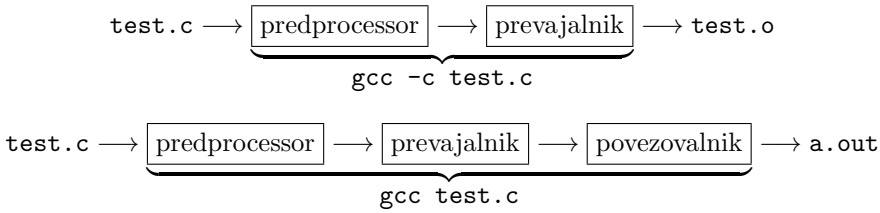
1 Predprocesor

1 Programski jezik C pri prevajanju uporablja predprocesor. Ta programerju omogoča predvsem naslednje:

1. Vključevanje drugih izvornih datotek v prevajano izvorno datoteko.
2. Uporaba makrojev v izvorni kodi.
3. Pogojno prevajanje delov prevajane izvorne datoteke.

2 Vsi ukazi predprocesorja se začnejo z znakom #, vsak od njih pa mora biti v svoji vrstici. Če je vrstica prekratka za celoten ukaz, se ukaz lahko razteza čez več vrstic. V tem primeru moramo vse vrstice ukaza razen zadnje končati z znakom "\".

3 [ZA ZAHTEVNEJŠEGA BRALCA] Uporaba predprocesorja je običajno samodejna. To pomeni, da prevajalnik, ki v širšem pomenu besede združuje več faz prevajanja, pred samim začetkom prevajanja izvorno datoteko najprej preda predprocesorju, ki izvorno kodo v njej predela in jo šele nato preda nazaj prevajalniku, da opravi prevajanje in nato morebiti še povezovanje:



1.1 Vključevanje drugih izvornih datotek v prevajano izvorno datoteko

4 Če želimo v izvorno datoteko vstaviti neko drugo izvorno datoteko, imamo zato na voljo dva ukaza:

- # **include** <*ime-datoteke*>
- # **include** "*ime-datoteke*"

Oba ukaza nadomestita vrstico, v kateri se nahajata, z vsemi vrsticami vrstice, katere ime je podano. Razlika je v tem, da prvi ukaz išče datoteko na sistemskih direktorijih, drugi pa začne iskanje na direktoriju, kjer se nahaja osnovna izvorna datoteka, če pa tam datoteke, ki jo je potrebno vstaviti, ne najde, nadaljuje z iskanjem na sistemskih direktorijih.

Vključevanje izvornih datotek deluje rekurzivno: če so v vstavljeni izvorni kodi novi ukazi # **include**, jih predprocesor izvrši in to ponavlja toliko časa, da razreši vse # **include** ukaze.

5 Oglejmo si to na majhnem primeru. Predpostavimo, da imamo v direktoriju `src`, kjer hranimo izvorno kodo, datoteko z imenom `include-1.c`:

⟨ `include-1.c` 5 ⟩ ==
```
int i;
```

6 Datoteko `include-1.c` vstavimo na želeno mesto v datoteko `include-example-1.c`, ki je ravno tako v direktoriju `src`, z ukazom "# **include** `"include-1.c"`":

⟨ `include-example-1.c` 6 ⟩ ==
```
int v1;
#include "include-1.c"
int v2;
```

7 Ko datoteko `include-example-1.c` prevedemo z ukazom

```
$ gcc -c include-example-1.c
```

, predprocesor izvorno kodo v datoteki `include-example-1.c` najprej predela tako, da vrstico "# **include** `"include-1.c"`" nadomesti z vrstico "**int** *i*;", zato prevajalnik prevede naslednjo izvorno kodo:

int *v1*;
int *i*;
int *v2*;

8 Ob tem povejmo, da prevajanje še vedno uspe, četudi se premaknemo en nivo po direktorijski strukturi višje:

```
$ cd ..  ; gcc -c src/include-example-1.c
```

Ukaz "# **include** `"include-1.c"`" namreč začne z iskanjem datoteke `include-1.c` na direktorijo osnovne izvorne datoteke `include-example-1.c`, torej na direktoriju `./src`.

9 [ZA ZAHTEVNEJŠEGA BRALCA] Če smo radovednejši, lahko zelo natančno ugotovimo, kakšno izvorno kodo predprocesor preda prevajalniku v prevajanje. V ta namen poženemo prevajanje datoteke `include-example-1.c` z ukazom

```
$ gcc -E include-example-1.c
```

in dobimo izpis

```
# 1 "include-example-1.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 325 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "include-example-1.c" 2
int v1;

# 1 "./include-1.c" 1
int i;
# 3 "include-example-1.c" 2
int v2;
```

A pozor: ta izpis se lahko od prevajalnika do prevajalnika malce razlikuje.

Vse vrstice, ki se začnejo z znakom #, so zgolj informacija prevajalniku, od kod izvira kakšna vrstica predelane izvorne kode — to je pomembno predvsem zato, da lahko prevajalnik pravilno izpiše vrstico, v kateri se pojavi morebitna napaka v izvorni kodi. Vrstica oblike

```
# 1 "include-example-1.c" 2
```

tako na primer prevajalniku pove, da je naslednja vrstica, torej vrstica "**int** *v1*;", 1. vrstica datoteke `include-example-1.c` in da se je predprocesiranje ravnokar vrnilo v nadaljevanje obdelave datoteke `include-example-1.c`. Več o tem si bralec lahko pogleda v

```
https://gcc.gnu.org/onlinedocs/cpp.pdf
```

10 Vključevanje izvornih datotek je pri programiranju v programskem jeziku C povsem običajno. Celo v začetniškem programčku pride do vstavljanja izvorne datoteke `stdio.h`:

⟨ include-example-2.c 10 ⟩ ==

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

11 Iz opisa obeh ukazov # **include** je jasno, da bi lahko v programčku `include-example-2.c` uporabili namesto ukaza # **include** `<stdio.h>` ukaz # **include** `"stdio.h"`. A v tem primeru obstaja ena nevarnost, bralec pa naj jo za vajo odkrije sam.

1.2 Uporaba makrojev v izvorni kodi

12 Makroji so enostavni ukazi, s katerimi predprocesorju povemo, kako naj določene dele izvorne kode nadomesti z drugimi. Poznamo dve vrsti makrojev: z in brez parametrov.

13 Makroje brez parametrov enostavno definiramo z ukazom oblike

- # **define** *ime-makroja nadomestno-besedilo*

S tem ukazom dosežemo, da bo predprocesor vsako besedo *ime-makroja* zamenjal z besedilom *nadomestno-besedilo*. Pri tem velja pravilo, da predprocesor vedno deluje na nivoju besed programskega jezika C: če pride do zamenjave imena makroja z nadomestnim besedilom, predprocesor vedno nadomesti celo besedo, nikoli zgolj le dela besede.

14 Makroje brez parametrov se pogosto uporablja kot nadomestilo za konstante, ki se pojavljajo na večih mestih v večjem programu. Če to ponazorimo na manjšem programu za izračun praštevil z Eratostenovim rešetom, začnemo z izvedno brez uporabe makrojev:

⟨ macro-example-1.c 14 ⟩ ==

```
#include <stdio.h>
#include <math.h>
  int is_prime[100 + 1];
  int main( )
  {
    int n;
    is_prime[0] = 0; is_prime[1] = 0;
    for (n = 2; n <= 100; n++) is_prime[n] = 1;
    for (n = 2; n <= (int)(sqrt(100)); n++)
      if (is_prime[n]) for (int m = 2 * n; m <= 100; m += n) is_prime[m] = 0;
    for (n = 0; n <= 100; n++) if (is_prime[n]) printf("%d ", n); printf("\n");
    return 0;
  }
```

15 Gornji program izračuna in izpiše vsa praštevila do 100, a če bi želeli izračunati in izpisati vsa praštevila do 1000, bi morali že v majhnem programu zamenjati število 100 s številom 1000 na štirih mestih. Da bi se izognili večkratnemu popravljanju programa, lahko z uporabo ukaza "# **define** *N* 100" dosežemo, da bomo v bodoče število 100 morali zamenjati s 1000 ali s katerim drugim število zgolj enkrat.

⟨ macro-example-2.c 15 ⟩ ==

```
#include <stdio.h>
#include <math.h>
#define N  100
  int is_prime[N + 1];
```

```
int main()
{
  int n;
  is_prime[0] = 0; is_prime[1] = 0;
  for (n = 2; n <= N; n++) is_prime[n] = 1;
  for (n = 2; n <= (int)(sqrt(N)); n++)
    if (is_prime[n]) for (int m = 2 * n; m <= N; m += n) is_prime[m] = 0;
  for (n = 0; n <= N; n++) if (is_prime[n]) printf("%d ", n); printf("\n");
  return 0;
}
```

16 Ob tem povejmo še to, da uporaba makrojev namesto konstant dodatno poveča čitljivost programa. V večjem programu lahko namreč ena in ista konstanta nastopa v različnih pomenih, z uporabo makrojev pa lahko uvedemo več različnih imen iste konstante — po eno ime za vsak pomen.

17 Makroje s parametri definiramo z ukazom oblike

- # **define** *ime-makroja*(*seznam-parametrov*) *nadomestno-besedilo*

Seznam parametrov je zaporedje imen parametrov, ki so ločeni z vejicami in se lahko pojavijo v nadomestnem besedilu.

18 Preprost primer makroja s parametrom je makro, ki omogoči izračun absolutne vrednosti svojega argumenta:

```
#define ABS(X)  (((X) >= 0) ? (X) : -(X))
```

19 Podobno lahko sestavimo makroja za izračun minimuma in maksimuma dveh vrednosti:

```
#define MIN(X, Y)  ((X) < (Y) ? (X) : (Y))
#define MAX(X, Y)  ((X) > (Y) ? (X) : (Y))
```

20 Pri definiciji makroja se moramo zavedati, da makro ni funkcija, temveč le pravilo, s katerim bo predprocesor namesto nas predelal izvorno kodo programa. Vzemimo za primer naslednji program:

⟨ macro-example-3.c 20 ⟩ ==

```
#include <stdio.h>
#define ABS(X)  (((X) >= 0) ? (X) : -(X))
  int f(int i)
  {
    static int n = 0;
    return i + (n++);
  }
  int main() { printf("%d\n", ABS(f(5))); return 0; }
```

21 Če bi se makro `ABS` obnašal kot funkcija, bi se njegov argument $f(5)$ izračunal pred uporabo makroja `ABS`. To bi pomenilo, da bi se izraz `ABS`$(f(5))$ izračunal kot `ABS`(5), kar bi se prepisalo v

$$(((5) >= 0) \ ? \ (5) : -(5))$$

in program bi izpisal `5`.

A makro se obnaša drugače. Najprej se opravi prepis izraza `ABS`$(f(5))$ v izraz

$$(((f(5)) >= 0) \ ? \ (f(5)) : -(f(5))) \quad .$$

V tem izrazu se najprej izvede prvi klic funkcije f, ki vrne 5, in izraz se spremeni v

$$(((5) >= 0) \ ? \ (f(5)) : -(f(5))) \quad .$$

Nato se izvede drugi klic funkcije f, ki zaradi povečanja statične spremenljivke n za 1 v prvem klicu funkcije f tokrat vrne 6, in zato je vrednost izraza 6. Program torej izpiše `6`.

Povedano drugače: če bi se makro obnašal kot funkcija, bi se njegov argument izračunal pred uporabo makroja, tako pa se izračuna po uporabi makroja — zato se funkcija f v izrazu `ABS`$(f(5))$ kliče dvakrat in ne zgolj enkrat.

22 Posebej je potrebno poudariti, da je uporaba navidez odvečnih oklepajev v definicijah makrojev `ABS`, `MIN` in `MAX` nujna, če želimo zagotoviti varno uporabo teh makrojev. Kot dokaz te trditve si oglejmo program, v katerem je makro `ABS` nevarno definiran brez oklepajev:

```
#include <stdio.h>
#define ABS(X)  X >= 0 ? X : -X      /* nevarno */
  int main( ) { printf("%d\n", -6 + ABS(5)); return 0; }
```

23 Če bi se makro `ABS` obnašal kot funkcija, bi se njegova vrednost izračunala pred seštevanjem. Izraz $-6 +$ `ABS`(5) bi se spremenil v izraz $-6 + 5$ in program bi izpisal `-1`.

A makro se obnaša drugače. Najprej se opravi prepis izraza $-6 +$ `ABS`(5) v izraz

$$-6 + 5 >= 0 \ ? \ 5 : -5 \quad ,$$

zato se najprej opravi seštevanje in izraz se spremeni v

$$-1 >= 0 \ ? \ 5 : -5 \quad .$$

Ta izraz ima vrednost -5 in zato program izpiše `-5`.

Povedano drugače: brez pazljive uporabe oklepajev okoli posameznega parametra in okoli celotnega nadomestnega besedila se nadomestno besedilo makroja lahko zlepi z besedilom, v katerem je makro uporabljen, rezultat pa je zato lahko precej drugačen od pričakovanega.

1.3 Pogojno prevajanje delov prevajane izvorne kode

24 Pomembna lastnost predprocesorja je tudi omogočanje pogojnega prevajanja izvorne kode. To pomeni, da lahko v izvorni kodi določimo, kateri deli izvorne kode naj se prevedejo in kateri naj se ne prevedejo.

25 Pogojno prevajanje omogočajo konstrukti oblike

- # **ifdef** *ime-makroja* *izvorna-koda* # **endif**
- # **ifndef** *ime-makroja* *izvorna-koda* # **endif**
- # **if** *izraz* *izvorna-koda* # **endif**

V vseh treh primerih lahko uporabljamo tudi ukaz # **else**.

26 Za konec se še enkrat vrnimo k Eratostenovemu rešetu in v programu iz razdelka 15 obdajmo ukaz "# **define** N 100" z ukazoma "# **ifndef** N" in "# **endif**", poleg tega pa prestavimo tabelo *is_prime* v funkcijo *main* ne da bi zahtevali statično lokalno tabelo. Pri tem zagotovimo, da nikoli ne zahtevamo prevelike tabele znotraj funkcije: če je N manjši ali enak 100, tedaj lahko uporabimo tabelo, sicer pa uporabimo kazalec in tabelo ustvarimo na kopici s klicem funkcije *malloc*.

⟨ `macro-example-4.c` 26 ⟩ ==

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#ifndef N
#define N  100
#endif
```

```
int main( )
{
  int n;
#if (N <= 100)
  int is_prime[N + 1];
#else
  int *is_prime;
  if ((is_prime = (int *) malloc((N + 1) * sizeof(int))) == NULL) {
    fprintf(stderr, "Not enough memory.\n"); exit(1); }
#endif
  is_prime[0] = 0; is_prime[1] = 0;
  for (n = 2; n <= N; n++) is_prime[n] = 1;
  for (n = 2; n <= (int)(sqrt(N)); n++)
    if (is_prime[n]) for (int m = 2 * n; m <= N; m += n) is_prime[m] = 0;
  for (n = 0; n <= N; n++) if (is_prime[n]) printf("%d ", n); printf("\n");
#if (N > 100)
  free(is_prime);
#endif
  return 0;
}
```

27 Brez spreminjanja izvorne datoteke `macro-example-4.c` bi predprocesor predelal izvorno kodo funkcije *main* v naslednjo izvorno kodo (pred tem pa bi seveda vključil še vse, kar je v datotekah `stdio.h`, `stdlib.h` in `math.h`):

```
int main( )
{
  int n;
  int is_prime[100 + 1];
  is_prime[0] = 0; is_prime[1] = 0;
  for (n = 2; n <= 100; n++) is_prime[n] = 1;
  for (n = 2; n <= (int)(sqrt(100)); n++)
    if (is_prime[n]) for (int m = 2 * n; m <= 100; m += n) is_prime[m] = 0;
  for (n = 0; n <= 100; n++) if (is_prime[n]) printf("%d ", n); printf("\n");
  return 0;
}
```

28 Če bi spremenili makro N na vrednost 1000, bi predprocesor predelal izvorno kodo funkcije *main* v naslednjo izvorno kodo (pred tem pa bi seveda vključil še vse, kar je v datotekah `stdio.h`, `stdlib.h` in `math.h`):

```
int main( )
{
  int n;
  int *is_prime;
  if ((is_prime = (int *) malloc((N + 1) * sizeof(int))) == NULL) {
    fprintf(stderr, "Not enough memory.\n"); exit(1); }
  is_prime[0] = 0; is_prime[1] = 0;
  for (n = 2; n <= N; n++) is_prime[n] = 1;
  for (n = 2; n <= (int)(sqrt(N)); n++)
    if (is_prime[n]) for (int m = 2 * n; m <= N; m += n) is_prime[m] = 0;
  for (n = 0; n <= N; n++) if (is_prime[n]) printf("%d ", n); printf("\n");
  free(is_prime);
  return 0;
}
```

29 Če program `macro-example-4.c` prevedemo z običajnim ukazom

```
$ gcc macro-example-4.c
```

, se program prevede natanko tako kot program `macro-example-2.c` iz razdelka 15, saj makro N ni definiran ne v `stdio.h` ne v `math.h`. A zaradi ukazov "# **ifndef** N" in "# **endif**" ga lahko sedaj prevedemo tudi takole:

```
$ gcc -DN=1000 macro-example-4.c
```

Določilo `-DN=1000` v ukazni vrstici določa, da se predprocesor zažene z vnaprej definiranim makrojem N, ki ima vrednost 1000, torej enako kot bi bil definiran z ukazom "# **define** N 1000". A ker je makro N pred začetkom predprocesiranja že definiran, pogoj v ukazu "# **ifndef** N" ni izpolnjen, se ukaz "# **define** N 100" zato izpusti. Skratka, brez popravljanja izvorne kode lahko program prevedemo za različne vrednosti N.

2 Pisanje velikih programov

30 Med učenjem programiranja večinoma pišemo kratke programe, pri katerih je zelo priročno, da je cel program napisan v eni sami datoteki. A pri profesionalnem programiranju v programskem jeziku C (kakor tudi v večini drugih programskih jezikov) se le redko zgodi, da je cel program napisan v eni sami datoteki. Ravno nasprotno, običajno program razdelimo na več datotek, razlogov zato pa je več:

- Program običajno želimo razdeliti na več del, od katerih vsak del opravlja eno funkcionalnost programa (takim delom pogosto pravimo moduli). Te dele pogosto razvijamo ločeno (pogosto jih razvijajo različni programerji), zato je priročno, da je vsak tak del program v svoji datoteki ali še pogosteje v svoji skupini datotek.
- Programski jezik C obravnava eno izvorno datoteko (in vse pripadajoče izvorne datoteke, ki so vključene vanjo med predprocesiranjem) kot eno enoto prevajanja. V okviru posamezne enote prevajanja pa programski jezik omogoča dodatno kontrola nad tem, kateri elementi posameznega dela programa so vidni v drugih delih programa in kateri niso. Elementi programa, ki so lahko vidni ali pa ne, so predvsem funkcije in spremenljivke.
- Z razbitjem programa na manjše enote prevajanja dosežemo, da je prevajanje programa med razvojem in vzdrževanjem programa bistveno hitrejše, saj lahko vsakič prevedemo le tisti del programa, ki je bil spremenjen. Pri majhnih programih, kjer prevajanje traja le nekaj trenutkov, razlike ne čutimo, pri prevajanju večjih programov, kjer celotno prevajanje lahko traja tudi po več deset minut, pa je razlika pomembna.

Na tem mestu se ne bomo ukvarjali z vprašanjem, kako razbiti program na ustrezne dele, temveč le, kako te dele predstaviti v programskem jeziku C in jih potem prevesti v delujoč program.

2.1 Prevajanje večih izvornih datotek v en program

31 Pri prevajanju programa, ki je sestavljen iz več datotek, je zelo pomembno, da razumemo, kako se različni deli programa najprej prevedejo in nato še povežejo v en delujoč program.

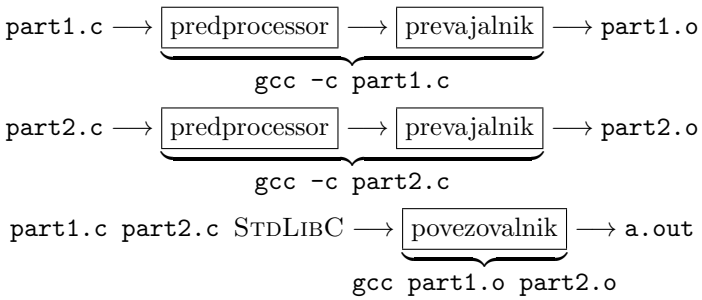
Prevajalnik za programski jezik C omogoča vsaj tri faze prevajanja programov:

1. *predprocesiranje*: Izvorno kodo, ki jo je napisal programer in shranil v datoteki s končnico `.c`, predprocesor (opisan v prejšnjem poglavju) predela v izvorno kodo, ki je namenjena prevajanju v ožjem smislu besede.
2. *prevajanje* (v ožjem smislu besede): Izvorno kodo, ki je predstavljena kot datoteka z besedilom programa v programskem jeziku C, prevajalnik prevede v strojno kodo in jo shrani v "objektno" datoteko s končnico `.o` (ime "objektna" nima nobene zveze z objektno usmerjenim programiranjem).
3. *povezovanje*: Povezovalnik poveže vse datoteke z objektno kodo in zahtevane knjižnice (recimo standardno knjižnico programskega jezika C) v en sam izvršljiv program. Pri povezovanju se zagotovi, da so vsi elementi programa, ki so vidni in uporabljeni v večih delih programa, vsak od njih pa definiran le v enem delu programa, prisotni.

32 Prevajanje programa, ki je sestavljen iz dveh izvornih datotek `part1.c` in `part2.c`, opravimo z naslednjim zaporedjem ukazov

```
$ gcc -c part1.c
$ gcc -c part2.c
$ ls *.o
part1.o part2.o
$ gcc part1.o part2.o
$ ls a.out
a.out
```

ali kot si to shematsko predstavljamo takole:



Vrstni red prvih dveh ukazov lahko tudi zamenjamo. Pri povezovanju nam ni potrebno posebej navesti standardne knjižnice, saj jo prevajalnik sam doda pri povezovanju.

33 [za zahtevnejšega bralca] Prevajanje v ožjem smislu besede je lahko neposredna pretvorba izvorne kode v strojno kodo, lahko pa prevajalnik najprej prevede izvorno kodo v zbirnik, ki se nato z drugim programom, običajno brez posredovanja programerja in zato zanj nevidno prevede v strojno kodo. Bralec je vabljen, da poskusi svoj prevajalnik prisiliti, da se pred prevajanjem zbirnika v strojno kodo ustavi, in si strojno kodo ogleda.

2.2 Vidnost elementov programa v posameznih enotah prevajanja

34 Na tem mestu ne moremo predstaviti zares velikega programa z 10^5 ali celo 10^6 vrsticami, zato si moramo kljub vsemu pomagati z manjšim, ki bo kljub temu služil za predstavitev ustreznih konceptov.

35 Kot primer vzemimo majhen program za izračun Fibonaccijevih števil in ga razbijmo na dva dela: en del naj obsega izračun Fibonaccijevih števil, drugi del pa izpis izračunanih Fibonaccijevih števil.

Glavni del programa, tisti s funckijo *main*, shranimo v datoteko `big-main-1.c` ("glavni" je seveda po tem, da se izvajanje programa začne, usmerja in z malo sreče tudi konča v tem delu, nikakor pa seveda ne po tem, da bi bila v njem najpomembnejši del programa — to je vendar izračun Fibonaccijevih števil).

Glavni del programa naj vsebuje funkcijo *main*, ta pa naj kliče funkcijo *fibbonacci*, ki je definirana v drugem delu programa, torej v datoteki `big-fibo-1.c`, in s tem avtomatično v drugi enoti prevajanja.

⟨ `big-main-1.c` 35 ⟩ ==
```
#include <stdio.h>
  int main( )
  {
    for (int n = 1; n <= 10; n++) {
      printf("fibonacci(n)=%d\n", fibonacci(n));
    }
    return 0;
  }
```

36 Drugi del programa, tisti, ki vsebuje funckijo *fibonacci* za izračun Fibonaccijevih števil, shranimo v datoteko `big-fibo-1.c`.

⟨ `big-fibo-1.c` 36 ⟩ ==
```
  int fibonacci(int n)
  {
    if ((n == 1) ∨ (n == 2)) return 1;
    else return (fibonacci(n − 1) + fibonacci(n − 2));
  }
```

37 Obe datoteki prevedemo najprej le do objektne datoteke:

```
$ gcc -c big-fibo-1.c
$ gcc -c big-main-1.c
big-main-1.c:6:28: warning: implicit declaration of function 'fibonacci' is
invalid
in C99 [-Wimplicit-function-declaration]
printf("fibonacci(n)=%d\n",fibb(n));
$ ls *.o
big-fibo-1.o big-main-1.o
```

Prevajalnik nam ob prevajanju datoteke `big-main-1.c` javi, da je moral privzeti, da je funkcija *fibonacci* definirana implicitno, saj v datoteki `big-main-1.c` ni njene definicije.

38 Kljub temu lahko oba dela programa povežemo z ukazom

```
$ gcc big-main.o big-fibo-1.o
```

in nato uspešno poženemo z ukazom `./a.out`.

39 Poglejmo, kaj bi se zgodilo, če bi skušali prevesti samo datoteko `big-main-1.c`:

```
$ gcc big-main-1.c
./del-pisanje-velikih-programov.w:55:28: warning: implicit declaration of
function 'fibonacci' is invalid in C99 [-Wimplicit-function-declaration]
printf("fibonacci(n)=%d\n",fibonacci(n));
^
1 warning generated.
Undefined symbols for architecture x86_64:
"_fibonacci", referenced from:
_main in big-main-1-18a8db.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Poleg opozorila, da je funkcija *fibonacci* definirana implicitno, bi dobili obvestilo o napaki, da povezovalnik ne najde simbola `_fibonacci`. Tako je tudi prav, saj funkcija *fibonacci* v datoteki, ki jo prevajamo, sploh ni definirana.

40 Poglejmo, kaj bi se zgodilo, če bi skušali prevesti samo datoteko `big-fibo-1.c`:

```
$ gcc big-fibo-1.c
Undefined symbols for architecture x86_64:
"_main", referenced from:
implicit entry/start for main executable
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Tokrat dobimo obvestilo, da povezovalnik ne najde simbola `_main`, ki ga program nujno potrebuje za zagon.

41 Kljub temu, da smo izvorni datoteki `big-main-1.c` in `big-fibo-1.c` uspeli povezati v delujoč program, prevajalnik opozori, da funkcija *fibonacci* v izvorni datoteki `big-main-1.c` ni definirana.

Vsako opozorilo prevajalnika je dobro vzeti resno in odpraviti vzrok za opozorilo. Zato v datoteki `big-main-2.c` funkcijo *fibonacci* deklariramo s prototipom in s ključno besedo **extern** povemo, da bo definirana v neki drugi enoti prevajanja, ki bo kazneje povezana s prevodom te enote.

Poleg tega v datoteki `big-main-2.c` deklarirajmo še spremenljivko *num_calls*, za katero s ključno besedo **extern** povemo, da bo definirana v neki drugi enoti prevajanja, ki bo kasneje povezana s prevodom te enote.

```
⟨ big-main-2.c 41 ⟩ ==
#include <stdio.h>
  extern int num_calls;
  extern int fibonacci(int n);
  int main( )
  {
    for (int n = 1; n <= 10; n++) {
      num_calls = 0;
      printf("fibonacci(n)=%d (using %d calls)\n", fibonacci(n), num_calls);
    }
    return 0;
  }
```

42 V datoteki `big-fibo-2.c` definiramo spremenljivko *num_calls* in funkcijo *fibonacci* običajno.

```
⟨ big-fibo-2.c 42 ⟩ ==
  int num_calls = 0;
  int fibonacci(int n)
  {
    num_calls++;
    if ((n == 1) ∨ (n == 2)) return 1;
    else return (fibonacci(n − 1) + fibonacci(n − 2));
  }
```

43 Če sedaj prevedemo in povežemo datoteki `big-main-2.c` in `big-fibo-2.c`, prevajalnik ne izpiše opozorila ter uspešno prevede in poveže program.

44 Na tem mestu si razjasnimo še razliko med definicijo in deklaracijo. Definicija zares opiše vse lastnosti elementa, ki ga definiramo, hkrati pa mora prevajalnik na osnovi definicije zares generirati ustrezen prevod. Deklaracija pove manj kot definicija in poda le tisti del opisa elementa, ki ga prevajalnik nujno potrebuje takrat, ko prevaja druge elemente, v katerem je deklarirani element uporabljen.

- Pri deklaraciji funkcije navedemo le prototip, ne pa tudi jedra funkcije, s katerim v definiciji funkcije opišemo, kako funkcija deluje in izračuna svoje rezultate.
- Pri deklaraciji spremenljivke navedemo le tip in ime, ne moremo pa določiti začetne vrednosti spremenljivke, kar lahko storimo zgolj v definiciji te spremenljivke.

Zavedajmo se, da je vsaka definicija hkrati tudi deklaracija, ne velja pa tudi obratno.

45 Funkcije in spremenljivke, ki so deklarirane ali definirane na globalnem nivoju (torej zunaj funkcij ali struktur) v neki datoteki in s tem v neki enoti prevajanja, so lahko deklarirane ali definirane na tri načine:

- Z uporabo ključne besede **extern**:

  Deklaracije funkcij in spremenljivk z uporabo ključne besede **extern**, na primer

  ```
  extern int num_calls;
  extern int fibonacci(int n);
  ```

  , povedo, da te funkcije in spremenljivke niso nujno definirane v tej enoti prevajanja (lahko pa so!). Pri prevajanju jih prevajalnik upošteva enako kot definicije, pri povezovanju pa preveri, da zares obstaja neka enota prevajanja, kjer so definirane.

- Z uporabo ključne besede **static**:

  Definicije funkcij in spremenljivk z uporabo ključne besede **static**, na primer

static int *num_calls*;
static int *fibonacci*(**int** n);

, povedo, da so te funkcije in spremenljivke definirane v tej enoti prevajanja, a pri povezovanju niso vidne v drugih enotah prevajanja.

- Brez uporabe ključnih besed **extern** in **static**:

  Definicije funkcij in spremenljivk brez uporabe ključnih besed **extern** in **static**, na primer

  int *num_calls*;
  int *fibonacci*(**int** n); { ... }

  , povedo, da so te funkcije in spremenljivke definirane v tej enoti prevajanja in da vidne tudi v vseh drugih enotah prevajanja.

  Deklaracije funkcij brez uporabe ključnih besed **extern** in **static**, na primer

  int *fibonacci*(**int** n);

  , povedo, da te funkcije niso nujno definirane v tej enoti prevajanja (lahko pa so!). Pri prevajanju jih prevajalnik upošteva enako kot definicije, pri povezovanju pa preveri, da zares obstaja neka enota prevajanja, kjer so definirane.

46 Če bi v datoteki `big-fibo-2.c` pri obeh definicah uporabili ključno besedo **static**, bi spremenljivka *num_calls* in funkcija *fibonacci* ne bili vidni zunaj enote prevajanja, ki jo sestavlja datoteka `big-fibo-2.c`.

```
static int num_calls = 0;
static int fibonacci(int n)
{
  num_calls++;
  if ((n == 1) ∨ (n == 2)) return 1;
  else return (fibonacci(n − 1) + fibonacci(n − 2));
}
```

47 Pri prevajanju in povezovanju bi zato dobili izpis

```
Undefined symbols for architecture x86_64:
"_fibonacci", referenced from:
_main in big-main-2.o
"_num_calls", referenced from:
_main in big-main-2.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

in zaradi manjkajočih simbolov `_fibonacci` in `_num_calls`, ki v funkciji *main* nista vidna, prevajalnik ne bi uspel povezati datoteke `big-main-2.o` in spremenjene datoteke `big-fibo-2.o` v delujoč program.

48 Ob natančnem branju bralec ugotovi, da pravzaprav ni nobene razlike, ali pri prototipu funkcije, ki je dejansko definirana v neki drugi enoti prevajanja, uporabimo ključno besedo **extern** ali ne. Prototipa

int *fibonacci*(**int** n);

in

extern int *fibonacci*(**int** n);

namreč povesta, da je nekje definirana funkcija *fibonacci*: v tej enoti prevajanja ali pa v kateri drugi. Prvo obliko se pri lepem programiranju uporablja takrat, ko je funkcija definirana v isti enoti prevajanja kot prototip, drugo pa takrat, ko je definirana drugje.

49 Pri spremenljivkah je drugače.

Definicija

int *num_calls*;

pove, da je spremenljivka *num_calls* definirana v tej enoti prevajanja in da mora zato prevajalnik v tej enoti prevajanja zagotoviti **sizeof** (*num_calls*) bytov pomnilnika za hranjenje te spremenljivke. Skratka, v vseh enotah prevajanja sme biti ena sama taka definicija.

Deklaracija

extern int *num_calls*;

pove, da je spremenljivka *num_calls* definirana v neki enoti prevajanja (lahko tudi v tej, kjer je ta definicija), vendar ne zahteva, da prevajalnik zagotovi **sizeof** (*num_calls*) bytov pomnilnika za hranjenje te spremenljivke, zato mora v neki drugi enoti prevajanja obstajati definicija brez ključnih besed **extern** in **static**.

2.3 Uporaba deklaracijskih datotek

50 Za lažje združevanje različnih delov programa v celoto, običajno uporabljamo deklaracijske datoteke. Te datoteke, ki imajo po ustavljeni rabi končnico `.h`, običajno vsebujejo

- deklaracije tipov,
- deklaracije funkcij,
- deklaracije spremenljivk in
- deklaracije makrojev.

51 Oglejmo si to še enkrat na primeru izračuna Fibonaccijevih števil. Namesto, da deklaraciji spremenljivke *num_calls* in funkcije *fibonacci* navedemo v datoteki `big-main-2.c`, ju zapišemo v deklaracijsko datoteko `big-fibo-2.h`.

⟨ `big-fibo-2.h` 51 ⟩ ==

```
extern int num_calls;
extern int fibonacci(int n);
```

52 Datoteko `big-main-2.c` sedaj lahko prepišemo takole:

```
#include <stdio.h>
#include "big-fibo-2.h"
int main( )
{
  for (int n = 1; n <= 10; n++) {
    num_calls = 0;
    printf("fibonacci(n)=%d␣(using␣%d␣calls)\n", fibonacci(n), num_calls);
  }
  return 0;
}
```

53 Na majhnem primeru izračuna Fibonaccijevih števil je razlika med eksplicitnim zapisom deklaracij v datoteki `big-main-2.c` in uporabo deklaracijske datoteke `big-fibo-2.c` skoraj zanemarljiva. A v večjih programih, kjer se določene elemente uporablja v večih delih programa, postane uporaba deklaracijskih datotek velika prednost, saj odpravlja potrebo po večkratnem oziroma vsakokratnem zapisu deklaracij, s tem pa zmanjšuje možnost napak.

54 Deklaracijska datoteka je pogosto vključena v mnoge izvorne datoteke, včasih pa celo tudi v druge deklaracijske datoteke. Da bi zaradi večkratnega vstavljanja deklaracijske datoteke ne prišlo do težav, celotno vsebino deklaracijske datoteke zapremo v objem konstrukta # **ifndef** ... # **endif**, ki se nanaša na makro, ki je namenjen zgolj zagotavljanju enkratnega prevajanja vsebine take deklaracijske datoteke.

55 V datoteki `big-fibo-2.h` bi za zagotavljanje zgolj enkratne deklaracije spremenljivke *num_calls* in funkcije *fibonacci* v vsaki enoti prevajanja uporabili makro z imenom `_FIBO_H_`. Ko se tako dopolnjena datoteka `big-fibo-2.h` prvič vključi v neko enoto prevajanja, makro `_FIBO_H_` ni definiran, zato se izvorna koda znotraj konstrukta # **ifndef** ... # **endif** prevede, z njo vred pa se definira tudi makro `_FIBO_H_`. Ko se datoteka `big-fibo-2.h` naslednjič vključi v isto enoto prevajanja, bo makro `_FIBO_H_` že definiran in izvorna koda znotraj konstrukta # **ifndef** ... # **endif** se ne bo še enkrat prevedla.

```
#ifndef _FIBO_H_
#define _FIBO_H_
  extern int num_calls;
  extern int fibonacci(int n);
#endif
```

56 Na tem mestu dodajmo, da obstaja nevarnost, da je makro `_FIBO_H_` pred mestom prve vključitve datoteke `big-fibo-2.h` že definiran. Programski jezik C za tak primer nima drugega odgovora kot odgovorno programiranje — programer mora žal sam zagotoviti, da do takega primera ne pride.

57 Za konec si oglejmo še primer, ko je potrebno v deklaracijsko datoteko vstaviti deklaracijo podatkovnega tipa. Vzemimo primer enosmerno povezanih seznamov (celih števil).

Funkciji *insert_a* in *insert_z*, ki bi ju recimo definirali v datoteki `list.c`, bi želeli uporabljati tudi v drugih enotah programa. A za razliko od funkcije *fibonacci* ti dve funkciji uporabljata podatkovni tip *list*, ki v drugih enotah prevajanja ni znan — v kolikor ga tam ne deklariramo.

Da razrešimo to težavo in hkrati poenostavimo zapis programa, prenesemo celotno deklaracij podatkovnega tipa *list* v deklaracijsko datoteko `list.h`, ki jo vključimo ne le v druge enote prevajanja, temveč tudi v `list.c`. Deklaraciji funkcij *insert_a* in *insert_z*, ki se zato pojavita v enoti prevajanja `list.c`, ne motita prevajalnika.

```
⟨ list.h 57 ⟩ ==
#ifndef _LIST_H_
#define _LIST_H_
  typedef struct _node {
    int value;
    struct _node *next;
  } node;
  typedef struct _node *list;
  extern list insert_a(int val, list lst);
  extern list insert_z(int val, list lst);
#endif
```