Development in C

Operating Systems UE
2022W

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

Technische Universität Wien
Computer Engineering
Cyber-Physical Systems

2022-10-13

Content

Part I (Oct 11)

- Create executables
  - Compile and link
  - Makefiles
- Program conventions
  - Argument parsing
  - Man pages
  - Error handling
  - Signals

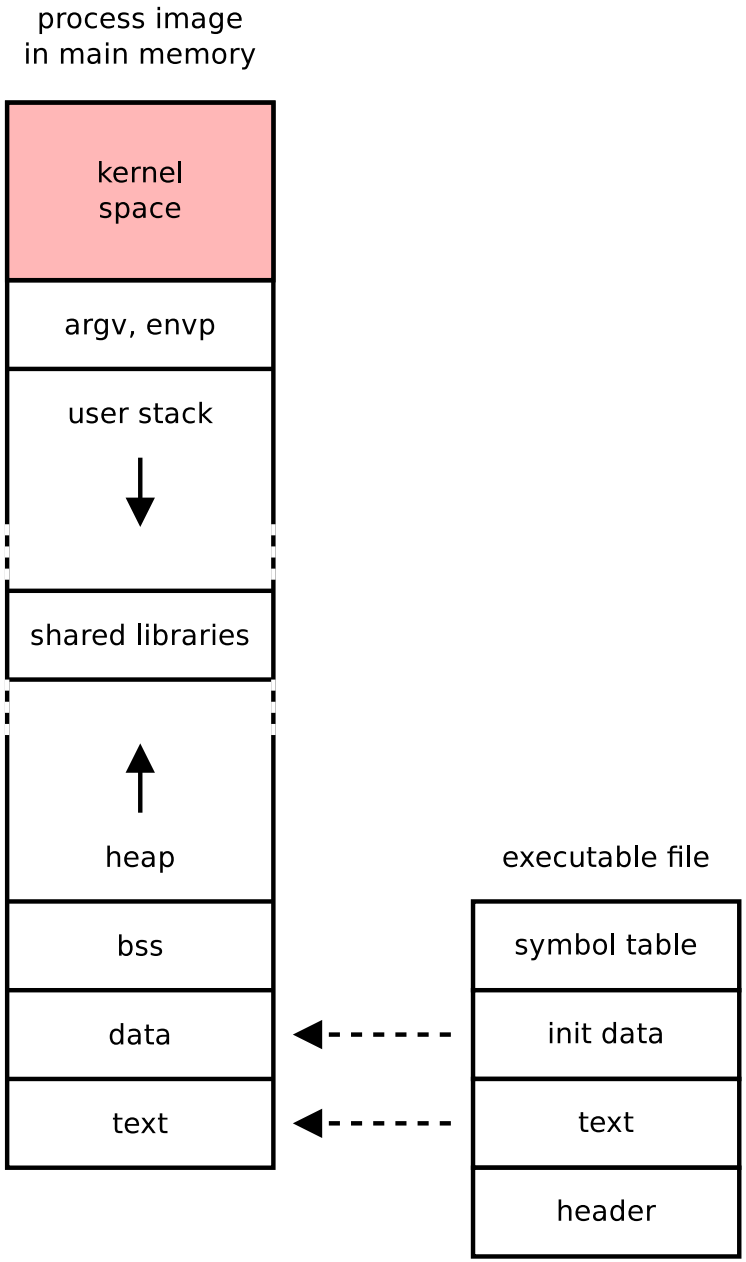
Part II (Oct 13)

- Memory management
  - Memory Areas in C
  - Dynamic Memory Management
- Error detection
  - Avoid errors
  - Static and dynamic program analysis
  - Debugging with gdb

Part II

Memory Management and Debugging

Memory Layout of a Process



Memory Management in C

- Classification
  - Memory region: static (bss,data), stack, heap
  - Requested size: constant, dynamic
  - Life span: program, block, individual by user
- Already discussed
  - Static variables (global definitions)
  - Local variables of fixed size (local definitions)
- Not yet discussed
  - Dynamic memory size
  - Flexible life span of memory

Global Variables

- Definition: (a) outside of functions or (b) `static` modifier

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

void f(void)
{
  static int keep;
  ...
}
```

- Life span: program (address always valid)
- Typical memory region: data or BSS
- Memory size: known, fixed
- Initialized
  - Explicitly initialized variables → Data
  - Implicitly initialized with 0 → BSS

Global Variables

Example 1

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

void f(void)
{
  hello[0] = 'H'; /* ?? */
}
```

OK

`hello` is an initialized array in the data segment.

Global Variables

Example 2

```
char *hello = "hello";

void f(void)
{
  hello[0] = 'H'; /* ?? */
}
```

Memory access error

`hello` is a pointer (in the data segment) to a string constant `"hello"` in the text segment.

Global Variables

Example 3

```c
const char hello[] = "hello";

void f(void)
{
  hello[0] = 'H';    /* warning (write const) */

  char *ptr = hello; /* discards const qualifier */
  ptr[0] = 'H';      /* ?? */
}
```

Memory access error

`hello` is (and remains) a constant array in the text segment.

Local Variables

- Definition: in a block (C89: at the beginning)

```
if (a > b) {
  int x;
  char c = 'A';
  ...
}
```

- Life span: until the end of the block
  - Memory released when block is left
  - Address loses validity
- Typical memory region: stack
- Memory size: known
  (Exception: arrays of variable size in C99)
- No implicit initialization (initial value undefined)

Local Variables

Example 1

```
void g(int *x)
{
  *x += 10;
}

int f(void)
{
  int a = 1; // <--
  g(&a);
  return a;
}
```

Stack:

Address	Name	Value
0xffc	a	1

Local Variables

Example 1

```
void g(int *x) // <--
{
  *x += 10;
}

int f(void)
{
  int a = 1;
  g(&a);
  return a;
}
```

Stack:

Address	Name	Value
0xffc	a	1
0xff8	x	0xffc

Local Variables

Example 1

```
void g(int *x)
{
  *x += 10; // <--
}

int f(void)
{
  int a = 1;
  g(&a);
  return a;
}
```

Stack:

Address	Name	Value
0xffc	a	11
0xff8	x	0xffc

Local Variables

Example 1

```
void g(int *x)
{
  *x += 10;
}

int f(void)
{
  int a = 1;
  g(&a);
  return a; // <--
}
```

Stack:

Address	Name	Value
0xffc	a	11

OK

Local Variables

Example 2

```
int *g(int x)
{
  int y = x + 10;
  return &y;
}

int f(void)
{
  int *a; // <--
  a = g(1);
  return *a;
}
```

Stack:

Address	Name	Value
0xffc	a	?

Local Variables

Example 2

```
int *g(int x)
{
  int y = x + 10; // <--
  return &y;
}

int f(void)
{
  int *a;
  a = g(1);
  return *a;
}
```

Stack:

Address	Name	Value
0xffc	a	?
0xff8	x	1
0xff4	y	11

Local Variables

Example 2

```
int *g(int x)
{
  int y = x + 10;
  return &y;
}

int f(void)
{
  int *a;
  a = g(1);
  return *a; // <--
}
```

Stack:

Address	Name	Value
0xffc	a	0xff4
0xff8	–	?
0xff4	–	?

Dereferencing a results in undefined behavior!

Local Variables

Stack Overflow

- Size of stack is usually bounded
  - Linux: `ulimit -s`
  - Symptom of stack overflow: Memory access error
- Example:

```
int fib(int n)
{
  if (n <= 1) {
    return 1;
  } else {
    return fib(n-1) + fib(n-2);
  }
}
...
fib(200000);
```

Dynamic Memory Allocation

Dynamic Memory Allocation

= request and release memory during runtime

- Typical memory region: heap
- Use when ...
  - Size not known until runtime
  - Life span depends on application
  - E.g.: read data of variable length, dynamic data structures
- No direct support in C $\rightarrow$ (portable) library:
  A memory allocator (malloc(3))
  - Manual memory management

`malloc`	Allocate memory of arbitrary size
`free`	Free previously allocated memory
`realloc`	Change size of allocated memory

Allocate Memory

```c
void *malloc(size_t size);

/* Examples */
int *arr   = malloc(sizeof (int) * length);
char *c    = malloc(sizeof (char));
int *x     = malloc(sizeof *x);
int *x_arr = malloc(sizeof (*x) * length);

typedef struct { int x, y, r; } circle_t;

circle_t *circ1 = malloc(sizeof (circle_t));
circle_t *circ2 = malloc(sizeof *circ2);
```

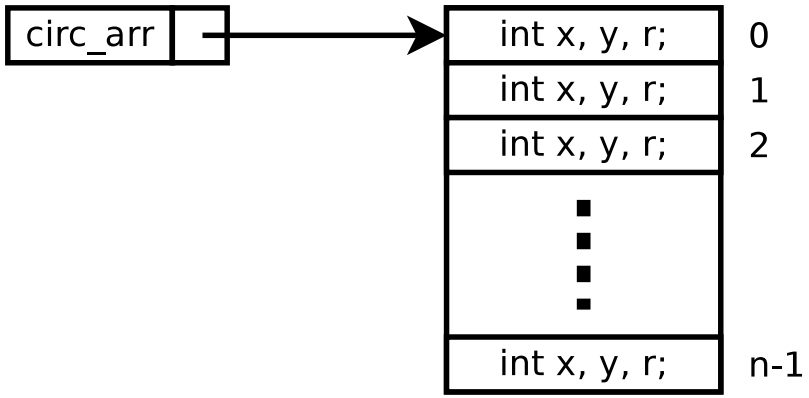
- Allocation of `size` bytes continuous memory
- Return value: Start address of the allocated memory
- Address identifies allocated memory
- Allocated memory is not initialized (cf. calloc(3))

Allocate Memory

```
typedef struct { int x, y, r; } circle_t;

circle_t *circ_arr = malloc(n * sizeof (circle_t));

for (int i = 0; i < n; i++) {
  circ_arr[i].x = 0;
  circ_arr[i].y = 0;
  circ_arr[i].r = i+1;
}
```



Recall: `pointer[i]` is equivalent to `*(pointer + i)`

Allocate Memory

```
typedef struct { int x, y, r; } circle_t;

circle_t *circ_arr = malloc(n * sizeof (circle_t));

for (int i = 0; i < n; i++) {
  circle_t *cur = &circ_arr[i];
  cur->x = 0;
  cur->y = 0;
  cur->r = i+1;
}
```

Why not `circle_t cur = circ_arr[i]`?

Copies the circle `circ_arr[i]` into `cur` $\rightarrow$ the copy `cur` is written, but the circles in `circ_arr` are not changed.

Free Allocated Memory

```
void free(void *ptr);
```

- `ptr` shall be `NULL` or a valid address
- Address is valid if
  - returned by `malloc()`, `realloc()` oder `calloc()`
  - not yet freed by `free()` or `realloc()`
- No effect if `ptr` is `NULL`

Change Size of Allocated Memory

```
void *realloc(void *ptr, size_t size);
```

- Changes size of the allocated memory identified by `ptr`
  - Size gets bigger → whole data is preserved
  - Size gets smaller → first `size` bytes of data are preserved
- Expects a valid address at `ptr` (like `free()`)
- Data may be copied → start address may change (return value)
- Failed allocation returns `NULL`, old memory area at `ptr` remains valid
- Special cases
  - `ptr` is NULL → same behavior as `malloc()`
  - `size` is 0 → same behavior as `free()`

Usage

malloc

```c
char *p;

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

if (p == NULL) {
   error_exit("malloc for p failed.");
} else {
   strncpy(p, "hallo", 6);
}

free(p); /* don't forget to finally free */
```

Prevent Memory Access Errors!

Check the return value (as with all functions)!

Usage

free

```
char *p, *q;

p = malloc(sizeof(char) * 6);
if (p == NULL) error_exit("malloc");
```

```
p += 3;


free(p);
```

✗

```
p += 3;
q  = p - 3;

free(q);
```

✓

```
free(p);

p[0] = 'H';
```

✗

Attention

- Pass only a valid starting address!
- Address gets invalid after `free`!

Example

`realloc` - a dynamically growing stack

```
typedef struct {
  int *items;
  unsigned int capacity, top;
} stack_t;

void push(stack_t *st, int x)
{
  if (st->top == st->capacity) {
    // need to grow
    int newcap = st->capacity + 10;
    int *newptr = realloc(st->items,
                          sizeof(int) * newcap);
    if (newptr == NULL) {
      // error; old data (st->items) is still valid
      ...
    }
    st->items = newptr; // st->items was deallocated
    st->capacity = newcap;
  }

  st->items[st->top++] = x;
}
```

Note on Security

```c
char *secret_key;

secret_key = malloc(sizeof(char) * 128);
load_key(secret_key);
/* use key for encryption, etc. */

free(secret_key);
```

- `secret_key` is not valid after `free()`
- content still resides somewhere in memory

Note on Security

```c
char *secret_key;

secret_key = malloc(sizeof(char) * 128);
load_key(secret_key);
/* use key for encryption, etc. */

for (int i = 0; i < 128; i++)
  secret_key[i] = '\0'; // erase key

free(secret_key);
```

- might be removed by compiler optimization
- should use OPENSSL_cleanse(3) or similar methods

Memory Mapping

Linux: mmap(2)

= maps a file into the virtual memory

- Advantages
  - No need to copy data into the address space of a process
  - Processes may use a common memory area
- Problematic with very small and very big files (clipping and fragmentation, respectively)
- Applications
  - Efficiently read and write files
  - Efficiently reallocate bigger memory areas (takes advantage of paging)
  - Inter-process communication over shared memory (a file)

On Debugging

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

– Brian Kernighan

On Testing

Program testing can be used to show the presence of bugs, but never to show their absence!

– Edsger Wybe Dijkstra

Coding Guidelines

- Coding guidelines
  - Homogeneous, comprehensible code
  - Avoid error-prone code constructs
  - Ease up automatic program verification and debugging
- Style guides (e.g., BSD kernel)
  - Homogeneous format of comments and code
  - Rules for names of macros, variables and functions
  - Recommendations and prohibitions (e.g., do not use pointers in boolean context) for code constructs and library functions

Guidelines for Safety-Critical Code

Example of Jet Propulsion Laboratory, Caltech [1]

- Programming style
  - Maximal one screen page per function
  - Limited use of the preprocessor
- Defensive Programming
  - Extensive use of assertions
  - Minimize scope of variables
  - Check all return values
  - Consider and remove all compiler warnings
- Simplify program tests
  - Introduce (static) loop bounds
  - Simple code constructs (no goto, no recursion)
  - No dynamic memory allocation after initialization
  - Limited use of pointers

Assertions

Design by Contract

Assertion

= dynamic test of preconditions, invariants (e.g., in loops) and postconditions (e.g., before and after some function)

```
void add_address_to_list(const char *street,
                         unsigned int no) {
  /* street must have < 40 characters */
  :
  item_t *item = malloc(sizeof *item);
  :
  /* item pointer must be valid */
  :
  /* given address now in list */
}
```

Assertions in C

- Implementation in C: assert(3)
  - Include with `<assert.h>`
  - Execution stops via abort(3) with an error message if an assertion is violated ($\rightarrow$ core dump)
  - Deactivate with `gcc -DNDEBUG ...`

```
void add_address_to_list(const char *street,
                         unsigned int no) {
  /* street must have < 40 characters */
  assert(strlen(street) < 40);
  :
  item_t *item = malloc(sizeof *item);
  :
  /* item pointer must be valid */
  assert(item != NULL && "item pointer is NULL");
  :
  /* given address now in list */
  assert(list_contains(street, no));
}
```

Toolchain Revisited

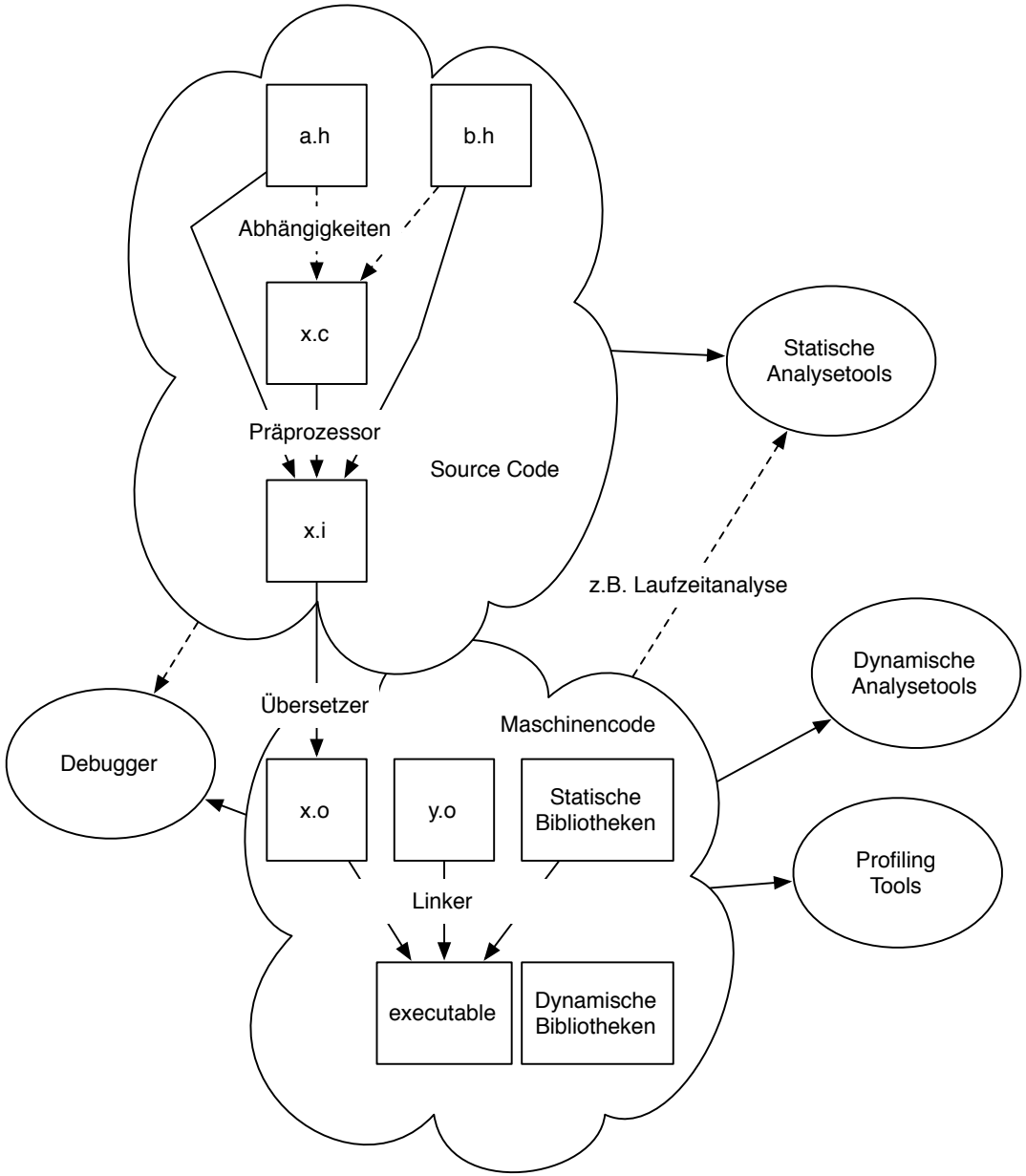


Image credits: Benedikt Huber

Program Analysis vs. Debugging

- Static analysis (inspection of code)
  - On source code or binary/executable
  - Simple form: compiler warnings (`-Wall`, clang)
  - Tools: splint, cppcheck, scan-build, BLAST, Frama-C, many other commercial tools
- Dynamic analysis (inspection during execution)
  - Instrumented program
  - Tools: ASan, valgrind, dmalloc
- Debugging (inspection of the running process)
  - Debug messages (`printf`)
  - Interactive, using a debugger (e.g., gdb)

Static Analysis

- Analyse source code of the program
  - Detects many typical errors (but not all)
  - Common problem: false positives → desensitization
- Error prevention
  - Warning on error-prone constructs
  - Stricter type rules
- Error detection
  - Access to uninitialized memory
  - Usage of invalid addresses
  - Violation of assertions
- Example: splint[1]
  - Additional annotations: increase accuracy, stricter assertions

[1] http://www.splint.org

Static Analysis

Example: splint

```
int main(int argc, char **argv)
{
  int c, opt_a, opt_o = 0;
  char *arg;
  while ((c = getopt(argc, argv, "a:o") != -1))
    // splint: Assignment of boolean to int
    // splint: Test expression for while not boolean, type int
  {
    switch (c) {
      case 'a':
        arg = optarg;
        opt_a++;
        // splint: Variable opt_a used before initialization
        // splint: Fall through case (no preceding break)
      case 'o':
        opt_o++;
        break;
      default:
        assert(0);
    }
  }
  return 0;
}
```

Dynamic Analysis

- Dynamic Analysis of an additionally gathered information during program execution
- $\rightarrow$ quality depends on test runs
- Example: AddressSanitizer (ASan)[2]
  - Instruments memory access during compilation, models address space
  - Detects access to freed memory, overflows, etc. (aborts program at first error)
  - Included in gcc 4.8 and above
  - Compile and link with `-fsanitize=address`

[2] https://github.com/google/sanitizers/wiki/AddressSanitizer

Dynamic Analysis

Example: AddressSanitizer

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

int main(int argc, char **argv)
{
  int buf[4] = {10, 20, 30, 40};
  char *test = malloc(20);

  printf("buf %d\n", buf[4]);

  for (int i = 0; i <= 20; ++i)
    test[i] = 'a' + i;
  free(test);

  test[0] = 'X';

  return 0;
}
```

Dynamic Analysis

Example: AddressSanitizer

```
$ make asan_ex DEFS=-fsanitize=address
gcc -std=c99 -pedantic -Wall [...] -fsanitize=address -g asan_ex.c -o asan_ex
$ ./asan_ex
=================================================================
==20208==ERROR: AddressSanitizer: stack-buffer-overflow on address
0x7ffe686ea3a0 at pc 0x000000400ae9 bp 0x7ffe686ea350 sp 0x7ffe686ea340
READ of size 4 at 0x7ffe686ea3a0 thread T0
    #0 0x400ae8 in main [...]/development_in_C/examples/asan_ex.c:9
    #1 0x7f355ef9382f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2082f
    #2 0x400928 in _start ([...]/development_in_C/examples/asan_ex+0x400928)

Address 0x7ffe686ea3a0 is located in stack of thread T0 at offset 48 in frame
    #0 0x400a05 in main [...]/development_in_C/examples/asan_ex.c:5

  This frame has 1 object(s):
    [32, 48) 'buf' <== Memory access at offset 48 overflows this variable
HINT: this may be a false positive if your program uses some custom stack
unwind mechanism or swapcontext
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow
                           [...]/development_in_C/examples/asan_ex.c:9 main
Shadow bytes around the buggy address:
  [...]
  0x10004d0d5450: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10004d0d5460: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 f1 f1
=>0x10004d0d5470: f1 f1 00 00[f4]f4 f3 f3 f3 f3 00 00 00 00 00 00
  0x10004d0d5480: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  [...]
==20208==ABORTING
```

Static Analysis

Example: cppcheck

```
$ cppcheck --enable=all main.c
Checking main.c ...
[main.c:9]: (error) Array 'buf[4]' accessed at index 4,
  which is out of bounds.
[main.c:13]: (error) Array 'test[20]' accessed at index 20,
  which is out of bounds.
(information) Cppcheck cannot find all the include files
  (use --check-config for details)
```

Static Analysis

Example: cppcheck

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

int main(int argc, char **argv)
{
  int buf[4] = {10, 20, 30, 40};
  char *test = malloc(20);

  if(argc > 5)
    printf("buf %d\n", buf[4]);

  for (int i = 0; i < 20; ++i)
    test[i] = 'a' + i;
  free(test);

  return 0;
}
```

Static Analysis

Example: cppcheck

```
cppcheck --enable=all main.c
Checking main.c ...
[main.c:10]: (error) Array 'buf[4]' accessed at index 4,
  which is out of bounds.
[main.c:6]: (style) The scope of the variable 'buf' can be reduced.
(information) Cppcheck cannot find all the include files
  (use --check-config for details)
```

Debugging

- Detect and correct errors
- Modify and test the program
  - Add assertions (assert(3))
  - Add debug messages
- Using a debugger
  - Halt and inspect a running process
  - Inspection of core(5) dumps

Debug Messages I

- Conditional activation using the preprocessor

```
gcc -g -DDEBUG ..
```

- Deactivated: performance not influenced

```
#ifdef DEBUG

#include <stdio.h>
#define debug(msg) \
    (void) fputs(msg, stderr)

#else

#define debug(msg) /* NOP */

#endif // DEBUG

...

debug("I reached this point");
```

Debug Messages II

- Extended messages using predefined and variadic macros[3]

```
#define debug(fmt, ...) \
  (void) fprintf(stderr, "[%s:%d] " fmt "\n", \
                 __FILE__, __LINE__, \
                 ##__VA_ARGS__)

...

debug("x=%d", x);

// gcc warning with '-std=c99 -pedantic':
debug("reached");
```

- Output:

```
[dbgmacro.c:16] x=41
[dbgmacro.c:18] reached
```

[3] see http://gcc.gnu.org/onlinedocs/cpp/Macros.html, section "Predefined Macros" and "Variadic Macros"

Debugging Tools

- Inspection during running process
- Tasks of a debugger
  - Halt the process at specific points (breakpoints)
  - Inspection of memory and registers
  - Process and display informationen (connection to source code)
  - Manipulation of registers and main memory

GDB

The GNU Project Debugger[4]

- Debugging of the executable
- Preparation
  - Compile with `-g` (debugging symbols)
- Optimizations complicate debugging
  - Eliminates variables, preprocessor macros
  - Structural changes (e.g., function inlining, loop optimization)
  - Attention: sometimes errors may only occur in optimized version

[4] http://www.gnu.org/software/gdb

GDB

Start Debugging

- Start the program in the debugger
  - `gdb myprogram` and `run [args]`
  - `gdb -args myprogram [args]` and `run`
  - Option `-tui` starts gdb with a text-based user interface
- Connect to running process
  - via process id (PID)
  - `gdb myprogram` and `attach 2345`
  - `gdb myprogram 2345` if a process exists with this PID
  - see also pidof(8), ps(1)
- Analyze a core dump
  - When process abnormaly terminates
  - In the `bash`-Shell first activate core dumps with: `ulimit -c unlimited`
  - `gdb -c core myprogram` or `gdb myprogram core`

GDB

Basic Commands

- Execute program until next breakpoint

Specify program	`file binary`
Start program	`run args...`
Continue execution after break	`continue`

- Execute program step-wise

Until the next line	`step`
Until the next instruction	`stepi`
Like `step`,	`next(i)`
but skips function calls	
Until a higher line number	`until`
or end of function	
Until the specified line	`until x.c:30`

Breakpoints

Breakpoint

= point in the program where the execution shall be interrupted

- Implementation in HW (support by CPU) or SW (modification of program code)
- Specification (attention if you use optimization)

File and line number	`break example.c:24`
Function entry	`break main`
Conditional breakpoint	`break f if (x>3)`
Modify breakpoint	`disable/enable/delete`

Watchpoints

Watchpoint

= interrupts execution when a variable changes

- Implementation in HW (up to 16 bytes on x86) or SW (slow, single stepping)
- Specification

Write	`watch myvar`
Read	`rwatch myvar`
Read/write	`awatch myvar`

- Applications
  - Observe variable over a longer period
  - If unknown, when the variable changes
- Catchpoints
  = interrupts execution in case of specific events
  - Process activities (`catch fork`, `catch exec`)
  - Syscalls (`catch syscall name`)

GDB

Inspect Variables

- Print current value of an expression

```
print argc                  $1 = 1
print argv[0]               $2 = "hello"
print/x &argc               $3 = 0xbfffff2b0
p *((int*)0xbfffff2b0))     $4 = 42
```

- Change value
  - `set variable x=0`
  - `set (x=0)`
- Print on each break with `display`

GDB

Signals

- Configurable as “breakpoint”
- SIGINT → control to debugger
- Selectively pass signals to the program
  - See `info signals`

```
$ gdb ./myprogram
(gdb) handle SIGUSR1 nostop print pass
(gdb) run
(gdb) signal SIGUSR1
```

GDB

References

- Tutorial: `http://beej.us/guide/bggdb/`
- Manual: `http://sourceware.org/gdb/onlinedocs/gdb/`
- gdb help:

```
$ gdb
(gdb) help
```

Summary

- Dynamic memory management and dynamic allocation of memory with malloc
- Error prevention (coding guidelines, assertions)
- Static (splint) and dynamic (ASan) program Analysis
- Debugging (debug messages and gdb)

Be prepared for the delivery talk!

The tutors might try to crash your program with malicious inputs and use valgrind, ASan and/or splint to find bugs.

Part III

Appendix

References I

G.J. Holzmann NASA/JPL Laboratory for Reliable Software.
The Power of 10: Rules for Developing Safety-Critical Code.
Computer, 39(6):95–97, June 2006.

Implementierung von `malloc`

- malloc Bibliotheksroutinen
  - Bibliothek (z.B. libc) verwaltet Speicherblöcke, die wiederum vom Betriebssystem angefordert wurden
  - Online Algorithmus (keine Annahmen über zukünftiges Allokationsverhalten)
- Example: Doug Lea's malloc (dlmalloc)[5]
  - Binning: Schnelle Allokation für kleine Blöcke, schnelle Suche für größere Blöcke
  - Meta-Informationen in "Boundary Tag" neben dem allokierten Speicherblock
  - Anforderung neuen Speichers mit sbrk(2) und mmap(2)

[5] http://g.oswego.edu/dl/html/malloc.html