Part 2.3 - The C Language

ECE_3TC31_TP/INF107

Florian Brandner

2024

The Standard Library

Reporting Errors (1)

```
void perror( const char *s );
```

Most functions of the C library report errors:

- Often as a special return value.
  - The return value does not explain why the error occurred.
- Many functions in addition set the **global variable** `errno`**:**
  - Defined in header `errno.h`.
  - Allows to store an error code as a number.
- `perror` allows to print a *readable* error message:
  - Defined in header `stdio.h`.
  - Based on the value of `errno`.
  - First *displays* the string `s`, provided as argument, followed by `": "`, and then `errno` explained

https://en.cppreference.com/w/c/io/perror
https://en.cppreference.com/w/c/error/errno

Reporting Errors (2)

```c
_Noreturn void exit( int exit_code );
```

Sometimes it does not make sens to continue after an error:

- First the error should be reported (e.g., using `perror`).
- Then the program should be terminated using the `exit` function:
  - Calling this function ends the program.
  - It takes an exit code as argument.
  - This code has the same meaning as the return value of `main`.
  - `EXIT_FAILURE` can be used to indicate an error.
- Of course it is possible to call `exit` without an error as well.
  - Use `EXIT_SUCCESS` in this case.

https://en.cppreference.com/w/c/program/exit

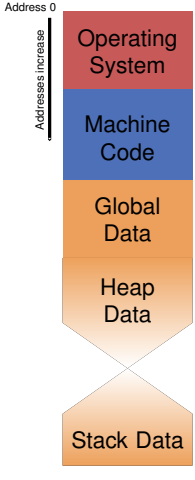
Managing Heap Memory

Memory Organization

The usual organization of the processor's memory:

- A part of the memory is reserved for the operating system.
- Another part for the machine code of the program.
- The rest is for storing data of the program:
  - *Global data*, accessible all the time.
  - *Stack data*, accessible only temporarily.
  - *Heap data*, explicitly managed by the programmer.

How can one use heap memory?



Heap Memory

Heap memory is managed explicitly by the programmer:

- The compiler does not place data or code on the heap by itself.
- The programmer explicitly has to:
  - **Allocate:**
    Reserve an address range on the heap, depending on the size of data to be stored.
  - **Free:**
    Free an allocated address range, when the data there is not needed any longer.
- This is important so that other heap operations do not accidentally modify data there.

Heap Memory Allocation

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

- Defined in the header `stdlib.h`.
- Reserves an address range of the given `size` (in bytes):
  - Returns the starting address as a pointer.
  - Returns a **null pointer**, if the heap is full (no memory space available).
- **Example**:

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

int main(int argc, char *argv[]) {
  int *heapData = malloc(sizeof(int)*4);   // Allocate an array of 4 integers on the heap.
  if (!heapData) {
    perror("malloc failed");               // Print error message, if any.
    return EXIT_FAILURE;
  }
  else return EXIT_SUCCESS;                // All memory is freed implicitly.
}
```

https://en.cppreference.com/w/c/memory/malloc

Freeing Heap Memory

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

- Defined in the header `stdlib.h`.
- Takes a valid heap pointer and frees it:
  - Valid pointer: allocated using `malloc` or a similar function and not freed before.
  - Otherwise the result is **undefined**.
- **Example**:

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
  int *heapData = malloc(sizeof(int)*4);
  if (!heapData) {
    perror("malloc failed");              // Print error message, if any.
    return EXIT_FAILURE;
  } else {
    free(heapData);                       // Explicitly free heap data.
    return EXIT_SUCCESS;
  }
}
```

https://en.cppreference.com/w/c/memory/free

Heap Memory Reallocation

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

- Defined in the header `stdlib.h`.
- Takes a heap pointer and changes the size of the reserved address range:
  - The new size can be smaller or larger.
  - It may or may not return the same pointer.
  - If the pointer changes, data is copied using the old size.
- The pointer needs to be a valid heap pointer (see `free`).
- Returns a **null pointer**, if the heap is full (no memory space available).
- **Example**:

```c
int main(int argc, char *argv[]) {
  int *heapData = malloc(sizeof(int)*4);
  if (heapData = realloc(heapData, sizeof(int))) {  // Free some space, but not all.
    perror("realloc failed");
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

https://en.cppreference.com/w/c/memory/realloc

File Input and Output

Manipulating Files

Files are managed by the operating system (OS):

- One has to tell the OS, which files will be manipulated and how.
- **Opening a file:**
  Tell the OS that a file is going to be manipulated.
- **Closing a file:**
  Tell the OS that the program no longer manipulates the file.
- **File operations:**
  Tell the OS that the file will be read and/or (over-)written.

The C standard library provide a higher-level API for manipulating files, based on the abstraction of (buffered) **I/O streams**.

Opening a File (1)

```
FILE *fopen( const char *filename, const char *mode );
```

- Defined in the header `stdio.h`.
- Open the given file for input/output operations.
  - `filename`:
    The name of the file to open.
  - `mode`:
    Indicates how the file will be manipulated (see next page).
- Returns a pointer to a `FILE` structure:
  - This is called a **file stream**.
  - The data structure itself is **unspecified** – programmers only manipulate the pointer.
  - On error a **null pointer** is returned.

https://en.cppreference.com/w/c/io/fopen
https://en.cppreference.com/w/c/io/FILE

Opening a File (2)

Possible modes for `fopen`:

mode	Description	File exists	File does not exist
`"r"`	Reading only	Read from start	Error
`"w"`	Writing only	Overwrite file	Create file
`"a"`	Writing only	Append at end	Create file
`"r+"`	Read/write	Read/write from start	Error
`"w+"`	Read/write	Overwrite file	Create file
`"a+"`	Read/write	Read/write at end	Create file

https://en.cppreference.com/w/c/io/fopen

Closing a File

```
int fclose( FILE *stream );
```

- Defined in the header `stdio.h`.
- Closes the given file:
  - The OS makes sure that all data is written to the storage device.
  - The file can no longer be manipulated using the `FILE` pointer.
- Returns 0 on success
  - On error the constant `EOF` is returned.
    (some negative value, often -1)

https://en.cppreference.com/w/c/io/fclose

Special File Streams

The C library defines three special file streams:

- Defined in the header `stdio.h`.
- `stdout`:
  A stream (aka. a `FILE` structure) for regular output (write only).
- `stderr`:
  A stream to report errors (write only).
- `stdin`:
  A stream to read input (read only).
- These streams are opened automatically on program start.

https://en.cppreference.com/w/c/io/std_streams

Example: File Output (1)

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

int main(int argc, char *argv[]) {
  FILE *f = fopen("output.txt", "w");      // Open/create file for writing.
  if (!f) {
    perror("fopen failed");
    return EXIT_FAILURE;
  }
  else {
    fprintf(f, "Hello World %d\n", 57);    // Write some text into the file.

    if (fclose(f)) {                       // Close the file.
      perror("fclose failed");             // Report an error message (on stderr)
      return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
  }
}
```

Content of `file-output.c`.

Example: File Output (2)

```
tp-5b07-26:~/tmp> ls
file-output.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g file-output.c -o file-output

tp-5b07-26:~/tmp> ls
file-output  file-output.c

tp-5b07-26:~/tmp> ./file-output

tp-5b07-26:~/tmp> ls
file-output  file-output.c  output.txt

tp-5b07-26:~/tmp> cat output.txt
Hello World 57
```

Formatted File Output

```c
int printf( const char *restrict format, ... );
int fprintf( FILE *restrict stream, const char *restrict format, ... );
int sprintf( char *restrict buffer, const char *restrict format, ... );
```

- Same principle as plain `printf`:
  - `fprintf`:
    - Takes an additional file argument.
    - Writes into that file.
  - `printf`:
    - Actually the same as `fprintf` writing to `stdout`.
  - `sprintf`: takes an additional file argument.
    - Takes a pointer to an array as additional argument.
    - Writes into that array (buffer).
- Format strings are the same.
- **Return value:**
  - Number of characters written to file/buffer.
  - On error a **negative number** is returned.

https://en.cppreference.com/w/c/io/fprintf

Format Strings Revisited (1)

`'%' <flags>? <width>? ('.' <width>?) <size modifier>? <format>`

- `width`:
  An integer number or `'*'`
  - For strings: Indicates the Number of characters to display.
  - For numbers: Number of digits to display (before or after the `'.'`).
- `flags`:
  On ore more of the following control characters:

Control Character	Description
`'-'`	Align printed text to the left (see `width`).
`'+'`	Always display sign symbol (+) also for positive numbers.
`' '`	Print a space instead of sign for positive numbers.
`'0'`	Padding with `'0'` characters instead of spaces (`' '`).

Format Strings Revisited (2)

- `modifier`:
  Indicate the type of integer and floating-point numbers to print.

Control String	Description
(default)	Expect a value of type `int` when printing numbers.
`"hh"`	Expect type `signed char`/`unsigned char`.
`"h"`	Expect type `short`/`unsigned short`.
`"l"`	Expect type `long`/`unsigned long` or `wchar_t`.
`"ll"`	Expect type `long long`/`unsigned long long`.
`"z"`	Expect type `size_t`.
`"L"`	Expect type `long double`.

Format Strings Revisited (3)

- `format`:

Control Character	Description
`'c'`	Displays a character symbol (use modifier).
`'d'`	Displays a signed integer value as decimal (use modifier).
`'u'`	Displays a unsigned integer value as decimal (use modifier).
`'x'`	Displays an integer value as hexadecimal (use modifier).
`'f'`	Displays an floating-point number as decimal (accepts modifier `L`).
`'e'`	Displays an floating-point number in exponent notation (accepts modifier `L`).
`'s'`	Displays all the characters of a string.
`'p'`	Displays the address of a pointer.

Example: Formatted Output (1)

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

int main(int argc, char *argv[])
{
  short s = 540;
  int i = 0xfbfb;
  float f = i * 1.133e5;
  static const char string[] = "|Left Aligned|";

  printf("size_t:%zd\n", sizeof(int));
  printf("Integer numbers (decimal): % hd and %+d\n", s, i);     // Printing of sign.
  printf("Integer numbers (hex): 0x%8x and 0x%08X\n", s, i);     // Padding with ' ' or '0'.
  printf("Floating-point numbers: %020.3f and %12.3e\n", f, f);  // Width before/after '.'.
  printf("String: %-20s is aligned\n", string);                  // Left aligned.

  return EXIT_SUCCESS;
}
```

Content of `print2.c`.

Example: Formatted Output (2)

```
tp-5b07-26:~/tmp> ls
print2.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g print2.c -o print2

tp-5b07-26:~/tmp> ls
print2 print2.c

tp-5b07-26:~/tmp> ./print2
size_t:4
Integer numbers (decimal):  540 and +64507
Integer numbers (hex): 0x     21c and 0x0000FBFB
Floating-point numbers: 0000007308643328.000 and    7.309e+09
String: |Left Aligned|       is aligned
```

Formatted File Input

```
int scanf( const char *restrict format, ... );
int fscanf( FILE *restrict stream, const char *restrict format, ... );
int sscanf( const char *restrict buffer, const char *restrict format, ... );
```

- Same principle as for `printf`:
  - `fscanf`:
    - Takes a file argument.
    - Reads from that file depending on format string.
  - `scanf`:
    - Actually the same as `fscanf` reading from `stdin`.
  - `sscanf`: takes an additional file argument.
    - Takes a pointer to an array as argument.
    - Reads from that array (buffer).
- Format strings are similar, but not quite the same.
- **Return value:**
  - Number of values read successfully.
  - Or EOF in case of an error.

https://en.cppreference.com/w/c/io/fscanf

Format Strings for Input (1)

Same basic idea as for `printf`:

- Regular characters (except whitespace and `'%'`):
  - Indicate that precisely such a character needs to be read.
  - Otherwise an error is signaled.
- Whitespace characters (e.g., `' '`, `'\t'`, `'\n'`, ...):
  - Consume all available consecutive whitespaces.
  - It is sufficient to indicate a single whitespace.
- The `'%` character again has special meaning:
  - It indicates that a value should be read from the file.
  - **Always expects an address where to store the read value, i.e., a pointer or array.**

https://en.cppreference.com/w/c/io/fscanf

Format Strings for Input (2)

```
'%' '*'? <width>? <size modifier>? <format>
```

- A preceding `*` indicates that the value should not be assigned to a pointer, just read.
- `width`:
  Indicates the maximum width to read (recommended particularly for strings).
- `size modifier`:
  Same as for `printf`, i.e., (`"hh"`, `"h"`, `"l"`, `"ll"`, `"z"`, and L).
- `format`:
  Same as for `printf`, i.e., (`'c'`, `'s'`, `'d'`, `'u'`, `'x'`, `f`, and `e`).
  - Reading a string with `"%s"` adds the null character.
    - Arrays/pointers receiving a string thus need size `width + 1`.
  - Indicating a width with `"%c"` reads `width` characters.
    - No null character is added.
  - Reading numbers skips leading whitespaces.

Example: Formatted Input (1)

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
  FILE *f = fopen(argv[1], "r");                  // Open file for reading.
  if (!f) {
    perror("fopen failed"); return EXIT_FAILURE;
  } else {
    char c, s[50];
    int i;
    int read = fscanf(f, "%s%d%c\n", s, &i, &c);  // Read a string, an integer, and a character.
    if (read != 3) {
      perror("fscanf failed"); return EXIT_FAILURE;
    }
    printf("%s\n%d\n0x%x\n", s, i, c);
    if (fclose(f)) {                              // Close the file.
      perror("fclose failed"); return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
  }
}
```

Content of `file-input.c`.

Example: Formatted Input (2)

```
tp-5b07-26:~/tmp> ls
file-input.c  input.txt

tp-5b07-26:~/tmp> cat input.txt
String_without_whitespace_plus_null_character
3
a

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g file-input.c -o file-input

tp-5b07-26:~/tmp> ls
file-input file-input.c input.txt

tp-5b07-26:~/tmp> ./file-input no-file.txt
fopen failed: No such file or directory

tp-5b07-26:~/tmp> ./file-input input.txt
String_without_whitespace_plus_null_character
3
0xa
```

Check Yourself!

Something (maybe) *unexpected* happened in the previous example.

1. Can you spot it?

   Hint:

   `hexdump` shows the content of `input.txt` as hexadecimal numbers (left) and characters (right).

```
tp-5b07-26:~/tmp> hexdump -C input.txt
00000000  53 74 72 69 6e 67 5f 77  69 74 68 6f 75 74 5f 77  |String_without_w|
00000010  68 69 74 65 73 70 61 63  65 5f 70 6c 75 73 5f 6e  |hitespace_plus_n|
00000020  75 6c 6c 5f 63 68 61 72  61 63 74 65 72 0a 33 0a  |ull_character.3.|
00000030  61 0a                                             |a.|
00000032
```

2. Correct the format string to get the intended behavior.

Answer

1. The character read by `scanf` is not the one that you might have expected:
   - The input file seemingly ends with a character `'a'`, right?
   - However, there are also *line feed* characters (see Escape Sequences).

```
tp-5b07-26:~/tmp> hexdump -C input.txt
00000000  53 74 72 69 6e 67 5f 77  69 74 68 6f 75 74 5f 77  |String_without_w|
00000010  68 69 74 65 73 70 61 63  65 5f 70 6c 75 73 5f 6e  |hitespace_plus_n|
00000020  75 6c 6c 5f 63 68 61 72  61 63 74 65 72 0a 33 0a  |ull_character.3.|
00000030  61 0a                                             |a.|
```

Character read!

   - The form feed character (`'\n'`, aka. `0xa`) was read, not the `'a'`.
2. It is sufficient to skip the whitespace(s) after reading the integer number, i.e., it suffices to insert a space after the `'d'`.

Example: Formatted Input Corrected (1)

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
  FILE *f = fopen(argv[1], "r");                  // Open file for reading.
  if (!f) {
    perror("fopen failed"); return EXIT_FAILURE;
  } else {
    char c, s[50];
    int i;
    int read = fscanf(f, "%s%d %c\n", s, &i, &c);  // Read a string, an integer, and a character.
    if (read != 3) {
      perror("fscanf failed"); return EXIT_FAILURE;
    }
    printf("%s\n%d\n0x%x\n", s, i, c);
    if (fclose(f)) {                              // Close the file.
      perror("fclose failed"); return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
  }
}
```

Content of `file-input2.c`.

Example: Formatted Input Corrected (2)

```
tp-5b07-26:~/tmp> ls
file-input2.c  input.txt

tp-5b07-26:~/tmp> cat input.txt
String_without_whitespace_plus_null_character
3
a

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g file-input2.c -o file-input2

tp-5b07-26:~/tmp> ls
file-input2 file-input2.c input.txt

tp-5b07-26:~/tmp> ./file-input2 no-file.txt
fopen failed: No such file or directory

tp-5b07-26:~/tmp> ./file-input2 input.txt
String_without_whitespace_plus_null_character
3
0x61
```

Strings

Strings are just arrays/pointers:

- Comparison (==) and assignment (=):
  - Do not compare/copy characters.
  - Instead **operate on pointers**.
  - May produce (maybe) confusing results:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
  char sa[] = "aaa";
  char sb[] = "aaa";
  // Always prints "false".
  if (sa == sb)
    printf("true\n");
  else
    printf("false\n");
  return EXIT_SUCCESS;
}
```

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
  char *sa = "aaa";
  char *sb = "aaa";
  // May print "true" or "false".
  if (sa == sb)
    printf("true\n");
  else
    printf("false\n");
  return EXIT_SUCCESS;
}
```

- Operator cannot be used to concatenate strings or numbers.

String Operations

```
int   strncmp( const char* lhs, const char* rhs, size_t count );
char *strncpy( char *restrict dest, const char *restrict src, size_t count );
char *strncat( char *restrict dest, const char *restrict src, size_t count );
char *strndup( const char *src, size_t size );
```

- Defined in header `string.h`.
- `strncmp`: Compare strings by lexicographic order, at most `count` characters. (returns 0 **when the strings are equal**, else positive or negative number).
- `strncpy`: Copy at most `count` characters from one string to another.
- `strncat`: Concatenate at most `count` characters of one string to another.
- `strndup`: Makes a copy of the string by allocating a new string on the heap.

https://en.cppreference.com/w/c/string/byte/strncmp
https://en.cppreference.com/w/c/string/byte/strncpy
https://en.cppreference.com/w/c/string/byte/strncat
https://en.cppreference.com/w/c/string/byte/strndup

Memory Area Operations

```c
int   memcmp( const void* lhs, const void* rhs, size_t count );
void *memcpy( void *restrict dest, const void *restrict src, size_t count );
void *memset( void *dest, int c, size_t count );
```

- Defined in header `string.h`.
- `memcmp`: Compare memory areas (**"byte strings"**) by lexicographic order, looking at the first `count` bytes.
  (returns 0 **when the memory areas are equal**, else positive or negative number).
- `memcpy`: Copy precisely `count` bytes from one area to another.
- `memset`: Set precisely `count` bytes to the value of `c`.

https://en.cppreference.com/w/c/string/byte/memcmp
https://en.cppreference.com/w/c/string/byte/memcpy
https://en.cppreference.com/w/c/string/byte/memset

Manipulating Addresses and Pointers

Manipulating Addresses and Pointers

In C memory addresses and pointers can be manipulated explicitly:

- **Casting:**
  - It is possible to cast integer values to pointers.
    (many aspects of this are **implementation-defined**)
  - It is possible to cast one pointer type to another.
    - Casts to/from `void*` (e.g., as for library functions before).
    - Casts to/from `char*` to access individual bytes.
- **Pointer Arithmetic:**
  - It is possible to address objects relative to pointers using `+/-` or `[]`.
  - Increment/decrement pointers.
- In all cases:
  - Alignment has to be respected.
  - Computed addresses using pointers have to be valid.
    (correspond to a global, local, or heap object)
  - Otherwise the result is usually **undefined**.

Addresses and Array Subscripting

Access the n-th element from the start of the pointer/array:

```c
#include <stdio.h>

// Array, stored sequentially in memory.
int array[] = {1, 2, 3, 4, 5, 6};

int main(int argc, char *argv[]) {
  // Point to beginning of array.
  int *ptr = array;

  printf("ptr    : %p\n", ptr);
  printf("&ptr[2]: %p\n", &ptr[2]);
  printf("ptr[2] : %d\n", ptr[2]);

  return EXIT_SUCCESS;
}
```

```
tp-5b07-26:~/tmp> ./pointer-address
ptr    : 0x402020
&ptr[2]: 0x402028
ptr[2] : 3
```

- Memory layout:

Address	64-bit Value: 32-bit Value	32-bit Value	Value of
0x402020	0x00000001	0x00000002	array
0x402028	0x00000003	0x00000004	array
0x402030	0x00000005	0x00000006	array
…	…	…	…
0x7f....	0x0000000000402020		ptr

- Memory computation:
  `0x402020` $+ 2 *$ `sizeof(int)`

Addresses and Pointer Arithmetic

Actually is the same as array subscripting:

```c
#include <stdio.h>

// Array, stored sequentially in memory.
int array[] = {1, 2, 3, 4, 5, 6};

int main(int argc, char *argv[]) {
  // Point to beginning of array.
  int *ptr = array;

  printf("ptr       : %p\n", ptr);
  printf("ptr + 2   : %p\n", ptr + 2);
  printf("*(ptr + 2): %d\n", *(ptr + 2));

  return EXIT_SUCCESS;
}
```

```
tp-5b07-26:~/tmp> ./pointer-address2
ptr       : 0x402020
ptr + 2   : 0x402028
*(ptr + 2): 3
```

- Memory layout:

Address	64-bit Value: 32-bit Value	32-bit Value	Value of
0x402020	0x00000001	0x00000002	array
0x402028	0x00000003	0x00000004	array
0x402030	0x00000005	0x00000006	array
…	…	…	…
0x7f....	0x0000000000402020		ptr

- Memory computation:

  $\texttt{0x402020} + 2 * \texttt{sizeof(int)}$

Pointer Arithmetic

Array subscripting and pointer arithmetic are similar:

- Compute an address relative to start of the array/pointer
  - `<Pointer-address>` + `<Index>` * `sizeof( <Type> )` or `<Pointer-address>` − `<Index>` * `sizeof( <Type> )`
  - The computed address might be smaller or larger than the pointer's address.
  - The index is automatically scaled by the element size (`sizeof`).
- Pointers can also be incremented or decremented
  - Use the pre-/post- inc-/decrement operators (`++` or `--`).
  - The address computation is the same.
  - Writes resulting address back into the pointer.

Example: Pointer Arithmetic (1)

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

// Array, stored sequentially in memory.
int array[] = {1, 2, 3, 4, 5, 6};

int main(int argc, char *argv[]) {
  int *ptr = array;                          // Point to beginning of array.
  for(int i = 0; i < 3; i++) {               // Increment pointer (element-wise).
    printf("ptr: %p\t*ptr: ", ptr);
    printf("0x%08x ", *ptr++); printf("0x%08x\n", *ptr++);
  }

  printf("\nbyteswise:\n");                  // Increment pointer (byte-wise).
  for(char *charptr = (char*)array; charptr < (char*)array + sizeof(array); charptr++)
    printf("%02x", *charptr);
  printf("\n");

  return EXIT_SUCCESS;
}
```

Content of `pointer-increment.c`.

Example: Pointer Arithmetic (2)

```
tp-5b07-26:~/tmp> ls
pointer-increment.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g pointer-increment.c -o pointer-increment

tp-5b07-26:~/tmp> ls
pointer-increment pointer-increment.c

tp-5b07-26:~/tmp> ./pointer-increment
ptr: 0x402030   *ptr: 0x00000001 0x00000002
ptr: 0x402038   *ptr: 0x00000003 0x00000004
ptr: 0x402040   *ptr: 0x00000005 0x00000006

byteswise:
01000000020000000300000004000000050000000600000
```

Check Yourself!

```c
// Array, stored sequentially in memory.
int array[] = {1, 2, 3, 4, 5, 6};
int main(int argc, char *argv[]) {
  int *ptr = array;                          // Point to beginning of array.
  for(int i = 0; i < 3; i++)                 // Increment pointer (element-wise).
    printf("ptr: %p\t*ptr: 0x%08x 0x%08x\n", ptr, *ptr++, *ptr++);

  printf("\nbyteswise:\n");                  // Increment pointer (byte-wise).
  for(char *charptr = (char*)array; charptr < (char*)array + sizeof(array); charptr++)
    printf("%02x", *charptr);
  printf("\n");

  return EXIT_SUCCESS;
}
```

1. This expression `(char*)array + sizeof(array)` seems wrong, since the index is scaled by the element size of the pointer, i.e., the compiler should compute something like `sizeof(array) * sizeof(int)`. The code thus should read the memory beyond the array. Where is the error in this reasoning?
2. There is actually something problematic in this code? Can you spot it?

Answer

1. The precedence of the cast, which is stronger than the addition, the expression thus has to be read as `((char*)array) + sizeof(array)`. The reasoning from above is thus not entirely wrong. The only issue is the element size, which is not `sizeof(int)` but `sizeof(char)`, since the pointer arithmetic operates on a `char` pointer.

2. The following code line is problematic:

```
printf("ptr: %p\t*ptr: 0x%08x 0x%08x\n", ptr, *ptr++, *ptr++);
```

   - The evaluation order of the side-effects of call arguments is **unspecified**.
   - The same variable `ptr` is modified twice.
   - Consequently, the behavior of this expression is **undefined**.

Lab Exercises

Work with pointers, structures, dynamic memory allocation, formatted input and output:

- Find a delimiter character in an array.
  - Simple pointer/array manipulation.
- Define a structure and manipulate a simple structure.
  - Simple pointer/structure manipulation.
  - Formatted output (`fprintf`, ...).
- Read formatted data into a (static) structure.
  - Formatted input and output (`fscanf`, `fprintf`, ...).
  - String manipulation (`strndup`, ...).
  - Error handling (`perror`, `exit`).
- Manipulating a structure on the heap.
  - Pointer manipulation.
  - Memory management (`malloc`, `free`, `realloc`).