Exercises in System Level Programming (SLP) – Summer Term 2024

Exercise 9

Maximilian Ott

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

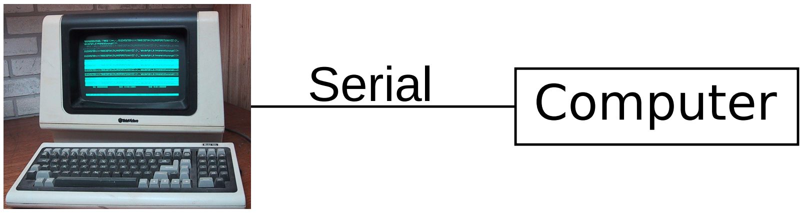
Chair in Distributed Systems
and Operating Systems

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Linux

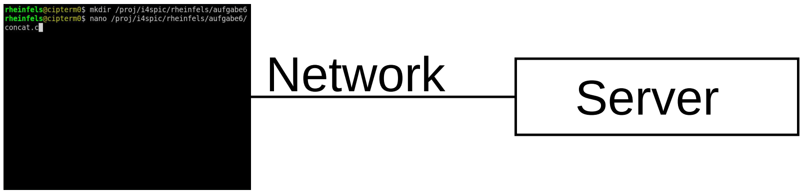
Terminal - History (simplified)

- When computers were bigger than today:



Televideo 925 (Public Domain: Wtshymanski @Wikipedia)

- When the internet was really slow:

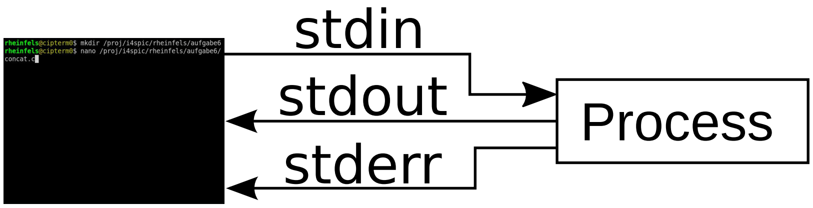


```
rheinfels@cipterm0$ mkdir /proj/i4spic/rheinfels/aufgabe6
rheinfels@cipterm0$ nano /proj/i4spic/rheinfels/aufgabe6/
concat.c
```

- Colors, position jumps, etc. are indicated by special character sequences

Terminal - Functionality

- Three standard streams for in- and output



```
rheinfels@cipterm0$ mkdir /proj/i4spic/rheinfels/aufgabe6
rheinfels@cipterm0$ nano /proj/i4spic/rheinfels/aufgabe6/
concat.c
```

`stdin` Input

`stdout` Output

`stderr` Error message

- Standard behaviour
  - Inputs are received from the keyboard
  - Outputs & error messages appear on the screen

Terminal - Redirect Standard Streams

- Write `stdout` into a file

```
find . > directories.txt
```

- Use `stdout` as `stdin` for other programs

```
cat directories.txt | grep tmp | wc -l
```

- Advantage of `stderr`
  - ⇒ Error messages are still displayed in the terminal
- Overview
  - `>` Write standard output `stdout` into file
  - `>>` Append standard output `stdout` to an existing file
  - `2>` Write error messages `stderr` into a file
  - `<` Read `stdin` from a file
  - `|` Use output of one command as input for another command

Shell - Important Commands

- Change directory with `cd`

```
# absolute path to the directory
cd /proj/i4spic/<login>/aufgabeX/

# relative path to the directory
cd aufgabe5/

cd ~          # user directory ($HOME)
cd ..         # parent directory
```

- List directory contents with `ls`

```
ls            # show files in current directory
ls -A         # also show hidden files
ls -lh        # show more meta data
```

Shell - Important Commands

- Copy file or directory with `cp`

```
# Copy file ampel.c from $HOME to the project directory
cp ~/ampel.c /proj/i4spic/xy42abcd/aufgabe5/ampel.c

# Copy directory aufgabe5/ from $HOME to the project directory
cp -r ~/aufgabe5/ /proj/i4spic/xy42abcd/
```

- (Permanently) Delete file or directory with `rm` (remove)

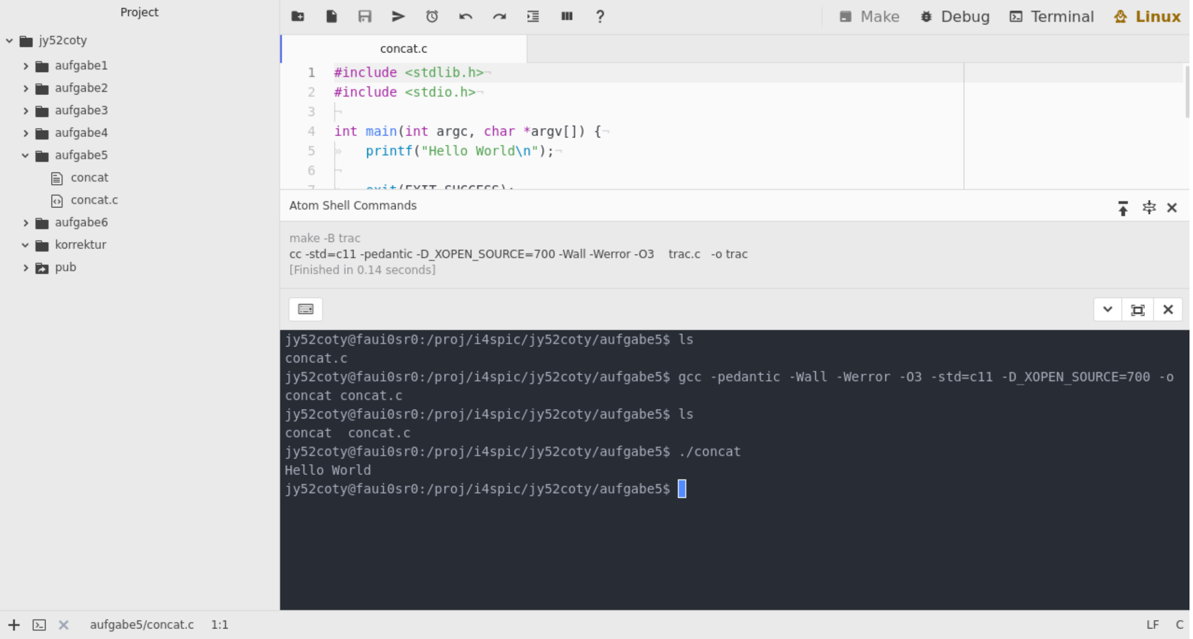
```
# Remove file test1.c inside the current directory
rm test1.c

# Remove subdirectory aufgabe1/ and all contained files
rm -r aufgabe1
```

Shell - Terminate Programs

- With a signal: `CTRL-C` (can be ignored by the program)
- Using another console: `killall concat` terminates all programs with the name “concat”
- Using the same console:
  - `CTRL-Z` stops the currently running process
  - `killall concat` then terminates all programs with the name `concat`
    - ⇒ Programs of other users shall never be terminated
  - `fg` continues the stopped process
- If nothing else works: `killall -9 concat`

SPiC IDE (Linux)



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

int main(int argc, char *argv[]) {
    printf("Hello World\n");

    exit(EXIT_SUCCESS);
```

```
make -B trac
cc -std=c11 -pedantic -D_XOPEN_SOURCE=700 -Wall -Werror -O3   trac.c  -o trac
[Finished in 0.14 seconds]
```

```
jy52coty@faui0sr0:/proj/i4spic/jy52coty/aufgabe5$ ls
concat.c
jy52coty@faui0sr0:/proj/i4spic/jy52coty/aufgabe5$ gcc -pedantic -Wall -Werror -O3 -std=c11 -D_XOPEN_SOURCE=700 -o
concat concat.c
jy52coty@faui0sr0:/proj/i4spic/jy52coty/aufgabe5$ ls
concat  concat.c
jy52coty@faui0sr0:/proj/i4spic/jy52coty/aufgabe5$ ./concat
Hello World
jy52coty@faui0sr0:/proj/i4spic/jy52coty/aufgabe5$ 
```

- **Terminal**: opens a terminal and starts a shell
  - efficient interaction with the system
  - optionally full screen
- **Debug**: starts the debug mode
- **Make**: see next week

Compiling & Executing

- Compile program with GCC

```
gcc -pedantic -Wall -Werror -O3 -std=c11 -D_XOPEN_SOURCE=700 -o
    ↪ concat concat.c
```

`gcc`	calls the compiler (GNU Compiler Collection)
`-pedantic`	activates warnings (different to the C standard)
`-Wall`	activates warnings (typical errors, e. g.: `if(x = 7)`)
`-Werror`	makes warnings into errors
`-O3`	activates optimizations (level 3)
`-std=c11`	sets the used standard to C11
`-D_XOPEN_SOURCE=700`	adds certain POSIX extensions
`-o concat`	specifies the name of the output file (standard: `a.out`)
`concat.c` …	file(s) that have to be compiled

- Execute the program with `./concat`
- All submitted assignments will be tested with these flags

Compilation & Execution

- Compile the program with GCC
  (including debug symbols and without optimizations)

```
gcc -pedantic -Wall -Werror -O0 -std=c11 -D_XOPEN_SOURCE=700 -g -
    ↪ o concat concat.c
```

`-O0`	prevents the compiler from optimizing the program
`-g`	produces debug symbols in the executable file

⇒ enables the debugger to create references to the source file

- Hint: Arrow key ⇑ iterates over previous commands

⇒ GCC command only has to be typed once

Valgrind

- Information about:
  - Memory leaks (malloc(3)/free(3))
  - Invalid memory accesses
- Ideal for debugging segmentation faults (SIGSEGV)
- Calls:
  - `valgrind ./concat`
  - `valgrind --leak-check=full --show-reachable=yes`
    `↪ --track-origins=yes ./concat`
- The output is way more useful, if the analyzed binary was built with debug symbols

Manual Pages

- Interface to the system reference manuals
- Divided into multiple sections
  1 Executable programs or shell commands
  2 System calls
  3 Library calls
  5 File formats and conventions (special data structures, etc.)
  7 Miscellaneous (e. g. terminal drivers, IP, ...)
- man pages are usually cited with the appropriate section: `printf(3)`

```
# man [section] term
man 3 printf
```

- Search for sections: `man -f term`
- Search man pages for a keyword: `man -k keyword`

Linux `libc`-API

- Trimmed (nicer) version of the man pages
- Only provide an overview and not a full specification
- Can be called from inside the SPiC-IDE (`Hilfe`-button when inside the Linux mode)
- Can be found on the website

  `https://sys.cs.fau.de/lehre/ss24/slp/exercises/linux-libc-doc`

- Our overview does not replace the man pages
- In the exam: Printed man pages!

Error Handling

Error Causes

- Errors can happen due to different reasons
  - System resources are completely exhausted
    - ⇒ `malloc(3)` fails
  - Invalid user inputs (e. g. non existent files)
    - ⇒ `fopen(3)` fails
  - Temporary errors (e. g. unavailable server)
    - ⇒ `connect(2)` fails

Error Handling

- Good software:
  - Detects the error
  - Handles error appropriately
  - Prints out a meaningful error message afterwards

- Can a program continue after an error occurred?

  Example 1: Determining the hostname of an IP address to add both values to a log file

  ⇒ Add IP address to the log, program can continue

  Example 2: Opening a file, that has to be copied, fails

  ⇒ Error handling: Copying impossible, terminate program
  ⇒ Or continue the copying process with the next file
  ⇒ Decision has to be made by the software developer

Errors in Library Functions

- Errors often occur in `libc` functions
  - Can (usually) be detected by the return value (man page)
  - Checking for errors is essential

- Error causes are usually written to `errno` (global variable)
  - Can be included with `errno.h`
  - Error codes are `> 0`
  - Error codes for all possible errors (refer to `errno(3)`)

- Only evaluate `errno` if an error was signaled
  - Functions are allowed to modify `errno` arbitrarily

    ⇒ `errno` can also be modified if no error occurred

Errors in Library Functions

- Print error codes:
  - `perror(3)`: Output on `stderr`
  - `strerror(3)`: Convert into error message (string)

Example:

```c
char *mem = malloc(...);

// Error case
if(NULL == mem) {
  fprintf(stderr, "%s:%d: malloc failed with reason: %s\n",
      __FILE__, __LINE__-5, strerror(errno));
  //alternativ: perror("malloc");

  exit(EXIT_FAILURE);
}
```

Extended Error Handling

- Indicating an error via the return value is not always possible
- Return value EOF: Error case **or** `End-Of-File`

```c
int c;
while ((c=getchar()) != EOF) { ... }
/* EOF or error? */
```

- Detection for I/O streams: `ferror(3)` bzw. `feof(3)`

```c
int c;
while ((c=getchar()) != EOF) { ... }
/* EOF or error? */
if(ferror(stdin)) {
  /* Error */
  ...
}
```

The Function main()

The main() Function

- Function `main( )`: Entry point of a C program
- Signature depends on its usage:
  - AVR: Only one program
    - ⇒ `void main(void)`
  - Linux: Multiple programs
    - ⇒ `int main(void)`
    - ⇒ `int main(int argc, char *argv[])`
- Parameters and return value used for communication

Command Line Parameters

- Command line arguments: Parameters for the program
- `main()` receives them as function parameters:
  - `argc`: Number of arguments
  - `argv`: Array of pointers to the arguments
  - ⇒ Array of strings
- First argument: program name

Command Line Parameters – Example

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

int main(int argc, char *argv[]) {
  for(int i = 0; i < argc; ++i) {
    printf("argv[%d]: %s\n", i, argv[i]);
  }

  return EXIT_SUCCESS;
}
```

```
$ ./commandline
argv[0]: ./commandline
$ ./commandline Hello world
argv[0]: ./commandline
argv[1]: Hello
argv[2]: world
```

Return Status

- Return status: Information for the caller
- Usual codes:
  - EXIT_SUCCESS: Execution succeeded
  - EXIT_FAILURE: Error occurred

Return Status – Example

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

int main(int argc, char *argv[]) {
  if(argc == 1) {
    fprintf(stderr, "No parameters given!\n");
    return EXIT_FAILURE;
  }

  // [...]

  return EXIT_SUCCESS;
}
```

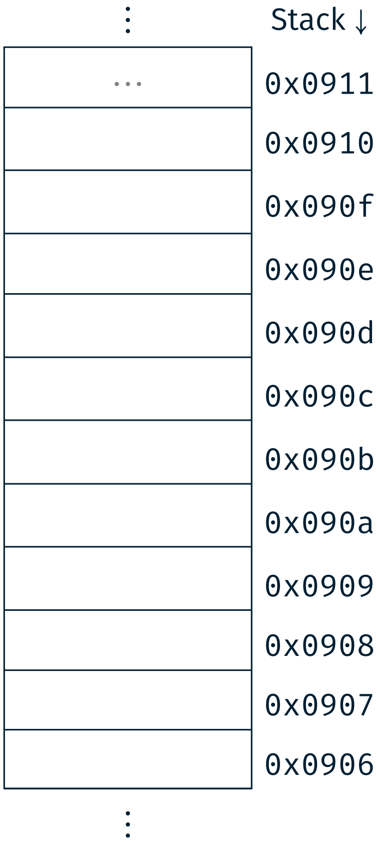
```
$ ./exitcode
No parameters given!
$ echo $?
1
$ ./exitcode Hello world
$ echo $?
0
```

C Strings in Detail

In Depth: Strings

- char: Single character (e. g. 'a')
- String: Array of chars (e. g. "Hello")
- C: Last char of a string: '\0'
  ⇒ Memory requirement: strlen(s) + 1

```c
char s[] = "World\n";
char c = s[0];
c = s[4];
char *s2 = s + 2;
c = s2[1];
```

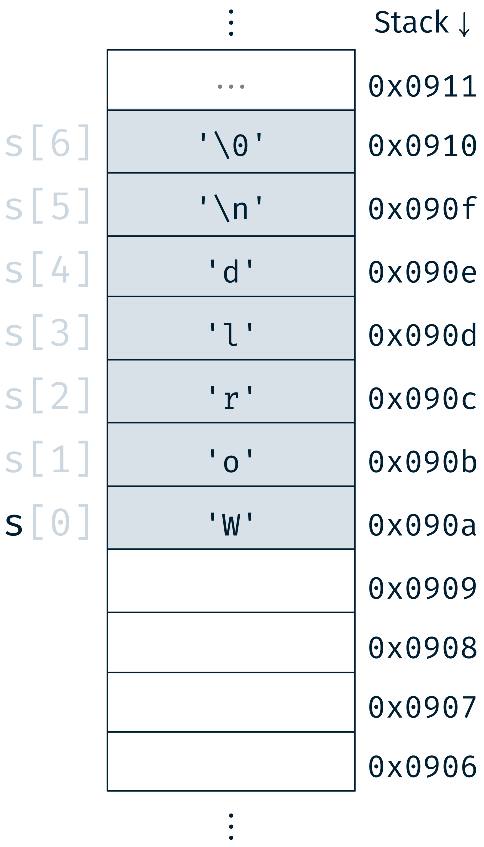


	Stack ↓
⋮	
…	0x0911
	0x0910
	0x090f
	0x090e
	0x090d
	0x090c
	0x090b
	0x090a
	0x0909
	0x0908
	0x0907
	0x0906
⋮	

In Depth: Strings

- char: Single character (e. g. 'a')
- String: Array of chars (e. g. "Hello")
- C: Last char of a string: '\0'
  - ⇒ Memory requirement: strlen(s) + 1

```
char s[] = "World\n";
char c = s[0];
c = s[4];
char *s2 = s + 2;
c = s2[1];
```

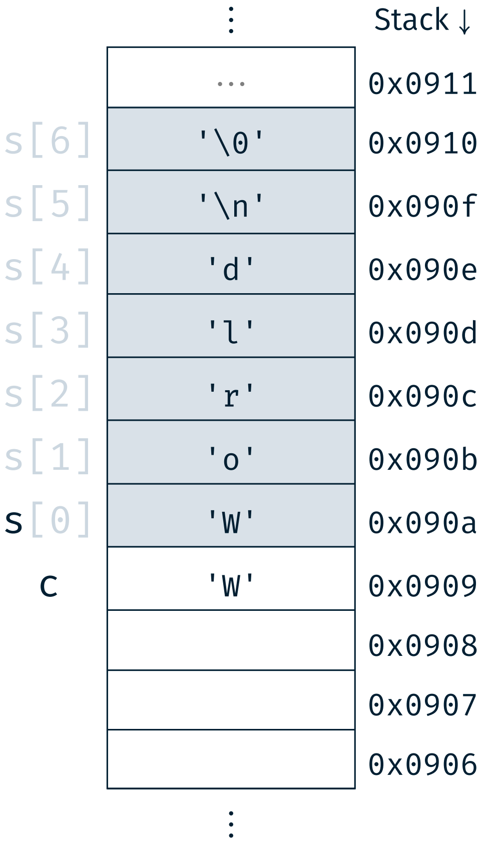


	⋮	Stack ↓
	...	0x0911
s[6]	'\0'	0x0910
s[5]	'\n'	0x090f
s[4]	'd'	0x090e
s[3]	'l'	0x090d
s[2]	'r'	0x090c
s[1]	'o'	0x090b
s[0]	'W'	0x090a
		0x0909
		0x0908
		0x0907
		0x0906
	⋮	

In Depth: Strings

- char: Single character (e. g. 'a')
- String: Array of chars (e. g. "Hello")
- C: Last char of a string: '\0'
  - ⇒ Memory requirement: strlen(s) + 1

```
char s[] = "World\n";
char c = s[0];
c = s[4];
char *s2 = s + 2;
c = s2[1];
```

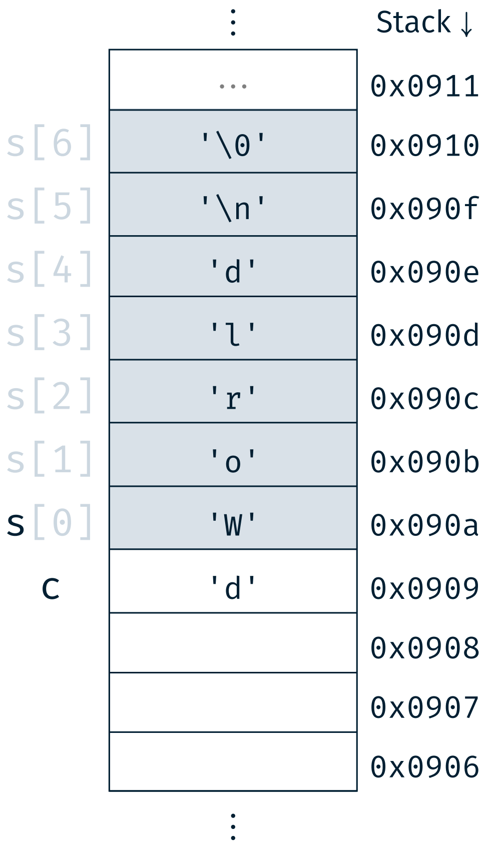


	⋮	Stack ↓
	...	0x0911
s[6]	'\0'	0x0910
s[5]	'\n'	0x090f
s[4]	'd'	0x090e
s[3]	'l'	0x090d
s[2]	'r'	0x090c
s[1]	'o'	0x090b
s[0]	'W'	0x090a
c	'W'	0x0909
		0x0908
		0x0907
		0x0906
	⋮	

In Depth: Strings

- char: Single character (e. g. 'a')
- String: Array of chars (e. g. "Hello")
- C: Last char of a string: '\0'
  - ⇒ Memory requirement: strlen(s) + 1

```c
char s[] = "World\n";
char c = s[0];
c = s[4];
char *s2 = s + 2;
c = s2[1];
```



⋮ Stack ↓

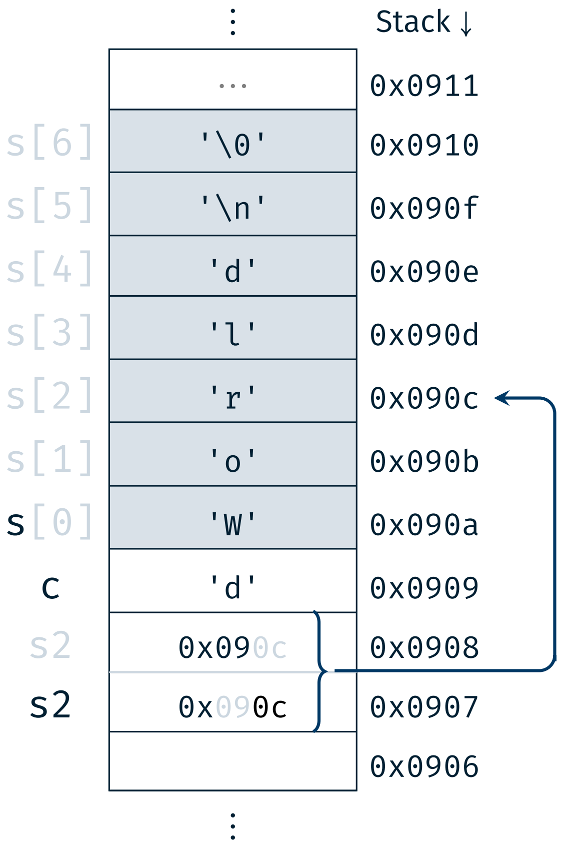
	...	0x0911
s[6]	'\0'	0x0910
s[5]	'\n'	0x090f
s[4]	'd'	0x090e
s[3]	'l'	0x090d
s[2]	'r'	0x090c
s[1]	'o'	0x090b
s[0]	'W'	0x090a
c	'd'	0x0909
		0x0908
		0x0907
		0x0906

⋮

In Depth: Strings

- char: Single character (e. g. 'a')
- String: Array of chars (e. g. "Hello")
- C: Last char of a string: '\0'
  - ⇒ Memory requirement: strlen(s) + 1

```c
char s[] = "World\n";
char c = s[0];
c = s[4];
char *s2 = s + 2;
c = s2[1];
```



⋮ Stack ↓

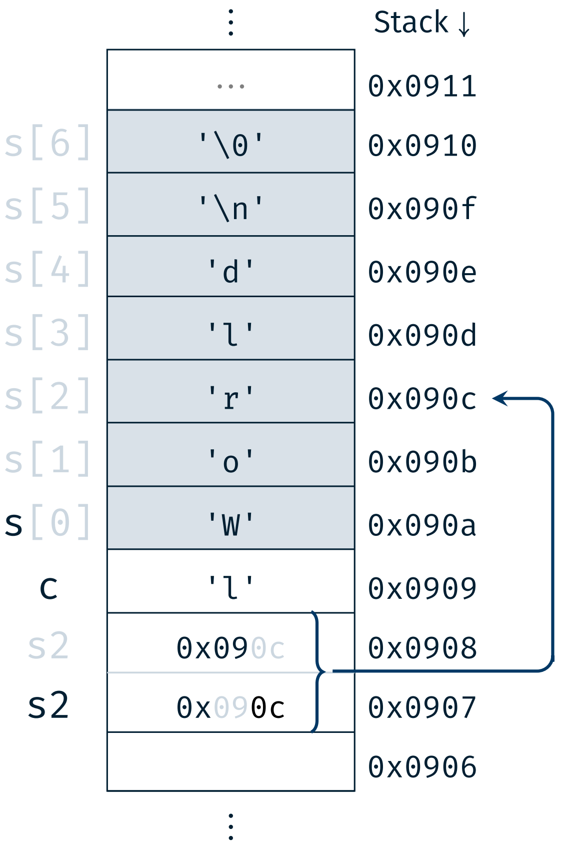
	...	0x0911
s[6]	'\0'	0x0910
s[5]	'\n'	0x090f
s[4]	'd'	0x090e
s[3]	'l'	0x090d
s[2]	'r'	0x090c
s[1]	'o'	0x090b
s[0]	'W'	0x090a
c	'd'	0x0909
s2	0x090c	0x0908
s2	0x090c	0x0907
		0x0906

⋮

In Depth: Strings

- char: Single character (e. g. 'a')
- String: Array of chars (e. g. "Hello")
- C: Last char of a string: '\0'
  - ⇒ Memory requirement: strlen(s) + 1

```c
char s[] = "World\n";
char c = s[0];
c = s[4];
char *s2 = s + 2;
c = s2[1];
```



	⋮	Stack ↓
	…	0x0911
s[6]	'\0'	0x0910
s[5]	'\n'	0x090f
s[4]	'd'	0x090e
s[3]	'l'	0x090d
s[2]	'r'	0x090c
s[1]	'o'	0x090b
s[0]	'W'	0x090a
c	'l'	0x0909
s2	0x090c	0x0908
s2	0x090c	0x0907
		0x0906
	⋮	

String Functions

- `size_t strlen(const char *s)`
  - Determine the length of a string s (without trailing NULL character)
- `char *strcpy(char *dest, const char *src)`
  - Copy a string `src` into a buffer `dest` (including NULL character)
  - Caution: Buffer overflow (⇒ `strncpy(3)`)
- `char *strcat(char *dest, const char *src)`
  - Concatenate a string `src` after an existing string inside the buffer `dest` (including NULL character)
  - Caution: Buffer overflow (⇒ `strncat(3)`)
- Documentation: `strlen(3)`, `strcpy(3)`, `strcat(3)`

String Functions – Exmaple

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

int main(void) {
  const char *hello = "Hello";
  const char *spic = "SPiC";

  char altered_string[11];  // Space for "Hello SPiC"

  strcpy(altered_string, hello);  // "Hello"
  strcat(altered_string, " ");    // "Hello "
  strcat(altered_string, spic);   // "Hello SPiC"
  strlen(altered_string);         // -> 10

  return EXIT_SUCCESS;
}
```

Assignment: concat

Assignment: concat

- Concatenate the passed command line arguments into a single string and output of this string
- Procedure:
  - determine the required length
  - allocate the buffer dynamically
  - fill the buffer step by step
  - output the string on stdout
  - free the dynamically allocated buffer

- Re-implement the string library functions (from `string.h`):
- Important: identical behaviour (even in case of an error)

```
  size_t str_len(const char *s)
  char  *str_cpy(char *dest, const char *src)
  char  *str_cat(char *dest, const char *src)
```

Dynamic Management of Memory

- `malloc(3)` allocates memory on the heap
  - reserves a minimum of `size` bytes of memory
  - returns a pointer to the start of the allocated memory
  - can potentially return an error
- `free(3)` frees the allocated memory again

```
char* s = (char *) malloc(...);
if(s == NULL) {
  perror("malloc");
  exit(EXIT_FAILURE);
}

// [...]

free(s);
```

Hands-on: Buffer Overflow

Hands-on: Buffer Overflow

- Program secured with a password

```
# Usage: ./print_exam <password>
./print_exam spic
Correct Password
Printing exam...
```

Hands-on: Buffer Overflow

- Program secured with a password

```
# Usage: ./print_exam <password>
./print_exam spic
Correct Password
Printing exam...
```

- Unchecked user inputs ⇒ buffer overflow

```
long check_password(const char *password) {
  char buff[8];
  long pass = 0;

  strcpy(buff, password);
  if(strcmp(buff, "spic") == 0) {
    pass = 1;
  }
  return pass;
}
```

Hands-on: Buffer Overflow

- Program secured with a password

```
# Usage: ./print_exam <password>
./print_exam spic
Correct Password
Printing exam...
```

- Unchecked user inputs ⇒ buffer overflow

```
long check_password(const char *password) {
  char buff[8];
  long pass = 0;

  strcpy(buff, password);
  if(strcmp(buff, "spic") == 0) {
    pass = 1;
  }
  return pass;
}
```

Hands-on: Buffer Overflow

```c
long check_password(const char *password) {
  char buff[8];
  long pass = 0;

  strcpy(buff, password);
  if(strcmp(buff, "spic") == 0) {
    pass = 1;
  }
  return pass;
}
```

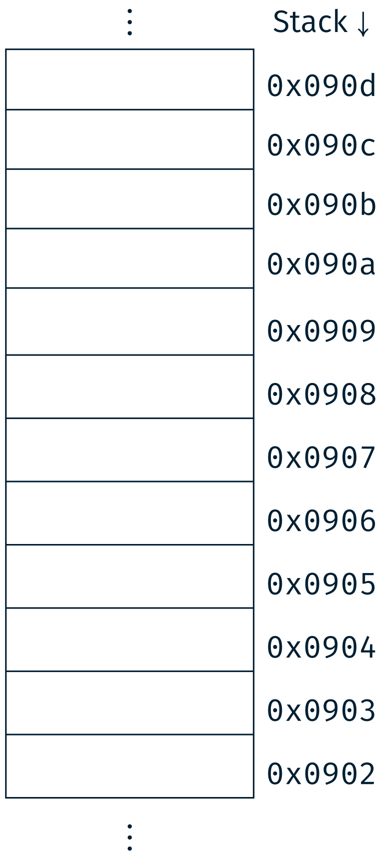
- Possible solutions
  - Check the user input
  - Allocate the buffer dynamically
  - Use of secure library functions ⇒ z.B. `strncpy(3)`

Buffer Overflow

```c
long pass = 0;
char buff[8];
strcpy(buff, password);

if(strcmp(buff, "spic")) {
  printf("Wrong Pass.\n");
} else {
  printf("Correct Pass.\n");
  pass = 1;
}

return pass;
```

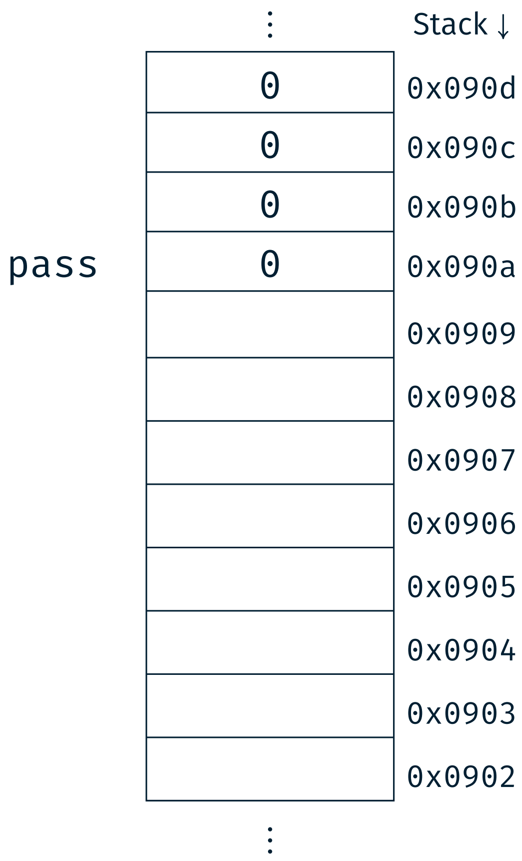


Buffer Overflow

```c
long pass = 0;
char buff[8];
strcpy(buff, password);

if(strcmp(buff, "spic")) {
  printf("Wrong Pass.\n");
} else {
  printf("Correct Pass.\n");
  pass = 1;
}

return pass;
```



Stack ↓

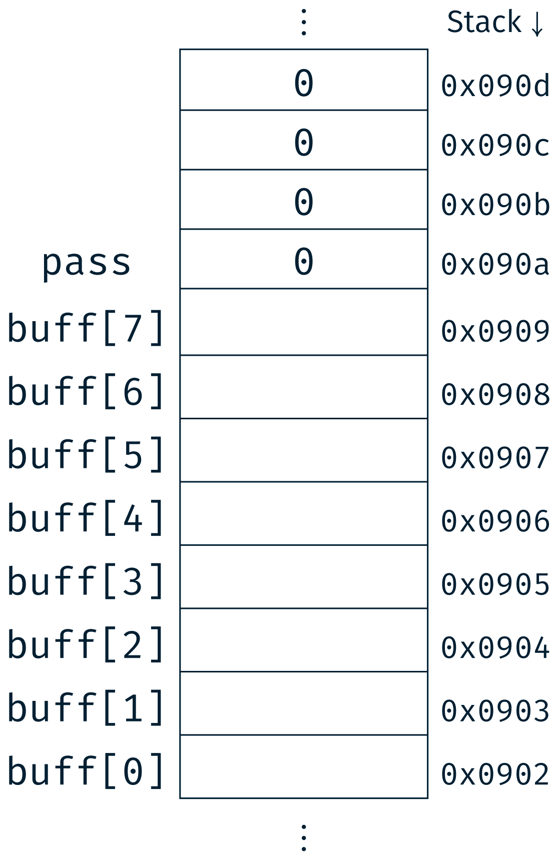
	⋮	
	0	0x090d
	0	0x090c
	0	0x090b
pass	0	0x090a
		0x0909
		0x0908
		0x0907
		0x0906
		0x0905
		0x0904
		0x0903
		0x0902
	⋮	

Buffer Overflow

```c
long pass = 0;
char buff[8];
strcpy(buff, password);

if(strcmp(buff, "spic")) {
  printf("Wrong Pass.\n");
} else {
  printf("Correct Pass.\n");
  pass = 1;
}

return pass;
```



Stack ↓

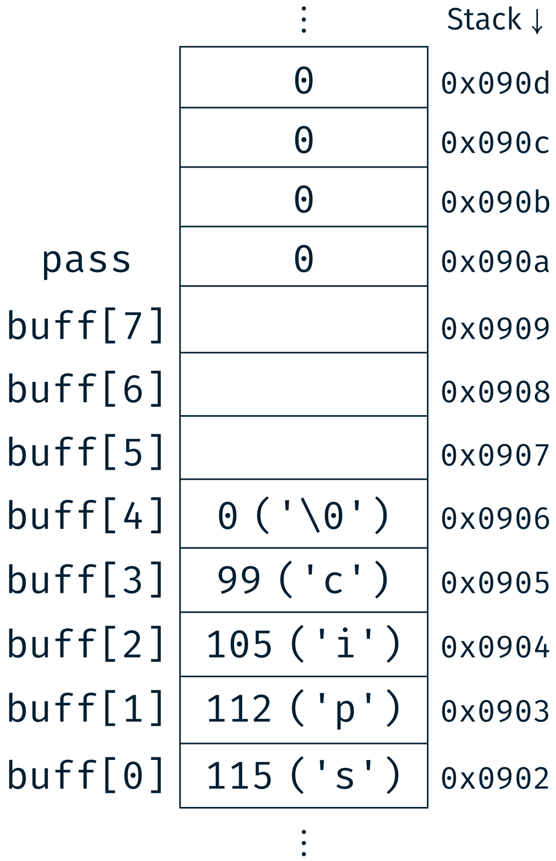
	⋮	
	0	0x090d
	0	0x090c
	0	0x090b
pass	0	0x090a
buff[7]		0x0909
buff[6]		0x0908
buff[5]		0x0907
buff[4]		0x0906
buff[3]		0x0905
buff[2]		0x0904
buff[1]		0x0903
buff[0]		0x0902
	⋮	

Buffer Overflow

```c
long pass = 0;
char buff[8];
strcpy(buff, password);

if(strcmp(buff, "spic")) {
  printf("Wrong Pass.\n");
} else {
  printf("Correct Pass.\n");
  pass = 1;
}

return pass;
```



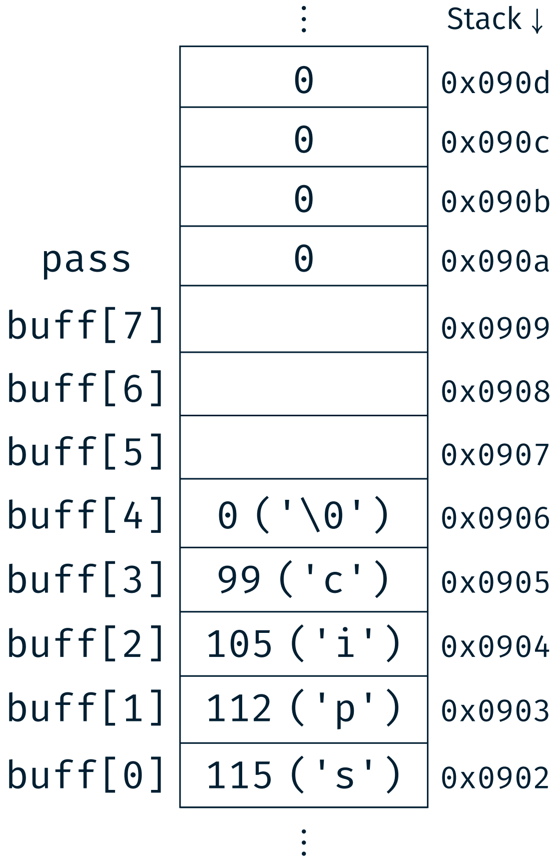
	⋮	Stack ↓
	0	0x090d
	0	0x090c
	0	0x090b
pass	0	0x090a
buff[7]		0x0909
buff[6]		0x0908
buff[5]		0x0907
buff[4]	0 ('\0')	0x0906
buff[3]	99 ('c')	0x0905
buff[2]	105 ('i')	0x0904
buff[1]	112 ('p')	0x0903
buff[0]	115 ('s')	0x0902
	⋮	

Buffer Overflow

```c
long pass = 0;
char buff[8];
strcpy(buff, password);

if(strcmp(buff, "spic")) {
  printf("Wrong Pass.\n");
} else {
  printf("Correct Pass.\n");
  pass = 1;
}

return pass;
```



	⋮	Stack ↓
	0	0x090d
	0	0x090c
	0	0x090b
pass	0	0x090a
buff[7]		0x0909
buff[6]		0x0908
buff[5]		0x0907
buff[4]	0 ('\0')	0x0906
buff[3]	99 ('c')	0x0905
buff[2]	105 ('i')	0x0904
buff[1]	112 ('p')	0x0903
buff[0]	115 ('s')	0x0902
	⋮	

Buffer Overflow

```c
long pass = 0;
char buff[8];
strcpy(buff, password);

if(strcmp(buff, "spic")) {
  printf("Wrong Pass.\n");
} else {
  printf("Correct Pass.\n");
  pass = 1;
}

return pass;
```

⋮ Stack ↓

	0	0x090d
	0	0x090c
	0	0x090b
pass	0	0x090a
buff[7]		0x0909
buff[6]		0x0908
buff[5]		0x0907
buff[4]	0 ('\0')	0x0906
buff[3]	99 ('c')	0x0905
buff[2]	105 ('i')	0x0904
buff[1]	112 ('p')	0x0903
buff[0]	115 ('s')	0x0902

⋮

Buffer Overflow

```c
long pass = 0;
char buff[8];
strcpy(buff, password);

if(strcmp(buff, "spic")) {
  printf("Wrong Pass.\n");
} else {
  printf("Correct Pass.\n");
  pass = 1;
}

return pass;
```

⋮ Stack ↓

	0	0x090d
	0	0x090c
	0	0x090b
pass	1	0x090a
buff[7]		0x0909
buff[6]		0x0908
buff[5]		0x0907
buff[4]	0 ('\0')	0x0906
buff[3]	99 ('c')	0x0905
buff[2]	105 ('i')	0x0904
buff[1]	112 ('p')	0x0903
buff[0]	115 ('s')	0x0902

⋮

Buffer Overflow

```c
long pass = 0;
char buff[8];
strcpy(buff, password);

if(strcmp(buff, "spic")) {
  printf("Wrong Pass.\n");
} else {
  printf("Correct Pass.\n");
  pass = 1;
}

return pass; // pass = 1
             // --> true
```

	⋮	Stack ↓
	0	0x090d
	0	0x090c
	0	0x090b
pass	1	0x090a
buff[7]		0x0909
buff[6]		0x0908
buff[5]		0x0907
buff[4]	0 ('\0')	0x0906
buff[3]	99 ('c')	0x0905
buff[2]	105 ('i')	0x0904
buff[1]	112 ('p')	0x0903
buff[0]	115 ('s')	0x0902
	⋮	

Buffer Overflow

```c
long pass = 0;
char buff[8];
strcpy(buff, password);

if(strcmp(buff, "spic")) {
  printf("Wrong Pass.\n");
} else {
  printf("Correct Pass.\n");
  pass = 1;
}

return pass;
```

	⋮	Stack ↓
	0	0x090d
	0	0x090c
	0	0x090b
pass	0	0x090a
buff[7]		0x0909
buff[6]		0x0908
buff[5]		0x0907
buff[4]		0x0906
buff[3]		0x0905
buff[2]		0x0904
buff[1]		0x0903
buff[0]		0x0902
	⋮	

Buffer Overflow

```c
long pass = 0;
char buff[8];
strcpy(buff, password);

if(strcmp(buff, "spic")) {
  printf("Wrong Pass.\n");
} else {
  printf("Correct Pass.\n");
  pass = 1;
}

return pass;
```

	⋮	Stack ↓
	0	0x090d
	0	0x090c
	0	0x090b
pass	0	0x090a
buff[7]		0x0909
buff[6]		0x0908
buff[5]		0x0907
buff[4]		0x0906
buff[3]	0 ('\0')	0x0905
buff[2]	111 ('o')	0x0904
buff[1]	111 ('o')	0x0903
buff[0]	102 ('f')	0x0902
	⋮	

Buffer Overflow

```c
long pass = 0;
char buff[8];
strcpy(buff, password);

if(strcmp(buff, "spic")) {
  printf("Wrong Pass.\n");
} else {
  printf("Correct Pass.\n");
  pass = 1;
}

return pass;
```

	⋮	Stack ↓
	0	0x090d
	0	0x090c
	0	0x090b
pass	0	0x090a
buff[7]		0x0909
buff[6]		0x0908
buff[5]		0x0907
buff[4]		0x0906
buff[3]	0 ('\0')	0x0905
buff[2]	111 ('o')	0x0904
buff[1]	111 ('o')	0x0903
buff[0]	102 ('f')	0x0902
	⋮	

Buffer Overflow

```c
long pass = 0;
char buff[8];
strcpy(buff, password);

if(strcmp(buff, "spic")) {
  printf("Wrong Pass.\n");
} else {
  printf("Correct Pass.\n");
  pass = 1;
}

return pass;
```

⋮ Stack ↓

	0	0x090d
	0	0x090c
	0	0x090b
pass	0	0x090a
buff[7]		0x0909
buff[6]		0x0908
buff[5]		0x0907
buff[4]		0x0906
buff[3]	0 ('\0')	0x0905
buff[2]	111 ('o')	0x0904
buff[1]	111 ('o')	0x0903
buff[0]	102 ('f')	0x0902

⋮

Buffer Overflow

```c
long pass = 0;
char buff[8];
strcpy(buff, password);

if(strcmp(buff, "spic")) {
  printf("Wrong Pass.\n");
} else {
  printf("Correct Pass.\n");
  pass = 1;
}

return pass; // pass = 0
             // --> false
```

⋮ Stack ↓

	0	0x090d
	0	0x090c
	0	0x090b
pass	0	0x090a
buff[7]		0x0909
buff[6]		0x0908
buff[5]		0x0907
buff[4]		0x0906
buff[3]	0 ('\0')	0x0905
buff[2]	111 ('o')	0x0904
buff[1]	111 ('o')	0x0903
buff[0]	102 ('f')	0x0902

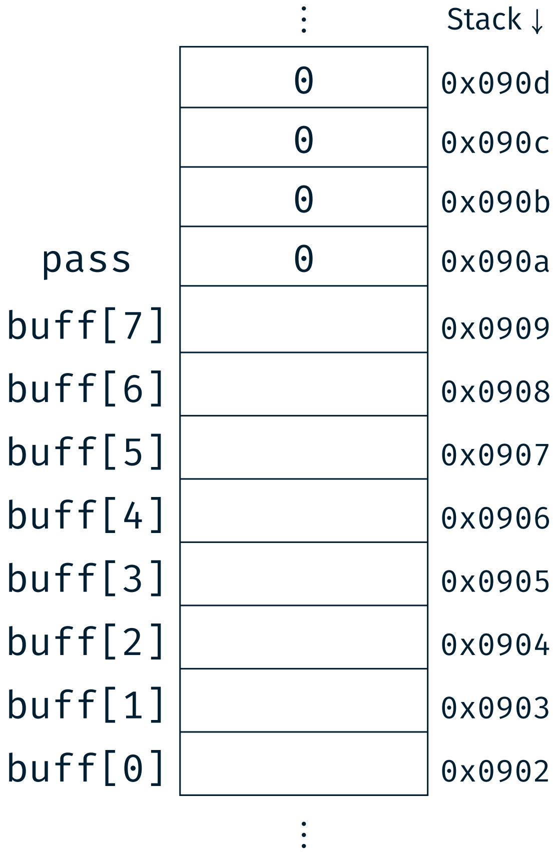
⋮

Buffer Overflow

```c
long pass = 0;
char buff[8];
strcpy(buff, password);

if(strcmp(buff, "spic")) {
  printf("Wrong Pass.\n");
} else {
  printf("Correct Pass.\n");
  pass = 1;
}

return pass;
```



Stack ↓

⋮

	0	0x090d
	0	0x090c
	0	0x090b
pass	0	0x090a
buff[7]		0x0909
buff[6]		0x0908
buff[5]		0x0907
buff[4]		0x0906
buff[3]		0x0905
buff[2]		0x0904
buff[1]		0x0903
buff[0]		0x0902

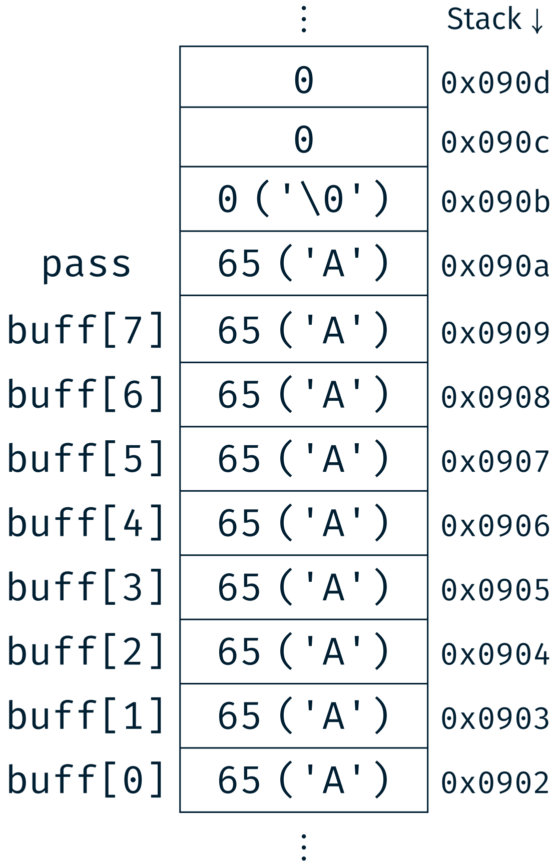
⋮

Buffer Overflow

```c
long pass = 0;
char buff[8];
strcpy(buff, password);

if(strcmp(buff, "spic")) {
  printf("Wrong Pass.\n");
} else {
  printf("Correct Pass.\n");
  pass = 1;
}

return pass;
```



Stack ↓

⋮

	0	0x090d
	0	0x090c
	0 ('\0')	0x090b
pass	65 ('A')	0x090a
buff[7]	65 ('A')	0x0909
buff[6]	65 ('A')	0x0908
buff[5]	65 ('A')	0x0907
buff[4]	65 ('A')	0x0906
buff[3]	65 ('A')	0x0905
buff[2]	65 ('A')	0x0904
buff[1]	65 ('A')	0x0903
buff[0]	65 ('A')	0x0902

⋮

Buffer Overflow

```c
long pass = 0;
char buff[8];
strcpy(buff, password);

if(strcmp(buff, "spic")) {
  printf("Wrong Pass.\n");
} else {
  printf("Correct Pass.\n");
  pass = 1;
}

return pass;
```

	⋮	Stack ↓
	0	0x090d
	0	0x090c
	0 ('\0')	0x090b
pass	65 ('A')	0x090a
buff[7]	65 ('A')	0x0909
buff[6]	65 ('A')	0x0908
buff[5]	65 ('A')	0x0907
buff[4]	65 ('A')	0x0906
buff[3]	65 ('A')	0x0905
buff[2]	65 ('A')	0x0904
buff[1]	65 ('A')	0x0903
buff[0]	65 ('A')	0x0902
	⋮	

Buffer Overflow

```c
long pass = 0;
char buff[8];
strcpy(buff, password);

if(strcmp(buff, "spic")) {
  printf("Wrong Pass.\n");
} else {
  printf("Correct Pass.\n");
  pass = 1;
}

return pass;
```

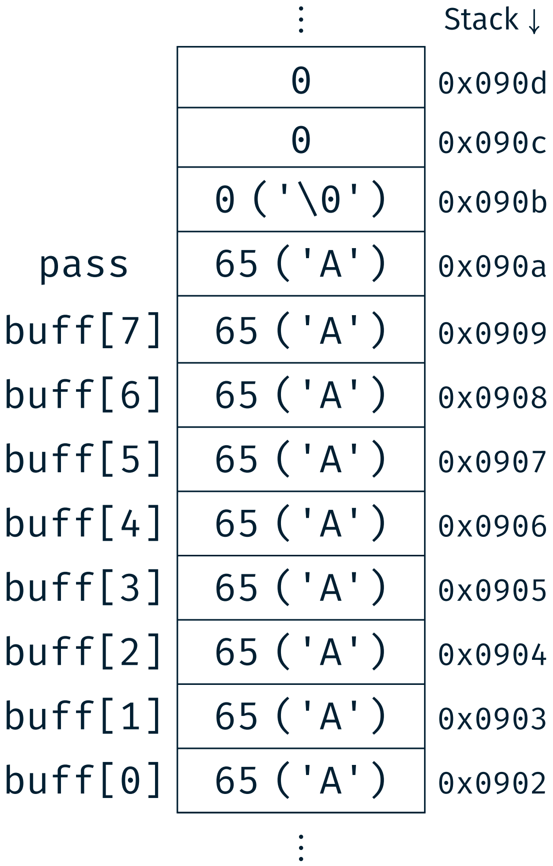
	⋮	Stack ↓
	0	0x090d
	0	0x090c
	0 ('\0')	0x090b
pass	65 ('A')	0x090a
buff[7]	65 ('A')	0x0909
buff[6]	65 ('A')	0x0908
buff[5]	65 ('A')	0x0907
buff[4]	65 ('A')	0x0906
buff[3]	65 ('A')	0x0905
buff[2]	65 ('A')	0x0904
buff[1]	65 ('A')	0x0903
buff[0]	65 ('A')	0x0902
	⋮	

Buffer Overflow

```c
long pass = 0;
char buff[8];
strcpy(buff, password);

if(strcmp(buff, "spic")) {
  printf("Wrong Pass.\n");
} else {
  printf("Correct Pass.\n");
  pass = 1;
}

return pass; // pass = 65
             // --> true
```



Stack ↓

	⋮	
	0	0x090d
	0	0x090c
	0 ('\0')	0x090b
pass	65 ('A')	0x090a
buff[7]	65 ('A')	0x0909
buff[6]	65 ('A')	0x0908
buff[5]	65 ('A')	0x0907
buff[4]	65 ('A')	0x0906
buff[3]	65 ('A')	0x0905
buff[2]	65 ('A')	0x0904
buff[1]	65 ('A')	0x0903
buff[0]	65 ('A')	0x0902
	⋮	

Hands-on: Linux, GCC & Valgrind

Screencast: `https://www.video.uni-erlangen.de/clip/id/18667`

Hands-on: Linux, GCC & Valgrind

- *Only online!*
- Goals:
    - Use SPiC IDE for Linux
    - Compile program from the command line
    - Practice th use of valgrind