UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

S05: Functions, pointers, stacks

A. Schaller

16-896-375

Prof.: Dr. Luggen Michael

March 24, 2021

Series 5

1 Scope and Lifetime of C variables

Question 1. *Add in the source file global1.c a declaration for a global external char varaible named c and assign in the main() function the character 'a' to it. Compile with make, and briefly explain the error that you get and correct the source code.*

When declaring an `extern` function or variable, we expect it to be declared on a global level elsewhere. Meaning the constant will be declare with the other constant of the program and this region of the program isn't editable.

In the error log of `gcc`, we can also see that the compile prefix our variable with an underscore. Which must be some way to avoid confusion between the variables he should handle in a special manner.

```
Undefined symbols for architecture x86_64:
"_c", referenced from:
    _main in global1-145fb1.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see
   invocation)
make: *** [global] Error 1
```

Listing 1: Error message of gcc

2 Function Call Frames: CPU Registers + Stack and Text Segments

```
int g(int n) {
    return n+3;
}

int f(int i, int g(int), int n) {
    return i + g(n);
}

void main() {
    int n;
    n = f(g(2+1), g, 2+1);
    return n;
}
```

Question 1. *draw the stack, data and text segments, and the CPU registers of g(2+1) of f()'s first argument*

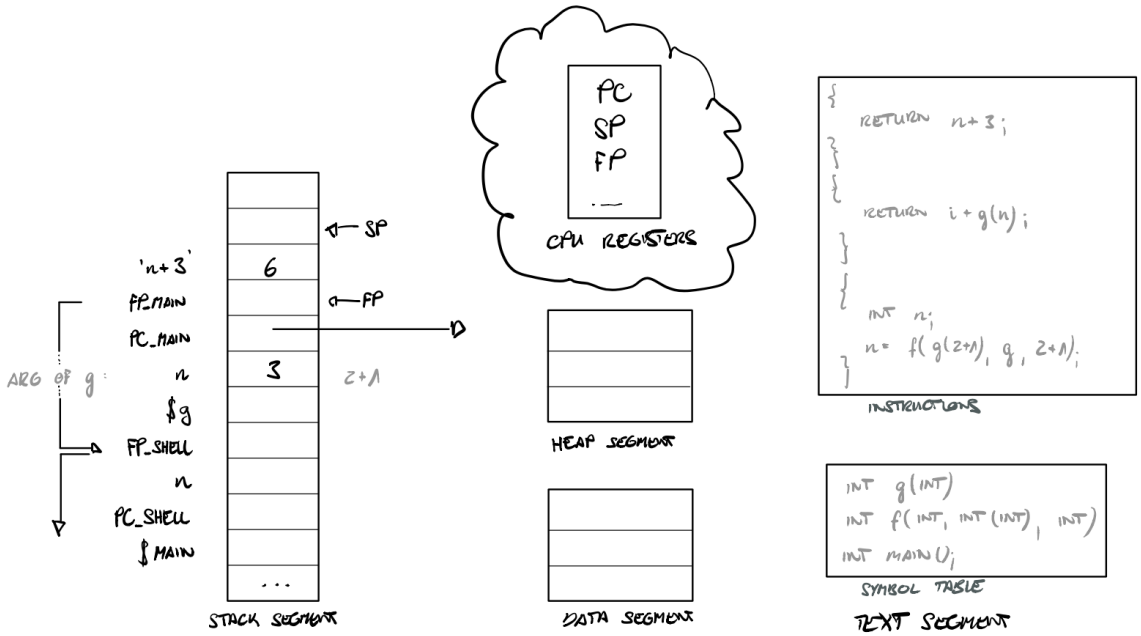

Figure 1: Stack segment, data segment and CPU registers of g(2+1) call

For the following questions, only the stack segments, at the point where the next instruction to be executed is the last semicolon ; of the function calls:

Question 2. *g(n) of f()'s body*

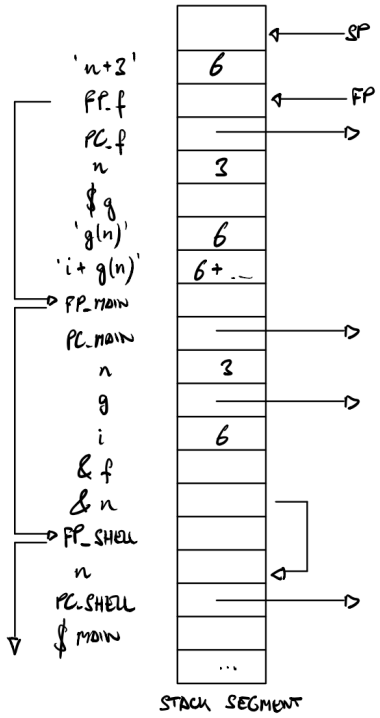

Figure 2: Stack segment of the g(n) call inside the f() body

Question 3. *f(...) of main()'s body*

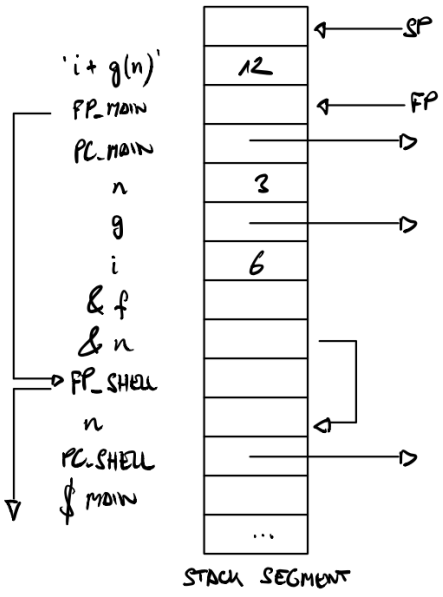

Figure 3: Stack segment of the f(...) call inside the main() body

Question 4. *main()*

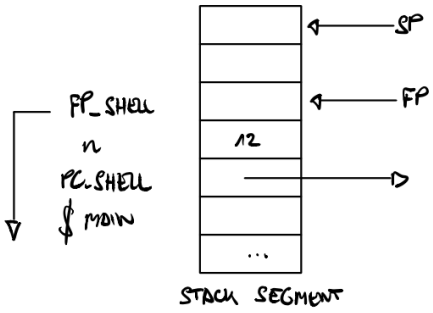

Figure 4: Stack segment at the end of the main() body

3 Pointers as Function Argument + Simplified Stack Segments

Question 1. *Draw the simplified (e.g. without the pc and fp pointers) stack segments for the function calls:*

1.1. *swap1(i, j)*

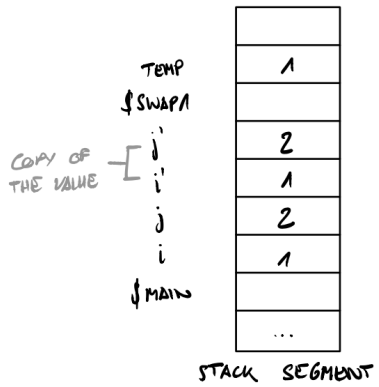

Figure 5: Stack segment of swap1(i, j) call

1.2. *swap1(&i, &j)*

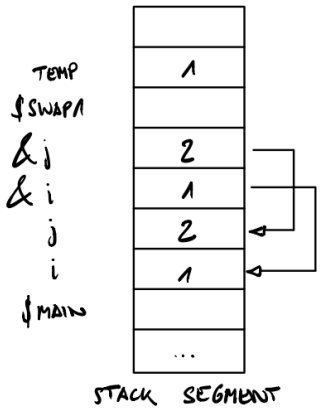

Figure 6: Stack segment of swap1(&i, &j) call

Question 2. *Then redo the Exercise 3 of Series 4, but this time for the function swap2()*

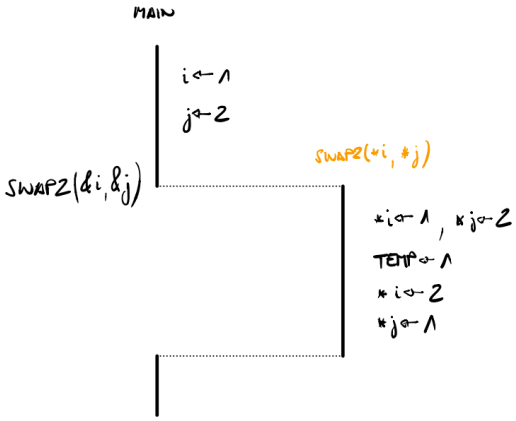

Figure 7: swap2 call control flow

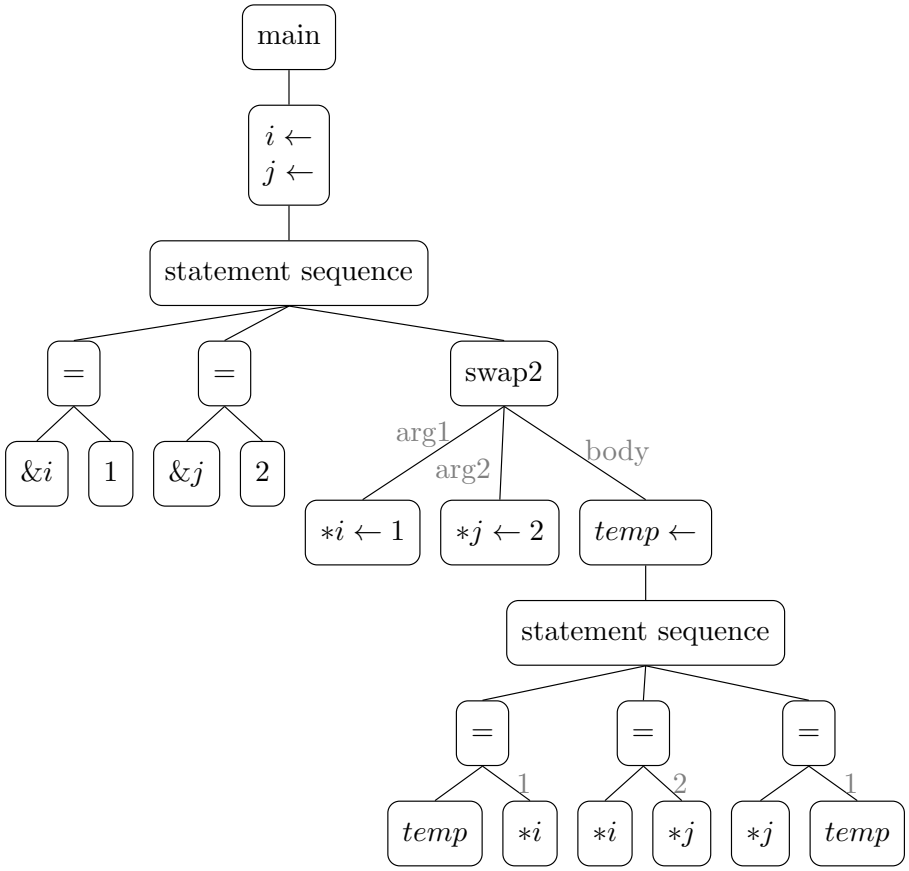

Figure 8: swap2 call AST

4 Malloc + Heap Segment

Question 1. *Draw the stack and heap segments when the pc register pointer points to the last semicolon ; of the following instruction:*

```
int * ip;
ip = malloc(sizeof int);
*ip = 3;
```

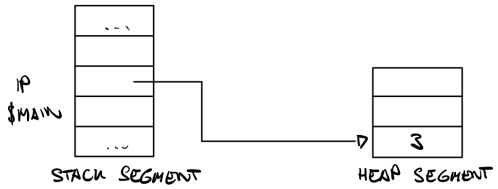

Figure 9: Stack and heap segments of the given program

Question 2. *And why is the casting (int *)malloc(sizeof int) not necessary?*

The casting is not necessary as any pointer have the same size/format anyway and the compiler helps doing the cast implicitely.

5 Calculator: A Stack Machine

5.1 Flat calculator

Question 1. *About system calls and cache memory. getch() operates as follows: if a character is present in the buffer it takes it, otherwise it calls getchar(). Explain why in the former case there is no system call involved, but there is one in the latter case. Explain also why this buffer acts as a kind of cache memory.*

1. No need for a system call if the buffer still has characters to read in his buffer, as the content of the buffer is solely handled by the program and the calls to `getch` or `ungetch`.

   However, to get the next char from the `stdin`, it will go through the system which helps the `stdin`, `stdout` and `stderr` pipelines.

2. In `getop`, we read as many digits as possible to compose an operation between 2 numbers. However, when we need to stop because we did read something else than a digit (like the following operation for example), we need to *cache* it when looking for the next operation (next call to `getop`).

Question 2. *About static local variables. Refactor getop() in the simplified case where it handles only operators and non-negative integers (and no floats as in the original version; this simplified code is present as comment in the full version) so that it doesn't need to use ungetch(), nor getch(). Test your solution on machine.*
Hint – use an internal static variable buf.

```c
#define BUFSIZE 100
#define GET_CACHED_OR_GETCHAR(buffer, i) (i > 0 ? buffer[--i] : getchar())

int getop(char s[]) {
    // like the global variables in getch/ungetch
    static char buffer[BUFSIZE];
    static int buffer_i = 0;

    // ...
    while ((c = GET_CACHED_OR_GETCHAR(buffer, buffer_i)) == ' ' || c == '\t')
        ;
    // ...
    while(isdigit(s[++i] = c = GET_CACHED_OR_GETCHAR(buffer, buffer_i)))
        ;
    // ...
    if (c != EOF) {
        if (buffer_i < BUFSIZE)
            buffer[buffer_i++] = c;
        else
            printf("Buffer is full: %s[%d]\n", buffer, buffer_i);
    }
    return NUMBER;
}
```

5.2 Modular calculator

Question 3. *About static global variables and functions. Why are the global variables in the files stack.c and ungetch.c declared as static, but not the functions?*

As they must be accessible in different methods: `push`/`pop` or `getch`/`ungetch`.

5.2.1 Header files

Question 4. *Why are the header files stack.h, getop.h and ungetch.h good programming practice?*

Including those header file at the top of a source file, will make sure that the compile knows that we are most likely gonna use the methods with the signature present in the header files. Methods that are implemented in their own `.c` files. The linker will later do his job to bind method calls to the proper method implementation.

Question 5. *Why are these files not protected by a conditional directive #ifndef (the case for stack.c is tricky)?*

It is better practice, especially when going back in time when compiling a running programs took much longer. Therefore, it was better to skip of going over the same header file twice.

However, having multiple similar includes are allowed, and now a days, with the powerful computer we have, it doesn't really matter.

Question 6. *Write a little program that demonstrate that this protection is not necessary.*

```c
#include <stdio.h>
#include <stdio.h> // just making sure to include it again

#include "./stack/stack.h"
#include "./stack/stack.h" // still no error here

int stack_error;

int main () {
    printf("No issue at all\n");
}
```