Concepts and Software Design for CPS

Lab 2: Introduction to C (Part 2)

Raphael Trumpp Binqi Sun

Chair of Cyber-Physical Systems in Production Engineering
Technical University of Munich
Munich, Germany

Lab 2: Overview

Compilation

Pointers and Arrays

Structures

File Input/Output

Assignment 2 (due: 11.11.2021)

Compilation

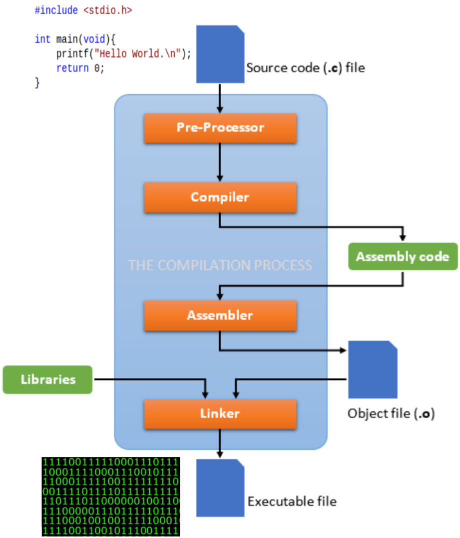
Compilation

The Four Stages of Compilation

Compilers turn source code written in C into binary machine code (*executable*). Usually we refer to a toolchain when talking about a compiler. The compiler is just one part of the toolchain.

Toolchains work in four stages:

- Preprocessing (Preprocessor)
- Compilation (Compiler)
- Assembly (Assembler)
- Linking (Linker)



The C preprocessor is used automatically by the C compiler to transform the code before compilation. The preprocessor:

- includes the text of header files (`#include`)
- removes comments (`//` and `/* */`)
- expands macros and constants (`#define`)
- conditionally eliminates parts of the files (`#ifdef`, `#endif`)

The following slides demonstrate these transformations.

Compilation

Preprocessing: Including Files

Before preprocessing:

```c
#include <stdio.h>

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

After preprocessing:

```c
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
...
extern int printf (const char *__restrict __format,
    ...);
...
void main() {
   printf("Hello, world!\n");
}
```

Compilation

Preprocessing: Constants and Comments

Before preprocessing:

```
#define X 10

void main() {
    // This is a comment in C
    /* This
       is
       a comment too */
    int x = X;
}
```

After preprocessing:

```
void main() {
    int x = 10;
}
```

Compilation

Preprocessing: Macros

Before preprocessing:

```c
#define SLICES 8
#define DIV(x) ( (x) / SLICES )

int main() {
    int a = 0, b = 10, c = 6;
    a = DIV(b + c);
    return a;
}
```

After preprocessing:

```c
int main() {
    int a = 0, b = 10, c = 6;
    a = ( (b + c) / 8 );
    return a;
}
```

Compilation

Preprocessing: Conditional compilation

The *if-else* directives `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif` and `#endif` are used for conditional compilation.

A common use for the *if-else* directives is to add platform specific source code into a program.

Before preprocessing:

```
#ifdef __unix__
#  include <unistd.h>
#elif defined _WIN32
#  include <windows.h>
#endif
```

After resolving conditionals on a Unix system:

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

After resolving conditionals on a Windows system:

```
#  include <windows.h>
```

Compilation

Compiler

A compiler translates from a programming language (*C code*) into a target language (*assembly code for a specific processor*).

```c
#include <stdio.h>

int main(void){
    printf("Hello World.\n");
    return 0;
}
```

→

```
section     .text
global      _start

_start:

    mov     edx,len
    mov     ecx,msg
    mov     ebx,1
    mov     eax,4
    int     0x80

    mov     eax,1
    int     0x80

section     .data

msg     db  'Hello, world!',0xa
len     equ $ - msg
```

The compiler performs:

- lexical analysis (e.g., finding all keywords, like: `for`, `if`, `float`)
- semantic analysis (i.e., determine whether a program makes sense)
- code optimization (e.g., a multiplication of a value by 2 might be more efficiently executed by left-shifting)
- code generation (i.e., replacing C statements with processor assembly instructions)

Compilation

Assembler

An assembler converts assembly code into binary machine code of a specific processor.

```c
#include <stdio.h>

int main(void){
    printf("Hello World.\n");
    return 0;
}
```

→

```asm
section     .text
global      _start

_start:

    mov     edx,len
    mov     ecx,msg
    mov     ebx,1
    mov     eax,4
    int     0x80

    mov     eax,1
    int     0x80

section     .data

msg     db  'Hello, world!',0xa
len     equ $ - msg
```

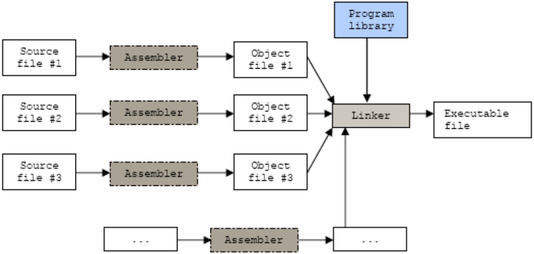
→

Every assembler has its own assembly language which is designed for exactly one specific computer architecture (e.g., *x86* or *ARM*).

Compilation

Linker

The linker combines one or more object files generated by the assembler into a single executable file.

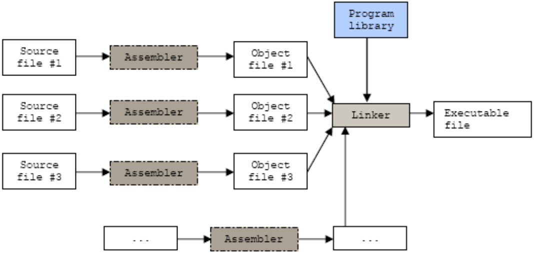


The linker knows where the functions are implemented. For instance, in the *Hello World!* example we need the function *printf* from the standard *C* library. The linker will "bind" the *printf* call with its implementation. It will also bind all function calls from different files of your project.

Compilation

Linker

The linker combines one or more object files generated by the assembler into a single executable file.



The linker's output is in Executable and Linkable Format (*ELF*). This format contains the program in different sections for data and code, which will be loaded/initialized differently by the OS, when you execute the program.

Compilation

Linker: Source Code Organization

We said, the linker can combine the code from multiple files in the project. However, there are some pitfalls with that.

First, the compiler needs to know how a function is declared at the moment you call it. Since the compiler reads the file from top to bottom, you need to declare functions above the line you use them. But you can define (implement) them later or even in another file.

The next slide shows a valid example on how multi-file projects have to be structured: Using header files (`.h`) and source files (`.c`).

Note: Only source files are compiled. The header files are copied into the source files by the preprocessor (replacing the line `#include <header_file.h>`).

Compilation

Linker: Source Code Organization - Example

File foo.h

```c
int foo(int x);  /* Function declaration */
```

File foo.c

```c
#include "foo.h"
int foo(int x) {  /* Function definition */
   return x + 1;
}
```

File main.c

```c
#include <stdio.h>
#include "foo.h"  /* Include declaration of foo */
void main() {
   int x = foo(2);  /* Use the function here */
   printf("%d\n", x);
}
```

Compilation

Linker: benefits

Modularity

- Programs can be written as a collection of smaller source files
- Common functions can be grouped into libraries for later reuse

Efficiency

- If one source file is changed, no need to recompile other files
- The memory footprint can be reduced

Compilation

Linker: static and dynamic linking

Static linking The references to the external functions and variables are resolved at *compile-time*. All functions and variables are copied into a stand-alone executable.

Dynamic linking The references to the external functions and variables are resolved at *run-time*. The shared libraries are loaded into the memory or, if already loaded, mapped to the program. The loading can occur when the program begins (load-time linking) or when the program is running and needs a given library (run-time linking).

Compilation

Linker: static and dynamic linking

Static linking	✓	fast (no dynamic querying)
	✓	no dependencies
	×	large memory footprint (duplication)
	×	minor changes in common functions require each application to be relinked
Dynamic linking	✓	only one copy of shared library in memory
	✓	shared libraries can be recompiled without any need to re-link the applications
	×	usually slower
	×	shared libraries must be installed in the system

Pointers and Arrays

Pointers and Arrays

Creating a pointer

All data is stored in the memory and each piece of data has an address in the memory.
A Pointer is a variable that contains an address as its value.
The type of a pointer describes the type of the data that it points to. Pointers are declared with a '*' before the pointed-to type:

```
type *var_name
```

Consider the following example:

```
int variable = 10;
int *ptr_to_variable = &variable;
```

We create an `int` variable with the value 10 ($\rightarrow$ 10 is now in memory). Then we create a "pointer to an `int`" variable (i.e., `int *ptr_to_variable`) that stores the address of the variable's data (`&variable` gives us the address of the variable's data).

Pointers and Arrays

Accessing pointed-to data

We can access data to which a given pointer points by using the dereferencing operator '*'. Since variable names are synonymes for the data they store, we can work with the dereferenced pointer just as we would with the original variable:

```
int variable = 10;
int *ptr_to_variable = &variable;
*ptr_to_variable = 20; // <- dereferencing (modify)
printf("The variable has the value %d "
       "and the pointer points to %d.\n",
       variable,
       *ptr_to_variable); // <- dereferencing (read)
```

This example will print: `The variable has the value 20 and the pointer points to 20.`

Note: We can still use the variable to access the exact same data.

Pointers and Arrays

Pointers as function arguments (1)

In *C* values are "passed by value" (copied) at a function call. Thus we cannot change the variables that were given as arguments:

```
void my_func(int a) {
    a = 20;
}
void main() {
    int var = 10;
    my_func(var);
    printf("The value is %d.\n", var);
}
```

This example will print: `The value is 10.`

The variable a is local to `my_func` and is copied to its scope (call by value).

Pointers and Arrays

Pointers as function arguments (2)

But we can access the data of a variable if we have a pointer to it:

```
void my_func(int *a) {
   *a = 20; // <- dereferencing (modify)
}
void main() {
   int var = 10;
   my_func(&var);
   printf("This time, the value is %d!\n", var);
}
```

This example will print: `This time, the value is 20!`

The variable a is local to `my_func` and is copied to its scope (call by value), but when dereferenced we can access the data of the variable `var`.
Note: You can try out those examples in VirtualC.

Pointers and Arrays

Pointers as function arguments (3)

We could also just return the value and override the initial variable value with the new value (returned values are also copied):

```
int my_func(int a) {
   return a + 20;
}
void main() {
   int var = 10;
   var = my_func(var);
   printf("The value is %d.\n", var);
}
```

This example will print: `The value is 30.`

But what if we have multiple parameters? We can only return one value.

Pointers and Arrays

Pointers as function arguments (4)

A fitting example is a swap function, that swaps the values of two variables:

```
void swap(int *a, int *b) {
    int a_was = *a; // store away a's original value
    *a = *b;        // read b's value and write it to a
    *b = a_was;     // restore a's original value to b
}
void main() {
    int varA = 10;
    int varB = 20;
    swap(&varA, &varB);
    printf("varA = %d, varB = %d.\n", varA, varB);
}
```

This example will print: `varA = 20, varB = 10.`

Pointers and Arrays

are closely related (1)

An array is a block of memory which stores data of a given type sequentially.

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

a:	1	2	3	4	5
	a[0]	a[1]	a[2]	a[3]	a[4]

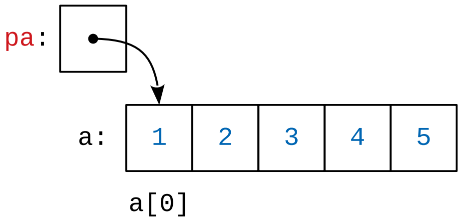
This array contains integer values. The values are residing in memory. Thus we can also access the memory by using pointers.

Pointers and Arrays

are closely related (2)

Accesses to array elements can be used just like variables. This way we can also get the address of an array element:

```
int *pa = &a[0];
```

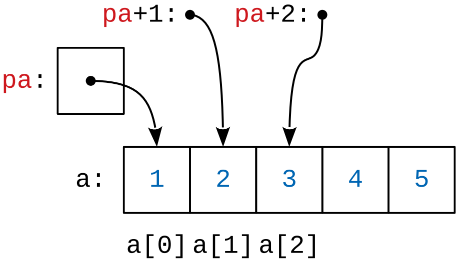


We created a new pointer variable that points to an integer (the array data's type). In this case we point to the first element in the array (index 0).

Pointers and Arrays

are closely related (3)

Addresses are encoded as unsigned integers. The range of which is predetermined by the computer architecture's memory bus width (e.g., 32 bit, 64 bit).
Since arrays store values sequentially and addresses are also a kind of integer, we can point to values relative to the pointer as well:



This is a form of "address arithmetic".

Pointers and Arrays

are closely related (4)

Addresses are in general aligned to bytes. E.g., the address 1000 references the 1000th byte in memory.

An important requirement to the example above is the size of the type our pointer points to. Knowing this, the compiler gives us a simple abstraction:
We used `pa+1`. Since our compiler knows that `pa` is an `int` pointer, the result of the computation will give us the address `pa+1*sizeof(int)`.

Pointers and Arrays

are closely related (5)

Looking back at arrays, our compiler does the exact same when accessing array elements by index. It uses the base address of the array and adds $index * sizeof(type)$ to figure out the elements address. Internally line 3 and 4 are equivalent:

```
int array[5] = { 1, 2, 3, 4, 5 };
int *array_ptr = array;     // same as: &array[0]
int three_a = array[2];
int three_b = *(array_ptr+2);
int three_c = array_ptr[2]; // works as well!
```

Even more: You can also use array indices on a pointer (see `three_c`)! This is how closely related arrays and pointers are.

Pointers and Arrays

Address Arithmetic (1)

Address arithmetic is often used when iterating through an array. This way we can increase a pointer in every execution of a loop.

This example shows a function searching for the letter 'a' in a string.

```
char* contains_a(char string[]) {
    char *ptr = &string[0];
    while( *ptr != '\0' ) { // zero terminated string
        if( *ptr++ == 'a' )  // post-fix increase of ptr
            return "does";
    }
    return "doesn't";
}
```

Note: Since arrays are so closely related to pointers, on a function call only the base address of the array is copied (call by value), not the whole array.

Pointers and Arrays

Address Arithmetic (2)

You can try out this example in VirtualC:

```
void main() {
    char string[] = "This is b test.";
    char *string_contains_a = contains_a(string);
    printf("'%s' %s contain an 'a'.\n",
           string, string_contains_a);
}
```

Pointers and Arrays

Strings - Literals (1)

In C, every string you write with '"..."' is called a "string literal".

```
printf("This is a test.");
//      ^^^^^^^^^^^^^^^^^^ a string literal
```

String literals are immutable character arrays stored in the program's binary.
This can be a problem if you use it to initialize a string variable!

Pointers and Arrays

Strings - Literals (2)

If you declare a string as a character array,

```
char a_string[] = "This is a string.";
```

you are creating an array (with its own memory section) that is initialized with the string literal `"This is a string."` (the characters are copied to the array).

However, if you declare a string as a character pointer,

```
char *b_string = "This is a string.";
```

it is implicitly constant, because the pointed-to string literal is constant (immutable character array). Thus the following crashes!

```
char *b_string = "This is a string.";
b_string[4] = 'x';
```

Prevent this bug by using a pointer to a `const char` instead:

```
const char *b_string = "This is a string.";
```

Pointers and Arrays

Command-line arguments

When we start a program from the command line, we can pass some arguments, e.g.:

```
firefox --new-window www.tum.de
```

The strings "--new-window" and "www.tum.de" are passed to the program as arguments in the main function.

This is a stub for a main function receiving those strings (argv is an array of strings (const char*) given as pointer):

```
int main(int argc, const char** argv) {
    printf("Num args: %d, first arg: %s\n",
           argc, argv[0]);
    return 0;
}
```

In VirtualC you can specify the command line arguments in the GUI when enabling: Debug > Set command line

We covered scopes and pointers. Consider the following example. Can you spot the problem:

```
int *create_array() {
    int new_array[5] = { 1, 2, 3, 4, 5 };
    return new_array;
}
void main() {
    int *array = create_array();
}
```

Pointers and Arrays

Memory: Stack and Heap (2)

```
int *create_array() {
    int new_array[5] = { 1, 2, 3, 4, 5 };
    return new_array;
}
void main() {
    int *array = create_array();
}
```

When talking about scopes, we said that the lifetime of a local variable is bound by the block it is declared in. Here the lifetime of `new_array` ends when the function `create_array` completes. But we return a pointer to that array! When using that pointer it points to invalid memory.

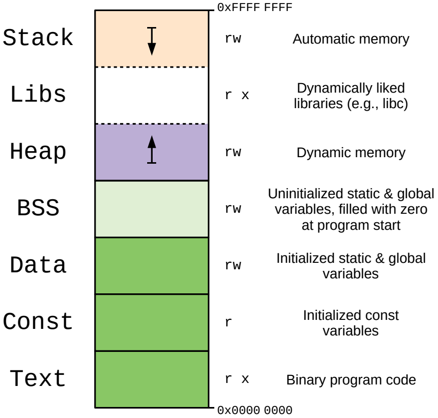
The array was allocated on **stack**. The stack will be released after the function is done executing and another function will use the stack. Accessing invalid memory is one of the most common fatal errors at runtime.

Pointers and Arrays

Memory: Stack and Heap (3)

The **stack** (light orange) is a section of memory that is used at runtime to temporarily store local/automatic variables in a Last-In-First-Out way (like a pile).

The **heap** (purple) is also a memory that can be used at runtime. As opposed to the stack, the data on the heap can be allocated and deallocated by the programmer.

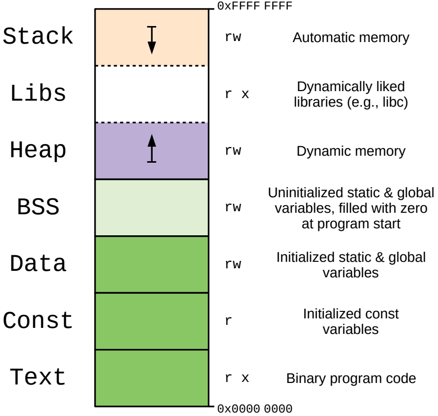


Pointers and Arrays

Memory: Stack and Heap (4)

The stack grows from higher to lower addresses when adding (push) values on it. When removing (pop) a value the stack shrinks again.

The heap is managed by the operating system (OS) and grows from lower to higher addresses. If we want to store something on the heap, we can request a memory section of specific size `malloc(size)` from the OS. It gives us a pointer to the allocated memory.



Structures

Structures

What is a structure?

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.

Examples:

- a payroll record comprising the name, the address, the social security number, and the salary of an employee
- a point in a 2-d space described by the x- and the y-coordinates

.

Structures

Structure declaration

We declare a structure that can be used to represent a point in a 2-d space as follows:

```
struct point {
    float x;
    float y;
} pt1;
struct point pt2, pt3 = { 12.0, 5.0 };
pt1.x = 2.0; pt1.y = 3.0;
pt2.x = 4.0; pt2.y = 5.0;
```

- the keyword ‘struct’ declares a structure
- here ‘point’ is a structure tag that can be used subsequently as a shorthand for the part of the declaration in braces
- the variables ‘x’ and ‘y’ are the members
- ‘pt1’, ‘pt2’, and ‘pt3’ are variables that represent structures of type ‘point’

Structures

Nested structures

We can define a rectangle using two points that are at two diametrically opposite corners

```
struct rect {
    struct point pt1;
    struct point pt2;
} rec1;
rec1.pt1.x = 1.0; rec1.pt1.y = 1.0;
rec1.pt2.x = 5.0; rec1.pt2.y = 4.0;
```

- Note that the ‘rect’ structure contains two ‘point’ structures
- A structure containing another structure, i.e., nested structures
- Also note how the member variables ‘x’ and ‘y’ of the member structure ’pt1’ and ‘pt2’ of the structure ‘rec1’ are denoted

A function can take structures as arguments and can also return a structure.

```
struct point midpoint(struct point pt1,
                      struct point pt2)
{
    struct point mp;
    mp.x = (pt1.x + pt2.x)/2;
    mp.y = (pt1.y + pt2.y)/2;
    return mp;
}
```

The above function returns the midpoint between the points 'pt1' and 'pt2'.

Note: Structures are also "Call-by-Value", i.e., they are copied to the function scope and, when returned, copied to the callers scope.

Structures

Structure pointers

Structure pointers are just like pointers to ordinary variables.

```
struct point pt1 = { 3.0, 4.0 };
struct point *pt2 = &pt1;
printf("pt1 is at (\%f,\%f)\n", (*pt2).x, (*pt2).y);
printf("pt1 is at (\%f,\%f)\n", pt2->x, pt2->y);
```

The output is as follows:

```
pt1 is at (3.0,4.0)
pt1 is at (3.0,4.0)
```

- The parentheses are necessary in '(*pt2).x' because the precedence of the structure member operator '.' is higher then '*'. Thus, if we write '*pt2.x', then this would imply '*(pt2.x)', which does not make sense.
- Otherwise, 'structure pointer -> member variable' is a short hand form used to refer to the member variable, as shown in the last line of the code.

We can define an array of structures similar to an array of ordinary variables.

We can represent a triangle using three vertices stored as an array of 3 structures of type ‘point’. Then, we can find the centroid of the triangle as follows:

```
struct point tr[3] = {{1.0,1.0},{4.0,1.0},{4.0,4.0}};
struct point centroid;
centroid.x = (tr[0].x + tr[1].x + tr[2].x)/3;
centroid.y = (tr[0].x + tr[1].x + tr[2].x)/3;
printf("The centroid is at (\%f,\%f)\n",
       centroid.x, centroid.y);
```

The output is as follows:

```
The centroid is at (3.0,2.0)
```

Structures

Typedef(1)

The keyword ‘typedef’ can be used to create new data type names. Mainly required for:

- easy portability of programs, handles machine-dependent data types
- easy documentation using self-explanatory names for data types

Example 1:

```
typedef char *String;
String str1 = "Hello World", str2;
strcpy(str2, str1);
printf("%s\n", str2);
```

The output is: `Hello World`
In the above example, we define a new variable type, i.e., ‘String’. Here, ‘String’ becomes synonymous to ‘char*’.

Structures

Typedef(2)

Example 2:

```
typedef struct {
    int day;
    char *month:
    int year;
} Date;
Date today = { 7, "May", 2020 };
printf("Today's date is: %s %d, %d",
       today.month, today.day, today.year);
```

The output is:

`Today's date is:  May 7, 2020`

In this example, we define a new data type ‘Date’ that is a structure storing a date using 3 member variables, i.e., ‘day’, ‘month’, and ‘year’.

- Note that using ‘typedef’, we can avoid typing ‘struct’ each time we declare a ‘Date’.

Structures

Bit-fields(1)

To save storage, we can pack several single-bit flags into a single machine word in applications like compiler symbol tables. Externally-imposed data formats, such as interfaces to hardware devices, also often require the ability to access specific bits. Define a set of “masks” corresponding to the relevant bit positions:

```
#define BIT0 01
#define BIT1 02
#define BIT2 04
```

or `enum {BIT0 = 01, BIT1 = 02, BIT2 = 04};`

We can make use of the masks as follows:

```
flags |= BIT0 | BIT1; // bit 0 & bit 1 are set to 1
flags &= ~(BIT1 | BIT2); // bit 1 & bit 2 are set to 0
if ((flags & (BIT0 | BIT2))==0){...} // if both bit 0
    & bit 2 are 0
```

Structures

Bit-fields(2)

Alternatively, C offers the option of defining and accessing fields within a word directly. A bit-field is a set of adjacent bits within a single implementation-defined storage unit.

```
struct {
    unsigned int flag0 : 1;
    unsigned int flag1 : 1;
    unsigned int flag2 : 1;
} flags;
```

This defines a variable table called 'flags' that contains three 1-bit bit-fields. The number following the colon represents the field width in bits. The fields are declared unsigned int to ensure that they are unsigned quantities.

Bit-fields are referenced similar to other structure members:

```
flags.flag0 = flags.flag1 = 1; //'flag0' and 'flag1' are set to 1
```

File Input/Output

File Input / Output

File pointer

A file pointer, points to a structure that contains information about the file. Some are: the location of a content buffer, the current character position in the buffer, file access permissions (read/write), and whether errors or special conditions have occurred.

The library `<stdio.h>` includes a structure declaration called `FILE`.

We can declare a file pointer as follows:

```
File *fp;
```

This says that 'fp' is a pointer to a `FILE`.

File Input / Output

Opening a file

The library function `fopen` is used to open a file such that it can be read or written into. This functionality is provided by the OS.

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

Example on how to invoke `fopen`:

```
FILE *fp = fopen("hello.txt", "w");
```

'`fp`' is the file pointer. The mode can be one of the following:

Mode	Description
"r"	reading the file from the beginning; returns an error if the file does not exist
"w"	writing to the file from the beginning; creates the file if it does not exist
"a"	appending to the end of the file; creates the file if it does not exist

File Input / Output

Writing to a file (1)

The library function putc is used to write a character to the file opened in write/append mode. As all the library functions it does its operation at the current character position (stored in the FILE structure). When finished it advances the current character position by 1.

```
int putc(int c, FILE *fp)
```

putc writes the character denoted by 'c' to the file pointed to by 'fp' and returns the character written, or EOF if an error occured.

Example:

```
putc('A', fp); // writes the character 'A' at the
    current position in the file pointed to by 'fp'
```

File Input / Output

Writing to a file (2)

The library function `fputs` is used to write a string to the file opened in write/append mode. It returns `EOF` if an error occured.

```
int fputs(char *line, FILE *fp);
```

Example:

```
char line[] = "Hello!\n";
fputs(line, fp);
// writes "Hello!\n" to the file
```

File Input / Output

Writing to a file (3)

The library function `fprintf` is used to write formatted output to the file opened in write/append mode. It is used similarly to `printf`, just with the file pointer as its first argument. It returnes the number of characters written to the file.

```
int fprintf(FILE *fp, const char* format, ...);
```

Example:

```
fprintf(fp, "Hello %d %s!", 2, "you");
// writes "Hello 2 you!" to the file
```

File Input / Output

Reading from a file (1)

The library function getc is used to get the next character from the file opened in read mode. Thereafter, it also increases the current character position by 1 in the file buffer.

```
int getc(FILE *fp)
```

getc returns the next character from the file referred to by the file pointer fp. It returns EOF for end of file or error. Example:

```
while(c = getc(fp) != EOF)
    printf("%c",c);
```

If the file contains the text: Hello World
The above code will give as output:
`Hello World`

File Input / Output

Reading from a file (2)

The library function `fgets` is used to read a line from the file opened in read mode. It copies characters from the file opened file to a given buffer (character array) until it reads `'\n'` or the file ends. It also will read only a maximum of `n` (given) characters. It returns zero if an error occured.

```
char *fgets(char *str, int n, FILE *fp);
```

Example:

```
char buffer[10];
int success = fgets(buffer, 10, fp);
printf("Read: %s!\n", buffer);
// assume the file contains "Hello.\n"
// this example prints: Read: Hello.
// !
```

It also advances the character position. Note that the read string contains the newline character.

File Input / Output

Reading from a file (3)

The library function `fscanf` is used to read formatted text from the file opened in read mode. It returns the number of correctly matched input values. The value EOF is returned if the end of input is reached before either the first successful conversion or a matching failure occurs.

```
int fscanf(FILE *fp, const char* format, ...);
```

Example:

```
int a, b;
fscanf(fp, "%d, %d", &a, &b);
printf("Read %d and %d!\n", a, b);
// assume the file contains "52, 48"
// this example prints: Read 52 and 48!
```

It also advances the character position. This way we can continuously read formatted data from a file.

File Input / Output

Closing a file

The library function `fclose` is used to close a file.

```
int fclose(FILE *fp)
```

It can be invoked as follows:

```
fclose(fp)
```

This terminates the connection between the file pointer and the external name that was established by `fopen`, freeing the file pointer. It flushes the buffer in which `putc` is collecting output. Remember to always close the files you opened when you don't want to access it anymore.

File Input / Output

Example

```
void write_example(void) {
  int x = 52, y = 48;
  FILE *fp = fopen("hello.txt", "w");
  fputs("Hello.\n", fp);
  fprintf(fp, "%d, %d\n", x, y);
  fclose(fp);
}
void read_example(void) {
  fp = fopen("hello.txt", "r");
  char buffer[10];
  fgets(buffer, 10, fp);
  printf("Line 1: %s\n", buffer);
  int a, b;
  fscanf(fp, "%d, %d\n", &a, &b);
  printf("Line 2: a=%d b=%d\n", a, b);
  fclose(fp);
}
```

File content:

```
Hello.\n
52, 48\n
```

File Input / Output

Working with strings (1)

There are some helpful string operations to keep in mind. You can find usage information on the internet.

`strcat(s,t)`	concatenate t to end of s
`strncat(s,t,n)`	concatenate n characters of t to end of s
`strcmp(s,t)`	return negative, zero, or positive for s < t, s == t, s > t
`strncmp(s,t,n)`	same as strcmp but only in first n characters
`strcpy(s,t)`	copy t to s
`strncpy(s,t,n)`	copy at most n characters of t to s
`strlen(s)`	return length of s
`strchr(s,c)`	return pointer to first c in s, or 0 if not present
`strrchr(s,c)`	return pointer to last c in s, or 0 if not present

File Input / Output

Working with strings (2)

There are two additional functions whose purpose you can propably guess from the file related functions we had before:

`int sprintf(char *str, const char *format, ...)`
Just like `fprintf` it writes formatted values (this time) to a string. The only difference is, that the string pointer `const char *s` is not increased. But function returns the number of written bytes.

`int sscanf(const char *s, const char *format, ...)`
Like `fscanf` it parses values from a string. This function also does not increase the string pointer `const char *s`. The return value specifies the number of matched values.

Assignment 2 (due: 11.11.2021)

Assignment 2

Task 1: Run Length Encoding (1)

Implement run-length encoding and decoding.

Run-length encoding (RLE) is a simple form of data compression, where runs (consecutive data elements) are replaced by just one data value and count.

For example we can represent the 53 original characters:
`WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWWWWB`
with only 13 characters:
`12WB12W3B24WB`

You can assume that the original string never contains any numbers. The library `<ctype.h>` specifies the following helpful function: `isdigit(character)`
Print the encoded and the decoded string to the terminal (printf).

Assignment 2

Task 1: Run Length Encoding (2)

You can use the following code structure as a start:

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

void encode(const char *original, char *encoded)
{
    //TODO
}

void decode(const char *encoded, char *decoded)
{
    //TODO
}

void main(void)
{
    const char original[100] = "
        WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWWWWB";
    char encoded[100];
    char decoded[100];
    printf("Original: %s\n", original);
    encode(original, encoded);
    printf("Encoded:  %s\n", encoded);
    decode(encoded, decoded);
    printf("Decoded:  %s\n", decoded);
}
```

Assignment 2

Task 2: Grading (1)

Implement a program that reads a file containing grading information, generate the final grade and write the new information to a file. Also output the information for the person with the highest score to the terminal.

The input file contains the following information per line separated by a comma:

- First name
- Last name
- Points assignment 1
- Points assignment 2
- Points assignment 3
- Points assignment 4
- Email address

Assignment 2

Task 2: Grading (2)

Two lines of an example input file:

```
Peter, Pasta, 15, 23, 15, 22, peter.pasta@tum.de
Lisa, Lasagna, 22, 25, 0, 12, lisa.lasagna@tum.de
```

Two lines of the resulting output file:

```
Peter Pasta <peter.pasta@tum.de>: 2.7
Lisa Lasagna <lisa.lasagna@tum.de>: 3.7
```

Terminal output: `Peter Pasta <peter.pasta@tum.de>: 2.7`

Use the following grading scheme (max points: 100)

Points	0.0	50.0	55.5	60.5	65.5	70.5	75.5	81.0	86.0	91.0	96.0
Grade	5.0	4.0	3.7	3.3	3.0	2.7	2.3	2.0	1.7	1.3	1.0

Hint: Use structures to save the values you read from the file. You can assume to never have more than 100 lines in the input file.

Assignment 2

Task 2: Grading (3) - Note on comma-separated input with fscanf / sscanf / scanf

Note: Consider the following code to reads two comma separated *strings*:

```
fscanf(fp, "%s,%s\n", string_1, string_2);
```

The C standard says: `%s` "matches a sequence of non-white-space characters." and a comma is not a whitespace.

Virtual-C does not implement scansets, so you have to use a workaround (`,,` is not matched in `%s` and accounts for a single `,`):

```
fscanf(fp,"%s,,%s\n",string_1,string_2);
```

If you use *gcc* or another standard compliant compiler, you can use scansets:

```
fscanf(fp,"%[^,],%s\n",string_1,string_2);
```

where `[^,]` matches all characters that are not (`^`) a comma. The above does: read all non-comma characters into `string_1`, then skip a comma, then read all non-whitespace characters into `string_2`, then skip a newline (so the next invocation starts in the next line).

Next Lab: Lab 3 on 11.11.2021

Next week: Q&A Meetings!
Please write an email describing the issues you'd like to discuss.
If you have questions, we meet on *Zoom*.

Next Lab: Lab 3 on 11.11.2021

- Topic: Digital Filters
- **Review meeting: Assignment 2**
- Hand out Assignment 3