The C Programming Language

Jörg Faschingbauer

Table of Contents

FASCHINGBAUER

Overview

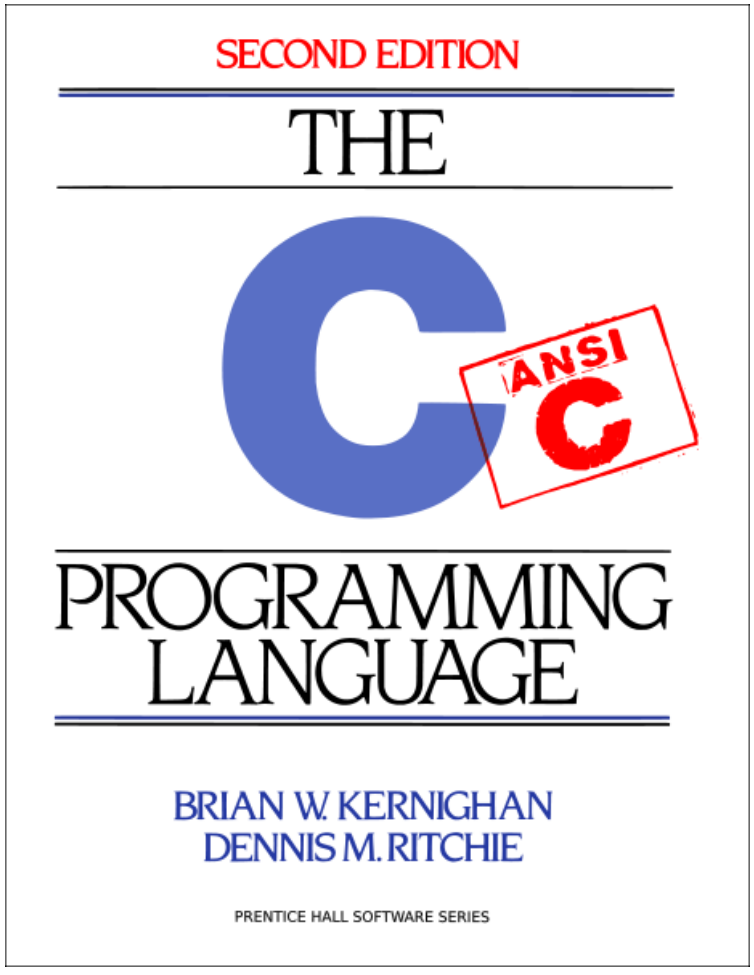
FASCHINGBAUER

Overview

FASCHINGBAUER

The Book (1)

FASCHINGBAUER



The Definitive Book

- Brian W. Kernighan, Dennis M. Ritchie
- First edition 1978 → "K&R" C
- Second (and most recent) edition 1988 → ANSI C
- Most recent standard: ISO/IEC 9899:2011 → "C11"

The Book (2)

FASCHINGBAUER

- C is a “small” language
- Few central concepts → simple — theoretically at least
- Complexity comes from the power of handling raw memory
- The book is *didactically perfect*
- Why should a C course be different?
- → This course follows the book (loosely)

The Beginning

FASCHINGBAUER

There was nothing ...

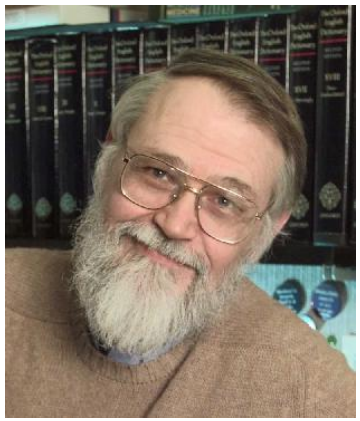
- ... but the PDP-11
- ... and a couple of cool guys
- Brian Kernighan, Dennis Ritchie → C
- Ken Thompson, Dennis Ritchie → first UNIX
- Ken Thompson → first Shell

The rest is history!

Cool Guys and Their Hobby

FASCHINGBAUER

Kernighan und Ritchie

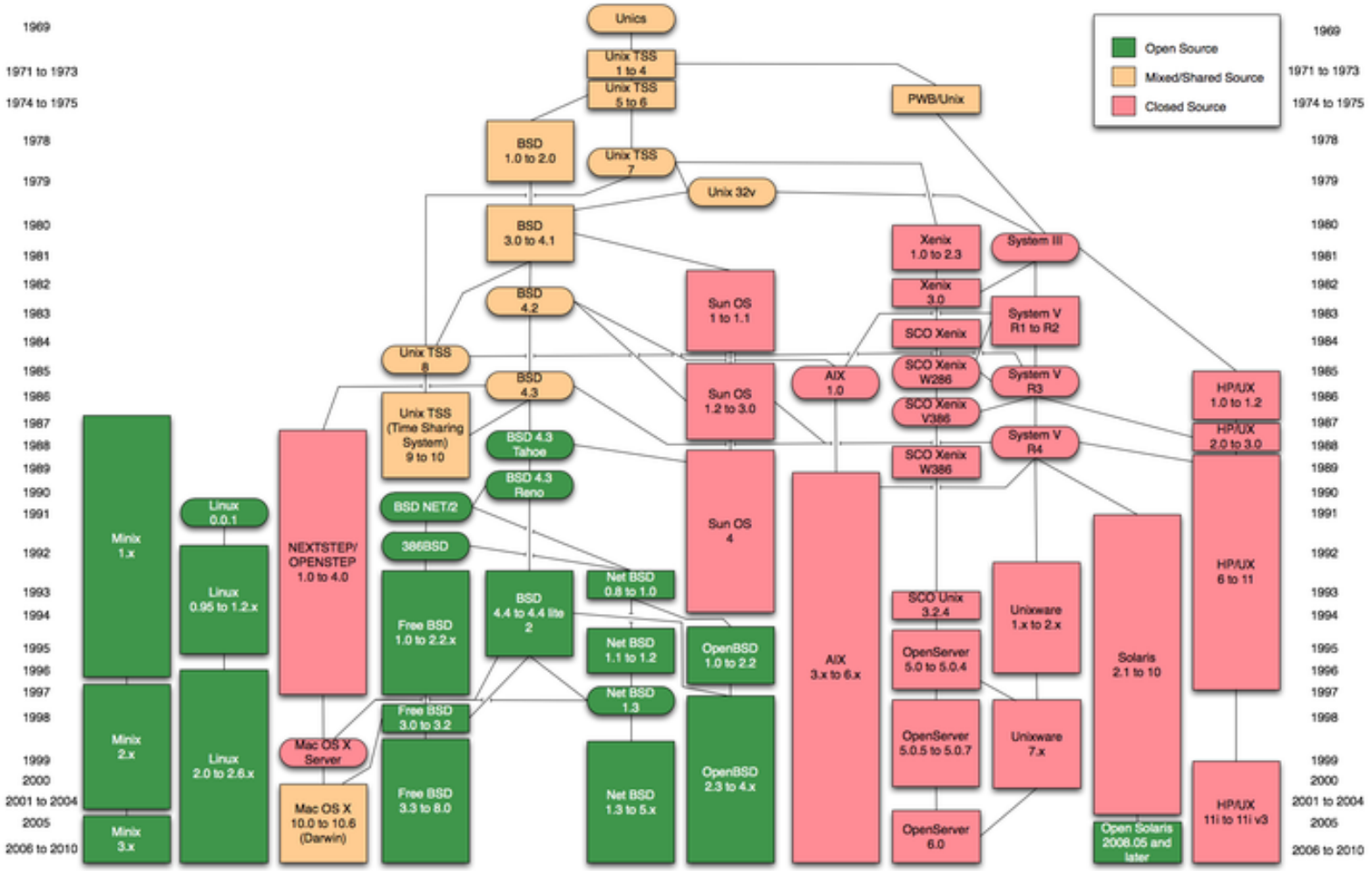


Brian Kernighan

Dennis Ritchie †2011

History of UNIX

FASCHINGBAUER



Overview

FASCHINGBAUER

My First Program (1)

FASCHINGBAUER

hello.c

```
#include <stdio.h>

void main(void)
{
  printf("hello, world\n");
}
```

Build and execute

```
$ gcc hello.c
$ ./a.out
hello, world
```

What we see ...

- A program consists of *functions* and *variables*
- Functions consist of *statements*
- Function call (`printf()`) is a statement
- `main()` is special
- Building appears simple but isn't
- `stdio.h`??

My First Program (2)

FASCHINGBAUER

```
#include <stdio.h>
```

Declarations from the *Standard IO Library* (for `printf()`)

```
void main(void)
```

Definition of `main()`. Required for a program.

```
{
    printf("hello, world\n");
}
```

Body of `main()`. *Calling* `printf()` with a *string* parameter/argument. `\n` is the *newline* character.

Character Arrays — Strings

FASCHINGBAUER

Strings are special in C

- *Character*: ...
- *String*: zero terminated character array
- Escape sequences, e.g. `\n` (newline), `\t` (tabulator), `\"`, `\0` (*null*)

`"hello,world\n"` corresponds to ...

h	e	l	l	o	,	w	o	r	l	d	\n	\0

Overview

FASCHINGBAUER

My Second Program (1)

FASCHINGBAUER

```
#include <stdio.h>

/* Fahrenheit/Celsius Table
   0 - 300, step 20 */
void main(void) {
    int fahr, celsius;
    int lower = 0, upper = 300, step = 20;

    fahr = lower;
    while (fahr <= upper) {
        celsius = 5 * (fahr - 32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

My Second Program (2)

FASCHINGBAUER

```
/* ... */
```

Comment (can span multiple lines)

```
int fahr, celsius;
```

Variable *definition*. Must come at the beginning of a *block*.

```
int lower = 0, upper = 300,
    step = 20;
```

Variable *definition* and *initialization*

My Second Program (3)

FASCHINGBAUER

```
while (fahr <= upper) {
    ...
}
```

Loop: *"While condition holds, execute body"*
Condition: `fahr` *is less or equal* `upper`

```
celsius = 5 * (fahr - 32) / 9;
```

Usual arithmetic (*expression*) → *usual operator precedence rules*

- Careful: *integer division brutally truncates decimal places!*
- More natural but always 0: `5/9 * (fahr-32)`

My Second Program (4)

FASCHINGBAUER

```
printf("%d\t%d\n", fahr, celsius);
```

- *Formatted output*
- → number of arguments can vary (?)
- %d obviously means "integer"
- **Important:** `printf()` is not part of the core language, but rather an ordinary *library function*
- → *standard library*

More Datatypes

FASCHINGBAUER

`int`	Integer, nowadays mostly 32 bits wide
`float`	Floating point number, mostly 32 bit
`char`	Single character (one byte, generally)
`short`	Smaller integer
`double`	*double precision* variant of `float`

- Width and precision of all datatypes is *machine dependent*!
- Compound datatypes: arrays, structures, ... ($\rightarrow$ later)

Exercises

FASCHINGBAUER

Following ugliness comes to mind:

- The output is not justified. (Hint: the format string "%6d" creates a 6 character wide right-justified field.)
- Integer arithmetic is inappropriate. Temperature conversion are better done in floating point.

Overview

FASCHINGBAUER

for: Loop Simplification (1)

FASCHINGBAUER

`while` **loop approach:**

- Fixed number of runs
- Counter is initialized
- Termination condition is evaluated
- Counter is incremented

This can be done simpler!

for: Loop Simplification (2)

FASCHINGBAUER

```
for (initialization; condition; step)
    ...
```

- *initialization* is evaluated exactly once — before entering the loop
- *condition* is evaluated everytime before the loop body is entered. *false* → loop termination
- *step* is evaluated after the loop body, and before the condition

initialization, *condition* and *step* are regular *statements*

Second Program, revisited

FASCHINGBAUER

```
#include <stdio.h>

void main(void)
{
    int fahr;
    for (fahr = 0; fahr <= 300; fahr = fahr+20)
        printf("%6d\t%6.2f\n", fahr, 5.0/9 * (fahr - 32));
}
```

- Wherever there can be a variable, there can be an *expression*
- A *block* ({ and }) is only necessary when the loop body consists of multiple *statements*
- `5.0/9 * (fahr - 32)` is `float` because `5.0` is

Exercise

FASCHINGBAUER

- Use a `for` loop to compute the temperature table. Do this backwards, 300 down to 0.

Overview

FASCHINGBAUER

C Preprocessor: Symbolic Constants

FASCHINGBAUER

One does not write number literals in a program!

- Inflexible
- Unreadable (a matter of taste though)
- Leads to duplicated code

C Preprocessor replaces symbols with arbitrary strings → *Macros*

```
#define LOWER 0
#define UPPER 300
#define STEP 20
```

Exercise

FASCHINGBAUER

- Modify the temparature table to use macros!

Overview

FASCHINGBAUER

The Outside World

FASCHINGBAUER

`stdio.h`: **functions and constants for I/O**

- Standard input and output
- File I/O
- Formatted
- Buffered

Most simple ones first:

```
int c;
c = getchar();
putchar(c);
```

cat for the Poor (1)

FASCHINGBAUER

```
#include <stdio.h>

void main(void)
{
    int c;

    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```

cat for the Poor (1)

FASCHINGBAUER

```
while (c != EOF)
```

- EOF — *End-of-File*
- != — not equal

But ugly code duplication: getchar() called twice

Abhilfe

```
while ((c = getchar()) != EOF)
    putchar(c);
```

- An *assignment* is an *expression* $\Longrightarrow$ has a value
- Caution: braces!

More Examples ...

FASCHINGBAUER

Counting input characters

```
long nc = 0;

while (getchar() != EOF)
    ++nc;
```

- ++: increment operator
- `long`: long integer (64 bit, mostly)

Same with `for` loop and empty body ...

```
long nc;

for (nc = 0; getchar() != EOF; ++nc);
```

More Examples — `if`

FASCHINGBAUER

Counting lines: `'\n'` terminates a line

```
int c, nl = 0;

while ((c = getchar()) != EOF)
    if (c == '\n')
        ++nl;
```

- `if`: alright
- `==`: equality (inappropriate with floating point numbers)
- `'\n'`: character constant for newline (linefeed), ASCII 10 (0A)

if, Formally

FASCHINGBAUER

if — else

```
if (expression)
    true-statement
else
    false-statement
```

Statement can be:

- Single statement (terminated with ';')
- Multiple statements, grouped inside {}

Operators, Formally

FASCHINGBAUER

Operators for use in expressions

==	Equality
!=	Inequality
&&	Boolean AND
||	Boolean OR
!	Boolean NOT

Exercises

FASCHINGBAUER

- Write a program that reads from standard input and counts characters, words, and lines. It prints the results on standard output. (Words are separated by one or more spaces, tabulators, or linefeeds.)
- Take into account error scenarios and corner cases. For example:
  - What if input strats with a space?
  - What if there are multiple separators between two words?

Overview

FASCHINGBAUER

Arrays — Next Program (1)

FASCHINGBAUER

Nonsensical but illustrative exercise: count digits, whitespace, and others.

```
int nwhite = 0, nother = 0;
```

Counter for whitespace and rest

```
int ndigit[10];

for (i = 0; i < 10; ++i)
    ndigit[i] = 0;
```

10 counters for digits 0..9.
Attention: Indexes start at 0

Arrays — Next Program (2)

FASCHINGBAUER

```
while ((c = getchar()) != EOF)
    if (c >= '0' && c <= '9')
        ++ndigit[c-'0'];
    else if (c == ' ' || c == '\t' || c == '\n')
        ++nwhite;
    else
        ++nother;
```

Matter of style ...

- There is only `if` and `else`
- No `elif` (as in Python, for example)
- $\Longrightarrow$ second `if` is *nested*

Exercise

FASCHINGBAUER

- For every possible character, count the number of occurrences in the input. At program termination (end of file), print a histogram as in the example below. Printable characters are output as-is, nonprintable characters are output as their ASCII values.

```
0   ... |
1   ... |
.
'a'... |xxxxxxxxxxxxxxxxxxxxxxxxxx
'b'... |xxxxxxxxxxxxxxxxxx
.
.
```

Overview

FASCHINGBAUER

Functions

FASCHINGBAUER

Function (subroutine, procedure): why?

- Externalizing code $\rightarrow$ multiple use
- Program structure
- Readability
- $\rightarrow$ Key to modularization

How?

- No difference between *function* and *procedure*
- Function call can be used as value (is an expression)
- *Except* return type is `void`

```
#include <stdio.h>

int power(int base, int n)
{
    int p = 1;
    while (n--)
        p *= base;
    return p;
}

void main(void)
{
    int i;
    for (i = 0; i < 10; ++i)
        printf("2^%d = %d\n", i, power(2, i));
}
```

A Nonsensical Example (1)

FASCHINGBAUER

```
int power(int base, int n)
```

- Function name
- Names *and types* of parameters
- Return type

- Parameters are *local* to function
- Parameter names only relevant inside function
- No conflicts with the outer world
- → caller may use name `base` and `i`

A Nonsensical Example (2)

FASCHINGBAUER

```
int p = 1;
```

Local variable

```
while (n--)
```

`n--` ... *Post increment*: expression's value is `n`'s value *before* increment

```
p *= base;
```

Shorthand for `p = p * base;`

```
return p;
```

Value of the function as seen by the caller

Definition vs. Declaration (1)

FASCHINGBAUER

To generate a function call, the compiler wants to know its *prototype* → error checks

- Number of parameters
- Types of parameters
- Return type

Historical baggage: implicit function declarations → best avoided using function declarations

- `-Wimplicit`: warning issued when function called without declaration
- `-Werror`: treat warnings as errors → hygiene

Definition vs. Declaration (2)

FASCHINGBAUER

Declaration: declares prototype without giving a definition

```
int power(int base, int n);
```

"I promise that the function will have this prototype, please check"

- Definition can be given later
- ... in the same file, after the call
- ... in a different file ($\rightarrow$ later)

By Value / By Reference

FASCHINGBAUER

Parameters are only passed by value

- Function receives a *local copy* of the caller's value
- Modifications not visible to the caller
- Pass by reference $\Longrightarrow$ pointers (later)

```
int power(int base, int n)
{
    while (n--) ...
}
```

Caller does not see modifications to `n`

Exercise

FASCHINGBAUER

- Modify `power.c` to only *declare* `power()` before the call to it. Give the implementation after the call, below `main()`.

Overview

FASCHINGBAUER

Strings: Mistake by Design?

FASCHINGBAUER

- Only what is necessary is built-in in C
- From today's point of view C is *the* language for hardware-oriented programming
- Invented to keep UNIX portable, independent from PDP-11 assembler
- $\rightarrow$ C itself is the language core — everything else belongs in *libraries*

Contradiction:

- Language core knows what string literals are
- 7-bit ASCII sufficed at that time $\rightarrow$ no multibyte character sets, no need for Unicode
- *But*: much later somebody claimed that "640K is enough"

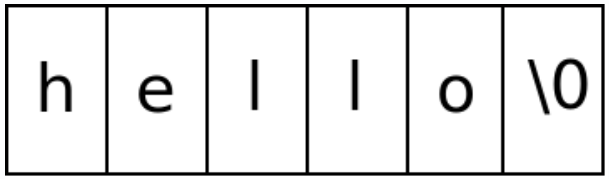
Strings: Definition

FASCHINGBAUER

String $\Longleftrightarrow$

- Array of characters ...
- ... terminated by a “null” character

```
char a_string[] = "hello";
```



Strings: Library Functions

FASCHINGBAUER

Functions from the *standard library*

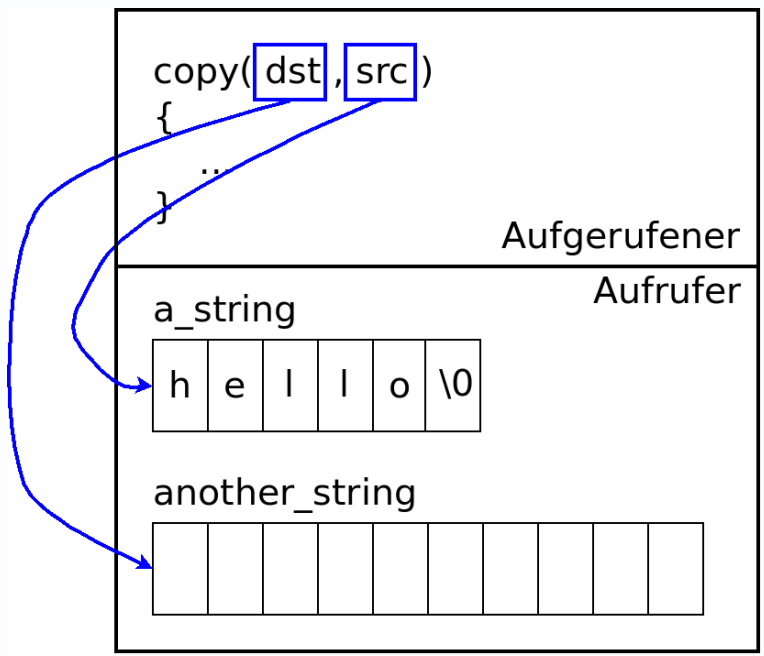
- `strlen(const char[])`
- `strcpy(char dest[], const char src[])`
- `strncpy(char dest[], const char src[], int maxlen)`
- `strcat(char dest[], const char src[])`
- `strncat(char dest[], char src[], int n)`
- `strcmp(const char lhs[], const char rhs[])`
- `strncmp(const char lhs[], const char rhs[], int maxlen)`
- Many more $\rightarrow$ `man -s 3 string`

Strings as Parameters

FASCHINGBAUER

Strings (like *arrays* in general) are passed as *pointers*
$\Longrightarrow$ *Modifications visible to the caller*

```
char a_string[] = "hello";
char another_string[10];
...
copy(another_string, a_string);
```



Strings: Dangers

FASCHINGBAUER

Low level definition leads to errors

- Copy: not enough memory allocated to hold the copy
- Forget to null-terminate when composing strings by hand
- ... many many more ...

Exercise

FASCHINGBAUER

- Write a (nonsensial) program that ...
  - Reads standard input line by line. To do so, implement a function `read_line(char line[], int maxlen)` (which internally uses `getchar()`)
  - Outputs the longest line at end of input.

Overview

FASCHINGBAUER

Lifetime

FASCHINGBAUER

Two classes of variables ...

- Local ("automatic"). Lifetime (and visibility) is confined to the function call.
  - All variables that we had so far are *automatic*
- Global ("extern"). Lifetime $\Longleftrightarrow$ program

Another point of view: *visibility* $\rightarrow$ later

Global Variables

FASCHINGBAUER

Global variables are evil

- They obscure program logic
- Side effects through unnecessary persistence
- Global *constants* are ok

There are use cases though ...

- Not a use case: laziness

Global Variables: How?

FASCHINGBAUER

- Functions (code) can only be defined in global scope
- Have access to *local variables* and *other global objects*

```
int i;

void f(void)
{
    printf("%d\n", i);
}
```

Overview

FASCHINGBAUER

Type System

FASCHINGBAUER

A type system is in place, but it is complicated ... at least with respect to built-in datatypes like integers and floating point numbers

- `signed` and `unsigned` variants
- `long` and `short` variants
- *Signedness* of `char` is machine dependent (i.e. *undefined*)
- Implicit type conversions
- Sign propagation
- ... and lots more ...
- $\rightarrow$ it is very important to understand how and why
- ... and to be defensive!

Overview

Variable- and Function Names

FASCHINGBAUER

By law: `[A-Za-z_]+[A-Za-z0-9_]*`

- Must start with letter or '_'
- Next may come digits
- Reserved names (z.B. `while`) not allowed

Examples

```
int _;
char c;
int c_89;
float _avg_temp; /* careful! */
int 1i; /* Error */
```

Be defensive: the standard states that names starting with underscores are reserved for standard libraries

Overview

FASCHINGBAUER

Standard Data Types And Their Sizes

FASCHINGBAUER

C knows about the following base types

- `char`: one byte in the machine's character set (mostly ASCII, nowadays)
- `int`: integer, as the processor architecture sees fit (nowadays 32 bits, mostly)
- `float`: *single-precision* floating point
- `double`: *double-precision* floating point

Attention: C does not specify the width of any of these types! ("machine dependent")

Integer Variants ("Qualifiers")

FASCHINGBAUER

Width modification

- `short int` (abbrev: `short`)
- `long int` (abbrev: `long`)
- `long double`

Helpers

- `CHAR_BITS`: a macro, bits per character (generally 8, nowadays)
- `sizeof`: operator, width in characters

Signs (for all integer types)

- `signed`
- `unsigned`

Widths

FASCHINGBAUER

Integer widths

Type	x86	amd64	arm
`char`	8	8	8
`int`	32	32	32
`short`	16	16	16
`long`	32	64	32

`<stdint.h>`

`int8_t`	`uint8_t`
`int16_t`	`uint16_t`
`int32_t`	`uint32_t`
`int64_t`	`uint64_t`

When to use which ...

- It depends (as always)
- Program flow (loop counters etc.) are natural integers (preferably `int` or `unsigned int`)
- In memory data structure where memory is tight: `short`
- Protocols, persistence: `<stdint.h>`

Overview

FASCHINGBAUER

Constants and Types

FASCHINGBAUER

Question: which type has e.g. `42`**?**

→ some more rules follow ...

`42, 052, 0x2A, 0b010010`	`int`
`42l, 42L`	`long`
`123.456f, 123.456F`	`float`
`123.456`	`double`
`123.456l, 123.456L`	`long double`
`'a', '\141', '\x61'`	`char`
`'\n'`	`char`

Character Constants: *Escape Sequences*

FASCHINGBAUER

\a	"Alert"
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage Return
\t	Horizontal TAB
\v	Vertical TAB
\\	Backslash
\?	Question mark
\'	Single Quote
\"	Double Quote
\ooo	Octal char value
\xhh	Hexadecimal char value

String Constants

FASCHINGBAUER

String $\Longleftrightarrow$ array of characters, terminated by null-byte

```
char hello[] = "hello,world\n";
char hello[] = "hello," "world\n";
char hello[] = "hello,"
   "world\n";
```

- Concatenated by compiler
- $\rightarrow$ String literals may span multiple lines

h	e	l	l	o	,	w	o	r	l	d	\n	\0

Character vs. String Constants

FASCHINGBAUER

Easily confused:

```
if ('x' == "x") { /* compiler error */
    ...
}
```

- `'x'` is a character
- `"x"` is a character array (a *string*)

Symbolic Constants (1)

FASCHINGBAUER

Preprocessor constants: the *good old* way to express symbolic constants

```
#define JAN 0
#define FEB 1
#define MAR 2
...
```

- Preprocessor replaces all occurences in text
- Often not desired
  - too brutal/stupid
  - alternative: manual maintenance of values $\rightarrow$ error prone

Symbolic Constants (2)

FASCHINGBAUER

Enumeration is often more appropriate

```
enum month {
    JAN,
    FEB,
    MAR,
    ...
};
```

- Value has integer type
- *Value* is irrelevant, only comparison is
- → `switch` statement

Overview

FASCHINGBAUER

Definitions und Initialization

FASCHINGBAUER

Variables must be known before they can be used

Explicit initialization

```
int lower, upper, step;
char c;

lower = 0;
upper = 300;
step = 20;
```

Implicit initialization

```
int lower = 0, upper = 300, step = 20;
char c;
```

Initialization of *Automatic* Variables

FASCHINGBAUER

Automatic variables

- Defined inside a function (at the beginning of a *block*)
- Initialized *at runtime* — everytime the code runs
- $\Longrightarrow$ arbitrary expressions possible

```
void some_function(void)
{
    /* draw random number out of 0..9 */
    int some_variable = random() % 10;
    ...
}
```

Initialization of *Global* Variables

FASCHINGBAUER

Global variables

- Defined in *global scope*
- Initialized *before program start*
- $\Longrightarrow$ only constant expressions possible (calculated at compilation time, by the compiler)

```
const double pi = 3.1415926535897932;
double some_nonsensical_number = pi / 2;
```

```
const char msg[] = "hallo";
char msg[] = "hallo"; /* possible compiler warning */
```

Overview

FASCHINGBAUER

Arithmetic Operators (1)

FASCHINGBAUER

Arithmetic Operators and Operands

Operator	Meaning	Operand type
*	multiplication	integer, floatingpoint
/	division	integer, floatingpoint
%	modulo	integer
+	addition	integer, floatingpoint
-	subtraction	integer, floatingpoint

Attention: integer division truncates!

Arithmetic Operators (2)

FASCHINGBAUER

Precedence rules

- Multiplication, division, modulo precede addition, subtraction
- Operators with same precedence are associated *from left to right*

30 / 2 + 1	(30 / 2) + 1	precedence
1 + 30 / 2	1 + (30 / 2)	precedence
1 - 2 + 3	(1 - 2) + 3	left-associative
30 / 3 % 2	(30 / 3) % 2	left-associative

Overview

FASCHINGBAUER

Relational Operators (1)

FASCHINGBAUER

Operator	Meaning	Operand type
>	greater	integer, floatingpoint
>=	greater equal	integer, floatingpoint
<	less	integer, floatingpoint
<=	less equal	integer, floatingpoint
==	equal	integer, floatingpoint
!=	not equal	integer, floatingpoint

Attention: == and != is *legal* for floatingpoint numbers, but not what you want!

Relational Operators (2)

FASCHINGBAUER

Precedence rules

1. All relational operators are preceded by arithmetic operators
2. >, >= ,<, <=
3. ==, !=

- Operators with equal precedence are associated from *left to right*

So what does that mean?

3 - 1 == 2	Arithmetic has precedence
'X' != 'U' == 1	It is true that 'X' is not 'U'
1 == ('X' != 'U')	Same, explicitly precedented
3 < 1 == 0 == 1	It is true that 3 is not less than 1
0 == 1 < 2	*What?!*

Logical (*Boolean*) Operators

FASCHINGBAUER

Logical expressions

&&	AND
||	OR

Precedence rules

- Boolean operators bind less strong than relational and arithmetic operators
- "&&" precedes "||"
- Operators with equal precedence are associated from *left to right*

Boolean Operators: Short-Circuit

FASCHINGBAUER

Short-circuit calculation

- Boolean expressions are only evaluated to the point where their truth value is known
- $\rightarrow$ Elegant and (for beginners at least) unreadable constructs

Counting leading blank lines

```
int c, num_lf = 0;

while ((c = getchar()) != EOF && c == '\n' && ++num_lf);
```

Overview

Implicit Type Conversions

FASCHINGBAUER

Bad news: C does not care much about widths and signs

- Assignment to narrower types simply cuts off
- Sign propagation is undefined
- Sign may change across signed/unsigned assignments
- $\rightarrow$ History is full of integer overflow bugs, sign bugs etc.
- GCC (and other compilers) has options that warn on possible type-bugs (can be very loud though)

Rules are not easy to comprehend — especially the "Why" $\rightarrow$ Examples ...

Sign Bugs

FASCHINGBAUER

Unsigned to signed, same width

```
unsigned int ui = 4294967295U;
int i = ui;
```

0xffffffff in 2's complement
→ i == -1

the other way around: signed to unsigned

```
int i = -1;
unsigned int ui = i;
```

-1 is 0xffffffff in 2's complement
→ ui == 4294967295U

This is **desired behavior from the very beginning** → no compiler error, no compiler warning!

- -Wsign-conversion (more global: -Wconversion)

Truncation

FASCHINGBAUER

```
unsigned long ul = 4294967296U;
unsigned int ui = ul;
```

Unscrupulous conversion (by brutal truncation) of a 64 bit number (0x100000000) to a 32 bit number
→ ui == 0

- -Wconversion

Sign Propagation

FASCHINGBAUER

```
char c = '\310';
int ic = c;
```

char is signed on x86_64
c == -56

- -Wconversion

Conversion Using Operators

FASCHINGBAUER

Hard rule: if an operator gets passed different types, then the "weaker" is converted to the "stronger" — the result is of the "stronger" type
What does that mean (disregarding `unsigned`):

- If one operand is `long double`, then the other is converted
- else, if one is `double`, ...
- else, if one is `float`, ...
- else, `char` and `short` are converted to `int`
- $\Longrightarrow$ `int` *is default type for arithmetic operations*

Conversion and unsigned (1)

FASCHINGBAUER

Hard rule: there is no hard rule. Well almost: when mixing `unsigned` and `signed` integers of the same width, then `signed` is converted to `unsigned` (Gosh!)
Additionally: widths are hardware defined!

```
-1L < 1U
```

True: `1U` becomes `1L`. `-1U` (`unsigned` 32) is less than the "stronger" `-1L` (`signed` 64), and fits in `signed` 64 *losslessly*

```
-1L < 1UL
```

False: `-1L` (`signed` 64) becomes `unsigned` 64, as dictated by the right hand side.

This is **desired behavior from the very beginning** → no compiler error, no compiler warning!

Conversion and unsigned (2)

FASCHINGBAUER

Beware of mixing!

- Not a problem if the `signed` part can never become negative
- Big problem otherwise!

```
int x;
unsigned int y;

if (x < y) ...
```

```
$ gcc -Wsign-compare ...
warning: comparison between signed and unsigned integer expres
```

Compiler Warnings

FASCHINGBAUER

All that is desired behavior (read: historical baggage) → compiler warnings have to be explicitly enabled

`-Wsign-conversion`	Sign could change
`-Wconversion`	Value and sign ...
`-Wsign-compare`	Comparison with mixed signed value
`-Wtype-limits`	E.g. `if (ui >= 0) ...`
`-Wall`	Selection of "good" warnings
`-Wextra`	... more good warnings
`-pedantic`	Does not hurt
`-Werror`	Anti-Sloppiness: *warnings become errors*

General advice: the more the better!

Last Warning

FASCHINGBAUER

C's datatypes are immensely hazardous. More hazardous is, though:

- Overengineering
- Messy design
- Loosing control over one's data structures
- Not knowing ranges of variables
- Not being open to program modification

Forced Conversion — Cast

FASCHINGBAUER

Should an automatic conversion be identified as being wrong (e.g. because the compiler warns), it can be *overridden* ...

```
int x;
unsigned int y;

if (x < (signed)y) ...
```

Overview

FASCHINGBAUER

Confusion: ++, --

FASCHINGBAUER

Increment- and decrement-Operators with a subtle difference

```
i = 5;
```

Operator	Name	Value of i afterwards	Value of expression
`++i`	Pre-Increment	6	6
`i++`	Post-Increment	6	5
`--i`	Pre-Decrement	4	4
`i--`	Post-Decrement	4	5

Confused to perfection (1)

FASCHINGBAUER

No confusion: copying string while ignoring all occurences of character c ...

```
void copy_and_omit(char dst[], const char src[], char c)
{
    int i = 0, j = 0;

    while (src[i] != '\0') {
        if (src[i] != c) {
            dst[j] = src[i];
            j = j + 1;
        }
        i = i + 1;
    }
    dst[j] = '\0';
}
```

Confused to perfection (2)

FASCHINGBAUER

Real men complain:

- *So many lines for a trivial thing?*
- *Multiple indexing does not perform!*

```
void copy_and_omit(char dst[], const char src[], char c)
{
    int i = 0, j = 0;
    char cur;

    while ((cur = src[i++]) != '\0')
        if (cur != c)
            dst[j++] = cur;
    dst[j] = '\0';
}
```

Übungen

FASCHINGBAUER

- Schreiben Sie eine Funktion (und das dazugehörige aufrufende Programm), die wie das zuvor gesehene Beispiel einen String kopiert, aber, anstatt ein bestimmtes Zeichen einfach auszulassen, mehrfache Vorkommen des Zeichen auf eines reduziert!
- Statten Sie die Funktion mit Überlaufschutz aus! Dabei erhält die Funktion einen extra Parameter, der angibt, wie gross der Ziel-String ist.

Overview

FASCHINGBAUER

Bit Manipulation

FASCHINGBAUER

C is a hardware language $\Longrightarrow$ many operators to manipulate individual bits

&	bitwise AND
|	bitwise OR
^	bitwise XOR
<<	shift left
>>	shift right
~	bitwise invert

Why?

- Manipulating hardware registers
- Saving space (e.g. persistence, protocols)
- ...

Bitwise AND and OR

FASCHINGBAUER

Extract/mask bits

```
      0x4b   01001011
 &    0x0c   00001100
-----------------------
      0x08   00001000
```

Add bits

```
      0x4b   01001011
 |    0x0c   00001100
-----------------------
      0x4f   01001111
```

Bitweises XOR

FASCHINGBAUER

Exclusive OR

```
    0x4b   01001011
^   0x0c   00001100
-------------------
    0x47   01000111
```

Shift Left

FASCHINGBAUER

0x03	<< 2	00000011
0x0c		00001100
0x03	<< 7	00000011
0x80		10000000

- Filled with zeroes from right
- Bits fall off to the left

Shift Right

FASCHINGBAUER

`0x30 >> 2`	`00110000`
`0x0c`	`00001100`
`0x30 >> 5`	`00110000`
	`?????001`

- Bits fall off to the right
- `unsigned`: filled with zeroes from left
- `signed`: *machine dependent*

→ Shift operations on signed entities is nonsense anyway

Inverting ("One's-Complement")

FASCHINGBAUER

~	0x4c	01001100
	0xb3	10110011

Overview

FASCHINGBAUER

Combined Operators

FASCHINGBAUER

The long way

```
i = i + 2;
arr[j] = arr[j] + 2;
```

The short way

```
i += 2;
arr[j] += 2;
```

- Less writing ($\rightarrow$ confusion)
- Expression evaluated *only once* $\rightarrow$ performance
- Applies to +, -, *, /, %, <<, >>, &, ^, |

Overview

FASCHINGBAUER

?: — Conditional Expression (1)

FASCHINGBAUER

The long way

```
int max(int a, int b)
{
    int z;

    if (a > b)
        z = a;
    else
        z = b;
    return z;
}
```

?: — Conditional Expression (2)

FASCHINGBAUER

The short way

```
int max(int a, int b)
{
    return (a > b)? a: b;
}
```

- Saving space
- Usable as expression

Overview

FASCHINGBAUER

Summary: Operators

FASCHINGBAUER

Operators seen thus far

- Arithmetic (unary and binary)
- Comparison
- Boolean
- Bitwise
- Assignment (combined)
- Conditional expression

Operators: Precedence und Associativity (1)

FASCHINGBAUER

Operator Table

- Ordered by precedence (strongest binding first)
- Left- or right associativity at equal precedence

Attention

- Precedence und associativity often not intuitive
  - E.g.: `x | y < z` $\Longleftrightarrow$ `x | (y < z)`
- $\rightarrow$ difficult to comprehend and remember
- *Use braces to make precedence explicit*
- $\rightarrow$ for yourself and for your successor

Operators: Precedence und Associativity (2)

FASCHINGBAUER

Operator	Associativity
`()`, `[]`, `->`, `.`	left
`!`, `~`, `++`, `--`, `+`, `-`, `*`, `&`, `(type)`, `sizeof`	right (unary operators)
`*`, `/`, `%`	left
`+`, `-`	left
`<<`, `>>`	left
`<`, `<=`, `>`, `>=`	left
`==`, `!=`	left
`&`	left
`^`	left
`|`	left
`&&`	left
`||`	left
`?:`	right
`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`	right
`,`	left

More Warnings

FASCHINGBAUER

The order in which operands are evaluated is unspecified (compiler dependent)

```
x = f() + g(); /* DANGER! */
```

The order in which function parameters are evaluated is unspecified (compiler dependent)

```
f(++i, i); /* DANGER! */
a[i] = i++; /* MORE DANGER */
```

Overview

FASCHINGBAUER

Overview

FASCHINGBAUER

Statement vs. Block

FASCHINGBAUER

Single statements terminated with ";"

```
a = 1;
f(a);
```

Block ("compound statement") is a group of statements $\rightarrow$ syntactically equivalent to a single statement

```
{
    a = 1;
    f(a);
}
```

Attention: no ";"

Overview

FASCHINGBAUER

Branches

FASCHINGBAUER

If *condition* holds true, then we do *this*, else we do *that* ...

```
if (condition)
    this
else
    that
```

this und *that* are statements ...

```
if (a < 0)
    a = -a;
else {
    a = a;
    fprintf(stderr, "alright\n");
}
```

True or False? What is Truth?

FASCHINGBAUER

```
if (condition)
    ...
```

- *condition* is an *expression*
- An *expression* has a value
- In `if` (other similar statements) its value is used as condition
- 0 ... condition does not hold $\rightarrow$ *false*
- Everything else ... $\rightarrow$ *true*

`else` is optional (1)

FASCHINGBAUER

`if` *may* be followed by an `else` branch (but need not)

```
if (condition)
    if (another-condition)
        this
    else
        that
```

Ambiguity: where does the `else` branch belong?

else is optional (2)

FASCHINGBAUER

Dangling else: compiler does not care about indentation $\Longrightarrow$ *careful*!

```
if (condition)
    if (another-condition)
        this
else
    that
```

Braces required!

```
if (condition) {
    if (another-condition)
        this
}
else
    that
```

Overview

FASCHINGBAUER

Style Matters

FASCHINGBAUER

Very popular: multiple cases in a row

```
if (condition)
    this
else if (another-condition)
    that
else if (one-more)
    ...
else
    rest
```

More appropriate in such cases: `switch`

Overview

FASCHINGBAUER

Case Distinctions

FASCHINGBAUER

Problem: `if - else if ... - else`

- Much typing
- ... especially when checking for equality of integers
- Direct jump table (compiler generated) would be more efficient

if - else if vs. switch

FASCHINGBAUER

```
if (c == ' ')
    ...
else if (c == '\n' ||
         c == '\t')
    ...
else
    ...
```

```
switch (c) {
    case ' ':
        ...
        break;
    case '\n':
    case '\t':
        ...
        break;
    default:
        ...
}
```

switch

FASCHINGBAUER

- Labels must only be *constants* (and constant expressions), no *variables*
  - known at compile time
- Equality → code starting at label is executed, until the end of `switch` statement
- `break`: *fall through* otherwise
- *fall through* sometimes desired, but mostly not → careful!
- `default` label is optional

When do I use it?

- Finite number of values (e.g. states of a *state machines*)
- `switch` over `enum` without `default`: compiler can warn about missing label → very useful!

Overview

FASCHINGBAUER

while: general purpose loop

FASCHINGBAUER

```
while (condition is true)
    do something
```

- Most general looping construct
- Serves all uses
- With a couple of extra variables everything is doable
- ... in many cases complicated though

→ for, do - while

From while to for (1)

FASCHINGBAUER

Iteration over sets of elements using while

Iteration using while

```
i = 0;
sum = 0;
while (i < 100) {
    sum += i;
    ++i;
}
```

Generally

```
init-expression
while (cond-expression) {
    body-statement
    next-expression
}
```

From while to for (2)

FASCHINGBAUER

The following constructs are equivalent:

```
init-expression
while (cond-expression) {
    body-statement
    next-expression
}
```

```
for (init-expression; cond-expression; next-expression)
    body-statement
```

Plus:

- *init-expression*, *cond-expression* and *next-expression* are **optional**

for, a Little Closer

FASCHINGBAUER

```
for (i = 0, sum = 0; i < 100; ++i)
    sum += i;
```

init-expression	`i = 0, sum = 0`
cond-expression	`i < 100`
next-expression	`++i`

Comma Operator

FASCHINGBAUER

Comma operator:

- The *expression* `expr-1, expr-2` has the value `expr-2`
- The *operator* "," is left-associative
- Precedence: lowest precedenceof all operators (see operator table)

```
wert = expr-1, expr-2; /* expr-2 */
wert = expr-1, expr-2, expr-3; /* expr-3 */
wert = 1, 2, 3; /* 3 */
```

for: Infamous Idioms

FASCHINGBAUER

C is infamous for excessive compactness ...

As above, only more compact

```
for (i = 0, sum = 0; i < 100; sum += i++);
```

Infinite loop

```
for (;;) {
    mach_was();
    sleep(5);
}
```

A crash, in microcontroller terminology

```
for (;;);
```

Overview

FASCHINGBAUER

do - while: Bedingung am Ende

FASCHINGBAUER

```
do
    do-something
while (condition);
```

- Condition is checked *after* body
- $\Longrightarrow$ body is executed *at least once*

Overview

FASCHINGBAUER

break and continue

FASCHINGBAUER

Loop control apart from the condition

- `break`: terminates innermost enclosing loop or `switch`
  - Nesting: outer loop/`switch` not concerned
- `continue`: next loop iteration
  - Loop condition is checked
  - `for`: *next-expression* evaluated

```
for (i = 0; i < strlen(input); ++i) {
    if (!isprint(input[i]))
        /* do nothing for nonprintable chars */
        continue;
    error = do_something(input[i]);
    if (error)
        break;
}
```

Overview

FASCHINGBAUER

Structured Programming vs. goto

FASCHINGBAUER

Structured programming:

- Only controlled program flow
- Loops and branches
- At most `break` and `continue`
  - only one level concerned

`goto` **is the exact opposite**

- Jump statement out of 10 nested loops
- $\rightarrow$ Massacre
- Edsger Dijkstra, 1968: "Goto Considered Harmful"
- $\rightarrow$ Plea for structured programming

goto: Definition

FASCHINGBAUER

C offers easy ways to do what you want

- Dennis Ritchie: "There is no spirit of C!"
- Jörg Faschingbauer: *there is*!

Definition of `goto`:

- `goto label;`
- `label` is the name of a place inside the function
- → `label` is only *locally* visible
- Naming rules: like a variable
- `goto` can jump to arbitrary places inside the function

goto: Use Cases

FASCHINGBAUER

Manifold, but ...

- Easily shot in the foot
- Suggestion: use judiciously *error handling*
- Linux Kernel is full of it

```
int do_complicated_stuff(void)
{
    while (...) {
        ...
        for (...) {
            if (error)
                goto out;
        }
    }
    return 0;
out:
    cleanup_mess();
    return -1;
}
```

Overview

FASCHINGBAUER

Overview

FASCHINGBAUER

Nesting

FASCHINGBAUER

- **Global** objects
  - Variables
  - Functions
- **Local** objects
  - Variables
- Functions cannot be defined *locally*

```
int global;

void f(void)
{
    int local;

    /* visibility: */
    local = global;
}
```

Modularization

FASCHINGBAUER

No non-trivial program consists of only one source file $\rightarrow$ *modularization*

- Code in multiple files
- Separate compilation
- Organization in (shared) libraries
- Combining (*linking*) of separately compiled entities into an executable program
- **Re-use**: building different programs from the same *modules*

Declaration vs. Definition

FASCHINGBAUER

Declaration: compiler needs to know things ("*objects*") in order to treat them right. It doesn't need to know where in memory they are though — only the type.
Definition: setting aside memory for objects.
A definition ist eine declaration, but not vice versa.

- *Variables*
  - Until now we only *declared* und at the same time *defined* variables
  - Pure declaration possible → ("`extern`")
- *Functions*: usually only *declared* before use, and *defined* separately.

A Monolithic Program

FASCHINGBAUER

```
void g(void); /* Declaration */
void main(void)
{
    g(); /* Use */
}
void f(void); /* Declaration */
void g(void) /* Definition */
{
    printf("g()\n");
    f(); /* Use */
}
void f(void) /* Definition */
{
    printf("f()\n");
}
```

Separate Compilation

FASCHINGBAUER

f.c

```
void f(void)
{
    printf("f()\n");
}
```

g.c

```
void f(void);
void g(void)
{
    printf("g()\n");
    f();
}
```

main.c

```
void g(void);
void main(void)
{
    g();
}
```

Built like so:

```
$ gcc main.c f.c g.c
```

Overview

FASCHINGBAUER

Variables: Declaration and Definition

FASCHINGBAUER

Functions

- Functions are complex
- → usually not written on one line
- **Readability** → separate declaration and definition
- ... even when defined and called inside the same source file

Variables

- Usually written on one line → declaration *and* definition
- → no need for a *declaration*
- **But**: how does one *declare* a variable (make it known to the compiler without allocating memory), and *define* it in a different file?

Variables: Separating Declaration from Definition

FASCHINGBAUER

main.c

```
extern int g_lobal;
void print_g(void);

void main(void)
{
    g_lobal = 100;
    print_g();
}
```

g.c

```
#include <stdio.h>

int g_lobal;

void print_g(void)
{
    printf("%d\n",
           g_lobal);
}
```

Variables: Separating Declaration from Definition

FASCHINGBAUER

Compiler and linker work together

- `extern` variable declaration → explicitly marked as *declaration*
- *Compiler* does *not* set aside memory
- There is no address yet → *Compiler* cannot insert address where variable is used
- → Inserts a *reference*, to be *resolved* by the linker

Overview

FASCHINGBAUER

Declarations: Problems (1)

FASCHINGBAUER

Declarations must exactly match corresponding definitions

main.c

```
extern int g_lobal;
void print_g(void);
void main(void)
{
    g_lobal = 100;
    print_g();
}
```

g.c

```
double g_lobal;
void print_g(
    const char[] format)
{
    ...
}
```

Declarations: Problems (2)

FASCHINGBAUER

Severe bugs

- *Incorrect linkage*: perception of *user* does not match definition
- Hard to detect: no tool support — only discipline and conventions
- At best: *segmentation fault* → crash
- At worst: appears to work, but in fact doesn't

Solution

- Centralize declarations → *header files*
- `#include "g.h"`, rather than giving declarations by hand

Declarations: Solutions

FASCHINGBAUER

g.h

```
#ifndef G_H
#define G_H

extern double g_lobal;
void print_g(
    const char[] format);

#endif
```

g.c

```
// have compiler check
// declaration/definition
// consistency
#include <g.h>

double g_lobal;

void print_g(
    const char[] format)
{
    ...
}
```

main.c

Overview

FASCHINGBAUER

Lifetime and Visibility (1)

FASCHINGBAUER

Time span where a variable is alive (its address is valid)

- The time of a function call → *automatic* variable
- The entire program

Visibility of a variable

- Inside the where it is defined → *internal* Variable
- Inside the file where is is defined
- Across the entire programx

Lifetime and Visibility (2)

FASCHINGBAUER

		Lifetime	
		Function	Program
Visibility	Function	local: `int i;`	local: `static int i;`
	File	—	global: `static int i;`
	Program	—	global: `int i;`

Automatic Variables

FASCHINGBAUER

- Visibility: function
- Lifetime: function

```
void f(void)
{
    int i;
}
```

Local `static` Variable

FASCHINGBAUER

- Visible only inside the function where it is defined
- Retaines it value across function calls

  - Visible: function
  - Lifetime: program

```
void f(void)
{
    static int i;
}
```

Global `static` Variable

FASCHINGBAUER

Visible only inside the file where it has been defined; retains its value during program lifetime

- Visible: file
- Lifetime: program

```
static int i;
```

Global Variable

FASCHINGBAUER

Visible across all files; retains its value during program lifetime

- Visible: program
- Lifetime: Program

```
int i;
```

Overview

FASCHINGBAUER

The C Preprocessor: *Why*!

FASCHINGBAUER

Dirty Hack!

- Has nothing to do with the language itself
- Invented to quickly solve problems in an *ad hoc* manner — completely ignoring the language
- Brutal and stupid text replacement
- $\rightarrow$ Please use cautiously!

#include

FASCHINGBAUER

Include the content of a file at the point in the source file where `#include` is written

Two incarnations, with a subtle difference

- `#include "file.h"`: search first in the directory where the "includer" is, and then along the search path
- `#include <file.h>`: searches only along the include path

No rules, but ...

- Header files contain declarations and macros
- Header files generally include header files: have to protect themselves against *multiple inclusion* → *include guards*
- `#include` always comes near the beginning of a source file — *never inside*

Macros: Text Replacement

FASCHINGBAUER

Macro: definition of a *token* that is brutally replaced

```
#define forever for (;;)

...
    forever {
        sleep(1);
        printf("No way out!");
    }
```

Macros: Constant Definition

FASCHINGBAUER

```
#define LOWER 0
#define UPPER 300
#define STEP 20

for (i = LOWER; i < UPPER; i += STEP)
    ...
```

Better: C99 `const` Keyword

```
const int LOWER = 0;
```

→ Typed *immutable* variable

Macros: Inline Replacement as *Function Call* (1)

FASCHINGBAUER

Original problem

- Function calls are slow
- Parameter passing → copy
- Return → copy

```
#define max(a, b) (((a) > (b))? (a) : (b))
...
x = max(1, 2);
```

→ Statement is *expanded* as if it were a function call

But ...

Macros: Inline Replacement as *Function Call* (2)

FASCHINGBAUER

Braces are necessary:

```
/* #define max(a, b) (((a) > (b))? (a) : (b)) */
#define max(a, b) ((a > b)? a : b)
...
x = max(p+q, r+s);
```

brutally expands to ...

```
x = (p+q > r+s) ? p+q : r+s;
```

→ **Operator precedence massacre!**

Macros: Inline Replacement as *Function Call* (3)

FASCHINGBAUER

C99: `inline` **keyword**

```
inline int max(int a, int b)
{
    return (a > b)? a : b;
}
```

Drawback: cannot use `max()` with different types

Macros: Inline Replacement as *Function Call* (4)

FASCHINGBAUER

One more thing:

```
#define max(a, b) (((a) > (b))? (a) : (b))
...
x = max(i++, j++);
```

brutally expands to ...

```
x = ((i++) > (j++)) ? (i++) : (j++);
```

→ **Parameters are evaluated more than once!**

Include Guards (1)

FASCHINGBAUER

a.h

```
#include "c.h"
```

b.h

```
#include "c.h"
```

c.h

```
extern int g_lobal;
```

main.h

```
#include "a.h"
#include "b.h"
```

→ **Error:** multiple declaration of `g_lobal`

Include Guards (2)

FASCHINGBAUER

Solution: define a "guard" macro *by hand* (!)

c.h

```
#ifndef HAVE_C_H
#define HAVE_C_H

extern int g_lobal;

#endif
```

OMG!

- *By hand* — after all, it's got nothing to do with C
- → Bugs/errors are the logical consequence (e.g. guard macro clashes)
- → GCC Extension: `#include_once`

The C Preprocessor: Last Words

FASCHINGBAUER

- C is low-level but not stupid
- *The C preprocessor is stupid*
- C programmers can take it
- Newbies not
- Unnecessary hurdle
- $\rightarrow$ *Stupid!*

Overview

FASCHINGBAUER

Conditional Compilation: Rules

FASCHINGBAUER

Directives

`#if`	Preprocessor condition (simple arithmetics, at most)
`#ifdef`	Definedness of a macro (regardless of its value)
`#ifndef`	Not-definedness of a macro
`#else`	(no comment)
`#elif`	as opposed to C's `else if`
`#endif`	(no comment)

Operators for use with `if` and `elif`

`defined`	Definedness of a macro
`!`	Boolean NOT
`&&`	Boolean AND
`||`	Boolean OR
`==`	Equal
`!=`	Unequal

Conditional Compilation: Examples

FASCHINGBAUER

Commenting out lines

```
#if 0 /* argh, there's a bug somewhere */
     int i;
     for (i=0; i<2; i--)
         do_something();
#else
     do_something();
     do_something();
#endif
```

Multiple Conditions Combined

```
#if defined DEBUG && NUMBER == 3
    fprintf(stderr, "NUMBER equals 3\n");
#endif
```

Conditional Compilation: Last Words

FASCHINGBAUER

Conditional compilation ...

- Doesn't make code more readable
- Begs for errors
- Is quite tempting to use in a hurry

Typical uses

- Same code on multiple OS's
  - Better to extract OS-specific concepts
  - Define clear separation between OS independent and OS dependent code
  - Avoid inline `#ifdef`'s (maintenance horror)
- "Release" and "Debug" versions of the same code base
  - Again: avoid inline `#ifdef`'s
  - Define macros that expand appropriately

Macros: Spanning Multiple Lines

FASCHINGBAUER

Macro definition can only span one line → line continuation

(Extremely Nonsensical) Multiline Macro

```
#define forever(body) \
    for (;;) { \
        body; \
    }


...
int x = 1;
forever(printf("%d\n", x); ++x;);
...
```

Macros: Multiple Statements as One Statement (

FASCHINGBAUER

A Block Is Not a Statement

```
#define do_much() \
    { \
        do_this(); \
        do_that(); \
    }

...
if (42)
    do_much(); /* ERROR! */
else
    do_less();
```

Macros: Multiple Statements as One Statement (

FASCHINGBAUER

Employ a little trick ...

Making A Block Into Statement

```
#define do_much() \
    do { \
        do_this(); \
        do_that(); \
    } while (0)
```

Silence Warnings of Microsoft's C Implementation

```
__pragma(warning(push))
__pragma(warning(disable:4127))
...
__pragma(warning(pop))
```

Stringification (1)

FASCHINGBAUER

Commmon problem: output a C expression

Macro Usage in Code

```
...
WARN_IF(i>10);
...
```

Should yield on `stderr`

Appearance on `stderr`

```
WARNING: i>10
```

Stringification (2)

FASCHINGBAUER

Solution: *Stringification*

```
#define WARN_IF(expr) \
    do { \
        if (expr) \
            fprintf(stderr, "WARNING: " #expr "\n"); \
    } while (0)
```

Macro argument is used twice ...

- evaluated as C in the `if` statemant
- converted into a C string using #

Token Pasting (1)

FASCHINGBAUER

Common Problem: construct C identifiers from macro parameters

Redundant Code

```
struct command
{
    char *name;
    void (*function) (void);
};

struct command commands[] =
{
    { "help", function_help },
    { "quit", function_quit }
};
```

Token Pasting (2)

FASCHINGBAUER

Solution: *Token Pasting*

```
#define COMMAND(name) { #name, function_ ## name }

struct command commands[] =
{
    COMMAND(help),
    COMMAND(quit)
};
```

Warnings and Errors

FASCHINGBAUER

```
void inject_virus(HANDLE doomed_process)
{
#ifdef WIN32
    void *foreign_mem = VirtualAllocEx(
        doomed_process,
        0,
        8192,
        MEM_COMMIT,
        PAGE_EXECUTE|PAGE_READWRITE);
    ...
#else
#   error cannot infect foreign processes
#endif
}
```

Predefined Macros (1)

FASCHINGBAUER

`__FILE__`	Name of current input file (C string)
`__LINE__`	Current line current input file (integer)

```
#define WARN_IF(expr) \
    do { \
        if (expr) \
            fprintf(stderr, "%s:%d: WARNING: " #expr "\n", \
                __FILE__, __LINE__); \
    } while (0)
```

- Gives the position where `WARN_IF` was expanded, *not* where `WARN_IF` was defined

The C Preprocessor: Last Words

FASCHINGBAUER

Always think twice! First thought is likely wrong.

- Inline preprocessorisms *pollute code*
- Code should be kept readable and obvious
- Push down preprocessorisms into (architecture) specific places
  - Well defined *selection macros*
  - Forwarding-Headers
  - Common abstractions
- Refactor immediately when smell detected!
  - It is like the pest!

Overview

FASCHINGBAUER

Overview

FASCHINGBAUER

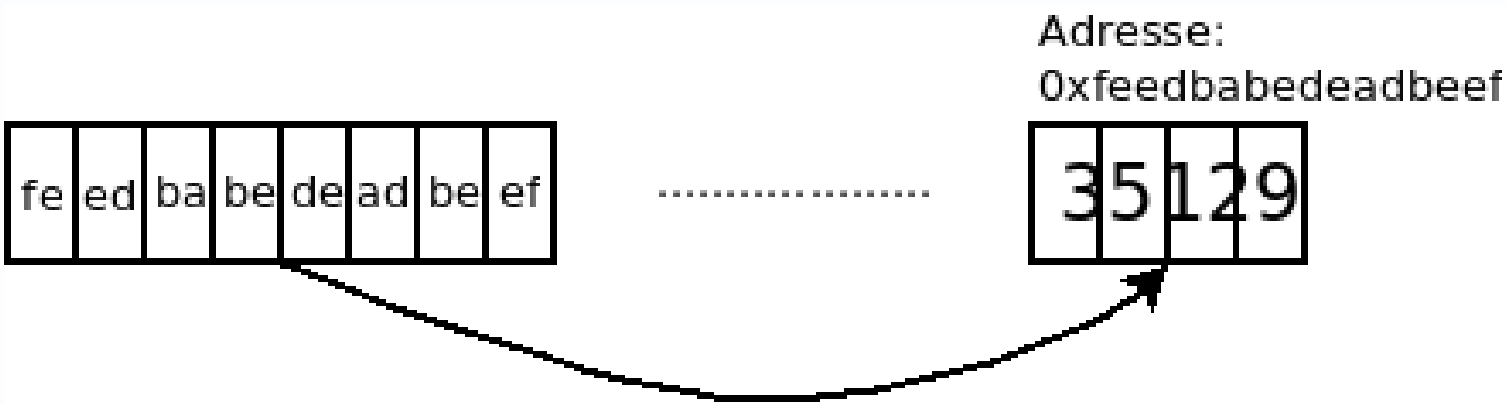
Pointers: Basics

FASCHINGBAUER

Pointer $\Longleftrightarrow$ Memory address

- Variable that *points* to another variable
- Basis for e.g. *call-by-reference*
- Simple in theory
- Practically difficult and dangerous

Pointer to integer (64 bit)



Pointer: Operators

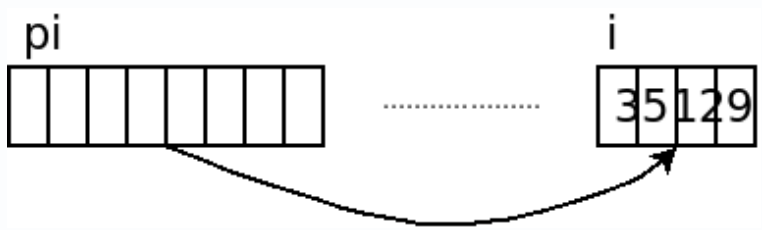
FASCHINGBAUER

Operations

- *Taking an address*: what is the address of the variable `i`?
- *Dereferencing*: what is the content of the memory location that a pointer points to?

Taking an address

```
int i = 35129;
int *pi;
pi = &i;
```



Dereferencing

```
int value = *pi;
/* value == 35129 */
```

More Examples

FASCHINGBAUER

```
int x = 1, y = 2;
int *pi; /* pointer to int */

pi = &x; /* "pi points to x" */
*pi == 1; /* true */
x = 42;
*pi == 42; /* true */
pi = &y;
*pi == 2; /* true */

*pi = *pi + 1;
*pi += 1;
y == 4; /* true */

pi = 0; /* null pointer */
```

Overview

FASCHINGBAUER

Call by Reference (1)

FASCHINGBAUER

- **Problem**: in C, parameters are passed *by-copy* — callee see *copies* of the caller's values.
- **Question**: how can I use a function to *modify* the caller's value?

```
void f(int a)
{
    a = 42;
}

void main(void)
{
    int i = 1;

    f(i);
    /* i is still 1 */
}
```

Call by Reference (2)

FASCHINGBAUER

Solution: pointer

```
void f(int *a)
{
    *a = 42;
}

void main(void)
{
    int i = 1;

    f(&i);
}
```

Exercise

FASCHINGBAUER

- Write a function `swap()` that excanges the content of two integer variables!

Overview

FASCHINGBAUER

It's Only Memory

FASCHINGBAUER

Pointers and arrays are closely related to each other

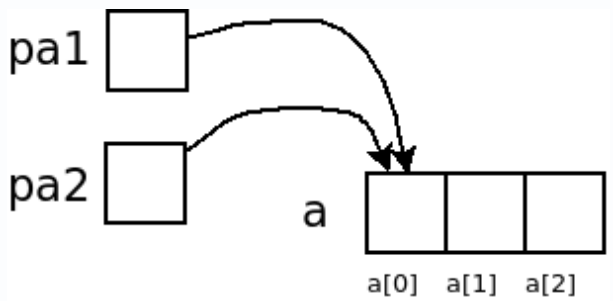
- Automatic conversion array $\rightarrow$ pointer
- Array-type function parameters are in fact pointers to the first (0-th) array element
- Index operator (`a[i]`) is *pointer arithmetic*
- *True strength of C*

```
int a[] = { 42, 1, 23 };
char str[] = { 'h', 'a', 'l', 'l', 'o', '\0' };

int *pa1 = &a[0];
int *pa2 = a;
```

Conversion Array → Pointer

FASCHINGBAUER

```
int a[3];
int *pa1 = &a[0];
int *pa2 = a;
```

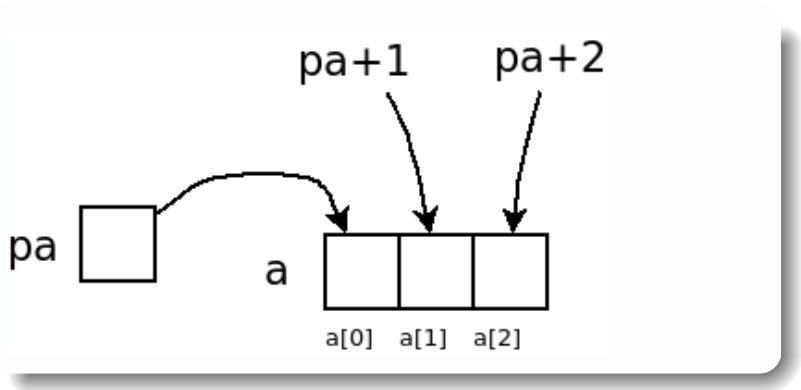


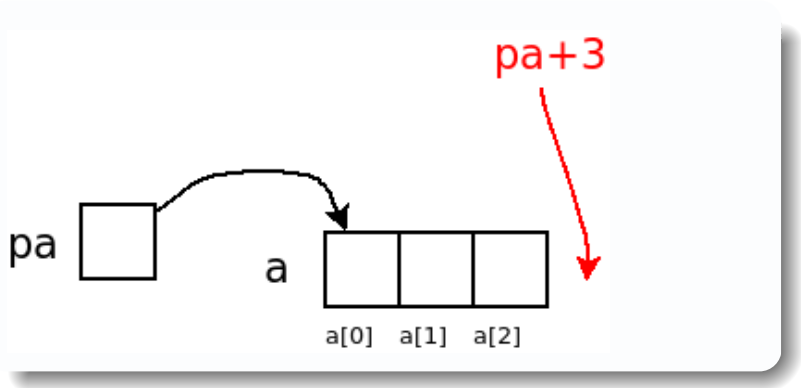
Pointer Arithmetic (1)

FASCHINGBAUER

Pointer and array index

- *Pointer + Integer = Pointer*
- Equivalent to subscript ("index") operator
- Just like subscript *there is no range check being made*
- → Errors happen
- *But*: performance!



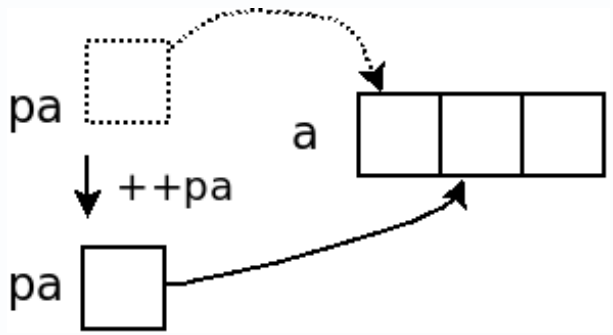


Pointer Arithmetic (2)

FASCHINGBAUER

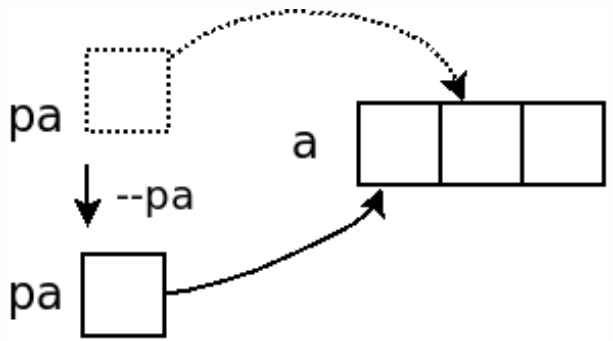
Pointer increment

```
int *pa = a;
++pa;
```



Pointer decrement

```
int *pa = &a[1];
--pa;
```

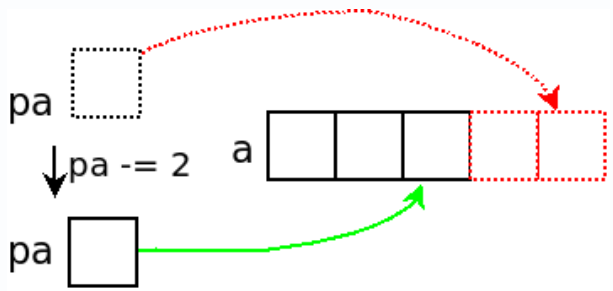


Pointer Arithmetic (3)

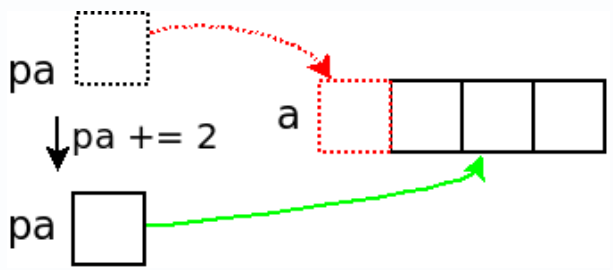
FASCHINGBAUER

Pointers don't necessarily have to point to something that is valid ...

```
*pa = a + 4;
pa -= 2;
i = *pa; /* ok */
```




```
*pa = a - 1;
pa += 2;
i = *pa; /* ok */
```

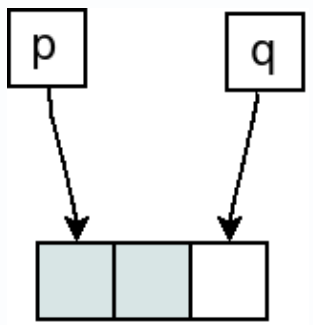


Pointer Arithmetic: Difference

FASCHINGBAUER

How many elements are there between two pointers?

```
p = &a[0];
q = &a[2];
num = q - p; /* 2 */
```



Often (C++ STL) it is done like so:

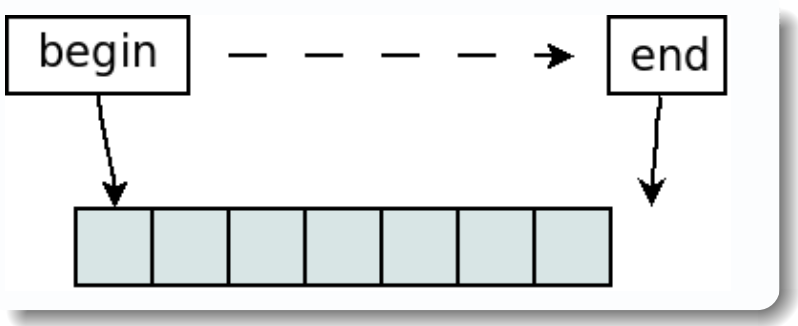
- *Beginning* of an array ("a set of elements" is the *pointer to the first element*
- *end* is *pointer to one past the last valid element*

Pointer Arithmetic: Array Algorithms

FASCHINGBAUER

Iterating over all elements of the array

```
int sum(const int *begin, const int *end)
{
    int sum = 0;

    while (begin < end)
        sum += *begin++; /* precedence? what? */
    return sum;
}
```



Beautiful, isn't it?

Pointer Arithmetic: Jump Width? (1)

FASCHINGBAUER

So far: pointer to `int` — how are arrays of other (even compound) types handled?

→ just the same!

- *Pointer* + *n*: points *n* elements further
- Type system is not stupid (only sometimes)
- Pointer know *which type* is being pointed to
- Be careful with `void` and `void*`: `sizeof(void)` is *undefined*!

Pointer Arithmetic: Jump Width? (2)

FASCHINGBAUER

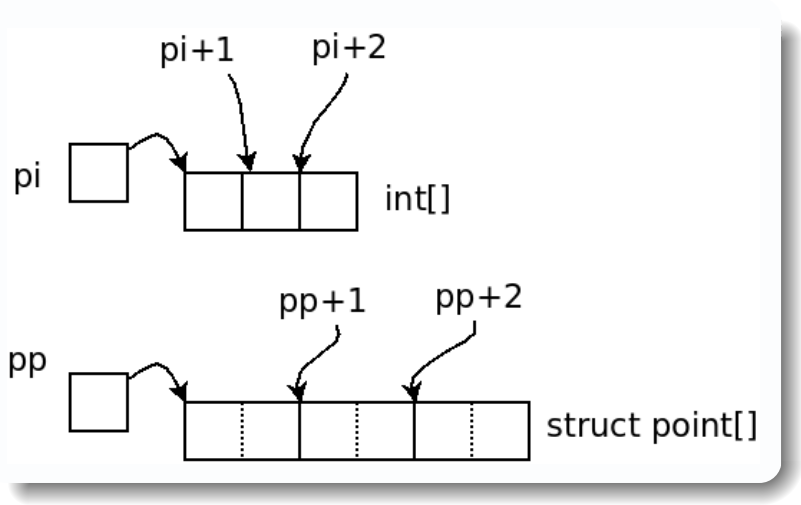
```
struct point
{
    int x, y;
};

struct point points[3], *begin, *end;

begin = points;
end = points + sizeof(points)/sizeof(struct point);

while (begin < end) {
    ...
    ++begin;
}
```

Pointer Arithmetic: Arbitrary Datatypes

FASCHINGBAUER

- *sizeof*: size (in bytes) of a type/variable

```
sizeof(int)
sizeof(struct point)
sizeof(i)
sizeof(pi)
sizeof(pp)
```



Overview

FASCHINGBAUER

`main()` can take parameters

FASCHINGBAUER

So far: `void main(void)`

- A process has an *exit status* → implicitly `int main(...)`
- → Compiler magic: `main()` is special
- C++: compiler error if `main()` does not return an `int` Returnwert hat
- No commandline arguments expected → `main(void)`

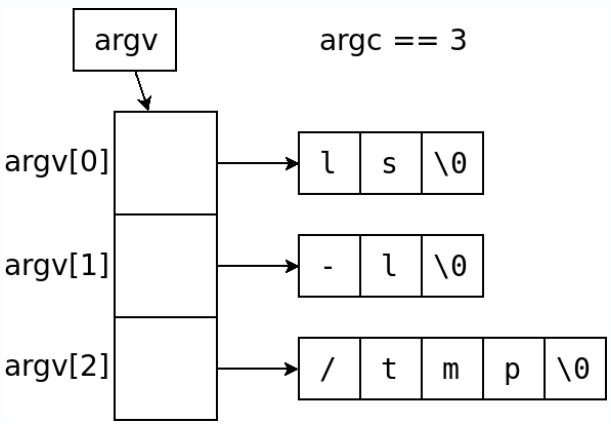
→ How are commandline arguments passed?

Commandline Arguments

FASCHINGBAUER

```
int main(int argc, char **argv)
{
    char *opt = argv[1]; /* "-l" */
    char *dir = argv[2]; /* "/tmp" */
    ...
}
```

ls -l /tmp



Overview

FASCHINGBAUER

Overview

FASCHINGBAUER

`struct`: compound datatypes

FASCHINGBAUER

So far we had ...

- Scalar datatypes: `int`, `float`, ...
- Pointers
- Now for some ... *design*

Fantasy:

- Pointers give us power to do more
- How do we build more complex data structures?
  - Linked lists
  - Balanced trees
  - ...

struct: how?

FASCHINGBAUER

Short and to the point ...

```
/* type declaration - no memory set aside */
struct point
{
    int x;
    int y;
};

/* set aside memory for two points */
struct point p1, p2;
```

- *New type:* `struct point`
- Used just the same as other types

Usage

FASCHINGBAUER

Operations

- Initialization
- Copy ...
  - Assignment
  - Parameter passing
  - Return from function
- Member access

```
/* initialization */
struct point p = { 42, 7 };

/* member access */
p.x = 1;

/* assignment */
p2 = p;
```

Nested Structures

FASCHINGBAUER

Nesting

- Nesting is possible
- But: structures become large through nesting
- *Call-by-value* (and `return`) makes *copies*!

```
struct rect
{
    struct point p1;
    struct point p2;
};
```

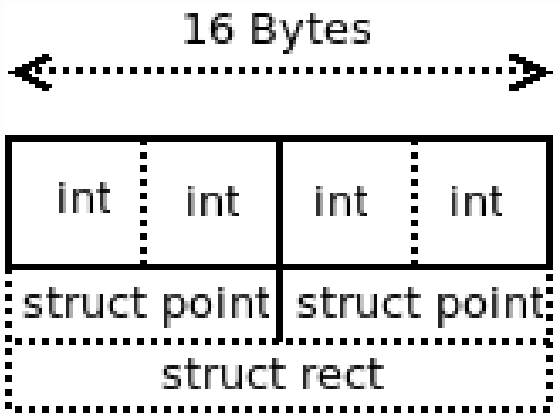
Memory Layout

FASCHINGBAUER

Structure $\Longleftrightarrow$ flat memory

- Linear sequence of bytes
- *Copy* (assignment, parameter, return) of memory is made

```
struct rect
{
    struct point p1;
    struct point p2;
};
```



Recursive Structures?

FASCHINGBAUER

```
struct xxx
{
    struct xxx x;
};
```

- That particular structure would be infinitely large
- → Compiler error
- → *Pointers*

Overview

FASCHINGBAUER

Parameters and Return (1)

FASCHINGBAUER

Returning entire structures

"Constructor"

```
struct point makepoint(int x, int y)
{
    struct point p;

    p.x = x;
    p.y = y;
    return p;
}
```

Parameters and Return (2)

FASCHINGBAUER

Entire structure as parameter

```
struct point addpoints(struct point lhs, struct point rhs)
{
    lhs.x += rhs.x;
    lhs.y += rhs.y;
    return lhs;
}
```

Question: does the caller see the modification of `lhs`?

Parameters and Return (3)

FASCHINGBAUER

Pointers to structures ("call by reference")

```
void addtopoint(struct point *lhs, struct point rhs)
{
    (*lhs).x += rhs.x; /* precedence! */
    (*lhs).y += rhs.y;
}
```

Pointers to structures are very common → shortcut "->"

```
void addtopoint(struct point *lhs, struct point rhs)
{
    lhs->x += rhs.x;
    lhs->y += rhs.y;
}
```

Overview

FASCHINGBAUER

Alias for Type Names

FASCHINGBAUER

Why?

- Semantics of a type is one story
- Implementation is another story
- Type names can become long

```
typedef unsigned long int uint64_t;
typedef int pid_t;
```

→ Type name and alias name are equivalent

Overview

FASCHINGBAUER

Overview

FASCHINGBAUER

Stack and Global Memory

FASCHINGBAUER

Stack

- One *stack frame* per function call
- Local variables live there
- $\rightarrow$ *Lifetime* is the duration of the function call

Global memory

- Global Variables
- "Allocated" at program start
- Lifetime: entire program

What's in between? $\rightarrow$ *explicit* lifetime

Dynamic Memory

FASCHINGBAUER

Heap memory

- *Not* part of the core language
- → implemented in the *C library*
- Lifetime is managed by the programmer
  - *Allocation*
  - *Deallocation*

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
```

Dynamic Memory — Usage

FASCHINGBAUER

```
struct point *p = malloc(sizeof(struct point));
do_something_with(p);
...
free(p);
```

New traps: as always, there is no checking done (as always, this is for performance reasons)

- *Memory leak*: forget to `free()` allocated memory
- `free()` a pointer that does not point to dynamically allocated memory
- `free()` a pointer that has already been deallocated

Exercises (1)

FASCHINGBAUER

Singly linked list: public functions ("methos")

```
int list_init(struct list *l);
int list_destroy(struct list *l);
int list_insert(
    struct list *l,
    const char *key, struct point p);
unsigned int list_remove(
    struct list *l,
    const char *key);
unsigned int list_count(
    const struct list *l,
    const char *key);
void list_print(
    const struct list *l);
```

Exercises (2)

FASCHINGBAUER

Singly linked list: public data structures

```
#define KEYLEN 31

struct point {
    int x;
    int y;
};

struct list {
    struct node *first;
};
```

Exercises (3)

FASCHINGBAUER

Singly linked list: internals

```
struct node {
    char key[KEYLEN+1];
    struct point point;

    struct node *next;
};
```

Exercises (4)

FASCHINGBAUER

- Implement a linked list as has been sketched above

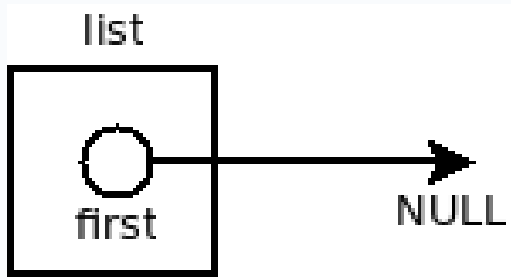
Exercises (5)

FASCHINGBAUER

Empty list

- Result of `list_init()`
- First element is `NULL`

```
struct list {
    struct node *first;
};
...
l->first = NULL;
```

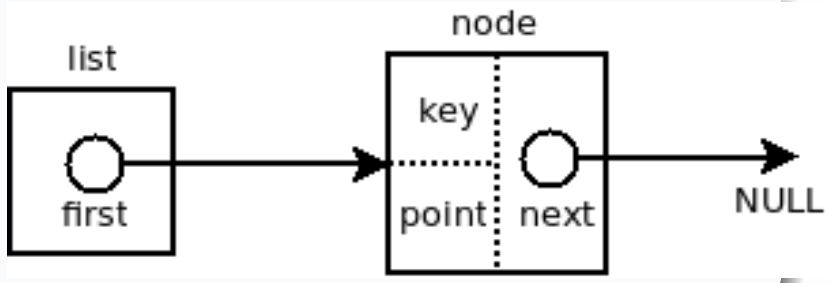


Exercises (6)

FASCHINGBAUER

List containing one element

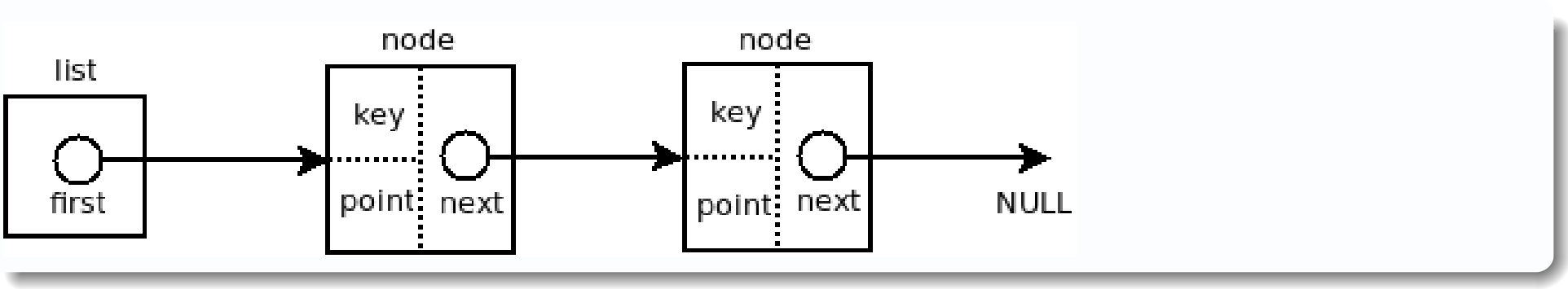
```
struct node {
    char key[keylen+1];
    struct point point;

    struct node *next;
};
...
strcpy(n->key, key);
n->point = point;
n->next = NULL;
```



Exercises (7)

FASCHINGBAUER

List containing two elements

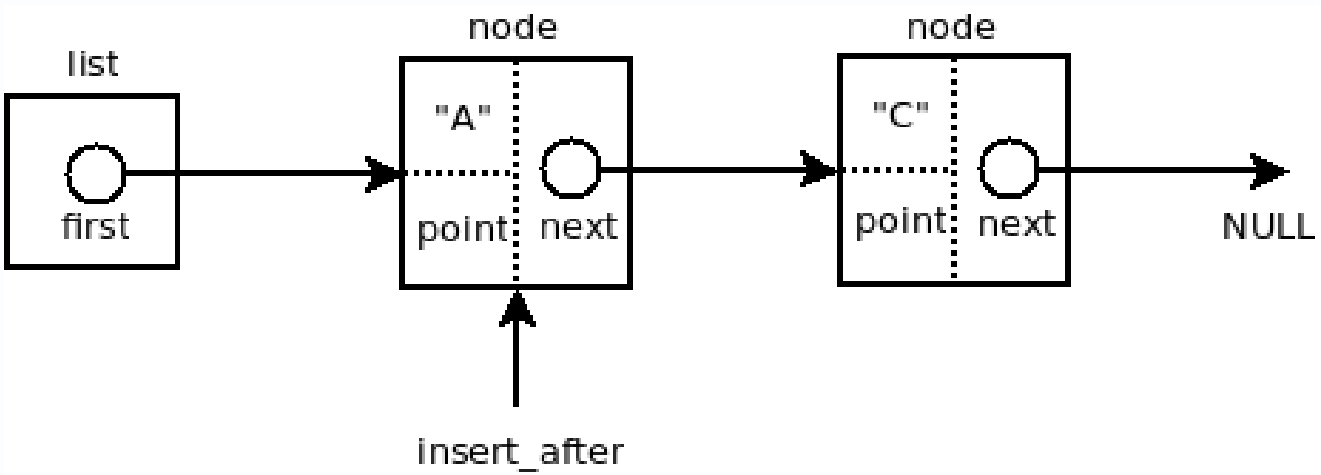


Exercises (8)

FASCHINGBAUER

Insertion: looking up the position

Where does "B" belong?



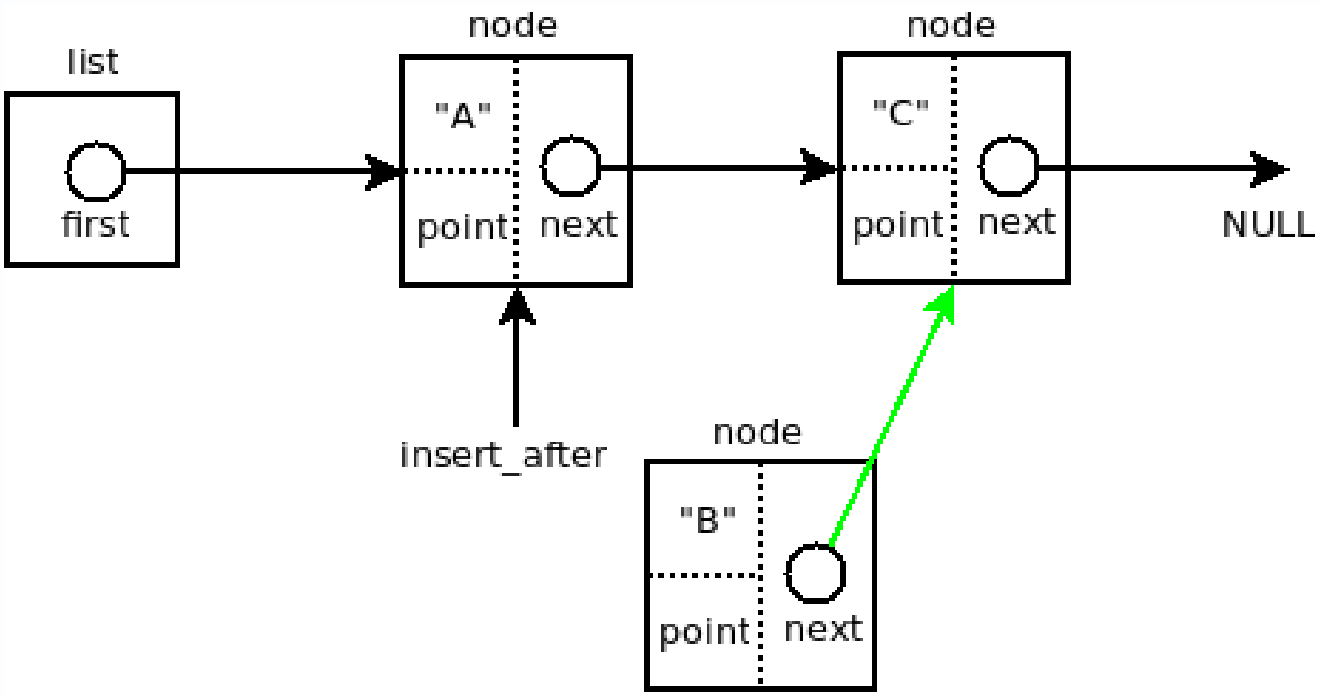
Exercises (9)

FASCHINGBAUER

Insertion: new `struct node`

- `malloc(sizeof(struct node))`
- Initialization: `key`, `data`, `next`

Where does "B" belong?

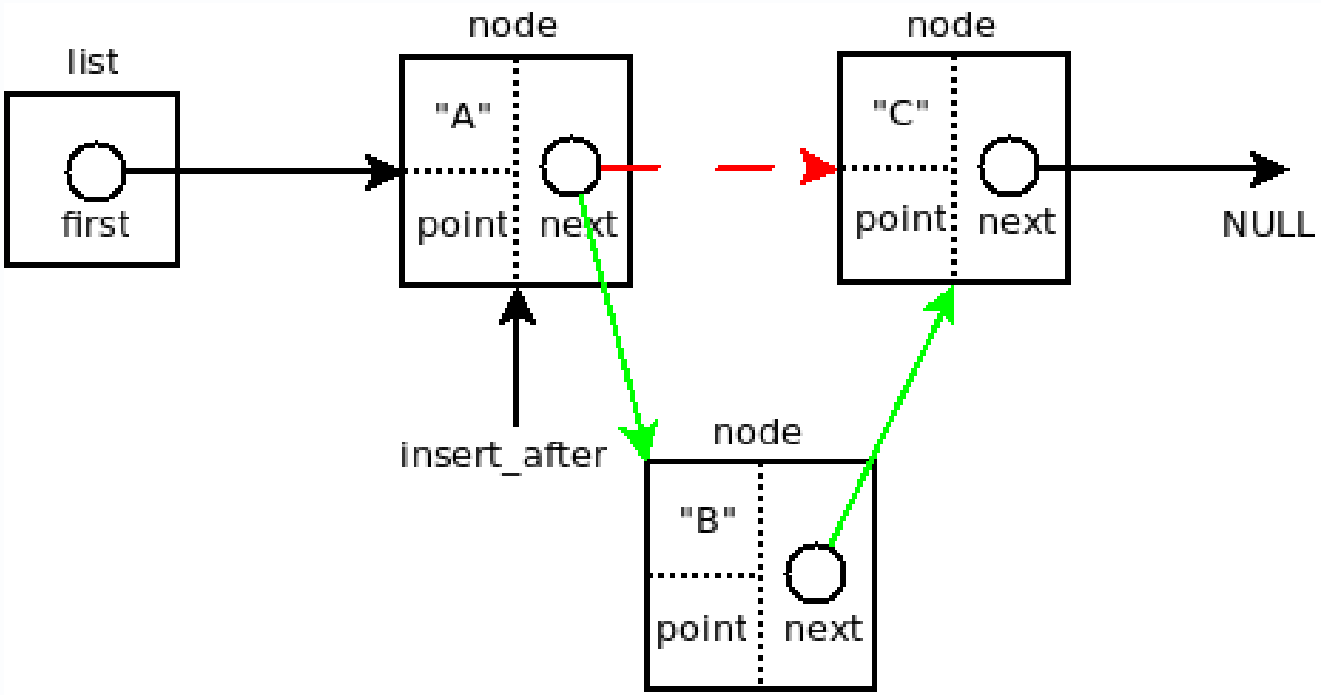


Exercises (10)

FASCHINGBAUER

Insertion: link new node

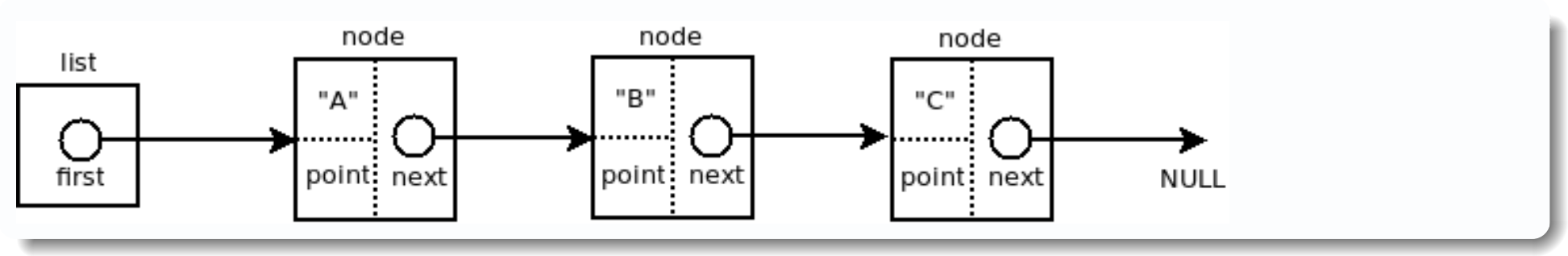
Cut old connection



Exercises (11)

FASCHINGBAUER

Insertion: done



Overview

FASCHINGBAUER

Overview

FASCHINGBAUER

`volatile`: The Lie (1)

FASCHINGBAUER

What `volatile` does:

- Prevents *compiler* optimization of everything involving the variable declared `volatile`
- Corollary: the variable must not be kept in a register

```
volatile int x;
```

Attention:

- All it does is provide a false impression of correctness
- Most of its uses are outright bugs

`volatile`: The Lie (2)

FASCHINGBAUER

What `volatile` doesn't:

- Variable can still be in a cache
  - *Variable is not at all sync with memory* when using *write-back* cache strategy
- Not a memory barrier → load/store reordering still possible (done by CPU, *not by compiler*)
- → *Not a replacement for proper locking*

Still broken: *load-modify-store*

```
volatile int use_count;

void use_resource(void)
{
  do_something_with_shared_resource();
  use_count++;
}
```

volatile: Valid Use: Hardware

FASCHINGBAUER

Originally conceived for use with hardware registers

- Optimizing compiler would wreak havoc
  - Loops would never terminate
  - Memory locations would not be written to/read from
  - ...

```
volatile int completion_flag;
volatile int out_word;
volatile int in_word;

int communicate(int word)
{
    out_word = word;
    while (!completion_flag);
    return in_word;
}
```

volatile: Valid Use: Unix Signal Handlers

FASCHINGBAUER

A variable might change in unforeseeable ways

- Signal handler modifies quit variable
- Optimizing compiler would otherwise make the loop endless

```
volatile int quit;

int main(void)
{
  while (!quit)
    do_something();
}
```

Overview

FASCHINGBAUER

Atomic Memory Access

FASCHINGBAUER

Why is this code broken for multithreaded programs?

```
volatile int use_count;

void use_resource(void)
{
  do_something_with_shared_resource();
  use_count++;
}
```

Atomic Memory Access: Load/Modify/Store

FASCHINGBAUER

Load-Modify-Store conflict

- Classic form of a *race condition*
- BTW: `volatile` is completely irrelevant!

Thread A		Thread B		
Instr	Loc	Instr	Loc	Glob
load	42			42
	42	load	42	42
inc	43			42
	43	inc	43	42
	43	store	43	43
store	43		43	43

Load/Modify/Store: Mutex

FASCHINGBAUER

```
static pthread_mutex_t use_count_mutex =
  PTHREAD_MUTEX_INITIALIZER;
int use_count;

void use_resource(void)
{
  do_something_with_shared_resource();

  pthread_mutex_lock(&use_count_mutex);
  use_count++;
  pthread_mutex_unlock(&use_count_mutex);
}
```

- **Drawback:** mutexes are expensive ($\rightarrow$ context switches)

Atomic Instructions

FASCHINGBAUER

For simple integers there is a simpler way to atomicity (GCC only)

fetch_and_add()

```
int use_count;

void use_resource(void)
{
  do_something_with_shared_resource();
  __sync_fetch_and_add(&use_count, 1);
}
```

More GCC "Builtins"

FASCHINGBAUER

GCC has a sheer number of builtins ...

- Atomic operations
- Arithmetic with overflow checking (built-in "functions" with a "success" return type)
- Pointer bounds checking

Visual C++ also has some builtins ("Intrinsics"), but I don't know these $\rightarrow$ check with MSDN

Overview

FASCHINGBAUER

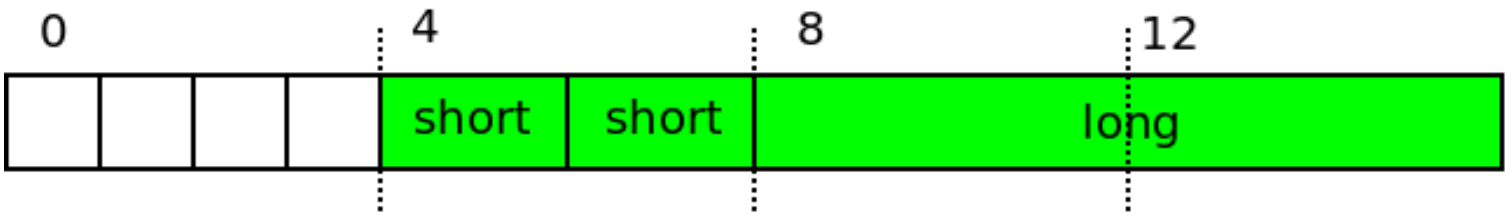
Data Alignment

FASCHINGBAUER

Data alignment: processors like data items of size N (a power of two) to exist at base addresses that are a multiple of N

→ **"Natural alignment"**

- A character can exist at any address (no alignment restriction)
- A 16 bit integer (`short` on most/all architectures) must align to a 2 byte boundary
- Same with 32 bit and 64 bit integers, `float` and `double`
- Pointers on a 32 bit architecture must align to a 4 byte boundary
- Pointers on a 64 bit architecture must align to a 8 byte boundary



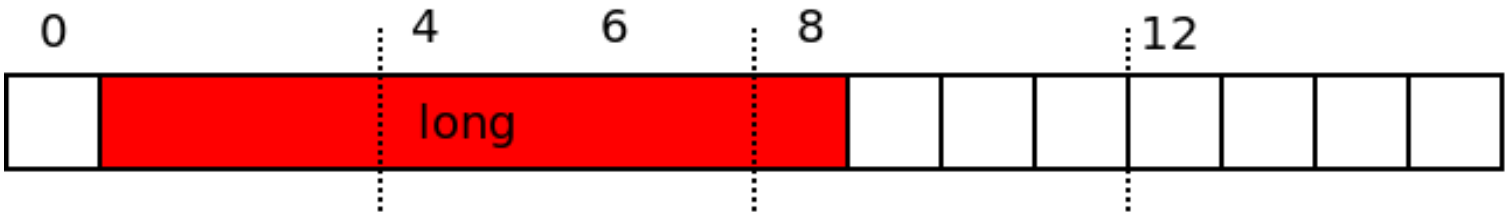
Data Alignment, Compilers

FASCHINGBAUER

- Compilers generally know about a machine's data sizes and alignment requirements
- Variables are placed at addresses that align them naturally
- Not normally a problem during development
- Except ...
  - Mixing 32 bit and 64 bit code (e.g. running a 32 bit executable on a 64 bit OS) $\rightarrow$ *different pointer sizes*, at least
  - Reading and interpreting binary data from *somewhere*

Unaligned Data Access (1)

FASCHINGBAUER



- Split in two memory accesses, combined by arithmetic (shift, bitwise OR)
- Architecture dependent
  - Done in hardware
  - Trap into OS, emulation in software
- Either way: non-negligible performance penalty
- $\rightarrow$ Play by the rules and just don't do it

How can I produce an unaligned access?

Unaligned Data Access (2)

FASCHINGBAUER

- The following code is not clean
- Works only because all is done to make unaligned access work

```
char dog[10];
char *p = &dog[1];
unsigned long l = *(unsigned long *)p;
```

- Future proof (but no faster) ...

```
char dog[10];
unsigned long l;
memcpy(&l, dog+1, sizeof(unsigned long));
```

Padding (1)

FASCHINGBAUER

What happens to structure members?

- No standalone variables which are freely allocated by the compiler
- Compiler is forbidden (per C/C++ standard) to rearrange members of a `struct`

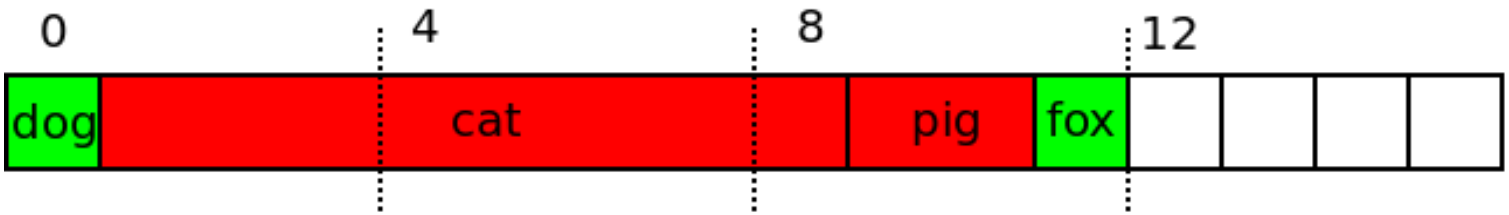
```
struct animals {
  char dog;              /* 1 byte */
  unsigned long cat;     /* 8 bytes */
  unsigned short pig;    /* 2 bytes */
  char fox;              /* 1 byte */
};
```

How large would this be? $1+8+2+1 == 12$?

Padding (2)

FASCHINGBAUER

```
struct animals {
  char dog;              /* 1 byte */
  unsigned long cat;     /* 8 bytes */
  unsigned short pig;    /* 2 bytes */
  char fox;              /* 1 byte */
};
```



- Quite naive structure layout: no compiler does this!
- Can be enforced by compiler specific structure attributes (GCC) or pragmas (Doze) → *Bogus!*

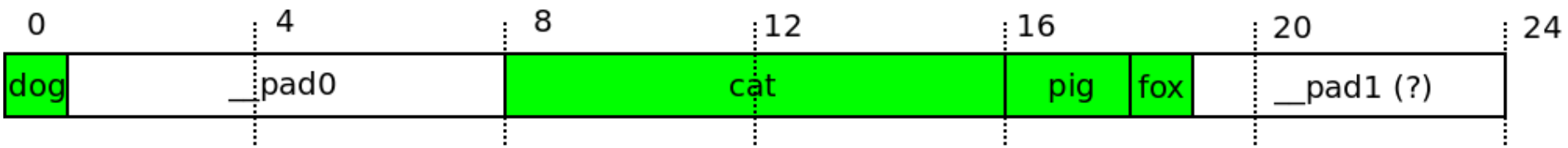
Padding (3)

FASCHINGBAUER

Compiler inserts padding

- Preserves order of members (dictated by law)
- Artificially guarantees aligned access

```
struct animals {
  char dog;              /* 1 byte */
  char __pad0[7];        /* 7 bytes */
  unsigned long cat;     /* 8 bytes */
  unsigned short pig;    /* 2 bytes */
  char fox;              /* 1 byte */
  char __pad1[5];        /* 5 bytes (?) */
};
```

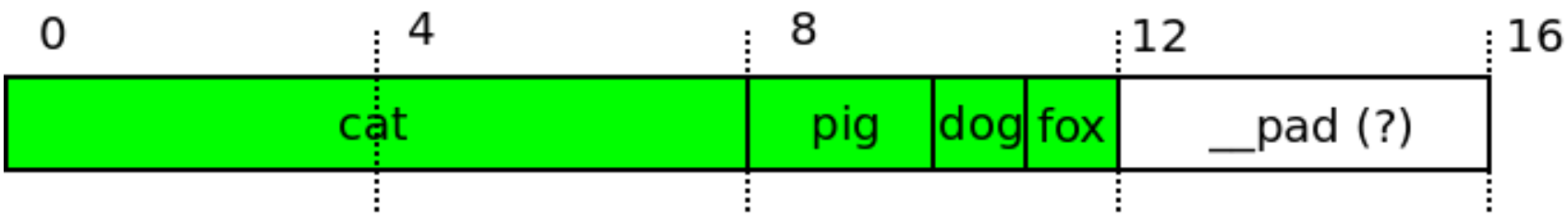


Padding (4)

FASCHINGBAUER

- **Bloat** in size
- → Rearrange members manually, ordered by decreasing size/alignment

```
struct animals {
  unsigned long cat;     /* 8 bytes */
  unsigned short pig;    /* 2 bytes */
  char dog;              /* 1 byte */
  char fox;              /* 1 byte */
};
```



Structure Alignment (1)

FASCHINGBAUER

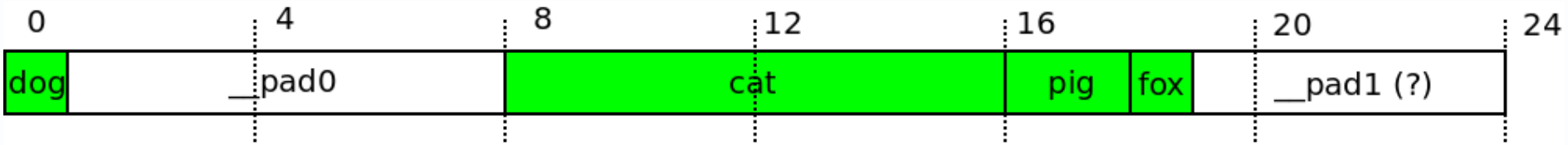
What is the alignment of a structure?

- Padding is applied inside a structure, to meet alignment requirements of all members
- On what addresses can a structure exist, then?
- $\Longrightarrow$ On all addresses where the member with the largest alignment can exist
- **Rule:** *The alignment of a structure is the alignment of the largest included type.*
- **Corollary:** *The alignment of a union is the alignment of the largest included type*

Structure Alignment (2)

FASCHINGBAUER

Remember?



- Largest member is `cat`, 8 bytes $\Longrightarrow$ structure's alignment is 8
- If we place the entire structure at address 8, `cat` is at 16 — which aligns it correctly

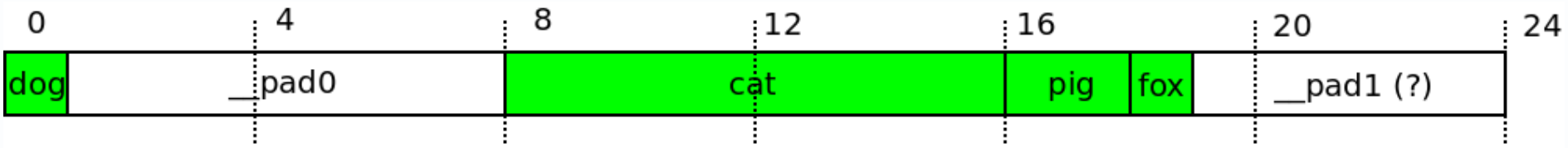
So, remember:

- **The alignment of a structure is the alignment of the largest included type.**
- **The alignment of a union is the alignment of the largest included type.**

And Arrays? (1)

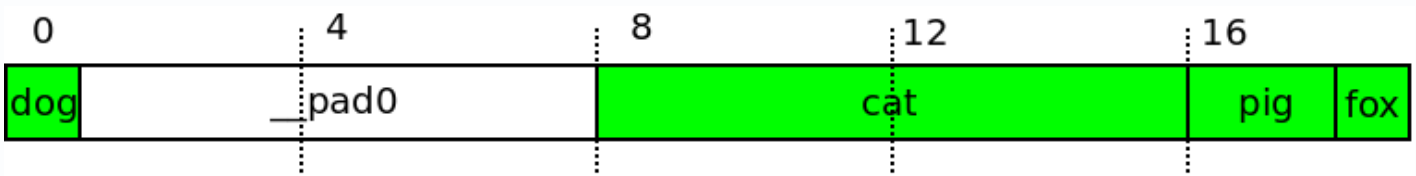
FASCHINGBAUER

Correctly Aligned



- There is a padding of 5 bytes at the end of the structure
- If we omit it, the alignment is not changed — only the structure becomes smaller in size (19 bytes, which is not only odd but prime)

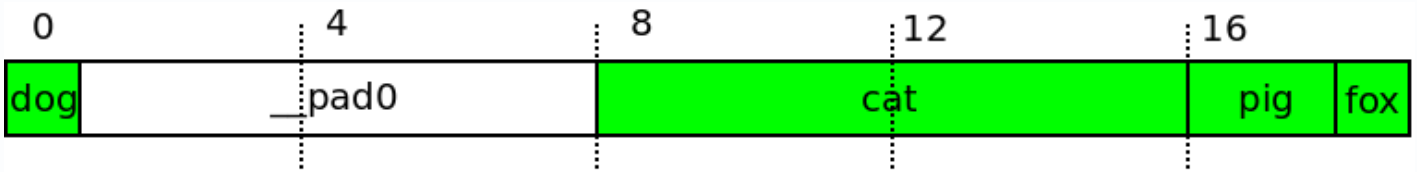
Bogus, no padding at end



And Arrays? (2)

FASCHINGBAUER

Prime-Aligned Structure



- This structure is clearly not suited for arrays
- Nearly all members of the second array element would be misaligned
- Would it suffice to end-pad the structure so its entire size is a multiple of 4?

And Arrays? (3)

FASCHINGBAUER

- Would it suffice to end-pad the structure so its entire size is a multiple of 4 (and not 8)?
- $\rightarrow$ No: the `cat` member of the second array element would then be misaligned

So, remember:

- **The size of a structure is a multiple of the alignment of the largest included type.**
- **The alignment of an array is the alignment of its base type.**

Overview

FASCHINGBAUER

Overview

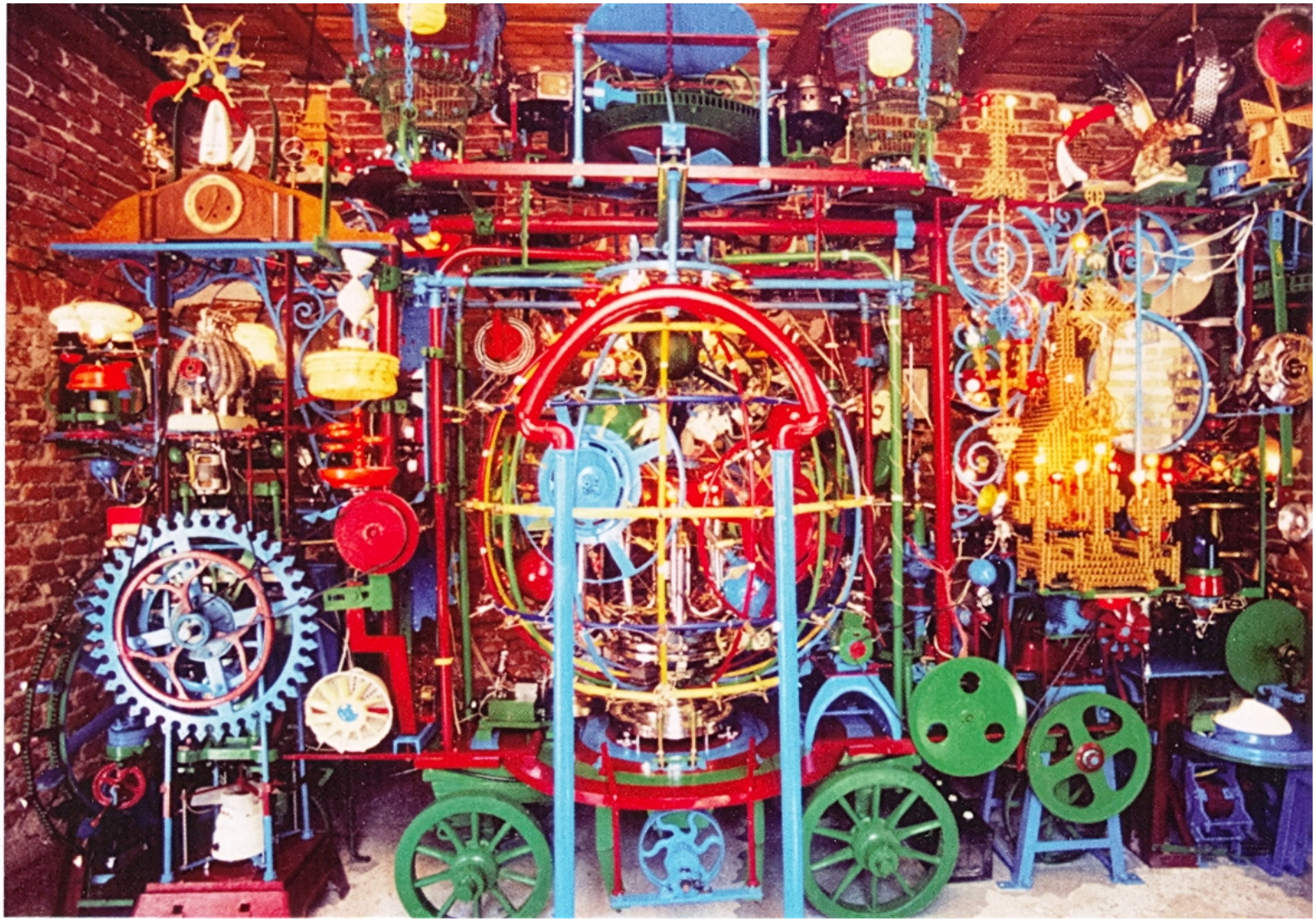
FASCHINGBAUER

Shooting Offense: Unobvious Solution

FASCHINGBAUER

Shooting Offense: Unobvious Problem

FASCHINGBAUER



Shooting Offenses

FASCHINGBAUER

We are all mature programmers, and we all know that some oddities are best rectified by firing the programmer

- Unexpected side effects
- Wrong documentation (none is far better)
- Nested loops to a depth of 10
  - With loop variables taken from somewhere in the middle of the alphabet (Fortran?)
- Obvious lazyness
- Obvious lack of respect for colleagues

Ambiguity

FASCHINGBAUER

Ambiguity is the root of all evil. Imagine ...

1. You have to take over maintenance of a large piece of code
2. $\rightarrow$ you have to understand it
3. You cannot guess from its name what a function does
4. Same with variables
5. Same with parameters
6. Return values have no obvious meaning
7. There are comments all over, obviously meant to overcome those shortcomings
8. Comments are mostly out-of-sync with the code

$\rightarrow$ *This will drive you mad!*
(... especially if it's your own code)

Readability

FASCHINGBAUER

Simple recipe for writing good code: you are able to understand what you did, even after three weeks of holiday/beer

- Chances are others will understand the code too
- Requires some discipline
- $\rightarrow$ Handcraft?

Code Smells

FASCHINGBAUER

Code smells for the following reasons

- *Comments* that explain how the code works
- *Long parameter lists*
- *Long nested `if/else` chains*
- *Hungarian notation*
- *... and many more*

We won't elaborate on that — this is not a programming course
Following a series of easy techniques to make C code readable/correct

Overview

FASCHINGBAUER

Standard Data Types: `size_t` (1)

FASCHINGBAUER

Sizes are everywhere

- Number of bytes in an allocated chunk of memory
- Number of elements in an array
- Number of microseconds until timer runs off
- Result of the `strlen()` function
- Result of the `sizeof` operator

This is what `size_t` is there for:

- Nobody has to worry about signedness (sizes simply don't become negative)
- $\rightarrow$ adds clarity

```
#include <unistd.h>
```

Standard Data Types: `size_t` (2)

FASCHINGBAUER

Consequences

- None (except for readability)
- (GCC) `-Wsign-compare`, `-Wtype-limits`, ...
  - *lots* of warnings when mixing
  - consider `-Wextra`
- → *Correctness* (up to a certain extent)

Find at least two Bugs!

```
size_t sum(int set[], size_t size)
{
    size_t sum = 0;
    while (size-- >= 0)
        sum += set[size];
    return sum;
}
```

Overview

FASCHINGBAUER

Discrete Values

FASCHINGBAUER

Many times an integer's value does not take the full possible range → `Discrete values`

- Command identifiers (e.g. Unix `ioctl`'s)
- Possible baud rates on a UART
- A state machine's state

Discrete Values — Traditional Approach (1)

FASCHINGBAUER

```
#define IDLE                0
#define WRITING_REQUEST     1
#define READING_RESPONSE    2
#define WAIT_RETRY          3

struct protocol_engine
{
  int state;
  ...
};
```

Traditional approach

- Declare a set of symbolic macros
- Let an integer carry one of these values

Drawback

- One cannot deduce valid values from looking at the type

Discrete Values — Traditional Approach (2)

FASCHINGBAUER

```
switch (engine->state) {
  case IDLE: ...;
  case WRITING_REQUEST: ...;
  case READING_RESPONSE: ...;
  case WAIT_RETRY: ...;
  default:
    error("bad state");
    break;
}
```

- `switch` is *the* statement for discrete values
- As everybody knows: `default` is obligatory

Questions

- Bad state? Why? How can this happen?
- The `switch` handles every possible value anyhow
- ... so why have a `default`?

Discrete Values — Wishlist

FASCHINGBAUER

Wishlist:

1. The value of a state is pointless. I don't want to think about it. I.e., `WRITING_REQUEST == 1` for no reason.
2. Separate type for a state, for
   - Readability
   - Type safety (to prevent mixing with e.g. integers)
3. Compiler support in `switch` like, "forgot to add `case` label for newly introduced state".

→ Fully met in C++, *only partly met* ("Type safety") in C

Discrete Values — enum

FASCHINGBAUER

```
enum state
{
  IDLE,
  WRITING_REQUEST,
  READING_RESPONSE,
  WAIT_RETRY
};

struct protocol_engine
{
  enum state state;
  ...
};
```

Advantage:

- **Wishlist item #1 and #2**
- Separate type
- Unambiguous when reading the code

But:

- The following is legal in C
- *Illegal in C++*

```
enum state s = 42;
```

Discrete Values — enum and switch

FASCHINGBAUER

Wishlist item #3: "forgot to add case label for newly introduced state"

Adding New State

```
enum state
{
  IDLE,
  WRITING_REQUEST,
  READING_RESPONSE,
  WAIT_RETRY,
  /* Error handling */
  PROTOCOL_ERROR
};
```

- State machines change
- E.g. towards the end of the project everybody wants error handling
- $\rightarrow$ Code needs to react upon the new state

Discrete Values — enum and switch

FASCHINGBAUER

```
switch (engine->state) {
  case IDLE: ...;
  case WRITING_REQUEST: ...;
  case READING_RESPONSE: ...;
  case WAIT_RETRY: ...;
  default:
    error("bad state");
    break;
}
```

"`default:`" **considered harmful**

- Eats all new states
- $\rightarrow$ *prevents the compiler from helping me*

Discrete Values — Close to Perfection

FASCHINGBAUER

```
switch (engine->state) {
  case IDLE: ...;
  case WRITING_REQUEST: ...;
  case READING_RESPONSE: ...;
  case WAIT_RETRY: ...;
  /* no default here! */
}
```

- GCC (at least) can warn about such cases
- `-Wswitch-enum`

```
$ gcc -Wswitch-enum ...
warning: enumeration value ‘PROTOCOL_ERROR’ not handled ...
$ gcc -Werror -Wswitch-enum ...
error: enumeration value ‘PROTOCOL_ERROR’ not handled ...
```

Overview

Visibility

FASCHINGBAUER

Compilation unit: the entity seen by *one* compiler call

- The C file that is being compiled
- All included header files
- Result is usually one object file

Symbol resolution:

- By compiler inside *one* compilation unit
- By linker *across multiple* compilation units
  - Among symbols that the linker *sees*

A Somewhat Contrived Example (1)

FASCHINGBAUER

Two compilation units linked into an executable ...

main.c

```
#include <stdio.h>

extern float avg(
  int *begin, int *end);

int main(void)
{
  int array[] =
    { 1, 2, 3, 4, 5 };
  printf("%f\n",
    avg(array, array+3));
  return 0;
}
```

avg.c

```
int sum(int *begin, int *end)
{
    int sum = 0;
    while (begin < end)
        sum += *begin++;
    return sum;
}
float avg(int *begin, int *end)
{
    return
      (float)sum(begin, end) /
      (end-begin);
}
```

A Somewhat Contrived Example (2)

FASCHINGBAUER

Function `sum()` in `avg.c` is globally visible

- Anybody could *declare* it and use it
  - Linker will resolve it (that's his job)
- Name could clash with another symbol in another compilation unit
  - Linker error ("duplicate symbol") when linking statically
  - Subtle bug when using shared libraries
- Innocent reader has to think twice
  - "Can I modify the function without telling anybody?"

→ **Ambiguity** that needs resolution!

The static Keyword (Hooray!)

FASCHINGBAUER

Solution: static — restrict visibility to the compilation unit

avg.c

```
static int sum(
  int *begin, int *end)
{
    int sum = 0;
    while (begin < end)
        sum += *begin++;
    return sum;
}
```

- Nobody has to think twice
- Nobody can use sum but the file it is defined in
- No name clashes
- **No ambiguity!**
- **Only readability!**
- Compiler can automatically inline the function
- ... with only 6 characters of effort

Overview

FASCHINGBAUER

Non-Modifiable Memory (1)

FASCHINGBAUER

Did you know the difference?

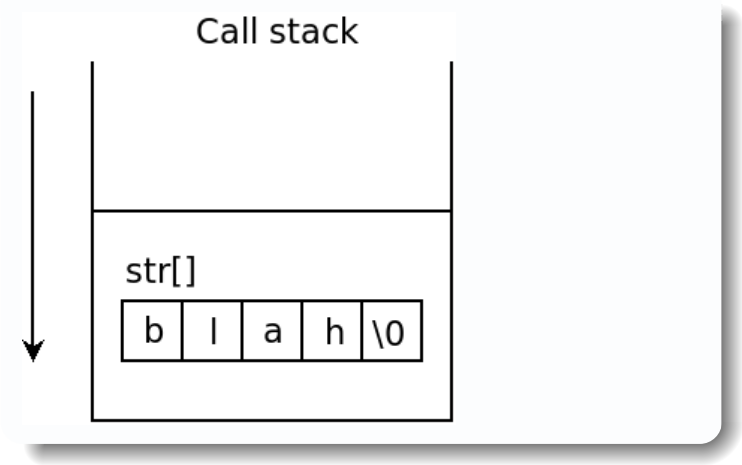
```
void f(void)
{
    char str[] = "blah";
    str[0] = 'x';
}
```

```
void f(void)
{
    char *str = "blah";
    str[0] = 'x';
}
```

Non-Modifiable Memory (2)

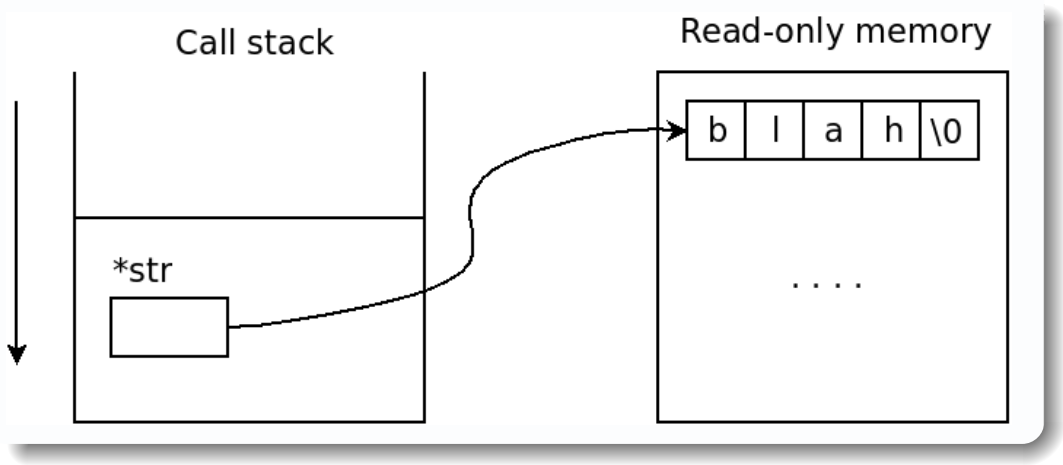
FASCHINGBAUER

```
char str[] = "blah";
```



- Array initialization
- Allocated on the *stack*, at *runtime*
- $\rightarrow$ *writable*

```
char *str = "blah";
```



- Allocated in *read-only memory*, at *compilation time*
- Pointer setup *at runtime*, to point there
- $\rightarrow$ *not writable*

The const Keyword (1)

FASCHINGBAUER

So there is already the concept of read-only data ...

- Sadly compilers generally issue no warnings
- (On Linux) Not an error, only on-demand duplication of a *shared* read-only memory page
- $\rightarrow$ *expensive*
- Unintended in most cases

```
$ gcc -Wwrite-strings ...
warning: initialization discards ‘const’ qualifier from
   pointer target type
```

A-ha: “const” qualifier!

The const Keyword (2)

FASCHINGBAUER

```
warning: initialization discards ‘const’ qualifier from
   pointer target type
```

- `char *str = "blah";`
- Obviously (no surprise) the compiler knows that `"blah"` is in read-only memory
- → String literals are `const char *`

```
const char *str = "blah";
```

Consequences:

- `str` cannot be written to
- → Code has to be fixed until compiler is happy
- → Correctness with minimal effort

const Variables

FASCHINGBAUER

Getting rid of the preprocessor (good idea) ...

```
const int MAX_BUCKETS = 64;
```

... is the same, compiler-wise, as ...

```
#define MAX_BUCKETS 64
```

Additional benefits ...

- MAX_BUCKETS has a type
- Not a stupid string substitution, but a regular C identifier
- "unused" warnings

const Parameters (1)

FASCHINGBAUER

```
int sum(int *begin, int *end);
```

Reading this declaration, we assume the following:

- It builds a sum
- It returns the result
- It operates on a range [begin, end)
- It does not modify the input data

Ambiguity alert:

- We can say nothing of the above for sure
- ... but we can help with the last item

const Parameters (2)

FASCHINGBAUER

```
int sum(const int *begin, const int *end);
```

Now we can say one thing for sure:

- It does not modify the input data

Consequences:

- sum() has to modified
- Not a big deal when only a few lines involved
- Can be a problem when code is large and complex
- → "const **pollution**"

Pointers, Pointers, Pointers ... (1)

FASCHINGBAUER

What's known so far:

- const can be applied to scalar types
- const can be applied to struct types (we don't know this, but it's a logical consequence)
- const, applied to pointers, keeps me from modifying what they point to

```
const int i;
int const j; /* same! */
const int *pi = &i;
int const *pj = &j;
```

Pointers, Pointers, Pointers ... (2)

FASCHINGBAUER

Mixing ...

```
int const i = 42;
int *pi = &i;
```

```
warning: initialization discards ‘const’ qualifier
  from pointer target type
```

- `pi` does not promise to *not modify* the value it points to
- Pointee is *read-only*
- Sadly this can only be a warning for historical reasons

Pointers, Pointers, Pointers ... (3)

FASCHINGBAUER

So, given that ...

```
int const i;
```

... is a read-only variable, ...

```
int * const pi;
```

... is a read-only variable:

- A pointer that cannot be modified
- But can be used to modify what it points to (it's an `int`, not an `int const`)

Pointers, Pointers, Pointers ... (4)

FASCHINGBAUER

```
int * const pi;
/* error: assignment of read-only variable ‘pi’ */
pi = NULL;
/* ok, compiles */
*pi = 42;
```

But is this correct?

Pointers, Pointers, Pointers ... (5)

FASCHINGBAUER

So what's this?

```
int i = 42
int const * const pi = &i;

/* error: assignment of read-only variable ‘pi’ */
pi = NULL;

/* error: assignment of read-only location ‘*pi’ */
*pi = 42;
```

Pointers, Pointers, Pointers ... (6)

FASCHINGBAUER

How about pointers that point to pointers?

```
int i = 42;
int *pi = &i;
int **ppi = &pi;

**ppi = 7;
*ppi = NULL;
```

Pointers, Pointers, Pointers ... (7)

FASCHINGBAUER

How about pointers that point to pointers that point to `const`? (Gosh)

```
int const i = 42;
int const *pi = &i;
int const **ppi = &pi;

ppi = NULL;
*ppi = NULL;
/* error: assignment of read-only location ‘**ppi’ */
**ppi = 7;
```

Pointers, Pointers, Pointers ... (8)

FASCHINGBAUER

How about pointers that point to non-modifiable pointers that point to `const`?

```
int const i = 42;
int const * const pi = &i;
int const * const *ppi = &pi;

ppi = NULL;
/* error: assignment of read-only location ‘*ppi’ */
*ppi = NULL;
/* error: assignment of read-only location ‘**ppi’ */
**ppi = 7;
```

Pointers, Pointers, Pointers ... (9)

FASCHINGBAUER

How about ...? (To be continued)

Overview

FASCHINGBAUER

Good Old Struct Initialization

FASCHINGBAUER

```
struct person
{
  char fn[16];
  char ln[16];
  int age;
  int height;
};
struct person me = { "Joerg", "Faschingbauer", 50, 172 };
```

As always: Ambiguity

- One can only guess as to what the initializer means
  - Imagine somebody's name is "Beman Dawes"
  - age? `height`? Or is it `weight`?
- Have to lookup the definition of `struct person`

C99 "Designated Initializer"

FASCHINGBAUER

```
struct person me = {
   .fn = "Joerg",
   .ln = "Faschingbauer",
   .age = 50,
   .height = 172
};
```

Consequences:

- A couple more characters of typing
- Safety: when member names (semantics?) change, the compiler *forces* checking
- Clarity

Overview

FASCHINGBAUER

Integer Types Are Ambiguous

FASCHINGBAUER

Using integers as parameters and return types obfuscates code

- Conversions happen automatically, without any notice by the compiler
- Worse: their semantics is not always clear
  - `size_t` helps to a certain extent
- Even more worse:
  - Mixing integers with different semantics
  - Changing semantics $\rightarrow$ **no** help by compiler

Example: error handling ...

Example: Ambiguous Error Schemes (1)

FASCHINGBAUER

Returns a "signed size type": negative on error, size written otherwise (Unix tradition: waste half of the domain for an occasional -1):

```
ssize_t send_frame(
  struct protocol_engine *eng,
  const struct frame *f);
```

Always returns a valid sum:

```
int sum(const int *begin, const int *end);
```

Example: Ambiguous Error Schemes (2)

FASCHINGBAUER

Automatic Conversion Massacre

```
unsigned int send_sum(
  struct protocol_engine *engine,
  const int *begin, const int *end)
{
  struct frame f;
  int retval = sum(begin, end);

  f.type = INT32;
  f.v_int32 = retval;
  retval = send_frame(engine, &f);

  return retval;
}
```

Example: Ambiguous Error Schemes (3)

FASCHINGBAUER

What are we trying to accomplish?

- `int sum()`: ok; sum of integers is an integer
  - Should think of overflow (gosh)
- `ssize_t send_frame()`: ok, but uses weird Unix style error reporting.
- `unsigned int send_sum()`: combines these in a spectacular way, and returns an application defined error number (0 for ok).

Imagine for a moment that there is one programmer who is able to code such crap ...

- Compiler happily converts between all these different integer types
- $\rightarrow$ Hell will break loose sooner or later

Artificial Integer Type safety

FASCHINGBAUER

Passing `struct` By-Value

```
struct point
{
  int x, y;
};
struct point addpoints(struct point lhs, struct point rhs);
```

`struct` **assignment only possible on equally typed values**

- Mixing impossible
- Why not wrap our integer error codes in `structs` of adequate type?

Error Schemes, Revisited

FASCHINGBAUER

What was our problem?

- Unix system calls have that weird "-1 on failure, examine global `errno` variable if so" scheme
  - Valid `errno` errors are always >0
- Application-defined `unsigned int` errors otherwise
- Mixing is prevented only by coding *very* carefully

Proposed solution: two dedicated error types ...

- `struct unix_error`, encapsulating a Unix error
- `struct app_error`, encapsulating the application's own error values

Error Schemes: Encapsulating Unix Details

FASCHINGBAUER

Sketch: Definition of unix_error

```
struct unix_error
{
    int errno;
};
static inline struct unix_error unix_error_create(int errno)
{
    struct unix_error e;
    e.errno = errno;
    return e;
}
static inline int unix_error_ok(struct unix_error e)
{
    return e.errno == 0;
}
```

Error Schemes: Using Encapsulated Stuff

FASCHINGBAUER

Sketch: Usage of Type Safe Errors

```
struct app_error send_sum(
  struct protocol_engine *engine,
  const int *begin, const int *end)
{
  struct unix_error uerr;
  ...
  uerr = send_frame(engine, &f);
  if (!unix_error_ok(uerr))
      return app_error_create(APP_OS_ERROR);
  ...
}
```

Wrap-Up: Artificial Type Safety

FASCHINGBAUER

Good news:

- It is possible to write entirely type safe code in C
- Using the right measures (inlining, small structs), no performance impact
- Greatly enhances maintainability

Bad news:

- A lot of explicit typing
- C++ can do the same with overloading and much less typing

Overview

FASCHINGBAUER

Valgrind

FASCHINGBAUER

Valgrind: debugging at its best (because it is so simple)

- Main target: memory errors
- Writing and reading beyond array bounds
- Usage of uninitialized memory
- Double `free`/`delete`
- Memory leaks

Drawback: considerable execution slowdown $\rightarrow$

- Race conditions not easily debugged
- Multithreading is hard generally
- Larger programs are not easily emulated $\rightarrow$ smaller test suites that are regularly checked with `valgrind`

Valgrind in Action (1)

FASCHINGBAUER

There are bugs that cannot be found because they

- *almost* never occur
- *almost* never are visible
- Cannot be reproduced in tests programs
- ...

Find the Bug!

```
#include <stdlib.h>
void main(void)
{
    char *bug = malloc(64);
    bug[64] = '\0';
}
```

Valgrind in Action (2)

FASCHINGBAUER

valgrind at Bug Search

```
$ valgrind ./a.out
...
Invalid write of size 1
   at 0x400552: main (array-bounds-write.c:5)
 Address 0x51bb072 is 0 bytes after a block of size 50
   at 0x4C28C6D: malloc (vg_replace_malloc.c:236)
   by 0x400545: main (array-bounds-write.c:4)
...
```

Valgrind in Action (3)

FASCHINGBAUER

Memory leak

```
$ valgrind --leak-check=full ./a.out
...
50 bytes in 1 blocks are definitely lost in loss rec..
   at 0x4C28C6D: malloc (vg_replace_malloc.c:236)
   by 0x400545: main (array-bounds-write.c:4)
...
```

→ *very* helpful!

Valgrind: more ...

FASCHINGBAUER

Uncovers many more types of errors:

- Usage of uninitialized variables
- *De*allocation errors (`free`/`delete`/`delete[]`)
- Erroneous system call usage
- ...

More information:

- `valgrind.org`
- `man valgrind` (as always)

Overview

FASCHINGBAUER

Overview

FASCHINGBAUER

Optimization — Introduction

FASCHINGBAUER

General Rules ...

- Focus on clean design → efficiency follows
- Optimization near the end of the project
- Proven hotspots need optimization
- *Proof through profiling*

"Premature optimization is the root of all evil"
Donald E. Knuth

Compute Bound or IO Bound? (1)

FASCHINGBAUER

Decide whether, what and how to optimize!

- Collect representative input data
- Why does the program take long?
- Where does it spend most of its time?
  - Userspace: this is where computation is generally done
  - Kernel: ideally very little computation

Compute Bound or IO Bound? (2)

FASCHINGBAUER

Checksumming From An Externel USB Disk

```
$ time sha1sum 8G-dev.img.xz > /dev/null
real 0m38.879s
user 0m3.349s
sys 0m0.375s
```

- `real`: total perceived run time ("wall clock time")
- `user`: total CPU time spent in userspace
- `sys`: total CPU time spent in kernel

Here: `user` + `sys` is *far less* than `real` → mostly IO

Compute Bound or IO Bound? (3)

FASCHINGBAUER

Checksumming From Internal SSD

```
$ time sha1sum 01\ -\ Dazed\ and\ Confused.mp3 1>/dev/null

real 0m0.128s
user 0m0.107s
sys 0m0.018s
```

Here: `user` + `sys` is *roughly equal* to `real`

- Almost no IO
- → Compute bound

What to do Next?

FASCHINGBAUER

Now that we know that our application is compute bound ...

- See where it spends most of its time $\rightarrow$ *profiling*
- Decide whether optimization would pay off
- Understand what can be done
- Understand optimizations that compilers generally perform

Overview

FASCHINGBAUER

Many Ways of Optimization

FASCHINGBAUER

There are many ways to try to optimize code ...

- Unnecessary ones
- Using better algorithms (e.g. sorting and binary search)
- Function call elimination (inlining vs. spaghetti)
- Loop unrolling
- Strength reduction (e.g. using shift instead of mult/div)
- Tail call elimination
- ...

Unnecessary Optimizations

FASCHINGBAUER

```
if (x != 0)
  x = 0;
```

- The rumour goes that this is not faster than unconditional writing
- Produces more instructions, at least

Inlining (1)

FASCHINGBAUER

Facts up front:

- Function calls are generally fast
- A little slower when definition is in a shared library
- Instruction cache, if used judiciously, makes repeated calls even faster
- But, as always: it depends

Possible inlining candidate

```
int add(int l, int r)
{
  return l + r;
}
```

Inlining (2)

FASCHINGBAUER

A couple rules

- Always write clear code
- Never *not* define a function because of performance reason
  - *Readability first*
  - Can always inline later, during optimization
- Don't inline large functions $\rightarrow$ instruction cache pollution when called from different locations
- Use `static` for implementation specific functions $\rightarrow$ compiler has much more freedom

Inlining (3)

FASCHINGBAUER

GCC ...

- Does not optimize by default
- Ignores explicit `inline` when not optimizing
- `-finline-small-functions` (enabled at `-O2`): inline when function call overhead is larger than body (even when not declared inline)
- `-finline-functions` (enabled at `-O3`): all functions considered for inlining $\rightarrow$ heuristics
- `-finline-functions-called-once` (enabled at `-O1`, `-O2`, `-O3`, `-Os`): all `static` functions that ...
- More $\rightarrow$ `info gcc`

Register Allocation (1)

FASCHINGBAUER

- Register access is orders of magnitude faster than main memory access
  - → Best to keep variables in registers rather than memory
- CPUs have varying numbers of registers
  - `register` keyword should not be overused
  - Ignored anyway by most compilers
- Register allocation
  - Compiler performs flow analysis
  - Live vs. dead variables
  - "Spills" registers when allocation changes

Compiler generally makes better choices than the programmer!

Register Allocation (2)

FASCHINGBAUER

GCC ...

- `-fira-*` (for Integrated Register Allocator)
- RTFM please
- A *lot* of tuning opportunities for those who care

Peephole Optimization

FASCHINGBAUER

- **Peephole**: manageable set of instructions; “window”
- Common term for a group of optimizations that operate on a small scale
  - Common subexpression elimination
  - Strength reduction
  - Constant folding
- Small scale $\rightarrow$ “basic block”

Peephole Optimization: Common Subexpression Elimination

Sometimes one writes redundant code, in order to not compromise readability by introducing yet another variable ...

```
a = b + c + d;
x = b + c + y;
```

This can be transformed to

```
tmp = b + c; /* common subexpression */
a = tmp + d;
x = tmp + y;
```

Peephole Optimization: Strength Reduction

FASCHINGBAUER

Most programmers prefer to say what they mean (fortunately) ...

```
x = y * 2;
```

The same effect, but cheaper, is brought about by ...

```
x = y << 1;
```

If one knows the "strength" of the operators involved (compilers tend to know), then even this transformation can be opportune ...

```
x = y * 3; /* y*(4-1) == y*4-y */
x = (y << 2) - y;
```

Peephole Optimization: Constant Folding

FASCHINGBAUER

Another one that might look stupid but readable ...

```
x = 42;
y = x + 1;
```

... is likely to be transformed into ...

```
x = 42;
y = 43;
```

Consider transitive and repeated folding and propagation → pretty results

Loop Invariants

FASCHINGBAUER

The following bogus code ...

```
while (1) {
  x = 42; /* loop invariant */
  y += 2;
}
```

... will likely end up as ...

```
x = 42;
while (1)
  y += 2;
```

At least with a minimal amount of optimization enabled (GCC: `-fmove-loop-invariants`, enabled with `-O1` already)

Loop Unrolling

FASCHINGBAUER

If a loop body is run a known number of times, the loop counter can be omitted.

```
for (i=0; i<4; i++)
  dst[i] = src[i];
```

This can be written as

```
  dst[0] = src[0];
  dst[1] = src[1];
  dst[2] = src[2];
  dst[3] = src[3];
```

- *Complicated heuristics*: does the performance gain outweigh instruction cache thrashing?
- $\rightarrow$ I'd keep my fingers from it!

Tail Call Optimization

FASCHINGBAUER

```
int f(int i)
{
   do_something(i);
   return g(i+1);
}
```

- `g()` is called *at the end*
- `f()`'s stack frame is not used afterwards
- **Optimization:** `g()` can use `f()`'s stack frame

CPU Optimization, Last Words

FASCHINGBAUER

Once more: **Write clean Code!**

- All optimization techniques explained are performed *automatically*, by the compiler
- Theory behind optimization is well understood → engineering discipline
- Compilers generally perform optimizations better than a programmer would
  - ... let alone portably, on different CPUs!
- "Optimization" is a misnomer → "Improvement"
  - Compiler cannot make arbitrary code "optimal"
  - Bigger picture is always up to the programmer
  - → Once more: **Write clean Code!**
- Work together with compiler → use `static`, `const`

GCC: Optimization "Levels"

FASCHINGBAUER

- `-O0`: optimization off; the default
- `-O1`: most basic optimizations; does as much as possible without compromising compilation time too much
- `-O2`: recommended; does everything which has no size impact, is unagressive, and doesn't completely chew compilation time
- `-O3`: highest level possible; somewhat agressive, can break things sometimes, eats up your CPU and memory while compiling
- `-Os`: optimize for size; all of `-O2` that doesn't increase size
- `-Og` (since GCC 4.8): "developer mode"; turns on options that don't interfere with debugging or compilation time

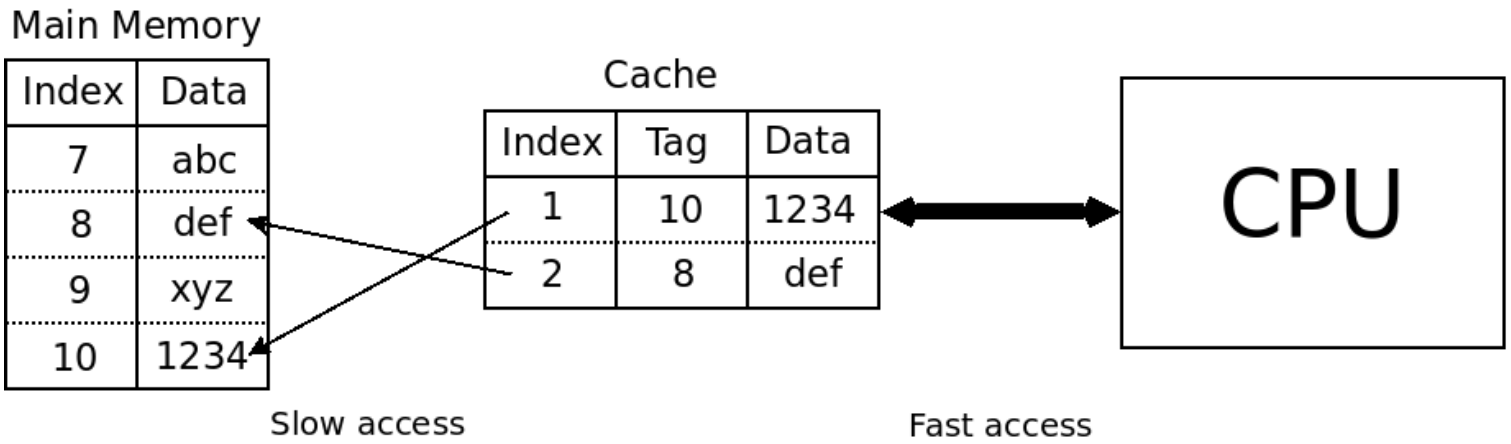
Overview

FASCHINGBAUER

Memory: Caches

FASCHINGBAUER

- Access to main memory is slow
- CPU memory cache to speed access up by magnitudes
- Organized in *cache lines* (~512 bytes each)
- Cache hierarchies



Locality of reference

FASCHINGBAUER

Rules to keep caches hot

- Group data together, so that nearby data is in the same cache line
- Use contiguous memory where possible; for example
  - Aggregation of structures
  - Sequential access in large (multidimensional?) arrays
  - Sorted arrays rather than fragmented tree structures
- Take care that data does not bounce back and forth between cache and main memory ("cache thrashing")
- $\rightarrow$ **Locality of reference**

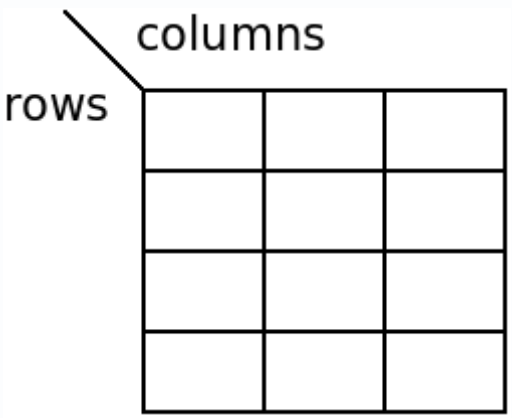
Multidimensional Arrays

FASCHINGBAUER

C Array Definition

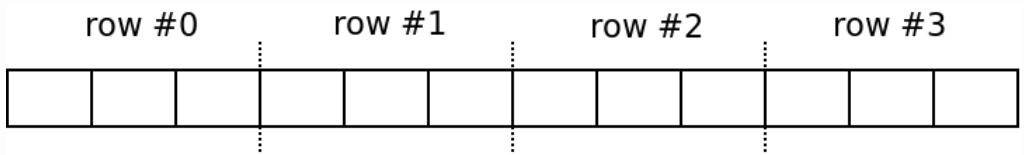
```
int array[4/*rows*/][3/*columns*/];
```

4x3 Matrix



Conceptually, a rectangular matrix

Memory layout



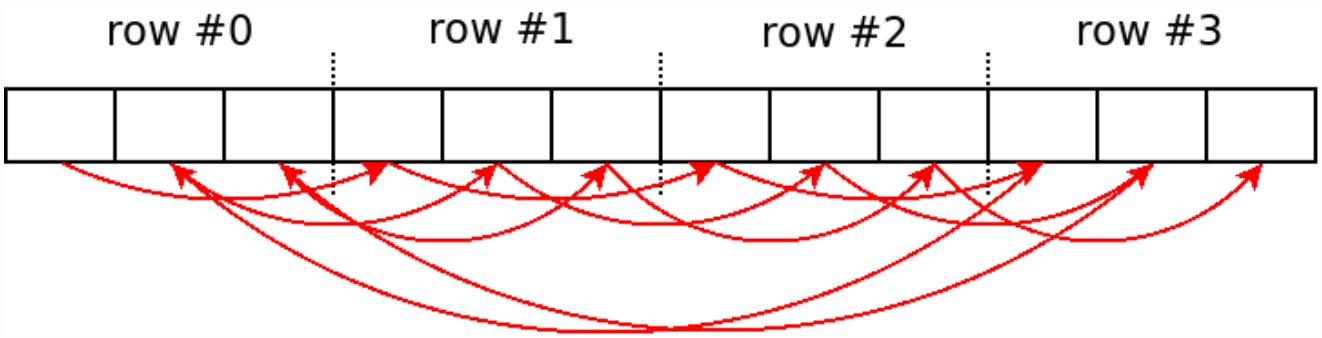
Physically, a linear array

Multidimensional Arrays: Cache Thrashing

FASCHINGBAUER

- Traversing the matrix columns-first is correct
- ... but not efficient

```
for (j=0; j<rows; j++)
  for (i=0; i<columns; i++)
    access(array[i][j]);
```

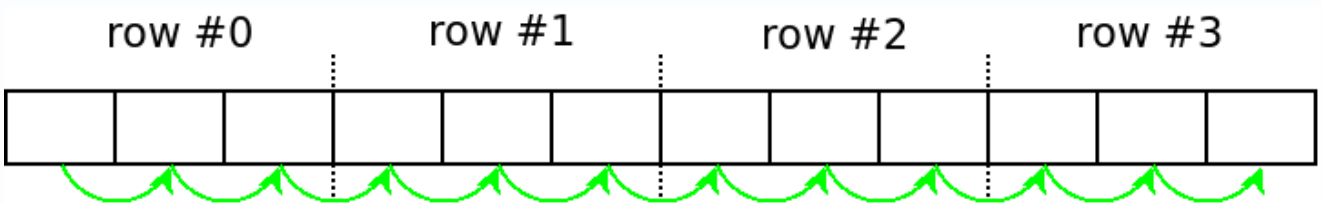


Multidimensional Arrays: Forward Indexing

FASCHINGBAUER

- Always traverse array *row-first*
- "Forward indexing"
- Best **Locality of reference**

```
for (i=0; i<rows; i++)
  for (j=0; j<columns; j++)
    access(array[i][j]);
```



Overview

FASCHINGBAUER

Overview

FASCHINGBAUER

Profiling — Famous Words

FASCHINGBAUER

"Premature optimization is the root of all evil"
Donald E. Knuth

"Premature optimization is the root of all evil"
Tony Hoare

"Optimizations always bust things, because all optimizations are, in the long haul, a form of cheating, and cheaters eventually get caught."
Larry Wall

"Measurement is a crucial component of performance improvement since reasoning and intuition are fallible guides and must be supplemented with tools like timing commands and profilers."
The Practice of Programming, Brian W. Kernighan and Rob Pike

Profiling — Introduction

FASCHINGBAUER

General Rules ...

- Focus on clean design → efficiency follows
- Optimization near the end of the project
- Proven hotspots need optimization
- *Proof through profiling*

How? On Linux ...

- `gprof`: compile time code instrumentation, single program
- `valgrind --tool=callgrind`: emulation
- `oprofile`: no instrumentation, global system view

Overview

FASCHINGBAUER

gprof — How it Works

FASCHINGBAUER

How does it work?

- *Compiler* inserts hooks into each function → counts number of calls
- *Signal handler* runs periodically to gather statistic information about each call

Compiler and Linker Calls

```
$ gcc -pg -c -o program.o program.c
$ gcc -pg -g -c -o program.o program.c # debug info
$ gcc -pg -o program program.o
```

- Running `program` creates a file `gmon.out` *in the current working directory*
- Interpreted by `gprof`

Using gprof

FASCHINGBAUER

Basic Usage

```
$ gprof program gmon.out
...
```

Prints plenty of information

- *Flat profile*: (sorted) list of functions and their numbers. Good to initially find out about the hot spots.
- *Call graph*: node-by-node listing of call graph
- *Explanations* of both (suppress with `--brief`)

gprof: Flat Profile

FASCHINGBAUER

```
$ gprof --brief --flat-profile program gmon.out
  %  cumulative  self              self    total
 time  seconds  seconds    calls us/call us/call name
 89.96     1.02    1.02 38000000    0.03    0.03 contains
  9.80     1.13    0.11  1000000    0.11    1.10 find_duplicat
```

- `% time`: percentage of entire runtime, including called subroutines
- `cumulative seconds`: same in seconds
- `self seconds`: time consumed by *the function alone*. The most valuable information → primary sorting criterion
- `calls`: total number of calls

gprof: Call Graph

FASCHINGBAUER

```
$ gprof --brief --graph program gmon.out
index % time    self  children    called     name
...
-----------------------------------------------
                0.11    0.99 1000000/1000000     main [1]
[2]     97.6    0.11    0.99 1000000         find_duplicates
                0.99    0.00 37000000/38000000     contains [3
-----------------------------------------------
...
```

- The `index` line is the *center* (what the node is about)
- Lines above: callers
- Lines below: callees
  - 3/291: 291 total calls, 3 attributed to one particular caller

gprof: Interpreting The Results

FASCHINGBAUER

1. Identify hot spots
   - *Flat profile* gives the hot spots → `self seconds` is the primary criterion
   - Scripting always possible if more is wanted
2. *Call graph* starting at the hot spots

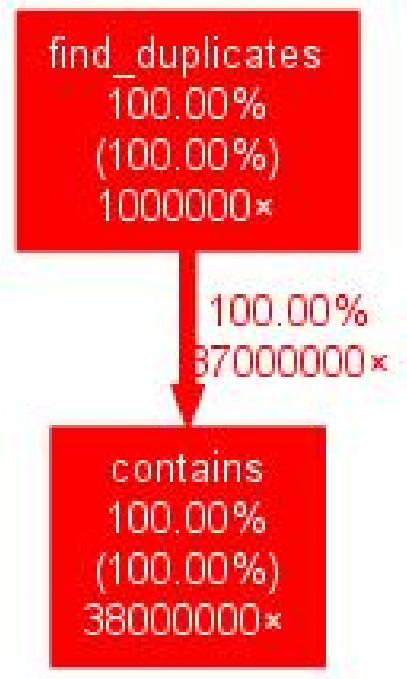
```
$ gprof --brief --graph -f contains program gmon.out
...
index % time    self  children    called     name
                0.00    0.00 1000000/38000000     main (6)
                0.00    0.00 37000000/38000000     find_duplicates
[1]      0.0    0.00    0.00 38000000         contains [1]
...
```

gprof: Visualization

FASCHINGBAUER

- One wishes that can be visualized
- We have no such luck

```
$ gprof program gmon.out | \
    gprof2dot | \
    dot -Tjpeg | \
    display -
```



Overview

FASCHINGBAUER

callgrind — How it Works

FASCHINGBAUER

How does it work?

- valgrind: run-time code instrumentation
- callgrind is a "tool" using valgrind infrastructure
- Call-graph analysis, optional cache and branch-prediction analysis

Compared to good old gprof ...

- Sluggishly slow (the price of run-time instrumentation)
- More accurate
- Nice graphical tool → kcachegrind

callgrind — How it is Used

FASCHINGBAUER

Compiler and Linker Calls

```
$ gcc -c -o program.o program.c
$ gcc -g -c -o program.o program.c # debug info
$ gcc -o program program.o
```

- No compiler attention needed
- Debug information only for source annotation ($\rightarrow$ kcachegrind)

Producing Output: callgrind.out.<pid>

```
$ valgrind --tool=callgrind ./program
...
$ ls callgrind.out.*
callgrind.out.16761
```

callgrind — Analysis Per Commandline

FASCHINGBAUER

Basically records the same information as gprof

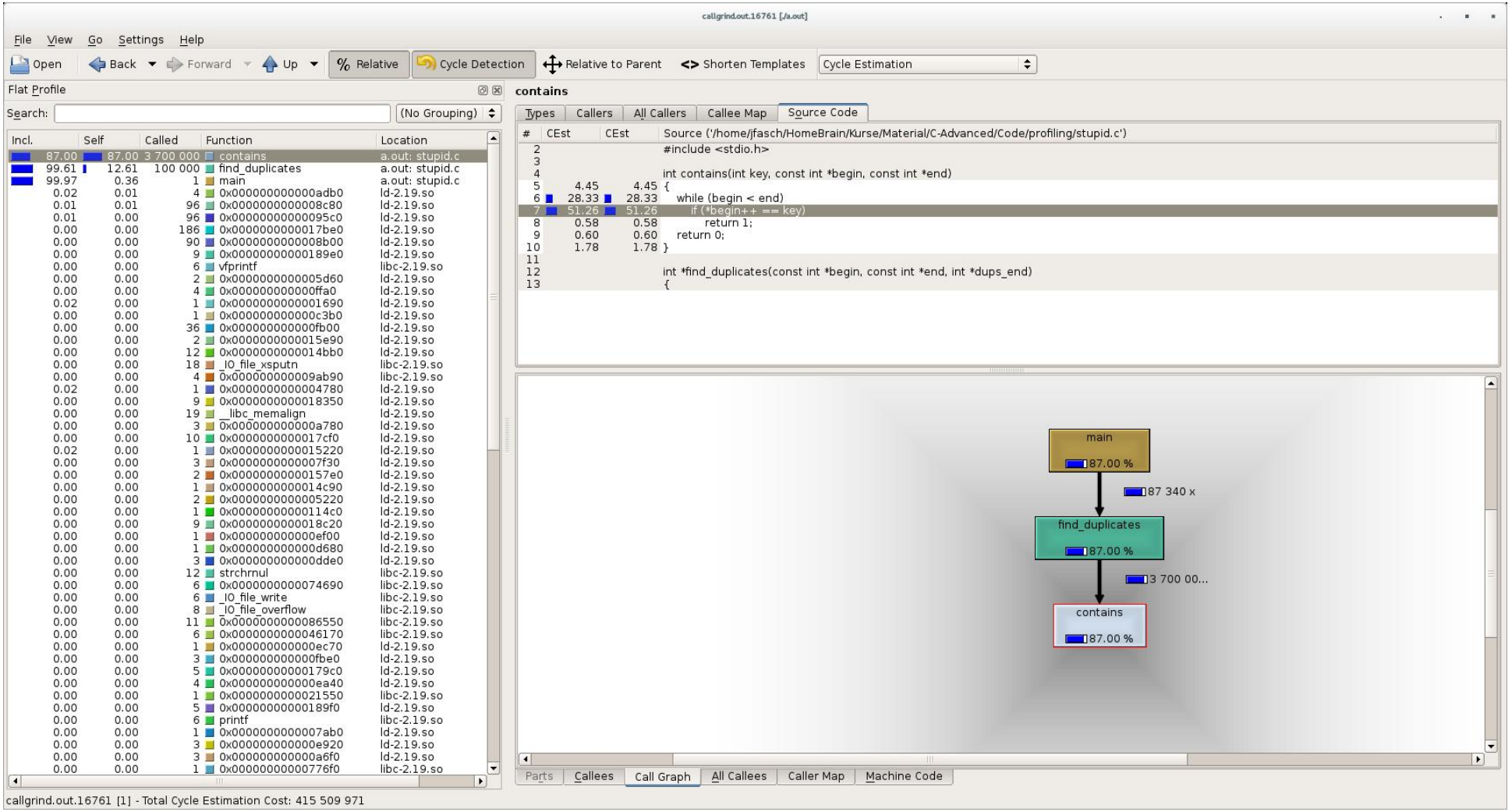
- Flat profile
- Call Graph
- All sorts of counters (can detect cache misses etc.)

Most basic analysis tool: callgrind_annotate

```
$ callgrind_annotate callgrind.out.16761
... unreadable but informative garbage ...
```

callgrind — Analysis With kcachegrind

FASCHINGBAUER



callgrind — Useful Options

FASCHINGBAUER

Instrumented code takes very long

- Start without instrumentation
- Switch on explicitly during runtime (by PID)

Start Without Instrumentation

```
$ valgrind --tool=callgrind --instr-atstart=no ./program
```

Switch on Instrumentation

```
$ callgrind_control --instr=on 16761
```

Overview

FASCHINGBAUER

oprofile — How it Works

FASCHINGBAUER

How does it work?

- Hardware based: CPUs have performance counters/events
- NMI, trapped by Linux kernel
- Samples sent to userspace

Compared to other tools ...

- Hardware → low overhead
  - 1-3% they say
- Support for wide variety of performance events
  - Cache miss
  - Branch prediction failure
  - Lots of others I don't understand (→ `ophelp`)
- *Just works!*

oprofile — Basic Usage (1)

FASCHINGBAUER

Profiling a single executable

```
$ operf ./program
```

- Creates output directory `oprofile_data`
- Used by reporting tools
  - `opreport`
  - `opannotate`
  - `oparchive`
  - `opgprof`

oprofile — Basic Usage (2)

FASCHINGBAUER

Report Everything

```
$ opreport
... long list of processes, library and kernel symbols ...
```

Report Symbols and Their Counters

```
$ opreport --symbols
```

Take Debug Information Into Account

```
$ opreport --debug-info
```

oprofile — Too Much Information

FASCHINGBAUER

- oprofile takes samples no matter what
- → too much information
  - Used shared libraries
  - Kernel

Exclude Shared Libraries and Kernel → gprof *Flat Profile*

```
$ opreport --symbols --exclude-dependent
...
samples  %        symbol name
2828     88.5410  contains
358      11.2085  find_duplicates
8         0.2505  main
```

oprofile — Call Graph

FASCHINGBAUER

- oprofile collects samples in non-maskable interrupt
- → time critical
- does not (by default) record *caller* information with every sample

Have operf Record Caller On Every Sample

```
$ operf --callgraph ./program
```

Report Callers and Callees

```
$ opreport --callgraph
... no easy reading here ...
```

oprofile — Kernel Symbols (1)

FASCHINGBAUER

Symbols come from files → file mappings

- Executables
- Shared libraries
- Kernel modules

Kernel itself is not a file!

- Bootloader loads kernel image into memory
- Kernel not necessarily contained in the file system
- → Flash memory, network (PXE boot), ...

```
$ operf ./program
$ opreport --symbols
...
8         0.4630  no-vmlinux                    /no-vmlinux
...
```

oprofile — Kernel Symbols (2)

FASCHINGBAUER

`operf` **needs some help ...**

Helping operf With Kernel Samples

```
$ operf --vmlinux=/root/linux-3.16.5-gentoo/vmlinux ./program
```

- `vmlinux` is not usually an artifact of the kernel build process
- → `make vmlinux`
- Redhat: `kernel-debuginfo`

oprofile — Kernel Modules

FASCHINGBAUER

Samples from kernel modules ...

- Attributed to *module name* in *operf* output
- ... not a file
- ... but can be found in a file
- → *opreport* (and friends) need to look it up

Directing opreport To Module Tree

```
$ opreport \
    --image-path=/lib64/modules/3.16.5-gentoo/kernel \
    --symbols
...
```

oprofile — System-Wide Profiling

FASCHINGBAUER

System-Wide Profiling: Why?

- Application ("appliance"?) consists of multiple processes
- One or more kernel drivers play a role
- → one wants to know more about the big picture

System Wide Profiling (have to be root for that)

```
# operf --system-wide
# operf --system-wide --vmlinux=/path/to/vmlinux
```

→ Samples from *everywhere* ...

- Userland processes
- Kernel code used by system calls
- Kernel code from interrupt service routines
- Kernel code from kernel threads

oprofile — Offline Profiling

FASCHINGBAUER

Absolutely cool: oparchive

- Samples gathered on production system
- Analyzed on development system
- → Transfer of every file involved

Archiving on Production Machine

```
$ operf ./program   # or whatever ...
$ oparchive --output-directory=$HOME/operf-output
$ tar -C $HOME -jcf operf-output.tar.bz2 operf-output
```

Later, at Home

```
$ tar jxf operf-output.tar.bz2
$ opreport archive:./operf-output
```

oprofile — Graphical Beauty

FASCHINGBAUER

Real men don't need no graphics — but anyway, here it is ...

Converting oprofile_data/ into callgrind Stuff

```
$ opreport -gdf|op2calltree
$ ls oprof.out.*
oprof.out.firefox                  oprof.out.NetworkManager
oprof.out.gnome-settings-daemon    oprof.out.nm-applet
oprof.out.gnome-shell              oprof.out.operf
oprof.out.gnome-terminal-server
```

- One callgrind file for each process oprofile_data/ has samples from
- ~~Decadently~~ Nicely viewable with kcachegrind

Dummy

FASCHINGBAUER