NTNU | Norwegian University of Science and Technology

Compiler Construction Operating Systems

Theoretical Exercise 1: Solutions and Discussion

Michael Engel

1.1 Parameter passing

```
#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?

- Typical piece of code demonstrating the shadowing (overlaying) of variable names in C. We have:
  - a global variable a
  - a local variable a declared inside increment_with_value
- main sees value of global a (23) when calling increment_with_value
- increment_with_value has its own local variable a (located on the stack) ➢ it increments its value instead of the global variable
- When it returns, its stack frame (with its local var. a) is invalidated
- C uses call-by-value semantics ➢ value of a does not change in main
- Thus, a still has the value 23, which is returned by main

1.2 Symbols

```
#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;
}
```

If we compile the program 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`. Why?

- `nm` 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
- variable `b` is a parameter to the function `increment_with_value`,
  ➢ it is a local variable which is located on the stack
- location of `b` is relative to the current stack pointer and depends on, e.g.
  - the stack location in memory
  - the call depth if increment_with_value was called recursively
- static analysis of the executable by `nm` is unable to retrieve `b`'s address

1.3 C arrays

```
#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

- The answer depends on the protective measures your C compiler employs
- String s is an array of char with 12 elements ➛it uses 12 bytes on the stack
- The other local variable foo is located *after* s on the stack
- strcpy copies contents of string t to the memory addresses starting at the first byte of s – but t has 14 characters plus terminating zero byte
  - last bytes of t overwrite the memory in which foo is stored: ASCII characters for digits 2 and 3 and the terminating zero byte: 50 (digit 2), 51 (digit 3), 0
- foo was initialized to zero using int foo = 0;
- So the four bytes of foo are: 50 (digit 2), 51 (digit 3), 0, 0
- Little endian byte order: $50 * 2^0 + 51 * 2^8 + 0 * 2^{16} + 0 * 2^{24} = 13106$

1.3 C arrays

```
#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);
}
```

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

- A traditional C compiler provides no memory protection.
- `strcpy` does not check if the string to be copied fits into the destination memory space (since a string is just a pointer to a zero-terminated array of `char`s)
- A copy of 14(+1) bytes into a 12 byte buffer writes over the end of the buffer, "spilling" into the next variable on the stack: `foo`
- This is a classic example of a *buffer overflow* security problem!

1.3 C arrays

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

int main(void) {
   int foo = 0;
   char s[12];
   char *t = "0123456789012З";

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

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 (segmentation fault or similar)
- We've seen that this is a typical buffer overflow
- 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-protect.
- More details can e.g. be found at https://mudongliang.github.io/2016/05/24/stack-protector.html

1.3 C arrays

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

int main(void) {
   static 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);
}
```

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 ➛ value of the variable has to be retained across function calls
  - Whenever the function is called again, `a` retains its previous value
- To enable this, `a` has to be stored outside of the stack
  - Thus, is is treated like a global variable and stored in a different memory area
  - 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
  - another serious security problem: *return-oriented programming*

1.4 Func's and vars

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

- rec() is a function ➛ 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 ➛ bss segment
- counter is an initialized global variable ➛ data segment.
- a is a local variable in main ➛ stored on the stack.
- number (parameter) is a local variable in rec ➛ also on the stack

```
#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;
}
```

1.4 Func’s and vars

```
#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;
}
```

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

- `rec` contains an ***endless recursion***:
  - For every subsequent invocation of `rec`, an additional frame (with storage space for `number` and a return address) is created on the stack, using memory (e.g. 8 bytes)
- After a (large) number of recursive calls, the stack will attempt to overwrite the heap
  - usually caught by the OS which kills the application
- Adding a local variable `char array[1000]` will increase each stack frame’s size
  - The program will crash even earlier, since the stack grows faster