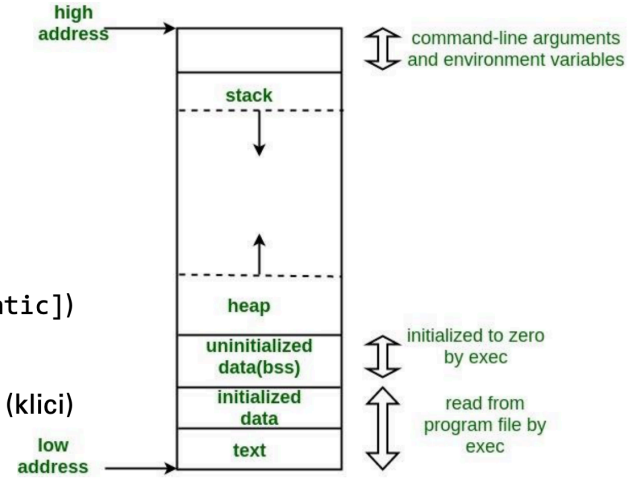
+- sklad: dajemo vrednosti gor in dol

- kopica: gor hranimo podatki, ki so preveliki za na sklad

Sklad/kopica?

- `.text`: koda v strojnem jeziku
- `.data`: inicializirani podatki (globalne spremenljivke `[const]`)
- `.bss`: neinicializirani podatki (npr. `[static]`)
- `Heap`: kopica -> `malloc`
- `Stack`: sklad -> lokalne spremenljivke, (klici) funkcij



- malloc = računalniku rečemo, da naj nam da spomin, lahko ga da ali pa ne; je sistemski klic:
  - pomembno tudi iz vidika varnosti vedeti, kam lahko nek proces dostopa, če imamo neke skrivnosti (ključe), ne morejo drugi procesi tistega brati in pisati
- ko dajemo neke konstante v program, grejo na stack
- heap in stack rasteta eden proti drugemu - to je že malo zaščita, da ne zraste preveč - takrat rabimo povečati procesno sliko

malloc:

- vrača void pointer
- `ptr = (int*) malloc(100 * sizeof(int));`
- ne veš koliko velik je int, zato rabiš sizeof
- `if (ptr == NULL)` => pomeni, da nismo dobili pomnilnika
- za sprostitev - free
- calloc in realloc

- calloc je kot malloc, ampak da vse še na 0
- če hočemo array povečati, rabimo realloc
- sistemski klici so dragi
- `a[0]` - v oklepaju povemo odmik od `a`
- `*(p + 0) = p[0]`
- `free` samo reče, da ne boš več uporabljal spomina, ampak ga ne izbriše; OS ta del prostora lahko da nekomu drugemu, ki potem čez piše
- če naredimo `malloc`, `free`, `malloc`, se bo drugi malloc najbrž izvedel na istem delu, proba reclaimat ta prostor; dobimo isti prostor, ker je tam v segmentu prazen prostor + hočemo ohranjati pomnilniško lokalnost (stvari, ki nam hkrati prav pridejo, jih želimo imeti čim bližje v pomnilniku, da jih več pride na predpomnilnik)

printf:

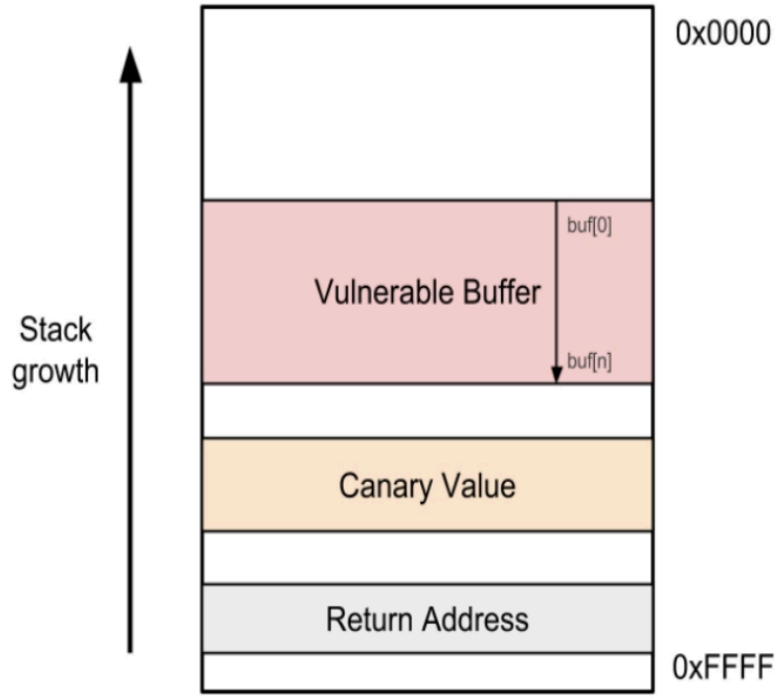
- `printf("%.0f", 13.5)` - izpiše 14, ker format zaokroži

scanf:

- bere do presledka
- če vpišemo daljši string, kot smo alocirali prostora, nam bo C še vedno pustil, ker bomo pisali po delu heapa, ki je del našega programa, če tudi nismo alocirali tega dela heapa; lahko prepišemo stvari, ki so naprej na heapu, čeprav tist del ni res del stringa, ki ga hočemo s scanf prebrati
- zato ne uporabljaš scanf

gets:

- stack smashing = na stack pišemo več kot nam je dovoljeno
- na stack damo med vrednosti, ki jih želimo uporabiti, "kanarčke" (8 bytov številka)
- ko prepišemo kanarčka, proces vidi, da je kanarček mrtev (tista številka ni taka kot bi morala biti) in se ubije
- lahko preskočiš kanarčka oz. ga prepišeš z isto vrednostjo in proces ne bo opazil, da si pisal tam



- če preskočimo kanarčka, lahko pišemo po return addressu in skočimo v neko funkcijo, za katero nismo avtorizirani

fgets:

- če vpišemo predolg string, ga odreže
- Windows cmd je bil včasih DOS operacijski sistem

Bash:

- včasih veliko bolj uporaben od visokonivojskih jezikov, ko delamo z datotekami
- /proc, psevdo datotečni sistem, vsaka mapa je PID, lahko spreminjamo informacije o procesu
- dvojni narekovaji delajo substitution za spremenljivke, enojni narekovaji pa ne

Assembly:

- pomemben, ker na koncu vse pride do tega

-
```c
#include <stdio.h>
#include <unistd.h>

int main()
{
        write(1, "Hello World!\n", 13);
        return 0;
}
```
-
```
Dump of assembler code for function main:
=> 0x0000555555555140 <+0>:     push   rbp
   0x0000555555555141 <+1>:     mov    rbp,rsp
   0x0000555555555144 <+4>:     sub    rsp,0x10
   0x0000555555555148 <+8>:     mov    DWORD PTR [rbp-0x4],0x0
   0x000055555555514f <+15>:    mov    edi,0x1
   0x0000555555555154 <+20>:    lea    rsi,[rip+0xea9]        #
0x555555556004
   0x000055555555515b <+27>:    mov    edx,0xd
   0x0000555555555160 <+32>:    call   0x555555555030 <write@plt>
   0x0000555555555165 <+37>:    xor    eax,eax
   0x0000555555555167 <+39>:    add    rsp,0x10
   0x000055555555516b <+43>:    pop    rbp
   0x000055555555516c <+44>:    ret
End of assembler dump.
```
- 0x10 = 16 bytov odštejemo, to je dovolj za dva pointerja
- edi = prvi argument
- rsi = drugi argument
- edx = tretji argument
- write je sistemski klic
- kako write() ve, kje dobiti argumente - vedno so na istih registrih
- https://x64.syscall.sh/
- e namest r pomeni, da imamo 32-bit, namesto 64
- na eax damo na koncu return value = 0 (xor s samim sabo nam da 0)