



C-ohjelmointi, viikko2 materiaalia

- q Tekstitiedostot ja niiden käyttö
- q Esikääntäjä, makrot ja ehdollinen kääntäminen



Tekstiedostot ja niiden käyttö

(Möldnerin kirjassa luku 5)

- n Tekstiedosto ó binääritiedosto
 - n Tiedostokahva
- n Tiedoston avaaminen ja sulkeminen
 - n Tiedoston käyttötavat
 - n Virhetilanteet avauksessa ja sulkemisessa
- n Standarditiedostot
- n Perus I/O-operaatiot tiedostoille
 - n Lukeminen ja kirjoittaminen
 - n Esimerkkejä ja idiomeja



Tekstitiedostot

- n Tiedostot ovat pelkkiä tavujonoja. Tiedoston päättää tiedoston lopetusmerkki **EOF**.
- n Kahdenlaisia tiedostoja; ero niiden käsittelyssä
 - n Tekstitiedostot käsitellään riveittäin. Joka rivin lopussa on rivin loppumismerkki ('uuden rivin alkamismerkki') **\n**.
 - n Binääritiedostossa ei ole mitään erityistavuja.
- n *Eri käyttöjärjestelmät käyttävät eri tapoja ilmaisemaan rivin tai tiedoston päättymistä!*



Tiedostokahva (filehandle)

- n Tiedostokahva on tiedostoon osoittava osoitin (pointer), jonka avulla tiedostoa käsitellään.
- n Tiedostokahvan määrittely:

```
FILE *kahva;  
FILE *tied1, *tied2;
```

```
typedef FILE* P_FILE;  
P_FILE tied1, tied2;
```

```
/*määritellään ensin  
tiedostokahvatyyppi */
```



Tiedoston avaaminen

- n Ennen käyttöä tiedosto on avattava käyttäen funktiota `fopen()`.
- n `Avaus yhdistää tiedoston tiedostokahvaan.`
- n Avattaessa ilmoitetaan `tiedoston nimi` sekä tiedoston `käyttötapa`.

```
kahva = fopen("testitiedosto", "r");  
tied1 = fopen("MyFile.txt", "w");  
tied2 = fopen("test.out", "wb");
```



Tiedoston käyttötavat

- **"r"** lukeminen jo olemassa olevasta tiedostosta
- **"w"** kirjoittaminen: olemassa olevan tiedoston päälle (overwrite) tai uuteen tiedostoon, joka luodaan vasta tässä (create)
- **"a"** kirjoittaminen joko olemassa olevan tiedoston perään (append) tai uuteen luotavaan tiedostoon
- **"r+"** lukeminen ja kirjoittaminen, muuten kuten **"r"**
- **"w+"** lukeminen ja kirjoittaminen, muuten kuten **"w"**
- **"a+"** lukeminen ja kirjoittaminen, muuten kuten **"a"**

- n Jos halutaan käsitellä tiedostoa binäärimuotoisesti, niin lisätään ylläoleviin merkki b: **"r+b"**.
- n Jos tiedosto on avattu sekä lukemista että kirjoittamista varten, I/O-operaatioiden välillä on aina kutsuttava jotain funktioista: `fseek()`, `fsetpos()`, `rewind()` tai `fflush()`.



Tiedoston avaaminen voi epäonnistua!

- n Jos tiedoston avaaminen ei onnistu, niin fopen palauttaa arvon **NULL** (= erityinen nolla-arvo), muuten osoittimen tiedostoon eli tiedostokahvan.
- n **Joka kerta** tiedostoa avattaessa on siis varmistuttava, että avaus onnistui!

Idioms

```
if (fileHandle = fopen(fname, fmode)) == NULL)
    /* toiminta virhetilanteessa */
```



Tiedostonimet, avattujen tiedostojen maksimimäärä

- n Tiedostonimi voi myös olla polkunimi.
 - n Eri käyttöjärjestelmissä on erilaisia polkunimiä. Näissä voi olla merkkejä, joilla on erityismerkitys C-kielessä. (Esim. DOS-järjestelmän \-merkki)
 - n \ on escape-merkki, joka muuttaa seuraavan merkin merkityksen => DOS:n \-merkki korvattava \\-merkeillä
 - n FILENAME_MAX (stdio.h) kertoo tiedostonimen maksimipituuden.
- n Avattujen tiedostojen määrä järjestelmässä on rajoitettu: enintään FOPEN_MAX (stdio.h) tiedostoa.



Tiedoston sulkeminen

- n Avattu tiedosto täytyy sulkea, kun sitä ei enää tarvita.
- n Tiedosto suljetaan käyttäen funktiota `fclose()`.
 - n Parametrina annetaan suljettavan tiedoston kahva.
 - n Jos sulkeminen epäonnistuu, niin `fclose` palauttaa EOF-merkin. **Aina varmistettava onnistuminen!**

\idioms

```
if (fclose(tied1) == EOF)
```

```
/* toiminta, kun tiedostoa ei saada suljettua */
```



Standarditiedostot: stdin, stdout ja stderr

- n Aina käytävissä olevat ennaltamääritellyt, tiedostoonsa 'kiinnikolvatut' tiedostokahvat
 - n Ei tarvitse erikseen avata eikä sulkea!

```
stdin    standardisyöttövirta; yleensä näppäimistö
stdout   standarditulostusvirta; yleensä näytölle
stderr   standardivirhetulostusvirta; eri kuin normaalitulostus
```

```
FILE *inkahva;
.....
inkahva = stdin;
```

/* inkahva on synonyymi stdin-kahvalle */



I/O-operaatiot tiedostoille

(perusoperaatiot)

- n Perusoperaatiot ovat hyvin samankaltaisia kuin pääte-I/O:ssa tai formatoidussa I/O:ssa käytetyt:

<code>int fgetc (fileHandle)</code>	~ <code>int getchar()</code>
<code>int fputc (int, fileHandle)</code>	~ <code>int putchar(int)</code>
<code>int fscanf (fileHandle,)</code>	~ <code>int scanf(...)</code>
<code>int fprintf (fileHandle,)</code>	~ <code>int printf(...)</code>

Merkin ja luvun lukeminen tiedostosta

- n Yhden merkin lukeminen tiedostosta

```
if ((c=fgetc(fileHandle)) == EOF)
/* toiminta tiedoston loppuessa */
```

- n Yhden luvun lukeminen tiedostosta

```
if (fscanf (fileHandle, "%d", &i) != 1)
/* toiminta lukemisen epäonnistuessa */
```



Esimerkki: Ohjelma lukee kolme lukua tiedostosta *luvut* ja tulostaa niiden summan näytölle.

```
int main() {  
    FILE *f;  
    double x, y, z;
```

Tiedoston avaamisfraasi!

```
    if((f = fopen("luvut", "r")) == NULL)  
    {  
        fprintf(stderr, "Ei avaudu: %s\n",  
            "luvut");  
        return EXIT_FAILURE;  
    }
```

```
if(fscanf(f, "%lf%lf%lf", &x, &y, &z) != 3) {  
    fprintf(stderr, "Tiedoston lukeminen \  
        epäonnistui\n");  
    return EXIT_FAILURE;  
}
```

Tiedoston
lukemisfraasi

```
printf("%f\n", x + y + z);
```

```
if(fclose(f) == EOF) {  
    fprintf(stderr, "Sulkeminen epäonnistui\n");  
    return EXIT_FAILURE;  
}
```

```
return EXIT_SUCCESS;
```

```
}
```

Tiedoston sulkemisfraasi



Rivin loppu ja tiedoston loppu

```
/* Lukee tiedostosta yhden rivin ja tulostaa sen  
näytölle */
```

```
while((c = fgetc(tkahva)) != '\n')  
    if (c == EOF) break;  
    else putchar(c);  
if(c != EOF) putchar(c);
```

```
/* Etsitään vain rivin loppu */
```

```
while((c = fgetc(tkahva)) != '\n');
```

```
/* Lasketaan rivin merkkien määrä */
```

```
while((c = fgetc(tkahva)) != '\n') ccount++;
```



Laadi ohjelma, joka laskee ja tulostaa tiedoston testi.txt rivien lukumäärän.

- q Määrittelyt
- q Avaa tiedosto testi.txt
- q tiedoston rivien lukumäärän laskeminen
 - q Niin kauan kuin tiedostoa riittää
(eli ei vielä EOF-merkki)
 - q Jos '\n' -merkki, niin kasvata rivien lukumäärää.
 - q Tulosta lukumäärä näytölle
- q Sulje tiedosto testi.txt



ungetc, feof

n ungetc (char, fileHandle);

- n 'Palauttaa luetun merkin tiedostoon' eli laittaa sen takaisin tiedoston lukupuskuriin, josta se luettavissa uudestaan. Itse tiedostoa ei muuteta.

- n while (ehto (c = fgetc (tiedostokahva)))
 prosessoi c;
 ungetc (c, tiedostokahva);

n feof(fileHandle);

- n Testaa tiedoston loppumista. Palauttaa 0 , jos tiedosto on loppu, muuten jonkun muun arvon.

Idioms

Tiedoston avaaminen:

```
if ((fileHandle = fopen(fname, fmode)) == NULL) ... /* failed */
```

```
if (close(fileHandle) == EOF) .... /* failed */
```

Tiedoston sulkeminen

```
if ((c = fgetc(fileHandle)) == EOF) ... /* error */
```

Yhden merkin lukeminen

```
if (fscanf (fileHandle, "%d", &i) !=1) ... /* error */
```

Yhden luvun lukeminen

```
while ((c= fgetc(fileHandle)) != '\n')
```

Rivin loppuun lukeminen

```
while ((c= fgetc(fileHandle)) != EOF)
```

Tiedoston loppuun lukeminen

Fraaseja tiedoston käsittelyyn



Esikäöntäjä, makrot ja ehdollinen kääntäminen

(Mülderin kirjan luku 6)

- n C-esikäöntäjä
- n Makrot
 - n Parametrittomat makrot
 - n Parametrilliset makrot
 - n Ennaltamääritellyt makrot
 - n Ulkoisten tiedostojen sisällyttämisen
- n Ehdollinen kääntäminen
 - n Eri tavat toteuttaa
 - n Käyttö virheenjäljityksessä
 - n Assert-makro ja sen käyttö
 - n Käyttö otsaketiedostoissa
 - n Käyttö siirrettävyyden lisäämiseksi



C-esikäntäjä

- n #-merkillä alkavat komentorivit
 - n tarkoitettu esikäntäjälle => oma syntaksi
 - n käsitellään ennen kääntämistä
- n Mihin käytetään?
 - n **Makrot**: korvataan teksti toisella tekstillä
 - n **Ulkoisten tiedostojen liittäminen**
 - n `#include <stdio.h>`
 - n **Ehdollinen kääntäminen**: vain osa lähdetiedostoa käännetään tietyn ehdon ollessa voimassa; hyötyä virheiden etsinnässä



Makrotyypit

n Parametrittomat makrot

- n Lyhennemerkinä; makronimi korvataan aina samalla tekstillä

n Parametrilliset makrot

- n Parametrit vaikuttavat korvaustekstiin => monipuolisempi, mutta helposti myös yllättäviä sivuvaikutuksia

n Ennaltamääritellyt makrot

- n C-toteutuksessa jo sisällä
- n Hyötyä virhetilanteessa

Parametriton makro

§ **#define makroNimi makroArvo**

```
#define PI 3.14
#define PII 3.14159265358979323\
846264338327950
#define SCREEN_W 80
#define SCREEN_H 25
```

n **MakroNimi ISOILLA KIRJAIMILLA!**

n **MakroArvo rivin loppuun ('\n'-merkkiin asti)**

n **'\ ' = jatkuu seuraavalle riville**

n **Huomaa: ei -=merkkiä eikä puolipistettä(;)!**



Makronimen korvaus makroarvolla

- n Esikäyttäjä korvaa lähdetiedostossa jokaisen makroNimi esiintymän makroArvon tekstillä.

```
#define PI 3.14
```

```
i=PI;
```

```
korvaus => i=3.14;
```

```
#define PII =3.14;
```

```
i=PII;
```

```
korvaus => i = =3.14.;
```

VÄÄRIN!!!!

Esimerkkejä makron käytöstä

```
#define PROMPT printf("Enter real value: ")
#define SKIP while(getchar() != '\n');
```

Käytä runsaasti sulkuja, etenkin aritmeettisissa lausekkeissa!

```
#define A 2 + 4
#define B A * 3
```

} => B = 2 + 4 * 3 = 14

Haluttiinko todella tätä?

```
#define A (2 + 4)
#define B (A * 3)
```

} => B = ((2+4) * 3) = 18



Parametrilliset makrot

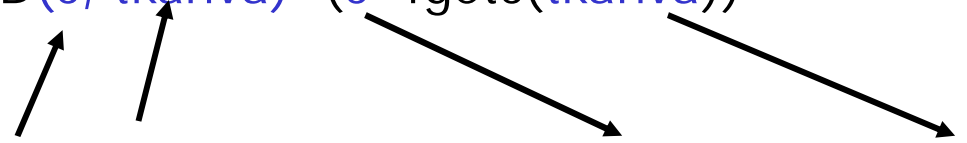
#define makroNimi(parametrit) makroArvo

esim.

```
#define READ(c, tkahva) (c=fgetc(tkahva))
```

....

```
if (READ(ch, tied1) == 'x') => if ( (ch = fgetc(tied1)) == 'x')
```



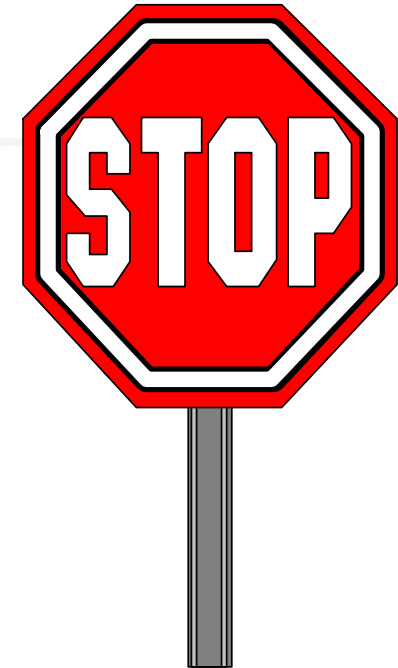
<eri asia kuin> if (ch = fgetc(tied1) == 'x')

=>>>

Siis käytä runsaasti sulkuja!

Varovaisuutta, huolellisuutta makrojen käytössä !

- n Makroja käytettäessä syntyy helposti sivuvaikutuksia
- n Makro voi helpottaa kirjoittamista, mutta koodin luettavuus voi kärsiä!



```
#define SQR(x) (x*x)  
SQR(z+1);  
=> (z+1*z+1)
```

```
#define SQR(x) ((x)*(x))  
SQR(z+1);  
=> ((z+1)*(z+1))
```



#-merkin muu käyttö

- n Kommenteissa ja merkkijonovakioissa olevia makronimiä ja parametreja ei hyväksytä!
 - n määrittelyssä parametrin eteen pitää laittaa #-merkki, jolloin sen arvo makron laajennuksessa laitetaan lainausmerkkien sisään
 - n `#parametri => "parametri"`
- n Merkkien yhteenliittäminen
 - n `#define TEMP(i) temp ##i`
 - n `TEMP(1) = TEMP(2) => temp1 = temp2`

```
#define EMPTY (maxUsed == 0)
#define ASSERT if (!(EMPTY ? current == 0 : \
    0 < current && current <=maxUsed)) {\
    fprintf(stderr, "invariant failed; current = %d \t; \
maxUsed= %d\n", current, maxUsed); \
    exit(1); }
```

EMPTY ? current == 0 : 0 < current && current <=maxUsed

ehdollinen lauseke



Ennaltamääritellyt makrot

`n` 4 kappaletta

`n` `__LINE__` lähdekoodin tämän rivin numero

`n` `__FILE__` tämän lähdekoodin nimi

`n` `__TIME__` käännoisaika

`n` `__STDC__` 1, jos kääntäjä noudattaa ANSI C:tä

```
if (n>10) { /* virhetilanne */
    fprintf (stderr, " liian suuri n:n arvo tiedoston %s
                rivillä %d ! \n", __FILE__, __LINE__);
    return EXIT_FAILURE;
}
```



Makromäärittelyn purkaminen

- n #undef PI
- n jos halutaan määritellä uudestaan PI, niin entinen määrittely on purettava
 - n Muuten voi tulla ongelmia!
 - n Makromäärittelyn voi purkaa myös kääntäjän komentorivillä

Ulkoisen tiedoston lisääminen lähdekoodiin

- n kaksi eri muotoa: miten lisättävä tiedostoa haetaan hakemistosta

\dioms

```
#include "filename" /* käyttäjän oma tiedosto, haetaan ensin nykyhakemistosta */
```

```
#include <filename> /* järjestelmän tiedosto, haetaan ensin systeemihakemistosta */
```

- n kootaan yleensä otsaketiedostoiksi **.h**



Standard Header Files

- stdio.h** - the basic declarations needed to perform I/O
- ctype.h** - for testing the state of characters
- math.h** - mathematical functions, such as `abs()` and `sin()`
(linking `-lm`)

Ehdollinen kääntäminen

(Conditional Compilation)

- n = tietyssä tilanteessa, tietyn ehdon ollessa tosi osa koodia jätetään kääntämättä
- n Käytetään
 - n virheenjäljityksessä, otsaketiedostoissa
 - n siirrettävää koodia tuottaessa
- n Kaksi eri tapaa

```
#ifdef macroname  
part1  
#else  
part2  
#endif
```

```
#ifndef macroName  
part1  
#else  
part2  
#endif
```

```
#if constantExpression1  
part 1  
#elif constantExpression2  
part2  
#else  
part3  
#endif
```

```
#if constantExpression1
```

```
    part 1
```

```
#elif constantExpression2
```

```
    part2
```

```
#else
```

```
    part3
```

```
#endif
```

- voi olla useita #elif –
osia

- #else voi puuttua

```
#if defined (name) ...
```

```
/*onko name määritelty?*/
```

```
#error textMessage
```

```
#if defined (__STDC__)
```

```
.....
```

```
#else
```

```
#error "Jotain pielessä"
```

```
#endif
```



Käyttö virheenjäljityksessä

- n Ehdollinen kääntäminen on 'poiskomentointia' parempi tapa poistaa kulloinkin turhat koodin osat.
 - n esim. virheenjäljitystä varten lisätyt tulostuskomennot C:ssä ei saa olla sisäkkäisiä kommentteja!

```
# if 0  
    poisjätettävä osa  
#endif
```

Näin sama lähdekoodi voi toimia sekä testiversiona että tuotantoversiona!

Testikoodin poisjättäminen

```
#define DEB /* vain määritelty */  
#ifdef DEB /*jokin virheenjäljitys lause  
          esim. tulostuslause */  
    printf("value of i = %d", i);  
#endif  
  
/* tuotantokoodia */
```

Kääntäjän
komento-
rivillä!

```
gcc -UDEB prog.c makromäärittely pois päältä  
gcc -DDEB prog.c makromäärittely päälle
```

Esimerkkejä:

```
int main() {  
    int i, j;  
    printf("Enter two integer values: ");  
    if(scanf("%d%d", &i, &j) != 2)  
        return EXIT_FAILURE;  
#ifdef DEB  
    printf("entered %d and %d\n", i, j);  
#endif  
    printf("sum = %d\n", i + j);  
    return EXIT_SUCCESS;  
}
```

Mitä hyötyä!

```

int i, j;
#ifdef DEB
    int res;
#endif
if(
#ifdef DEB
    (res =
#endif
    scanf("%d%d", &i, &j)
#ifdef DEB
    )
#endif
    ) != 2 )

```

```

#ifdef DEB
{
    switch(res) {
        case 0: printf("both values were
                    wrong\n");
                break;
        case 1: printf("OK first value
                    %d\n", i);
                break;
        case EOF: printf("EOF\n");
                 break;
        case 2: printf("both OK\n");
                break
    }
}
#endif
...

```

Enemmän informaatiota!

```
int main() {
    const char SENTINEL = '.';
    int aux, maxi=0;
#ifdef DEBUG
    printf(" Virheenjäljitys päällä: kopioidaan kaikki merkit\n");
#endif
    while(1) {
        if ((aux = getchar()) == EOF || aux == SENTINEL) break;
#ifdef DEBUG
        putchar(aux);
        putchar('\n');
#endif
        if (aux > maxi)
        {
#ifdef DEBUG
            printf("Suurin merkki on nyt: %c\n", aux);
#endif
            maxi = aux;
        }
    }
#ifdef DEBUG
    putchar('\n');
#endif
    printf("Suurin merkki on: %d\n", maxi);
    return EXIT_SUCCESS;
}
```

Assert-makro virheenjäljityksessä (1)

assert (int lauseke) (*assert.h*)

Diagnostiikkatietojen kirjoittaminen standardivirhetiedostoon (stderr)

- Jos lauseke on epätosi (false, 0), niin virhetiedostoon kirjoitetaan lauseke, lähdekooditiedoston nimi ja rivin numero:

Assertion failed: ehto, file tiednimi, line rivinro

(Tiedostonimi ja rivinumero saadaan makroista `__FILE__` ja `__LINE__`.)
ja ohjelman suoritus keskeytetään `abort()`-funktiolla.

- assert-makroilla varmistetaan, että ohjelma toimii kuten sen loogisesti oletetaan toimivan: ennakkoehtoja, jälkiehtoja, oletuksia muuttujien arvoista.
- Esimerkkejä lausekkeista:

```
assert (i >= 0)
assert (b*b - 4*a*c >= 0)
assert (0 <= i && i < size)
```



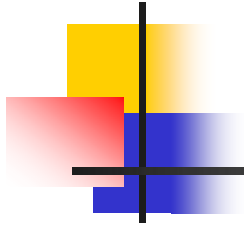

Assert-makro virheenjäljityksessä (2)

- n Assert-makron toimintaa säätelee makron **NDEBUG** (= *no debug*) määrittely. Jos NDEBUG on määritelty, niin assert ei tee mitään.
- n *Oletusarvoisesti* **assert()** on käytössä ja valvoo ohjelman toimintaa. Tarkistukset poistetaan ohjelmasta määrittelemällä makro **NDEBUG** joko makromäärittelyssä **#define NDEBUG** tai kääntäjän parametrina **gcc -DNDEBUG**.



Esimerkki assert-koodista

```
/* Assert.cc for GNU C/C++ */
/* #ifdef ASSERT ○ # ifndef NDEBUG */
#ifdef ASSERT
void AssertionFailure(char *exp, char *file, char *baseFile, int line)
{
    if (!strcmp(file, baseFile)) {
        fprintf(stderr,
            "Assert(%s) failed in file %s, line %d\n", exp, file, line);
    } else {
        fprintf(stderr,
            "Assert(%s) failed in file %s (included from %s), line %d\n",
            exp, file, baseFile, line);
    }
}
#endif
```



```
#include <assert.h>
void open_record(char *record_name) {
    assert(record_name != NULL);
    /* Rest of code */
}

int main(void) {
    open_record(NULL);
}
```



Makro otsaketiedostojen suojana

- n Ehdollisella kääntämisellä varmistetaan, että otsaketiedosto käännetään vain kerran: kukin otsaketiedosto ympäröidään makrolla, joka suoritetaan vain kerran.
 - n Useasta osasta koottavaan ohjelmaan tulee helposti sama otsaketiedosto useaan kertaan => käännösvirhe
- n Makro nimetään otsaketiedoston mukaan:
screen.h => käytetään makronimeä SCREEN_H

Idioms

```
#ifndef SCREEN_H  
#define SCREEN_H  
/*otsaketiedoston sisältö */  
#endif
```

```
#include "screen.h"  
.....  
#include "screen.h"
```

Käännetään vain kerran!



Ehdollinen kääntäminen ja siirrettävyys

- n Erilaisissa ympäristöissä toimivien ohjelmien kehittämiseen:

```
#if IBMPC /* ehtolauseke */  
#include <ibm.h>  
#else  
#include <generic.h>  
#endif
```

```
#ifdef IBMPC /*makromäärittely*/  
typedef int MyInteger  
#else  
typedef long MyInteger  
#endif
```

Mitä seuraava ohjelma tulostaa?

```
#define LOW    -2
#define HIGH  (LOW+5)
#define PR(arg)  printf("%d\n", (arg))
#define FOR(arg) for(; (arg); (arg)--)
#define SHOW(x) x
int main(){
int i = LOW;
int j = HIGH;
    FOR(j)
        switch(j) {
            case 1: PR(i++);
            case 2: PR(j);
                    break;
            default: PR(i);
        }
    printf ("\n%s\n", SHOW(3));
    return EXIT_SUCCESS;
}
```

Mitä seuraava ohjelma tulostaa?

```
#define LOW    -2
#define HIGH  (LOW+5)
#define PR(arg)  printf("%d\n", (arg))
#define FOR(arg) for(; (arg); (arg)--)
#define SHOW(x) x

int main(){
int i = LOW;    int i = -2;
int j = HIGH;  int j = (-2 + 5); /* = 3*/
  FOR(j)      for (; (j); (j)--)
    switch(j) {
      case 1: PR(i++);  printf("%d\n", (i++));
      case 2: PR(j);    printf("%d\n", (j));
                break;
      default: PR(i);   printf("%d\n", (i));
    }
  printf ("\n%s\n", SHOW(3));
  return EXIT_SUCCESS;
}
```