Programiranje 2

Laboratorijske vježbe

LV5
Organizacija izvornog kôd-a

Fakultet elektrotehnike računarstva i informacijskih tehnologija Osijek
Kneza Trpimira 2b
www.ferit.unios.hr

Uvod

Kako programi počinju rasti do nekoliko tisuća ili desetak tisuća linija kôda i više, takve programe je potrebno rastaviti na više datoteka radi bolje organizacije izvornog kôda.

Postoji mnogo razloga zbog čega je praktično podijeliti izvorni kôd na više datoteka, a neki od razloga su sljedeći:

- Ako se u izvornom kôdu koji ima npr. duljinu deset tisuća linija kôd-a promijeni jedna linija, potrebno je prevesti cjelokupni izvorni kôd. Proces prevođenja u nekim slučajevima može potrajati nekoliko sati. Upravo zbog navedene situacije, ako je izvorni kôd raspodijeljen u više datoteka potrebno je kompilirati samo datoteku u kojoj je napravljena promjena.
- Raspoređivanje izvornog kôda u više datoteka doprinosi preglednosti cjelokupno kôda tako što se u različite datoteke smjeste deklaracije i definicije funkcija, konstante, definicije složenih tipova podataka.... Ovim načinom se pospješuje snalaženje, pronalaženje i dorađivanje određenog dijela kôd-a.
- U slučaju gdje više ljudi radi na jednom programu, daleko je praktičnije da svaki programer piše programski kôd u vlastitoj datoteci.
- Ako je programski kôd pažljivo podijeljen u datoteke, tada se pojedini dijelovi kôda mogu iskoristiti u drugim projektima. Također se pojedini dijelovi kôd-a mogu dijeliti tijekom izrade više paralelnih projekata i time ako se napravi promjena u nekoj datoteci, taj kôd će biti ažuriran u svim projektima.

Izvorni kôd koji je podijeljen u više datoteka mora biti jednako konzistentan kao kada je cijeli program napisan u jednoj datoteci.

Faze dobivanja izvršne datoteke - Prevođenje

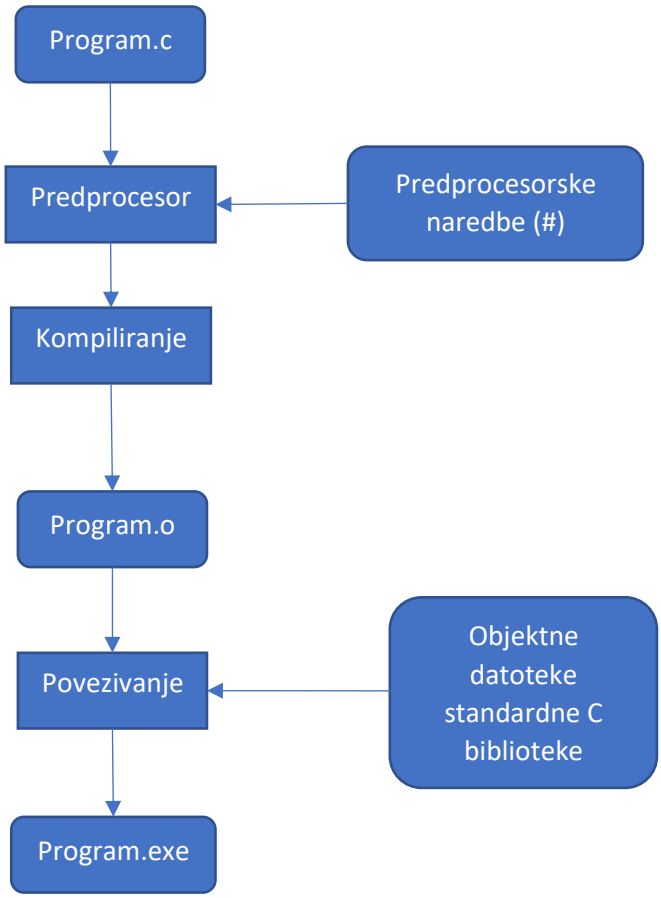
Kako bi se dobila izvršna datoteka potrebno je proći kroz više koraka:

- Predprocesor (engl. *preprocessor*),
- Kompiliranje (engl. *compiling*),
- Povezivanje (engl. *linking*).

Prvi razlog razdvajanja faza prevođenja C kôda kojima se dobiva izvršna datoteka je zbog toga što je lakše implementirati različite poslove u specifične okvire i time smanjiti kompleksnost samog programa za izradu izvršne datoteke.

Drugi razlog, ako se jednom prođe kroz proces prevođenja, nije potrebno prilikom promjene jedne datoteke ponovno sve kompilirati, već je potrebno kompilirati samo datoteku u kojoj se napravila promjena. Kompiliranjem se dobivanju objektne datoteke koje će služiti povezivaču za povezivanje u izvršnu datoteku.

Ovim načinom se omogućava povezivaču jednostavno povezivanje kompiliranih datoteka izvornog kôda koje su dio standardne biblioteke. Njihove objektne datoteke će poslužiti tijekom povezivanja s objektnim datotekama dobivenim prevođenjem korisnički kreiranih datoteka izvornog kôda. Proces dobivanja izvršne datoteke iz izvorne datoteke prikazan je na slici 1.



Slika 1. *Proces prevođenja izvorne datoteke*.

Razdvajanje između procesa kompiliranja i povezivanja pojednostavljuje otkrivanje pogrešaka u kodu (engl. *bug*). Poruke pogreške tijekom kompiliranja su obično pogreške u sintaksi, npr. nezatvorena zagrada, ili nenavedena deklaracija funkcije prije poziva funkcije (engl. *compile-time errors*). Pogreške tijekom povezivanja su obično navedene o nedostatku definicije funkcije (engl. *link-time errors*). Čak kada se uspješno poveže objektni kôd i dobije se izvršna datoteka može se dogoditi da program dobro radi s nekim podacima, a s drugima se ponaša nepredvidivo, u tome slučaju radi se o pogreškama pri izvođenju (engl. *run-time errors*).

Kako bi se lakše rukovalo s više datoteka koje se kreiraju tijekom izrade programa koriste se integrirana razvojna okruženja (engl. *Integrated development environment*), tzv. *IDE* s kojim se najčešće isporučuje kompilator i povezivač, npr. *Visual Studio*, *Eclipse*, *Netbeans*, *CodeBlocks*.... Integrirano razvojno okruženje izradu programa smješta u projekt, te se brine o preglednom razmještaju datoteka zaglavlja, izvornog kôd-a, objektnih datoteka, te svih pomoćnih datoteka prilikom procesa kompiliranja i povezivanja. Također IDE se brine o vremenskom periodu (engl. *timestamp*) svake datoteke izvornog kôd-a koja se prevodi i njezine objektne datoteke. U slučaju izmjene određene datoteke izvornog kôd-a u projektu prevele bi se one datoteke izvornog kôd-a koje imaju noviji vremenski period od njezinih objektnih datoteka.

Predprocesor

Ako se u programu nalaze predprocesorske naredbe koje počinju sa znakom (#), kao što je *#include* kojom se uključuje datoteka zaglavlja standardnih i/ili korisnički definiranih datoteka, *#define* kojom se uključuju makro definicije/funkcija, tada se uvodi još jedan korak u stvaranju izvršne datoteke. Predprocesor stvara privremenu datoteku u koju se kopira sav kôd iz uključene datoteke zaglavlja ili se obavlja zamjena svih makro definicija. Tako stvorena datoteka izvornog kôd-a predstavlja čisti C kôd koji se prosljeđuje kompilatoru na daljnju obradu.

Kompiliranje

Kompiliranje se odnosi na procesiranje izvornog kôd-a s ekstenzijom (.c ili .cpp), te kreiranje objektne datoteke s ekstenzijom (.o ili .obj). Ako se kompilira više datoteka izvornog kôd-a, dobit će se jednako toliko objektnih datoteka, gdje je važno naglasiti kako se kompilira svaka datoteka zasebno. Objektne datoteke su zapravo dobivene instrukcije u strojnom jeziku koje su prevedene iz izvornog kôd-a, ali u obliku koji se još uvijek ne može pokrenuti. U slučaju korištenja *IDE* kao što je *Visual Studio*, cijeli ovaj proces je skriven od korisnika. Prilikom kompiliranja provodi se sintaksna provjera, te se provjerava odgovara li poziv funkcija njihovim deklaracijama. Svaka datoteka mora sadržavati deklaracije funkcija koje se unutar te datoteke pozivaju. Prevoditelju je dovoljno poznavanje naziva funkcije i koje parametre prima, a kako prevoditelj "vidi" samo datoteku koju prevodi ne može usporediti deklaraciju funkcije sa stvarnom definicijom funkcije koja je u odvojenoj datoteci.

Povezivanje

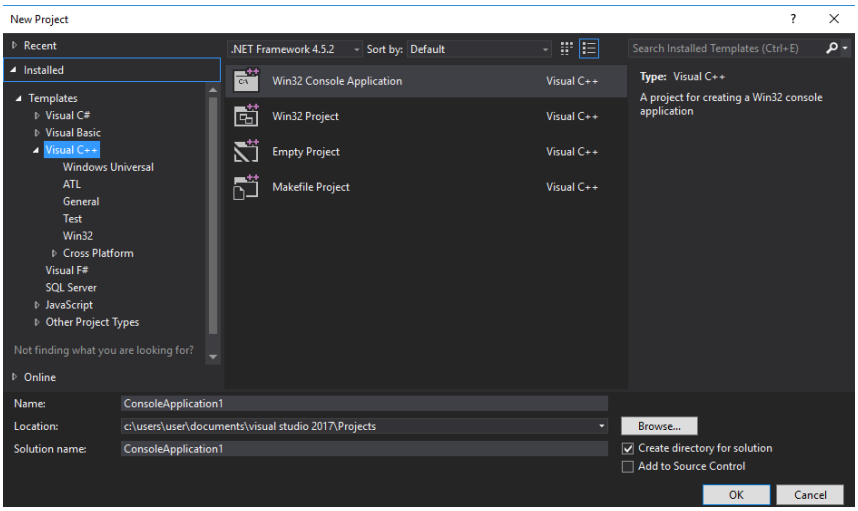
Tek nakon što se isprave sve pogreške uočene prilikom kompiliranja i kôd se uspješno kompilirao može se pristupiti povezivanju. Povezivanje se odnosi na kreiranje jedne izvršne datoteke s ekstenzijom (.exe) dobivene iz više objektnih datoteka. Povezivač povezuje definiciju funkcije iz jedne objektne datoteke s pozivom funkcije iz druge objektne datoteke. Budući da povezivač nasuprot kompilatoru ima uvid u sve datoteke izvornog kôda unutar projekta, on može utvrditi u kojoj objektnoj datoteci se

nalazi definicija neke funkcije, a u kojoj poziv funkcije. Ukoliko neka funkcija nema definicije niti u jednoj datoteci izvornog kôda, takvu grešku može utvrditi jedino povezivač. U ovom koraku povezivač može obavijestiti korisnika u slučaju da neka funkcija nema definiciju. Svaka definicija funkcije je jednoznačno određena povratnim tipom i imenom funkcije.

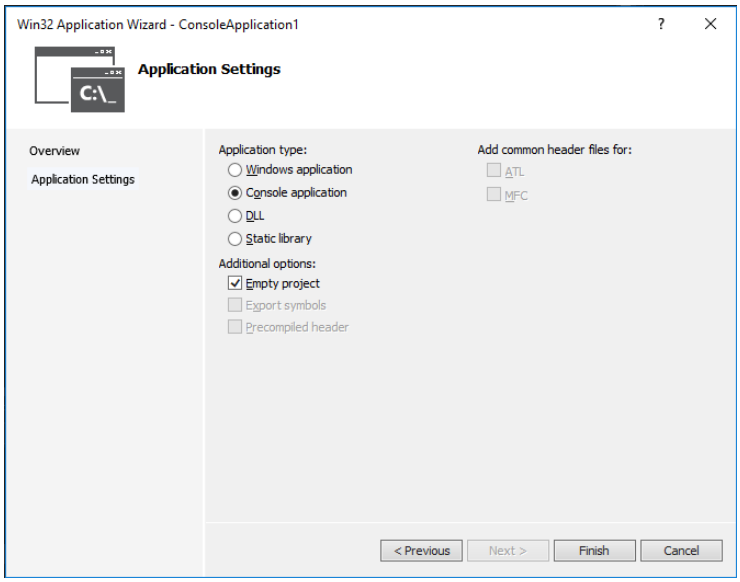
Organiziranje kôd-a u Visual Studi-u

Visual Studio omogućava jednostavno upravljanje projektima s više datoteka izvornog kôd-a. Potrebno je unutar postojećeg projekta ispravno dodati novu datoteku izvornog kôd-a i razvojno okruženje će prepoznati tu novu datoteku i uključit je u proces prevođenja. U tekstu koji slijedi bit će pokazano kako ispravno dodati nove datoteke u projekt.

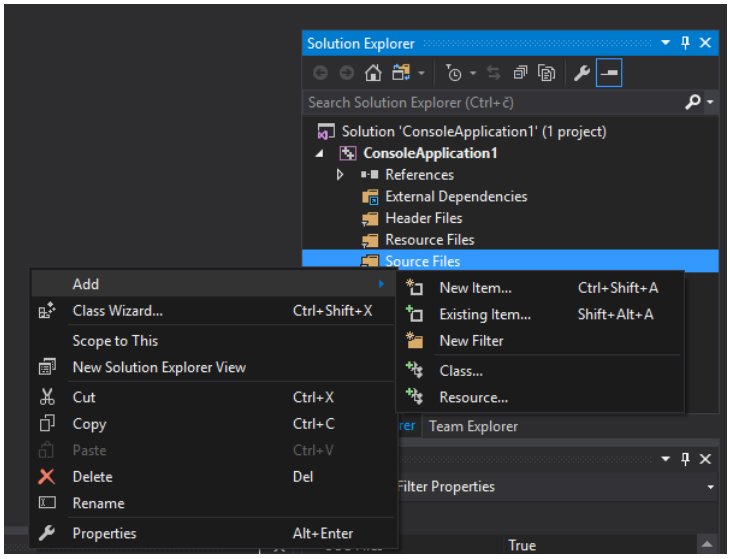
1. Potrebno je kreirati novi projekt u Visual Studiu (**File -> New -> Project**),
2. Pod *Templates* potrebno je odabrati *Visual C++*, te *Win32 Console Application* i napisati ime aplikacije



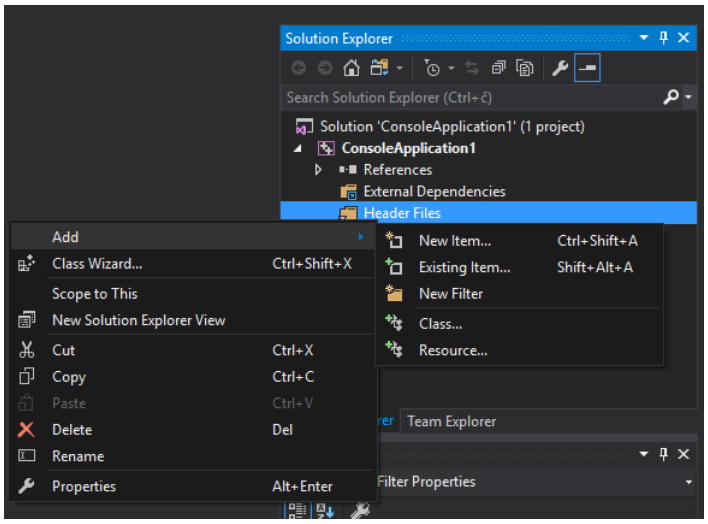
3. Pod stavkom *Additional options* isključiti *Precompiled header* i *Security Development Lifecycle (SDL) checks*, te odabrati *Empty project*,



4. Nova .c datoteka se dodaje u *Solution Exploreru* tako da se desnom tipkom miša odabere na mapu *Source Files* i odabere **Add -> New Item**. Zatim se odabere *C++ File* (.cpp) i navede se ime nove datoteke izvornog kôda.



5. Novu .h datoteku se dodaje u *Solution Exploreru* tako da se desnom tipkom miša odabere na mapu *Header Files* i odabere **Add -> New Item**. Zatim se odabere *Header File* (.h) i navede se ime nove korisnički kreirane datoteke zaglavlja.



Povezivanje identifikatora

Identifikator predstavlja ime varijabli, polja, funkcija.... Za pravilnu organizaciju kôda potrebno je razlučiti doseg imena pojedinih identifikatora. Varijable deklarirane unutar bloka vidljive su samo unutar toga bloka, analogno se odnosi i na varijable deklarirane unutar funkcija. Varijable koje su deklarirane izvan funkcija, vidljive su u cijelom kôd-u unatoč tome što se kôd može rasprostirati kroz više datoteka.

Unutarnje povezivanje

Varijable i funkcije koje su vidljive samo unutar datoteka imaju *unutarnje povezivanje* (engl. *internal linkage*). Kod unutarnjeg povezivanja mogu se imati identifikatori istog imena globalno deklarirani u različitim datotekama, međutim potrebno je poduzeti sljedeće mjere. Potrebno je sve globalne varijable i funkcije deklarirati tako da budu lokalne u svakoj datoteci, a to se postiže upotrebom ključne riječi *static* ispred tipa podatka kako bi se izbjeglo međusobno sukobljavanje imena identifikatora. Korisnički definirani tipovi podataka kreirani ključnom riječi *typedef* vidljivi su samo unutar datoteke u kojoj su deklarirani isto tako i varijable koje su deklarirane kao globalne konstante s ključnom riječi *const* imaju unutarnje povezivanje, ali njih je najbolje staviti u datoteku zaglavlja. Ovim načinom identifikatori su lokalni u različitim datotekama i povezivač neće prijaviti pogrešku tijekom povezivanja.

Primjer 1: Globalne funkcije i varijable s ključnom riječi *static* i *const*.

main.c

```c
void prvaIspis(void);
void drugaIspis(void);

int main(void) {
	prvaIspis();
	drugaIspis();
	return 0;
}
```

prva.c

```c
#include <stdio.h>

static char p[] = "Ovo je prva datoteka";
const int br = 1;

static void ispis(void) {
	printf("Funkcije ispis() iz prve datoteke\n");
}

void prvaIspis(void) {
	printf("p: %s\n", p);
	ispis();
	printf("br: %d\n", br);
}
```

druga.c

```c
#include <stdio.h>

static char p[] = "Ovo je druga datoteka";
const int br = 2;

static void ispis(void) {
	printf("Funkcije ispis() iz druge datoteke \n");
}

void drugaIspis(void) {
	printf("p: %s\n", p);
	ispis();
	printf("br: %d\n", br);
}
```

Primjer 2: Globalne varijable deklarirane tipom podatka kreiranim ključnom riječi *typedef*.

main.c

```c
void prvaIspis(void);
void drugaIspis(void);

int main(void) {
	prvaIspis();
	drugaIspis();
	return 0;
}
```

prva.c

```c
#include <stdio.h>

typedef float VAR1;
typedef double VAR2;

VAR1 var1 = 1.0f;
VAR2 var2 = 1.0;

void prvaIspis(void) {
	char p[] = "Ovo je prva datoteka";
	int br = 1;
	printf("p: %s\n", p);
	printf("br: %d\n", br);
	printf("var1: %f\n", var1);
	printf("var2: %lf\n", var2);
}
```

druga.c

```
#include <stdio.h>

typedef int VAR1;
typedef long VAR2;

VAR1 var1 = 2;
VAR2 var2 = 2L;

void drugaIspis(void) {
	char p[] = "Ovo je druga datoteka";
	int br = 2;
	printf("p: %s\n", p);
	printf("br: %d\n", br);
	printf("var1: %d\n", var1);
	printf("var2: %ld\n", var2);
}
```

Vanjsko povezivanje

Identifikatori koji su prilikom povezivanja vidljivi i u drugim datotekama izvornog kôd-a imaju *vanjsko povezivanje* (engl. e*xternal linkage*). Takvi identifikatori mogu se koristiti i u drugim datotekama izvornog kôd-a i zbog toga u cijelom programu smije postojati samo jedna varijabla ili funkcija s tim imenom. Ključnom riječi *extern* ispred tipa podatka globalno deklarirane varijable ili funkcije unutar datoteke izvornog kôd-a, omogućava se vanjsko povezivanje, odnosno dohvatljivost iz svih datoteka. Umetanjem ključne riječi *extern* ispred deklaracije kompilatoru se daje do znanja da je ta varijabla ili funkcija definirana u drugoj datoteci.

Primjer 3: Primjer eksplicitnog postavljanja globalno vidljivih varijabli i funkcija ključnom riječi *extern*

main.c

```
void prvaIspis();

int main(void) {
	prvaIspis();
	return 0;
}
```

prva.c

```c
#include <stdio.h>

extern int br2;
extern int drugaIspis(int);

void prvaIspis() {
	char p[] = "Ovo je prva datoteka";
	int br = 1;
	printf("p: %s\n", p);
	printf("br: %d\n", br);
	printf("br2: %d\n", br2);
	printf("extern: %d\n", drugaIspis(br));
}
```

druga.c

```c
#include <stdio.h>

int br2 = 2;

int drugaIspis(int temp) {
	char p[] = "Ovo je druga datoteka";
	printf("p: %s\n", p);
	return ++temp;
}
```

Datoteka zaglavlja

U datoteku zaglavlja stavlja se općenito sve što se nalazi na vrhu datoteke izvornog kôd-a:

- Komentari (za datoteku zaglavlja),
- definicija strukture i unija,
- *typedef* (korisnički tipovi podataka),
- deklaracije (prototip) funkcija,
- deklaracije globalnih varijabli,
- definicija konstanti (ključna riječ *const*),
- makro definicije i funkcije,
- enumeracije (pobrojenja),
- predprocesorske naredbe za uvjetno izvršavanje,
- uključivanje drugih zaglavlja predprocesorskom naredbom *#include*

Gore spomenute točke su prijedlozi, a ne pravilo. Pri uključivanju zaglavlja u zaglavlje važno je ograničiti se samo na neophodna zaglavlja kako bi se izbjeglo stvaranje prevelikih ovisnosti, ostala se zaglavlja mogu navesti u samoj datoteci izvornog kôd-a.

Kao što je već od prije poznato, prije prvog poziva funkcije potrebno je navesti deklaraciju funkcije. Deklaracija (prototip) funkcije mora biti izdvojena od definicije (implementacije) funkcije. Deklaracija se izdvaja u zasebnu datoteku zaglavlja koja se predprocesorskom naredbom *#include* uključuje na početak datoteke izvornog kôd-a. Kako se deklaracija funkcije sada nalazi u datoteci zaglavlja, njezina izmjena se može izvršiti samo u datoteci zaglavlja i time izbjeći potencijalne pogreške prilikom višestrukog ispravljanja. Radi preglednosti, dobra je praksa datotekama zaglavlja dati isto ime kao što imaju datoteke s implementacijom. Datoteke zaglavlja imaju ekstenziju (.h ili .hpp), dok datoteke s implementacijom imaju ekstenziju (.c ili .cpp).

Korisnički definirane datoteke zaglavlja

Potrebno je zamijetiti kad se uključuju datoteke zaglavlja iz standardne biblioteke potrebno je ime zaglavlja staviti između znakova manje/veće (< >), a korisnički definirane datoteke zaglavlja stavljaju se unutar dvostrukih navodnika (" ").

U svim datotekama zaglavlja trebaju se nalaziti predprocesorske naredbe *#ifndef-#define-#endif*. Pomoću ovog ispitivanja izbjegava se mogućnost višestrukog umetanja sadržaja datoteka.

```
#ifndef HEADER_H
#define HEADER_H
// deklaracije i definicije koje idu u zaglavlje
#endif // HEADER_H
```

Kada predprocesor prvi puta skenira datoteku izvornog kôd-a i prvi puta naiđe na predprocesorsku naredbu za uključivanje datoteke *#include "header.h"* makro ime *HEADER_H* nije definirano, te se umeće kôd između *#ifndef* i *#endif* u datoteku izvornog kôd-a. Unutar bloka *#ifndef* i *#endif* nalazi se naredba *#define HEADER_H* kojom se makro ime *HEADER_H* definira na neku neodređenu vrijednost vezano za datoteku izvornog kôd-a u kojoj je umetnuto to zaglavlje. Zato u slučaju ponovljenog

uključivanja datoteke *#include "header.h"* u istu datoteku izvornog kôd-a, makro ime *HEADER_H* je definirano i predprocesor će preskočiti umetanje kôd-a između *#ifndef* i *#endif* te će se time izbjeći novo umetanje sadržaja iz spomenute datoteke zaglavlja. Ovakvo osiguranje od nepotrebnog višestrukog uključivanja sadržaja datoteke zaglavlja sa spomenutim predprocesorskim naredbama *naziva se zaštita uključivanja* (engl. *include guard*). Teoretski ako datoteka zaglavlja ne bi imala zaštitu uključivanja i ako bi se unutar iste datoteke izvornog kôd-a ponovno uključila ista datoteka zaglavlja, unutar datoteke izvornog kôda, ponovio bi se dva puta isti sadržaj datoteke zaglavlja gdje bi u tom slučaju kompilator javio pogrešku kako se u deklaracijama pojavljuju identifikatori istog naziva, a poznato je od prije kako nije moguće imati identifikator istog imena unutar istog područja kôda.

Primjer 4: Primjer uključivanja korisnički kreirane datoteke zaglavlja u datoteku izvornog kôd-a.

header.h

```c
#ifndef HEADER_H
#define HEADER_H
	void prvaIspis(void);
	void drugaIspis(void);
	typedef int VAR;
	const VAR var = 5;
#endif // HEADER_H
```

main.c

```c
#include <stdio.h>
#include "header.h"
// #include "C:\\headers\\header.h"

int main(void) {
	prvaIspis();
	drugaIspis();
	printf("var: %d\n", var);
	return 0;
}
```

prva.c

```c
#include <stdio.h>

static char p[] = "Ovo je prva datoteka";
static int br = 1;

void prvaIspis(void) {
	printf("p: %s\n", p);
	printf("br: %d\n", br);
}
```

druga.c

```c
#include <stdio.h>

static char p[] = "Ovo je druga datoteka";
static int br = 2;

void drugaIspis(void) {
	printf("p: %s\n", p);
	printf("br: %d\n", br);
}
```

Treba primijetiti kako definicija makro imena unutar datoteke zaglavlja vrijedi samo unutar datoteke izvornog kôd-a u koju je uključena. Jednom definirano makro ime onemogućava ponovno uključivanje iste datoteke zaglavlja samo za tu datoteku izvornog kôd-a. Kada se započne kompiliranje sljedeće datoteke izvornog kôd-a unutar koje se uključi ista datoteka zaglavlja, predprocesor će je uključiti, ali samo za tu datoteku izvornog kôda jer se u njoj prvi puta spominje. Svaka datoteka izvornog kôd-a predstavlja zasebnu kompilatorsku cjelinu.

Zadaci za vježbu

1. Napisati C program koji će omogućiti zauzimanje memorije za dvodimenzionalno polje *polje2d* cjelobrojnih vrijednosti tipa short duljine *m* redova i *n* stupaca. Unutar dvodimenzionalnog polja potrebno je ispod sporedne dijagonale pronaći najveći parni broj i tim parnim brojem prepisati sve vrijednosti iznad sporedne dijagonale. Ispisati novonastalo dvodimenzionalno polje. Popuniti dvodimenzionalno polje pseudo-slučajnim vrijednostima, kao i varijable *m*, *n*.
    - U *main()* funkciji (u glavnom dijelu programa) postaviti *m*, *n* na pseudo-slučajnu vrijednost iz traženog intervala, (3<m<=18), (3<n<=25) i pozvati sve ostale funkcije za upravljanjem dvodimenzionalnim poljem.
    - Napisati funkciju *zauzimanjeMatrice()* pomoću koje se dinamički zauzima memorijski prostor da dvodimenzionalno polje (u potpunosti rukovati memorijom). Funkcija *zauzimanjeMatrice()* vraća zauzeto dvodimenzionalno polje, a za parametre prima broj redova i stupaca.
    - Napisati funkciji *popunjavanjeMatrice()* pomoću koje treba popuniti dvodimenzionalno polje pseudo-slučajnim vrijednostima iz intervala [-1550, 250]. Funkcija p*opunjavanjeMatrice()* ne vraća vrijednost, a kao parametar prima dvodimenzionalno polje, te broj redova i stupaca.
    - Napisati funkciju *najveciParniBroj()* pomoću koje je potrebno pronaći najveći parni broj ispod sporedne dijagonale dvodimenzionalnog polja. Funkcija *najveciParniBroj()* vraća vrijednost, a kao parametar prima dvodimenzionalno polje, broj redova i stupaca.
    - Napisati funkciju *novaMatrica()* pomoću koje treba najvećim parnim brojem prepisati sve vrijednosti iznad sporedne dijagonale. Funkcija *novaMatrica()* ne vraća vrijednost, a kao parametar prima dvodimenzionalno polje, broj redova i stupaca, te najveći parni broj.
    - Napisati funkciju i*spisMatrice()* pomoću koje treba ispisati dvodimenzionalno polje u matričnom obliku. Funkcija i*spisMatrice()* ne vraća vrijednost, a kao parametar prima dvodimenzionalno polje, te broj redova i stupaca.
    - Napisati funkciju *brisanjeMatrice()* pomoću koje se treba osloboditi memorija. Funkcija *brisanjeMatrice()* vraća *NULL* vrijednost, a kao parametar prima dvodimenzionalno polje, te broj redova.

    Program organizirati u više datoteka!
    Koristiti isključivo pokazivačku notaciju!