- bitovski operatori, njihove vrednosti, sve to prikazati na x i y celobrojnim promenljivim
- &, |, ^, ~, <<, >>
primer c fajl

- zasto se barata sa bitovima
- racunar komunicira sa perifernim uredjajima, mis, tastatura…, kroz kanale za komunikaciju, kroz koje se prenose bitovi
- hardver se pravi tako da bude sto jednostavniji za proizvodnju

Realna situacija u kojoj nam treba nekoliko razlicitih flagova koji oznacavaju odredjena svojstva:
- jedan nacin da se koriste razlicite celobrojne promenljive za zapis svakog flaga, svakog svojstva
- drugi nacin, posto nam je za svako svojstvo dovoljno samo 0 – nije aktivan, i 1 – jeste aktivan, odnosno dovoljan nam je jedan bit za jedno svojstvo…
Mozemo koristiti pojedinacne bitove u zapisu jedne celobrojne promenljive

- ako zelimo da se na tastaturi aktivira jedno dugme,
prvi bit – numlock
drugi bit – caps lock
treci bit – scroll lock

- da zapakujemo informacije u jedan bajt
- komunikacija sa jednostavnim hardverom

- simulacija skupova, prvi bit – prozor maksimizovan, drugi bit – prozor aktivan

- alternativa koriscenje celobrojnih promenljivih,

Kompilacija razlicitih fajlova:

biblioteka.c ----- .o fajl
main.c ----- .o fajl nakon toga linkovanje kao II korak

pozivi funkcija koje su definisani u ostalim programia se oznacavaju kao reference koje ce naci tek nakon linkovanja

i kreira se izvrsni fajl na kraju a.out

U njemu se nalaze odvojeni:
- podaci o globalnim promenljivim
- podaci o konstantama
- prostor za funkcije

a.out zovemo masinskim kodom, direktno se pokrece na procesoru
kada operativni sistem pokrece program postoji jos jedan korak linkovanja… i tu postoje neke nerazresene reference koje ce biti razresene prilikom samog pokretanja programa…
- neke greske se vide tek prilikom pokretanja programa
- debager gdb na vezbama – korak po korak se izvrsava program, prati se tok izvrsavanja programa, sve fje koje se povezuju, vide se vrednosti promenljivih

prevodi se program sa parametrom -g
dodaje dodatne informacije koje omogucavaju detaljnije pokretanja programa
veci i sporiji izvrsni fajl, to nije izvrsni fajl koji se salje dalje vec u fazi pisanja programa

- u memoriji se cuvaju i podaci i program
- pre izvrsavanja program se ucitava u RAM koji je organizovan u Fon Nojmanovoj arhitekturi:

- segment koda – izvrsni kod programa, nase fje koje su kompajlirane, instrukcije za procesor…
- ne menja se u toku izvrsavanja programa pa moze da se deli
- ako imamo vise instanci istog programa, i segment koda se deli izmedju tih instanci i sve instance pristupaju istom kodu

- segment podataka – globalne promenljive, konstante, staticke promenljive i konstante;
ima bar 2 dela, jedan deo prave promenljive (podaci koji mogu da se menjaju) i
---- ovi podaci se ne dele izmedju vise instanci istog programa (zato sto ti podaci mogu da se menjaju)
deo koji je Read only, konstante (menjanje ovih podataka zavrsava rad programa od strane OS)
---- ovi podaci mogu da se dele izmedju vise instanci istog programa

… ostatak je memorija koja se koristi u toku rada programa
… videli smo da promenljive mogu da imaju automatski zivotni vek i dinamicki zivotni vek (malloc)
- hip segment (dinamicki zivotni vek, malloc alocira memoriju i vrati pokazivac na memoriju… dinamicka alokacija memorije se radi kasnije;
alokacija je nasumicna kad god pozovemo malloc; free oslobadja tu memoriju i memorija postaje slobodna za novu alokaciju)
- stek segment

fja main poziva f1, poziva f2, poziva f3…
svaka fja treba da zna
kada se zavrsi fja f1, na neki nacin moramo da znamo koja je sledeca komanda …
ako imamo promenljivu x unutar main-a, i ako promenimo vrednost x unutar neke druge fje, promena lokalne promenljive ne menja vrednost spoljnog x

```
int main(){
int x…

f1(x)
}

int f1(int x){
int x… // promena ovog x ne menja x iz main-a
}
```

- svaka fja, svaki poziv fje, ima svoje promenljive i mora da ima mesto za cuvanje tih promenljivih.
- kad izadjemo iz fje promenljive koje su interno postojale u fji nestaju, zato sto su te promenljive automatskog zivotnog veka
- ti podaci mogu da budu izbrisani

- main → f1 → f2 → f3 → scanf
- kada f1 pozove f2, f1 se zaustavlja i poziva se f2,

- kada f2 zavrsi svoj poziv onda se nastavlja f1
- f2 poziva f3, f2 se zaustavlja dok se ne zavrsi fja f3, kada f3 zavrsi vraca se kontrola f2 i ona nastavlja sa svojim izvrsavanjem
- f3 poziva scanf, ucitava se vrednost, zavrsava scanf, vraca se kontorla f3 koja nastavlja da se izvrsava;
- kada f3 zavrsi, ona vraca kontrolu f2, kada ona zavrsi, vraca kontrolu f1 i nastavlja se tok
- kada f1 zavrsi vracamo se u main, nastavljamo dalje i main se zavrsava i program se zavrsava

- ako fja poziva drugu fju – ta druga ce biti kraceg zivotnog veka nego ova prva – pre ce se zavrsiti

za promenljive deklarisane u okviru fja odgovara nam da imamo Last In First Out, da poslednja stvar koja se ubacuje u strukturu bude prva koja se izbacuje

LIFO organizacija memorije za automatske lokalne promenljive unutar fja,
STEK struktura podataka i zato se zove STEK SEGMENT MEMORIJE

uvek alociramo sledeci blok memorije koji se oslobadja kad neka fja zavrsi...

...blokovi nisu obavezno sekvencijalno rasporedjeni, (termin virtuelna memorija – dodatno, omogucava deljenje koda izmedju razlicitih procesa)

scanf	Argument format stringa, &x,...
f3	
f2	X, y, z,...
f1	Argumenti i lokalne promenljive za datu fju: x, y
Stek segment za main fju	argc, argv, x, y

Kada scanf zavrsi, sve njegove promenljive se brisu i brise se taj deo memorije
- f3 nastavlja sa radom… kad se zavrsi f3, ----|| ----
- f2 …
- f1 …

- svaki od ovih blokova (stek okvira) je potpuno nezavisan od prethodnih, odvojene su potpuno promenljive
- stek okvir – blok memorije koji se rezervise za svaki poziv fje tokom izvrsavanja te fje
- alocira se prilikom poziva fje
- oslobadja se kada se fja zavrsi

- sam stek segment je ogranicen prostor, postoji gornji limit i moguce je da se popuni
- tada OS gasi program i ispisace poruku “Stack overflow”
- ovo se desava definisanjem prevelikog niza u nekoj fji
- greska se nalazi tako sto se proveri velicina nizova

- ili greske kod rekurzije – nema izlaza iz rekurzije i beskonacni niz poziva rekurzivnih funkcija (na narednom casu)

- ako fja ima komplikovan izraz u return naredbi:
return (x*3 + y*2) – (4*z + 7*w)…
\ / \ /
. .
\ /
.

kreiraju se privremene vrednosti za medjurezultate drveta izraza izracunavanja
sve ove tackice se cuvaju u stek okviru ove fje
dok ne izracunamo zavrsni rezultat koji se vraca sa return

- prilikom poziva f1 → f2 → f3...
prilikom poziva f3 moramo da znamo gde u f2 treba da se vrati, - na koju adresu, tj. Na koju instrukciju se vraca
moguce je da ima vise poziva fje f3… i ta adresa povratka se razlikuje
-
- moramo da znamo koliki je bio stek okvir poslednje fje, pa mozemo da pamtimo gde stek okvir pocinje
- pamtimo adresu pocetka stek okvira

- kada se fja zavrsi automatski ce se kontrola vratiti u roditeljsku fju
- pokazivac na trenutni stek okvir se vraaca na adresu prethodnog stek okvira

- vracamo se u roditeljsku fju koja nastavlja da se izvrsava

- sam stek okvir za zavrsenu fju se ne brise eksplicitno, vec on vise nije aktuelan
- ne moramo da brisemo
- kad se pozove neka nova fja, pravi se novi stek okvir i dolazi preko starog stek okvira i ta memorija ce biti prezapisana

- ovo se cesto desava kada se memorija oslobadja – evidentireno je da ta memorija moze da se dodeli ponovo ali bez brisanja podataka

NEZELJENI SPOREDNI EFEKTI
- posledica toga moze da bude da se u uzastopnim pozivima fje f1, promenljiva koja je neinicijalizovana u kodu moze narednim komandama promeniti vrednost
i u drugom pozivu neinicijalizovana promenljiva ce imati vrednost koja je dodeljena u prvom pozivu
- otuda ce se razlikovati pozivi
- cinjenica da se memorija ne prazni kada se pravi novi stek okvir moze dovesti do uzimanja starih podataka
→ obavezno da kada se promenljiva deklarise, da dobije i pocetnu vrednost
- cak i za staticke promenljive i cak i za globalne promenljive (koje se inicijalizuju na 0)
- uvek sve treba inicijalizovati