# C Traps & Pitfalls

## Operating Systems UE
## 2022W

David Lung, Florian Mihola, Andreas Brandstätter,
Axel Brunnbauer, Peter Puschner

Technische Universität Wien
Computer Engineering
Cyber-Physical Systems

2022-10-06

# About C...

Andrew Koenig - AT&T Bell Labs:

> *The C language is like a carving knife: simple, sharp, and extremely useful in skilled hands. Like any sharp tool, C can injure people who don't know how to handle it.*

# Overview

1. Lexical pitfalls
2. Syntactic pitfalls
3. Semantic pitfalls
4. Common pitfalls
   - Implicit initialization
   - Dynamic memory allocation
   - Pointers, arrays and strings
   - Makros
5. Coding/Style guidelines

# Part I

# Lexical Pitfalls

# Lexical Pitfalls

- At compile time the source code is split into tokens two times
- Preprocessor (e.g., for makro replacements)
- Compiler itself to create an executable program

# Multi Character Token

- Compiler has to decide whether two characters that follow each other correspond to the same token or whether they are two tokens

```
y = x/*p
```

# Multi Character Token

- Compiler has to decide whether two characters that follow each other correspond to the same token or whether they are two tokens

```
y = x/*p
```

# Multi Character Token

- Compiler has to decide whether two characters that follow each other correspond to the same token or whether they are two tokens

```
y = x/*p
```

# Multi Character Token

- Compiler has to decide whether two characters that follow each other correspond to the same token or whether they are two tokens

```
y = x/*p /* p points at divisor */;
/* Compiler interprets first "/*" as
 * the beginning of a comment (use -Wall) */

/* better */
y = x / *p  /* p points at divisor */;
y = x /(*p) /* p points at divisor */;
y = x / *p; /* comment after statement */
```

# = vs. ==

- = to assign values
- == to evaluate values

# = vs. ==

- = to assign values
- == to evaluate values

```
if (x = y) /* set x to y and check if not zero */
{
  foo();
}
```

# = vs. ==

- = to assign values
- == to evaluate values

```
if (x = y) /* set x to y and check if not zero */
{
  foo();
}

while (c == ' ' || c = '\t' || c == '\n')
{
  c = getc(f);
}
```

# = vs. ==

- If you really want to assign a value and evaluate it, you should do an explicit evaluation

# = vs. ==

- If you really want to assign a value and evaluate it, you should do an explicit evaluation

```
if ( (x = y) != 0 )
{
  foo();
}
```

# = vs. ==

- If you really want to assign a value and evaluate it, you should do an explicit evaluation

```
if ( (x = y) != 0 )
{
  foo();
}

x = y;
if ( x != 0 )
{
  foo();
}
```

# & and — vs. && and ——

- & and | are bitwise operations
- && and || are logical opertions

# & and — vs. && and ——

- & and | are bitwise operations
- && and || are logical opertions

```
int a = 0x4, b = 0x2;

if (a && b) /* true */
{
    foo();
}
```

# & and — vs. && and ——

- & and | are bitwise operations
- && and || are logical opertions

```
int a = 0x4, b = 0x2;

if (a && b) /* true */
{
    foo();
}

if (a & b) /* false */
{
    foo();
}
```

# Characters

- Data type for characters: `char`
- ’ ’ for character-literals

# Characters

- Data type for characters: `char`
- ’ ’ for character-literals
  - Just another way to write an integer
  - Code of the character (depends on encoding)

# Characters

- Data type for characters: `char`
- ’ ’ for character-literals
  - Just another way to write an integer
  - Code of the character (depends on encoding)
- Escape sequences with preceding \
  - For control chars or other chars that can’t be written in the source code

# Characters

- Data type for characters: `char`
- '' for character-literals
  - Just another way to write an integer
  - Code of the character (depends on encoding)
- Escape sequences with preceding \
  - For control chars or other chars that can't be written in the source code

```
char c = 'a';   /* 97 in ASCII */
char d = '\n'; /* newline */
```

# Strings

- Data type for strings: `char*` (or `char[]`)
- `""` for string literals
  - of type `const char*`

# Strings

- Data type for strings: `char*` (or `char[]`)
- `""` for string literals
  - of type `const char*`

```
char *string = "hallo";
```

# Strings

- Data type for strings: `char*` (or `char[]`)
- `""` for string literals
  - of type `const char*`

```
char *string = "hallo";

/* same behavior, difference will be
 * explained later: */
char string[] = "hello";
char string[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

# Part II

# Syntactic Pitfalls

# Declarations

- `int x, ((y));`

# Declarations

- `int x, ((y));`
  - x and y are int

- `int *i, j;`

# Declarations

- `int x, ((y));`
  - x and y are int
- `int *i, j;`
  - i is pointer on int
  - j is int
- `int f();`

# Declarations

- `int x, ((y));`
  - x and y are int
- `int *i, j;`
  - i is pointer on int
  - j is int
- `int f();`
  - Function that returns int
- `int *g();`

# Declarations

- `int x, ((y));`
  - x and y are int
- `int *i, j;`
  - i is pointer on int
  - j is int
- `int f();`
  - Function that returns int
- `int *g();`
  - Function that returns a pointer to an int
- `int (*h)();`

# Declarations

- `int x, ((y));`
  - x and y are int
- `int *i, j;`
  - i is pointer on int
  - j is int
- `int f();`
  - Function that returns int
- `int *g();`
  - Function that returns a pointer to an int
- `int (*h)();`
  - Pointer to a function that returns an int

# Cast

- Cast is like a declaration, but without a name for a variable, without semicolons and in brackets

# Cast

- Cast is like a declaration, but without a name for a variable, without semicolons and in brackets
- Declaration: `int (*h)();`

# Cast

- Cast is like a declaration, but without a name for a variable, without semicolons and in brackets
- Declaration: `int (*h)();`
- Cast: `(int (*)()) h`
  - casts h to a function pointer

# Example

Call of a function, whose address is at memory location 0

- Try: `(*0)();`

# Example

Call of a function, whose address is at memory location 0

- Try: (*0)();
  - Error: `0` has the wrong type (`int`)
  - We need an object of type void (*)();

# Example

Call of a function, whose address is at memory location 0

- Try: (*0)();
  - Error: `0` has the wrong type (`int`)
  - We need an object of type void (*)();
- Solution: Cast

# Example

Call of a function, whose address is at memory location 0

- Try: `(*0)();`
  - Error: `0` has the wrong type (`int`)
  - We need an object of type `void (*)();`
- Solution: Cast
  - `(*(void(*)())0)();`

# Operator Precedence

- Always pay attention to the order of the evaluation

```
if (flags & FLAG) ...
```

# Operator Precedence

- Always pay attention to the order of the evaluation

```
if (flags & FLAG) ...

/* better to be more explicit? */
if (flag & FLAG != 0) ...
```

# Operator Precedence

- Always pay attention to the order of the evaluation

```
if (flags & FLAG) ...

/* better to be more explicit? */
if (flag & FLAG != 0) ...

/* PROBLEM: != is stronger than & */
/* Compiler's interpretation: */
if (flag & (FLAG != 0))
```

# Operator Precedence cntd.

```
r = h<<4 + 1;
```

# Operator Precedence cntd.

```
r = h<<4 + 1;
/* + is stronger, compiler's interpretation: */
r = h << (4 + 1);

c = a---b;
```

# Operator Precedence cntd.

```
r = h<<4 + 1;
/* + is stronger, compiler's interpretation: */
r = h << (4 + 1);

c = a---b; /* c = a-- - b; */
```

- Make it explicit! It's better to have too much brackets than too few.

# Operator Precedence Table[1]

| Operator | Explanation |
|---|---|
| `()` | Function call |
| `[]` | Array indexing |
| `. ->` | Structure element selection |
| `++ --` | Postfix increment/decrement |
| `++ --` | Prefix increment/decrement |
| `+ - ! ~` | Unary operatoren: plus/minus/logical Negation/bitwise negation |
| `(type)` | Type casts |
| `* &` | Dereferencing/Address operator |
| `sizeof` | Determining the memory consumption in byte |
| `* / %` | Multiplication/division/modulo |
| `+ -` | Addition/Subtraction |
| `<< >>` | Bitweise shift left/right |

[1] https://en.cppreference.com/w/c/language/operator_precedence

# Operator Precedence Table

| Operator | Explanation |
|---|---|
| `< <= > >=` | Relational operators: smaller than/smaller or equal |
| `== !=` | Relational operators: qual to/not equal to |
| `&` | bitwise AND |
| `^` | bitwise exclusive OR |
| `\|` | bitwise inclusive OR |
| `&&` | logical AND |
| `\|\|` | logical OR |
| `?:` | Ternary operator |
| `=` | Assignment |
| `+= -= *=` | Ass. with addition/subtraction/multiplication |
| `/= %= &=` | Ass. with division/modulo/bitwise AND |
| `^= \|=` | Ass. with exclusive OR/inclusive OR |
| `<<= >>=` | Ass. with bitweise shift left/right |
| `,` | Separation operator |

# Semicolons

```
if (x[i] > big);
  big = x[i];
```

# Semicolons

```
if (x[i] > big);
  big = x[i];
```

- A new value to `big` will always be assigned!

# Semicolons

```
if (x[i] > big);
  big = x[i];
```

- A new value to `big` will always be assigned!
- `if`/`while`/`for`... expect a statement
  - Single statement
  - Block

# Semicolons

```
if (x[i] > big);
  big = x[i];
```

- A new value to `big` will always be assigned!
- `if`/`while`/`for`... expect a statement
  - Single statement
  - Block
- Semicolon following the condition is an empty statement

# switch

- When using `switch` you shall not forget to add a `break`, otherwise the next `case` blocks will be evaluated
- Always add a `default` case

# switch

- When using `switch` you shall not forget to add a `break`, otherwise the next `case` blocks will be evaluated
- Always add a `default` case

```
int color = 2;

/* bad */
switch (color) {
  case 1: printf("red");
  case 2: printf("green");
  case 3: printf("blue");
}
```

# switch

- When using `switch` you shall not forget to add a `break`, otherwise the next `case` blocks will be evaluated
- Always add a `default` case

```
int color = 2;

/* bad */
switch (color) {
  case 1: printf("red");
  case 2: printf("green");
  case 3: printf("blue");
}
/* prints "greenblue" */
```

# Dangling else

- `else` always refers to the nearest (, not valid) `if` condition

# Dangling else

- `else` always refers to the nearest (, not valid) `if` condition

```
if (x == 0)
  if (y == 0) error();
else {
  f(x+y);
}
```

# Dangling else

- `else` always refers to the nearest (, not valid) `if` condition

```
if (x == 0)
  if (y == 0) error();
else {
  f(x+y);
}

/* compiler's interpretation: */
if (x == 0) {
  if (y == 0)
    error();
  else {
    f(x+y);
  }
}
```

# Part III

# Semantic Pitfalls

# Pointer and Arrays

- C organizes arrays as linearly continuing memory block

# Pointer and Arrays

- C organizes arrays as linearly continuing memory block
- C89: Size of array is fixed at compile time, C99: variable sizes are possible (z.B.: `char buf[argc]`)

# Pointer and Arrays

- C organizes arrays as linearly continuing memory block
- C89: Size of array is fixed at compile time, C99: variable sizes are possible (z.B.: `char buf[argc]`)
- Two array options:
  1. Evaluate the size of the array
  2. Return address of element `0`

# Pointer and Arrays

- C organizes arrays as linearly continuing memory block
- C89: Size of array is fixed at compile time, C99: variable sizes are possible (z.B.: `char buf[argc]`)
- Two array options:
  1. Evaluate the size of the array
  2. Return address of element `0`
- All other operations are realized with pointer operations
  - (z.B. Indexing)

# Pointer and Arrays

- C organizes arrays as linearly continuing memory block
- C89: Size of array is fixed at compile time, C99: variable sizes are possible (z.B.: `char buf[argc]`)
- Two array options:
  1. Evaluate the size of the array
  2. Return address of element 0
- All other operations are realized with pointer operations
  - (z.B. Indexing)

```
int a[] = {1, 2, 3, 4, 5};

printf("%d\n", a[3]); /* OK */
printf("%d\n", 3[a]); /* OK! but strange.. */
```

# Arrays as Parameter

- Passing an array as parameter ends up in converting the parameter to a pointer which points to the first element
  - The array is not copied!

# Arrays as Parameter

- Passing an array as parameter ends up in converting the parameter to a pointer which points to the first element
  - The array is not copied!
- Compiler converts array-parameter declaration in the appropriate pointer declaration

```
int main(int argc, char **argv) { ... }
int main(int argc, char *argv[]) { ... }
/* both statements are the same */
```

# Arrays as Parameter

- Passing an array as parameter ends up in converting the parameter to a pointer which points to the first element
  - The array is not copied!
- Compiler converts array-parameter declaration in the appropriate pointer declaration

```
int main(int argc, char **argv) { ... }
int main(int argc, char *argv[]) { ... }
/* both statements are the same */
```

- This conversion (equivalence) is valid only for parameters

# Differences: Pointer and Arrays[2]

- Pointers and arrays are equivalent in C, right?

[2] http://www.c-faq.com

# Differences: Pointer and Arrays[2]

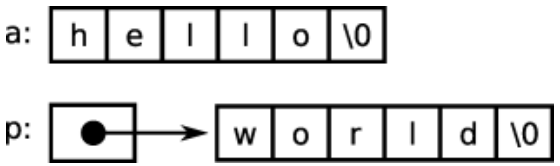
- Pointers and arrays are equivalent in C, right?
- Nope! But they behave similar

[2] http://www.c-faq.com

# Differences: Pointer and Arrays[2]

- Pointers and arrays are equivalent in C, right?
- Nope! But they behave similar

```
char a[] = "hello";
char *p = "world";
```

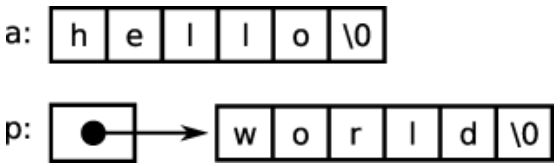



[2] http://www.c-faq.com

# Differences: Pointer and Arrays[2]

- Pointers and arrays are equivalent in C, right?
- Nope! But they behave similar

```
char a[] = "hello";
char *p = "world";
```



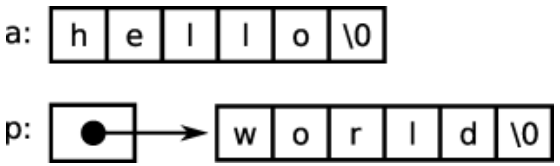

- `p[3]`: Start at `p`, get the value (the address), add `3 * sizeof(*p)` to the address and get the element the address is pointing to

[2] http://www.c-faq.com

# Differences: Pointer and Arrays[2]

- Pointers and arrays are equivalent in C, right?
- Nope! But they behave similar

```
char a[] = "hello";
char *p = "world";
```




- `p[3]`: Start at `p`, get the value (the address), add `3 * sizeof(*p)` to the address and get the element the address is pointing to
- `a[3]`: Start at `a`, get the address of the first element, add `3 * sizeof(a[0])` to the address and get the element the address is pointing to

[2] http://www.c-faq.com

# Differences: Pointer and Arrays

```
int a[] = {1,2,3};
int *p = a; /* Address of first element
               same as: &a[0] */
```

# Differences: Pointer and Arrays

```c
int a[] = {1,2,3};
int *p = a; /* Address of first element
               same as: &a[0] */

printf("%d %d\n", a[2], p[2]); /* 3 3 */
```

# Differences: Pointer and Arrays

```
int a[] = {1,2,3};
int *p = a; /* Address of first element
               same as: &a[0] */

printf("%d %d\n", a[2], p[2]); /* 3 3 */

printf("%ld %ld %ld\n",
  sizeof(a), sizeof(p), sizeof(*p);
  /* on someone's pc: 12 8 4 */
```

# Differences: Pointer and Arrays

```c
int a[] = {1,2,3};
int *p = a; /* Address of first element
               same as: &a[0] */

printf("%d %d\n", a[2], p[2]); /* 3 3 */

printf("%ld %ld %ld\n",
  sizeof(a), sizeof(p), sizeof(*p);
  /* on someone's pc: 12 8 4 */

#define NRELEMENTS(a) (sizeof(a) / sizeof(a[0]))
/* Works for arrays, but not for pointers! */
```

# Strings

- Strings (as char[]) cannot be assigned to
  - Exception: initialization

# Strings

- Strings (as char[]) cannot be assigned to
  - Exception: initialization
- Use string functions
  - `#include <string.h>`

# Strings

- Strings (as char[]) cannot be assigned to
  - Exception: initialization
- Use string functions
  - `#include <string.h>`

```
char buffer[256] = "init"; /* OK */

buffer = "hello";          /* Error */
strcpy(buffer, "hello");   /* OK */
```

# Strings

- Strings (as char[]) cannot be assigned to
  - Exception: initialization
- Use string functions
  - `#include <string.h>`

```
char buffer[256] = "init"; /* OK */

buffer = "hello";          /* Error */
strcpy(buffer, "hello");   /* OK */
```

- Strings as char* can be assigned to
  - Uses pointer only
  - nothing is copied

# Strings

- Strings (as char[]) cannot be assigned to
  - Exception: initialization
- Use string functions
  - `#include <string.h>`

```
char buffer[256] = "init"; /* OK */

buffer = "hello";          /* Error */
strcpy(buffer, "hello");   /* OK */
```

- Strings as char* can be assigned to
  - Uses pointer only
  - nothing is copied

```
char *buf = "abc";

buf = "def"; /* OK */
```

# Important string functions

| Function | Explanation |
|---|---|
| strlen(3) | Length of the string (excl. terminating '\0') |
| strstr(3) | Locates a substring |
| strcpy(3), strncpy(3) | Copies a string |
| strdup(3), strndup(3) | Duplicates a string |
| strcmp(3), strncmp(3) | Compares strings |
| strtol(3), strtoll(3) | String $\rightarrow$ integer conversion |
| strtof(3), strtod(3), strtold(3) | String $\rightarrow$ floating-point number conversion |

- The number in brackets refers to the chapter in the man-page (`man 3 strlen`)

# Order of Evaluation[3]

- The order of evaluation of operators is defined for four operators only

[3] https://en.cppreference.com/w/c/language/eval_order

# Order of Evaluation[3]

- The order of evaluation of operators is defined for four operators only
  - `&&`
  - `||`
  - `?:`
  - `,`

[3] https://en.cppreference.com/w/c/language/eval_order

# Order of Evaluation[3]

- The order of evaluation of operators is defined for four operators only
  - `&&`
  - `||`
  - `?:`
  - `,`
- `a < b && c < d`

[3] https://en.cppreference.com/w/c/language/eval_order

# Order of Evaluation[3]

- The order of evaluation of operators is defined for four operators only
  - `&&`
  - `||`
  - `?:`
  - `,`
- `a < b && c < d`
  - `a < b` will definitely be evaluated before `c < d`

[3] https://en.cppreference.com/w/c/language/eval_order

# Order of Evaluation[3]

- The order of evaluation of operators is defined for four operators only
  - `&&`
  - `||`
  - `?:`
  - `,`
- `a < b && c < d`
  - `a < b` will definitely be evaluated before `c < d`
  - if `a` before `b` or `b` before `a` is evaluated is specific to the compiler

[3] https://en.cppreference.com/w/c/language/eval_order

# Order of Evaluation[4]

```
while (i < n)
  y[i] = x[i++];
```

- The order of evaluation is not defined again

[4] https://en.cppreference.com/w/c/language/eval_order

# Order of Evaluation[4]

```
while (i < n)
  y[i] = x[i++];
```

- The order of evaluation is not defined again
- Thought model: A tree is constructed with the help of precedence and associativity. The order in which the leaves are evaluated is not defined

[4] https://en.cppreference.com/w/c/language/eval_order

# Part IV

# Common Pitfalls

# Implicit Initialization

- `int i; /* global variable */`

# Implicit Initialization

- `int i; /* global variable */`
  - $\texttt{i} = 0$
- `static int j;`

# Implicit Initialization

- `int i; /* global variable */`
  - `i` = 0
- `static int j;`
  - `j` = 0
- `int main() { int k; }`

# Implicit Initialization

- `int i; /* global variable */`
  - `i` = 0
- `static int j;`
  - `j` = 0
- `int main() { int k; }`
  - Value of `k` is undefined!

# Implicit Initialization

- `int i; /* global variable */`
  - `i` = 0
- `static int j;`
  - `j` = 0
- `int main() { int k; }`
  - Value of `k` is undefined!
- In principle there is no implicit initialization
  - Exception: global and static variables
- Initialize explicitly!

# malloc

- Check return value (as you do it every time)
  - Very important when using `malloc`

# malloc

- Check return value (as you do it every time)
  - Very important when using `malloc`

```
char *p;
p = malloc(sizeof (*p) * 6);
/* strlen + 1 */
if (p == NULL)
{
  bailout();
}
else
{
  strcpy(p, "hello");
}
free(p); /* do not forget to "free"! */
```

# free

- Dynamically allocated memory needs to be freed

# free

- Dynamically allocated memory needs to be freed
- Pay attention when using pointer arithmetic

```
char *p = malloc(sizeof (*p) * 6);
...

p += 3;
free(p); /* undefined behavior (segfault!) */
```

# free

- Dynamically allocated memory needs to be freed
- Pay attention when using pointer arithmetic

```
char *p = malloc(sizeof (*p) * 6);
...

p += 3;
free(p); /* undefined behavior (segfault!) */

p -= 3;
free(p); /* we're good again */
```

# free

- Dynamically allocated memory needs to be freed
- Pay attention when using pointer arithmetic

```
char *p = malloc(sizeof (*p) * 6);
...

p += 3;
free(p); /* undefined behavior (segfault!) */

p -= 3;
free(p); /* we're good again */
```

- Never ever free memory two times
  - double free
  - undefined behavior (segfault!)

# That's not the way..

```
char str1[] = "text";      /* OK */
char *str2 = str1;         /* OK */
char *str3;                /* OK */
strcpy(str2, "out of bounds"); /* Error */
strcpy(str3, "hello");     /* No memory available */
```

# That's not the way..

```
char str1[] = "text";      /* OK */
char *str2 = str1;         /* OK */
char *str3;                /* OK */
strcpy(str2, "out of bounds"); /* Error */
strcpy(str3, "hello");     /* No memory available */

char *str = "constant";
strcpy(str, "hello");
/* Tries to override constant memory area */
/* no warning, but segfault! */
```

# That's not the way..

```c
char str1[] = "text";      /* OK */
char *str2 = str1;         /* OK */
char *str3;                /* OK */
strcpy(str2, "out of bounds"); /* Error */
strcpy(str3, "hello");     /* No memory available */

char *str = "constant";
strcpy(str, "hello");
/* Tries to override constant memory area */
/* no warning, but segfault! */

char str[MAX_LENGTH];
strcpy(str, str_from_user);
/* A vicious user can create an overflow */
```

# That's fine

```
char str[MAX_LENGTH];

strncpy(str, str_from_user, MAX_LENGTH - 1);
str[MAX_LENGTH - 1] = '\0';
```

# That's fine

```
char str[MAX_LENGTH];

strncpy(str, str_from_user, MAX_LENGTH - 1);
str[MAX_LENGTH - 1] = '\0';
```

```
$ man 3 strncpy #man strncpy
The strncpy() function is similar, except that at most n
bytes of src are copied.  Warning: If there is no null
byte among the first n bytes of src, the string placed in
dest will not be null terminated.
```

# That's fine

```
char str[MAX_LENGTH];

strncpy(str, str_from_user, MAX_LENGTH - 1);
str[MAX_LENGTH - 1] = '\0';
```

```
$ man 3 strncpy #man strncpy
The strncpy() function is similar, except that at most n
bytes of src are copied.  Warning: If there is no null
byte among the first n bytes of src, the string placed in
dest will not be null terminated.
```

- Note: OpenBSD developer fixed the problem more than 10 years ago (`strlcpy`)

# That's fine

```
char str[MAX_LENGTH];

strncpy(str, str_from_user, MAX_LENGTH - 1);
str[MAX_LENGTH - 1] = '\0';
```

```
$ man 3 strncpy #man strncpy
The strncpy() function is similar, except that at most n
bytes of src are copied.  Warning: If there is no null
byte among the first n bytes of src, the string placed in
dest will not be null terminated.
```

- Note: OpenBSD developer fixed the problem more than 10 years ago (`strlcpy`)
  - Not in C standard library $\Rightarrow$ problem with portability
  - in C11: `strcpy_s` and `strncpy_s`

# Makros

```
#define SQR(x) x*x

int a = 2;
int b = 2;

SQR(a)
```

# Makros

```
#define SQR(x) x*x

int a = 2;
int b = 2;

SQR(a)
```

# Makros

```
#define SQR(x) x*x

int a = 2;
int b = 2;

SQR(a)    /* 4 */
SQR(a+b)
```

# Makros

```c
#define SQR(x) x*x

int a = 2;
int b = 2;

SQR(a)    /* 4 */
SQR(a+b)  /* a + b * a + b == 8 */
SQR(a++)
```

# Makros

```
#define SQR(x) x*x

int a = 2;
int b = 2;

SQR(a)    /* 4 */
SQR(a+b)  /* a + b * a + b == 8 */
SQR(a++)  /* a++ * a++, undef!, a == 4 */
```

# Makros

```
#define SQR(x) x*x

int a = 2;
int b = 2;

SQR(a)    /* 4 */
SQR(a+b)  /* a + b * a + b == 8 */
SQR(a++)  /* a++ * a++, undef!, a == 4 */

a = 2;
SQR(++a)  /* ++a * ++a, undef!, a == 4 */
```

# Makros: Multiple Statements

```
#define CMDS \
  a = b; \
  c = d;

if (var == 23)
  CMDS
else
  return;
```

# Makros: Multiple Statements

```
#define CMDS \
  a = b; \
  c = d;

if (var == 23)
  CMDS
else
  return;
```

Is converted to:

```
if (var == 23)
  a = b;
c = d;
else /* syntax error */
  return;
```

# Makros: Multiple Statements

```
#define CMDS \
  a = b; \
  c = d;

if (var == 23)
  CMDS
else
  return;
```

Is converted to:

```
if (var == 23)
  a = b;
c = d;
else /* syntax error */
  return;
```

Without `else`, `c = d` is always executed!

# Makros: Multiple Statements: Solution

```
#define CMDS \
{ \
  a = b; \
  c = d; \
}
```

# Makros: Multiple Statements: Solution

```
#define CMDS \
{ \
  a = b; \
  c = d; \
}
```

Is converted to:

```
if (var == 23)
{
  a = b;
  c = d;
}
else
  return;
```

Part V

# Coding Guidelines

# Guidelines/Conventions/Standards

- K&R (see material)
- GNU Coding Standards: `http://www.gnu.org/prep/standards/`
- Linux Coding Style: `/usr/src/linux/Documentation/CodingStyle`
- OpenBSD Kernel Style: `http://man.openbsd.org/OpenBSD-current/man9/style.9`
- CERT Secure Coding: `https://www.securecoding.cert.org/confluence/display/seccode/SEI+CERT+Coding+Standards`
- The Power of Ten: `http://spinroot.com/gerard/pdf/P10.pdf`
- OSUE Conventions

However (unfortunately), they are not compatible

Part VI

# Outlook

# Material

Material:

- C Programming Language - Kernighan & Ritchie
- C Traps and Pitfalls - Andrew Koenig
- https://en.wikibooks.org/wiki/C_Programming
- https://de.wikibooks.org/wiki/C-Programmierung
- http://www.c-faq.com/