Indhold

1 Indledning

Denne note viser kort og kontant, hvordan man udfra erfaring i TRINE-programmering med lethed kan skrive simple C-programmer.

Dette skal ses som et eksempel på, at TRINE-sproget omfatter generelle principper, som man vil kunne genkende i andre almindelige imperative programmeringssprog.

For de vigtigste mekanismer i TRINE vil der blive præsenteret en slags oversættelse til C-syntaks. Dette vil dog ikke kunne sættes sammen til en fuldstændig automatisk oversættelse fra TRINE til C; formålet er kun at tydeliggøre den begrebsmæssige sammenhæng. Man skal heller ikke opfatte denne note som en egentlig introduktion til C-programmering. De to programmeringssprog tjener forskellige formål, og specielt omfatter C mange faciliteter, der ikke nævnes her. Men forhåbentlig vil denne præsentation lette overgangen en smule. Den kanoniske C-bog er *The C Programming Language (Second Edition), Kernighan & Ritchie, Prentice Hall Software Series*.

For at øge læseligheden vil TRINE-programmer blive skrevet i den fra DAT1-noterne kendte stil, `hvorimod C-programmer skrives i denne font`.

2 Sætninger og udtryk

Handlingen i et C-program beskrives ligesom i TRINE ved hjælp af sætninger og udtryk. En sætning er dog ikke i sig selv et program. Den mindste programenhed, som vi vil betragte, er af formen:

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <math.h>
#define true 1;
#define false 0;
typedef int bool;
typedef FILE *file;
typedef char *string;

main(){ ... }
```

Syntaksen `#include` angiver, at vi benytter nogle standardbiblioteker, `#define` og `typedef` giver nogle nyttige synonymer, og `main(){...}` omgiver den sætning, der skal udføres.

En vigtig syntaktisk krølle er, at i C skal alle sætninger, *undtagen* indskudte sætninger, afsluttes med et semikolon.

2.1 Simple typer

I TRINE er de simple typer Int, Bool, Char og Real. De modsvares i C af typerne `long`, `int`, `char` og `double`. Bemærk, at Int svarer til lange heltal i C, at Bool blot er heltal, og at Real svarer til pseudoreelle tal med dobbelt præcision. C har desuden typerne `float` af pseudoreelle tal med enkelt præcision og `short` af korte heltal.

Syntaksen for konstanterne af disse typer er ligesom i TRINE. Der er forskellige svært gennemskuelige implicitte konverteringer mellem værdier af disse typer.

At sandhedsværdierne er heltal betyder, at `0` svarer til false og at alle andre heltal svarer til true. I vores standardprogram benyttes de to `#define`-erklæringer og `typedef`-erklæringen til at tillade syntaksen `bool`, `true` og `false` i programmer.

2.2 Tilstande og indskudte sætninger

Hovedsætningen består af nøgleordet `main()` efterfulgt af en indskudt sætning. En indskudt sætning skrives i krøllede parenteser. Som i TRINE indeholder den en lokal erklæring af variabler og typer efterfulgt af en sætning.

Erklæringer af variabler skrives som typens navn efterfulgt af en række variabelnavne afsluttet med et semikolon, som fx:

```
int i,j,size;
bool found;
```

Der findes ingen standardværdier i C; den initielle værdi af nyskabte variabler er udefineret. En tilordning i TRINE af formen `x:=u` ser her ud som:

```
x = u;
```

hvilket ikke skal forveksles med en sammenligning! Der er ikke multiple tilordninger eller ombytninger, men C har flere andre snedige mekanismer til at opdatere variabler.

2.3 Værdiudtryk

TRINE og C har, ligesom alle andre programmeringssprog, omtrent de samme operatorer på simple værdier. Man kan dog ikke være sikker på, at operatorerne har de samme prioriteter, så man bør sætte rigeligt med parenteser. Her er en lille tabel, der viser syntaksen for nogle kendte operatorer:

Trine	C
+	+
-	-
*	*
/	/
mod	%
%	/
$\vee$	||
$\wedge$	&&
$\neg$	!
$=$	==
$\neq$	!=
$<$	<
$>$	>
$\leq$	<=
$\geq$	>=

Visse af de indbyggede værdiprocedurer oversættes som følger:

Trine	C
abs	`abs`
ti	`atol`
floor	`floor`
ceil	`ceil`
sqrt	`sqrt`
sin	`sin`
cos	`cos`
arctan	`atan`
readchar	`getchar`

Procedurerne `ic` og `ci` indsættes implicit af C-oversætteren, hvor de er nødvendige. En speciel egenskab ved C-programmer er, at sætninger kan opfattes som værdiudtryk og vice versa. Når et værdiudtryk bruges som en sætning, smides den beregnede værdi væk; det er således kun eventuelle sideeffekter, der registreres. For hver slags sætning er der defineret en mere eller mindre oplagt værdi, som er dens resultat som værdiudtryk. Det mest nyttige tilfælde er, at den beregnede værdi af en tilordning er den tilordnede

værdi; det vil sige, at i sætningen:

```
x = y = z = 87
```

vil alle tre variabler blive tilordnet værdien 87.

2.4 Kontrolstrukturer

TRINE har to kontrolstrukturer for henholdsvis selektion og iteration. C har utallige variationer over disse temaer. Her vises blot nogle mulige oversættelser. En selektion af formen:

if $B_1 \rightarrow S_1$
& $B_2 \rightarrow S_2$
& ...
& $B_n \rightarrow S_n$
fi

modsvares af:

```
if (B1)
   {S1}
else if (B2)
   {S2}
...
else if (Bn)
   {Sn}
```

Det ofte forekommende specialtilfælde:

if $B \rightarrow S_1$
& true $\rightarrow S_2$
fi

kan skrives som:

```
if (B)
   {S1}
else
   {S2}
```

En iteration af formen:

do B $\rightarrow$ S **od**

modsvares af:

```
while (B) {S}
```

Det ofte forekommende specialtilfælde:

i:=0
do i<u $\rightarrow$
    S
    i:=i+1
od

kan skrives som:

```
for (i = 0; i < u; i++) {S}
```

En mere indviklet, generel iteration må stykkes sammen af flere kontrolstrukturer. Sætningen:

do $B_1 \rightarrow S_1$
 & $B_2 \rightarrow S_2$
 & ...
 & $B_n \rightarrow S_n$
od

kan efterlignes med:

```
while (1) {
  if (B1) {S1; continue;}
  if (B2) {S2; continue;}
  ...
  if (Bn) {Sn; continue;}
  break;
}
```

Der er ingen nondeterministiske valg i C.

2.5 Udskrivning

I TRINE har alle værdier kanoniske tekstformater, der kan udskrives med write-sætningen; i modsætning hertil kan kun tekster og simple værdier udskrives i C. Den generelle sætning er:

```
printf(format,U1,U2,...,Un)
```

Her er `Ui`'erne værdiudtryk og `format` er en tekstkonstant, der for hvert `Ui` angiver dets type efter følgende skema:

Type	Format
int	`%i`
char	`%c`
float	`%f`
string	`%s`

Da vi selv har fundet på typen `bool`, har den ikke noget format. Man kan også benytte format-teksten til at omgive de udskrevne værdier med noget tekst. Et eksempel er følgende:

```
printf("Kvadratroden af %i er %f\n",87,sqrt(87))
```

der udskriver teksten `Kvadratroden af 87 er 9.327379`. Notationen `"\n"` betyder linjeskift, svarende til TRINE's eol.

2.6 Indlæsning

TRINE's read-sætning modsvares af `printf`'s inverse:

```
scanf(format,X1,X2,...,Xn)
```

hvor `Xi`'erne nu skal være pointere (mere herom i afsnit 4). Hvis `x` er en `int` variabel, så vil sætningen:

```
scanf("%i",&x)
```

indlæse et heltal i `x`. Man kan også bruge `scanf` mere avanceret. Hvis `x` er en `int` variabel og `y` er en `float` variabel, så vil sætningen:

```
scanf("Kvadratroden af %i er %f\n",&x,&y)
```

indlæse teksten `"Kvadratroden af 87 er 9.327379"` og tilordne `87` og `9.327379` til henholdsvis `x` og `y`. Man kan dog ikke bruge `scanf` til indlæsning af tekster, da den standser ved det første blanke tegn og ikke ved linjeskift. Til dette formål skal man i stedet bruge sætningen `gets(&x)`.

2.7 Filer

C har ikke nogen analogi til TRINE's load eller dump. Til gengæld kan filer læses og skrives på samme måde som tastaturet og skærmen. En fil er i C en værdi af typen `file`. En fil i et omgivende UNIX-system skal *åbnes* inden den kan behandles. Hvis `f` er en variabel af type `file` og `n` er et filnavn, det vil sige en tekstkonstant i det sædvanlige format, så vil:

```
f = fopen(n,"r")
```

sætte `f` til at være filen `n` klargjort til læsning. Nu kan man læse sig igennem filen med sætninger som:

```
fscanf(f,format,X1,X2,...,Xn)
```

der er helt analoge til `scanf`. Hvis filen åbnes som:

```
f = fopen(n,"w")
```

kan man skrive på den med:

```
fprintf(f,format,U1,U2,...,Un)
```

Det tidligere indhold af filen forsvinder, når man skriver på denne måde. Hvis filen åbnes med:

```
f = fopen(n,"a")
```

vil man skrive i forlængelse af det eksisterende indhold. Når man er færdig med en fil, skal den lukkes igen med sætningen:

```
fclose(f)
```

Hvis en fil ikke findes, når den åbnes til skrivning, så vil den blive oprettet.

3 Regulære typer

De basale typekonstruktører i C svarer netop til de regulære typer, der kendes fra TRINE, dog uden så mange operatorer.

3.1 Navngivne typer

Som i TRINE kan man definere synonymer for eksisterende typer i begyndelsen af en indskudt sætning. Syntaksen:

Type N = T

modsvares i C af:

```
typedef T N;
```

Rekursive typer er dog ikke mulige.

3.2 Lister

Lister har i C en fast længde, der angives ved definitionen. TRINE-typen:

Type L = **List**(T)

kan fx skrives som:

```
typedef T L[100];
```

hvis listerne alle skal være af længde 100. Således kan C-syntaksen:

```
L x;
```

bedst sammenlignes med TRINE-konstruktionen:

Var x: L
x := **List**(?-T | 100)

bortset fra, at C-listens elementer starter med være udefinerede. Syntaksen `x.(i)` modsvares af:

```
x[i]
```

der angiver den `i`'te delvariabel af `x`. Der findes ikke andre operatorer på lister; specielt mangler tilordninger og sammenligninger af hele lister.

3.3 Tekster

En tekst er i begge sprog blot en liste af tegn, og har som sådan også en fast længde i C. Det er dog muligt at lave visse operatorer, der kendes fra generelle TRINE-lister. C tillader længderne af tekster at variere op til den erklærede øvre grænse, idet tegnet med ASCII-værdi 0 bruges til at markere afslutningen. Lad `x`, `y` og `z` være variabler af forskellige slags tekster, og lad `x` være lang nok. Så er følgende sætninger analoge:

TRINE	C
|x|	`strlen(x)`
x=y	`strcmp(x,y)==0`
x:=y	`x=strdup(y)`
x:=y++z	`strcpy(x,y);` `strcat(x,z)`
x:=y(i..j)	`strncpy(x,y+i,j-i);` `x[j-1]='\0'`

Disse operationer tager samme tid som i TRINE bortset fra, at `strlen` er lineær i tekstens længde. Et nyttigt specialtilfælde er, at teksten y(i..|y|) kan udskrives som som det bizarre udtryk `y+i`. Typen `string` kan indeholde tekster af forskellige, men individuelt faste længder.

3.4 Produkter

Produkttypen:

Type P = **Prod**(x_1:T_1, ...,x_n: T_n)

skrives i C som:

```
typedef struct P {
   T1 x1;
   ...
   Tn xn;
} P;
```

Her kan man tilordne men ikke sammenligne hele værdier. Man kan heller ikke skrive konstante udtryk, men det er muligt at give nyskabte variabler en startværdi, idet:

```
P p = { U1, U2, ..., Un };
```

i TRINE svarer til:

Var p: P
p:=P(U_1, U_2, ...,U_n)

Hvis der ingen startværdier gives, så har hver delvariabel en udefineret værdi. Som i TRINE udvælges en delvariabel med syntaksen `p.xi`.

3.5 Summer

Sumtypen:

Type S = **Sum**(x_1: T_1, ...,x_n: T_n)

kan i første omgang tilnærmes med C-typen:

```
typedef union S {
   T1 x1;
   ...
   Tn xn;
} S;
```

Forskellen er, at en værdi af en `union`-type ikke har nogen variant-komponent. Det vil sige, at man ikke kan skelne mellem de forskellige varianter. Således vil TRINE-sætningen:

s:=S(x_i: U_i)

simpelthen blive skrevet som:

```
s.xi=Ui
```

og **is**(s,x_i) kan ikke besvares. Man benytter ofte en mere hensigtsmæssig indkodning, hvor variant-komponenten eksplicit repræsenteres. Således skrives typen som:

```
typedef struct S {
  enum{x1_,...,xn_} tag;
  union {
     T1 x1;
     ...
     Tn xn;
  } val;
} S;
```

Dette er et produkt med to komponenter `tag` og `val`. Typen af `tag` er mængden af de navne (forsynet med `"_"`), der angiver de forskellige varianter. Typen af `val` er den tidligere `union`. Nu bliver:

$$s := S(x_i{:}U_i)$$

skrevet som:

```
s.val.xi=Ui;
s.tag=xi_
```

og **is**(s,x_i) kan besvares af:

```
s.tag==xi_
```

Delvariablen for den `i`'te variant udvælges med:

```
s.val.xi
```

En komplikation ved denne løsning er, at der ikke må findes to forskellige summer, der har et fælles variantnavn.

Man kan programmere sum-skabelonen med en specielt effektiv kontrolstruktur, idet:

if **is**$(s,\ x_1) \rightarrow S_1$
& **is**$(s,\ x_2) \rightarrow S_2$
...
& **is**$(s,\ x_n) \rightarrow S_n$
fi

kan skrives som:

```
switch (s.tag) {
  case x1_: S1; break;
  case x2_: S2; break;
  ...
  case xn_: Sn; break;
}
```

Her foretages valget i konstant tid ved hjælp af tabelopslag.

3.6 Typeækvivalens

Typeækvivalens i C minder om definitionen i TRINE, i den forstand at synonymer defineret med `typedef` ikke er signifikante. Der er dog ikke den samme strukturelle ækvivalens. To identiske anvendelser af en typekonstruktør, der står to forskellige steder i programmet, er ikke ækvivalente. Undtaget fra dette er dog pointertyper.

4 Pointere

Pointere er et så simpelt begreb, at det er praktisk taget identisk i alle sprog. C har dog et mere intents forhold til pointere end de fleste andre sprog, så de dukker op oftere, end man skulle forvente.

Definitionen:

Type P = **Pointer**(T)

modsvares af:

```
typedef T *P;
```

Nedenstående tabel giver en oversættelse af de almindelige operationer:

TRINE	C
nil	`NULL`
P(?-T)	`(P)malloc(sizeof(T))`
ref(p)	`*p`
dispose[p]	`free(p); p=NULL`
pointto[x]	`&x`

Som en ofte brugt forkortelse kan man skrive `p->x` i stedet for `(*p).x`.

5 Procedurer

C har omtrent det samme procedurebegreb som TRINE. Der findes dog hverken variabelparametre eller variabelprocedurer; man bruger i stedet pointere. TRINE proceduren:

Proc P(x_1: T_1,... x_n: T_n)
  S
end P

modsvares af:

```
void P(T1 x1, ..., Tn xn)
{ S }
```

og værdiproceduren:

Proc P(x_1: T_1,... x_n: T_n) → (T)
  S
end P

tilsvarende af:

```
T P(T1 x1, ..., Tn xn)
{ S }
```

C har ligesom TRINE en `return`-sætning, der angiver den resulterende værdi.

Procedurer kan i C ikke defineres i indskudte sætninger, men kun på yderste niveau af programmet, uden for `main()`. Det betyder specielt, at man ikke kan have lokale procedurer.

Generelle rekursive procedurer er mulige, men kun hvis man prædefinerer procedurerne. Dette sker ved skrive deres første linje en ekstra gang øverst i programmet. Den præcise struktur ses i eksemplet i afsnit 7.

Man kan efterligne variabelparametre ved hjælp af pointere. Det sker efter følgende skabelon. TRINE-syntaksen:

```
Proc A[x: Int](y: Int)
    x:=x+y
end A

A[z](87)
```

modsvares af:

```
void A(int *x, int y)
{ (*x)=(*x)+y; }

A(&z,87)
```

Variabelprocedurer kan efterlignes tilsvarende. Man kan observere, at `&(*x)` er det samme som `x`, hvilket er nyttigt ved variabelparametre i rekursive kald.

6 Rekursive typer

C har ikke rekursive typer, men de kan efterlignes med pointere som beskrevet i TRINE-notens afsnit 15.3. Det betyder også, at man må leve med at skrive procedurer til sammenligning, kopiering, udskrift og så videre. Da vi allerede har set hvordan summer og produkter efterlignes, er fremgangsmåden ligetil. Den rekursive TRINE-type:

Type Expr = **Sum**(num: Int, plus, minus: Arg)
Type Arg = **Prod**(left, right: Expr)

oversættes således til:

```
typedef struct Expr_ *Expr;
typedef struct Arg_ *Arg;
typedef struct Expr_ {
          enum{num_,plus_,minus_} tag;
          union {
            int num;
            Arg plus,minus;
          } val;
        } Expr_;
typedef struct Arg_ {
          Expr left,right;
        } Arg_;
```

Denne type bliver anvendt i afsnit 7, hvor man kan se eksempler på opbygning og gennemløb af rekursive træer.

7 Eksempel: regneudtryk

I dette afsnit viser vi et lille TRINE-program og en tilsvarende version i C. Programmet indlæser et simpelt regneudtryk, opbygger et rekursivt udtrykstræ og beregner dets værdi. Programmet er en version af eksemplet med venstreassociative regneudtryk fra TRINE-notens afsnit 13.2, hvor vi dog har fjernet brugen af de polymorfe sekvenser. Oversættelsen til C er lavet systematisk udfra de tidligere afsnit. Det derved opnåede program er ikke væsentligt forskelligt fra, hvad man ville opnå ved at skrive direkte i C, selv om det selvfølgelig til en vis grad afspejler en særlig "TRINE-stil".

7.1 Programmet i TRINE

(+ **Type** Expr = **Sum**(num: Int, plus, minus: Arg)

```
Type Arg = Prod(left, right: Expr)
Type Response = Sum(ok: Expr, error: Text)

Proc BuildExpr[s: Text, i: Int, r: Response]
    BuildTerm[s, i, r]
    if is(r, ok) ∧ (i< | s |) →
        if s.(i) = '+' →
            i := i+1
            (+ Var q: Response
               BuildExpr[s, i, q]
               if is(q, ok) → r.ok := Expr(plus: Arg(r.ok, q.ok))
               & true → r := q
               fi
            +)
        & s.(i) = '-' →
            i := i+1
            (+ Var q: Response
               BuildExpr[s, i, q]
               if is(q, ok) → r.ok := Expr(minus: Arg(r.ok, q.ok))
               & true → r := q
               fi
            +)
        fi
    fi
end BuildExpr

Proc BuildTerm[s: Text, i: Int, r: Response]
    if i< | s | →
        if '0' ≤ s.(i) ≤ '9' →
            r := Response(ok: Expr(num: ci(s.(i))-ci('0')))
```

```
            i:=i+1
        & s.(i) = '(' →
            i:=i+1
            (+ Var q: Response
               BuildExpr[s, i, q]
               if is(q, ok) →
                    if (i< | s |) ∧ (s.(i) = ')') →
                         i:=i+1
                         r:=q
                    & true → r:=Response(error: ") expected")
                    fi
               & true → r:=q
               fi
            +)
        & true → r:=Response(error: "Digit or ( expected")
        fi
    & true → r:=Response(error: "Incomplete expression")
    fi
end BuildTerm

Proc Eval[e: Expr] → (Int)
    if is(e, num) → return e.num
    & is(e, plus) → return Eval[e.plus.left]+Eval[e.plus.right]
    & is(e, minus) → return Eval[e.minus.left]-Eval[e.minus.right]
    fi
end Eval

Var s: Text

write("Write an expression: ")
read[s]
```

```
    (+ Var r: Response
       Var i: Int
       i := 0
       BuildExpr[s, i, r]
       if is(r, error) → write(r.error)
       & i< | s | →
            write("Unexpected text: ")
            write(s(i .. | s |))
       & true → write("The value is: ", Eval[r.ok])
       fi
    +)
    write(eol)
+)
```

7.2 Programmet i C

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <math.h>
#define true 1
#define false 0
typedef int bool;
typedef FILE *file;
typedef char *string;

typedef struct Expr_ *Expr;
typedef struct Arg_ *Arg;
typedef struct Expr_ {
          enum{num_,plus_,minus_} tag;
          union {
            int num;
            Arg plus,minus;
          } val;
        } Expr_;
typedef struct Arg_ {
          Expr left,right;
        } Arg_;

typedef struct Response {
          enum{ok_,error_} tag;
          union {
            Expr ok;
            string error;
          } val;
        } Response;

void BuildExpr(string s, int *i, Response *r);
void BuildTerm(string s, int *i, Response *r);
int Eval(Expr e);
```

```
void BuildExpr(string s, int *i, Response *r)
{ BuildTerm(s,i,r);
  if ((r->tag==ok_) && (*i < strlen(s)))
     { if (s[*i]=='+')
          { *i=*i+1;
            { Response q;
              BuildExpr(s,i,&q);
              if (q.tag==ok_)
                 { Arg a;
                   a=(Arg)malloc(sizeof(Arg_));
                   a->left=r->val.ok;
                   a->right=q.val.ok;
                   r->tag=ok_;
                   r->val.ok=(Expr)malloc(sizeof(Expr_));
                   r->val.ok->tag=plus_;
                   r->val.ok->val.plus=a;
                 }
              else *r=q;
            }
          }
       else if (s[*i]=='-')
          { *i=*i+1;
            { Response q;
              BuildExpr(s,i,&q);
              if (q.tag==ok_)
                 { Arg a;
                   a=(Arg)malloc(sizeof(Arg_));
                   a->left=r->val.ok;
                   a->right=q.val.ok;
                   r->tag=ok_;
                   r->val.ok=(Expr)malloc(sizeof(Expr_));
                   r->val.ok->tag=minus_;
                   r->val.ok->val.minus=a;
                 }
              else *r=q;
            }
          }
}    }
```

```
void BuildTerm(string s, int *i, Response *r)
{ if (*i < strlen(s))
     { if (('0' <= s[*i]) && (s[*i] <= '9'))
          { r->val.ok=(Expr)malloc(sizeof(Expr_));
            r->tag=ok_;
            r->val.ok->tag=num_;
            r->val.ok->val.num=(s[*i]-'0');
            *i=*i+1;
          }
       else if (s[*i] == '(')
          { *i=*i+1;
            { Response q;
              BuildExpr(s,i,&q);
              if (q.tag==ok_)
                 { if ((*i < strlen(s)) && (s[*i] == ')'))
                      { *i=*i+1;
                        *r=q;
                      }
                   else
                      { r->tag=error_;
                        r->val.error=") expexted";
                      }
                 }
              else *r=q;
            }
          }
       else
          { r->tag=error_;
            r->val.error="Digit or ( expected";
          }
     }
  else
     { r->tag=error_;
       r->val.error="Incomplete expression";
     }
}
```

```
int Eval(Expr e)
{ switch (e->tag) {
    case num_:   return e->val.num;
                 break;
    case plus_:  return Eval(e->val.plus->left)+
                        Eval(e->val.plus->right);
                 break;
    case minus_: return Eval(e->val.minus->left)-
                        Eval(e->val.minus->right);
                 break;
  }
}

char s[100];

main() {
  printf("Write an expression: ");
  scanf("%s",s);
  { int i=0;
    Response r;
    BuildExpr(s,&i,&r);
    if (r.tag==error_)
       { printf("%s",r.val.error); }
    else if (i < strlen(s))
       { printf("Unexpected text: ");
         printf("%s",s+i);
       }
    else
       { printf("The value is: %i",Eval(r.val.ok)); }
  }
  printf("\n");
}
```

8 Forskelle

Gennemgangen indtil nu har fokuseret på lighederne mellem Trine og C. Der er dog også mange betydelige forskelle. Trine understøtter visse begreber, der ikke har oplagte analogier i C. Det drejer sig om:

- Rekursive typer; vi kender dog den generelle teknik til at efterligne disse ved hjælp af pointere.
- Beskyttelse, datatyper og polymorfi; her er der forskellige svar. Beskyttelse og datatyper kan erstattes med selvdisciplin. Polymorfi kan ikke efterlignes, undtagen med copy-and-paste af koden. Klassebegrebet i C++ råder dog netop bod på disse mangler.
- Trinvis forfinelse; Trine's konkrete udgave af denne teknik er nok et særsyn, men man kan jo fint gøre brug af tankegangen alligevel.
- Processer og kanaler; her er der ikke noget oplagt svar. Man er nødt til at benytte sig direkte af indbyggede UNIX-primitiver, men det er jo også netop sådan Trine er implementeret.

Omvendt har C blandt andre følgende faciliteter, der ikke understøttes i Trine:

- Direkte adgang til maskinens lager; man kan arbejde på lagerceller og bitmønstre uden at gå gennem typesystemet.
- Syntaks for mange specialtilfælde; der er særligt effektive variationer over de basale sætninger og udtryk.
- Fuld adgang til UNIX-systemet; C og UNIX er udviklet sammen, og skillefladen mellem dem er nogle gange lidt utydelig.
- Separat oversættelse af biblioteker; dette er en nødvendighed ved udvikling af store systemer.
- Mange standardbiblioteker og avancerede udviklingsværktøjer.

9 Oversættelse

Ligesom for TRINE-programmer gælder det, at C-programmer skal oversættes inden de kan udføres. Hvis programmet er på filen `program.c`, så oversættes det med UNIX-kommandoen:

```
gcc -o program program.c -lm
```

Dette skaber en fil `program`, der kan udføres som en UNIX-kommando.

Hvis programmet indeholder syntaksfejl, så skrives fejlmeddelelser ud på skærmen. Da C har en temmelig kryptisk syntaks, kan man regne med, at fejlmeddelelserne er endnu dårligere end dem, man får fra TRINE. Hvis man i eksemplet fra afsnit 7 erstatter `typedef char *string` med trykfejlen `typedaf char *string`, så får man 28 fejlmeddelelser der involverer 18 forskellige programlinjer; kun den første af disse har nogen relevans til fejlen. UNIX-kommandoen:

```
lint program.c
```

giver en velment kritik af den anvendte programmeringsstil. Med værktøjet `gdb` kan man undersøge tilstanden af en kørende C-program, hvilket er meget nyttigt ved fejlfinding.

10 Simple Biblioteker

C-programmer kan laves til biblioteker i stil med TRINE's `@`-filer. Der er dog her den store fordel, at biblioteket kan oversættes separat en gang for alle. Dette betyder for større anvendelser en meget hurtigere oversættelse. Dette afsnit giver et eksempel på udviklingen af et simpelt bibliotek: cykliske stakke fra notens afsnit 15.5. TRINE-versionen består af følgende typer og procedurer på filen `cyclic.tri`.

```
Type Stack = Pointer(Node)
Type Node = Prod(val: Int, last, next: Stack)

Proc Init[S: Stack]
    S := nil
end Init

Proc Empty[S: Stack] → (Bool)
    return S = nil
end Empty

Proc Push[S: Stack](i: Int)
    if S = nil →
        S := Stack(Node(i, nil, nil))
        ref(S).last := S
        ref(S).next := S
    & true →
        (+ Var T: Stack
           T := Stack(Node(i, ref(S).last, S))
           ref(ref(S).last).next := T
           ref(S).last := T
           S := T
        +)
    fi
end Push

Proc Top[S: Stack, i: Int]
    i := ref(S).val
end Top

Proc Rotate[S: Stack]
    if S ≠ nil → S := ref(S).next fi
end Rotate
```

Det tilsvarende C-bibliotek består af to filer: `cyclic.h` og `cyclic.c`. Filen `cyclic.c` indeholder den direkte oversættelse af TRINE-programmet.

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <math.h>
#define true 1;
#define false 0;
typedef int bool;
typedef FILE *file;
typedef char *string;

typedef struct Node *Stack;
typedef struct Node {
  int val;
  Stack last,next;
} Node;

void Init(Stack *S)
{ (*S)=NULL; }

bool Empty(Stack *S)
{ return (*S)==NULL; }

void Push(Stack *S, int i)
{ if ((*S)==NULL) {
    (*S)=(Stack)malloc(sizeof(Node));
    (*S)->val=i;
    (*S)->last=(*S);
    (*S)->next=(*S);
  } else {
    Stack T;
    T=(Stack)malloc(sizeof(Node));
    T->val=i;
    T->last=(*S)->last;
    T->next=(*S);
    (*S)->last->next=T;
    (*S)->last=T;
    (*S)=T;
}}
```

```
void Top(Stack *S,int *i)
{ (*i)=(*S)->val; }

void Rotate(Stack *S)
{ if ((*S)!=NULL) {
    (*S)=(*S)->next;
  }
}
```

Dette program oversættes med UNIX-kommandoen:

```
gcc -c cyclic.c
```

Dette skaber en oversat fil med navn `cyclic.o`. Filen `cyclic.h` skal kun indeholde typedefinitionerne og prædefinitioner af procedurerne.

```
typedef struct Node *Stack;
typedef struct Node {
  int val;
  Stack last,next;
} Node;

void Init(Stack *S);

bool Empty(Stack *S);

void Push(Stack *S, int i);

void Top(Stack *S,int *i);

void Rotate(Stack *S);
```

En brug af biblioteket ser i TRINE ud som følger.

```
(+ @"cyclic.tri"

   Var C: Stack
   Var n, i: Int
   Init[C]
   n:=0
   read[i]
   do i≠0 →
        Push[C](i)
        n:=n+1
        read[i]
   od
   do n>0 →
        Top[C, i]
        write(i, eol)
        Rotate[C]
        n:=n-1
   od
+)
```

Det tilsvarende C-program ser således ud.

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <math.h>
#define true 1;
#define false 0;
typedef int bool;
typedef FILE *file;
typedef char *string;

#include "cyclic.h"
```

```
main(){
  Stack C;
  int n,i;

  Init(&C);
  n=0;
  scanf("%i",&i);
  while (i!=0) {
    Push(&C,i);
    n=n+1;
    scanf("%i",&i);
  }
  while (n>0) {
    Top(&C,&i);
    printf("%i\n",i);
    Rotate(&C);
    n=n-1;
  }
}
```

Hvis ovenstående program findes på filen `brug.c` skal det oversættes med UNIX-kommandoen:

```
gcc -o brug brug.c cyclic.o -lm
```

Dette giver så slutteligt en udførbar fil `brug`. Bemærk, at vi her kun bruger den oversatte fil `cyclic.o`.