Programiranje 1

Organizacija izvornog i izvršnog programa

Milena Vujošević Janičić

www.matf.bg.ac.rs/~milena

Programiranje 1

Pregled

Od izvornog do izvršnog programa

- Pisanje programa — prevođenje — izvršavanje
- Iako proces prevođenja jednostavnih programa početnicima izgleda kao jedan, nedeljiv proces, on se sastoji od više faza i podfaza (a od kojih se svaka i sama sastoji od više koraka).
- Osnovne faze prevođenja
  - Pretprocesiranje
  - Kompilacija
  - Povezivanje
- Tokom svih faza prevođenja, može biti otkrivena i prijavljena greška u programu.

Pregled

Pretprocesiranje

- Faza pretprocesiranja je pripremna faza kompilacije.
- Kompilator ne obrađuje tekst programa koji je napisao programer, već tekst programa koji je nastao pretprocesiranjem
- Jedan od najvažnijih zadataka pretprocesora je da omogući da se izvorni kôd pogodno organizuje u više ulaznih datoteka.
- Pretprocesor izvorni kôd iz različitih datoteka objedinjava u tzv. *jedinice prevođenja* i prosleđuje ih kompilatoru.
- Na primer, u `.c` datoteke koje sadrže izvorni kôd uključuju se *datoteke zaglavlja*
- Rezultat rada pretprocesora, može se dobiti korišćenje GCC prevodioca navođenjem opcije `-E` (na primer, `gcc -E program.c`).

Pretprocesor

- Suštinski, pretprocesor vrši samo jednostavne operacije nad tekstualnim sadržajem programa i ne koristi nikakvo znanje o jeziku C. Pretprocesor ne analizira značenje naredbi napisanih u jeziku C već samo *pretprocesorske direktive*
- Dve najčešće korišćene pretprocesorske direktive su `#include` (za uključivanje sadržaja neke druge datoteke) i `#define` koja zamenjuje neki tekst, *makro*, drugim tekstom.
- Pretprocesor omogućava i definisanje makroa sa argumentima, kao i uslovno prevođenje (određeni delovi izvornog programa se prevode samo ukoliko su ispunjeni zadati uslovi).

include

- U datoteku koja se prevodi, sadržaj neke druge datoteke uključuje se direktivom `#include`.

  ```
  #include "ime_datoteke"
  ili
  #include <ime_datoteke>
  ```

- U prvom slučaju, datoteka koja se uključuje traži se u okviru posebnog skupa direktorijuma *include path* (koja se većini kompilatora zadaje korišćenjem opcije `-I`) i koja obično podrazumevano sadrži direktorijum u kojem se nalazi datoteka u koju se vrši uključivanje.
- Ukoliko je ime, kao u drugom slučaju, navedeno između znakova < i >, datoteka se traži u sistemskom *include* direktorijumu u kojem se nalaze standardne datoteke zaglavlja, čija lokacija zavisi od sistema i od C prevodioca koji se koristi.

define

- Pretprocesorska direktiva `#define` omogućava zamenjivanje niza karaktera u datoteci, *makroa* (engl. macro) drugim nizom karaktera pre samog prevođenja. Njen opšti oblik je:

```
#define originalni_tekst novi_tekst
```

- U najjednostavnijem obliku, ova direktiva koristi se za zadavanje vrednosti nekom simboličkom imenu, na primer:

```
#define MAX_LEN 80
```

`define`

- Ovakva definicija se koristi da bi se izbeglo navođenje iste konstantne vrednosti na puno mesta u programu.
- Simboličko ime se može lako promeniti — izmenom na samo jednom mestu.
- Simboličko ime nikako ne treba mešati sa promenljivom (čak ni sa promenljivom koja je `const`).
- Za ime `MAX_LEN` u memoriji se ne rezerviše prostor tokom izvršavanja programa, već se svako njeno pojavljivanje, pre samog prevođenja zamenjuju zadatom vrednošću (u navedenom primeru — vrednošću 80).

define

- Zamene se ne vrše u konstantnim niskama niti u okviru drugih simboličkih imena
- Na primer, direktiva `#define MAX_LEN 80` neće uticati na naredbu `printf( "MAX_LEN is 80");` niti na simboličko ime `MAX_LEN_VAL`.

define

- Moguće je definisati i pravila zamene sa argumentima od kojih zavisi tekst zamene. Na primer, sledeća definicija

  ```
  #define  max(A, B)  ((A) > (B) ? (A) : (B))
  ```

  definiše tekst zamene za `max(A, B)` koji zavisi od argumenata.
- Tekst `max(x+2, 3*y)` biće zamenjen tekstom `((x+2) > (3*y) ? (x+2) : (3*y))`.

define

- Tekst `max(2, 3)` i slični ne predstavljaju poziv funkcije i nema nikakvog prenosa argumenata kao kod pozivanja funkcija.
- Šta se dešava za `max(a++, b++)` ?

define

- Tekst `max(2, 3)` i slični ne predstavljaju poziv funkcije i nema nikakvog prenosa argumenata kao kod pozivanja funkcija.
- Šta se dešava za `max(a++, b++)` ?
  `((a++) > (b++) ? (a++) : (b++))`

define

- Važno je voditi računa i o zagradama u tekstu zamene, da bi bio očuvan poredak primene operacija.

  ```
  #define kvadrat(x)  x*x
  ```

  primeni na `kvadrat(a+2)`, tekst zamene će biti `a+2*a+2`, a ne `(a+2)*(a+2)`, kao što je verovatno željeno i zbog toga bi trebalo koristiti:

  ```
  #define kvadrat(x)  (x)*(x)
  ```

define

- Navedena definicija, međutim, i dalje ne daje ponašanje koje je verovatno željeno.

define

- Navedena definicija, međutim, i dalje ne daje ponašanje koje je verovatno željeno.
- Naime, kada se makro primeni na izraz `a/kvadrat(b)`, on će biti zamenjen izrazom `a/(b)*(b)`, što je ekvivalentno sa `(a/b)*b` (a ne sa sa `a/(b*b)`). Zbog toga je bolja definicija:

```
#define kvadrat(x)  ((x)*(x))
```

Kompilacija

- *leksička analiza* — izdvajanje *leksema*, osnovnih jezičkih elemenata;
- *sintaksička analiza* — kreiranje sintaksnog stabla;
- *semantička analiza* — provera semantike i transformacija kreiranog stabla;
- *generisanje međukôda* — generisanje kôda na jeziku interne reprezentacije;
- *optimizacija međukôda* — optimizovanje generisanog kôda;
- *generisanje kôda na mašinskom jeziku* — prevođenje optimizovanog kôda u objektne module.

Kompilacija

- Kompilacijom se od svake jedinice prevođenja gradi zasebni *objektni modul* (engl. object module)
- Objektni moduli sadrže programe (mašinski kôd funkcija) i podatke (memorijski prostor rezervisan za promenljive).
- Iako su u mašinskom obliku, objektni moduli se ne mogu izvršavati.

Povezivanje

- Povezivanje je proces kreiranja jedinstvene izvršne datoteke od jednog ili više objektnih modula koji su nastali ili kompilacijom izvornog kôda programa ili su objektni moduli koji sadrže mašinski kôd i podatke standardne ili neke nestandardne biblioteke
- Pored *statičkog povezivanja*, koje se vrši nakon kompilacije, postoji i *dinamičko povezivanje*, koje se vrši tokom izvršavanja programa (zapravo na njegovom početku).
- Opcijama kompajlera, kompilaciju i povezivanje je moguće razdvojiti

Punilac

- Nakon uspešnog prevođenja može da sledi faza izvršavanja programa.
- Izvršni programi smešteni su u datotekama na disku i pre pokretanja učitavaju se u glavnu memoriju.
- Za ovo je zadužen program koji se naziva *punilac* ili *louder* (engl. loader) koji je obično integrisan u operativni sistem.
- Njegov zadatak je da proveri da li korisnik ima pravo da izvrši program, da prenese program u glavnu memoriju, da prekopira argumente komandne linije u odgovarajući deo memorije programa koji se pokreće, da inicijalizuju određene registre u procesoru i na kraju da pozovu početnu funkciju programa.

Pregled

Organizacija izvornog koda

- Problemi koje ste do sada rešavali obuhvatali su male i jednostavne programe koji su bili precizno definisani
- Za organizovanje složenih programa neophodno je
  - napraviti funkcionalnu dekompoziciju problema, tj. podelu složenih zadataka na jednostavnije poslove i izdvajanje u zasebne funkcije
  - definisanje odgovarajućih tipova podataka i organizovanje podataka definisanjem odgovarajućih promenljivih.
- Primer: matematički softver

Doseg, životni vek i povezanost

- Postoj tri važna pojma: doseg, životni vek i povezanost
- Odnos između dosega, životnog veka i povezanosti je veoma suptilan i sva ova tri aspekta objekata određuju se na osnovu mesta i načina deklarisanja tj. definisanja objekata, ali i primenom kvalifikatora `auto`, `register`, `static` i `extern`

Doseg identifikatora

- Podelu koda po datotekama omogućavaju različiti mehanizmi
- *Doseg identifikatora* (engl. scope) određuje da li se neka imena mogu koristiti u čitavim jedinicama prevođenja ili samo u njihovim manjim delovima (najčešće funkcijama ili još užim blokovima).
- Postojanje promenljivih čija je upotreba ograničena na samo određene uske delove izvornog kôda olakšava razumevanje programa i smanjuje mogućnost grešaka i smanjuje međusobnu zavisnost između raznih delova programa.

Doseg identifikatora

- *doseg nivoa datoteke* (engl. file level scope) koji podrazumeva da ime važi od tačke uvođenja do kraja datoteke;
- *doseg nivoa bloka* (engl. block level scope) koji podrazumeva da ime važi od tačke uvođenja do kraja bloka u kojem je uvedeno;
- *doseg nivoa funkcije* (engl. function level scope) koji podrazumeva da ime važi u celoj funkciji u kojoj je uvedeno; ovaj doseg imaju jedino labele koje se koriste uz `goto` naredbu;
- *doseg nivoa prototipa funkcije* (engl. function prototype scope) koji podrazumeva da ime važi u okviru prototipa (deklaracije) funkcije; a.

Doseg identifikatora

- Najznačajni nivoi dosega su doseg nivoa datoteke i doseg nivoa bloka.
- Identifikatori koji imaju doseg nivoa datoteke najčešće se nazivaju *spoljašnji* ili *globalni*
- Identifikatori koji imaju ostale nivoe dosega (najčešće doseg nivoa bloka) nazivaju *unutrašnji* ili *lokalni*

Primer

```
int a; /* a je globalna promenljiva - doseg nivoa datoteke */

void f(int c) { /* f je globalna funkcija - doseg nivoa datoteke */
          /* c je lokalna promenljiva - doseg nivoa bloka (tela funkcije f) */
  int d;  /* d je lokalna promenljiva - doseg nivoa bloka (tela funkcije f) */

  void g() { printf("zdravo"); }
  /* g je lokalna funkcija - doseg nivoa bloka (tela funkcije f) */

  for (d = 0; d < 3; d++) {
     int e; /* e je lokalna promenljiva - doseg nivoa bloka (tela petlje) */
     ...
  }
  kraj:   /* labela kraj - doseg nivoa funkcije */
}

/* h je globalna funkcija - doseg nivoa datoteke */
void h(int b); /* b - doseg nivoa prototipa funkcije */
```

Primer

```
void f() {
    int a = 3, i;
    for (i = 0; i < 4; i++) {
      int a = 5;
      printf("%d ", a);
    }
}
```

Životni vek promenljivih

- U određenoj vezi sa dosegom, ali ipak nezavisno od njega je pitanje trajanja objekata (pre svega promenljivih).
- *Životni vek (engl. storage duration, lifetime)* promenljive je deo vremena izvršavanja programa u kojem se garantuje da je za tu promenljivu rezervisan deo memorije i da se ta promenljiva može koristiti.
- Postojanje promenljivih koje traju samo pojedinačno izvršavanje neke funkcije znatno štedi memoriju, dok postojanje promenljivih koje traju čitavo izvršavanje programa omogućava da se preko njih vrši komunikacija između različitih funkcija modula.

Životni vek promenljivih

- U jeziku C postoje sledeće vrste životnog veka:
  - *statički (engl. static)* životni vek koji znači da je objekat dostupan tokom celog izvršavanja programa;
  - *automatski (engl. automatic)* životni vek koji najčešće imaju promenljive koje se automatski stvaraju i uklanjaju prilikom pozivanja funkcija;
  - *dinamički (engl. dynamic)* životni vek koji imaju promenljive koje se alociraju i dealociraju na eksplicitan zahtev programera.
- Životni vek nekog objekta se određuje na osnovu pozicije u kôdu na kojoj je objekat uveden i na osnovu eksplicitnog korišćenja nekog od kvalifikatora `auto` (automatski životni vek) ili `static` (statički životni vek).

Životni vek promenljivih

- Lokalne promenljive podrazumevano su *automatskog* životnog veka (sem ako je na njih primenjen kvalifikator `static` ili kvalifikator `extern`.
- Iako je moguće i eksplicitno okarakterisati životni vek korišćenjem kvalifikatora `auto`, to se obično ne radi jer se podrazumeva.
- Početna vrednost lokalnih automatskih promenljivih nije određena.

Životni vek promenljivih

- Automatske promenljive „postoje“ samo tokom izvršavanja funkcije u kojoj su deklarisane
- Formalni parametri funkcija, tj. promenljive koje prihvataju argumente funkcije imaju isti status kao i lokalne automatske promenljive.
- Na lokalne automatske promenljive i parametre funkcija moguće je primeniti i kvalifikator `register`

Statički životni vek

- Globalne promenljive deklarisane su van svih funkcija i njihov doseg je nivoa datoteke tj. mogu se koristiti od tačke uvođenja, u svim funkcijama do kraja datoteke.
- Životni vek ovih promenljivih uvek je *statički*, tj. prostor za ove promenljive rezervisan je tokom celog izvršavanja programa: prostor za njih se rezerviše na početku izvršavanja programa i oslobađa onda kada se završi izvršavanje programa.
- Prostor za ove promenljive obezbeđuje se u segmentu podataka.
- Ovakve promenljive se podrazumevano inicijalizuju na vrednost 0 (ukoliko se ne izvrši eksplicitna inicijalizacija).

Lokalne statičke promenljive

- U nekim slučajevima poželjno je čuvati informaciju između različitih poziva funkcije (npr. potrebno je brojati koliko puta je pozvana neka funkcija).
- Jedno rešenje bilo bi uvođenje globalne promenljive, statičkog životnog veka, međutim, zbog globalnog dosega tu promenljivu bilo bi moguće koristiti (i promeniti) i iz drugih funkcija, što je nepoželjno.
- Zato je poželjna mogućnost definisanja promenljivih koje bi bile statičkog životnog veka (da bi čuvale vrednost tokom čitavog izvršavanja programa), ali lokalnog dosega (da bi se mogle koristiti i menjati samo u jednoj funkciji).

Lokalne statičke promenljive

- U deklaraciji lokalne promenljive može se primeniti kvalifikator `static` i u tom slučaju ona ima statički životni vek — kreira se na početku izvršavanja programa i oslobađa prilikom završetka rada programa.
- Tako modifikovana promenljiva ne čuva se u stek okviru svoje funkcije, već u segmentu podataka.

Lokalne statičke promenljive

- Ukoliko se vrednost statičke lokalne promenljive promeni tokom izvršavanja funkcije, ta vrednost ostaje sačuvana i za sledeći poziv te funkcije.
- Ukoliko inicijalna vrednost statičke promenljive nije navedena, podrazumeva se vrednost `0`.
- Statičke promenljive se inicijalizuju samo jednom, konstantnim izrazom, na početku izvršavanja programa tj. prilikom njegovog učitavanja u memoriju.
- Doseg ovih promenljivih i dalje je nivoa bloka tj. promenljive su i dalje lokalne, što daje željene osobine.

Primer

```
#include <stdio.h>

void f() {
  int a = 0;
  printf("f: %d ", a);
  a = a + 1;
}

void g() {
  static int a = 0;
  printf("g: %d ", a);
  a = a + 1;
}

int main() {
  f(); f(); g(); g();
  return 0;
}
```

Povezanost identifikatora

- *Povezanost identifikatora* (engl. linkage) u vezi je sa deljenjem podataka između različitih jedinica prevođenja i daje mogućnost korišćenja zajedničkih promenljivih i funkcija u različitim jedinicama prevođenja (modulima), ali i mogućnost sakrivanja nekih promenljivih tako da im se ne može pristupiti iz drugih jedinica prevođenja.

Povezanost

Jezik C razlikuje identifikatore:

- *bez povezanosti* (engl. no linkage),
- identifikatore sa *spoljašnjom povezanošću* (engl. external linkage) i
- identifikatore sa *unutrašnjom povezanošću* (engl. internal linkage).

Identifikatori bez povezanosti

- Bez povezanosti su najčešće lokalne promenljive (bez obzira da li su automatskog ili statičkog životnog veka), parametri funkcija, korisnički definisani tipovi, labele itd.

```
int f() {
   int i;
   ...
}
```

```
int g1(int i) {
   static int j;
}
int g2() {
   static int i, j;
   ...
}
```

```
struct i { int a; }
```

Spoljašnja povezanost i kvalifikator `extern`

- Spoljašnja povezanost identifikatora omogućava da se isti objekat koristi u više jedinica prevođenja.
- Sve deklaracije identifikatora sa spoljašnjom povezanošću u skupu jedinica prevođenja određuju jedan isti objekat (tj. sve pojave ovakvog identifikatora u različitim jedinicama prevođenja odnose se na jedan isti objekat), dok u celom programu mora da postoji tačno jedna definicija tog objekta.
- Tako se jedan identifikator koristi za isti objekat u okviru više jedinica prevođenja.

extern

- Kvalifikator `extern` najčešće se koristi isključivo kod programa koji se sastoje od više jedinica prevođenja i služi da naglasi da neki identifikator ima spoljašnju povezanost.
- Jedino deklaracije mogu biti okvalifikovane kvalifikatorom `extern`.
- Samim tim, nakon njegovog navođenja nije moguće navoditi inicijalizaciju promenljivih niti telo funkcije (što ima smisla samo prilikom definisanja).
- Pošto `extern` deklaracije nisu definicije, njima se ne rezerviše nikakva memorija u programu već se samo naglašava da se očekuje da negde (najčešće u drugim jedinicama prevođenja) postoji definicija objekta koji je se `extern` deklaracijom deklariše (i time uvodi u odgovarajući doseg).

extern

- Ipak, postoje slučajevi kada se spoljašnja povezanost podrazumeva i nije neophodno eksplicitno upotrebiti `extern` deklaraciju.
- Sve globalne funkcije i promenljive jezika C podrazumevano imaju spoljašnju povezanost (osim ako na njih nije primenjen kvalifikator `static` kada je povezanost unutrašnja).

Primer

```
#include <stdio.h>
int a;
int b = 3;


void f() {
  printf("a=%d\n", a);
}
```

```
#include <stdio.h>
extern int a;
void f();

int g() {
  extern int b;
  printf("b=%d\n", b);
  f();
}

int main() {
  a = 1;
  /* b = 2; */
  g();
  return 0;
}
```

Unutrašnja povezanost i kvalifikator `static`

- Globalni objekat ima unutrašnju povezanost ako se kvalifikuje kvalifikatorom `static`
- Osnovna uloga unutrašnje povezanosti obično je u tome da neki deo kôda učini zatvorenim, u smislu da je nemoguće menjanje nekih njegovih globalnih promenljivih ili pozivanje nekih njegovih funkcija iz drugih datoteka.

Primer

```
#include <stdio.h>
static int a = 3;
int b = 5;

static int f() {
  printf("f: a=%d, b=%d\n",
         a, b);
}

int g() {
  f();
}
```

```
#include <stdio.h>

int g(); /* int f(); */
int a, b;

int main() {
   printf("main: a=%d, b=%d\n",
   a, b);
   g(); /* f() */
   return 0;
}
```

Pregled

Organizacija memorije izvršnog programa

- Način organizovanja i korišćenja memorije u fazi izvršavanja programa može se razlikovati od jednog do drugog operativnog sistema.
- Kada se izvršni program učita u radnu memoriju računara, biva mu dodeljena određena memorija i započinje njegovo izvršavanje.
- Dodeljena memorija organizovana je u nekoliko delova koje zovemo segmenti ili zone:
  - segment kôda (engl. code segment ili text segment);
  - segment podataka (engl. data segment);
  - stek segment (engl. stack segment);
  - hip segment (engl. heap segment).

Segmenti u memoriji

Fon Nojmanova arhitektura računara predviđa da se u memoriji čuvaju podaci i programi.

- Segment kôda — u ovom segmentu nalazi se sâm izvršni kôd programa, ukoliko se pokrene više instanci jednog programa, ovaj segment može da bude zajednički za sve instance (zavisi od operativnog sistema)
- Segment podataka — u ovom segmentu čuvaju se određene vrste promenljivih koje su zajedničke za ceo program: globalne promenljive, promenljive koje imaju statički životni vek, kao i konstantni podaci (najčešće konstantne niske); ukoliko se pokrene više instanci jednog programa, svaka instanca ima svoj zaseban segment podataka
- Hip segment — u ovom segmentu nalaze se dinamički alocirani podaci

Stek segment

- Stek segment ili programski stek poziva (engl. call stack) čuva sve podatke koji karakterišu izvršavanje funkcija.
- Podaci koji odgovaraju jednoj instanci funkcije organizovani su u takozvani stek okvir (engl. stack frame).
- Stek okvir jedne instance funkcije, između ostalog, sadrži:
  - argumente funkcije;
  - lokalne promenljive (promenljive deklarisane unutar funkcije);
  - međurezultate izračunavanja;
  - adresu povratka (koja ukazuje na to odakle treba nastaviti izvršavanje programa nakon povratka iz funkcije);
  - adresu stek okvira funkcije pozivaoca.

Stek segment

- Veličina stek segmenta obično je ograničena.
- Zbog toga je poželjno izbegavati smeštanje jako velikih podataka na segment steka.

Primer

```
int main() {
int a[1000000];
...
}
```

```
int a[1000000];
int main() {
...
}
```

Stek segment

- Stek poziva je struktura tipa LIFO ("last in - first out") — stek okvir se može dodati samo na vrh steka i sa steka se može ukloniti samo okvir koji je na vrhu
- Stek okvir za instancu funkcije se kreira onda kada funkcija treba da se izvrši i taj stek okvir se oslobađa (preciznije, smatra se nepostojećim) onda kada se završi izvršavanje funkcije.

Stek segment

- Ako izvršavanje programa počinje izvršavanjem funkcije main, prvi stek okvir se kreira za ovu funkciju.
- Ako funkcija main poziva neku funkciju f, na vrhu steka, iznad stek okvira funkcije main, kreira se novi stek okvir za ovu funkciju.
- Ukoliko funkcija f poziva neku treću funkciju, onda će za nju biti kreiran stek okvir na novom vrhu steka.
- Kada se završi izvršavanje funkcije f, onda se vrh steka vraća na prethodno stanje i prostor koji je zauzimao stek okvir za f se smatra slobodnim (iako on neće biti zaista obrisan).

Stek segment

Nacrtati stek okvire za izvršavanje sledećeg koda:

Primer

```
int kub(int a) {
  return a*a*a;
}
int suma_kubova_i_uvecanje(int a[], int n) {
  int rezultat = 0, i;
  for(i=0; i<n; i++)
    rezultat += kub(a[i]++);
  return rezultat;
}
int main() {
  int a[3]={0, 1, 2}, s;
  s = suma_kubova_i_uvecanje(a, sizeof(a)/sizeof(int));
  printf("Rezultat je %d\n", s);
  return 0;
}
```

Pregled

Greške u programu

- I u fazi prevođenja i u fazi izvršavanja mogu se javiti greške i one moraju biti ispravljene da bi program funkcionisao ispravno.
- Greška se može javiti u bilo kojoj fazi prevođenja

Primer — pretprocesiranje

```
#Include<stdio.h>
int main() {
  int a = 9;
  if (a == 9)
     printf("9");
  return 0;
}
```

Primer — pretprocesiranje

```
#Include<stdio.h>
int main() {
  int a = 9;
  if (a == 9)
     printf("9");
  return 0;
}
```

```
primer.c:1:2: error: invalid preprocessing
                     directive #Include
```

Primer — kompilacija — leksicka analiza

```
#include<stdio.h>
int main() {
  int a = 09;
  if (a == 9)
     printf("9");
  return 0;
}
```

Primer — kompilacija — leksicka analiza

```
#include<stdio.h>
int main() {
  int a = 09;
  if (a == 9)
     printf("9");
  return 0;
}
```

```
primer.c:4:9: error: invalid digit "9" in octal constant
```

Primer — kompilacija — sintaksna analiza

```
#include<stdio.h>
int main() {
  int a = 9;
  if a == 9
     printf("9");
  return 0;
}
```

Primer — kompilacija — sintaksna analiza

```
#include<stdio.h>
int main() {
  int a = 9;
  if a == 9
     printf("9");
  return 0;
}
```

```
primer.c:5:6: error: expected ‘(’ before ‘a’
```

Primer — kompilacija — semantička analiza

```
#include<stdio.h>
int main() {
  int a = 9;
  if ("a" * 9)
     printf("9");
  return 0;
}
```

Primer — kompilacija — semantička analiza

```
#include<stdio.h>
int main() {
  int a = 9;
  if ("a" * 9)
     printf("9");
  return 0;
}
```

```
primer.c:5:8: error: invalid operands to binary *
                     (have ‘char *’ and ‘int’)
```

Primer — povezivanje

```
#include<stdio.h>
int main() {
  int a = 9;
  if (a == 9)
     print("9");
  return 0;
}
```

Primer — povezivanje

```
#include<stdio.h>
int main() {
  int a = 9;
  if (a == 9)
     print("9");
  return 0;
}
```

```
primer.c:(.text+0x20): undefined reference to ‘print’
collect2: ld returned 1 exit status
```

Primer — upozorenja

```
#include<stdio.h>
int main() {
  int a = 9;
  if (a = 9)
     printf("9");
  return 0;
}
```

Primer — upozorenja

```
#include<stdio.h>
int main() {
  int a = 9;
  if (a = 9)
     printf("9");
  return 0;
}
```

```
primer.c:4: warning: suggest parentheses around assignment
            used as truth value
```

Primer — upozorenja

```
#include<stdio.h>
int main() {
  int a = 9;
  if (a / 0)
     printf("9");
  return 0;
}
```

Primer — upozorenja

```
#include<stdio.h>
int main() {
  int a = 9;
  if (a / 0)
     printf("9");
  return 0;
}

primer.c:4: warning: division by zero
```

Primer — upozorenja

```
#include<stdio.h>
int main() {
  int a = 9;
  if (9 == 9)
     printf("9");
  return 0;
}
```

Primer — upozorenja

```
#include<stdio.h>
int main() {
  int a = 9;
  if (9 == 9)
     printf("9");
  return 0;
}
```

```
primer.c:3: warning: unused variable ‘a’
```

Greške u programu

- I u fazi prevođenja i u fazi izvršavanja mogu se javiti greške i one moraju biti ispravljene da bi program funkcionisao ispravno.
- Greške tokom izvršavanja mogu biti takve da program ne može dalje da nastavi sa izvršavanjem, ali mogu da budu takve da se program nesmetano izvršava, ali da su rezultati koje prikazuje pogrešni.
- Dakle, to što ne postoje greške u fazi prevođenja i to što se program nesmetano izvršava, još uvek ne znači da je program ispravan, tj. da zadovoljava svoju specifikaciju i daje tačne rezultate za sve vrednosti ulaznih parametara.

Greške u programu

- Ispravnost programa u tom, dubljem smislu zahteva formalnu analizu i ispitivanje te vrste ispravnosti najčešće je van moći automatskih alata.
- Ipak, sredstva za prijavljivanje grešaka tokom prevođenja i izvršavanja programa znatno olakšavaju proces programiranja i često ukazuju i na suštinske propuste koji narušavaju ispravnost programa.

Pronalaženje grešaka

- Pre konačne distribucije korisnicima, program se obično intenzivno testira tj. izvršava za različite vrednosti ulaznih parametara.
- Ukoliko se otkrije da postoji greška tokom izvršavanja (tzv. *bag* od engl. bug), vrši se pronalaženje uzroka greške u izvornom programu (tzv. *debagovanje*) i njeno ispravljanje.
- U pronalaženju greške mogu pomoći programi koji se nazivaju *debageri* i koji omogućavaju izvršavanje programa korak po korak (ili pauziranje njegovog izvršavanja u nekim karakterističnim tačkama), uz prikaz međuvrednosti promenljivih.

Literatura

Slajdovi su pripremljeni na osnovu materijala iz devetog poglavlja knjige:
Filip Marić, Predrag Janičić: Programiranje 1
Za pripremu ispita nisu dovoljni slajdovi, potrebno je koristiti knjigu!