Robotiehitamise juhend

Mikrokontrolleri programmeerimine C keeles Põhiline C keelest

Autorid: Alar Ainla
Alvo Aabloo

Juhendi koostamist on toetanud EITSA

SISUKORD

Sissejuhatus

C keele lõi **Dennis Ritchie** *(Bell Laboratories)* 1970 ndate aastate alguses, kes arendas seda algselt PDP-11 riistvaral ja UNIX operatsioonisüsteemil. Kuigi see oli algselt loodud UNIXil kasvas õigepea ka teiste operatsioonisüsteemide kasutajate huvi C keele vastu. C keel on suurepärane keskkond, koodi lihtsuse, kompaktsuse ja laia võimaluste hulga tõttu. Selle kõige tõttu on ta enamasti esimene kõrgtaseme keel saadaval kõikidele uutele arvutitele, mikrokontrolleritest, minikompuutritest kuni suurarvutiteni. C headus seisneb selles, et ta lubab programmeerijal kasutada väga suurt hulka vahendeid alates kõrgtasemest *(high level)* kuni väga madala tasemeni *(low level)*, mis läheneb juba assembler keelele. C annab programmeerijale suhteliselt vabad käed, nõudes sellega ka muidugi vastutust (Pascal kutsub välja fatal error'i, Cga oled sa enamustes neis küsimustes oma pead, eriti kui tegu on mikrokontrolleritega)

Selles C keeld tutvustavas juhendis on põhi rõhk, nagu nimigi ütleb suunatud mikrokontrolleri programmeerimisele (st. arvuti millel puuduvad ketta seadmed, faili süsteem, klassikalised sisend-väljund (IO) seadmed, nagu klaver, printer, hiir, monitor ning mille mälu mahud ja kiirused on nende arvutitega, mis meil laua peal seisvad, võrreldes üliväikesed). Konkreetselt vaatleme väikese ja odava *(Embedded)* roboti ehitamiseks sobivat Texas Instruments® (TI) *Mixed Signal Processor* seeria **MSP430F** jaoks mõeldud C kompilaatorit ja *library*'t (IAR Embedded Workbench, mille piiratud koodi suurusega varjant on vabalt saadav TI kodulehelt www.ti.com . On olemas ka ilma piiranguteta vaba **gcc** kompilaator MSP protsessoritele http://mspgcc.sourceforge.net (nii Linuxi, kui Windowsi variandid, mõlemad GNU vabavara)

Esimene proov

Kui laua arvuti puhul tehtaks esimeseks programmiks HelloWorld, siis mikrokontrolleri puhul oleks väga sobilik teha mingisugune "tulukese" – valgusdioodi (LED) vilgutaja, see annab juba informatsiooni selle kohta, kas kontroller korralikult käima hakkas (toide, ostsillator jms.) ning kas suutsite talle programmi sisse laadida. Järgneva näite puhul tuleks panna valgusdiood jala P1.0 külge, kuidas seda täpselt teha loe juhendist (Arendus keskkondade installeerimine)

Programm

```
#include  <msp430x14x.h>
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;           // Peata watchdog timer
  P1DIR |= 0x01;                      // Sea P1.0 väljundiks
  for (;;)
  {
    unsigned int i;
    P1OUT ^= 0x01;                    // Inverteeri P1.0
```

```
    i = 50000;                          // Viide
    do (i--);
    while (i != 0);
  }
}
```

Selgitus

`#include  <msp430x14x.h>`

See on ***h****eader* (päis) faili lugemine. *Header* failist saadakse funktsioonide prototüübid, makrod, muutujad, andme struktuurid, mida saab pärast kasutada (Nagu WDTCTL jne.)

`void main(void)`

See on main (ehk peamine) funktsioon, mis käivitatakse siis kui on kontroller käivitub või pärast reset'it.

`WDTCTL = WDTPW + WDTHOLD;`

Selle WDTCTL registri muutmisega peatatakse *watchdog timer.* Kui seda ei peatataks, siis genereeriks ta perioodiliselt reset'eid.

`P1DIR |= 0x01;`

Sea pordis P1 esimene jalg (0) väljundiks. Selleks seatakse P1DIR registris vastav bitt 1 ks. Biti seadmist 1 ks tehakse enamasti "või" tehtega.

`for (;;)`

Tingimusteta for-tsükkel on lõbmatu tsükkel. See tähendab, et tema järel loogeliste sulgude vahel oleva osa täitmist korratakse lõpmatult.

`unsigned int i;`

Muutuja i deklareerimine, muutuja tüübiks on märgita täisarv.

`P1OUT ^= 0x01;`

Exclusive-Ori (XOR) kasutamine, et muuta P1OUT registris 0is bit vastupidiseks.

`i = 50000;`

Muutujale i omistatakse kümnend arv 50000

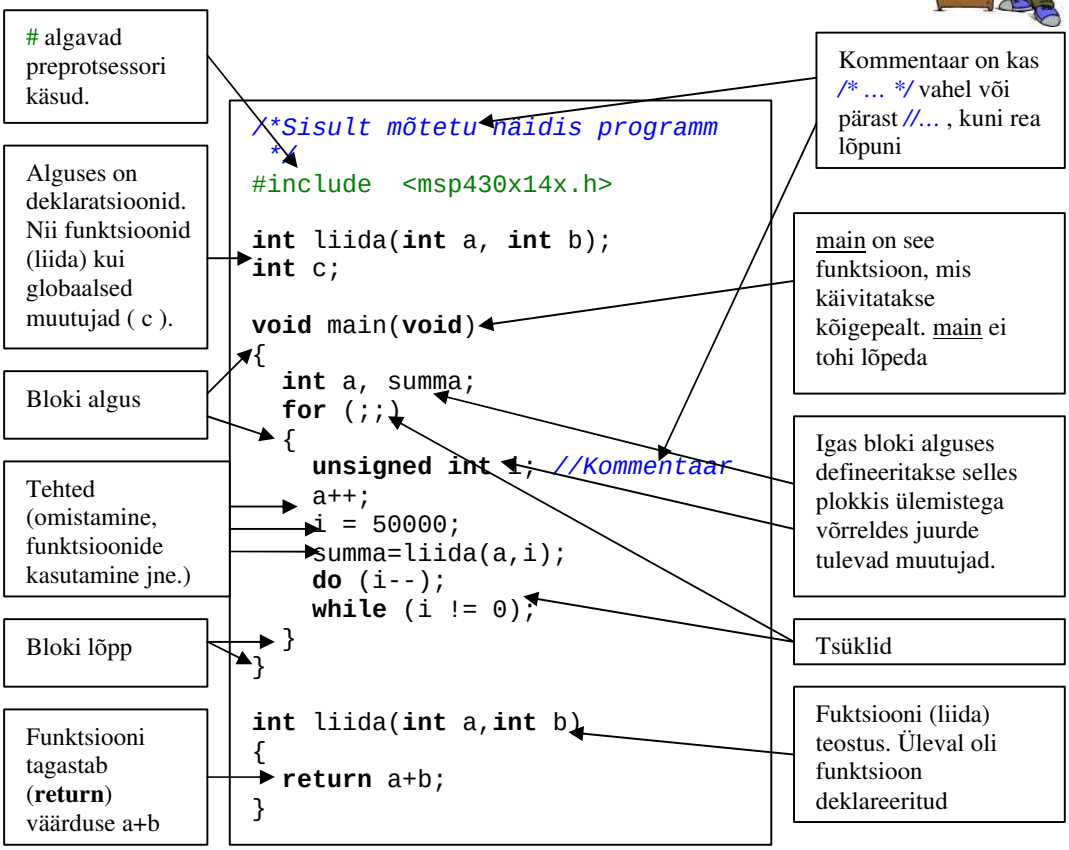
```
do (i--);
while (i != 0);
```

Muutuja i vähendamine nii kaua, kuni i on "0" (null), kui muutuja ei ole 0 siis lahuta temast "1". Sellise tsükli täitmine võtab aega, järelikult toimib see viitena.

C programmi põhi struktuur

IAR C on üldjoondes vastav ANSI C standartile.

Siin joonisel on toodud üks näidis C programmist, mis peaks andma aimu C programmi struktuurist, kus ja kuidas paiknevad funktsiooni ja muutuja deklaratsioonid, kommentaarid jne.



C keele reserveeritud sõnad

Reserveeritud sõnad on keele põhi elemendid, reserveeritud sõnu ei saa kasutada on muutujate, andmetüüpide ja funktsioonide nimedena.
Järgnev on ANSI C reserveeritud sõnade nimekiri:

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

Põhi andmetüübid

Arvuti hoiab andmeid mälu baitides. Igas baidis on 8 biti, mis võivad omada väärtusi 0 või 1. Ehk kõik andmed on kuidagiviisi kodeeritud mingisugusesse hulka 0 de ja 1 de kombinatsiooni. Seda kuidas mingeid andmeid esitakse ja tõlgendatakse ongi määratud tegelikult andmetüüpidega.
Näide:
Bitid: 11101110011010110010100000000000 võivad omada tähendust
-685835856737519998756450625 7.975 kui andmetüübiks on **float** (ujukoma arv) või

4000000000 kui tüübiks on **long** (täisarv). Ehk täiesti erinev !!!

Põhi andmetüübid jagunevad:

- Täisarvu tüübid - **integer**
- Ujukoma (reaalarvu) tüübid - **float**
- Viidatüüp - **pointer**
- **enum** tüüp

Täisarvu tüübid

Täisarvu tüübid on põhilised andmetüübid üldse. Täisarvu tüübid omakorda jagunevad, kas märgiga (**signed**) või märgita (**unsigned**) tüüpideks.
Näide:
Märgita char tüüpi bait 10011001 (kuueteiskümnend süsteemis 0x99) teisendatuna kümnend arvuks on 153. On näha, et selliselt saab esitada (1 baidi abil) täisarvud 0..255
Sama märgiga char tüübina oleks kümnend arvuna -103. On näha et selliselt saab esitada (1 baidi abil) täisarvud -128…127

Kuidas teisendada märgita arvu?

Märgita arvu teisendatakse nagu tavalist kahend arvu
Kui arv esitub kahendarvuna biti jadana $b_n...b_2b_1b_0$ siis kümnendarvu saab arvutada

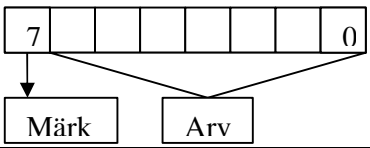
$$DEC = \sum_{i=0}^{n} b_i \cdot 2^i$$

Näide:
10100011 on

$$1\cdot 2^7 + 0\cdot 2^6 + 1\cdot 2^5 + 0\cdot 2^4 + 0\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 128 + 32 + 2 + 1 = 163$$

Kuidas teisendada märgiga arvu?

Märgiga arvu puhul näitab kõige kõrgem bit märki, ülejäänud bitid teisenduksid sama reegli järgi, mis toodud eelmises punktis. Kui kõige kõrgem bit on '1' siis on tegu nullist väiksema arvuga.



Kui signed char tüüpi arvu kõrgeim bit on 1 (märk) siis kogu arv arvutatakse **Arv-0x7F** (0x7F=128) (vt. joonist)
Näited:

HEX(16)	BIN(2)	Kümnend (signed)	**Kümnend (unsigned)**
0x00	00000000	0	0
0x34	00110100	52	52
0x7F	01111111	127	127
0x80	10000000	0-128=-128	128
0xA3	10100011	35-128=-93	163
0xFF	11111111	127-128=-1	255

Analoogiliselt on tehtud ka **int** ja **long** tüüpidega

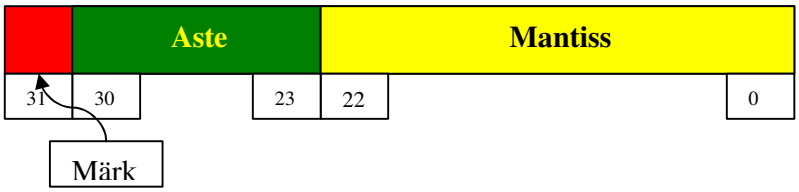
Täisarvu tüüpide nimed ja suurused

Tüübi nimi	Suurus (baiti)	Vahemik (astemetega)	Vahemik	Märkus
unsigned char	1		0..255	**char** loetakse vaikimisi selleks
signed char	1		-128…127	
char				Sama mis **unsigned char**, kui kasutada –c võtit siis **signed char**
short	2	$-2^{15}...2^{15}-1$	-32768…32767	
int				Sama mis **short**
unsigned short	2	$0...2^{16}-1$	0…65535	
unsigned int				Sama mis **unsigned short**
long	4	$-2^{31}...2^{31}-1$	-2147483648 … 2147483647	
unsigned long	4	$0...2^{32}-1$	0…4294967295	

Ujukoma tüübid

Ujukoma tüübitega esitatakse reaalarve, nende muutumis vahemik on väga suur, kuid arvud on ligikaudsed. Arvutuste käigus tehakse ümardusi jne. Antud kompilaatoris on ujukoma tüüpide nimed **float, double, long double**, millele sisuliselt vastab üks ja sama tüüp. Ujukoma tüübi pikkus on **4 baiti** ja vahemik $\pm 1.18 \cdot 10^{-38}...\pm 3.39 \cdot 10^{38}$. Täpsus on ligikaudu **7 kümnend kohta**. Ujukoma arvudega saab teha + , - , / , * tehteid.

Kuidas ujukoma arvud paiknevad mälus?



Ujukoma arv on 4 baiti pikk.

Sellise arvu väärtus avaldub:

$$(-1)^{Märk} \cdot 2^{Aste} \cdot 1.Mantiss$$

Viidatüüp

Viit on andmetüüp, mis sisaldab mälu adressi. Ta viitab mingitele andmetele. Viidatüübi suurus on **2 baiti**. Sellise viidaga saab viidata adressidele vahemikus (0x0000…0xFFFF) ehk 64KB.

enum tüüp

enum teeb iga objekti lühimasse täisarvu tüüpi (char, short, int, long) mis on vajalik antud väärdse talletamiseks. enum tüübi pikkus on **1…4 baiti**

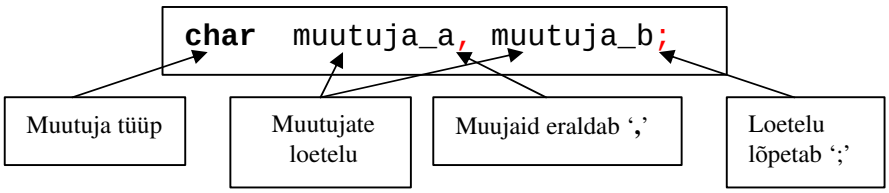
Biti väljad (bit fields)

On tegelikult täisarvu tüüpide eri juht, mis võimaldab teha vahepealsete pikustega täisarve. Selle mõte on enamasti kokku hoida mälu või kasutada teda koos eri

funktsiooni registritega (SFR). Kui võimalik, siis oleks kasulik biti väljadest hoiduda, kuna nad on MSP protsessoril aeglased.

Muutuja deklaratsioon

Kuidas deklareeritakse muutujat?



Muutuja deklaratsioonis on esimesel kohal tüüp, mis tüüpi muutujat soovitakse deklareerida. Sellele järgneb komadega eraldatud muutujate loetelu, mida deklareeritakse, deklaratsiooni lause, nagu enamuse C, lõpetab ‘;’ (semikoolon)

Kus deklareeritakse muutuja?

Globaalsed muutujad deklareeritakse väljaspool kõiki funktsioone – alguses. Globaalsed muutujad on kättesaadavad kõigis funktsioonides.
Näide:

```
//Faili algus
int globaalne_muutuja;  //Globaalne muutuja, mis on sama nii
                        //ka funktsioon_B s funktsioon_A s kui
void funktsioon_A()
{     …     }
int funktsioon_B(int a, int b)
{     …     }
```

Lokaalsed muutujad deklareeritakse alati bloki alguses enne kõiki käsu lauseid, mis selles blokis on.
Näide:

Õige näide

```
void main(void)
{       //Bloki algus
        int a, b; //Muutuja deklaratsioon
        //Käsu laused
         while(a!=0)
        { //Bloki algus
                char c; //Muutuja
                        //deklaratsioon

                  //Käsu laused
                a=a-2;
        }
}
```

Vigane näide

```
void main(void)
{       //Bloki algus
        int a, b; //Muutuja deklaratsioon
        a=a+b; //Käsu lause
        int c; /*Vigane muutuja
        deklaratsioon, kuna bloki alguse
ja muutuja deklaratsiooni vahele ei tohi
jääda käsu lauseid*/
         while(a!=0)
        {
        }

}
```

Muutuja deklaratsioonide ulatus – scope - "nähtavus"

Ehk kuhu deklareeritud muutuja paistab.
Muutuja paistab kogu bloki piires, milles ta deklareeritud on ja kõigis selle ploki alamplokides.
Näide: vaatame, kuhu paistavad muutujad **a** ja **b**

```
void funktsioon(void)
{
        int a; //Deklareerime muutuja a
        //Siin paistab a
        if( … )
        {       int b; //Deklareerime muutuja b
                //Siin paistab a ja b
                for(…)
                {
                        //Siin paistab a ja b
                }
        }
        while()
        {       //Siin paistab a
        }
}
void funktsioon2(void)
{
        //Ei paista ei a, ega b
}
```

Muutujate algväärtustamine deklareerimisel

Muutujaid võib ka deklareerimisel algväärtustada, kuid ainult konstandiga.
Algväärtustamine ei tohi sisaldada teisi muutujaid, funktsioone jne.
Näide:

```
char a=0, b=2; //Algväärtustamine konstandiga
```

Staatilised muutujad

Staatilised muutujad on üks liik lokaalseid muutujaid, mis on kättesaadavad blokkis, milles nad on defineeritud ja selle bloki alamblokides. Erinevus tavalisest muutujast on see et blokist väljumisel (näiteks funktsiooni töö lõppedes) selle muutuja väärtus ei lähe kaduma, vaid säilub. Kui programm satub töö käigus jälle selle muutuja nähtavus ulatusse siis on sama väärtus uuesti kätte saadav.

Näide: deklaratsioonist

```
static char muutuja;
```

Staatilisi muutujaid saab ka algväärtustada, kuid seda tehakse vaid üks kord.
Näide: staatilise muutuja kasutamisest.

```
void main(void)
{
        int a;
        a=tagasta();    //a saab väärtuseks 19
        a=tagasta();    //a saab väärtuseks 18
        a=tagasta();    //a saab väärtuseks 17
        //..jne
}

int tagasta(void)
{
        static char i=20; /*Esimesel korral algväärtustatakse muutuja, edaspidi
funktsiooni poole pöördudes enam mitte. Väärtus säilub ka pärast funktsiooni lõppu */
        i=i-1;
        return i;
}
```

Konstandid

Konstandid on muutujad, millel on fikseeritud väärtus, mida ei saa muuta. Konstante on mõtekas kasutada, kuna nende hoidmiseks pole vaja ruumi muut ehk RAM mälus (mis on mikrokontrolleritel limiteeritud suurusega)
Näide: deklaratsioonist

```
const char konstant=0x34;
```

Muutuja deklaratsioonidega on seotud veel sellised teemad nagu: viidad, massiivid, biti väljad, andmestruktuurid

Põhi tehted ja avaldised

Omistamine ja avaldiste üldkujud

Omistamise käigus saab omistamis märgi "**=**" vasakpoolne muutuja sama väärtuse parem poolse avaldisega. Näited:

```
muutuja = avaldis;      //üldine kuju
a=2*(a+b);              //Avaldistes kasutatakse sulge, mis määravad tehete järjekorra
a=(3==s);
b=2*funktsioon(c,2)+d/2
//jne…
```

Aritmeetika tehted

Aritmeetika tehted on +,-,*,/,%

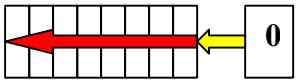
Tehe	Kirjeldus
+	Liitmine, kõigi arvu tüüpide jaoks. Näide: `c=a+d; //kui a=2, d=3 siis c=5`
-	Lahutamine, kõigi arvu tüüpide jaoks. Näide: `c=a-d; //kui a=3, d=2 siis c=1`
*	Korrutamine, kõigi arvu tüüpide jaoks. Näide: `c=a*d; //kui a=3, d=2 siis c=6`

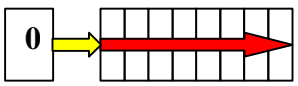
/	Jagamine, kõigi arvu tüüpide jaoks. Näide
	`c=a*d; //kui a=3, d=2 siis c=1 juhul kui täisarvud, ehk täisosa`
	`c=a*d; //kui a=3, d=2 siis c=1.5E00 juhul kui ujukoma arvud`

%	Modulo, täisarvuga jagamisel jääk. `c=a%d; //kui a=7, d=4 siis jääk c=3`

Bittide nihutamine (shift)

Mikrokontrollerite ja üldse riistvara programmeerimise juures on oluline mõista biti operatsioone, mida "päris" arvutite programmeerimise (riistvarast kaugema/abstraktsema) juures nii sagedasti ei kasutata. Mikrokontrolleritel on resursse suhteliselt vähe aga biti operatsioonid suudab kontroller teha riistvaraliselt ja väga kiiresti (võrreldes korrutamise ja jagamisega)
Oluline on tähele panna, et üks biti nihutus võrdub 2 korrutamise või jagamise tehtega, sõltuvalt nihutamise suunast, see võimaldab 2x olulist kiiruse võitu. Teiseks on biti operatsioonid olulised jada koodide genereerimisel ja vastuvõtmisel ja paljudel muudel juhtudel.

<table>
<tr><th>Tehe</th><th>Kirjeldus</th></tr>
<tr><td><<</td><td>Biti nihutus vasakule. (Nihutus 1 võrra võrdub 2 ga korrutamisega.)
Süntaks:
Bait<<nihutus_mitme_võrra
Näide: <code>c=a<<3; /*c võrdub baidiga a, mille bite on nihutatud 3 võrra vasakule*/</code>
Näited: <code>c=a<<n; //Kus c, a, n on char tüüpi, aga nihutada võib ka teisei tüüpe */</code>
<table>
<tr><th>a=</th><th>n=</th><th>Siis c=</th></tr>
<tr><td>0x2E=00101110</td><td>1</td><td>01011100</td></tr>
<tr><td>0x2E=00101110</td><td>2</td><td>10111000</td></tr>
<tr><td>0x2E=00101110</td><td>3</td><td>01110000</td></tr>
<tr><td>0x2E=00101110</td><td>4</td><td>11100000</td></tr>
</table>


</td></tr>
<tr><td>>></td><td>Biti nihutus paremale. (Nihutus 1 võrra võrdub 2 ga jagamisega.)
Süntaks:
Bait>>nihutus_mitme_võrra
Näide: <code>c=a>>3; /*c võrdub baidiga a, mille bite on nihutatud 3 võrra paremale*/</code>
Näited: <code>c=a>>n; //Kus c, a, n on char tüüpi, aga nihutada võib ka teisei tüüpe */</code>
<table>
<tr><th>a=</th><th>n=</th><th>Siis c=</th></tr>
<tr><td>0x2E=00101110</td><td>1</td><td>00010111</td></tr>
<tr><td>0x2E=00101110</td><td>2</td><td>00001011</td></tr>
<tr><td>0x2E=00101110</td><td>3</td><td>00000101</td></tr>
<tr><td>0x2E=00101110</td><td>4</td><td>00000010</td></tr>
</table>


</td></tr>
</table>

Biti kaupa tehted (bitwise)

Mikrokontrollerite programmeerimise juures kõige olulisemad tehted.

AND, OR, XOR, NOT (ehk inverteerimine). Biti kaupa tehete korral võetakse mõlema operandi (väärtused millega tehe sooritatakse) ja tehakse vastavate bitide vahel loogilised tehted. (Ehk kui **c=a&b** siis **c.0=a.0&b.0** ja **c.1=a.1&b.1** jne …)

<table>
<tr><th>Tehe</th><th>Kirjeldus</th></tr>
<tr><td>&</td><td>
AND (ja) tehe. On 1 kui mõlemad pooled on 1. (A&B =1 kui nii A ja B on 1)
<table>
<tr><th>A</th><th>B</th><th>A&B</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>
Süntaks:
A & B
Näited: <code>c=a&b; //Kus a ja b on char, aga võivad olla ka muud tüüpi</code>
<table>
<tr><td>c</td><td>010100100</td></tr>
<tr><td>a</td><td>110100100</td></tr>
<tr><td>b</td><td>01110111</td></tr>
</table>
Näitest paistab ka AND tehte rakendus: teda kasutatakse registrites bittide nullimiseks – clear bit.
Näide:
<code>REG=REG&0xFE; /*0xFE=11111110, selline käsk nullib REG 0nda biti, muud bitid jäävad muutmata. 0xFD muudaks vastavalt 1nda biti, 0xFC nii 0 kui 1 biti jne… (mask) */</code>
</td></tr>
<tr><td>|</td><td>
OR (või) tehe. On 1 kui kas või üks operantidest on 1
<table>
<tr><th>A</th><th>B</th><th>A|B</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>
Süntaks:
A | B
Näited: <code>c=a|b; //Kus a ja b on char, aga võivad olla ka muud tüüpi</code>
<table>
<tr><td>c</td><td>01011000</td></tr>
<tr><td>a</td><td>00111000</td></tr>
<tr><td>b</td><td>01010000</td></tr>
</table>
Näitest paistab ka OR tehte rakendus: teda kasutatakse registrites bitide seadmiseks (1 ks) – set bit.
Näide:
<code>REG=REG|0x01; /*0x01=00000001, selline käsk seab REG 0nda biti, muud bitid jäävad muutmata. 0x02 muudaks vastavalt 1nda biti, 0x03 nii 0 kui 1 biti jne… (mask) */</code>
</td></tr>
<tr><td>^</td><td>
XOR, Exclusive OR (välistav-või) tehe. On 1 kui operandid on erinevad ja 0 kui operandi on samad.
<table>
<tr><th>A</th><th>B</th><th>A^B</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
</table>
</td></tr>
</table>

1	0	1
1	1	0

Süntaks:
A ^ B
Näited: `c=a^b; //Kus a ja b on char, aga võivad olla ka muud tüüpi`

c	11010100
a	01011100
b	10001000

Näitest paistab ka XOR tehte rakendus: teda kasutatakse registrites bittide inverteerimiseks (ehk oleku vastupidiseks pööramiseks – ***toggle bit***)

Näide:
```
REG=REG^0x01; /*0x01=00000001, selline käsk inverteerib REG 0nda biti, muud bitid jäävad muutmata. 0x02 muudaks vastavalt 1nda biti, 0x03 nii 0 kui 1 biti jne… (mask) */
```

~ — **NOT (inverteerimis)** tehe. Muudab bitide olekud vastupidiseks

A	~A
0	1
1	0

Süntaks:
~A
Näited: `c=~a; //Kus a on char, aga võib olla ka muud tüüpi`

c	11000110
A	00111001

See tehe ei leia nii suurt kasutust kui eelmised

Võrdlus tehted (relational operators and equality)

Võrdlus tehted on arvude võrdlemiseks, kas üks arv on teisest suurem, väiksem, võrdne. Need tehted tagastavad alati väärtuseks kas 0 või 1 ehk tõeväärtus (tõene – ***true*** või väär - ***false***). Erinevalt paljudest keeldest (nagu C++, Java, Pascal) C **ei oma** sellist andmetüüpi nagu **bool (boolean)**, selle asemel tagastavad need tehted väärtusi **täisarvu tüüpi** (kas char, int, bit field jne). Tõeväärtuse väärale olekule vastab olek 0 ja tõesele 1 (hiljem loogika tehete juures selgub, et kõik 0 st erinevad väärtused on sisult tõesed)

Tehe	Kirjeldus
= =	**Võrdub**. Võrdleb kahte arvu, tagastab tõese väärtuse, kui arvud on võrdsed Süntaks: **A= =B** Näited: `c=a==b; /*Võrdleb a ja b. Kui a ja b on samad siis c=1, vastasel korral c=0, c on täisarvu tüüpi (int, char)*/`
!=	**Ei võrdu**. Võrdleb kahte arvu, tagastab tõese väärtuse, kui arvud on erinevad Süntaks: **A!=B**

a	b	c
2	3	**0**
4	4	**1**
6	5	**0**

Näited:
```
c=a==0x34; d=(e==-56); //jne
if(a==b){  //Tingimus lauses
}
```

<table>
<tr><td></td><td>
Näited: <code>c=a!=b; /*Võrdleb a ja b. Kui a ja b on erinevad siis c=1, kui samad siis c=0, c on täisarvu tüüpi (int, char)*/</code>
<table>
<tr><th>a</th><th>b</th><th>c</th></tr>
<tr><td>2</td><td>3</td><td>1</td></tr>
<tr><td>4</td><td>4</td><td>0</td></tr>
<tr><td>6</td><td>5</td><td>1</td></tr>
</table>
Näited:
<code>c=g+a!=0x34; d=(e!=-56); //jne
if(a!=b){ //Tingimus lauses
}</code>
</td></tr>
<tr><td><</td><td>
Väiksem, kui. Võrdleb kahte arvu, tagastab tõese väärtuse, kui märgi vasakul poolel olev arv on väiksem, kui paremal poolel olev.
Süntaks:
A<B
Näited: <code>c=a<b; /*Võrdleb a ja b. Kui a on väiksem kui b siis c=1, vastasel korral c=0, c on täisarvu tüüpi (int, char)*/</code>
<table>
<tr><th>a</th><th>b</th><th>c</th></tr>
<tr><td>2</td><td>3</td><td>1</td></tr>
<tr><td>4</td><td>4</td><td>0</td></tr>
<tr><td>6</td><td>5</td><td>0</td></tr>
</table>
Näited:
<code>c=g+a<0x34; d=(e<-56); //jne
if(a<b){ //Tingimus lauses
}</code>
</td></tr>
<tr><td>></td><td>
Suurem, kui. Võrdleb kahte arvu, tagastab tõese väärtuse, kui märgi vasakul poolel olev arv on suurem, kui paremal poolel olev.
Süntaks:
A>B
Näited: <code>c=a>b; /*Võrdleb a ja b. Kui a on suurem kui b siis c=1, vastasel korral c=0, c on täisarvu tüüpi (int, char)*/</code>
<table>
<tr><th>a</th><th>b</th><th>c</th></tr>
<tr><td>2</td><td>3</td><td>0</td></tr>
<tr><td>4</td><td>4</td><td>0</td></tr>
<tr><td>6</td><td>5</td><td>1</td></tr>
</table>
Näited:
<code>c=g+a>0x34; d=(e>-56); //jne
if(a>b){ //Tingimus lauses</code>
</td></tr>
<tr><td><=</td><td>
Väiksem või võrdne. Võrdleb kahte arvu, tagastab tõese väärtuse, kui märgi vasakul poolel olev arv on väiksem, kui paremal poolel olev, või võrdne sellega
Süntaks:
A<=B
Näited: <code>c=a<=b; /*Võrdleb a ja b. Kui a on väiksem kui b või võrdne b ga siis c=1, vastasel korral c=0, c on täisarvu tüüpi (int, char)*/</code>
<table>
<tr><th>a</th><th>b</th><th>c</th></tr>
<tr><td>2</td><td>3</td><td>1</td></tr>
<tr><td>4</td><td>4</td><td>1</td></tr>
<tr><td>6</td><td>5</td><td>0</td></tr>
</table>
Näited:
<code>c=g+a<=0x34; d=(e<=-56); //jne
if(a<=b){ //Tingimus lauses
}</code>
</td></tr>
<tr><td>>=</td><td>
Suurem või võrdne. Võrdleb kahte arvu, tagastab tõese väärtuse, kui märgi vasakul poolel olev arv on suurem, kui paremal poolel olev, või võrdne sellega
Süntaks:
A>=B
Näited: <code>c=a>=b; /*Võrdleb a ja b. Kui a on suurem kui b või võrdne b ga siis c=1, vastasel korral c=0, c on täisarvu tüüpi (int, char)*/</code>
</td></tr>
</table>

a	b	c
2	3	**0**
4	4	**1**
6	5	**1**

Näited:

```
c=g+a>=0x34; d=(e>=-56); //jne
if(a>=b){  //Tingimus lauses
}
```

Loogilised tehted (logical)

Loogilised tehted on tehted, mille operantideks on tõeväärtused ja tulemuseks tõeväärtus (0, 1). Operandid loetakse tõeseks kui nad erinevad 0 st (tüüpiliselt 1) ja vääraks kui nad on võrdsed 0 ga. Andmetüüpideks on harilikult täisarvu tüübid.

Tehe	Kirjeldus
&&	**AND (loogiline JA)**. Tehe tagastab väärtuse, mis on tõene, kui mõlemad operandid on tõesed. Süntaks: **A&&B** (Tõene kui A JA B on tõesed)
||	**OR (loogiline VõI)**. Tehe tagastab väärtuse, mis on tõene, kui kas või operantidest on tõene Süntaks: **A || B** (Tõene kui kas A või B on tõene)
!	**NOT (loogiline EITUS)**. Tehe muudab tõeväärtuse vastupidiseks. Ehk tõese vääraks ja väära tõeseks. Süntaks: **!A** (Tõene, kui A on väär)

Näited loogiliste tehete kasutamisest

```
c=a&&0x0A; /*0x0A on 0 st erinev (st. tõene) seega kogu avaldise tõesus sõltub ainult a
st, ehk see on samane avaldisega */
c=a;
c=a&&0x00; //Kuna üks pool on 0 (väär) siis kogu avaldis on alati väär, sõltumata a st
c=0;    //Eelmine avaldis on sama väärne sellega

c=(a&&b)||d;
/* c on tõene kui loogilise VõI tehte kas või ükski pool on tõene, ehk siis kui d on
tõene või kui (a&&b) on tõene. a&&b on omakorda tõene kui nii a kui ka b on tõesed
Muutuja väärtus on tõene, kui ta on nullist erinev */

c=((!a)&&b)||(a&&(!b));
/*On tõene, kui a ja b pole korraga samas tõeväärtus olekus, ehk XOR tehe
Vaatame, kuidas see tuleb: c on tõene, kui kas || tehte vasak või parem pool on tõene.
Vasak pool on tõene kui && tehte nii parem kui vasak pool on tõesed, ehk b on tõene ja a
on väär (kuna a tõeväärtus on pööratud). Paremal poolel on aga täpselt vastupidi, selle
tõesuseks on vajalik, et a oleks tõene ja b väär.*/

c=(a<5)&&(a>0);
/*c on tõene kui a asub 0 ja 5 vahel. Vaatame, kuidas see tuleb: selleks, et c oleks
tõene peavad nii && (ja) tehte parem kui vasak pool olema tõesed. Vasak pool on tõene,
kui a on väiksem 5 st, parem aga siis kui a on suurem 0 st.*/
c=!((a>=5)||(a<=0)); //See on samane eelmise avaldisega, võib veenduda, et see on nii...

if((a<5)&&(a>0))
{  //Seda tingimus lause osa täidetakse siis kui a asub 0 ja 5 vahel
   //...
}
```

Tingimuslik avaldis (conditional operator)

Süntaks:

```
d=a?b:c;
```

Mida see teeb?

See on tingimuslik avaldis, mis saab vastavalt a (loogiline avaldis) tõesusele väärtuseks kas b (kui a on tõene) või c (juhul kui a on väär)

Näited:

```
d=a?-5:5; /*d saab väärtuseks -5, kui a on nullist erinev (tõene) või
5, kui a on null (väär)*/
d=(a<0)?1:b;  /*d saab väärtusks 1, kui a on 0 st väiksem või b, kui
see tingimus pole täidetud*/
```

Avaldiste lühendid

Teatud eriti palju kasutamist leidvatel omistus lausetel on C olemas lühendid. Mida tasub meeles pidada, kuna neid rakendatakse praktikas väga sagedasti.

Pikalt	Lühendatult
A=A+B	A+=B
A=A-B	A-=B
A=A*B	A*=B
A=A/B	A/=B
A=A%B	A%=B
A=A<<B	A<<=B

A=A>>B	A>>=B
A=A&B	A&=B
A=A|B	A|=B
A=A^B	A^=B
A=A+1	A++ või ++A
A=A-1	A—- või --A

Post (pärast) ja pre (enne) increment (kasvatamine) ja decrement (kahandamine)

Arvude kasvatamiseks ja kahandamiseks ühe võrra on Cs veel eraldi konstruktsioonid

A++, ++A, A--, --A

Mis neil vahet on?

Avaldis	On samaväärne…	Kirjeldus
B=A++;	B=A; A=A+1;	**Post increment**, ehk enne võetakse väärtus ja siis kavatatakse.
B=++A;	A=A+1; B=A;	**Pre increment**, ehk enne kasvatatakse väärtus ja seejärel omistatakse.
B=A--;	B=A; A=A-1;	**Post decrement**. Analoogiline ainult, et negatiivses suunas.
B=--A;	A=A-1; B=A;	**Pre decrement**

Näited:

Olgu

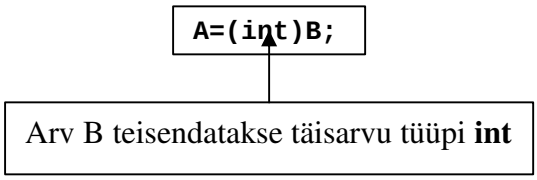
A=1; B=5;

Muutujate algväärtused

Avaldis	Muutujate lõppväärtused
C=(A++)*B;	A=2, B=5, C=5
C=(++A)*B; C=(A=A+1)*B; //Sama	A=2, B=5, C=10
C=(A--)*B;	A=0, B=5, C=5
C=(--A)*B; C=(A=A-1)*B; //Sama	A=0, B=5, A=0

Tüübi teisendused

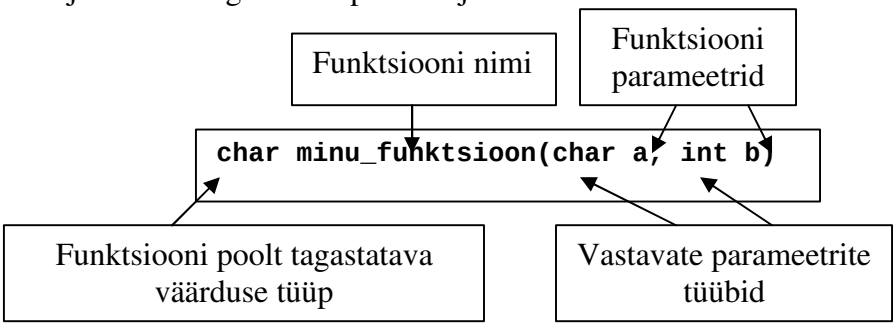
Tüübi teisendused on selleks et andmeid viia ühest tüübist teise. Näide:



Tüübi teisendusi kasutatakse erinevat tüüpi andmete vahelistes üleminekutes, aga nende kasutamine nõuab ka programmeerijalt täiendavat tähelepanu, sest kõiki tüüpe nii teisendada ei saa. Näiteks: stringist numbrit, selleks tuleb kasutada eri funktsiooni. Kompileerimine võib küll vigadeta laabuda, aga tulemus ei ole see mida oodatakse.

Funktsiooni deklaratsioon ja kasutus

Funktsioon on programmeerimise üks olulisemaid konstruktsioone, see võimaldab programmi struktureerida ja mitmes kohas tehtavaid samu tegevusi kokku võta üheks funktsiooniks.
Kuidas funktsioone teha?
Funktsiooni kirjeldamine algab tema päise kirjutamisest



Päis kirjeldab ära **funktsiooni nime**, kui palju ja mis tüüpi (samuti mis järjekorras) on **parameetrid** (väärtused mida saab funktsiooni väljakutsumisel talle ette anda Näiteks: ln(x) funktsiooni parameeter on x) ja selle mis tüüpi on funktsiooni poolt tagastatav väärtus.
Kui funktsioonil ei ole parameetreid siis kirjutatakse parameetrite koha peale **void**, mis tähendab parameetri puudumist (void on üks spetsiifiline abi andmetüüp Cs)

```
int minu_funktsioon(void)
```

Kui funktsioon ei tagasta mingit väärtust (Pascali mõttes on see **procedure**) siis kirjutatakse samuti tagastatava väärtuse tüübi koha peale **void**

```
void sinu_funktsioon(int parameeter_a, int param_b, char jne_c)
```

Funktsiooni päise järele kirjutatakse funktsiooni keha – see kus midagi tehakse

```
void sinu_funktsioon(int parameeter_a, int param_b, char jne_c)
{
      //Siia tuleb funktsiooni keha, ehk see mida tehakse
}
```

Kuidas funktsioon tagastab väärtuse ja lõpetab töö?
Funktsiooni töö lõpetamine ja väärtuse tagastamine käib **return** käsuga. väärtus, mida soovitakse tagastada kirjutatakse sõna return järgi. Returni saab kasutada ka nende

funktsioonide puhul, mis väärtust ei tagasta, siis ei kirjutata return i järgi mitte midagi. Return ei pea olema viimane käsk funkstiooni kehas, samuti võib teda esineda seal mistahes arvul. Funktsioon lõpetab ka oma töö siis, kui programmi täitmine jõuab funktsiooni keha lõpuni.

```
char sinu_funktsioon(int parameeter_a, int param_b, char jne_c)
{
      //Siia tuleb funktsiooni keha, ehk see mida tehakse
      //Käsud ...
      return tagastatav_vaardus;
}
```

Funktsiooni deklareerimine

Et funktsiooni saaks kasutada, selleks tuleb lähteteksti faili algusesse funktsiooni päis (ilma kehata) uuesti välja kirjutada. Sellel juhul kirjutatakse veel päise järgi '**;**' (semikoolon). Näide:

```
int minu_funktsioon(void);
```

Näited:

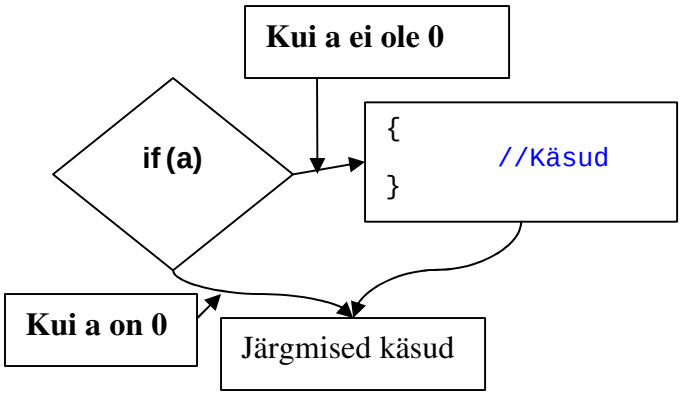
```
//Faili algus
int faktoriaali_leidmine(int parameeter); //Deklaratsioon

void main(void)
{       int a, c, d;
        a=6;
        c=2*faktoriaali_leidmine(a); //Kasutamine c=1440
        d=faktoriaali_leidmine(a+1); //Kasutamine d=5040
        while(1); //Mikrokontrollerite korral ei tohi main meetod kunagi ära lõpeda
}

int faktoriaali_leidmine(int parameeter)
{
        int e=1; //Funktsiooni sisemine (lokaalne) muutuja
        while(1) //Lõputu tsükkel
        {
                e*=parameeter--; //Iga kord korutame ja vähendame parameetrit
                if(parameeter==0){ return e; } /*Kui parameeter on 0 siis tagastame
                                                tulemuse ja lõpetame*/
        }
}
```

Tingimus laused (if… else)

Tingimus lausetega saab valida milliseid käske täidetakse sõltuvalt tingimuse tõesusest **if tähenda, kui tõene siis … (teeme järgnevat, if-i järel olev blokk)**

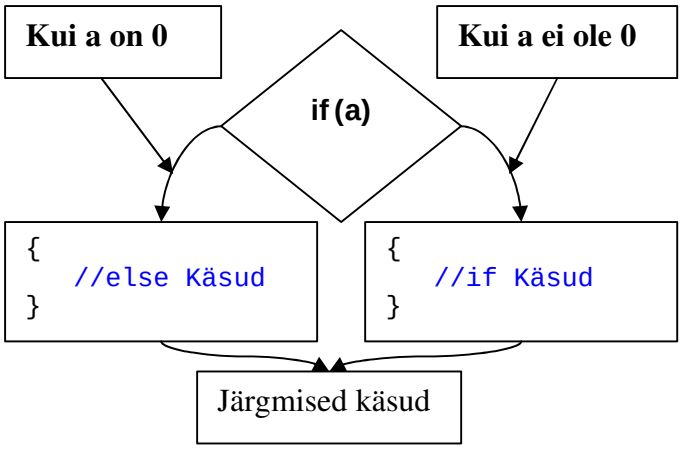
if lause

```
if(a)
{
//Käsud
/*Nende sulgude sees olevad
käsud täidetakse ainult siis,
kui a on tõene (ehk 0 st
erinev)*/
}
//Järgmised käsud
```



if…else lause
if … else tähendab, <u>kui</u> tõene siis teeme …(if-i järel olev blokk) aga <u>muidu</u> teeme … (else järel olev blokk)

```
if(a)
{
//Käsud
/*Nende sulgude sees olevad
käsud täidetakse ainult siis,
kui a on tõene (ehk 0 st
erinev)*/
}else
{
//Käsud
/*Nende sulgude sees olevad
käsud täidetakse ainult siis,
kui a ei ole tõene (ehk 0)*/
}
//Järgmised käsud
```



Näited:

```
if(a>0) a--; b++;   /*Kui a>0 siis tehakse a--. b++ tehakse a väärtusest
sõltumatult*/

if(a>0){ a--; b++; } c++; /*Kui a>0 siis tehakse a–ja b++. c++ tehakse
a st sõltumatult*/

if(a<0){ a++; }else{a--; } b++; /*Kui a<0 siis tehakse a++ muul juhul (
ehk a>=0) tehakse a--. b++ tehakse a väärtusest sõltumata.*/

if(!a){a=100;}
else if(a<0){a++;}
else{a--; }
b++;
/*Selles näites on kolm erinevat võimalust
1) Kui !a, mis on samaväärne a==0 sellisel juhul a=100 ja järgmine käsk
on b++ jne ..
2) Kui a<0, sellisel juhul tehakse a++ ja järgmine käsk on b++ jne …
```

```
3) Muul juhul (ehk siis kui a>0), siis
tehakse a-- ja järgmine käsk on b++
jne..*/
```

Käsud mis täidetakse kui **a** on **1**

```
switch(a)
{     case 1: b=2; c=4; break;
      case 2: b=1; c=5; break;
      case 3: b=0; c=9; break;
      default: b=0; c=0;
            break;
}
/*Järgmised käsud – lülitus
lause lõpp*/
```

Lülitus laused (switch … case)

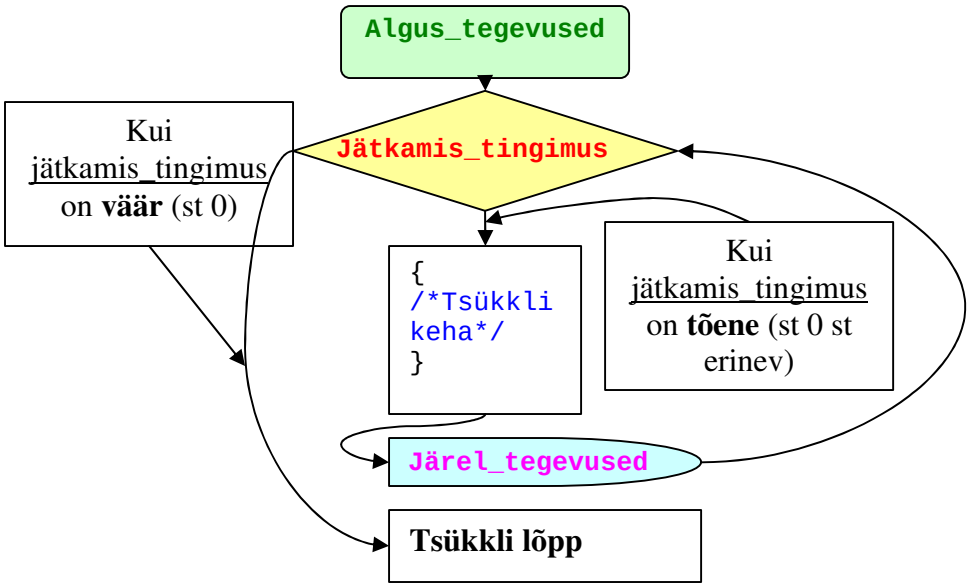
Lülitus lause võimaldab valida tegevusi vastavalt avaldise/muutja väärtusele. **switch** *(lüliti)* võtmesõna järel on kas avaldis või muutuja. (Näites on see *a*), mille väärtuse järgi hakatakse valima. Iga **case** *(juhtum)* märksõna järel on väärtus (Näites 1, 2, 3), kui *a* väärtus langeb mõnega neist kokku, siis täidetakse vastava **case** järel olevad käsud. **break** lõpetab lülitus lause (bloki), kui jätta break kirjutamata siis täidetakse ka kõikide järgnevate case ide järel olevad käsud kuni **breakini** või lülitus lause lõpuni. **default**'i järel olev täidetakse siis, kui *a* väärtus ei esine case'ide seas.
Näites:

Kui a on ..	Siis b saab olema ..	Siis c saab olema ..
1	2	4
2	1	5
3	0	9
Muude väärtuste korral	0	0

For - tsükkel

Tsükkel, nagu ka nimi juba ütleb, on mingi korduvalt täidetav koodi osa, nii ka for tsükkel. Seda tsüklit kasutatakse enamasti loendus tsüklina.

```
//Süntaks:
for(algus_tegevused; jätkamis_tingimus;järel_tegevused)
{
//Tsükli keha, kood mida täidetakse
}
//Tsükli lõpp
```



C keeles for tsükkel on paljude teiste keeltege (Pascal, BASIC), kus see on enamasti piiratud ainult loendamisfunktsiooniks, ülivõimas.
Näited:
for tsükli peamine kasutus – lihtsa loendus tsüklina. Näiteks on faktoriaali leidmine

```
n=3; //Faktoriaal mitmest
faktoriaal=1;
for(a=1; a<=n; a++)
{ faktoriaal*=a; } //Sama mis faktoriaal=faktoriaal*a;
//Tulemus on faktoriaal=6
```

Kuidas see toimib?

1) Tsüklisse sisenemisel seatakse **a=1** siis
2) Kontrollitakse jätkamistingimust, kas **a<=n**, kuna 1<=3 siis läbitakse tsükli keha
3) **faktoriaal=faktoriaal*a**, ehk faktoriaal=1*1=1
4) Siis tehakse järel tegevus, ehk **a++**, siis a=2
5) Kontrollitakse jätkamistingimust, kas **a<=n**, kuna 2<=3 siis läbitakse tsükli keha
6) **faktoriaal=faktoriaal*a**, ehk faktoriaal=1*2=2
7) Siis tehakse järel tegevus, ehk **a++**, siis a=3
8) Kontrollitakse jätkamistingimust, kas **a<=n**, kuna 3<=3 siis läbitakse tsükli keha
9) **faktoriaal=faktoriaal*a**, ehk faktoriaal=2*3=6
10) Siis tehakse järel tegevus, ehk **a++**, siis a=4
11) Kontrollitakse jätkamistingimust, kas **a<=n**, kuna 4 ei ole <=3 siis tsükkel lõpetatakse.
12) Tulemuseks on 6, mis on õige vastus.

For-tsükkel aga võib olla ka palju keerukam. Näiteks:

```
for(a=0, b=5; (b>0)&&(a!=10); a+=2, b--, i=a*b)
{ }
```

Mis see on???

- For tsüklis võib olla mitmeid algus tegevusi, siis need eraldatakse omavahel "," (komadega). Siin kohal on need `a=0, b=5`
- Jätkamis tingimus võib olla mistahes loogiline avaldis. Siin kohal jätkatakse kui `(b>0)&&(a!=10)` ehk b>0 ja a ei ole 10
- For tsüklis võib olla mitmeid järel tegevusi, siis need eraldatakse omavahel "," (komadega). Siin kohal on need `a+=2, b--, i=a*b`

```
for(a=5; a; a--)
```

Mis see on???

See on loendus tsükkel, kus loendatakse alla suunas 5,4,3,2,1, sest kui a saab võrdseks 0-ga (0 on väär jätkamise tingimus) siis tsükkel katkestatakse

```
for(a=0;; a++)
```

Mis see on???

See on loendus tsükkel, kus loendatakse positiivses suunas. Kuna aga jätkamis tingimust pole, siis on see lõputu tsükkel, seda tuleb katkestada mingil muul moel näiteks võtme sõnaga **break**, vt break.

```
for(;;)
```

Mis see on???

See on lõputu tsükkel, mis on samaväärne näiteks **while(1)** ga. Puuduvad nii algus tegevus, jätkamise tingimus, kui ka järeltegevus. Algus tegevus võib sisuliselt olla realiseeritud enne for tsüklit ja järel tegevus võib olla tsükli kehas.

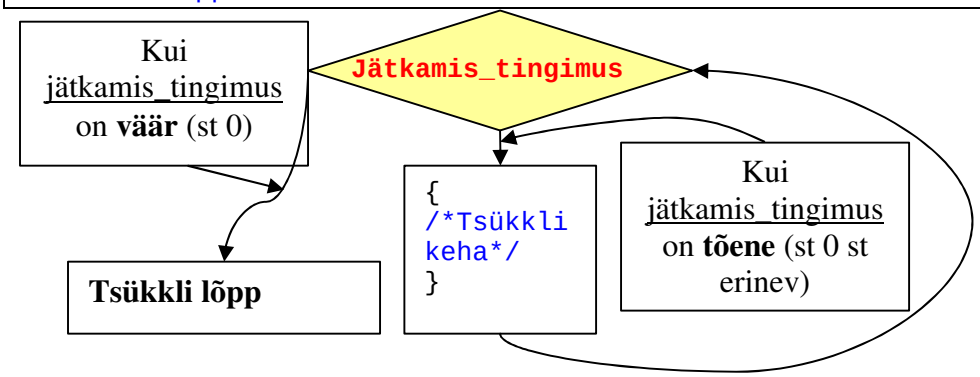
Alguses toodud faktoraali võib realiseerida ka nii …, üldse ilma tsükli kehata.

```
n=3; faktoriaal=1;
for(a=n;a;faktoriaal*=(a--)); /*Kui ühtegi käsku pole siis on (;) märk
otse for tsükli järel ... Mõtle kuidas see näide töötab ☺ */
```

while-tsükkel

while tsükkel on **for** tsükli lihtsustatud juht. For tsüklist saaks while tsükli kui tühjaks jätta, nii algus kui järel tegevused, ehk **for(;jätkamis_tingimus;)**

```
//Süntaks:
while(jätkamis_tingimus)
{
//Tsükli keha, kood mida täidetakse
}
//Tsükli lõpp
```



Näited:

Sama faktoriaali arvutus while tsükliga

```
n=3; faktoriaal=1;
while(n) //Korda tsüklit nii kaua kuni n saab 0 ks
{     faktoriaal*=n;
      n--; //Vähenda iga kord n 1 võrra
}
```

Sama võib kirjutada ka kompaktsemalt

```
n=3; faktoriaal=1;
while(n) faktoriaal*=(n--);
```

```
/*Kui on üks käsk siis võib selle kirjutada ilma sulgudeta. Tsükli
"mõju piirkond" ulatub kuni (;) märgini*/
```

Aga mitte nii …(vigane näide)

```
n=3; faktoriaal=1;
while(n) faktoriaal*=n; n--;
/*n-- ei ulatu tsüklisse, seega seda ei muudeta → tegu on lõpmatu
tsükkiga, mis ei arvuta faktoriaali*/
```

do … while-tsükkel

do … while tsükkel on peamiselt nagu **while** tsükkel, ainult selle vahega, et jätkamis tingimust kotrollitakse pärast tsükli keha täitmist, mitte enne nagu while tsüklis. Selline tsükkel läbitakse enne kindlasti üks kord ja siis vaadatakse kas jätkamis tingimus on täidetud ja kas veel läbitakse või lõpetatakse.

```
//Süntaks:
do
      //Tsükli keha, kood/laused mida täidetakse
      //asub siin do ja while vahel
while(jätkamis_tingimus); //while järel on ;
//Tsükli lõpp
```

Tsüklite kontroll – break, continue

break – tsükli katkestamine

Tsüklis olles on võimalik tsükli täitmist ka muul viisil katkestada, kui oodata selle lõpu tingimust. Funktsioonist tagasi pöördumise, **return** käsu, kasutamisel tsüklid katkestatakse. Samuti on tsüklit võimalik katkestada **break** käsuga, selle järel jätkatakse tsüklile järgnevate käskude täitmist.

Näide:

```
for(;;)
{  if(a>10){ break; }
   a+=2;
}
a=2; //Tsüklile järgnevad käsud …
```

break käsu järel jätkatakse tsükkli järel olevate käskude täitmisega.

break käsku saab kasutada kõigi kolme tsükli tüübi korral.

Näide: faktoriaali arvutamine lõpmatu tsükliga

```
faktoriaal=1; a=1; n=3;
for(;;) //Lõputu tsükkel, sama oleks ka while(1)
{
      faktoriaal*=a; //faktoriaali arvutus
      if(a>=n){ break; } //Kui a väärtus saab võrdseks n siis lõpeta
      a++; //Iga kord suurenda a väärtust
}
```

continue – tsükli jätkamine

continue käsuga saab minna järgmisele tsükli ringile. Sisuliselt minnakse tsükli keha lõppu, siis kontrollitakse uuesti tsükli jätkamise tingimust (for puhul tehakse enne seda veel järel tegevused) ning kui jätkamise tingimus on täidetud minnakse uuele ringile.

Näide:

```
while(a) //Kontrollitakse jätkamis tingimust
{
      if(a>10) { continue; } //Kui a>10 mine järgmisele ringile
      a+=3;
      //…
   //Tsükli keha lõpp, sealt minnakse uuesti algusesse ..
}
```

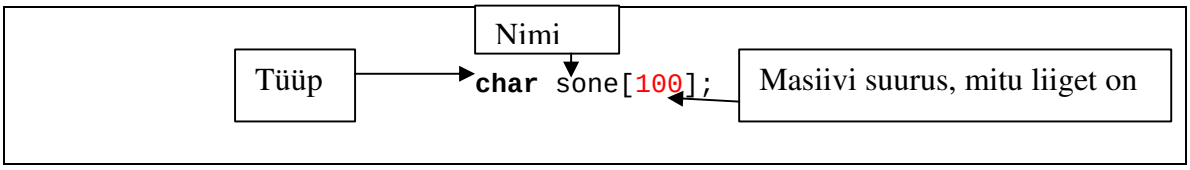
Massiivid

Massiivid on olulised andme objektid, kus hoitakse suurt hulka sama tüüpi andmeid. Andmete kätte saamine ja panemine massiivi toimub numbriliste indeksite järgi.

Kuidas massiive deklareerida?

Üldiselt kehtivad massiivide deklareerimisel ja kasutamisel samad reeglid, mis tavaliste muutujate korral, välja arvatud mõned erandid - nagu, massiive ei saa omavahel võrrelda (sisu järgi), neid ei saa omistada, funktsioonid ei tagasta massiive jne.

Näide massiivi deklareerimisest



Massiivide kasutamine???

char tüüpi massiivid on eriti palju kasutust leidvad kuna neis hoitakse sõnesid (***string***), mis on tegelikult char jaga (iga char sisaldab ühe märgi ASCII koodi, vt. ascii koodi tabelist)

Cs on massiivi esimese liikme indeks 0 ja viimasel MASSIIVI_SUURUS-1

Näited: Suurima elemendi leidmine massiivist

```
char massiiv[100]; // Massiiv
/*Tuleb tähelepanna, et Cs algavad indeksid 0 st ja kui me reserveerime
endale massiivi suurusega 100 elementi, siis selle massiivi viimase
elemendi indeks on 99*/

suurim=massiiv[0]; //Oletame et suurim asub pesas “0”
for(a=0; a<100; a++) //Käime tsükliga kõik pesad 0..99 läbi
{
    //a saab väärtusi 0,1,2, .... 98, 99
    if(suurim<massiiv[a]) { suurim=massiiv[a]; }
    /*Kui leiame et mõni element on suurem kui suurim, siis asendame
      suurima, uue leitud suurimaga*/
}
```

Andmete viidad

Viit ***(pointer)*** on andmetüüp, mis sisaldab mingit mälu aadressi. Viidad on samuti väga olulised.

Kuidas ~~viidatüüpi~~ deklareerida?

```
char  *viit_charile;
```

Kui soovitakse deklareerida viida tüüpi siis pannakse muutuja nimele ette * (tärn).
Viitasid kasutatakse täpselt samamoodi nagu kõiki tavalisi muutujaid – neid saab omistada, võrrelda (kas nad viitavad samale aadressile), funktsioonid võivad tagastada viitu jne.
Võrdlus näide:

Tavalised muutujad	Viidad

char a=2;
char c=3;
c=a;
c=5;
//Lõpuks on väärtused a=2 ja c=5
	char a=2;
	char *c; //viit char ile
	c=&a; /* & märk muutuja ees annab selle muutuja addressi mälus. Omistame selle addressi viidale. Nüüd viitab viit c muutujale a */
	c=5; / Omistame c ga viidatud addressile ehk a le
	Lõpuks on a=5 ja c= a address kontrolleri mälus */

Viitade kasutamise eri omadused

Näited:

```
/*Olgu muutujad:
 *c address 4000
 *a address 4004 */
int *c;           //2 baidine
int a;            //2 baidine
c=0x4003    /*Omistamine
            viidale, st. */
a=c;        /* a=0x4003
  see on c mälu pesa sisu*/
a=*c;       /* a=0x4006
see on selle mälu pesa sisu,
mis asub addressil (0x4003), mis
on c mälu pesas*/
a=&c;       /* a=0x4000
see on c mälu pesa address*/
c=&a;  /* c=0x4004, omistame c
mälu pesale a mälu pesa
addressi, c viitab nüüd a le*/
*c=0x1111; /* c=0x4004 ja
a=0x1111, ehk omistame
mälupesale, mille address on c
mälupesas*/
c+=2; /*Suurendame c 2 võrra
nüüd c=0x4006*/
*c+=2;/*suurendame c ga viidatud
mälu pesa ehk addressil 0x4006
olevate andmete väärtust 2 võrra
enne oli 0x5650 nüüd siis
0x5652*/
```

Mälu pesa address	Mälu pesa sisu (word)
0x4000 (c)	0x4003
0x4002	0x3456
0x4003	0x4006
0x4004 (a)	0x2B34
0x4006	0x5650

Viidad ja massiivid

```
char a[100]; //a on konstantne viit, massiivi mälu algusele
char *a;
c=a; /*Mis on samaväärne c=&a[0], &a[0] tähendab massiivi esimese
elemendi address, ehk massiivi algus address*/
//c viitab pärast seda tehet massiivi algusele
//Pärast seda …
    *c ja a[0] on samaväärsed
```

```
     *(c+1) ja a[1] on samaväärsed jne.
     c+1 on samaväärne &a[1] jne…
```

Viidad ja dünaamiline mälu haldamine.

Viitadega on korraldatud ka dünaamiline mälu haldamine, ehk juhud, kus alguses pole teada, kui palju mälu on vaja vaid seda otsustatakse programmi töö ajal ja siis küsitakse mälu

Näide:

```
//Mälu haldamis funktsioonid on library's stdlib
#include <stdlib.h>

const char MALU_MAHT=30; //Mälu maht baitides
char *a;
a=(char*)malloc(MALU_MAHT);
/*malloc funktsioon otsib ettenähtud hulgal vaba mälu, kui nii suurt
mälu bloki ei õnnestunud leida, siis tagastab tagastab 0, muidu mälu
bloki algus addressi*/
if(!a)
{
      //Viga nii suurt vaba mälu bloki polnud
}
//Muidu
//Täida saadud mälu blok 0 dega
for(i=0; i<MALU_MAHT; i++) *(c+i)=0;
//Pärast kasutust tuleb mälu uuesti vabastada funktsiooniga free
free(a);
```

Funktsioonide viidad

Samuti nagu on võimalik viidata andme mälu väljadele on võimalik viidata ka funktsioonidele. Kuigi funktsioonidele viitamine leiab mõnevõrra vähem kasutust on see siiski vahest kasulik võimalus realiseerida C hilist seostamist (ütleb midagi neile kes on tuttavad objekt orienteeritud programmeerimise (Javaga)). Funktsiooni viit nagu ka andme viit on 2 baiti pikk ja viitab ta funktsiooni algusele mälus. Funktsiooni viit on muutuja nagu tavaline viit või muutuja, temaga on võimalikud samad operatsioonid, võrdlemine, omistamine, võib olla funktsiooni parameeter, funktsioon võib selle tagastada.

Näide:

```
int liida(int a, int b) { return a+b; }  //Funkstioonid
int lahuta(int a, int b) { return a-b; } //Funktsioonid

void kasutab(char a)
{     //Kasutame hilist seostamist
      int (*tehe)(int, int); //Deklareerime funktsiooni viida

      //Seame viidale vastava funktsiooni, millele me tahame viidata
      switch(a)
      {     case '+': tehe=&liida; break;
            case '-': tehe=&lahuta; break;
      }
      //Nüüd võib kasutada
/*Kui a oli '+' siis on tehe liida funktsioon, kui '-' siis lahuta
funktsioon*/
      tehe(2,3);
```

```
      tehe(5,6);
}
//Nii saab tööajal valida millist konkreetset funktsiooni selle nime
//all (tehe) kasutada
```

Funktsiooni viidad võivad olla funktsiooni parameetrid

```
int liida(int a, int b) { return a+b; }  //Funkstioonid
int lahuta(int a, int b) { return a-b; } //Funktsioonid

int arvuta(int (*mis_funktsiooniga)(int, int), int a, int b)
{
      return mis_funktsiooniga(a, b);
}

//Kasutus ..
arvuta(&liida,2,3); //Tagastab väärtuseks 5
arvuta(&lahuta,3,1); //Tagastab väärtuseks 2
```

Funktsioon võib tagastada viida funktsioonile

```
int (*saa_tehe(char funktsiooni_parameeter))(int, int)
{     switch(a)
      {     case '+': return &liida; break;
            case '-': return &lahuta; break;
      }
}
//Kasutamine
int (*tehe)(int, int);
tehe=saa_tehe('+');
tehe(2,3); //Tagastab tulemuseks 5
```

Funktsiooni viitadest võib koostada massiive

```
typedef int (*f_viit)(int, int); //Defineerime tüübi
//Deklareerime massiivi nagu harilikult
f_viit massiiv[2];
massiiv[0]=&liida;
massiiv[1]=&lahuta;
massiiv[0](3,2); //Tagastab 5
massiiv[1](3,2); //Tagastab 1
```

Struktuurid ja tüübidefineerimine – typedef, struct

struct

Päris tihti tuleb ette olukordi, kus andmeid oleks mõtekas struktureerida, parema ülevaatlikuse ja lühema ning loetavama koodi huvides. **struct** konstruktsiooni ongi selleks, et luua andmestruktuure ja andmeid grupeerida.
Näide: oletame, et meil on 3 sammmootorit, igaühe juhtimiseks on vaja kahte parameetrit kiirus ja suund. Siis üks võimalus on deklareerida muutujad nii …

```
char M1_kiirus;
char M2_kiirus;
char M3_kiirus;
char M1_suund;
char M2_suund;
char M3_suund;
```

Hulka kompaktsem ja elegantsem on seda teha aga nii …

```
struct {
      char kiirus, suund; } M1, M2, M3;

/*Ehk defineerisime uue andme tüübi, mis on struktuur, mis sisaldab
kiirust ja suunda. Muutujad M1, M2, M3 on sellest tüübist*/
//Sellist tüüpi andmete kasutamine käib nii...
//M1 suund on
M1.suund //See on samaväärne esimeses näites olnud M1_suund muutujaga
//Näide:
M1.suund=tagasi;
M1.kiirus=200;
M2.kiirus=M3.kiirus+2; //M2 liigub 2 võrra kiiremini kui M3
//jne..
```

typedef

typedef võimaldab defineerida sootuks uusi andmetüüpe (näiteks struktuuridest), mida siis edaspidi saab kasutada suhteliselt sarnaselt harilike tüüpidega, nagu char, float jt. Näited:

```
//Defineerime uue tüübi nimega Mootor
typedef struct { char kiirus, suund; } Mootor;
//Kasutame seda tüüpi deklareerimisel
Mootor M1, M2, M3;
Mootor Massiiv[3]; //Massiiv mille elemendid on tüübist Mootor
Mootor *M;  //Viit sellele tüübile
//Kasutamine
M1.kiirus=M2.kiirus-10; //M1 liigub 10 võrra aeglasemalt, kui M2
M3.suund=!M2.suund; //M3 liigub vastas suunas M2 võrreldes
//Massiivi kasutamine
for(i=0; i<3; i++) Massiiv[i].kiirus=10*i;
//Viida kasutamine
M=&M1;
//Nüüd M viitab M1 le
(*M).kiirus=10; //seame M iga viidatud, ehk M1, kiiruse 10 ks
//Viida kasutamisel on ka C teine konstruktsioon
M->kiirus=10; //Samaväärne eelmise käsuga.
/*Uute tüüpide kasutamine võimaldab ka deklareerida lihtasamaid
funktsioone, N: kasutame viitu funktsiooni päises*/
void liiguta(typ *mot)
{
      //Funktisoon saab kasutada ja muuta
      //mot->kiirus ja mot->suund
}
//funktsiooni kasutamine
liiguta(&M1);
```

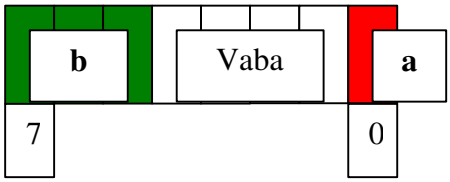
struct tüüp võib sisaldada omakorda viitu

```
typedef int (*f_viit)(int, int); //Defineerime funtsiooni viida tüübi
//Defineerime struktuuri, mis sisaldab viitu
typedef struct {
   char a;  //Harilik muutuja
   char *b; //Viit
   f_viit c; //Funktsiooni viit
 } Tyyp;
//Deklareerimine .. nagu struct ikka
```

```
Tyyp M;  //Tavaline
Tyyp *Viit; //Viit
//Kasutamine
M.b=&M.a; //b viitab a le
*M.b=5;   //b viitab a le→ a väärtus saab 5 ks
M.c=&liida; //Vaata ülesse poole, funktsiooni viitade juurde
M.c(3,4); //Tagastab 3+4=7
//Viidaga
Viit=&M;  //Viit hakab viitama M ile
Viit->a;  //on 5
Viit->b;  //on a address, kuna b viitab a le
*Viit->b; //Kuna b viitab a le ja a on 5 → selle avaldise väärtus on 5
Viit->c(5,6); //Selle avaldise väärtus on 11
```

struct tüüp ja bitiväljad (bit fields)

```
typedef struct
{ char a:1; //1 esimene bit, madalaim
  char :4;   //4 biti jääb vahele
  char b:3;  //3 bitti, kõrgemad
} Tyyp;
//Mis see on???
/*Tegelikult kogu selle struktuuri pikkus on 1 bait, mis on omakorda
ära jagatud muutujate a, b vahel*/
Kusjuures need paiknevad selles baidis nii...
```



Preprotsessori käsud

Preprotsessor töötleb lähte faile enne, kui neid päriselt kompileerima hakatakse. Preprotssori käsustik on enamasti küllaldki mahukas, nii et sellest saaks eraldi pika juhendi kirjutada. Kuna väiksemate projektide juures (nagu see roboti ehitus) pole see nii oluline siis käsitleme vaid neid preprotsessori käske, mis tõesti võivad vajalikeks osutuda C preprotsessori käsud algavad # märgiga

#include

Selle käsuga saab laadida teisi C **(.h, .c)** faile. See on kasulik, et hoida ühte faili väiksemana, siis on see ka paremini jälgitav. Samuti võib funktsioonid vastavalt tüübile jagada erinevatesse failidesse (soovitav teha ka korduv kasutamise eesmärgil). Funktsioonide kasutamiseks projektis tarvitseb siis vaid kasutada #include käsku ja see fail oma failile külge laadida. Samuti laaditakse juba eelkompileeritud library'te (eelkomileerimine on kasulik, kuna aitab suurendada kompileerimise kiirust, pisikeste mikrokontroller projektide korral pole see eriti oluline) päisfaile **(.h)** lõpuga

```
//Kui laaditakse faile, mis asuvad include failide pathis siis on < >
//Tüüpiliselt on see \include\ kataloog
#include <stdlib.h>
//Kui laaditakse faile, mis asuvad antud failiga samas kaustas siis “”
#include “minu_lisa_funktsioonid.c”
```

Näide:
Fail: minu_lisa_funktsioonid.c

```
//Minu funktsioonid
int lisa_f(int a)
{     return 2*a+2;     }
```

Fail: main.c

```
//Minu pohi fail – see mida kompileeritakse
#include "minu_lisa_funktsioonid.c"
//Võtsime sellest teisest failist, funktsioonid oma projekti
void main(void)
{
      int a;
      //Kasutame funktsiooni
      a=lisa_f(5);
      for(;;);
}
```

#define

Võimaldab defineerida makrosid. Mis need on? Saab defineerida koodi osasid, mis enne kompileerimist ära asendatakse. Seda on kasulik teha näiteks koodi selguse huvides, samuti selle huvides, kood oleks hiljem hõlpsamini muudetav.

Näide:

```
#define MASSIIVI_SUURUS 10
char massiiv[MASSIIVI_SUURUS];
for(i=0; i<MASSIIVI_SUURUS; i++) massiiv[i]=0;
```

Preprotsessor asendab enne kompileerimist MASSIIVI_SUURUS e koodis 10 ga. Kui aga üritada koodi muuta, siis piisab, kui seda teha ühes kohas, see teeb muutmise märksa kiiremaks ja vähem vea ohtlikuks.

Define makro võib sisaldada ka parameetreid

```
#define TINGIMUS(x) (x<10)
//Makro parameeter on mingi koodi lõik, mis siis asendatakse

for(a=0; TINGIMUS(a+2); a++) /*midagi*/ ;

if(TINGIMUS(b)&&(c==0))
{
      //Midagi
}
//Pärast preprotsessori tehtud asendusi oleks kood sisuliselt selline
for(a=0; (a+2<10); a++)

if((b<10)&&(c==0))
{
}
```

#pragma

Antakse kompilaatorile ette parameetreid, mis määrab kuidas midagi tuleb kompileerda (tõlgendada)

Näiteks:

```
#pragma bitfields=default
```

ütleb kompilaatorile, et sellele järnevas koodis tuleb bit fielde (biti välju) luua nii, et esimesed muutujad paiknevad madalamatel bitidel (täitmist alustatakse madalamatest bitidest)

```
#pragma bitfields=reversed
```

aga seda, et baidi täitmist alustatakse vastupidi, ehk siis kõrgematest bitidest
Võimalike parameetreid on tegelikult palju

Teegid

IAR kompilaatoriga tuleb kaasa ka hulganisti library'si ehk juba defineeritud funktsioone, makrosid, andmetüüpe, mida saab kohe kasutada.
Märkide töötlus *(CHARACTER HANDLING)* **ctype.h**
Madala taseme käsud *(LOW LEVEL ROUTINES)* **icclbutl.h**
Matemaatika (cos, sin, log jne.) *(MATHEMATICS)* **math.h**
Sisend/väljund *(IO)* **stdio.h**
Üldised vahendid (teisendused, mälu haldamine) *(GENERAL UTILITIES)* **stdlib.h**
Sõne (tekst) töötlus *(STRING HANLING)* **string.h**
Jt… (vaata lisasid)