Concepts and Software Design for CPS

Lab 1: Introduction to C (Part 1)

Raphael Trumpp Binqi Sun

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

Lab 1 (Part 1): Overview

Variables, Types and Operators

Control Flow

Functions

Assignment 1 (due: 28.10.2021)

Literature

Lab 1 loosely follows the book *The C Programming Language*:

- Chapter 1
- Chapter 2
- Chapter 3: omitted 3.6, 3.8
- Chapter 4: 4.1-4.3

Brian W. Kernighan and Dennis M. Ritchie.

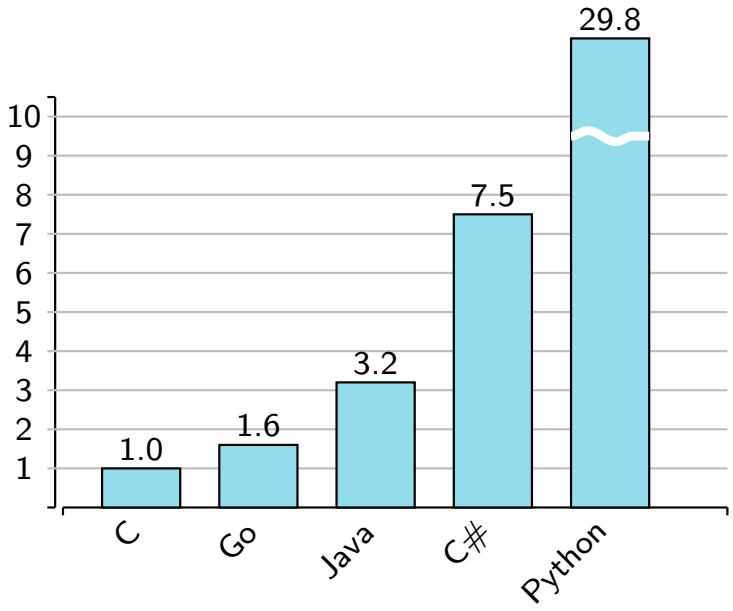
The C Programming Language.

Prentice Hall Professional Technical Reference, 2nd edition, 1988.

`Link to the book`

Why learning C?

Fast



C has very little software overhead. Support for many features in modern programming languages (e.g., garbage collection, dynamic typing) can degrade the performance of the application.

Lingua Franca

Modern programming languages share common fundamental concepts with C. Learning C will help you to quickly understand other programming languages.

Popular

Apr 2019	Change	Programming Language
1		Java
2		C
4	^	Python
3	v	C++
6	^	C#

According to the TIOBE Index, C is the second most popular programming language, behind Java.

Why learning C?

Embedded Programming

C is the dominant language in embedded systems programming. Embedded processors are usually limited in computational power and I/O capabilities, and have restrictions on power usage. It is comparatively easier to write a mission-critical real-time application (e.g., anti-lock brakes) in C, as it provides a low level memory access and a straightforward conversion to processor instructions. C programs are faster and more reliable.

System-Level Programming

Microsoft Windows, OS X, Linux and Android kernels are mostly written in C. Also, the world's most popular databases (e.g., Oracle Database, MySQL) are coded in C. The firmware of many "Internet of Things" devices and autonomous cars are written in C to meet real-time requirements and provide high performance.

Hello World!

A C program consists of:

- *functions* - statement with computing operations to be done
- *variables* - values used during the computation

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

`#include<stdio.h>` tells the compiler to include the standard input/output library which defines the function *printf*

printf(args) displays its arguments *args* (e.g., "hello, world\n")

`'\n'` is the newline character

printf can use placeholders to print variable values of different types (e.g., %d - `int`, %f - `float/double`).

Variables, Types and Operators

Variables

Basics

Variables and constants are the basics objects in a program.

Variable declaration should have:

- type
- name
- (optionally) initial value

```
#include<stdio.h>
int a;
int main() {
    float b;
    printf("a=%d, b=%f\n", a, b);
}
// When executed prints:
// a=0, b=6381.423785902
```

Variables can be defined inside of a function (=local/automatic) or outside (=global).

Variables

Initialization and Scope

An important difference lies in the **implicit** initialization of variables if no initial value is given: *Global* variables are initialized to zero. *Local* variables are **not** initialized and can contain random values!

In the previous example

- `a` is a global variable and will be initialized to zero and
- `b` is not initialized, which is why it can hold a random value.[1]

The scope (lifetime) of a *local* variable is bound by the block $\{\ldots\}$ it is defined in. I.e., the moment the execution leaves a block, all variables defined in the block are invalidated!
A *global* variable can be accessed from anywhere in the program. Its scope is bound by the start end end of the program itself.

[1] This is compiler & OS dependent.

Data Types

The basic data types in C:

- char a single byte, capable of holding one ASCII character (e.g., 'a')
- int a typically 2 or 4 bytes integer (e.g., 42)
- float and double a single- and double-precision floating point, typically 4 for float and 8 bytes for double (e.g., 4.99)

```
#include <stdio.h>
main() {
    printf("int size: %ld\n", sizeof(int));
    printf("double size: %ld\n", sizeof(double));
}
```

Basic Operations

The basic arithmetic operators are +, -,*,/, and % (remainder). They are defined for all basic data types.

```
#include <stdio.h>
main() {
    int a = 2, b = 4, c;

    c = a + b;
    printf("c=%d\n",c);
    c = b / a;
    printf("c=%d\n",c);
}
```

Type Conversion

Promotion

Values of a specific type can be converted to a different type. This happens either automatically (also called promotion) or manually (also called (type) casting).

```
char a = 97;
int b = a + 1; // a is promoted to type int
char c = (char)(b + 1); // (b+1) is casted to char
printf("%d %d %d\n", a, b, c);
```

Automatic conversion happens when the resulting type can hold all values of the smaller type in assignments and operations. Here are the basic rules (for double, float, int and char):

- If either operand is double, convert the other to double.
- Otherwise, if either operand is float, convert the other to float.
- Otherwise convert char to int.

Type Conversion

Casting

Type casting is used in case a value should be assigned to a smaller type, e.g., assigning an `int` to a `char` or a `float` to an `int`.

Note that casting can be lossy and can change the value. E.g., the `char` type can hold only values from -128 to +127, so this example results in a negative number:

```
char a = 127;
int b = a + 1; // a is promoted to type int
char cb = (char)b; // (b) is casted to char
printf("%d %d %d\n", a, b, cb);
// prints: 127 128 -128
```

When casting floating point values (float/double) to an integer type (int/char), fractional digits are always cropped. Extremely high or low values can also be distorted.

Tipp: Make sure the value you are casting is in the expected range before casting to avoid unexpected results.

Arrays

Basics

An *array* is a data structure holding values of the same type in a given order. Each value can be addressed by its index. In *C*, array indexes always start at 0.

```
int a[4];
```

In the above definition, *a* is an array of type `int` and size 4: a block of 4 consecutive `int` objects a[0], a[1], a[2] and a[3].

```
#include <stdio.h>
main() {
    int a[4] = {7, 22, 4, 17}
    printf("%d %d %d %d\n", a[0], a[1], a[2], a[3]);
    a[2] = 0;
    printf("%d %d %d %d\n", a[0], a[1], a[2], a[3]);
}
// When executed prints:
// 7 22 4 17
// 7 22 0 17
```

Arrays

Strings

A *string* is an array of characters. The special character '\0' marks its end (also called: zero terminated string).

index	0	1	2	3	4	5
value	H	e	l	l	o	\0

```
#include <stdio.h>

int main() {
    char str[6] = "Hello";
    printf("%s you!\n",str);
}
// When executed prints:
// Hello you!
```

printf uses the placeholder %s to print a string.

Increment/Decrement Operators

Use ++ and - - for incrementing and decrementing by 1.

```
#include <stdio.h>

main() {
    int a = 2;

    a++;
    printf("a=%d, ",a);
    printf("a=%d, ",a++);
    printf("a=%d, ",a);
    printf("a=%d\n",++a);
}
// When executed prints:
// 3, 3, 4, 5
```

- ++a increments *a* before its value is used
- a++ increments *a* after its value has been used

Constants

Constants are evaluated during compilation rather than at runtime and cannot change the value while variables can.

```
#include <stdio.h>
#define PI 3.14159

main() {
    int radius = 20;

    /* calculate area of a circle */
    double area = PI * radius * radius;

    printf("Area of the circle is %8.2f\n",area);
}
// When executed prints:
// Area of the circle is  1256.64
```

Tip: Avoid unnamed values ("magic numbers") in your code. It will be more *readable* and easier to *modify*.

Bitwise Operators

Decimal-binary-hexadecimal conversions

In numeral systems, the *radix* (or *base*) is the number of digits needed to represent numbers.

- decimal system radix is 10
- binary system radix is 2
- hexadecimal system radix is 16

In the decimal system, we use the 10 digits from 0 through 9. When a number "hits" 9, the next number will not be another different symbol, but a "1" followed by a "0".

In the binary system, the radix is 2, we use only two digits: 0 and 1. When counting, after a digit hits "1", instead of "2" it jumps straight to "10", followed by "11" and "100".

Bitwise Operators

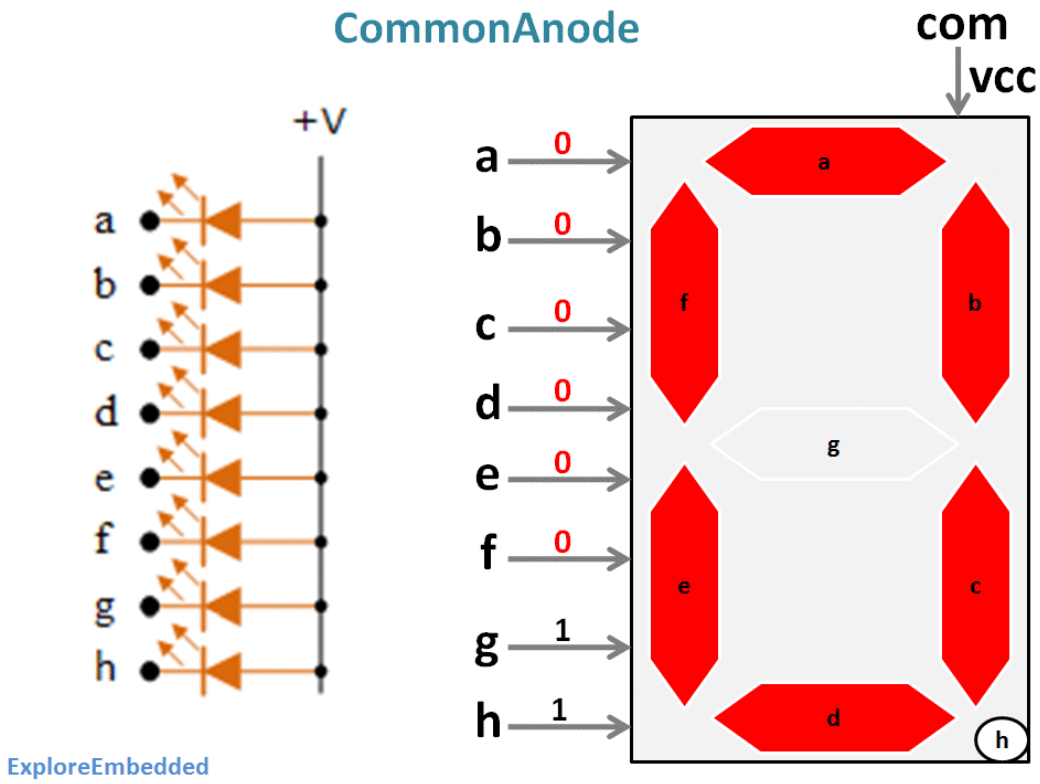
Decimal-binary-hexadecimal conversions

Decimal	Binary	Hexadecimal
0	00000000	0
1	00000001	1
2	00000010	2
3	00000011	3
4	00000100	4
5	00000101	5
6	00000110	6
7	00000111	7
8	00001000	8
9	00001001	9
10	00001010	A
11	00001011	B
12	00001100	C
13	00001101	D
14	00001110	E
15	00001111	F

Bitwise Operators

Why you may need it?

Microcontrollers use I/O pins to interact with the outside world. When several pins are combined together, they form an I/O port. The I/O port can be connected to LEDs, different sensors or a bunch of motors. You write to I/O ports to send the output signals and read I/O ports to get the input from external devices.

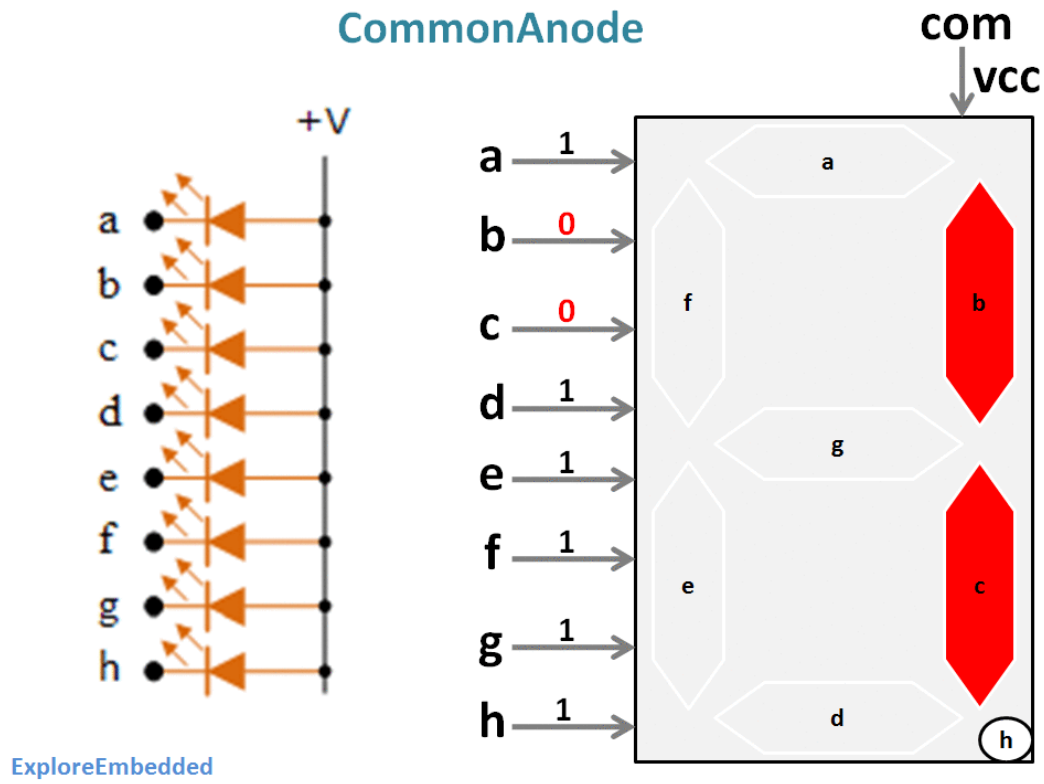


https://exploreembedded.com/wiki/Interfacing_Seven_Segments_with_PIC16f877a

Bitwise Operators

Why you may need it?

Microcontrollers use I/O pins to interact with the outside world. When several pins are combined together, they form an I/O port. The I/O port can be connected to LEDs, different sensors or a bunch of motors. You write to I/O ports to send the output signals and read I/O ports to get the input from external devices.

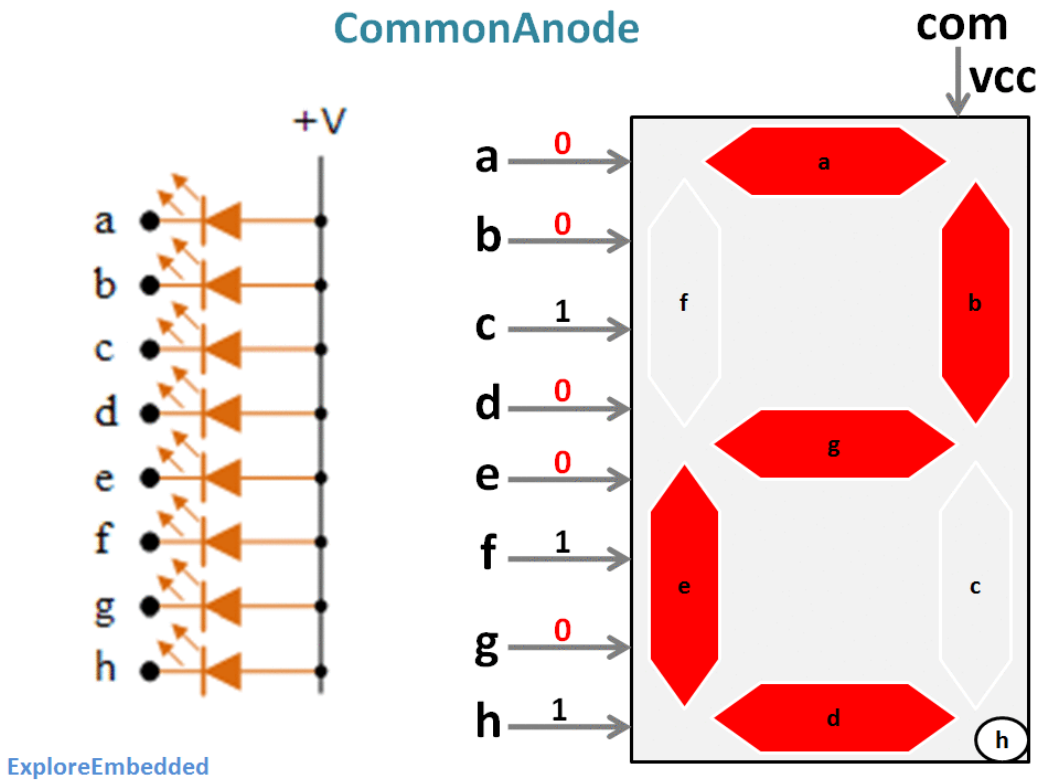


https://exploreembedded.com/wiki/Interfacing_Seven_Segments_with_PIC16f877a

Bitwise Operators

Why you may need it?

Microcontrollers use I/O pins to interact with the outside world. When several pins are combined together, they form an I/O port. The I/O port can be connected to LEDs, different sensors or a bunch of motors. You write to I/O ports to send the output signals and read I/O ports to get the input from external devices.

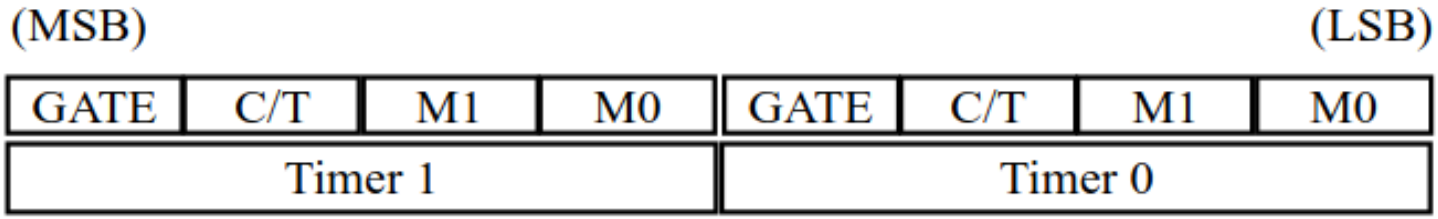


https://exploreembedded.com/wiki/Interfacing_Seven_Segments_with_PIC16f877a

Bitwise Operators

Why you may need it?

Microcontrollers have several system registers. Reading or writing to these registers is used to enable or disable certain functionalities. Each bit represents a different functionality. You must be able to set and clear particular bits within the system registers.



M1	M0	Mode
0	0	13-bit timer
0	1	16-bit timer
1	0	8-bit autoreload timer
1	1	split timer

Clearing bit 0 and setting bit 1 in the register TMOD of a 8051 microcontroller will configure its timer in 16-bit mode.

Bitwise Operators

AND, OR, XOR, LSL, LSR, NOT

There are six operators for bit manipulation in *C*:

- `&` bitwise AND
- `|` bitwise OR
- `^` bitwise exclusive OR
- `<<` left shift
- `>>` right shift
- `~` bitwise invert (one's complement)

Using hex values in C programs:

```
main()
{
    int a = 0x5555;
    printf("0x%08x\n", a);
}
```

printf needs %x for hexadecimal format and "08" to print 8 digits

Bitwise Operators

Bitwise AND

The bitwise AND operator & is often used to clear a set of bits

```
main()
{
    int a = 0x00005555, b = 0x000000FF;
    printf("0x%08x & 0x%08x = 0x%08x\n", a, b, a&b);
}
// When executed prints:
// 0x00005555 & 0x000000FF = 0x00000055
```

Bitwise Operators

Bitwise OR

The bitwise OR operator | is used to set some set of bits

```
main()
{
    int a = 0x00005555, b = 0x000000FF;
    printf("0x%08x | 0x%08x = 0x%08x\n", a, b, a|b);
}
// When executed prints:
// 0x00005555 | 0x000000FF = 0x000055FF
```

Bitwise Operators

Bitwise NOT

The unary operator $\sim$ converts each:

- 1-bit into a 0-bit
- 0-bit into a 1-bit

```
main()
{
    int a = 0x00005555;
    printf("~0x%08x = 0x%08x\n", a, ~a);
}
// When executed prints:
// ~0x00005555 = 0xFFFFAAAA
```

Bitwise Operators

LSL and LSR

The shift operators << and >> perform left and right shifts. The vacated bits are set to zero.

```
main()
{
    int a = 0x00005555;
    printf("0x%08x << 4 = 0x%08x\n", a, a<<4);
    printf("0x%08x >> 4 = 0x%08x\n", a, a>>4);
}
// When executed prints:
// 0x00005555 << 4 = 0x00055550
// 0x00005555 >> 4 = 0x00000555
```

Tip: Shifting to the left by n positions is equivalent to multiplication by 2^n. Shifting to the right by n positions is equivalent to division by 2^n.

Assignment Operators

Let *op* be an operator from the arithmetic operators $\{+, -, *, /, \%\}$ or the bitwise operators $\{<<, >>, \&, |\}$ and *var*1 and *var*2 two different variables.

$$\text{var1 op= var2 is equivalent to var1 = var1 op var2}$$

For example:

```
#include <stdio.h>
main() {
    int a = 0;

    a+=2;
    printf("a=%d\n",a);
}
```

In the above example a+=2 is equivalent to a=a+2.

Control Flow

Control Flow

Statements and Blocks

A line terminated by a semicolon ';' is called a statement:

```
printf("Hello World!\n"); // is a statement
```

Multiple statements can be grouped together to a compound statement (also called block), by using the braces { and }.

```
{ // <-- start of a block
    printf("Hello ");
    printf("World");
    printf("!\n");
} // <-- end of a block
```

Blocks are of use in Control Flow statements, introduced later.

Control Flow

Logical Expressions

To change the flow of a program execution we want to create conditional branches. Conditions are specified by *logical expressions*. There are 6 logical operators in C: `>`, `>=`, `<`, `<=`, `==`, `!=`. The first 4 are intuitive, the last are: equal, not equal. Logical operators are written between two values ("infix"):

```
height >= width
```

Logical operators can be connected using `&&`, `||`, respectively: logical and, logical or. E.g.:

```
height >= width && size != original_size
```

Tip: Operator precedence is defined (see slide 26), but for readability its better to use brackets `(...)` around expressions.
Tip2: The values `0` is considered `false`, every other value is `true`.

Control Flow

Precedence and Order of Evaluation

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

Precedence (top to bottom) and Associativity of Operators

Control Flow

If-Else Statement

The general structure of If-Else is as follows:

```
if ( expression1 )
    statement1;
else
    statement2;
```

Only if `expression1` is true (non-zero) `statement1` is executed.
Otherwise `statement2` is executed.
The `else` part is optional.
Blocks allow executing multiple statements in the `if` or `else` part.
If-Else is also a statements and therefore enables nesting:

```
if ( expression1 )
    if ( expression2 )
        statement;
```

Control Flow

Else-If

An optional addition to If-Else is one or multiple Else-If:

```
if ( expression1 )
    statement1;
else if ( expression2 )
    statement2;
else if ( expression3 )
    statement3;
else
    statement4;
```

```
// This is equivalent
if ( expression1 )
    statement1;
else
{
    if ( expression2 )
        statement2;
    else
    {
        if ( expression3 )
            statement3;
        else
            statement4;
    }
}
```

Control Flow

Switch Statement

Sometimes we want to execute statements based on one variable's state. For that, using a Switch statement is preferred over an If-Else-If-Else-If-... Statement. There can be an arbitrary amount of cases.

```
switch ( expression ) {
    case const-expr1: statements1
    case const-expr2: statements2
    default: statements3
}
```

Const expressions have to be evaluatable to a constant value at compile time (like: constant numbers or characters, *e.g.*, 42, 'a'; but **not** a variable)!
If `expression` matches none of the const expressions, the `default` case is executed (`statements3`).

Control Flow

Switch Statement (cont.)

Cases are "fall-through", *i.e.*, when a case is matched all statements until the end of the switch block are executed! This can be prevented with the statement `break;`, e.g.:

```
switch ( char_variable ) {
    case 'a': // falling through to match both cases
    case 'A':
        printf("It's the alphabet's first letter.");
        break;
    default:
        printf("It's some other letter.");
}
```

You should use this "fall-through" feature with care! Always comment why you use a "fall-through".

Control Flow

While-Loop

While a While-Loops `expression` is true (non-zero) it executes its `statement` repetitively. Before the first execution of the `statement` the `expression` is checked.

```
while (expression)
    statement;
```

Control Flow

For-Loop

A For-Loop works similarly to a While-Loop but has two additional parts: The initialization expression (`expr1`) and the iteration expression (`expr3`).

```
for (expr1; expr2; expr3)
    statement;
```

```
// This is equivalent
expr1;
while (expr2) {
    statement;
    expr3;
}
```

There is one intuitive exception to the plain equivalence, which is in the statement `continue;` (see next slide).

Control Flow

Break and Continue

The execution of loops can be influenced by

- `break`, which jumps out of the loop when executed,
- `continue`, which jumps to the next iteration of the loop. (In case of `for`, `continue` still executes `expr3`.)

E.g.:

```
int i;
for (i=0; i<10; i=i+1) {
    if (i == 5)
        break; // <--
    printf("%d,", i);
}
// When executed prints:
// 0,1,2,3,4,
```

```
int i;
for (i=0; i<10; i=i+1) {
    if (i == 5)
        continue; // <--
    printf("%d,", i);
}
// When executed prints:
// 0,1,2,3,4,6,7,8,9,
// Note: 5 is not present
```

Functions

Functions

Basics

Function definitions have the form:

```
return-type function-name(argument declarations)
{
    declarations and statements
}
```

The *return-type* is the type of the returned value (with a `return` statement). If no value is supposed to be returned the *return-type* is `void`.
Argument declarations are a comma separated list of variable declarations that have to be supplied to the function, where the function is called (used).

Functions

Arguments - Call by Value

C always uses the Call-By-Value convention. I.e., each variable that is passed to a function as an argument is "copied". Thus changes to an argument's value do not persist to the caller:

```
void print_next_number(int number)
{
    number = number + 1;
    printf("The next number is: %d\n"), number);
}
void main()
{
    int number = 5;
    print_next_number(number);
    printf("The current number is: %d\n"), number);
}
```

This example will print:

```
The next number is: 6
The current number is: 5
```

Functions

Arguments - Call by Value (cont.)

As always there are exceptions to the rule. In C references **to** a variable (arrays and pointers (next lab)) are also copied, but only the reference, **not** the data that the reference refers to. The referred to data can thus be changed!

```
void add_one_to_first(char string[])
{ // this changes the referred to data:
    string[0] = string[0] + 1;
}
void main()
{
    char name[] = "Peter";
    printf("My name is %s\n", name);
    add_one_to_first(name);
    printf("My name is %s\n", name);
}
```

This example will print:

```
My name is Peter
My name is Qeter
```

Functions

Arguments - Call by Value (cont.)

Not copying referred data allows us to return more than one value from a function! Consider the following function:

```
void get_first_3_primes(int result[])
{
    result[0] = 2;
    result[1] = 3;
    result[2] = 5;
}
```

We can call the function like this:

```
int p[3];
get_first_3_primes(p);
printf("First 3 primes: %d, %d, %d", p[0],p[1],p[2]);
```

This example will print: `First 3 primes: 2, 3, 5`

Assignment 1 (due: 28.10.2021)

Assignment 1

Task 1: Count Bit Occurrences

In some cases it is interesting to know how many of the bits in a number are 1. E.g., assume you are reading multiple switches (each bit represents the state of one switch) on an I/O port and you want to know how many are switched on. Additionally it should be possible to specify a mask of switches (bits) that should be considered.

For example: We read 200 (`0b11001000`) from the input and want to know how many of the 4 "lowest" switches are on (Mask = `0x0F`). The result would be: 1

Task 1: Write a function to determine how many one bits are present in a given number (of type `unsigned char`) and with a given bit mask (of type `unsigned char`).

Next Meetings

28.10.2021 *Review*

28.10.2021 Deadline for Assignment 1
(submission via **Moodle**)

28.10.2021 Review Meeting for Lab 1 - attendance is mandatory

- Presentation of Assignment 1 (Task 1)
- Slides: Introduction to C (Part 2)
  - Program Structure and Compilation
  - Pointers and Arrays
  - Structures
  - Input and Output
  - The UNIX System Interface
- Hand out of Assignment 2