Hacking in C

Programming language C

↓

raw access to platform
(so less help to stay out of trouble)

Syntax: rules which say what legal program texts are.

Semantics: The meaning of "legal" programs

Compiler:

0 & Preprocessor —> add and remove code from your file (#include)

1. represent all data as bytes
2. decide where pieces of data are stored (memory managment)
3. translate all operations to basic instruction set of the CPU.
4. provide some "hooks" so that at runtime the CPU and OS can handle function calls.

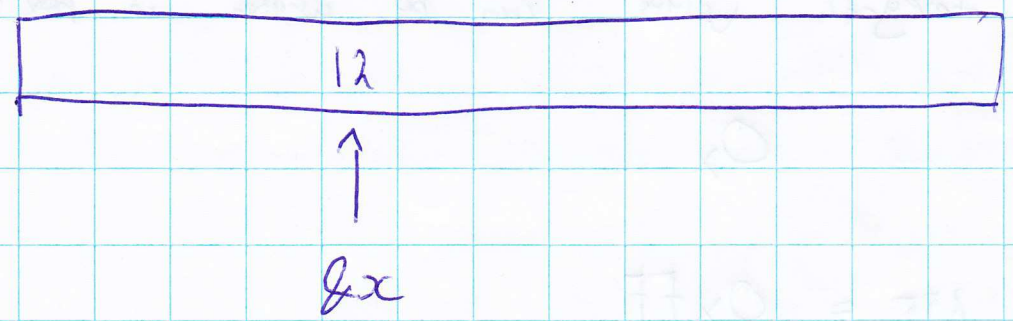
Data types:

character 'a'
string "Hello World"
floating point number 1.345
array of int's {1, 2, 3, 4, 5}
complex number 1.0 + 3.0 * I

Data is stored in:
- a register
- cache
- ram
- harddisk

We can find out where data is allocated using the & operation:

```
int x = 12
printf("%p", &x)
```



Char: simplest data type

↓

one byte

```
char c = '2';
```

↓

```
char e = 50;
```

the same

Signed vs unsigned

↓ ↓

-128 ... 127 0...255 (char)

Integral values can be written in hex using:

0x....

Ex 255 = 0xFF

Explicit type casts

```
int i = 23456;
char c = (char) i; // drops the higher order bits
float f = 12.345;
i = (int) f; // drops the fractional part
```

Some printing notations:

- %i = integer
- %o = octal notation
- %x = hex notations
- %X = hex with capitals

---

stdint.h : library for standard bit-sized variables

stdint.h defines variables with standard bit lengths:

- ~~uint_16_t~~ uint16_t = unsigned 16 bit integer

Representation :

storing a long long x = 1 in memory:

Big endian: 00 00 00 01

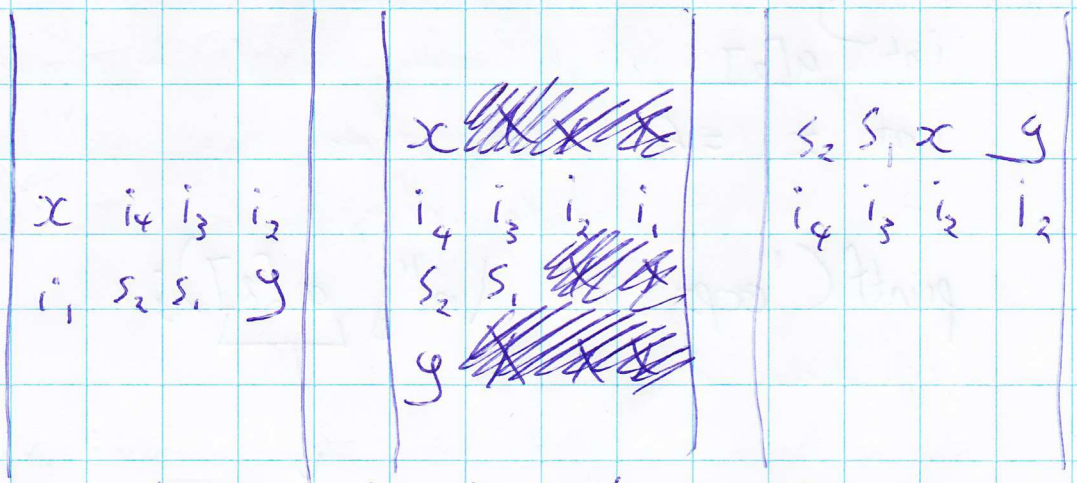
Little endian: 01 00 00 00

II Data alignment

Compiler introduce padding or change the order of data in memory to improve alignment.

```
char  x;
int   i;
short s;
char  y;
```

Can be aligned as:



A: minimal memory B: optimal aligned C: compromise?

CPU loads words (4 bytes) of data every time. So A is a waste of instruction power.

Arrays: a collection of data elements with the same type and a constant size

↓

Bounds are not checked
(can cause buffer overflow attacks)

Program:

```
int y = 7
int a[2]
int x = 6

printf("oops %i \n", a[2]);
```

↓

can be x or y
No guarantees
↳ program can also crash

By overrunning an array we can try to reverse-engineer the memory layout.

Alignment: arrays have a guaranteed alignment:

a[1] is always allocated to the right of a[0]

Arrays are always ~~pad~~ passed by reference to functions:

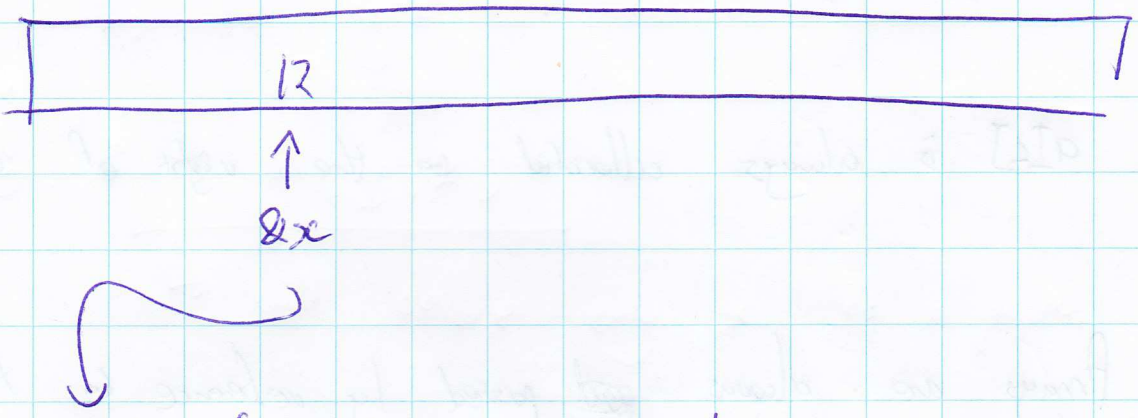
```
void increase_elt (int x[]) {
    x[1] = x[1] + 23;
}

int a[2] = {1, 2};
increase_elt (&a);
```

Will return: 25

Pointers

```
int x = 12;
```



will return the memory adress of x

↓

32 bit: 4 byte adress
64 bit: 8 byte adress

Declaration of pointers:

```
int *p;    // pointer to int
float *f;  // pointer to float
```

& Dereferencing:

```
int y;
int z;
int *p;
```

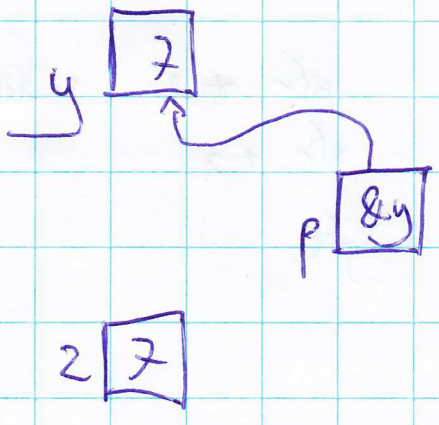
Creating a pointer to y:

```
y = 7;
p = &y; // assign adress of y to p
```

Getting the value of the pointer:

```
z = *p; // gives z the value of what p
          points to.
```

dereferencing



```
int *p = &y;
int* p = &y;
```

↳ the same

```
int* x, y, z;
```

↳ will declare only x as a pointer.

Pointer arithmetic: + and - can be used on pointers, adding 1 to a pointer will go to the next location.

```
int *ptr;
char *str;
```

ptr + 2 means ptr + 2 * sizeof(int)

str + 2 means str + 2

↳ char = 1 byte

Strings:

```
char *msg = "hello, world!";
char *t   = msg + 6;
```

will point to the string world.

Aliases: Two pointers pointing to the same location

Pointers to numbers: intptr_t can be used to inspect memory without the need to worry about how pointers will behave when adding stuff

```
int *p;
intptr_t i = (intptr_t) p;
p++;
i++;
```

p++ adds 1 * sizeof(int)

i++ adds one.

String problems: a string is an array of char

1. There are no bounds checked
2. You have to make sure the string is properly terminated with '\0'

III Memory management

Process management:

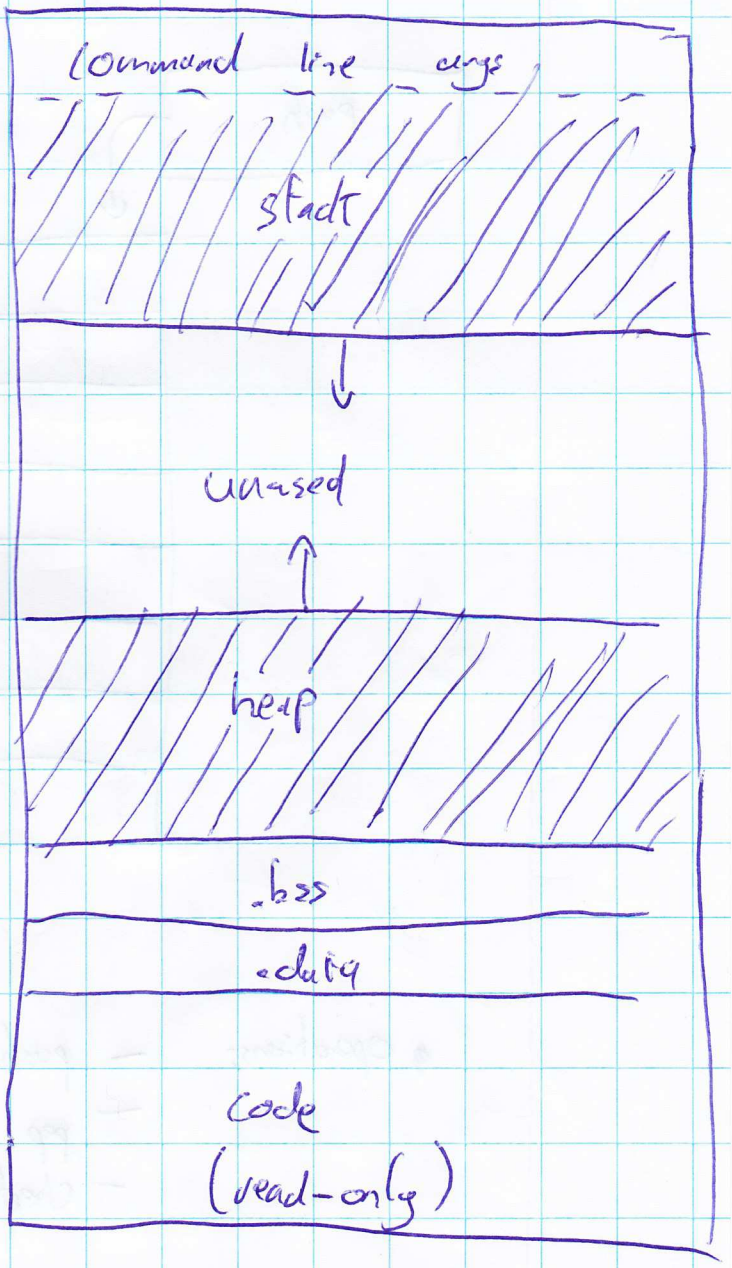
Stack: - local variables
- environment variables

heap: - dynamic memory

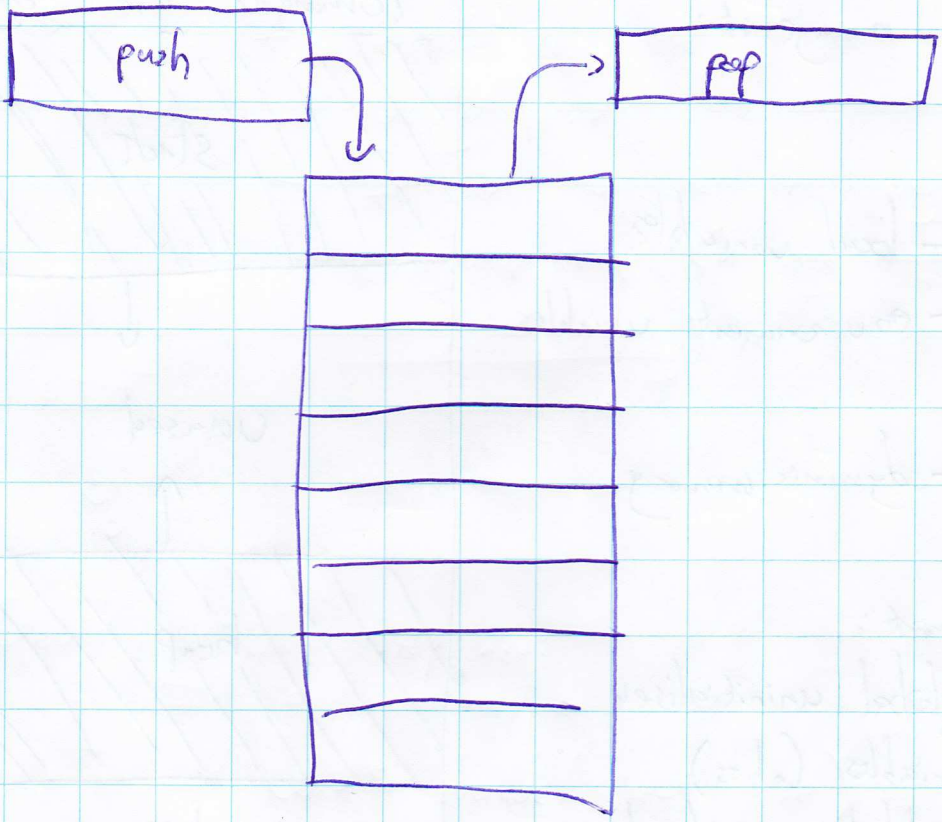
data segment:
- global uninitialised variables (.bss)
- global initialised variables (.data)

Code

Code segment: - read-only



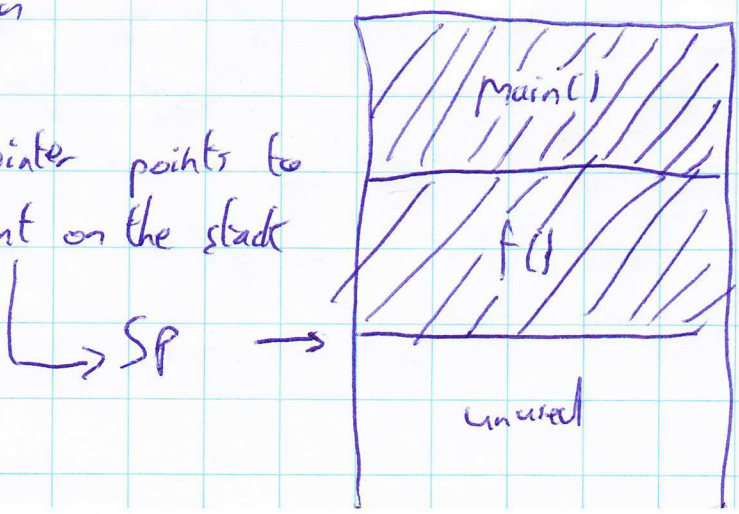
Stack: ~~first in~~ last in, first out

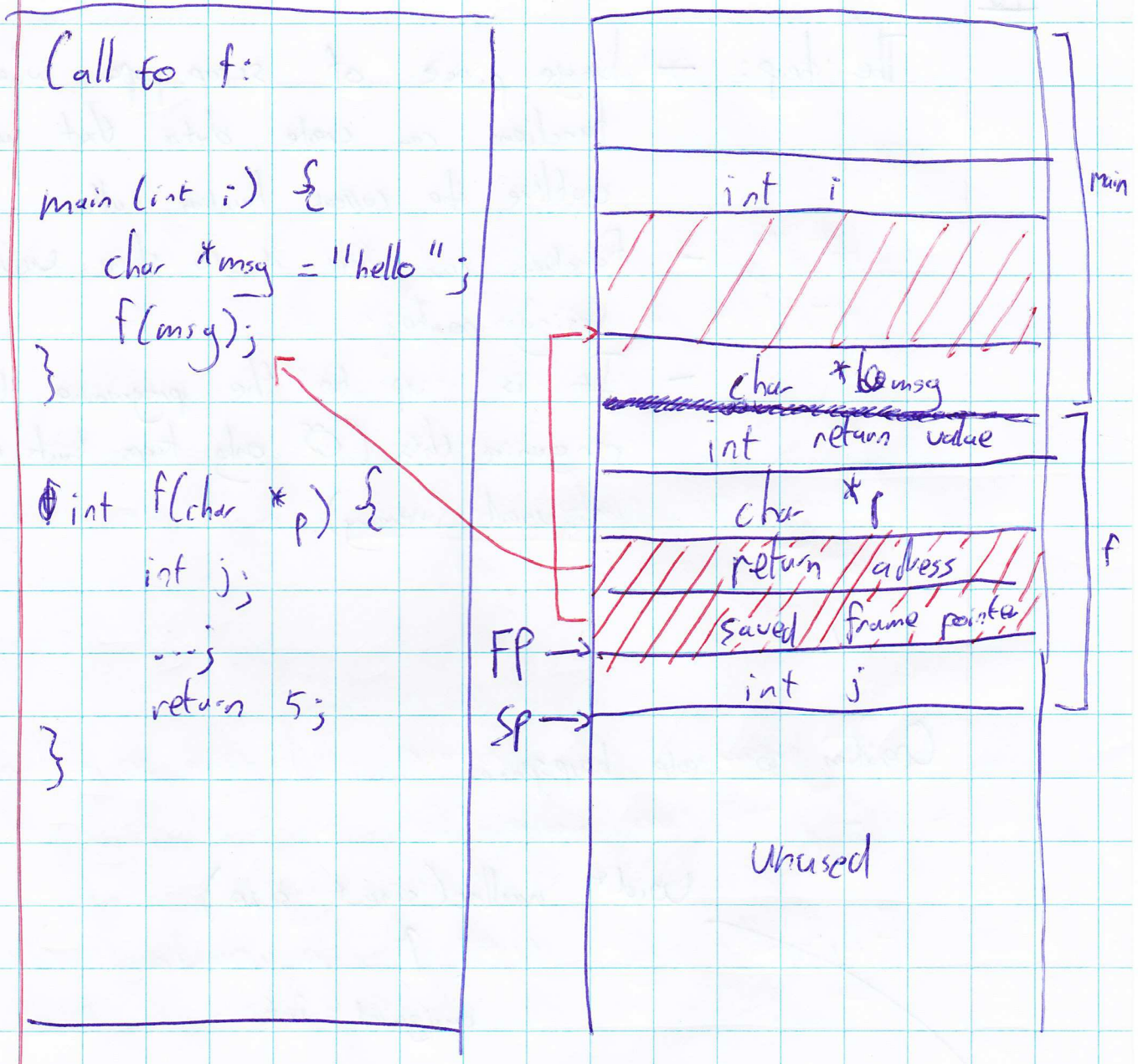


- operations:
  - push
  - pop
  - check (if the stack is empty)

The stack consists of <u>stack frames</u> for each function

The stack pointer points to the last element on the stack





The return adress points to a piece of code, the frame pointe to the frame pointe of the previous frame.

→ sometimes data can be stored in registers

IV

The heap: — large piece of scrap paper where functions can create data that will outlive the current function call.
— Functions can use it to share values using pointers
— It is up to the programmer to organise this (OS only keeps track of unused memory)

Operation to create heap space:

```
void* malloc(size_t size)
```

↑ unsigned int

This function returns a pointer to where the data is allocated, if it fails the pointer will be null.

A void* can be converted to any pointer without a cast:

```
int* a = void* malloc(100);
```

Programs should always check if malloc fails:

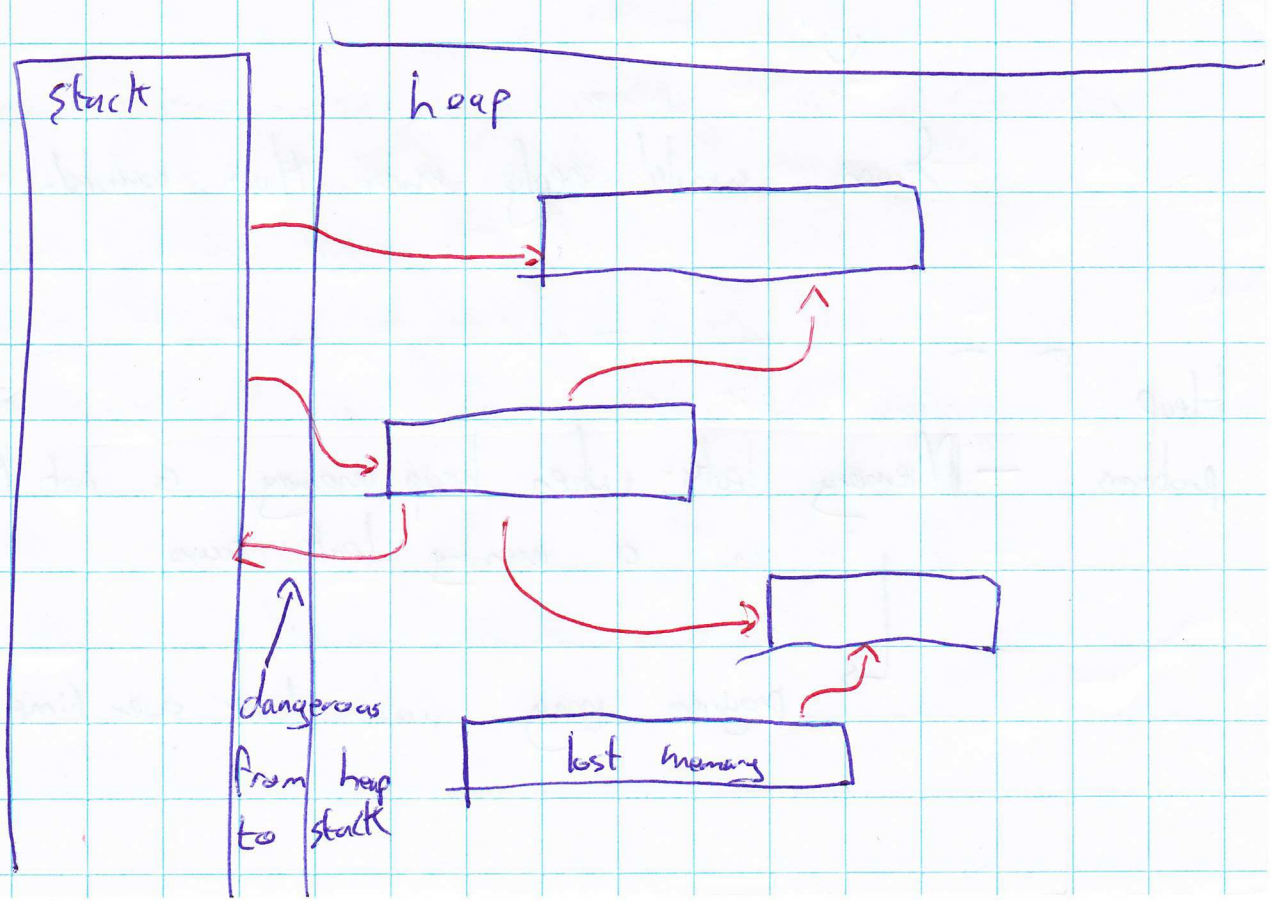
```
int** *table = malloc(len * sizeof(int));
if (table = NULL) { exit(); }
```

When the heap memory is no longe needed, it should be freeed:

```
void free(void* p)
```

So after the table functia should be:

```
free(table);
```



Memory initialization

```
int i;
printf("i is %i.\n", i);
```

Memory is <u>not initialised</u> so i can have any value.

↓

A program can behave differently each time

Heartbleed bug: check if a connection is still alive

↓

Server would reply more than needed.

Heap problems

- Memory leaks: when heap memory is not freed a memory leak occurs.
  - ↳ program may run slower over time

- Dangling pointers: a pointer to memory that has been de-allocated is called a dangling pointer

- Free: never double free (errors)

```
char *x = malloc(1000);
Free(x);
Free(x); // error
```

Garbage Collection: in modern languages there is a garbage collector that clears unused memory

↓

Downside: performance.

Stack vs Heap

Stack:

- variables are allocated and de-allocated automatically
- allocation is much faster than for the heap
- data can be used without pointers
- data needs have to be known at compile time
- stack space may run out (infinite recursion)
- size is fixed

Heap:

- (de)-allocation has to be done manually
  → error prone
- allocation of heap memory is slower than for stack memory
- to access data on the heap, you must use pointers
  → error prone
- more flexible, and must to be used when data needs are not known at compile time
- heap space may run out, but can grow during the lifetime

V

Attacking the stack

Goals : - leak data (heartbleed)
- corrupt data
- corrupt program execution:
  a: crashing
  b: doing more interesting stuff

Attacks result in breaking:

1. Confidentiality
2. Integrity (of data and program execution)
3. availability

Format string attacks

A format string: "j is %i"

↓

without a value given to "%i", "%i" will reinterpret the top of the stack as an int and print it.

Normal syntax:

```
printf("j is %i.\n", j);
```

↓

When this is executed, it will print the top of the stack.

```
int main(int argc, char** argv) {
    int pincode = 1234;
    printf(argv[1]);
}
```

This program will print the stack when giving malicious input (such as %x%x%x).

- printf("%s"); will print interpret the top of the stack as a pointer and will print from that pointer untill a null terminator.

- printf("%n"); will interpret the top of the stack as a pointer and will write the number of printed chars to that address

Malicious strings:

- \xEF\xCD\xCD\xAB %x %x ... %x %s

  with the right number of %x, an atacker can read the data stored on adress ABCDCDEF

- \xEF\xCD\xCD\xAB %x %x ... %x %n

  with the right numbe of %x, an atacker can write data to ABCDCDEF

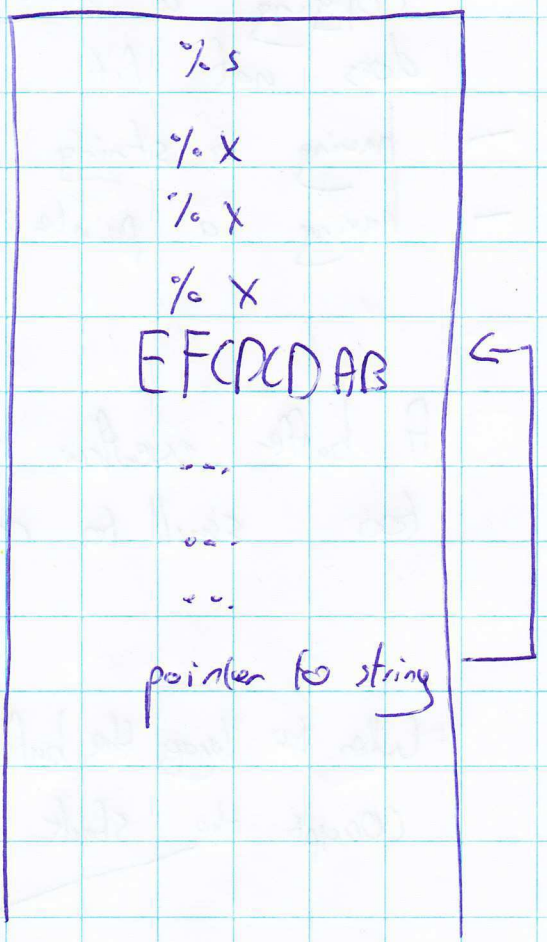
Stack layout:

printf("\xEF\xCD\xCD\xCD %x %x ... %x %s");

use this adress for %s →

Second %x →

First %x →



Fix:

```
printf(str);
```

write this instead:

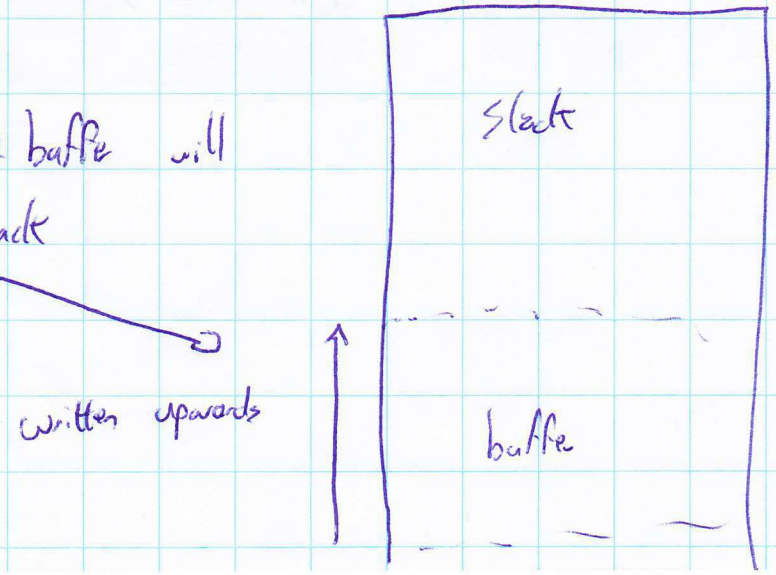
```
printf("%s", str);
```

Buffe overflows

- going outside of array bounds
- copying a string into a buffe where it does not fit
- having a string without a NULL terminator
- having a pointer pointing to the wrong place

A buffe overflow occurs when the buffe is too small to contain the give information

When too large the buffe will corrupt the stack



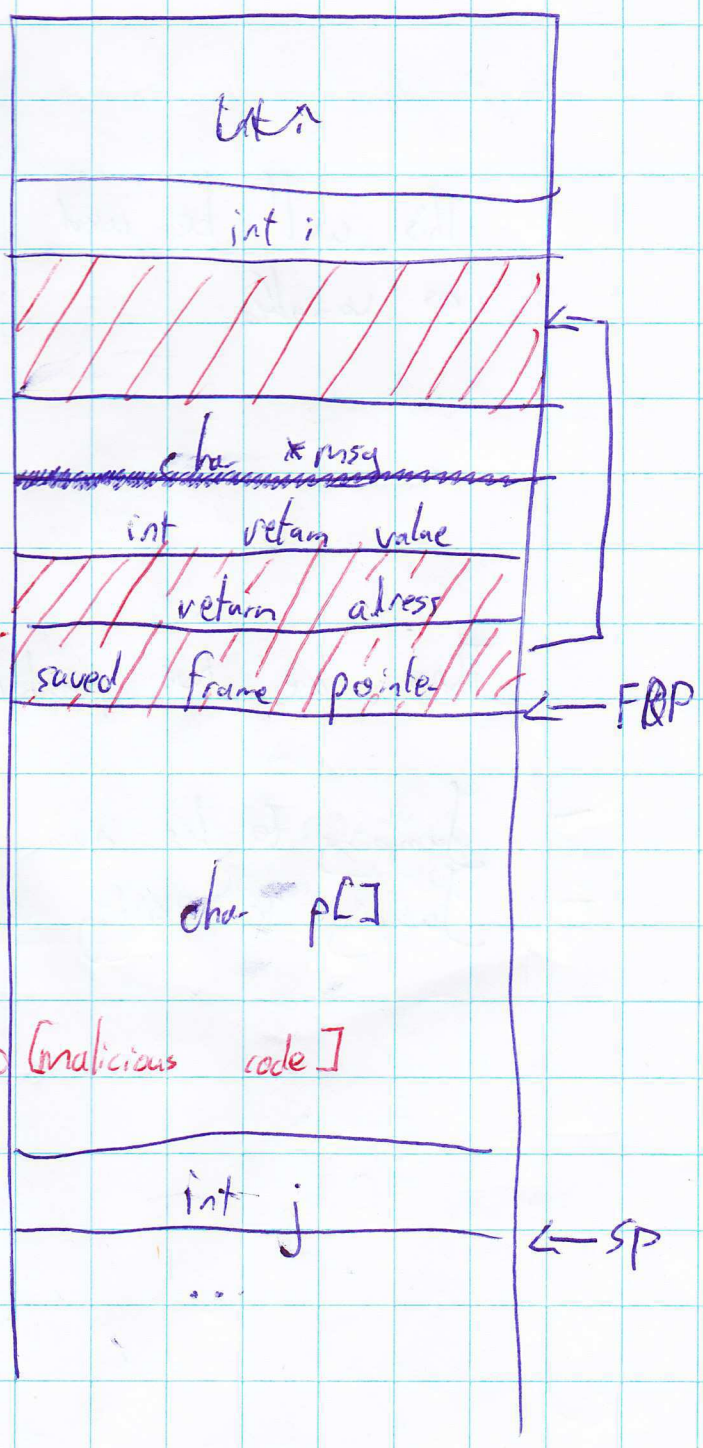
Gets ([buffer]);

gets will ask the user input and will store it in a given buffer. DON'T EVER USE THIS ☹

Corrupting the stack:

```
main(int i) {
    char *msg = "hello";
    f();
    printf("%s", msg);
}

int f() {
    char p[20];
    int j;
    gets(p);
    return 1;
}
```



We could put code in the p[] and overwrite the return adress to return to p[]:
(in red)

When corrupting the frame pointer to point to p[], the execution will continue and will use variables stored below the frame pointer.

This will be used as variables.

There are two possibilities:

- Jumping to his own malicious code
- Jumping to existing code with a malicious stack frame

Spawning shell code

- we need to get some shell code in memory
- overwrite the return adress on the stack to the place where the shell code is

↓

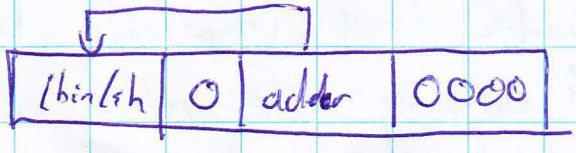
The attacker can then do anything within the rights & permissions of the attacked program.

```
void main(int argc, char **argv) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);
}
```

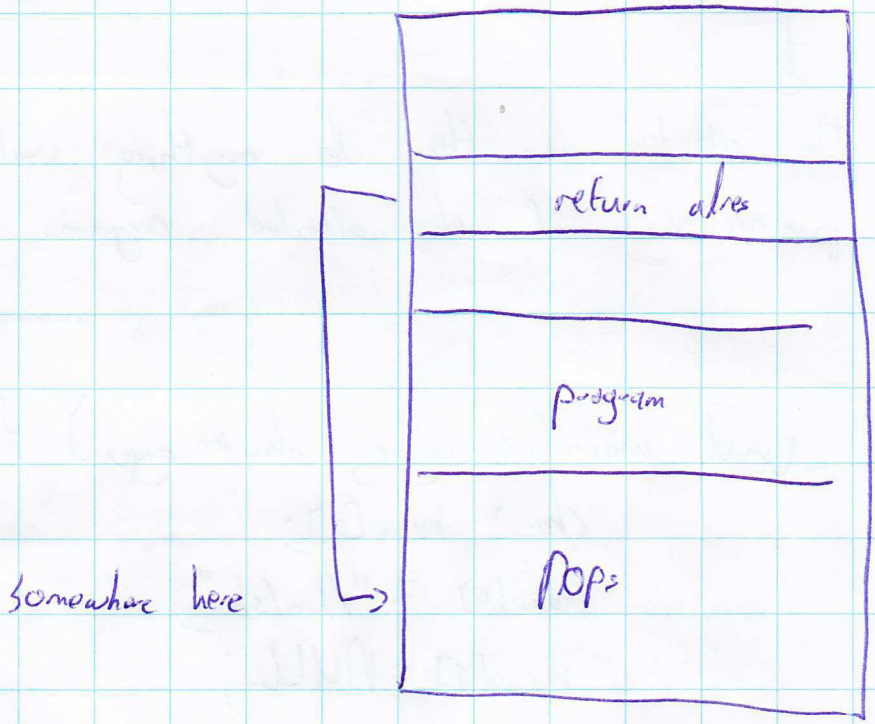
null terminated string

adress of that string followed by NULL



In most cases, the exact adress of the buffer is unknown, there are two solutions:

1. NOP sled (0x90)



2. Jump using a register

Find a registe that points to the buffer:

- ESP
- EAX

Locate an instruction that jumps/calls using that register & overwrite the return adress with the adress of that instruction.

VI Defenses

Strategies:

- ~~Defend~~ Prevent
- Detect
- React

Different levels:

- Program level → remove insecure code
- Compile level → detect insecure code
- Operating system level → make exploitation more difficult

Stack protection with canaries:

detect if some buffer overflow wrote over the boundry of the stack frame.

The compiler inserts code to:

- add a canary between local variables
- at the end of each function, it checks if the canary is still alive
- when this is not the case abort the program

↓

- foo ~~→~~ fstack-protector option

Strong canaries:

- Generate a random canary each time
- Put '\0' inside a canary

↓

Overflowing with some string copying function will become harder.

- Let the canary be the XOR value of some "master" canary value and the return address on the stack.

Another tactic to deny execution on the stack: <u>make the stack non-executable.</u>