https://folk.ntnu.no/michaeng/tdt4205_21/
https://folk.ntnu.no/michaeng/tdt4186_21/
michael.engel@ntnu.no

Solutions for Theoretical exercises
Spring 2021

Solutions for Theoretical Exercises 1

Introduction to C programming

Please submit solutions on Blackboard by Friday, 29.01.2021 14:00h

1.1 Parameter passing

Consider the following C program:

```
#include <stdio.h>

int a = 23;
void increment_with_value (int a, int b) {
  a += b;
}

int main(void) {
  increment_with_value(a, 1);
  return a;
}
```

Without compiling and running the program, indicate which value is returned by the `main` function?
Briefly explain your answer.

This is a typical piece of code demonstrating the shadowing (overlaying) of variable names in C.
We have a global variable `a` as well as a local variable `a` declared inside of the `increment_with_value` function. This results in `main` seeing the value of the global variable (23) when it calls `increment_with_value`.
Since `increment_with_value` has its own local variable `a` (which is located on the stack), it increments its value instead of the global variable. After `increment_with_value` returns, its stack frame (containing its local variable `a`) is invalidated. Since the function does not return a value, it is simply discarded (an optimizing C compiler would probably even remove the statement `a += b;` since its results are not used).
Since C uses call-by-value semantics (the value of a parameter variable is copied to the called function), the value of `a` does not change in `main`. Accordingly, its value in `main` remains 23, which is the return value of the `main` function.
(Note that the explanations here go into much more detail than what is required for your answer.)

1.2 Symbols

If we compile the program shown above using `gcc -std=c11 -Wall -o test test.c` and execute `nm test` afterwards, the `nm` output does not contain a memory address for variable `b`.
Briefly explain why `b` is not listed.
The `nm` Unix command only gives the values for statically allocated variables, i.e. global initialized (data segment) and uninitialized (bss segment) variables, variables declared `static` inside of functions, and text segment symbols such as functions.
The variable `b` is a parameter to the function `increment_with_value`, thus it is a local variable which is located on the stack. The location of this variable is relative to the current stack pointer and depends on, e.g. the stack location in memory and the call depth if `increment_with_value` was called recursively. Accordingly, the static analysis of the executable ELF file which is performed by `nm` is unable to retrieve `b`'s address.

1.3 C arrays

Consider the following C program:

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

int main(void) {
  int foo = 0;
  char s[12];
  char *t = "01234567890123";

  printf("foo %p\n  s %p\n", &foo, s);
  strcpy(s, t);
  printf("foo = %d\n", foo);
}
```

a. *Without compiling and running the program*, give the value printed for `foo`.

b. Describe briefly the problem that shows up in the given code which results in this output.

(Solution for a and b)

The answer to this question depends on the protective measures your C compiler employs (that's why I wanted you to think about it...).

A traditional C compiler provides no memory protection. The problem in our example is that the string `s` is declared as an array of char with 12 elements, thus it uses 12 bytes on the stack. The other local variable `foo` is located *after* `s` on the stack. Where exactly depends on your architecture. Usually, a variable of length n bytes has to be stored at an address that is divisible by n. For an `int` variable such as `foo`, this usually means a 4-byte alignment. Thus, one would expect that the first byte of `foo` is the byte directly following the last byte of `s`.

The program uses the `strcpy` function to copy the contents of the string `t` to the memory addresses starting as the first byte of `s`. Note that the string has 14 characters plus the terminating zero byte. Thus, the last bytes of `t` overwrite the memory in which `foo` is stored on the stack. These last bytes are the ASCII characters for the digits 2 and 3 and the terminating zero byte.

So the bytes making up `foo` are overwritten by the following bytes (given in decimal here):

50 (digit 2), 51 (digit 3), 0

What is the value of the remaining byte of `foo`? At the start of `main`, foo was initialized to zero using `int foo = 0;`. Thus, at the start of `main`, all four bytes of `foo` are zero. Afterwards, the first three bytes get overwritten. So, after calling `strcpy`, the values of the four bytes of `foo` are:

50 (digit 2), 51 (digit 3), 0, 0

What is the *integer* value of the variable afterwards? This depends on the *byte order* of your processor! Most current processors use *little endian* byte order. This means that the least significant byte (containing bits 0–7 of the 32 bit integer value) is stored in the first byte (lowest address), with the remaining bytes following in order.

Accordingly, bits 0–7 of `foo` have the value 50, bits 8–15 the value 51 and bits 16–32 are all zero. Accordingly, the value of the int variable `foo` is:

$50 + 256 \cdot 51 = 13106$

c. Modern C compilers protect against the problems shown in this example. For `gcc` or `clang`, find out which command line option can be used to enable this protection.

If you try to compile and run the program on a current system/compiler, it will probably crash with a segmentation fault or similar. The effect shown in the example is a typical `buffer overflow`, which even today is one of the most common causes for security problems in C code. Modern C compilers protect against this, e.g. by reordering variables on the stack or employing special *canary* values on the stack to detect a buffer that overflowed.

A modern compiler can be instructed to omit these protections, e.g. by using the command line option `-fno-stack-protector`. More details can be found at `https://mudongliang.github.io/2016/05/24/stack-protector.html`.

d. What would the output be if line 5 was replaced by

```
static int foo = 0;
```

Briefly explain whether this change would solve the underlying problem.

Declaring a variable as `static` inside a function tells the compiler that the value of the variable has to be retained across function calls, i.e. whenever the function would be called again, the variable retains its value from the previous invocation (note that `main` is just a C function, even though you would probably not call `main` iteratively).

To retain its value, the variable has to be stored outside of the stack (which is destroyed after the return from the function). Thus, is is treated like a global variable and stored in a different memory area. Accordingly, overwriting the string `s` in `main` is unable to overwrite `foo` any longer.

However, other elements on the stack could be overwritten, e.g. the *return address*, which is also stored on the stack for some architectures (e.g. x86). The overwriting of return addresses on the stack using buffer overflows is another serious security problem, leading to attacks called *return-oriented programming*. (Note that this last paragraph of the explanation wasn't required for a correct answer but should serve to give you some more insight into what could happen.)

1.4 Functions and variables

Consider the following C program:

```
#include <stdio.h>

const int c = 1;
int d, counter = 0;

unsigned int rec(unsigned int number) {
  counter++;
  return rec(counter);
}

int main(void) {
  int a = rec(c);
  printf("%d\n", a);
  return 0;
}
```

a. Which memory segments are the function `rec()`, variables `c, d, counter, and a` as well as parameter `a` located in?

There was a typo in this question, it should read "as well as parameter `number`". Sorry...

- `rec()` is a function, its symbol is in the `text` segment.
- `c` is a `const` variable. In most systems, constant data types have a special write-protected memory segment `rodata` (read-only data).
- `d` is an uninitialized global variable which is stored in the `bss` segment.
- `counter` is an *initialized* global variable which is stored in the `data` segment.
- `a` is a *local* variable in `main`, so it is stored on the stack.
- `number` (parameter) is a *local* variable in `rec`, it is also stored on the stack.

b. What happens if you execute the compiled program? What changes if you add a local variable `char array[1000]` to function `rec`?

The function `rec` contains an *endless recursion*. Thus, for every subsequent invocation of `rec`, an additional stack frame (consisting, e.g., of storage space for the parameter `number` and a return address) is created on the stack, using some memory (e.g., 8 bytes). After a (large) number of iterations, the stack will attempt to write into memory of the C heap, which is usually caught by the OS and causes the application to be terminated.

When you also add a local variable `char array[1000]`, each stack frame requires much more space on the stack. Accordingly, the system will run out of stack space at a much earlier iteration. It might be difficult to see on modern systems, but the program would be terminated `earlier`.