Programski jezik C

Izvorni in izvršni program – Minimalni program – Skalarne spremenljivke – Skalarna polja in tekstovni nizi – Strukturirane spremenljivke – Logični testi in krmilni stavki – Zunanje funkcije – Izbrane zunanje funkcije – Lastne funkcije – Dinamična rezervacija pomnilnika – Standardni vhod in izhod – Datotečni vhod in izhod – Programski argumenti in stikala – Optimizacija – Predprocesiranje programa – Program v več datotekah – Prevajanje in povezovanje – Lastne knjižnice

Izvorni in izvršni program

Izvorni program je zaporedje smiselnih ukazov, ki jih napišemo z editorjem v tekstovno datoteko. Ukazi morajo biti taki, da so razumljivi prevajalnemu programu za jezik C, prevajalniku.

Prevajalnik je program, ki, ko ga poženemo iz ukazne vrstice, čita izvorni program, iz njega izdela izvršni program in ga zapiše v binarno datoteko. Izvršni program vsebuje ukaze, ki so razumljivi računalnikovemu procesorju. Nekateri teh ukazov so klici na druge izvršne programe v ustreznih knjižnicah.

Izvršni program zaženemo iz ukazne vrstice. Ukazna lupina, bash, ga naloži v pomnilnik. Če je potrebno, naloži tudi ustrezne knjižnice. Potem začne računalnik delati po naloženem programu.

Minimalni program

Minimalni program je zapisan v eni datoteki:

```
/* Minimal program */
int main(void)
{
   return 0;
}
```

Vse, kar je med oznakama /* in */, prevajalnik ignorira. To je komentar, namenjen le bralcu izvornega programa.

Program ima obliko "funkcije", sestavljene iz "glave" in "telesa" med znakoma { in }.

Glava vsebuje ime funkcije ter njene vhodne in izhodne argumente. Ime funkcije mora biti main. Vhodni argument void pomeni, da funkcija main pri zagonu ne pričakuje ukaznovrstičnih parametrov. Izhodni argument int pomeni, da bo funkcija ob zaključku dostavila lupini celoštevilčni indikator, kako je svoje delo opravila.

Telo vsebuje zaporedje ukazov, v zgornjem primeru le enega. Vsak ukaz se zaključi s podpičjem. Ukaz return pomeni, naj funkcija konča z delom in v lupinsko spremenljivko ? zapiše vrednost 0.

Skalarne spremenljivke

Lokacije v prevedenem programu rezerviramo in poimenujemo z ukazi:

```
char    c;
int     i, j;
float   x, y, z;
```

S char deklariramo spremenljivko v dolžini 1 zlog, v katero bomo spravljali cela števila -127..127. Int deklarira 4 zloge za cela čtevila -32767..32767, float pa 8 zlogov za realna števila na intervalu +-1e-37..1e37 (z natančnostjo na 6 cifer). Na nekaterih računalnikih so rezervirane dolžine daljše.

Ime spremenljivke lahko vsebuje črke in številke. Na prvem mestu mora biti črka. Ime ne sme presegati 31 znakov. Prevajalnik razlikuje male in velike črke. Vse spremenljivke, ki jih bomo v programu uporabljali, morajo biti predhodno deklarirane.

Vsebina deklarirane spremenljivke je sprva neznana. V programu vanje shranjujemo vrednosti:

```
c = 100;
i = 1000;
x = 3.14;
y = 5.6e-5;
```

Spremenljivke polnimo tudi iz drugih že napolnjenih spremenljivk ali njihovih matematičnih izrazov; precedenčna pravila so ista, kot v matematiki: najprej množenje in deljenje, nato seštevanje in odštevanje.

Drugače uporabimo oklepaje.

```
j = i;
z = (1+x)*(1-x)/y;
```

Aritmetični operatorji:

```
+   seštevanje
-   odštevanje
*   množenje
/   deljenje
    dve celi števili dasta celoštevilčni kvocient
%   modulus
    dve celi števili dasta celoštevilčni modulus
```

Vrednost izraza mora biti regularna: deljenje z 0 ni dovoljeno; vrednost spremenljivke ali izraza mora biti znotraj veljavnega območja; desna stran mora biti enakega tipa kot leva, oziroma njegov podtip (float <- int <- char). Matematične funkcije (glej nadaljevanje) vračajo tip double, ki je nadtip tipa float z več številkami v mantisi in eksponentu. Double se pri zapisu v float ustrezno reducira.

Logični izrazi imajo vrednost 1 (true) ali 0 (false);

```
i = (x < y);
```

Relacijski in logični operatorji:

```
<       manjši
<=      manjši ali enak
>       večji
>=      večji ali enak
==      enak
!=      neenak
&&      logični AND
||      logični OR
```

Relacijski operatorji so nižje prioritete kot aritmetični. Logična operatorja sta nižje prioritete kot relacijski.

Skalarna polja in tekstovni nizi

Enodimenzionalno polje N skalarjev, recimo float, rezerviramo in poimenujemo z ukazom:

```
    float   a[N];
```

Posamezni elementi so numerirani od 0 do N-1. Prvi element je dosegljiv kot a[0] in zadnji kot a[N-1]. Posamezne elemente uporabljamo prav tako kot navadne skalarne spremenljivke. Posebno skalarno polje je enodimenzionalni niz tipa char, tekstovni niz:

```
    char s[N];
```

Elemente polnimo na poseben način:

```
    s[1] = 'A';
```

Prevajalnik pretvori črko A po kodu ASCII v 65. Nek element mora biti enak '\0' in s tem označuje konec teksta v nizu.

Dvodimenzionalno polje z M x N elementov:

```
    float   a[M][N];
```

Numeriranje, dostopnost in uporaba posameznih elementov je analogna.

Strukturirane spremenljivke

Standardni skalarni tipi spremenljivk so char, int in float. Iz njih lahko sestavim poljuben nov strukturiran tip, na primer tip complex:

```
    typedef struct {
        float   re;
        float   im;
    } complex;
```

Spremenljivko tipa complex deklariram potem kot

```
    complex z;
```

Posamičen element spremenljivke z je dostopen, na primer, kot z.re. Elemente uporabljamo kot navadne skalarne spremenljivke. Strukturno spremenljivko lahko v celoti kopiramo v drugo.

Poleg skalarnih sestavnih delov lahko struktura vsebuje tudi

polje ali drugo strukturo. Tvorimo lahko tudi polje struktur.

Logični testi in krmilni stavki

Ukazi v funkciji se izvajajo zaporedno. Tok lahko uravnavamo s krmilnimi stavki.

```
/* N-kratno ponavljanje */
for (i=1; i<=N; i=i+1) {
    …
}
```

Enkrat samkrat, na začetku, se izvede prvi del i=1. Potem se testira i<=N. Če je true, se izvede telo, sicer preskoči. Na koncu izvedbe telesa se izvede tretji del i=i+1 in ponovi test drugega dela.

```
/* Pogojno izvajanje */
if (BOOL) {
    …
}
```

Testira se logični izraz BOOL, recimo x < 10. Če je true, se izvede telo, sicer preskoči.

```
/* Pogojna razvejitev v dve veji */
if (BOOL) {
    …
}
else {
    …
}
```

Testira se BOOL. Če je true, se izvede prvo telo, sicer drugo.

```
/* Pogojna razvejitev v več vej */
if (BOOL1) {
    …
}
else if (BOOL2) {
    …
}
else {
    …
}
```

Testira se BOOL1. Če je true, se izvede prvo telo, sicer se testira BOOL2 in ustrezno ravna naprej.

```
/* Pogojno ponavljanje */
while (BOOL) {
    …
}
```

Testira se BOOL. Če je true, se izvede telo in ponovno testira izraz BOOL. Če je false, se telo preskoči in nadaljuje s prvim naslednjim ukazom.

```
/* Neskončno ponavljanje s prekinitvijo */
for (;;) {
    …
    if (BOOL) break;
    …
}
```

Zunanje funkcije

Program lahko uporablja že prevedeno kodo, zapisano v knjižnici libc ali kaki drugi. To so zunanje funkcije.

```
/* Program with external functions */
double sin(double);
double modf(double, double*);
int strcpy(char*, char*);
int main(void)
{
    float   x, y, z;
    char    s[10];
    x = 3.14;
    y = sin(x);        /* Return sine of x */
    z = modf(y,&x);    /* Return fractional part of y,
                          store integer part in x */
    strcpy(s,"HELLO"); /* Copy HELLO to s */
}
```

Funkcija prejema od glavnega programa podatke preko svojih argumentov, zapisanih med (in). Glavni program funkciji posreduje kopijo vsebine skalarne spremenljivke (x) ali njen dejanski naslov (&x). Funkcija potem s tem računa. Glavnemu programu vrne svojo vrednost, hkrati pa lahko spremeni tudi vsebino naslovljene spremenljivke. Funkcije vračajo le

skalarne vrednosti. Glavni program jih lahko shrani ali ignorira. Skalarne spremenljivke lahko posredujemo funkciji kot kopije ali kot naslove, druge spremenljivke pa zgolj kot naslove prvega elementa. Pri tem velja sintaktična olajšava dvomljive sorte, da namreč naslov vektorske spremenljivke &s[0] posredujemo kot s, ne kot &s.

Glave vseh klicanih funkcij – funkcijski prototipi – morajo biti zapisane na začetku programa. Tako prevajalnik pri vsakem klicu funkcije lahko preveri, če je klic pravilen.

Izbrane zunanje funkcije

Funkcije za delo s sistemom:

```
system("ls")   izvede lupinski ukaz ls
```

Funkcije za delo z nizi. Argumenta s in t sta niza:

```
strcat(s,t)     doda t na konec s
strcmp(s,t)     vrne 0, če enaka
strcpy(s,t)     kopira t v s
strlen(s)       vrne dolžino s
```

Matematične funkcije. Vsi argumenti so double, vse funkcije vračajo double:

```
sin(x)      sinus
cos(x)      kosinus
tan(x)      tangens
asin(x)     arcus sinus
acos(x)     arcus kosinus
atan(x)     arcus tangens
exp(x)      eksponentna funkcija
log(x)      logaritem
log10(x)    desetiški logaritem
pow(x,y)    x^y
sqrt(x)     kvadratni koren
fabs(x)     absolutna vrednost
```

Lastne funkcije

Večji program je iz očitnih razlogov sestavljen iz funkcije main in iz zaporedja drugih funkcij, ki jih main uporablja (kliče) neposredno ali posredno (iz drugih

funkcij). Vrstni red ni predpisan. Smiselno je postaviti main na prvo mesto. Izvajati se začne main.

```
/* Program with internal functions*/
float funct1(float, float);
int funct2(float*, float);
int funct3(float*, int);
int funct4(float**, int, int);
int main(void)
{
   int n;
   float a, b, c;
   float A[10], B[10][10];
   n = 10; a = 1.0; b = 1.0;
   A[1] = 1.0; B[1][1] = 1.0;
   c = funct1(a,b);    /* c becomes 2.0 */
   funct2(&a,b);       /* a becomes 2.0 */
   funct3(A,n);        /* A[1] becomes 2.0 */
   funct4(B,n,n);      /* B[1][1] becomes 2.0 */
   return 0;
}
float funct1(float x, float y)
{
    float z;
    z = x+y;
    return z;
}
int funct2(float* x, float y)
{
    *x = *x+y;
    return 0;
}
int funct3(float* X, int N)
{
    X[1] = X[1]+1.0;
    return 0;
}
int funct4(float** X, int M, int N)
{
    X[1][1] = X[1][1]+1.0;
    return 0;
}
```

Glave vseh klicanih funkcij – funkcijski prototipi – morajo biti zapisane na začetku programa.

Deklaracije spremenljivk znotraj funkcije so poznane le

znotraj funkcije, so lokalne.

Glavni program ob klicu pošlje funkciji funct1 kopijo vsebine svoje spremenljivke a, ki se zapiše v funkcijino spremenljivko x. Originalna spremenljivka a funkciji ni dostopna. Če funkcija spremeni vsebino x, se vsebina a ne spremeni.

Glavni program funkciji funct2 pošlje naslov svoje spremenljivke a, in sicer kot &a. Ta naslov je neko kardinalno število. Funkcija ga shrani v svojo kazalčno spremenljivko x. Originalna spremenljivka a je sedaj dostopna funkciji, in sicer kot *x (Wirth bi jo naredil dostopno kar kot x). Če funkcija spremeni vsebino *x, s tem spremeni vsebino a.

Funkciji funct3 posredujemo enodimenzionalno skalarno polje lahko le kot naslov prvega elementa &A[0]. Okrajšamo ga lahko v A (Wirth bi ga okrajšal v &A). Originalni element A[i] je dosegljiv kot X[i].

Funkciji funct4 posredujemo dvodimenzionalno skalarno polje lahko le kot naslov prvega elementa &B[0][0]. Okrajšamo ga lahko v B. Originalni element B[j][i] je dosegljiv kot X[j][i].

Funkcija programu main vrača vrednost preko svojega stavka return. Vrnjene vrednosti lahko ignoriramo. Funkcija je lahko brez stavka return; tedaj jo deklariramo kot tipa void.

Dinamična rezervacija pomnilnika

Ni treba, da že prevajalniku povemo, koliko pomnilnika naj rezervira za spremenljivke. To lahko naredimo šele v času izvajanja programa; prostor rezerviramo, uporabimo in sprostimo.

```
float*    p;
p = (float*)malloc(sizeof(float));
if (p == NULL) {
    printf("Error: Out of memory\n");
    exit(1);
}
*p = 1.0;
free(p);
```

Spremenljivka p je kazalec na spremenljivko tipa float, ki jo želimo ustvariti. Uporabimo lahko poljuben skalarni tip ali strukturo. Funkcija malloc rezervira del pomnilnika s pravšnjo dolžino. Naslov novoustvarjene spremenljivke vrne v p. Če pomnilnika ni možno rezervirati, ker ga morda ni na voljo, je vrnjeni naslov NULL (0) in program se prekine. Na novo ustvarjena sprmenljivka je dostopna kot *p. Na koncu rezervirani pomnilnik sprostimo s funkcijo free.

Rezerviramo lahko vektor n elementov tipa float ali kakega drugega skalarnega tipa ali strukture:

```
float*    p;
n = 10;
p = (float*)calloc(n,sizeof(float));
if (p == NULL) {
    printf("Error: Out of memory\n");
    exit(1);
}
p[1] = 1.0;
free(p);
```

Standardni vhod in izhod

Posamezen znak pošljemo v zaslonov medpomnilnik (buffer) kot putchar(c). Tam se znaki nabirajo. Na zaslon se izpišejo, ko v medpomnilnik pošljemo znak '\n'.

Posamezen znak čitamo iz medpomnilnika tipkovnice kot c = getchar(). Znaki v medpomnilniku postanejo dostopni, ko odtipkamo ENTER.

Niz znakov pošljemo na zaslon kot puts(s). Funkcija avtomatsko doda '\n'.

Niz znakov čitamo s tipkovnice kot gets(s). Prebrani '\n' se nadomesti z '\0'.

Nize in števila kot nize izpisujemo na zaslon s funkcijo

```
printf("%s %d %f %e\n",c,s,i,x,y);
```

Funkcija izpiše niz s, celo število i in realno število x, vse v standardnem formatu. Med sabo jih loči s presledki. Na koncu doda '\n'. Formate lahko določimo tudi bolj natančno:

```
    %5s     Minimalna širina polja 5 znakov
    %7d     Minimalna širina polja 7 znakov
    %8.2f   Min širina 8, od tega 2 decimalki
    %8.2e   Min širina 8, od tega 2 decimalki mantise
```

Vsi izpisi so desno poravnani v svojih poljih. Negativne formatirne številke, recimo %-8.2f, pomenijo levo poravnavo.

Nize in števila kot nize čitamo s tipkovnice s funkcijo

```
    scanf("%s %d %f %e",s,&i,&x,&y);
```

Funkcija čita, dokler zahteva format. Eno vhodno polje je zaporedje ne-belih znakov (beli znaki so SPACE, TAB, LF, CR, FF). To pomeni, da funkcija lahko čita vhod tudi v naslednji vrstici.

Namesto formatnega izpisa na zaslon lahko formatiramo v niz s in potem niz zapišemo na zaslon:

```
    sprintf(s,FORMAT,VARLIST);
    puts(s);
```

Namesto formatnega čitanja iz tipkovnice lahko čitamo v niz s in potem tega razčlenjujemo:

```
    gets(s);
    sscanf(s,FORMAT,VARLIST);
```

Datotečni vhod in izhod

```
    FILE*   f;
    FILE*   g;
    f = fopen("input.txt","r");
    g = fopen("output.txt","w");
    …
    fclose(f);
    fclose(g);
```

Spremenljivka f je kazalec na vhodno datoteko, g na izhodno. Prva datoteka je odprta kot tekstovna (znaki 0..127) za branje, "r". Druga je tekstovna za pisanje, "w". Branje in pisanje binarne datoteke (znaki -127..127) je možno z "rb" in "wb". Na koncu datoteki zapremo s fclose.

Branje in pisanje datotek po znakih:

```
char    c;
for (;;) {
    c = fgetc(f);
    if (c==EOF) break;
    fputc(c,g);
}
```

Čitamo zlog za zlogom s funkcijo fgetc. Vsak zlog sproti zapišemo na izhodno datoteko s putc. Končamo, ko v vhodnem toku znakov naletimo na znak EOF (tipično -1).

Branje in pisanje tekstovnih datotek po nizih:

```
char s[80];
for (;;) {
    fgets(s,sizeof(s),f);
    if (s==NULL) break;
    fputs(s,g);
}
```

Funkcija fgets prečita niz znakov do vključno prvega LF ali največ 79 znakov ter zapiše v s (brez LF). Funkcija fputs prepiše s na izhod in doda LF.

Formatirano branje in pisanje tekstovnih datotek:

```
int i;
float x;
for (;;) {
   i = fscanf(f,"%e\n",&x);
   if (i==EOF) break;
   fprintf(g,"%e\n",x);
}
```

Branje in pisanje niza poljubnih elementov:

```
int i;
complex z[N];
i = fread(z,sizeof(z),N,f);
i = fwrite(z,sizeof(z),N,g);
```

Funkcija prečita/zapiše N elementov tipa complex in vrne število prebranih/zapisanih elementov.

Programski argumenti in stikala

Pri klicu programa mu lahko damo parametre, recimo imena datotek, ki jih naj obdeluje.

```
int main(int argc, char** argv)
{
…
}
```

Število argumentov je v argc. Argumenti so nizi. Prvi niz, argv[0], vsebuje ime programa, ostali njegove ukaznovrstične parametre.

Optimizacija

Pišemo z berljivostjo kot prvim ciljem. Prireditvene stavke pišemo posebej in jih ne vključujemo v druge izraze.

Šele če prevedeni program ni dovolj hiter ali je prevelik, začnemo optimizirati. Najprej optimiziramo najbolj kritično točko in vidimo, ali zadostuje.

Iz zanke vzamemo vse operacije, ki se ne spreminjajo.

Uporabljamo poceni operacije namesto dragih:

Operacija	Relativna cena
printf, scanf	1000
malloc, free	800
trig function	500
float operation	100
integer * /	20
function call	10
integer + -	5
pointer dereference	2
&& || !	1

Predprocesiranje programa

Tekstovni predprocesor preoblikuje tekst, nakar ga posreduje

prevajalniku. Ukazne vrstice, namenjene predprocesorju, se začno z znakom #.

Vse nize MAX, ki jih najde v programu, nadomesti z nizom 100:

```
#define MAX 100
float x[MAX];
```

Na izbrano mesto vstavi funkcijske prototipe iz zunanje datoteke myfile.h v lokalnem ali kakem drugem imeniku:

```
#include "myfile.h"
#include "/home/local/include/myfile.h"
```

Na izbrano mesto vstavi funkcijske prototipe iz zunanje datoteke stdio.h v standardnem sistemskem imeniku:

```
#include <stdio.h>
```

Program v več datotekah

Večje programe je zaradi preglednosti in hitrejšega prevajanja smiselno zapisati v več izvornih datotek. V prvo datoteko gre glavni program, v drugo prototipi njegovih lastnih funkcij. Podprogrami se zapišejo v eno ali več dodatnih datotek; v vsaki je po ena ali več funkcij.

Glavni program uvaža (#include) datoteko s prototipi funkcij. Isto velja za vsako datoteko s funkcijami.
V datoteki prototipov je vsak prototip označen kot zunanji, na primer

```
extern float funct1(float, float);
```

Prevajanje in povezovanje

Program v eni datoteki:

```
bash> gcc -Wall -ansi -pedantic -lm -g -o EXEFILE SRCFILE
```

Stikalo -Wall povzroči izpisovanje vseh opozoril pri prevajanju. Stikali -ansi in -pedantic preverjata, ali je program skladen s standardom ANSI ter opozorita na neskladja. Stikalo -l povzroči povezovanje s knjižnico

libm.so. To je potrebno, če izvorni program vsebuje funkcije iz te knjižnice. Eksplicitno je treba navesti vse potrebne knjižnice razen osnovne, libc.so. Stikalo -g ukazuje, naj bodo v izvršnem programu vključeni dodatni ukazi, potrebni za iskanje in odpravljanje morebitnih napak (razhroščevaneje).

Program v več datotekah prevajamo po delih in na koncu povežemo. Najprej prevedemo datoteko(e) s podprogrami:

```
bash> gcc -c util.c
```

Naredi se knjižnica util.o. Nato prevedemo datoteko z glavnim programom:

```
bash> gcc -c main.c
```

Naredi se main.o. Na koncu vse skupaj povežemo:

```
bash> gcc -o main main.o util.o
```

Naredi se izvršni program main. Če kasneje kaj spremenimo v eni izmed datotek, je potrebno prevesti zgolj to datoteko ter vse skupaj znova povezati.

Lastne knjižnice

Včasih je smiselno narediti lastno knjižnico podprogramov. Naredimo jo iz dveh izvornih datotek; v eni so prototipi funkcij, v drugi pa funkcije same. Slednja mora tudi uvažati datoteko s prototipi.

Knjižnico naredimo takole:

```
bash> gcc -fPIC -Wall -ansi -pedantic -g -c libutil.c

bash> gcc -g -shared -o libutil.so.M.N libutil.o
```

Najprej iz izvorne datoteke naredimo predmetno. Stikalo -fPIC naredi kod, ki ga lahko sočasno uporablja več programov. Nato iz predmetne datoteke naredimo deljeno izvršno; to pove stikalo -shared. Ime se mora začeti z lib. Dobljeno knjižnico zapišemo v standardni imenik, recimo /usr/lib, ustvarimo standardni povezavi nanjo ter osvežimo seznam razpoložljivih knjižnic:

```
bash> ln -s libutil.so.M.N libutil.so.M
bash> ln -s libutil.so libutil.so.M
bash> ldconfig
```

Ko je knjižnica teko narejena in nameščena, jo uporabljamo kakar vsako drugo knjižnico.