NTNU | Norwegian University of Science and Technology

Operating Systems Compiler Construction

Discussion of PE1 – 05.02.2021

Michael Engel

1.1 Recursion in C

- Write a simple C program (rec_sum.c) that calculates the sum of the numbers 1 to n using a recursive function int sum_n(int n). For example, a call to sum_n(5) should return the value 15. After calling the function, print out its return value like this:
  - The sum of numbers from 1 to 5 is 15.
- Use printf(3) to create the output. Please refer to the C crash course slides for details on printf.

1.1 Recursion in C

```c
#include <stdio.h>

int sum_n(int n) {
  if (n == 1) return n;
  return n + sum_n(n-1);
}

int main(void) {
  int n = 100000;
  printf("The sum of numbers from 1 to %d is %d.\n",
         n, sum_n(n));
}
```

```
$ gcc -o rec_sum rec_sum.c
$ ./rec_sum
The sum of numbers from 1 to 100000 is 705082704.
```

1.1 Recursion in C

```c
#include <stdio.h>

int sum_n(int n) {
  if (n == 1) return n;
  return n + sum_n(n-1);
}

int main(void) {
  int n = 1000000; // was 100000
  printf("The sum of numbers from 1
to %d is %d.\n",
         n, sum_n(n));
}
```

a. Experiment with different (also large) values for the parameter n.

   Why does the program fail to run correctly until its end beginning with a certain value of n?

   What is this value on your computer?

```
$ gcc -o rec_sum rec_sum.c
$ ./rec_sum
The sum of numbers from 1 to 100000 is 705082704.
# Change n to 1000000 and recompile:
$ gcc -o rec_sum rec_sum.c
$ ./rec_sum
Segmentation fault: 11
```

1.1 Recursion in C

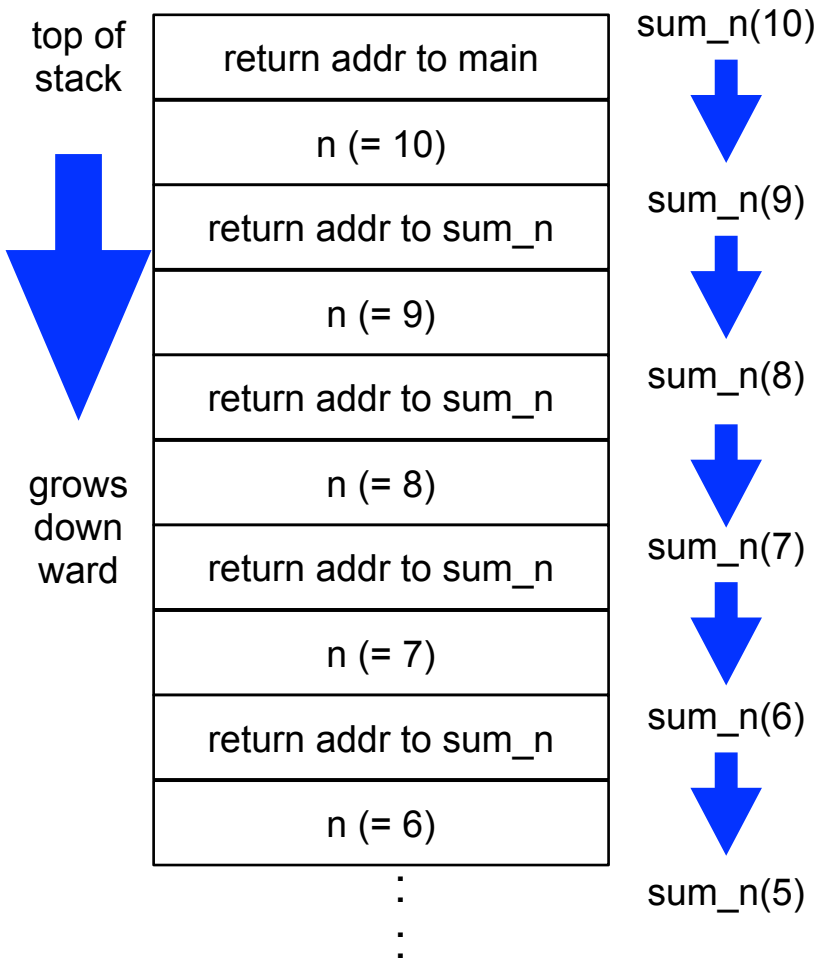
```
$ gcc -o rec_sum rec_sum.c
$ ./rec_sum
The sum of numbers from 1 to 100000 is 705082704.
# Change n to 1000000 and recompile:
$ gcc -o rec_sum rec_sum.c
$ ./rec_sum
Segmentation fault: 11
```

- **What happened here?**
  - Program runs correctly for n = 100.000
  - "Segmentation fault: 11" for n = 1.000.000
- *"A segmentation fault occurs when a program attempts to access a memory location that it is not allowed to access, or attempts to access a memory location in a way that is not allowed"*
  - So our program performs a non permitted memory access when n is too large!

1.1 Recursion in C

```
int sum_n(int n) {
  if (n == 1) return n;
  return n + sum_n(n-1);
}
```

- **What happened here?**
  - the function sum_n calls itself *recursively* n times
- Each recursive level stores local variables on the *stack*
  - *"I don't see any local variables in sum_n?!?"*
- The parameter n as well as additional information (e.g. the return address) is also stored on the stack ➛ at least 8 bytes per level!
- So, for a value of n =
  - 100.000 ➛ 800.000 bytes
  - 1000.000 ➛ 8000.000 bytes
- The stack size is limited to a certain size in most Unix systems



1.1 Recursion in C

```
$ gcc -o rec_sum rec_sum.c
$ ./rec_sum
The sum of numbers from 1 to 100000 is 705082704.
# Change n to 1000000 and recompile:
$ gcc -o rec_sum rec_sum.c
$ ./rec_sum
Segmentation fault: 11
```

- **When does the program crash?**
  - Program runs correctly for n = 100.000
  - Segmentation fault for n = 1.000.000
- Use a *bisection approach (divide and concquer):*
  - try the average of the two values n = (n1+n2)/2
  - if there is a correct result, recurse for interval [n,n2] else [n1,n]
- 550.000: segmentation fault ➛ try [100.000,550.000]: n = 325.000
- 325.000: segmentation fault ➛ try [100.000,325.000]: n = 212.500:
- 212.500: "The sum of numbers from 1 to 212500 is 1103394770."

1.1 Recursion in C

- 550.000: segmentation fault ➛ try [100.000,550.000]: n = 325.000
- 325.000: segmentation fault ➛ try [100.000,325.000]: n = 212.500
- 212.500: "The sum of numbers from 1 to 212500 is 1103394770."
  ➛ try [212.500,325.000]: n = 268750
- 268.750: segmentation fault ➛ try [212.500,268.750]: n = 240.625
- 240.625: "The sum of numbers from 1 to 240625 is **-1114455447**."

- Wait, a ***negative sum?***
  - This is an ***integer overflow***
  - Two's complement 32 bit integers have a range of -2^{31} (-2,147,483,648) to $+2^{31}-1$ (+2,147,483,647)
  - If the sum is > $+2^{31}-1$, bit 31 (the most significant bit, MSB) is set

    ➛ **interpreted as negative number**

    Integer overflows are not caught in C (too much overhead!)
- So it's a good question what actually constitutes a "correct result"

1.1 Recursion in C

```
$ gcc –o rec_sum rec_sum.c
$ ./rec_sum
The sum of numbers from 1 to 100000 is 705082704.
# Change n to 1000000 and recompile:
$ gcc –o rec_sum rec_sum.c
$ ./rec_sum
Segmentation fault: 11
```

- **Typing and compiling is very tedious…**
  - Can we find the maximum value of n ***automatically?***
- Idea 1: loop inside the C main function

```
int main(void) {
  int n1 = 100000, n2 = 1000000, n;
  while (1) {
     sum_n((n1+n2)/2);
     if (no_crash) n1 = (n1+n2)/2;
               else n2 = (n1+n2)/2;
  }
}
```

Unfortunately, the loop cannot continue to run when the program crashes…

We'll see how to handle situations like this when we discuss signals in Unix!

1.1 Recursion in C

```
$ gcc -o rec_sum rec_sum.c
$ ./rec_sum
The sum of numbers from 1 to 100000 is 705082704.
# Change n to 1000000 and recompile:
$ gcc -o rec_sum rec_sum.c
$ ./rec_sum
Segmentation fault: 11
```

- Idea 2: loop in the shell that runs the bisection
  - This approach would actually work (but would be slow)
  - Here, you should try to pass the value for n on the command line

```
int main(int argc, char **argv) {
  // In real life, we should check for errors here
  int n = atoi(argv[1]);
  printf("The sum of numbers from 1 to %d is %d.\n",
         n, sum_n(n));
}
```

1.1 Recursion in C

```c
#include <stdio.h>
int a;
int b = 42;
char c[23];
int sum_n(int n) {
  static int d;
  int e;
  if (n == 1) return n;
  return n + sum_n(n-1);
}
int main(void) {
  double f;
  int n = 100000;
  printf("&a = %p\n", &a);
  printf("&b = %p\n", &b);
  // … and the others…
  printf("sum to %d is %d.\n",
         n, sum_n(n));
}
```

In addition, create a number of different variables (different types, global, local, initialized, uninitialized) in your program and print their addresses in memory in the main() function.

You can print addresses of variables using printf(3) like this:

```
printf("Address of foo is %p\n", &foo);
```

1.1 Recursion in C

```c
#include <stdio.h>
int a;
int b = 42;
char c[23];
int sum_n(int n) {
  static int d;
  int e;
  if (n == 1) return n;
  return n + sum_n(n-1);
}
int main(void) {
  double f;
  int n = 100000;
  printf("&a = %p\n", &a);
  printf("&b = %p\n", &b);
  // … and the others…
  printf("sum to %d is %d.\n",
         n, sum_n(n));
}
```

Printing the address of some variables in main doesn't work:

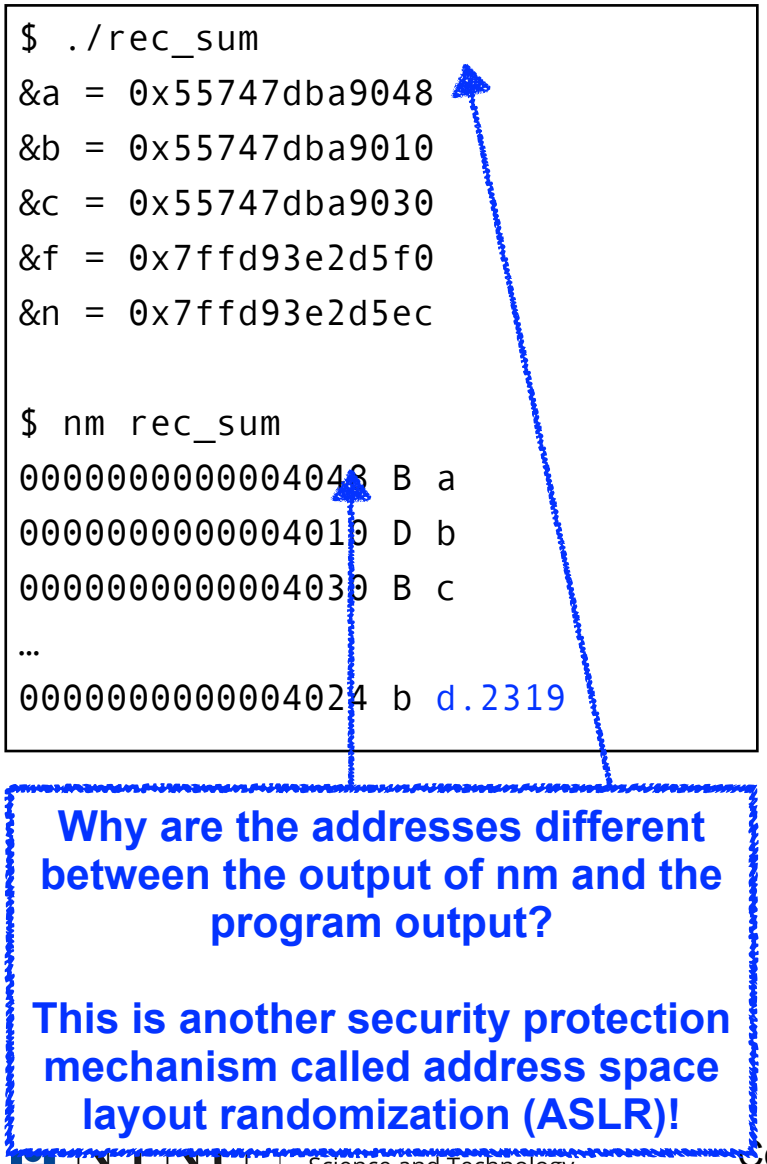
```
printf("&d= %p\n", &d);
printf("&e= %p\n", &e);
```

```
$ gcc -o rec_sum rec_sum.c
rs1.c:20:24: error: use of undeclared identifier 'd'
  printf("&d = %p\n", &d);
                      ^
rs1.c:21:24: error: use of undeclared identifier 'e'
  printf("&e = %p\n", &e);
```

Why?

- e is local in sum_n ➛ not visible in main
- d is static local in sum_n
  - ➛ not visible in main
    - but treated as global

1.1 Recursion in C



We can print the addresses of all *visible* variables inside of main

To print the addresses of ***d*** *and* ***e****, you would have to print them inside the function* `sum_n`*!*

The Unix `nm` tool can give you the addresses of all global variables

In addition, it provides the address for the static variable `d`, which was internally given the unique name `d.2319`!

1.1 Recursion in C

```
$ cat rec_sum.c
…
int main(void) {
  double f;
  int n = 100000;
…

$ ./rec_sum
&a = 0x55747dba9048
&b = 0x55747dba9010
&c = 0x55747dba9030
&f = 0x7ffd93e2d5f0
&n = 0x7ffd93e2d5ec
```

b. Which distance (in bytes) do the addresses of two variables have that are declared one after the other in main()?

Explain why the distance is the one you see

Here, we first have f, then n on the stack:
`&f = 0x7ffd93e2d5f0`
`&n = 0x7ffd93e2d5ec`

(remember – the stack grows downwards in memory!). So the distance of the two variables in memory is:
`0x7ffd93e2d5f0-0x7ffd93e2d5ec = 4`

This is the size of the int variable n

1.1 Recursion in C

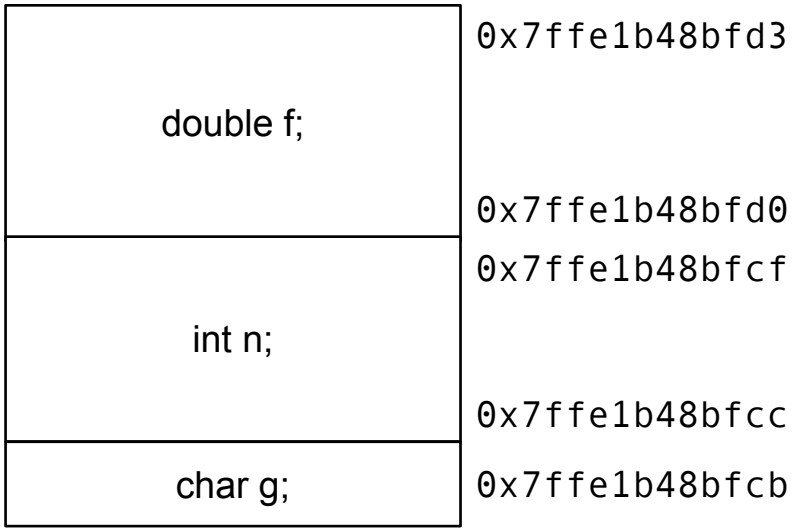
```
$ cat rec_sum.c
…
int main(void) {
  double f;
  char g;
  int n = 100000;
  printf("&f = %p\n", &f);
  printf("&g = %p\n", &g);
  printf("&n = %p\n", &n);
$ ./rec_sum
&f = 0x7ffe1b48bfd0
&g = 0x7ffe1b48bfcb
&n = 0x7ffe1b48bfcc
```

Depending on your compiler, its settings (flags) and your OS, local variables can be reordered. In addition, variables using > 1 byte are usually naturally *aligned* in memory

b. Which distance (in bytes) do the addresses of two variables have that are declared one after the other in main()?

Let's try to add a char variable g in between f and n now!

Here, the compiler **reordered** the variables that are on the stack now:



1.1 Recursion in C

```
$ ./rec_sum
&a = 0x55747dba9048
&b = 0x55747dba9010
&c = 0x55747dba9030
&f = 0x7ffd93e2d5f0
&n = 0x7ffd93e2d5ec
```

On Linux, an executable program is not loaded at virtual memory address 0, but at a higher address

You can also print the addresses of the functions `main` and `sum_n` to see where they are located in memory

c. Why is a global int variable located at a completely different address?

Here, we meant "completely different from the local variables". We'll try to improve the precision of our questions in the future :-).

Local variables are located on the stack, which grows downward from "high" addresses (`0x7fffffffffff` here)

The data (and bss) segment for global variables are located low in memory, usually behind the text segment (here: `0x55xxxxxxxxxx`)

1.1 Recursion in C

```
$ cat rec_sum.c
…
int sum_n(int n) {
  int e;
  printf("&e = %p\n", &e);
  if (n == 1) return n;
  return n + sum_n(n-1);
}
$ ./rec_sum
&e = 0x7fff2c079ef4
&e = 0x7fff2c079ec4
&e = 0x7fff2c079e94
&e = 0x7fff2c079e64
&e = 0x7fff2c079e34
&e = 0x7fff2c079e04
&e = 0x7fff2c079dd4
&e = 0x7fff2c079da4
&e = 0x7fff2c079d74
&e = 0x7fff2c079d44
…
```

d. Why does the address of a local variable in the recursive function ***decrease*** the higher the level of recursion is?

This question was probably a bit redundant, but I wanted you to experiment a bit more.

Since the stack where the local variable is located (here: **e**) grows downwards, a new ***stack frame*** is allocated below the current in memory for each recursion.

Each recursion has its own copy of local variables stored in the stack frame, so the addresses of the local vars. ***decrease***

1.1 Recursion in C

```
$ cat rec_sum.c
…
int sum_n(int n) {
  double f;
  char g;
  int n = 100000;
…
$ cc -g -o rec_sum rec_sum.c
```

Question from the OS Q&A yesterday: "Adding all the printfs should be unnecessary – isn't there a better way to get at the addresses?"

Unfortunately, you cannot iterate over local variables in C (no *introspection*), so this can't be done from within a program. We could use a ***debugger*** such as **gdb**

```
$ gdb ./rec_sum
(gdb) b main
Breakpoint 1 at 0x11d2: file rs1.c, line 10.
(gdb) run
Starting program: /home/me/rs1
Breakpoint 1, main () at rs1.c:10
10   int main(void) {
```

Unfortunately, gdb doesn't provide an easy way to print the addresses of local variables…

```
(gdb) info locals
f = 0
g = 0 '\000'
n = 32767
(gdb)
```