Vežba 2 : GCC i Make

GCC

Originalni GNU C Compiler (GCC) je razvijen od strane Ričarda Stolmana (*Richard Stallman*), osnivača GNU projekta i pored Linusa Torvaldsa, začetnika Linux operativnog sistema. 1984 godine GNU projekat je kreiran kako bi omogućio Unix-like operativni sistem kao besplatan softver (free software), u cilju promovisanja slobode programiranja. Vremenom je GCC obuhvatio podršku za druge programske jezike (C++, Objective-C, Java, Fortrana, Ada) tako da GCC danas znači GNU Compiler Collection. Instalirana verzija GCC alata se može proveriti komandom gcc --version.

GNU Toolchain sadrži:

- GCC kompajler (osnovna komponenta)
- GNU Make (automatizovan alat za kompajliranje i build-ovanje aplikacija
- GNU Binutils (binarni alati uključujući linker i asembler)
- GNU Debugger (GDB)
- GNU Autotools
- GNU Bison

GCC je portabilan i može se izvršavati na mnogim operativnim sistemima. Predstavlja prirodno okruženje za Linux i Unix-like OS. Na Windows operativnim sistemima se može pronaći u sklopu MinGW i Cygwin programa. Takođe, postoji i cross-kompajler koji omogućava pravljenje izvršnih fajlova za različite platforme (između ostalog ARM uređaje). GCC je kompajler za C, dok se kompajler za C++ naziva G++.

Ukoliko se napravi fajl po imenu *hello.c* sa sledećim sadržajem, on se može kompajlirati pomoću naredbe: *gcc hello.c*

```
#include <stdio.h>
int main()
{
        printf("Hello, world!\n");
        return 0;
 }
```

U tom slučaju rezultat kompajliranja će biti *a.out* binarni (izvršni) fajl. GCC predpostavlja da je cilj kompajliranja izvršni fajl te ga pravi sa generčkim imenom a.out. Izvršni fajlovi se u Linux-u pokreću pomoću komande *./ime_binarnog_fajla*. Ovo praktično znači: pozovi fajl sa nazivom *ime_binarnog_fajla* iz trenutnog direktorijuma. Moguće je pokrenuti program samo sa *ime_binarnog_fajla* ukoliko se putanja fajla nalazi u PATH environment varijabli. Ukoliko korisnik želi dodeliti alternativno ime izvršnom fajlu, to se omogućava pomoću parametra **-o** (output). Naredba: *gcc -o hello hello.c* kompajlira *hello.c* fajl, ali u ovom slučaju je parametrom output naznačeno da je cilj kompajliranja izvršni fajl po imenu *hello*.

Pri kompajliranju fajlova je moguće odrediti dodatne opcije kao što su:

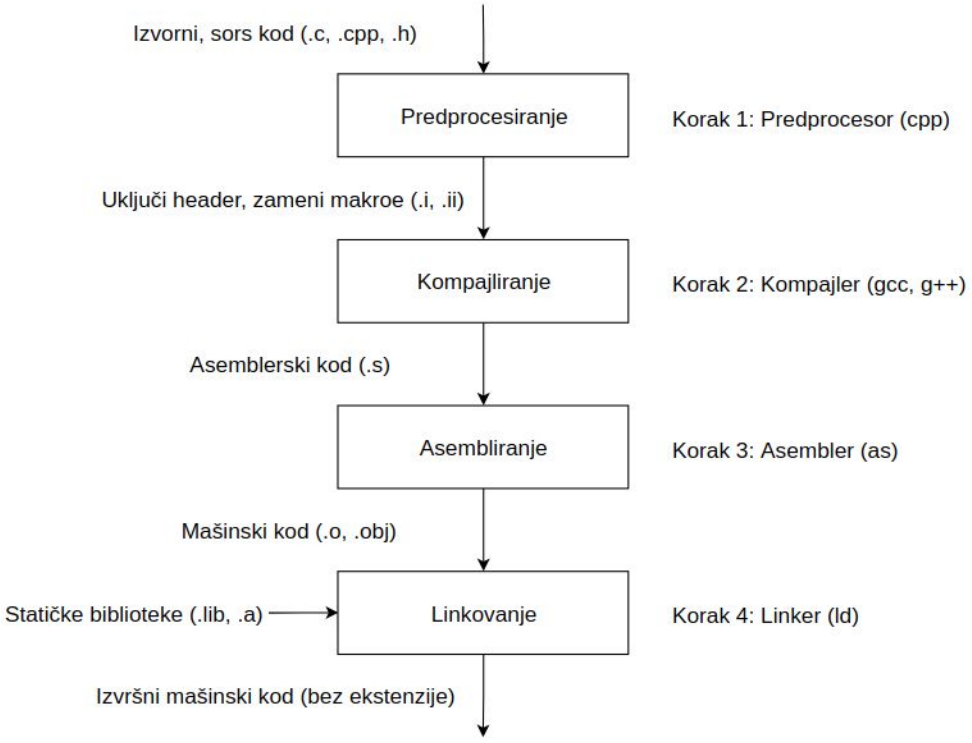
-Wall prikazuje sve Warning poruke

-g generiše simbole potrebne za debagovanje korišćenjem GDB programa.

Moguće je posebno kompajlirati i povezivati izvršni fajl:

- gcc -c hello.c //kao rezultat se dobija hello.o
- gcc -o hello hello.o //kao rezultat se dobija hello

GCC proces kompajliranja se sastoji iz više koraka koji su prikazani na slici 1.



Slika 1: Proces kompajliranja C programa

Svaki od prethodno navedenih koraka se može izvršiti zasebno:

- Pozivanje predprocesora koji izvršava uključivanje .h fajlova u .c fajlove i vrši ekspanziju makroa:
  - **cpp hello.c > hello.i**
- Kompajliranje, rezultat je asemblerski kod za dati procesor:
  - **gcc -S hello.i**
- Asembliranje, tj. Konvertovanje asemblerskog koda u mašinski kod u formi objektnog fajla:
  - **as -o hello.o hello.s**
- Povezivanje sa bibliotekama
  - **ld -o hello hello.o ...biblioteke…**

Header fajlovi i biblioteke

Biblioteka je kolekcija pred-kompajliranih objektnih fajlova koji mogu da se povezuju sa izvršnim fajlom pomoću linker-a (npr printf(), sqrt()). Postoje dva tipa biblioteka: statičke i deljene.

Statičke biblioteke imaju ekstenziju .a (Linux) odnosno .lib (Windows) i prilikom povezivanja sa objektnim fajlom, kod iz biblioteke se kopira u izvršni fajl. Ovakav tip biblioteke se može kreirati korišćenjem programa **ar**.

Deljene biblioteke imaju ekstenziju .so (Linux) odnosno .dll (Windows). Kada se program povezuje sa ovakvom bibliotekom, samo se mala tebela kreira u izvršnom fajlu. Pre nego što se krene sa izvršavanjem, operativni sistem učitava mašinski kod zahtevan za eksterne funkcije i ovaj proces je poznat kao dinamičko povezivanje (dynamic linking). Na ovaj način se dobijaju manji izvršni fajlovi i čuva se prostor na disku jer se deljene biblioteke dele između više izvornih fajlova. Osim toga, nadogradnja sistema je moguća bez ponovnog rekompajliranja izvornog koda (samo je potrebno zameniti deljenu biblioteku).

Prilikom kompajliranja, kompajleru su neophodni header fajlovi.Za svaki .h fajl korišćen od strane .c fajlova (korišćenjem #include direktive) kompajler traži include putanju. Ove putanje se specificiraju korišćenjem –Idir opcije ili environment varijable CPATH. Pošto su imena .h fajlova poznata, putanja je dovoljna.

Linker-u su potrebne biblioteke da bi razrešio eksterne reference na druge objektne fajlove ili biblioteke.Linker traži tzv. putanju biblioteke za sve biblioteke koje su potrebne kako bi se kreirao izvršni fajl. Ova putanja se specificira sa –Ldir opcijom ili environment varijablom LIBRARY_PATH. Dodatno, potrebno je tačno navesi i ime biblioteke: Na Unix –like i Linux operativnim sistemima za biblioteku libxxx.a

potrebno je navesti –lxxx opciju. U Windows operativnom sistemu, navodi se celo ime kao lxxx.lib

GCC Environment varijable:

- PATH: u njoj se traže izvršni fajlovi kao i deljene biblioteke (.dll, .so).
- CPATH: služi za pretraživanje include putanja za header fajlove. Ovaj direktorijum se pretražuje nakon direktorijuma specificiranog sa –Idir opcijom. Dodatno, C_INCLUDE_PATH i CPLUS_INCLUDE_PATH se mogu koristiti da se specificiraju C/C++ header fajlovi ukoliko je programski jezik naznačen tokom pred-procesiranja.
- LIBRARY_PATH: U njemu se traže biblioteke, nakon što se pretraže putanje specificirane sa –Ldir opcijom.

Dodatni GCC alati:

- “file“ služi za prikaz tipa objektnog ili izvršnog fajla> file hello.o> hello.o: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (using shared libs), for GNU/Linux 2.6.32
- “nm” služi za izlistavanje tabele simbola u okviru objektnih fajlova. Obično se koristi da bi se utvrdilo da li je neka funkcija definisana u objektnom fajlu. Slovo T u drugoj koloni označava da je funkcija definisana, dok U označava da nije i da mora biti razrešena od strane linkera:
  > nm hello.o
  > 0000000000000000 T main
  U puts
- “ldd“ proverava izvršni fajl i prikazuje sve deljene biblioteke koje su neophodne > ldd hello
  > linux-vdso.so.1 => (0x00007ffdcad43000)
  libc.so.6=>/lib/x86_64-linux-gnu/libc.so.6 (0x00007fddb6b1a000)
  /lib64/ld-linux-x86-64.so.2 (0x00007fddb6ee4000)

GNU Make

Komande za kompajliranje i linkovanje C programa mogu biti kompleksne i mnogobrojne. Alat “make” automatizuje složene aspekte kreiranja izvršnog fajla od izvornog koda. Ovaj alat koristi fajl po imenu “makefile” koji sadrži pravila kako se kreiraju izvršni fajlovi krenuvši od izvornih fajlova (.c,.cpp,.h). Pravila se pišu u unapred definisanom formatu u makefile-u, nakon čeka se proces kompajliranja pokreće iz terminala komandom “make”. Makefile mora imati jedan od sledeća tri imena kako bi bio prepoznat od strane make alata: makefile, Makefile ili GNUmakefile.

Postoje knjige napisane o “make”-u, a ovde ćemo samo pomenuti osnovna pravila i elementarno korišćenje. Dokumentaciju o “make”-u je moguće dobiti pomoću naredbi “make -help” ili “man make” koji izlistava tzv man pages.

Makefile se sastoji od više “pravila” koja imaju isti format:

odredište: preduslov1 preduslov2 ...
    komanda
    komanda

Pravilo se sastoji od tri dela:

- Odredište (target)
- Lista preduslova (pre-requisites list)
- Komande (commands)

Odredište i lista preduslova su odvojeni dvotačkom, a sve komande počinju tabom.

Takođe bitno je napomenuti da *makefile* nisu samo skriptovi koje treba izvršiti jedan za drugim. Alatka make prvo analizira čitavu datoteku makefile da bi napravila stablo zavisnosti mogućih odredišta i njihovih preduslova, potom se iteracijama kreće kroz to stablo da bi napravila potrebne datoteke.

Za jednostavan “Hello World” C program koji se nalazi u *hello.c* fajlu i samo ispisuje poruku u terminalu, makefile bi mogao izgledati kao sledeći:

```
all : hello
hello : hello.c
    gcc -o hello hello.c
clean :
    rm hello
```

Svako od navedenih pravila definiše pomoću kojih komandi je moguće dobiti odredište (target) koristeći preduslove (pre-requisites). Na primer, prethodni makefile govori da se “hello” izvršni fajl može dobiti od izvornog “hello.c” fajla, pokretanjem naredbe “gcc -o hello hello.c”. Pokretanje komande “make” bez argumenata pokreće prvo pravilo na koje se naiđe. U primeru navedenom ranije, prvo pravilo na koje se naiđe je “all” koje kao preduslov ima “hello”. Make ne može da pronađe fajl “hello” (barem u prvom pozivu), tako da traži odgovarajuće pravilo kako bi ga kreirao. To pravilo je “hello” koje ima preduslov “hello.c”, koji postoji u direktorijumu. Pokreće se komanda “gcc -o hello hello.c” U ovom trenutku se dobije “hello” izvršni fajl koji je preduslov za pravilo “all”, te se sada ono može pokrenuti. Budući da ispod ovog pravila ne postoji nijedna komanda, ovo pravilo ne radi ništa. Time se izvršavanje programa “make” završava.

Ukoliko odredišni fajl u nekom pravilu postoji, komanda će se pokrenuti samo u slučaju ako je preduslov noviji od odredišta (obratite pažnju da su i jedan i drugi zapravo fajlovi). Drugim rečima, komanda će se pokrenuti samo ako je neki od preduslova menjan od poslednjeg pokretanja make komande. U suprotnom slučaju, nema smisla pokretati naredbu ponovno. Kao povrda toga, ako ponovo pokrenemo “make” komandu dobijamo sledeći ispis:

make: Nothing to be done for “all”

Prilikom pozivanja make komande, moguće je specificirati odredište, npr “make clean” uklanja fajl hello. Ukoliko se pokrene “make” bez odredišta pozvaće se prvo odredište na koje se naiđe: “make all” je isto kao i “make” u primeru od gore

Ukoliko komanda ne počinje za karakterom TAB, u terminalu se dobija poruka:

*“makefile:4: *** missing separator.Stop.”*

Ukoliko ne postoji makefile u trenutnom direktorijum dobija se greska:

*“make:*** No targets specified and no makefile found. Stop.”*

Malo detaljniji makefile za prethodni slučaj bi mogao izgledati ovako:

```
all: hello
hello: hello.o
	gcc -o hello hello.o
hello.o: hello.c
	gcc -c hello.c
clean:
	rm hello.o hello
```

Sada će se zasebno kompajlirati i linkovati program. Prvo pravilo na koje se nailazi je “all:hello”, i budući da nema “hello” izvršnog fajla, traži se pravilo za kreiranje njega. Pravilo “hello” ima preduslov “hello.o” i pošto ni taj fajl ne postoji, na sličan način se traži pravilo za njega. Pravilo za “hello.o” ima preduslov “hello.c”, koji postoji u direktorijumu. Pokreće se komanda “gcc -c hello.c”. Rekurzivno unazad, pravilo “hello” će dalje da pokrene komandu “gcc -o hello hello.o”, dok pravilo “all” ne radi zapravo ništa. U ovom će poziv “make clean” naredbe izbrisati i objektni fajl “hello.o”. Makefile može da sadrži komentare koji počinju znakom “**#**” i važe do kraja reda u kome su napisani. Dugačke linije mogu biti “prelomljene” korišćenjem kose crte (backslash) ”\“. Pravila su obično organizovana tako da generalnija pravila idu prva: npr. prvo pravilo u primeru od malopre je pravilo “all” koje rekurzivno poziva sva ostala pravila.

Tzv lažna odredišta (phony targets) su odredišta koja ne predstavljaju fajlove. Na primeru od malopre “clean” bi bio jedan primer takvog lažnog odredišta. Ukoliko je odredište fajl, biće proverena zastarelost njegovih preduslova, dok se lažna pravila uvek izvršavaju. Standardna lažna pravila su “all”, “clean” i “install”. Kako bi se naznačilo da je neko odredište lažno koristi se oznaka “.PHONY : odredište” neposredno pre pravila. Preporučuje se da se lažna odredišta uvek naznače, te prethodni primer bi trebao da izgleda:

```
.PHONY: all
all: hello
hello: hello.o
	gcc -o hello hello.o
hello.o: hello.c
	gcc -c hello.c
.PHONY: clean
clean:
	rm hello.o hello
```

Primer 1:

Dat je jednostavan program pomoću tri fajla:

1. **calculate.c**

```
#include "helper.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main (int argc,char** argv)
{
        int a = atoi(argv[1]);
        int b = atoi(argv[2]);
        int res = pomnozi(a,b);
        printf("Rezultat mnozenja je: %d\n",res);
}
```

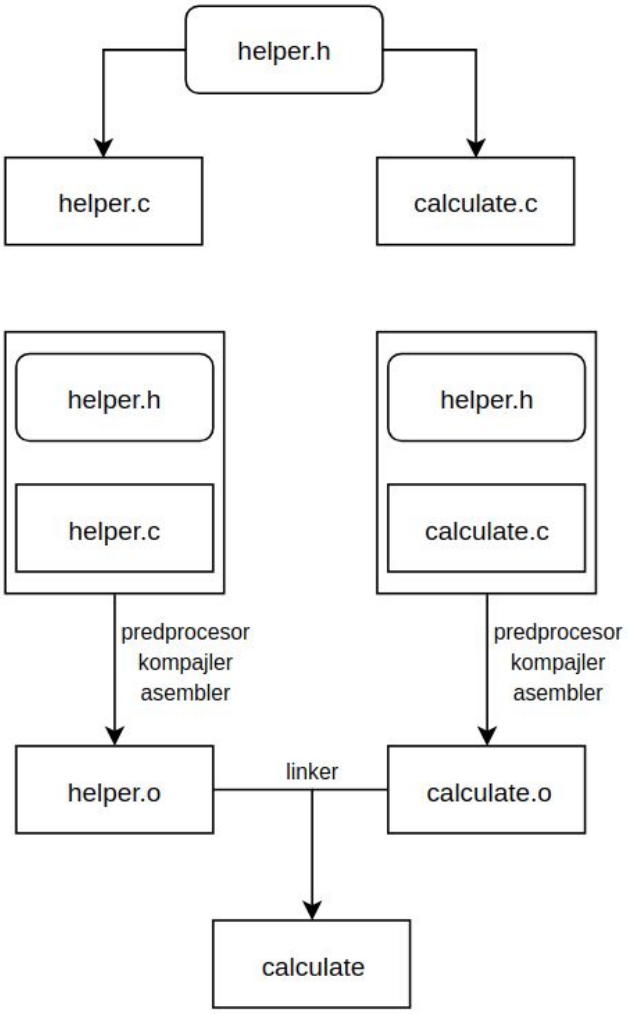
2. **helper.h**

```
#ifndef HELPER_H
#define HELPER_H
extern int pomnozi(int, int);
#endif
```

3. **helper.c**

```
#include "helper.h"
int pomnozi(int a, int b)
{
        return b*a;
}
```

U fajlu calculate.c se nalazi glavna(main) funkcija koja prima parametre prilikom njenog poziva. Parametri se parsiraju u integer brojeve a zatim se nad njima poziva funkcija pomnozi() i rezultat se ispisuje u terminalu. Funkcija ima svoju deklaraciju u helper.h fajlu, a svoju definiciju u helper.c fajlu. Potrebno je napraviti makefile koji kompajlira prethodni program u objektne fajlove a zatim ih linkuje u izvršni. Ukoliko se neki od izvornih fajlova promeni, makefile treba da rekompajlira sve fajlove na koje on utiče. Proces kompajliranja u ovom slučaju je prikazan na sledećoj slici:



Slika 2. Zavisnosti među fajlovima

Fajlovi helper.c i calculate.c uključuju helper.h fajl, pa će stoga objektni fajl calculate.o zavisiti od calculate.c i helper.h fajlova, a helper.o zavisiti od helper.c i helper.h fajlova. Izvršni fajl calculate zavisi od oba objektna fajla. Shodno ovome se piše makefile:

```
all: calculate
calculate: calculate.o helper.o
	gcc -o calculate calculate.o helper.o
calculate.o: calculate.c helper.h
	gcc -c calculate.c
helper.o: helper.c helper.h
	gcc -c helper.c
clean:
	rm -f calculate *.o
```

Zadatak 1:

Dat je program sa sledećim izvornim fajlovima:

1. **calculate.c**

```
#include "helper.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main (int argc,char** argv){
        int a = atoi(argv[1]);
        int b = atoi(argv[2]);
        int res = pomnozi(a,b);
        printf("Rezultat mnozenja je: %d\n",res);
}
```

2. **helper.h**

```
#ifndef HELPER_H
#define HELPER_H
extern int pomnozi(int, int);
#endif
```

3. **helper.c**

```
#include "helper.h"
#include "helper2.h"
int pomnozi(int a, int b){
        return pomnozi_sabiranjem(a, b);
}
```

4. **helper2.h**

```
#ifndef HELPER2_H
#define HELPER2_H
extern int pomnozi_sabiranjem(int,int);
extern int pomnozi_normalno(int,int);
#endif
```

5. **helper2.c**

```
#include "helper2.h"
#include <string.h>
#include <stdio.h>
int pomnozi_normalno(int a, int b){

        printf("Pozvana funkcija pomnozi_normalno\n");
        return a*b;
}
int pomnozi_sabiranjem(int a, int b){
        printf("Pozvana funkcija pomnozi_sabiranjem\n");
        int i;
        int result;
        for (i=0; i<a; i++)
                result += b;
        return result;
}
```

Nacrtati grafik zavisnosti fajlova a zatim napisati odgovarajući Makefile.

Promenljive

Sve promenljive unutar makefile-a su istog tipa: one sadrže sekvence karaktera, nikada numeričke vrednosti. Kada god make pokrene neko pravilo, on evaluira sve promenljive koje se nalaze u odredištu, preduslovima i komandama. Postoje dva tipa promenljivih : rekurzivno proširene i jednostavno proširene. Pri definiciji promenljive, operator dodele vrednosti definiše koji će se tip promenljive koristiti (= za rekurzivne, := za jednostvane). Vrednosti neke promenljive se pristupa pomoću operatora $(ime_promenljive). Ukoliko je ime promenljive jedno slovo ili znak, nema potrebe za zagradama.

- Kod rekurzivno proširenih promenljivih se ugneždene reference na promenljive čuvaju pomoću njihovih naziva sve dok se promenljiva ne evaluira kada se zamenjuju vrednostima. Primer:
  DEBUGFLAGS = $(CFLAGS) -ggdb
  U prethodnoj liniji vrednost $(CFLAGS) se neće evaluirati sve dok se ne iskoristi $(DEBUGFLAGS).

- Jednostvano proširene promenljive se zamenjuju vrednostima odmah pri definicji promenljive, te se takve vrednosti čuvaju do njihovog korišćenja. Primer:
  OBJ = hello.o
  TESTOBJ := $(OBJ) profile.o
  OBJ = hi.o
  U ovom slučaju će se pri definiciji TESTOBJ promenljive $(OBJ) odmah zameniti sa hello.o, te redefinisanje OBJ promenljive u trećoj liniji neće imati uticaja na vrednost $(TESTOBJ).

Sem operatora **=** i **:=** često se koristi i **+=** koji konkatanira string na promenljivu.
Postoje već definisane (automatske) promenljive koje imaju različite vrednosti u zavisnosti u kojem se pravilu koriste:

$@	ime fajla odredišta
$*	ime fajla odredišta bez ekstenzije
$<	ime fajla prvog preduslova
$^	imena svih preduslova,razdvojenih bez dupliranja
$+	isto kao prethodno, ali sa dupliranjima
$?	imena svih preduslova novijih od odredišta, razdvojenih space-om

Ako pogledamo jednostavan makefile koji smo napravili za “Helmlo World” program, koristeći automatske promenljive možemo napraviti ekvivalentan makefile:

```
.PHONY: all
all: hello
hello: hello.o
	gcc -o $@ $<
hello.o: hello.c
	gcc -c $<
.PHONY: clean
clean:
	rm hello.o hello
```

Šablonska pravila

Šablonsko pravilo, koje koristi karakter preklapanja “%” umesto imena fajla, može se primeniti kako bi se kreiralo odredište, ukoliko nema eksplicitnog pravila. Pravila koja se definišu na ovakav način se zovu implicitna pravila. Pravilo u primeru ispod govori: ukoliko je potrebano napraviti objektni fajl “ime.o” napravi ga iz istoimenog C fajla “ime.c” pomoću komande “gcc -c ime.c”.

```
%.o: %.c
	gcc -c $<
```

“Hello World” primer makefile-a bi mogli predstaviti pomoću šablonskih pravila na sledeći način:

```
.PHONY: all
all: hello
%: %.o
	gcc -o $@ $<
%.o: %.c
	gcc -c $<
.PHONY: clean
clean:
	rm hello.o hello
```

Virtuelna putanja

Moguće je koristiti VPATH kako bi se specificirao direktorijum u kome se mogu tražiti preduslovi i odredišta (fajlovi). Na primer:

```
VPATH = src include
```

Takođe, moguće je koristiti vpath kako bi se specificirao tip fajla koji se traži u direktorijumu. Na primer:

```
vpath %.c src
vpath %.h include
```

Prva linija govori da se ".c" fajlovi traže u direktorijumu "src", a ".h" fajlovi u direktorijumu "include".

Zavisnosti

Zavisnosti su pravila koja ne definišu komande već samo zavisnosti među fajlovima. Kada napravi ovakvo pravilo očekuje se da postoji dodatno pravilo koje definiše komandu koja govori kako se taj fajl dobija. Primer:

```
Main.o : Main.c Test1.h Test2.h
Test1.o : Test1.c Test1.h
Test2.o : Test2.c Test2.h
```

Zaključak

Sve što je pomenuto prethodno je samo deo prave moći koju poseduje Make alat, a samim tim i težine razumevanja istog. Razumevanje prethodnih mogućnosti je dovoljno da se naprave Makefile-ovi za ovaj kurs. Za kraj je dat primer makefile-a koji koristi sve prethodne opisane tehnike kao i nekoliko bash alata. Ovaj makefile sam pronalazi zavisnosti među objektnim i izvornim fajlovima, te pri promeni nekog od fajlova rekompajlira samo neophodne.

Prve dve linije makefile-a definišu dve rekurzivne promenljive. Prva promenljiva "sources" koristi ugrađenu funkciju "wildcard" da u trenutnom direktorijumu pronađe sve fajlove sa ekstenzijom ".c". Imena ovih fajlova će biti sačuvana u promenljivoj u formatu liste. Promenljiva "objs" uzima ovu listu sa imenima C fajlova i ekstenzije ".c" zamenjuje ekstenzijama ".o". Time promenljiva "objs" predstavlja listu svih objektnih fajlova u projektu. Poslednja promenljiva "result" čuva ime izvršnog fajla

koji je cilj kompajliranja. Lažno odredište “all” kao preduslov daje izvršni fajl $(result) koji postaje cilj kompajliranja. Pravilo “$(result): $(objs)” govori da se izvršni fajl dobija linkovanjem svih objektnih fajlova u projektu. Sledeće pravilo “%.o: %.c” je šablonsko: ukoliko je potreban neki objektni fajl, njega je potrebno kompajlirati iz istoimenog C fajla. Poslednje šablonsko pravilo “%.d: %.c” definiše na koji se način prave tzv dependency fajlovi, u kojima će se u čuvati zavisnosti između objektnih i izvornih fajlova. Budući da se pri kompajliranju iz svakog C fajla generiše jedan objektni fajl, svaki C fajl će sebi imati pridružen fajl zavisnosti sa ekstenzijom “.d”. Dependency fajl se pravi tako što se “.c” fajl predprocesira sa dodatnim ispisivanjem poruka o zavisnosti fajlova, u formatu opisanom u prethodnom poglavlju (Test1.o : Test1.c Test1.h). Ovo se radi komandom “ $(CC) -MM $(CPPFLAGS) $<” nakon čega se iskoristi operator preusmeravanja stringa “>” u istoimeni dependency fajl “$@”. Pri svakom pokretanju makefile-a naredba “-include $(sources:.c=.d)” učitava zavisnosti iz dependency fajlova. Ukoliko dependency fajl ne postoji, on će biti generisan pomoću prethodno opisanog pravila “%.d: %.c”. Ukoliko se neki C fajl promenio, moguće je da su se promenile predprocesorske “include” direktive, pa će se njegov dependency fajl ponovo generisati. Poslednje pravilo je “clean” koje uklanja izvršni fajl $(result), a potom i sve objektne i dependency fajlove iz trenutnog direktorijuma. Svaki put kada se pokrene neka komanda, u terminalu će se ispisati odgovarajuća poruka pomoću echo komande.

```
sources=$(wildcard *.c)
objs=$(sources:.c=.o)
result=hello
all: $(result)
$(result): $(objs)
	@echo -n "Building output binary: "
	@echo $@
	$(CC) -o $@ $(objs)
%.o: %.c
	@echo -n "Compiling source into: "
	@echo $@
	$(CC) -c $<
%.d: %.c
	@echo -n  "Creating dependancy: "
	@echo $@
	@rm -f $@; $(CC) -MM $(CPPFLAGS) $< > $@;
-include $(sources:.c=.d)
.PHONY: clean
clean:
	@rm -rf $(result) *.o *.d
	@echo "Clean done.."
```