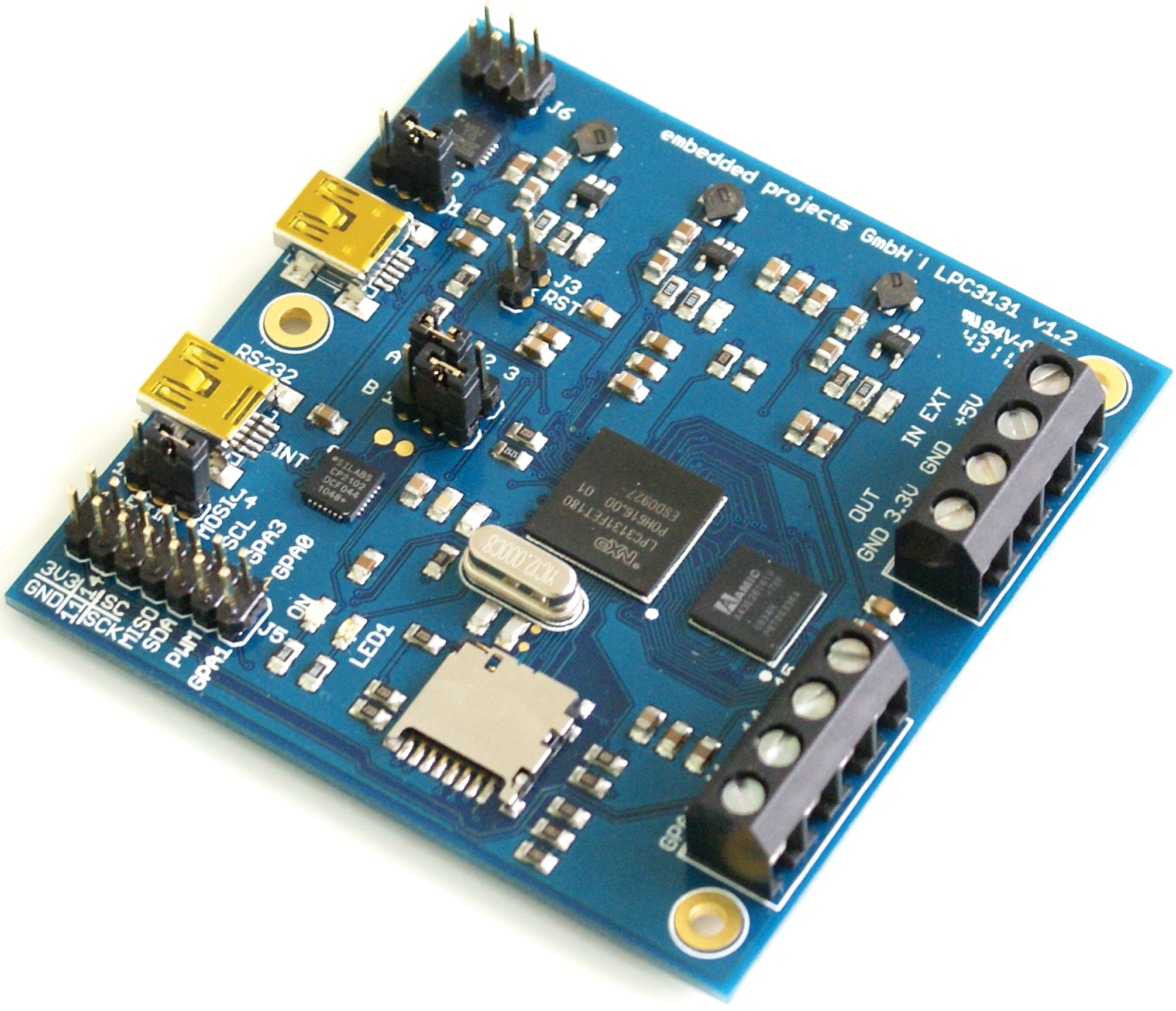
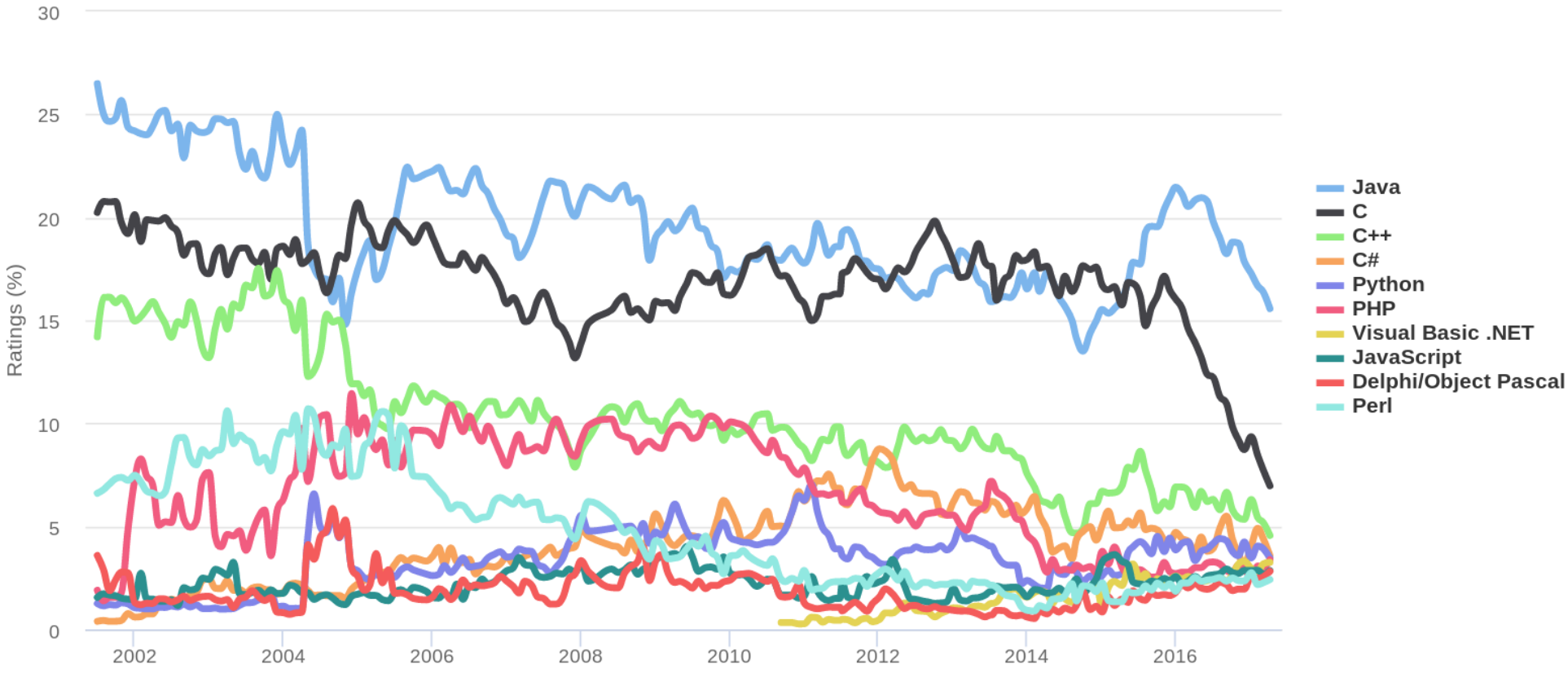
Dark Corners of C

Michael Hartmann

16. Augsburger Linux-Infotag

22. April 2017


embedded projects GmbH | LPC3131 v1.2
J6
J3
RST
RS232
INT
MOSI
SCL
GPA3
GPA0
3V3
GND
SCK
MISO
SDA
PWR
GPA1
J5
ON
LED1
OUT
GND
3.3V
GND
IN EXT
+5V


30
25
20
15
10
5
0
Ratings (%)
2002
2004
2006
2008
2010
2012
2014
2016
Java
C
C++
C#
Python
PHP
Visual Basic .NET
JavaScript
Delphi/Object Pascal
Perl

Die Sprache

Keywords

```
auto break case char const continue default do
double else enum extern float for goto if int
long register return short signed sizeof static
struct switch typedef union unsigned void
volatile while
```

Keywords

```
auto break case char const continue default do
double else enum extern float for goto if int
long register return short signed sizeof static
struct switch typedef union unsigned void
volatile while
```

Keywords

```
auto break case char const continue default do
double else enum extern float for goto if int
long register return short signed sizeof static
struct switch typedef union unsigned void
volatile while
```

Keywords

```
auto break case char const continue default do
double else enum extern float for goto if int
long register return short signed sizeof static
struct switch typedef union unsigned void
volatile while
```

Kommentare

```
1 int x = 2, y = 10, z;
2 int *p = &x;
3
4 /* Fehler */
5 z = y/*p;
6
7 /* syntaktisch korrekt */
8 z = y/ *p;
```

Kommentare

```
1 int x = 10, y;
2 y = x//* kommentar */2;
3 printf("%d\n", y);
```

C89: 5 C99: Fehler

VLA

Variable length arrays

- mandatory in C99
- optional in C11 (?!)

VLA

Variable length arrays

- mandatory in C99
- optional in C11 (?!)

For such an object that does have a variable length array type, its lifetime extends from the declaration of the object until execution of the program leaves the scope of the declaration. If the scope is entered recursively, a new instance of the object is created each time. The initial value of the object is indeterminate.

VLA

Variable length arrays

- mandatory in C99
- optional in C11 (?!)

For such an object that does have a variable length array type, its lifetime extends from the declaration of the object until execution of the program leaves the scope of the declaration. If the scope is entered recursively, a new instance of the object is created each time. The initial value of the object is indeterminate.

Stack? Heap?

VLA

```
1  double sum(size_t nmax) {
2      double sum = 0, buf[nmax];
3
4      for(size_t i = 0; i < nmax; i++)
5          buf[i] = 1./(1+i);
6
7      for(size_t i = 0; i < nmax; i++)
8          sum += buf[i];
9
10     return sum;
11 }
12
13 sum(100);       // 5.18738...
14 sum(10000000); // Segmentation Fault
```

Macros

Macros

```
1 #define SQUARE(x) x*x
2
3 SQUARE(2+1);
```

Macros

```
1 #define SQUARE(x) x*x
2
3 SQUARE(2+1); // 2+1*2+1 = 5
```

Macros

```
1 #define SQUARE(x)  (x)*(x)
2
3 SQUARE(2+1);
```

Macros

```
1 #define SQUARE(x)  (x)*(x)
2
3 SQUARE(2+1);       // (2+1)*(2+1) = 9
```

Macros

```
1 #define SQUARE(x)  (x)*(x)
2
3 SQUARE(2+1);      // (2+1)*(2+1) = 9
4 1./SQUARE(2+1); // 1/.(2+1)*(2+1) = 1.
```

Macros

```
1 #define SQUARE(x)  ((x)*(x))
2
3 1./SQUARE(2+1);       // 1/.(2+1)*(2+1) = 1.
```

Macros

```
1 #define SQUARE(x)  ((x)*(x))
2
3 1./SQUARE(2+1);       // 1/.(2+1)*(2+1) = 1.
4 SQUARE(3000000000); // -494665728
```

Macros

```
1 #define SQUARE(x)  ((x)*(x))
2
3 1./SQUARE(2+1); // 1/.(2+1)*(2+1) = 1.
4 SQUARE(3000000000); // -494665728
5 SQUARE(3000000000.); // 9e18
```

Macros

```
1 #define MIN(X, Y)   ((X) < (Y) ? (X) : (Y))
2
3 min(a,b++);
4 min(x+y,foo(z));
```

Macros

```
1  #define MIN(X, Y)  ((X) < (Y) ? (X) : (Y))
2
3  min(a,min(b,min(c,d)));
4
5  /* becomes */
6  ((a) < (((b) < (((c) < (d) ? (c) : (d))) ?
7  (b) : (((c) < (d) ? (c) : (d))))) ?
8  (a) : (((b) < (((c) < (d) ?
9  (c) : (d))) ? (b) : (((c) < (d) ?
10 (c) : (d))))));
```

Macros

```
1 #define assert(e) \
2         if(!e) panic(__FILE__, __LINE__)
3
4 int x = 0, y = 1;
5 assert(x > y);
6
7 /* becomes */
8 if(!0 > 1) /* 1 > 1 */
9     panic(...);
```

Macros

```
1  #define assert(e) \
2          if(!(e)) panic(__FILE__, __LINE__)
3
4  if(x > 0 && y > 0)
5      assert(x > y);
6  else
7      assert(y > x);
8
9  /* becomes */
10 if(x > 0 && y > 0)
11     if(!(x > y))
12         panic(...);
13     else
14         if(!(y > x))
15             panic(...);
```

Macros

```
1  #define assert(e) \
2          { if(!(e)) panic(__FILE__, __LINE__); }
3
4  if(x > 0 && y > 0)
5      assert(x > y);
6  else
7      assert(y > x);
8
9  /* becomes */
10 if(x > 0 && y > 0)
11     { if(!(x > y)) panic(...); };
12 else
13     { if(!(y > x)) panic(...); };
```

Macros

```
1 /* correct */
2 #define assert(e) \
3   (void)((e)||panic(__FILE__, __LINE__))
4
5 /* should also work */
6 #define assert(e) \
7         do { if(!(e)) panic(__FILE__, __LINE__); } \
8                 while(0)
```

Standardbibliothek

gets

```
1 char *gets(char *s);
```

Never use `gets()`. Because it is impossible to tell without knowing the data in advance how many characters `gets()` will read, and because `gets()` will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security.

realloc

```
1  char *buf = (char *)malloc(elems*sizeof(char));
2  if(buf == NULL)
3          ...
4
5  ...
6
7  buf = realloc(buf, newelems*sizeof(char));
8  if(buf == NULL)
9      /* Memory leak */
10     ...
```

lgamma

The `lgamma()` function returns the natural logarithm of the absolute value of the Gamma function. The sign of the Gamma function is returned in the external integer `signgam` declared in `<math.h>`. [...]

lgamma

Since using a constant location `signgam` is not thread-safe, the functions `lgamma_r()`, `lgammaf_r()`, and `lgammal_r()` have been introduced; they return the sign via the argument `signp`.

lgamma

Since using a constant location `signgam` is not thread-safe, the functions `lgamma_r()`, `lgammaf_r()`, and `lgammal_r()` have been introduced; they return the sign via the argument `signp`.

```
lgamma_r(), lgammaf_r(), lgammal_r():
    _BSD_SOURCE || _SVID_SOURCE
```

errno

```
1 library_function();
2 if(errno)
3     /* Fehlerbehandlung */
```

errno

```
1 errno = 0;
2 library_function();
3 if(errno)
4     /* Fehlerbehandlung */
```

errno

```
1 ret = library_function();
2 if(error_condition)
3     /* inspect errno */
```

Undefined behavior

Undefined behavior

```
1 int buffer[] = { 1,2,3 };
2 int x = buffer[3];
3 printf("%d\n", x);
```

mögliche Ausgabe

- 0
- 42
- -1
- `Uberschreibe Festplatte...`

Was ist undefiniert?

- lesen uninitialisierter Variablen
- Dereferenzieren von wilden Pointern, Out of Bounds Array Accesses
- Dereferenzieren des `NULL` Pointers
- Casting: `int *` nach `float *` casten und Pointer dereferenzieren
- Integer Division durch 0
- *An unmatched ‘ or " character is encountered on a logical source line during tokenization.*
- ...

Kürzeste Funktion mit undefinierten Verhalten

```
1 int f(int x) { return -x; }
2
3 int main(void)
4 {
5     printf("f(%d)=%d\n", INT_MIN, f(INT_MIN));
6     return 0;
7 }
```

auf meinem Rechner:

```
f(-2147483648)=-2147483648
```

Zweierkomplement

Bits	Wert
011	3
010	2
001	1
000	0
111	−1
110	−2
101	−3
100	-4

Aber: Der Compiler kann machen, was er will!

Vorteile

- uninitialisierte Variablen: `memset` für Arrays teuer...
- Signed integer overflows:

```
1 int i;
2 i+1 > i; /* true */
3 i*2/2; /* i */
4
5 /* Schleife wird N+1 mal ausgefuehrt */
6 for(i = 0; i <= N; ++i) {
7         ...
8 }
```

- Out of Bounds Array Access: Checks zur Laufzeit teuer, würde Binärkompatibilität mit alten Programmen brechen

Undefiniertes Verhalten und Optimierungen

```
1 void f(int *P) {
2     int dead = *P;
3     if (P == NULL)
4         return;
5     *P = 4;
6 }
```

Zwei Optimierungen:

- Dead Code Elimination
- Redundant Null Check Elimination:

Verhalten hängt von der Reihenfolge der Optimierungen ab!

Undefiniertes Verhalten und Optimierungen

Dead Code Elimination:

```
1 void f(int *P) {
2       int dead = *P; /* dead code */
3       if (P == NULL)
4           return;
5       *P = 4;
6 }
```

Undefiniertes Verhalten und Optimierungen

Redundant Null Check Elimination:

```
1 void f(int *P) {
2     if (P == NULL) /* Check nicht redundant */
3         return;
4     *P = 4;
5 }
```

Undefiniertes Verhalten und Optimierungen

Code mit zuerst DCE und danach RNCE:

```
1 void f(int *P) {
2       if (P == NULL)
3           return;
4       *P = 4;
5 }
```

Undefiniertes Verhalten und Optimierungen

Und jetzt anders herum!

Undefiniertes Verhalten und Optimierungen

Redundant Null Check Elimination:

```
1 void f(int *P) {
2       int dead = *P;
3       if (P == NULL) /* P darf nicht NULL sein */
4           return;
5       *P = 4;
6 }
```

Undefiniertes Verhalten und Optimierungen

Dead code elimination:

```
1 void f(int *P) {
2       int dead = *P; /* dead code */
3       *P = 4;
4 }
```

Undefiniertes Verhalten und Optimierungen

Code mit zuerst RNCE und danach DCE:

```
1 void f(int *P) {
2       *P = 4;
3 }
```

Undefiniertes Verhalten und Security

```
1  void process_something(int size) {
2      /* check for integer overflow */
3      if (size > size+1)
4          abort();
5
6      ...
7
8      /* alles ok hier */
9      char *string = malloc(size+1);
10     read(fd, string, size);
11     string[size] = 0;
12     do_something(string);
13     free(string);
14 }
```

Undefiniertes Verhalten und Security

Signed integer Overflows sind undefiniert...

Nach Optimieren sieht der Code vermutlich so aus:

```
1  void process_something(int size) {
2      ...
3
4      /* alles ok hier */
5      char *string = malloc(size+1);
6      read(fd, string, size);
7      string[size] = 0;
8      do_something(string);
9      free(string);
10 }
```

Real-world Beispiel

```
1  void __devexit agnx_pci_remove (struct pci_dev *pdev)
2  {
3      struct ieee80211_hw *dev = pci_get_drvdata(pdev);
4      struct agnx_priv *priv = dev->priv;
5
6      if (!dev)
7          return;
8
9      ...
10 }
```

gcc

`-fdelete-null-pointer-checks`:

Assume that programs cannot safely dereference null pointers, and that no code or data element resides there. This enables simple constant folding optimizations at all optimization levels. In addition, other optimization passes in GCC use this flag to control global dataflow analyses that eliminate useless checks for null pointers; these assume that if a pointer is checked after it has already been dereferenced, it cannot be null.

Note however that in some environments this assumption is not true. Use `-fno-delete-null-pointer-checks` to disable this optimization for programs that depend on that behavior.

Undefiniertes Verhalten und Debugging...

```
1  void f(bool b) {
2      if(b)
3          printf("true\n");
4      else
5          printf("false\n");
6  }
7
8  int main(int argc, char *argv[]) {
9      bool *p = (bool *)malloc(sizeof(bool));
10
11     f(*p);
12     f(!*p);
13
14     return 0;
15 }
```

Undefiniertes Verhalten und Debugging...

```
$ gcc -Wall -Wextra -O2 truefalse.c
$ ./a.out
false
true
```

Undefiniertes Verhalten und Debugging...

```
$ clang -Wall -Wextra -Oz truefalse.c
$ ./a.out
false
false
```

Helfer

Helfer

- Compiler Warnungen: `-Wall -Wextra`
- Clang Static Analyzer
- Clang's `-fsanitize` Option
- valgrind
- gcc integer overflow builtins:
  `__builtin_add_overflow`,
  `__builtin_mul_overflow`, ...

Clang Static Analyzer

```
5  void f(bool b) {
6      if(b)
7          printf("true\n");
8      else
9          printf("false\n");
10 }
11
12 int main(void) {
13     bool *p = (bool *)malloc(sizeof(bool));
14
15     f(*p);
16     f(!*p);
17
18     return 0;
19 }
```

Function call argument is an uninitialized value

valgrind

```
$ valgrind ./a.out

==27502== Conditional jump or move depends on
uninitialised value(s)
==27502== at 0x400573: f (in a.out)
==27502== by 0x400467: main (in a.out)
==27502==
false
...
```

Clang's `-fsanitize` Option

```
$ clang -fsanitize=undefined,memory truefalse.c
$ ./a.out
==27607== WARNING: MemorySanitizer:
use-of-uninitialized-value
    #0 0x7f53cadbd873 (a.out+0x94873)
    #1 0x7f53c9c3cb44 (libc.so.6+0x21b44)
    #2 0x7f53cadbd38c (a.out+0x9438c)

SUMMARY: MemorySanitizer:
use-of-uninitialized-value ??:0 ??
Exiting
```

Vielen Dank für die Aufmerksamkeit!

Credits and links:

- What Every C Programmer Should Know About Undefined (LLVM)
- A Guide to Undefined Behavior in C and C++ (Embedded in Academia)
- Both true and false (mark shroyer, dot com)
- Some dark corners of C (Rob Kendrick)
- Clang Static Analyzer

Bonus Folien

Find a bug

```
1 int div(int a, int b)
2 {
3       assert(b != 0);
4       return a / b;
5 }
```

Find a bug

```
1 const char *str = "Hallo␣Welt";
2 size_t length = strlen(str);
3 for(size_t i = length - 1; i >= 0; i--)
4     putchar(str[i]);
```

Find a bug

```
1 int my_strlen(const char *x)
2 {
3       int len = 0;
4       while(*x++)
5           len++;
6
7       return len;
8 }
```

Kann die Funktion `inf` zurückgeben?

```
1 double div(double x, double y) {
2     return x/y;
3 }
```

Was ist `add(UINT_MAX,1)`?

```
1 unsigned int add(unsigned int x, unsigned int y)
2     return x+y;
3 }
```