restrict, static & inline Keywords in C

Markus Fasselt

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen
Seminar "Effiziente Programmierung in C"

December 19, 2013

Table of Contents

1. restrict
2. static
3. inline

restrict

lickr.com/photos/joanet/2995263088/

restrict

- included in C99
- `restrict` is a type qualifier on pointer variables
- does not restrict anything!
- says there is no alias for an address / variable

restrict

- included in C99
- `restrict` is a type qualifier on pointer variables
- does not restrict anything!
- says there is no alias for an address / variable

alias

"In computing, aliasing describes a situation in which a data location in memory can be accessed through different symbolic names in the program."

http://en.wikipedia.org/wiki/Aliasing_(computing)

Example Code

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

Example Code

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

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

Example Code

```
1  int main(int argc, char** argv)
2  {
3      int a = 1; int b = 2; int c = 3;
4
5      update(&a, &b, &c);
6      printf("Expected Result: %d %d\n", a, b);
7
8      a = 1; b = 2; c = 3; // reset values
9
10     update_restrict(&a, &b, &c);
11     printf("Actual Result:   %d %d\n", a, b);
12
13     return 0;
14 }
```

```
1  void update(int *a, int *b, int *c)
2  {
3      *a += *c;
4      *b += *c;
5  }
```

Example Code

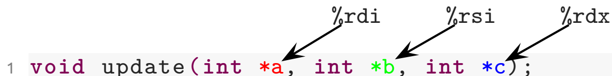
```
int main(int argc, char** argv)
{
    int a = 1; int b = 2; int c = 3;

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

    a = 1; b = 2; c = 3; // reset values

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

    return 0;
}
```

```
Expected Result: 4 5
Actual Result:   4 5
```

How It Works

```c
void update(int *a, int *b, int *c);
```

How It Works

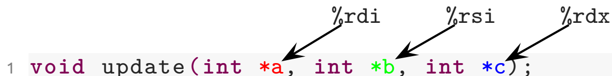



```
1 void update(int *a, int *b, int *c);
```

```
1 update:
2     movq    (%rdx), %rax
3     addq    %rax,   (%rdi)
4     movq    (%rdx), %rax
5     addq    %rax,   (%rsi)
6     ret
```

var	reg	val	*val
a	%rdi	0x123	1
b	%rsi	0x124	2
c	%rdx	0x125	3
-	%rax		-

How It Works

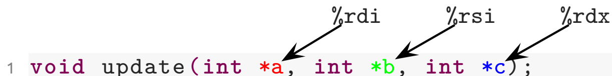



```
1 void update(int *a, int *b, int *c);
```

```
1 update:
2     movq    (%rdx), %rax
3     addq    %rax,   (%rdi)
4     movq    (%rdx), %rax
5     addq    %rax,   (%rsi)
6     ret
```

var	reg	val	*val
a	%rdi	0x123	1
b	%rsi	0x124	2
c	%rdx	0x125	3
–	%rax	3	–

How It Works

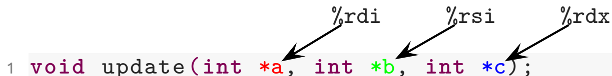



```
1 void update(int *a, int *b, int *c);
```

```
1 update:
2     movq    (%rdx), %rax
3     addq    %rax,   (%rdi)
4     movq    (%rdx), %rax
5     addq    %rax,   (%rsi)
6     ret
```

var	reg	val	*val
a	%rdi	0x123	4
b	%rsi	0x124	2
c	%rdx	0x125	3
-	%rax	3	-

How It Works

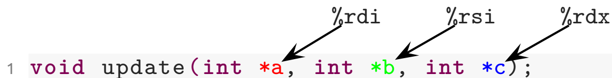



```
1 void update(int *a, int *b, int *c);
```

```
1 update:
2     movq    (%rdx), %rax
3     addq    %rax,   (%rdi)
4     movq    (%rdx), %rax
5     addq    %rax,   (%rsi)
6     ret
```

var	reg	val	*val
a	%rdi	0x123	4
b	%rsi	0x124	2
c	%rdx	0x125	3
–	%rax	3	–

How It Works

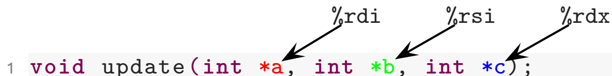



```
1 void update(int *a, int *b, int *c);
```

```
1 update:
2     movq    (%rdx), %rax
3     addq    %rax,   (%rdi)
4     movq    (%rdx), %rax
5     addq    %rax,   (%rsi)
6     ret
```

var	reg	val	*val
a	%rdi	0x123	4
b	%rsi	0x124	5
c	%rdx	0x125	3
-	%rax	3	-

How It Works




```
1 void update(int *a, int *b, int *c);
```

```
1 void update_restrict(int *restrict a, int *restrict
      b, int *restrict c);
```

```
1 update:
2     movq    (%rdx), %rax
3     addq    %rax,   (%rdi)
4     movq    (%rdx), %rax
5     addq    %rax,   (%rsi)
6     ret
```

```
1 update_restrict:
2     movq    (%rdx), %rax
3     addq    %rax,   (%rdi)
4     addq    %rax,   (%rsi)
5     ret
```

Passing Incorrect Parameters

```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);
}
```

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

Passing Incorrect Parameters

```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);
}
```

```
Expected Result: 2 4
Actual Result:   2 3
```

Passing Incorrect Parameters

```c
void update_restrict(int *restrict a, int *restrict
    b, int *restrict c);
```

```asm
update_restrict:
    movq    (%rdx), %rax
    addq    %rax,   (%rdi)
    addq    %rax,   (%rsi)
    ret
```

var	reg	val	*val
a	%rdi	0x123	1
b	%rsi	0x124	2
c	%rdx	0x123	1
–	%rax		–

Passing Incorrect Parameters

```
1 void update_restrict(int *restrict a, int *restrict
      b, int *restrict c);
```

```
1 update_restrict:
2     movq    (%rdx), %rax
3     addq    %rax,    (%rdi)
4     addq    %rax,    (%rsi)
5     ret
```

var	reg	val	*val
a	%rdi	0x123	1
b	%rsi	0x124	2
c	%rdx	0x123	1
–	%rax	1	–

Passing Incorrect Parameters

```c
void update_restrict(int *restrict a, int *restrict
    b, int *restrict c);
```

```asm
update_restrict:
    movq    (%rdx), %rax
    addq    %rax,   (%rdi)
    addq    %rax,   (%rsi)
    ret
```

var	reg	val	*val
a	%rdi	0x123	2
b	%rsi	0x124	2
c	%rdx	0x123	2
–	%rax	1	–

Passing Incorrect Parameters

```c
void update_restrict(int *restrict a, int *restrict
    b, int *restrict c);
```

```
update_restrict:
    movq    (%rdx), %rax
    addq    %rax,   (%rdi)
    addq    %rax,   (%rsi)
    ret
```

var	reg	val	*val
a	%rdi	0x123	2
b	%rsi	0x124	3
c	%rdx	0x123	2
-	%rax	1	-

Performance

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

- 100.000.000.000 Iterations
- without restrict: 249.086421s
- with restrict: 83.690911s

⇒ Performance Benefit: 2,98x faster

Why should I use it?

- Better Performance :)
- `restrict` is recommended to be used in every new code
- You don't need to, if you don't want to!
- (However, be careful!)

static
http://www.flickr.com/photos/38102502@N07/3757737766/in/photolist-6J4owb

Translation Unit (nota bene)

- is the output of the preprocessor / input to the compiler
- is a single .c file after preprocessing
  - header files have been included
  - #ifdef's have been evaluated
  - macros have been expanded
- basically, same translation unit means the same file

static

- used in variable and function defintions
- can be applied on
  1. global variables and functions
  2. variable definitions inside a block

Static Functions and (global) Variables

- static functions and variables are only visible to the current translation unit

```
1 static void static_function();
2 static int c = 0;
3
4 static void static_function()
5 {
6     printf("this is a static function, which can't be
          called from another file...\n");
7 }
```

Static Functions and (global) Variables

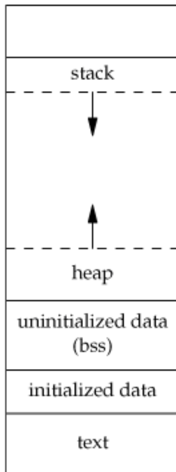
- static functions and variables are only visible to the current translation unit

```
1 static void static_function();
2 static int c = 0;
3
4 static void static_function()
5 {
6     printf("this is a static function, which can't be
          called from another file...\n");
7 }
```

+ encapsulate your private functions
+ provides access control

Static Variables (inside a block)

- initialized on compile time
- with initial value
  $\Rightarrow$ stored in data area
- without initial value (or 0)
  $\Rightarrow$ stored in bss area
- variable keeps state
  between invocations



(http://www.cs.bgu.ac.il/~caspl122/wiki.files/lab2/ch07lev1sec6/ch07lev1sec6.html)

Static Variables (inside a block): Example

```c
void static_variable()
{
    int a = 0;
    static int b = 0;

    a++;
    b++;

    printf("static variable: a=%d, b=%d\n", a, b);
}
```

Static Variables (inside a block): Example

```
 1 void static_variable()
 2 {
 3     int a = 0;
 4     static int b = 0;
 5 
 6     a++;
 7     b++;
 8 
 9     printf("static variable: a=%d, b=%d\n", a, b);
10 }
```

+ keeps state between invocations
− not thread-safe

inline
http://www.flickr.com/photos/joanet/2995263088/

inline

- included in C99
- alternative to macros
- hint to compiler, to embed function body

Example

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

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

$\Rightarrow$ will be substituted to

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

Example: Assembler

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

$\Rightarrow$ will be substituted to

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

inline

- can be combined with `static`
- inline functions can be called from other translation units when defined in header
- `gcc` substitutes only inline function in the same translation unit

Limitations

- Recursive Functions
- inline function with external linkage shall not use private objects (no problem with `gcc`)

Performance

```
1 for(i = 0; i < iterations; i++)
2 {
3     a = add(a, b);
4 }
```

- measured with simple addition-function (like above)
- `100.000.000.000` Iterations
- separate call: `67.553960s`
- inlined: `22.290012s`

⇒ Performance Benefit: 3,03x faster

The Compiler is smarter than you!

the compiler...

- can detect inlinable function without extra hint
- can ignore inline hint

The Compiler is smarter than you!

the compiler...

- can detect inlinable function without extra hint
- can ignore inline hint

but you can force the compiler to do or not to do inlining...!
e.g. for gcc:

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

Should I use inline?

+ can save extra function call
+ may improve instruction cache performance
+ can make other optimizations possible
− may increase final program size

  ⇒ it may be worth doing

- (but today the compiler often does this job for you)

Summary

- restrict
  - hint to the compiler that there is no alias for a variable
  - helps to increase performance
- static
  - access control for private variables and functions
  - static variables inside blocks
- inline
  - similar to marcos
  - hint to the compiler to substitute function calls with its body
  - helps to increase performance

Questions?

Sources

- Peter Prinz, Ulla Kirch-Prinz: C kurz und gut (1st edition, 2002)
- Jürgen Wolf: C von A bis Z (3rd edition, 2009)
- C99 Standard: ISO/IEC 9899:1999
- `http://cellperformance.beyond3d.com/articles/2006/05/demystifying-the-restrict-keyword.html` (November 2013)
- `http://stackoverflow.com/questions/572547/what-does-static-mean-in-a-c-program` (November 2013)
- `http://www.greenend.org.uk/rjk/tech/inline.html` (November 2013)
- `http://en.wikipedia.org/wiki/Inline_expansion` (November 2013)
- `http://www.cs.bgu.ac.il/~caspl122/wiki.files/lab2/ch07lev1sec6/ch07lev1sec6.html` (December 2013)

Thank you for your attention!

http://www.fotocommunity.com/pc/pc/display/29036692