it loads `(%rdx)` into `%eax` and then it calculates `%eax` plus `(%rsi)`. The CPU has to reload the variable `c` in the first step as the referenced value could have changed in the previous operation due to an alias. With the knowledge that there is no alias for `c`, the compiler can save this unnecessary call.
The following chapter explains, how the code will be optimized by using the restrict keyword.

2.2.1 Optimizing the code with restrict

To optimize the function in listing 2.2 by using the restrict keyword, it only needs a simple modification: the parameter `c` needs to be flagged as restricted. The altered function can be seen in listing 2.4.

Listing 2.4: `update`-Function with restrict

```
void update_restrict(int *a, int *b, int *restrict c)
{
    *a += *c;
    *b += *c;
}
```

That this simple modification brings the desired effect can be seen in the assembly code in listing 2.5: The second loading call (movl) has been removed as the compiler now knows, that the referenced value in `%rdx` could not have changed.

Listing 2.5: The assembler-code to listing 2.4

```
update_restrict:
    movl    (%rdx), %eax
    addl    %eax, (%rdi)
    addl    %eax, (%rsi)
    ret
```

be used

2.3 Assigning an aliased variable to a restricted variable

As the compiler removes some instruction calls when the restrict keyword is used, an incorrect usage can lead to undefined behavior and bugs, that are difficult to find. The compiler does not verify that a restricted variable has no alias, so it does not throw an error if the restrict keyword is used wrong.

The following example will demonstrate, what happens when the function in listing 2.4 is called with `a` as an alias for `c`.

Listing 2.6: `update`-Function

```c
int main(int argc, char** argv)
{
    int a = 1;
    int b = 2;

    update(&a, &b, &a);
    printf("Expected Result: %d %d\n", a, b);

    a = 1; b = 2; // reset values

    update_restrict(&a, &b, &a);
    printf("Actual Result:   %d %d\n", a, b);
}
```

In this example, there are just two variables, `a` and `b`. The address of `a` is passed both as first and last parameter. This has the effect, that within the update function, `a` is an alias of `c`.
The `update` function calculates in the first step `*a = *a + *a = 1 + 1`. The variable `a` has now the value 2. As the function works with pointers, the second addition uses the new value of `a` and calculates `*b = *a + *b = 2 + 2`. The variable `b` has now the value 4. So the expected result is that `a` has the value 2 and `b` has the value 4.

When the `update_restrict` function is called with these parameters, the behavior is different. As mentioned above, the compiler assumes that the value of `*c` has not changed due to the restrict keyword. However, the first calculation alters the value of `a`, which is also used in the second addition `b` (through the alias `c`).
After the first calculation, `a` has the same value as above, which is 2. But now the CPU does not reload the value of `c` and executes the second addition with 1 as value for `a`. So the actual value of `b` is now 3, but 4 was expected.

In this example, the compiler does not throw any error or give a warning. The developer has to find the origin of the wrong result by himself, which can be a nearly impossible thing in more complex situations. In conclusion, it is important to be careful, when using this keyword.

2.4 Performance

By comparing the assembly codes of the non-restrict and the restrict version of the code above, it is easy to see that in the restrict version there is one instruction call less than the non-restrict version. This let assume, that the optimization with restrict can improve the performance of an application. To verify this assumption, the following performance test has been made.

Listing 2.7: Performance-Test

```
void update(int *a, int *b, int *c)
{
    igned long long iterations = 100000000000;
    for(i = 0; i < iterations; i++)
    {
        *a += *c;
        *b += *c;
    }
}
```

The `update_performance_test` function in listing 2.7 runs the two additions 100 trillion times within a for-loop. To test the performance of the restrict version, the parameter `c` has been marked as restrict.
The test has been executed on the server cluster of the German Climate Computing Center in Hamburg.

The result of the performance test was, that the version without restrict took 249.086421s and with restrict 83.690911s. So in this simple example the restrict version is 2.98 times faster.

3 The static-Keyword

In C, the static keyword has two different functions. On the one hand, it provides an access control for functions and global variables. On the other hand, the keyword offers local variables which keep their state during multiple invocations. In this chapter both functionalities will be explained.

3.1 Static functions

On functionality of the static keyword can be applied on functions. To define a function as static, the static keyword has to be included in the signature of the function. It does not matter in which position the keyword is mentioned, it can be inserted before or after the return type and other optional keywords, but must be inserted before the function name. In listing 3.1 there is an example function, which is declared as static.

Listing 3.1: Definition of a global static variable

```
static void static_function();
```

A function defined as static can only be called by functions in the same translation unit[1]. This means in general that a static function can only be called from other functions in the same file. Static functions can not be called from functions in other files.

3.2 Static variables

In C there are two types of static variables which are quite different in their functionality. Their functionality depends not on different definitions, but on their location in the code. The static keyword on global variables, which scope is the current translation unit, has another effect than the static keyword on variables inside an arbitrary block.
As the definitions of static variables always look the same, an example can be found in listing 3.2.

Listing 3.2: Definition of a global static variable

```
static int a;
```

Just like the static keyword on static function, the keyword can be positioned everywhere before the variable name.

3.2.1 Global static variables

The static keyword has the same effect on global variables as on functions. It provides an access control to forbid an access from other translation units, as it is explained in chapter 3.1 for static functions.

3.2.2 Static variable inside an arbitrary block

In C a variable can only accessed in the same block (or in a nested block of it) where it was defined. In a subsequent block, previous variables are destroyed[2] and are no longer

[1]A translation unit is a single source file after it has been processed by the preprocessor. Among other things the preprocessor includes header files, evaluates #ifdef's and expands macros. Afterwards it can be compiled by the c compiler

[2]These variables are not really destroyed as their values can still exist in memory, but the variable can no longer be used to access its value.

accessible. A variable becomes initialized every time the processor enters its block to execute it. This has the effect, that a variable defined inside a loop will be (re-) initialized in each iteration and loses its previous value.

Listing 3.3: Definition of a global static variable

```c
for (int i = 0; i < 3; i++)
{
    int j = 0;
    printf("%d\n", j);
    j++
}
```

In listing 3.3 an example of the effect described above can be found. It consists of a `for`-loop, in which a variable `j` is defined with the initial value 0. At the end of the block, `j` will be increased by one. As the variable `j` will be reinitialized in every iteration, this code snippet will output three times a `0`.

To increment the variable `j` within each iteration, the variable can be defined as static. In this case, the variable is initialized only once at compile time and keeps its state during the whole program execution including multiple function invocations. The static variable now acts similar to a global variable, which keeps its state, but is only visible to the block in which it was defined.

However, there is one downside of using static variables: static variables are not thread safe. In case a applications needs to make use of threads, it might be better to avoid using static variables and to define them in a higher scope.

4 The inline-Keyword

Macros are well known in C. A lesser-known feature, which is similar to macros, is the inline keyword. It was included in the C-standard with C99 and will be explained in this chapter.

In many programming languages, it is quite common to cluster the program code into several functions. This improves the readability of the code and allows an easy reusage of code snippets. However, one downside of clustering code into multiple functions is, that each function call takes some extra time: the processor needs to allocate space on the stack, save the return address and put the parameters onto the stack.

To save the time of an extra function call, the compiler has the opportunity to substitute any function call with the called functions body. As it would be very inefficient to substitute every function call, the compiler only substitutes well-chosen functions. In most cases, this are small and often called functions. If the function is to big, it may be inefficient to substitute it, as there aren't enough registers in the CPU to handle all variables simultaneously.

By using the inline-keyword on a function, the developer gives a hint to the compiler, to substitute all invocation of this function. As the inline keyword is just a suggestion to the compiler, the compiler can accept or ignore this hint. Often the compiler looks for inlinable function itself and comes to similar conclusions as the developer.

In listing 4.1 the two functions `add` and `some_function` can be found. The `add` function is flagged with the inline keyword and will be called in `some_function`. If the compiler comes to the same conclusion and embeds the `add` function the resulting `some_function` function looks like listing 4.2.

Listing 4.1: `add` is a inlinable function

```c
inline static int add(int s1, int s2)
{
    return s1 + s2;
}

int some_function(int a, int b)
{
    return add(a, b);
}
```

Listing 4.2: The resulting `some_function`-function after substitution

```c
int some_function(int a, int b)
{
    return a + b;
}
```

This effect can also be seen on the assembly level. In listing 4.3 the corresponding assembly code to listing 4.1 can be found. After substituting the add function, the assembly code in listing 4.4 matches the the substituted C source code in listing 4.2.

Listing 4.3: Assembly code of listing 4.1

```
add:
    leal    (%rdi,%rsi), %eax
    ret
some_function:
    call    add_inline
    rep ret
```

Listing 4.4: Assembly code of listing 4.2

```
some_function:
    leal    (%rdi,%rsi), %eax
    ret
```

4.1 Limitations

There are some limitations when an function should be embedded. First, recursive functions can not be embedded. It is not possible to nest all possible function calls, as the depth of the stack trace depends on parameters and is unknown at compile time. As a second limitation, functions with external linkage should not use private variables, that are only visible in the current translation unit. These variables are no longer accessible, when the function is embedded in another translation unit. This may not be a problem with gcc, as the gcc compiler only substitutes functions, that are called in the same translation unit in which they were defined.

4.2 Forcing the compiler to inline a function

The compiler draws its own conclusions, which function calls are substituted and which not. It may ignore the developers advice completely. In most cases, the compiler finds the best solution, but sometimes the developer wants to decide whether a function should be embedded or not. Most compilers offer some extra keywords, that can be used to enforce a specific behavior.

The gcc compiler offers the attribute `__attribute__(( always_inline))` to force a substitution and `__attribute__((noinline))` to prevent a substitution. The inline keyword is useless in this case and will be ignored. An exapmle usage can be seen in listing 4.5.

Listing 4.5: Forcing a specific inline behavior with gcc

```
inline __attribute__((always_inline)) int
   ↪ inline_function(int a, int b);
inline __attribute__((noinline)) int no_inline_function(int
   ↪ a, int b);
```

4.3 Performance

As mentioned above, a separate function call takes additional time. This leads to the assumption, that function call substitution improves the performance of an application. To validate this assumption, the code in listing 4.6 has been used to do a performance test. The `add` function conforms to listing 4.1 and calculates the sum of two variables.

The following code was executed two times: at the first time the `add` function call was substituted, the second time it was not. To force the right behavior, the attributes explained in chapter 4.2 were used.

Listing 4.6: Performance-Test

```
unsigned long long iterations = 100000000000;
for(i = 0; i < iterations; i++)
{
    a = add(a, b);
}
```

The result of the performance test was, that the execution with a separate function call took 67.553960s. Executing the test with the `add` function inlined took only 22.290012s. In this simple example the version that embeds the function is 3.03 times faster.
This result can't be overgeneralized, since it is a very simple example. But it shows, that function substitution can have a large impact on the performance.

5 Summary

The examples in the chapters before have shown, that it can be very easy to improve the performance of an application by just using the right keywords. This paper has explained three of these, but the C programming language consists of many more ways to optimize the performance of an application. To develop efficient applications, it is important to have a wide understanding of all these features.

Bibliography

Acton 2006 ACTON, Mike: *Demystifying The Restrict Keyword.* May 2006. – URL `http://cellperformance.beyond3d.com/articles/2006/05/demystifying-the-restrict-keyword.html`. – Access date: 11/16/2013

Bendersky 2009 BENDERSKY, Eli: *What does "static" mean in a C program?* Februar 2009. – URL `http://stackoverflow.com/questions/572547/what-does-static-mean-in-a-c-program`. – Access date: 11/16/2013

Greenend GREENEND: *Inline Functions In C.* – URL `http://www.greenend.org.uk/rjk/tech/inline.html`. – Access date: 11/16/2013

ISO 1999 ISO: *Programming languages — C.* December 1999. – URL `http://cs.nyu.edu/courses/fall12/CSCI-GA.2110-001/downloads/C99.pdf`

Prinz and Kirch-Prinz 2002 PRINZ, Peter ; KIRCH-PRINZ, Ulla: *C kurz und gut.* 2002

Wikipedia WIKIPEDIA: *Inline Functions in C.* – URL `http://en.wikipedia.org/wiki/Inline_expansion`. – Access date: 11/16/2013

Wolf 2009 WOLF, Jürgen: *C von A bis Z.* 3. Auflage. 2009

Listings

List of Tables