Sisältö

1. Johdanto

1.1 Yleistä

- Suurten ohjelmien organisointi vaikeaa C:ssä.
- 1980 Stroustrup lisäsi C:hen olio-ohjelmoinnin (OOP = Object Oriented Programming).
- C⊂C++: C-ohjelma on (muutamia poikkeuksia lukuunottamatta) myös C++-ohjelma.

1.2 OOP

Olio (objekti, object)

- Olio on kokonaisuus, joka sisältää sekä datan että sitä käsittelevän koodin.
- Olion data ja koodi voi olla pelkästään olion itsensä käytettävissä; olion yksityiset (*private*) osat ovat suojattavissa.
- Koodi + data kokonaisuus = kapselointi (*encapsulation*).
- Ohjelmoijan kannalta olio näyttää (tavalliselta) muuttujalta.

Monimuotoisuus (polymorphism)
Funktion (operaation, toimituksen) merkitys riippuu sovellettavan datan tyypistä; esim +:

`1 + 1`	kokonaislukujen yhteenlasku
`1.2 + 1.0`	liukulukujen yhteenlasku

Perimä (inheritance)
Olio voi periä toisen olion ominaisuuksia.
Esim. *puikula* on *peruna*, joka kuuluu luokkaan *ruoka*.

1.3 Lyhyt katsaus C++:aan

Esim.

```
#include <iostream>
main()
{
  std::cout << "Heippa\n";
}
```

- `#include <iostream>` liittää koodiin tiedoston `iostream` sisällön.
- Tiedostossa `iostream` esitellään olio `cout` (standardi output) ja operaattori `<<`.
- Tunnukset määritellään jossakin nimiavaruudessa (*namespace*). Standardikirjasto, josta `iostream` on osa, määrittelee tunnuksensa nimiavaruudessa `std`. Operaattorilla `::` valitaan nimiavaruus: `std::cout` viittaa nimiavaruudessa `std` määriteltyyn olioon `cout`.
- Vakiomerkkijonot "-merkkien välissä.
- \+jokin merkki tarkoittaa erikoismerkkiä; `\n` = uusi rivi.
- Käskyt päätetään `;`-merkkiin.
- `main(){...}` esittelee funktion nimeltä `main`.
- Jokaisessa ohjelmassa täytyy olla `main`-niminen funktio, josta aloitetaan ohjelman suoritus.
- C++ erottaa pienet ja isot kirjaimet:
  `cout`≠`Cout`≠`couT`
- Kirjoitusasu on vapaa.

Huom. Nimiavaruuksien kvalifiointi voidaan välttää spesifioimalla käytettävät nimiavaruudet `using`-direktiiveillä. Esim.

```
#include <iostream>
using namespace std;
main()
{
  cout << "Heippa\n";
}
```

Vaikka tämän tapainen menettely ei yleensä ole suositeltavaa, oletamme että jatkon koodiesimerkeissä on useimmiten voimassa `using`-direktiivejä kirjoittamatta niitä eksplisiittisesti näkyviin.
Huom. Tällä hetkellä tiettävästi yksikään kääntäjä ei vielä aivan täysin toteuta 1997 hyväksyttyä ISO/ANSI-standardia. Esim. useimpien kääntäjien `iostream`-kirjaston tunnukset eivät kuulu nimiavaruuteen `std`. Sen sijaan ne on määritelty kaikkien nimiavaruuksien ulkopuolella, ns. globaalissa nimiavaruudessa. Globaalin nimiavaruuden tunnuksiin voidaan viitata sellaisenaan, esim. `cout << "Heippa\n";`.
Monista esittelytiedostoista on olemassa kaksi versiota:

- kun tiedoston nimessä on tarkenne `.h`, esim. `iostream.h`, tunnukset on määritelty globaalissa nimiavaruudessa.
- kun tiedoston nimessä ei ole tarkennetta, esim. `iostream`, tunnukset on määritelty jossakin nimiavaruudessa.

Esim.

```
// Muunnos tuumista cm:ksi
#include <iostream.h>
main()
{
  int inch = 0;
  cout << "Tuumia = ";
  cin >> inch;
  cout << inch;
  cout << " tuumassa on ";
  cout << inch*2.54;
  cout << "cm\n";
}
```

- Jokainen muuttuja on määriteltävä (tai esiteltävä).
- `int inch` määrittelee kokonaislukumuuttujan `inch`.

- Muuttujat voidaan alustaa määriteltäessä.
- Operaattori `>>` (määritelty `iostream.h`:ssa) lukee oliosta `cin` (standardi input) luvun muuttujaan `inch`.
- `<<` on vasemmalle assosiatiivinen; voidaan kirjoittaa myös

```
cout << inch << " tuumassa on "
  << inch*2.54 << "cm\n";
```

- `//`-merkkien jälkeinen loppurivi tulkitaan kommentiksi.
- Kommentti voi olla myös muotoa `/*...*/` ja voi esiintyä kaikkialla, missä välilyönti on sallittu.

Perustyypit

`bool`	Boolen muuttuja
`char`	kokonaisluku
`short`	kokonaisluku
`int`	kokonaisluku
`long`	kokonaisluku
`float`	liukuluku
`double`	liukuluku

Vakiot

Boolen muuttuja	`true`, `false`
Kokonaisluku	`1`, `-5`,...
Liukuluku	`1.2`, `1.4e-7`,...
Merkit	`'a'`, `'A'`, `'\n'`,...

Aritmeettiset operaattorit

`+`	yhteenlasku
`-`	vähennyslasku
`*`	kertolasku
`/`	jakolasku
`%`	jakojäännös

Vertailuoperaattorit

`==`	yhtäsuuruus
`!=`	erisuuruus
`<`	pienempi
`>`	suurempi
`<=`	pienempi tai yhtäsuuri
`>=`	suurempi tai yhtäsuuri

Johdetut tyypit

Operaattorit

`*`	osoitin
`&`	viittaus (C++)
`[]`	taulukko
`()`	funktio

luovat uusia tyyppejä.

Esim.

```
char v[10];     // 10 merkin taulukko
```

Huom. Taulukon indeksit alkavat 0:sta, ts. `v`:n alkiot ovat `v[0]`,`v[1]`,...,`v[9]`.

Lohko

- Merkkien `{...}` väliin suljettu käskylista.
- Lohkossa määritellyt muuttujat tunnetaan vain lohkon sisällä.
- Lohkoa voidaan käsitellä yhtenä käskynä.

If-rakenne

Esim.

```
int i, j;
char c;
cin >> i >> c >> j;
if( c == '+' )
   cout << i + j << "\n";
else if( c == '-' )
   cout << i - j << "\n";
else {
   cerr << "En tiedä, mitä tehdä!\n";
   exit( 1 );
}
```

Switch-rakenne

Esim.

```
switch( c ) {
    case '+':
        cout << i + j << "\n";
        break;
    case '-':
        cout << i - j << "\n";
        break;
    default:
        cerr << "en tiedä, mitä tehdä!\n";
        exit( 1 );
    }
```

- `break` siirtää ohjelman suorituksen ulos `switch`-rakenteesta.
- `default`-osa ei ole pakollinen.

While-rakenne

Esim.

```
char p[10], q[10];
    ...
int i = 0;
while( i < 10 ) {
     q[i] = p[i];
     i++;
}
```

- `++`-operaattori kasvattaa operandinsa arvoa yhdellä.
- `--`-operaattori pienentää operandinsa arvoa yhdellä.
- Muuttujia ei ole pakko määritellä funktion alussa (C++).

For-rakenne

Muoto:
`for( alustus; ehto; lopputoimet ) käsky;`

Esim.
```
for( int i = 0; i < 10; i++ ) q[i] = p[i];
```

Huom. Muuttuja voidaan määritellä myös for-silmukan alustuksessa (C++).

Funktiot

Oletetaan, että jossakin tiedostossa on koodi
```
float pow( float x, int n )
{
   if( n < 0 ) {
      cerr << "Negatiivinen eksponentti!\n";
      exit( 1 );
   }

   switch( n ) {
      case 0:  return 1;
      case 1:  return x;
      default: return x*pow( x, n - 1 );
   }
}
```

- `float pow(...)` ilmoittaa, että `pow` on funktio, jonka arvona on `float`-tyyppinen luku.
- Funktion argumentit luetellaan tyyppeineen argumenttilistassa: `( float x, int n )`.
- `cerr` on standardivirhetulostus.
- `exit()` on standardikirjastossa määritelty funktio; päättää ohjelman suorituksen.
- `return`-käskyllä palautetaan funktion arvo.
- Funktio voi kutsua itseään (*rekursio*).
- Argumentit välitetään ns. *call by value* -mekanismilla; funktio saa vain argumenttien arvot, ei todellisia kutsussa esiintyviä argumentteja.
- Argumenttien muutokset eivät välity kutsuvaan ohjelmaan.
- *Call by reference*-mekanismissa välitetään kutsuttavalle funktiolle viittaukset (so. osoitteet) argumentteihin (esim. Fortran).

Jossakin toisessa tiedostossa voi olla koodi
```
extern float pow( float, int ); // Prototyyppi
main()
{
   for( int i = 0; i < 10; i++ )
     cout << pow( 2, i ) << "\n";
}
```

- Prototyyppi (funktion esittely) kertoo kääntäjälle funktion ja sen argumenttien tyypit.
- `extern` ilmoittaa, että `pow` on määritelty jossakin muussa tiedostossa; ei ole välttämätön funktioiden yhteydessä.

Eri funktioilla voi olla sama nimi. Tällöin argumenttien tyyppien ja/tai lukumäärän tulee poiketa toisistaan. Sanotaan, että funktio on *ylikuormattu* (*overloaded*) (C++).

Esim.
```
float pow( int, int );
float pow( double, double );
    ...
x = pow( 2, 10 );    // pow( int, int );
y = pow( 2.0, 5.0 ); // pow( double, double );
```
Jos funktio ei palauta arvoa, sen tyypiksi on määriteltävä `void`.

Ohjelman rakenne

- Tyypillisesti ohjelma koostuu useista tiedostoista.
- Prototyypit ja muut yleiset esittelyt on syytä koota ns. *otsikkotiedostoon* (*header file*).
- `#include <`*tiedosto*`>` etsii *tiedostoa* standardihakemistosta, esim. `C:\Program Files\...\Vc7\include`.
- `#include "`*tiedosto*`"` etsii *tiedostoa* työhakemistosta.

Esim. Tiedostossa `tyypit.h`:
```
// tyypit.h
void f();
```
Tiedostossa `main.cpp`:
```
// main.cpp
#include "tyypit.h"
main()
{
   f();
}
```
Tiedostossa `huuhaa.cpp`:
```
// huuhaa.cpp
#include <iostream.h>
#include "tyypit.h"
void f()
{
   cout << "Huu Haa!" << "\n";
}
```

Luokat (C++)

Esim.
```
class complex{
  float re;
  float im;
public:
  void polar( float r, float phi );
  float abs();
};
```

- On määritelty uusi muuttujatyyppi. Lause
  `complex z1, z2;`
  määrittelee muuttujat eli *oliot* `z1` ja `z2`. Sanotaan, että ne ovat luokan `complex` *ilmentymiä* eli *instansseja*.
- `re` ja `im` ovat yksityisiä (private) *jäseniä (member)*.
- Jäsenet `polar` ja `abs` ovat julkisia (public); niihin voidaan viitata ilmentymien ulkopuolelta.
- Funktiojäseniä (`polar` ja `abs`) sanotaan *metodeiksi*.
- Jos `class`-sanan sijasta käytetään `struct`-sanaa, ovat luokan jäsenet automaattisest julkisia (ellei esim. `private:`-määreellä toisin ilmoiteta).

Metodien toteutus (implementation)
Esim.
```
void complex::polar( float rad, float angle )
{
  re = rad*cos( angle );
  im = rad*sin( angle );
}

float complex::abs()
{
  return sqrt( re*re + im*im );
}
```

- `::` on *luokkaerottelu*-operaattori;
  `complex::abs()` ilmoittaa, että on kyse `complex` luokan metodista `abs`.
- Metodit voivat viitata luokkansa yksityisiin jäseniin.

Viittaus olion jäseniin
Esim.
```
float x;
complex z;
  ...
z.polar( 5, 0.3 );
x = z.abs();
```

Operaattorien ylikuormaus
Esim.
```
class complex {
  ...
public:
  ...
  float real() { return re; }
  float imag() { return im; }
  void set( float r, float i )
    { re = r; im = i; }
  complex operator+( complex );
};
complex complex::operator+( complex z )
{
  complex sum;
  float r = re + z.real();
  float i = im + z.imag();
  sum.set( r, i );
  return sum;
}
```

- Funktion runko voidaan antaa esittelyn yhteydessä, ns. *inline*-funktio.
- Koska `re` ja `im` ovat yksityisiä jäseniä, tarvitaan julkiset funktiot viittaamaan niihin (`real()`, `imag()`,...).
- Operaattoria `+` käytetään kuten
  ```
  complex z1, z2, z3;
    ...
  z3 = z1 + z2;
    ...
  ```
  Tämä on sama kuin
  `z3 = z1.operator+( z2 );`
  ts. operaattorin `+` määrittelyssä esiintyvät muuttujat `re` ja `im` ovat vasemmanpuoleisen operandin jäseniä ja argumentti on oikeanpuoleinen operandi.

Muodostimet (constructors)
Esim.
```
class complex {
  ...
public:
  complex( float r, float i )
    { re = r; im = i; }
  complex( float r ) { re = r; im = 0; }
  complex() { re = 0; im = 0; }
  ...
};
  ...
complex z1( 1, 1 ), z2( 1 ), z3;
```
Muuttujaa määriteltäessä sille annetaan alkuarvo

- eksplisiittisesti, kuten `z1` ja `z2`:
  - `z1` alustetaan käyttäen *muodostinta* `complex::complex( float, float )`
  - `z2` alustetaan muodostimella `complex::complex( float )`
- implisiittisesti, kuten `z3`:
  - `z3` alustetaan *oletusmuodostimella* `complex::complex()`.
  - Mikäli ohjelmoija ei määrittele oletusmuodostinta, tuottaa kääntäjä sellaisen automaattisesti.

Johdetut luokat
Määritellään tason pistettä kuvaava luokka:
```
class point {
  float x;
  float y;
public:
```

```
  point() { x = 0; y = 0; }
  point( float s, float t ) { x = s; y = t; }
  void move( float dx, float dy )
    { x = x + dx; y = y + dy; }
};
```

Ympyrän määrää sen keskipiste ja säde:

```
class circle : public point {
  float r;
public:
  circle( float s, float t, float u ) :
    point( s, t ) { r = u; }
  circle( float u ) : point() { r = u; }
  circle() : point() { r = 1; }
};
```

- `circle : public point` tarkoittaa, että `circle` *perii* kaikki `point`-luokan ominaisuudet; `point` on *kantaluokka* ja `circle` on *johdettu* luokka.
- `public point` tarkoittaa, että `point`-olioiden julkiset jäsenet ovat myös `circle`-olioiden julkisia jäseniä, esim. `move(...)`.
- Muodostimissa
  `circle(...) : point(...) {...}`
  `: point(...)` ilmoittaa, miten kantaluokan jäsenet alustetaan.
- Kantaluokkaan kuuluvat jäsenet alustetaan *ennen* johdetun luokan omia jäseniä.
- Johdettu luokka voi määritellä uudelleen (*override*) kantaluokan metodit.

2. Esittelyt ja vakiot

2.1 Esittelyt (declarations)

Ennen nimen, tunnuksen (*identifier*) käyttöä se on esiteltävä.

Esim.

```
char ch;
int count=1;
struct complex { float re, im; }
complex z;
complex sqrt( complex );
extern int glob_count;
typedef complex point;
float real( complex p ) { return p.re; }
const double pi = 3.1415926535897932;
class huuhaa;
```

- Esittely sitoo nimeen tyypin.
- Usein esittely on myös määrittely (*definition*).
  - Muuttujien `ch`, `count`, `z` ja `pi` esittelyt ovat määrittelyjä; kääntäjä varaa niille tilaa muistista.
  - Muuttujat `count` ja `pi` alustetaan määriteltäessä.
  - Kääntäjä ei salli muuttujan `pi` arvoa muutettavan myöhemmin; `pi` on vakio.
  - `struct complex {...}` määrittelee uuden tyypin: `complex`.
  - `point` on määritelty tyypin `complex` synonyymiksi.
  - `real()` määrittelee funktion.
- `extern` ilmoittaa, että esiteltävä muuttuja on määritelty jossakin muualla, esim. toisessa tiedostossa.
- Funktioiden esittelyn yhteydessä `extern` on tarpeeton: `sqrt()` on jossakin muualla määritelty funktio.
- `huuhaa` on luokka, jonka määrittely seuraa myöhemmin.
- Jokainen käytetty tunnus on määriteltävä ohjelmassa täsmälleen yhden kerran.
- Tunnus voidaan esitellä useamman kerran. Jokaisessa esittelyssä täytyy tyypin olla sama.
- Määrittelyssä funktioille, tyypeille ja vakioille (`real()`, `complex`, `point` ja `pi`) annetut "arvot"ovat pysyviä. Esim. luokkaa `complex` ei saa myöhemmin määritellä uudelleen.

Vaikutusalue (scope)

Esitelty tunnus on käytettävissä vain tietyssä ohjelman alueessa.

- Funktion sisällä esitellyn tunnuksen vaikutusalue alkaa esittelystä ja päättyy esittelyn sisältävän lohkon loppuun: ns. paikallinen (local) tunnus.
- Funktion tai luokan ulkopuolella esitelty tunnus on voimassa esittelykohdasta esittelyn sisältävän tiedoston loppuun: ns. globaali tunnus.
- Lohkon sisällä esitelty tunnus voi kätkeä globaalin tai ympäröivässä lohkossa esitellyn tunnuksen. Poistuttaessa lohkosta tunnukselle palautuu sen alkuperäinen merkitys.

```
int x;          // globaali x

void f()
{
  int x;        // kätketään globaali x
  x = 1;        // sijoitus paikalliseen x
  {
    // Piilotetaan edellinen paikallinen x
    float x;
    x = 2.5;
  }
  x = 5;        // ensimmäinen paikallinen x
}

x = 7;          // globaali x
```

Kätkettyyn globaaliin tunnukseen voidaan viitata operaattorilla `::` (C++).

```
int x;          // globaali x

void f()
{
  int x = 1;    // kätketään globaali x
  ::x = 2;      // sijoitus globaaliin x
}
```

Kätkettyihin paikallisiin tunnuksiin ei voida viitata millään keinoin.

Elinikä

- Olio on alue koneen muistissa; oliolla on osoite.
- Lauseketta, joka viittaa olioon, ts. johonkin, jolle on varattu muistitilaa, sanotaan v-arvoksi (*lvalue*). Vain v-arvot voivat esiintyä sijoituslauseen vasemmalla puolella; kaikkiin v-arvoihin ei voida sijoittaa (`const`-muuttujat).
- Olio luodaan (pääsääntöisesti) sillä hetkellä, kun ohjelman suoritus on edennyt olion määrittelylauseeseen.
- Olio tuhotaan (pääsääntöisesti) sillä hetkellä, kun ohjelman suoritus on poistunut olion vaikutusalueelta.

Kuitenkin

- Globaaleja tunnuksia vastaavat oliot luodaan ja alustetaan vain yhden kerran —ennen `main`-funktion suoritusta.
- Globaalit oliot tuhotaan ohjelman päättyessä.
- `static`-muistinvarausluokkaan määritellyt paikalliset oliot luodaan ja alustetaan ennen `main`-funktion suoritusta ja tuhotaan ohjelman päättyessä.
- Niille `static`-muuttujille, joita ei eksplisiittisesti alusteta, annetaan alkuarvoksi 0.
- Operaattoreilla `new` ja `delete` voidaan luoda ja tuhota olioita, joiden elinikä on täysin ohjelmoijan kontrolloitavissa (C++).

Ohjelma

```
int a = 1;

void f()
{
  int b = 1;   // alustetaan aina
               // f:ään tultaessa
  static int c = 1; // alustetaan vain kerran
  cout << c++ << ". kerralla a = "
       << a++ << " ja b = " << b++ << "\n";
}

main()
{
  while( a < 4 ) f();
}
```

tulostaa

```
1. kerralla a = 1 ja b = 1
2. kerralla a = 2 ja b = 1
3. kerralla a = 3 ja b = 1
```

2.2 Tyypit

Perustyypit

Kokonaislukutyypit

- `char`, yleensä 1 tavun suuruinen.
- `short int`, yleensä 2 tavun suuruinen. `int` määre ei ole tässä välttämätön; voidaan jättää pois.
- `int`, yleensä 2 tavun suuruinen.
- `long int`, yleensä 4 tavun suuruinen. `int` määre ei ole tässä välttämätön; voidaan jättää pois.
- Jokaista kokonaislukutyyppiä voi edeltää atribuutti `unsigned`, jolloin ne esittävät etumerkittömiä kokonaislukuja. Lukualue on silloin $0 - 2\times$vastaavan etumerkillisen tyypin maksimiarvo.

Liukuluvut

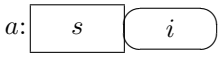
- `float`, yleensä 4 tavun suuruinen.
- `double`, yleensä 8 tavun suuruinen.

Implisiittinen tyyppien muunnos (conversion)

- Perustyyppejä voi käyttää sekaisin sijoituksissa ja lausekkeissa.
- Mikäli mahdollista arvot muunnetaan siten, että ei menetetä informaatiota.

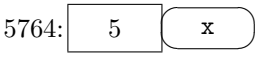
Osoittimet (pointers)

- Osoitin on muuttuja, jonka arvona on jonkin muuttujan osoite.
- Jos `v` on muuttuja, on `&v` ko. muuttujan osoite.
- Jos `p` on osoitin (muuttuja, lauseke), niin lausekkeen `*p` arvo on `p`:n osoittaman muistipaikan sisältö.
- Jos `T` on jokin tyyppi, niin `T*` on tyyppi "osoitin `T`:hen".



- i on muuttujan nimi.
- a on muistipaikan osoite.
- s on muuttujan arvo, ts. muistipaikan sisältö.

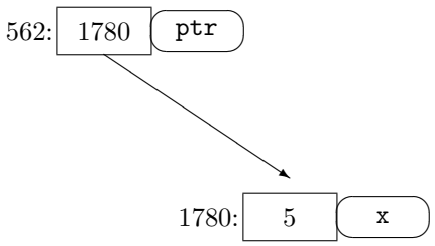
Esim. määrittelylauseesta `int x = 5;` generoituu muistiin



Todellinen osoite riippuu kääntäjästä, koneesta, käyttöjärjestelmästä ja ajoaikaisesta ympäristöstä.
Koodia

```
int x, *ptr;
...
ptr = &x;
x = 5;
```

vastaa konfiguraatio on

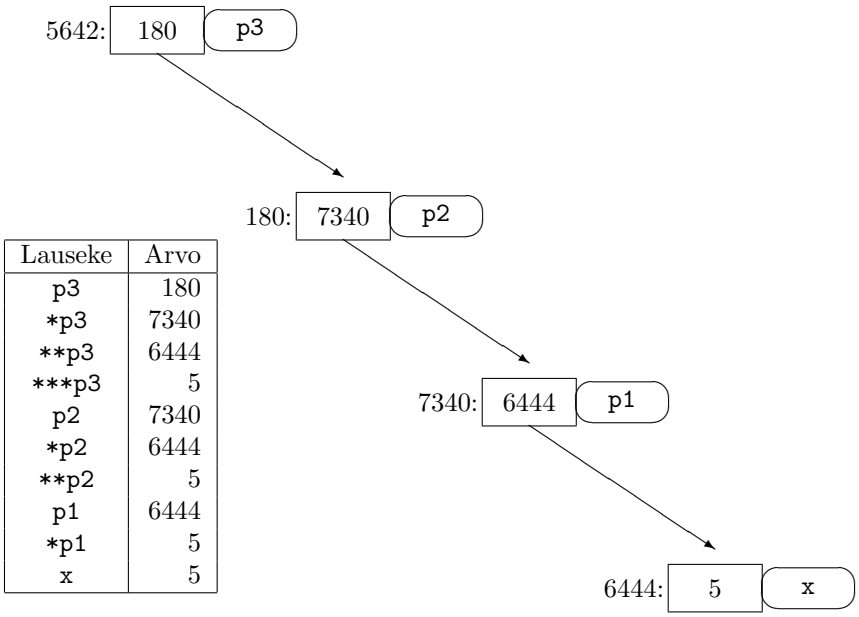


Lausekkeen `*ptr` arvo on sama kuin lausekkeen `x`, ts. 5.

Osoitin voi olla moninkertaisesti epäsuora. Koodi

```
int x, *p1, **p2, ***p3;
x = 5;
p1 = &x;
p2 = &p1;
p3 = &p2;
```

muodostaa konfiguraation



Lauseke	Arvo
p3	180
*p3	7340
**p3	6444
***p3	5
p2	7340
*p2	6444
**p2	5
p1	6444
*p1	5
x	5

Esim. Ohjelma

```
void swap( int*, int*);

main()
{
  int x = 5,
      y = 7;

  swap( &x, &y );
  cout << "x = " << x
       << ", y = " << y << "\n";
}

void swap( int *a, int *b )
{
  int z = *a;
  *a = *b;
  *b = z;
}
```

tulostaa

```
x = 7, y = 5
```

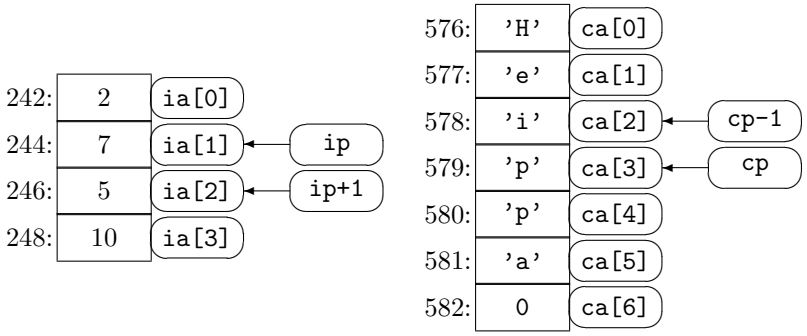
Osoitinaritmetiikkaa

Olkoot p, p_1 ja p_2 osoittimia tyyppiä `T` oleviin olioihin ja olkoon n kokonaisluku. Osoittimille on voimassa aritmeettiset toiminnot

- $p + n$ on osoitin `T`-tyyppiseen olioon, jonka osoite on $p + n \times$ (olion koko).
- $p - n$ on osoitin `T`-tyyppiseen olioon, jonka osoite on $p - n \times$ (olion koko).
- p`++` osoittaa muistissa seuraavaan `T`-tyyppiseen olioon.
- p`--` osoittaa muistissa edelliseen `T`-tyyppiseen olioon.

- $p_1 - p_2$ on kokonaisluku, joka ilmoittaa, kuinka monta `T`-tyyppistä oliota on muistipaikkojen p_1 ja p_2 välillä.

```
int ia[4] = { 2, 7, 5, 10 };
char ca[7] = { 'H', 'e', 'i', 'p',
               'p', 'a', '\0' };
int *ip = &ia[1];
char *cp = &ca[3];
```



Merkkijonoista

- C *ei* tunne merkkijonomuuttujia.
- Merkkijonot esitetään merkkitaulukkoina.
- ASCII-merkkijonon päättyminen on tapana ilmaista merkillä `\0`; useimmat ASCII-jonoja käsittelevät kirjastofunktiot noudattavat tätä sopimusta.

```
// Merkkijonon pituus
int strlen( char *p )
{
  int i = 0;
  while( *p++ != '\0' ) i++;
  return i;
}
...
char cc[1000];
int l;
...
l = strlen( &cc[0] );
```

Huom.
Operaattorien `[]` (taulukko) ja `()` (funktio) prioriteetti on korkeampi kuin osoitinoperaattorin `*`. Siksi

- `int *f( char, char* );` esittelee funktion, joka palauttaa arvonaan osoittimen kokonaislukuun, ja jonka argumentteina ovat merkki ja merkkiosoitin.
- `int (*f)( char, char*);` määrittelee muuttujan `f` siten, että se on osoitin kokonaisluvun palauttavaan funktioon, jonka argumentteina ovat merkki ja merkkiosoitin.
- `int *v[10];` määrittelee 10 alkioisen vektorin, jonka alkioina ovat osoittimet kokonaislukuihin.
- `int (*v)[10];` määrittelee muuttujan `v`, joka on osoitin 10 kokonaisluvun muodostamaan vektoriin.

Taulukot

Jos `T` on jokin tyyppi, niin `T[]` on `T`-tyyppisten olioiden taulukko. Esim.

`float v[3];`	3 liukuluvun vektori
`int a[2][5];`	2 viiden kokonaisluvun vektoria
`char *cp[10];`	10 merkkiosoittimen vektori

- Taulukon indeksit juoksevat 0:sta (koko-1):een.
- Taulukot voidaan alustaa määriteltäessä kuten
  `int a[] = { 1, 4, 2 };`
  Kääntäjä laskee tällöin taulukon koon (tässä 3).
- Merkkitaulukot voidaan alustaa kuten
  `char c[] = "Heippa!";`
  Kääntäjä liittää merkkijonon loppuun automaattisesti päättävän loppu-`'\0'`:n. Taulukon koko on siis merkkijonon pituus + 1 (tässä 8).

Moniulotteiset taulukot

- Esitetään vektoreiden vektoreina. Esim.
  `char c[2][5];`
  määrittelee kaksialkioisen vektorin, jonka molemmat alkiot ovat tyyppiä `char[5]`.
- Talletetaan muistiin siten, että oikean puoleinen indeksi juoksee nopeimmin. Tämä on huomioitava myös alustettaessa, esim. määritelmässä
  ```
  char c[2][5] = {
       'a', 'b', 'c', 'd', 'e',
       '1', '2', '3', '4', '5'
       };
  ```
  c:n ensimmäinen vektori muodostuu 5 kirjaimesta ja toinen 5 numerosta.

```
// Vuodenpäivä
int yearday( int y, int m, int d )
{
  static int mdays[2][13] = {
        // Tavallinen vuosi
    0, 31, 28, 31, 30, 31, 30,
       31, 31, 30, 31, 30, 31,
        // Karkausvuosi
    0, 31, 29, 31, 30, 31, 30,
       31, 31, 30, 31, 30, 31
  };
  int leap =
    (y%4 == 0 && (y%100 != 0 || y%400 == 0));
  for( int i = 1; i < m; i++ )
    d = d + mdays[leap][i];
  return d;
}
```

Logiikasta

- `&&` on looginen *ja*.
- `||` on looginen *tai*.
- Loogisten lausekkeiden arvo on 1, jos lauseke on tosi ja 0, jos lauseke on epätosi.
- Kääntäen 0 vastaa totuusarvoa *epätosi* ja nollasta poikkeava kokonaisluku totuusarvoa *tosi*.

Esim.

```
// Merkkijonon pituus
int strlen( char *p )
{
  int i = 0;
  while( *p++ )
    i++;
  return i;
}
```

Taulukot ja osoittimet

Taulukon nimi on osoitin taulukon ensimmäiseen alkioon. Koodi

```
int strlen( char* );
char v[] = "Huuhaa";
char *p = v;
cout << strlen( v ) << " = "
     << strlen( p ) << "\n";
```

tulostaa

```
6 = 6
```

- Määrittely `char *p = v;` on yhtenevä määrittelyn `char *p = &v[0];` kanssa.
- Lauseke `v[i]` on yhtenevä lausekkeen `*(v+i)` kanssa (joka on sama kuin `*(p+i)`).
- Taulukon tunnus on vakio, ts. lausekkeet `v = v + 2` ja `v++` ovat kiellettyjä.

Huom.
Osoittimien vähennyslasku on määritelty ainoastaan silloin, kun molemmat osoittimet viittaavat samaan vektoriin (vaikkakaan kääntäjä ei voi tätä tarkistaa).

```
int v1[10];
int v2[20];
int i, *p;
i = &v1[5] - &v1[3];  // 2
i = &v1[5] - &v2[3];  // epämääräinen
p = v2 + 2;           // p = &v2[2];
p = v2 - 2;           // *p epämääräinen
```

Tietueet

Tietue on (lähes) mielivaltaista tyyppiä olevien alkioiden muodostama kokonaisuus. Esim.

```
struct osoite {
  char *nimi;     // Kuka Lienee
  int numero;     // 19
  char *katu;     // Mikätie
  char *kaupunki; // Oulu
  char maa[3];    // FIN
  long postinro;  // 90570
};
```

määrittelee uuden tyypin `osoite` (C++). Tyyppiä `osoite` olevat muuttujat määritellään kuten muutkin muuttujat. Tietueen jäseniin viitataan `.`-operaattorilla.

```
osoite pp;
pp.kaupunki = "Oulu";
pp.postinro = 90630;
```

Huom. Merkkijonovakion (`"..."`) arvo on osoitin jonon ensimmäiseen merkkiin.

Tietuetyyppiset muuttujat voidaan alustaa taulukkojen tavoin. Esim.

```
osoite AA = {
  "Aku Ankka",
  13, "Ankankuja",
  "Ankkalinna", {'U', 'S', 'A'},
  90630
};
```

Muodostimien käyttö on kuitenkin suositeltavampaa (C++).

Huom. Jäsentä `AA.maa` ei voi alustaa merkkijonolla `"USA"`, sillä sen vaatima tila on 4 merkkiä (loppu-\0).

Tietuetyyppisiä muuttujia voidaan sijoittaa, käyttää funktion argumentteina ja palauttaa funktion arvona, esim.

```
osoite nykyinen;

osoite aseta_osoite( osoite uusi )
{
  osoite vanha = nykyinen;
  nykyinen = uusi;
  return vanha;
}
```

Tietueen tyyppinimi on käytettävissä välittömästi. Esim.

```
struct link {
  link *next;
  link *prev;
};
```

Huom.

- Tietueen koko (tavuina) ei ole yleensä yksittäisten jäsenten kokojen summa. Konearkkitehtuurista riippuen kääntäjä voi joutua sijoittamaan esim. `int`-tyyppiset oliot parillisiin osoitteisiin, jolloin tietueeseen voi jäädä "reikiä".
- C:n operaattori `sizeof` laskee operandinsa koon tavuina, esim. `sizeof( osoite )`.
- Tietuetyyppistä oliota ei saa määritellä ennen kuin tietueen määritelmä on täydellinen. Esim. määritelmä

  ```
  struct laiton {
      laiton alkio;
  };
  ```

  on virheellinen.

Tietueet C:ssä

C:ssä on esittelyissä ja määrittelyissä käytettävä avainsanaa `struct`, ts. muuttuja `AA` on määriteltävä

```
struct osoite AA;
```

Samoin

```
struct osoite nykyinen;

struct osoite aseta_osoite(struct osoite uusi)
{
  struct osoite vanha = nykyinen;
  nykyinen = uusi;
  return vanha;
}
```

ja

```
struct link {
  struct link *next;
  struct link *prev;
};
```

Tyyppien yhtenevyys

Lauseilla

```
struct s1 { int a; };
struct s2 { int a; };
```

määritellyt tietuetyypit `s1` ja `s2` *eivät* ole samoja vaikka niillä on samat jäsenet. Esim. määrittely

```
s1 x;
s2 y = x; // tyypit eivät täsmää
```

on virheellinen. Tietuetyypit poikkeavat myös perustyypeistä: määrittely

```
s1 x;
int i = x; // tyypit eivät täsmää
```

on virheellinen.

typedef

Tyyppiyhtenevyyksiä voidaan määritellä, kuten

```
typedef int Pituus;
typedef char* Pchar;
typedef float (*Pfun)( int, char* );
```

- `Pituus` on yhtenevä tyypin `int` kanssa.
- `Pchar cp;` määrittelee muuttujan `cp` osoittimeksi merkkiin.
- `Pfun f;` määrittelee muuttujan `f` siten, että se on osoitin funktioon, jonka argumenttina on kokonaisluku ja merkkiosoitin ja jonka arvona on `float`-liukuluku.

Tietuetyypit C:ssä

C:ssä on tapana kirjoittaa esim.

```
struct s { int a; float f; };
typedef struct s S;
```

tai

```
typedef struct s { int a; float f; } S;
```

tai

```
typedef struct { int a; float f; } S;
```

Tunnus `S` on nyt uusi tyyppinimi, ts.

```
S x;
```

määrittelee muuttujan `x` siten, että se on ko. tietue.

Viittaukset (references)

Jos `T` on tyyppi, niin `T&` on tyyppi "viittaus `T`:hen". Viittaus on olion vaihtoehtoinen nimi. Esim.

```
int i = 1;
int& r = i;  // r ja i viittaavat samaan
             // int-olioon
int x = r;   // x = 1
r = 2;       // i = 2
```

- Viittaava muuttuja on alustettava määriteltäessä.
- Viittauksiin kohdistetut toimenpiteet vaikuttavat tosiasiassa siihen olioon, johon viittaus viittaa; esim.
  `r++;`
  kasvattaa muuttujan `i` arvoa yhdellä.
- Kääntäjä toteuttaa viittaukset osoittimina.

Viittauksia käytetään pääasiassa toimenpiteitten toteutukseen ohjelmoijan määrittelemille tyypeille. Esim. operaattori `<<` lausekkeessa `cout << x` on määritelty siten että sen arvona on viittaus tyyppiä `ostream_withassign` olevaan olioon `cout`.
Viittauksia käytetään myös funktion argumentteina. Esim.

```
void swap( int&, int& );  // Prototyyppi
main()
{
  int x = 1;
  int y = 2;
  swap( x, y ); // x = 2, y = 1
}

void swap( int& a, int& b )
{
  int c = a;
  a = b;
  b = c;
}
```

2.3 Vakiot

Kokonaislukuvakiot

Kokonaislukuvakiot voidaan ilmaista

- desimaalisina, kuten
  `0  154  -2765987463`
- oktaalisina, kuten
  `0  0154  077`
  Oktaalivakiot alkavat luvulla `0`, jota seuraavat numerot ovat välillä 0–7.
- heksadesimaalisina, kuten
  `0x0  0x2  0x6e 0xffff`
  Heksadesimaaliluvut alkavat merkkijonolla `0x` tai `0X`, jota seuraavat merkit ovat 16-järjestelmän numeroita välillä 0–9 tai `a, b, c, d, e` tai `f` (myös isot kirjaimet kelpaavat).
- ASCII-merkkeinä.

Merkkivakiot

Merkkivakiot ilmaistaan yläpilkkujen (') välissä

- ASCII-merkkinä: `'a' 'A' '0'`
- 1–3 numeroisena oktaalilukuna: `'\0' '\6' '\60' '\137'`
- 1–3 numeroisena heksadesimaalilukuna: `'\x5' '\x30' '\x06f'`
- standardinimenä

`'\a'`	"piip"
`'\b'`	backspace
`'\f'`	uusi sivu
`'\n'`	uusi rivi
`'\r'`	telan palautus
`'\t'`	tabulaattori
`'\v'`	vertikaalinen tabulaattori
`'\\'`	\
`'\''`	yläpilkku
`'\"'`	lainausmerkki

Liukulukuvakiot

Esim.
`1.23  .23  0.23  1.  1.0  1.2e10  1.23e-15`
Liukulukuvakio on tyypiltään `double`.

Huom. `1.23 e-15` ei ole liukuluku; välilyönti ei ole sallittu.

Merkkijonot (strings)

Merkkijonovakiot ilmaistaan lainausmerkkien (") välissä ASCII-merkkien, \-merkkiä seuraavien oktaalilukujen, heksadesimaalilukujen tai standardimerkkinimien jonona. Esim.
`"Huu haa\a\n"`
Kääntäjä liittää jonon loppuun automaattisesti merkkijonon päättävän \0-merkin. Merkkijonovakio on tyypiltään "sopivan suuruinen merkkitaulukko"(merkkien lukumäärä + 1).

Const

Atribuutti `const` tekee oliosta vakion. Esim.
`const float Pi = 3.14159265;`

- `const`-muuttujat on alustettava.
- `const` modifioi tyyppiä, kuten:
  - `const char *pc = "abcdef";` määrittelee `pc`:n osoittimeksi vakioon; esim. `pc[3]='h';` on kielletty, mutta `pc = "ghi";` on sallittu.
  - `char *const cp = "abcdef";` määrittelee `cp`:n vakio-osoittimeksi; esim. `cp[3] = 'h';` on sallittu, mutta `cp = "ghi";` on kielletty.

Enum

Kokonaislukuvakioita voidaan määritellä myös avainsanalla `enum`:
`enum {X, Y, Z};`
on yhtenevä määritelmien

```
const int X = 0;
const int Y = 1;
const int Z = 2;
```
kanssa. Numeroinnille (*enumeration*) voidaan antaa nimi, jota voidaan käyttää myöhemmin tyyppinimenä (C++). Esim.
```
enum suunta { X, Y, Z };
float nopeus[3];
suunta s = Z;
nopeus[X] = 37.5;
nopeus[s] = 12.6;
```
Oletusnumeroinnin (0, 1, 2, ...) sijaan voidaan `enum`-vakioiden arvot antaa eksplisiittisesti:
```
enum status {
   overflow = 0x800,
   zero = 0x40,
   carry = 1
};
```

Enum C:ssä

C:ssä `enum`-muuttujat on määriteltävä avainsanalla `enum`:
```
enum suunta { X, Y, Z };
float nopeus[3];
enum suunta s = Z;
nopeus[X] = 37.5;
nopeus[s] = 12.6;
```

3. Operaattorit ja lausekkeet

3.1 Operaattorit

C++:n (C:n) operaattori on tyypiltään

- unaarinen: yksi operandi, esim. `*` lausekkeessa `*ptr`.
- binäärinen: kaksi operandia, esim. `+` lausekkeessa `a + b`.
- Infix-operaattori: `a + b` (binäärinen).
- Postfix-operaattori: `a++` (unaarinen).
- Prefix-operaattori: `++a` (unaarinen).

Operaattorien suoritusjärjestyksen määrää

- presedenssitaso: mitä pienempi taso, sitä aikaisemmin operaattoria sovelletaan. Esim. lauseke `a + b * c` on sama kuin `a + ( b * c )` (`*` on tasolla 4 ja `+` tasolla 5). Suluilla (`()`) voidaan suoritusjärjestystä muuttaa.
- oikealle assosiatiivisuus: `a = b = c` on sama kuin `a = ( b = c )` ja `*p++` on sama kuin `*(p++)`
- vasemmalle assosiatiivisuus: `a + b + c` on sama kuin `( a + b ) + c`

Unaari- ja sijoitusoperaattorit sekä ehdollinen lauseke (`?:`) ovat oikealle assosiatiivisia, muut vasemmalle.
Operaattorin ylikuormaus (C++) ei muuta sen tyyppiä tai suoritusjärjestystä.

Presedenssitaso 1

Op	Kuvaus	Syntaksi
`::`	luokkaerottelu	*luokkanimi*`::`*jäsen*
`::`	globaali erottelu	`::`*nimi*

`::` vain C++:ssa

Presedenssitaso 2

Op	Kuvaus	Syntaksi
`()`	funktiokutsu	*lauseke*`(`*lausekelista*`)`
`()`	muodostin	*tyyppi*`(`*lausekelista*`)`
`[]`	indeksointi	*osoitin*`[`*lauseke*`]`
`.`	jäsenvalinta	*nimi*`.`*jäsen*
`->`	jäsenvalinta	*osoitin*`->`*jäsen*
`sizeof`	olion koko	`sizeof` *lauseke*
`sizeof`	tyypin koko	`sizeof(`*tyyppi*`)`

`->` `p->j` on lyhennysmerkintä lausekkeelle `(*p).j` Esim.

```
struct complex { float r, float i };
complex z, *zp;
zp = &z;
zp->r = 5; // sama kuin (*zp).r = 5;
```

`sizeof` Olion tai tyypin koko tavuina. Esim. koodi

```
int i[100];
cout << sizeof i << " = "
     << 100*sizeof(int) << "\n";
```

tulostaa (WINDOWSissa)

```
400 = 400
```

Presedenssitaso 3

Op	Kuvaus	Syntaksi
`!`	looginen ei	`!`*lauseke*
`~`	1:n komplementti	`~`*lauseke*
`+`	unaarinen plus	`+`*lauseke*
`-`	unaarinen miinus	`-`*lauseke*
`++`	post-inkrementointi	*v-arvo*`++`
`++`	pre-inkrementointi	`++`*v-arvo*
`--`	post-dekrementointi	*v-arvo*`--`
`--`	pre-dekrementointi	`--`*v-arvo*
`&`	osoite	`&`*v-arvo*
`*`	sisältö	`*`*lauseke*
`()`	tyyppimuunnos (cast)	(*tyyppi*) *lauseke*
`new`	luonti (muistivaraus)	`new` *tyyppi*
`delete`	tuhoaminen	`delete` *osoitin*
`delete[]`	taulukon tuhoaminen	`delete[]` *osoitin*

`~` on bitittäinen toiminto; jos `char a`= 00101101_2, niin `~a`= 11010010_2.

`++` Lausekkeen `i++` arvo on muuttujan `i` arvo ennen lisäystä ja lausekkeen `++i` arvo on muuttujan `i` arvo lisäyksen jälkeen. Esim. koodi

```
int i = 0;
while( i++ < 4 ) cout << i << " ";
cout << "\n";
i = 0;
while( ++i < 4 ) cout << i << " ";
cout << "\n";
```

tulostaa

```
1 2 3 4
1 2 3
```

`--` Lausekkeen `i--` arvo on vähennystä edeltävä `i`:n arvo ja lausekkeen `--i` arvo vähennyksen jälkeinen arvo.

`()` Cast-toiminnolla tehdään eksplisiittinen tyyppimuunnos (konversio). Esim.

```
char c[100];
char *cp = c;
int i = 2, j = 3;
float z;
z = (float)i/(float)j;
*(int *)cp = 1992;
```

Huom. C++-ohjelmissa tulisi käyttää cast-operaattorin `()` sijasta ns. turvallisia tyyppikonversioita. Samankaltaisten tyyppien välinen konversio voidaan tehdä `static_cast`-operaattorilla, esim.

```
z = static_cast<float>(i)/static_cast<float>(j);
*static_cast<int*>(cp) = 1992;
```

Luomis- ja tuhoamisoperaattorit (C++)

`new`

Operaattori `new` varaa muistista tilaa oliolle. Toiminnon arvona on osoitin operandintyyppiseen olioon. Esim.

```
struct complex { float r, i; };
int *ip;
complex *cp, *cpp;
ip = new int[100];
ip[12] = 576;
*(ip + 13) = 48;
cp = new complex;
cp->r = 1.0;
cpp = new complex[10];
cpp[5].r = 5.0;
(cpp + 5)->i = 1.0;
```

Jos luotavalle oliolle on määritelty muodostin, tämän parametrit voidaan antaa suluissa; esim.

```
struct complex {
  float r, i;
  complex( float R, float I ) { r = R; i = I;}
  complex() { r = 0; i = 0; }
};

complex *cp = new complex( 1, 0 );
complex *zp = new complex;
int *ip = new int(3);
```

`*cp` alustetaan muodostimella `complex::complex( float, float )` ja `*zp` oletusmuodostimella `complex::complex()`. Perustyypeille (`char, int,...`) on määritelty muodostimet luonnollisella tavalla.

Jos muistissa ei ole tilaa luotavalle oliolle, tuloksena on nollaosoitin (`0`), joka ei voi olla minkään todellisen olion osoite.

`delete`

Tuhoamisoperaattorilla `delete` palautetaan systeemille aiemmin muistista varattu alue. Taulukolle varattu tila vapautetaan operaattorilla `delete[]`.

Operaattorin `delete` operandina *täytyy* olla operaattorin `new` antama osoite. Esim.

```
int *ip, *ipp;
ip = new int[200];
ipp = ip++;
delete[] ipp;  // ok
delete[] ip;   // virhe
```

Dynaaminen muistinhallinta C:ssä

C:ssä oliolle voidaan varata dynaamisesti tilaa ja vapauttaa olioiden käyttämä tila standardikirjaston funktioilla. Näistä tärkeimmät ovat (prototyypit tiedostossa `stdlib.h`):

- `void *malloc( size_t` *size* `);`
  Allokoi muistiblokin, jonka koko on *size* tavua ja palauttaa arvonaan `void *`-tyyppisen osoittimen lohkon ensimmäiseen tavuun. Mikäli tilaa ei ole tarpeeksi, arvona on osoitin `NULL`, joka on määritelty mm. tiedostossa `stdlib.h` (mallista riippuvaksi) kokonaisluvuksi 0. Samoin `size_t` on `typedef`-

määritelty kokonaislukutyypiksi.

- `void *calloc( size_t` *nitems*`, size_t` *size* `);`
  Varaa muistia *nitems* kappaleelle *size* tavun kokoista oliota ja palauttaa osoittimen ensimmäisen olion ensimmäiseen tavuun tai `NULL`in, mikäli muistia ei ole tarpeeksi.

- `void free( void *`*block* `);`
  Palauttaa systeemille aiemmin varatun lohkon. Osoittimen *block* täytyy olla funktion `calloc` tai `malloc` antama osoite.

Huom. `void*` on tyyppi "osoitin mihin tahansa"; se muuntuu automaattisesti minkä hyvänsä tyyppiseksi osoittimeksi ja kääntäen mikä tahansa osoitin muuntuu automaattisesti `void*` osoittimeksi.

Esim.
```
typedef struct{ float r, i; } complex;
complex *zp1, *zp2;
zp1 = malloc( 100*sizeof( complex ) );
zp2 = calloc( 100, sizeof( complex ) );
(zp1 + 3 )->r = 1.5;
zp2[5].r = zp1[3].r;
free( zp1 );
free( zp2 );
```

Presedenssitaso 4

Op	Kuvaus	Syntaksi
`.*`	jäsenviittaus	*luokka*`.*`*osoitin*
`->*`	jäsenviittaus	*luokkaosoitin*`->*`*osoitin*

Luokan jäsenten viittausoperaattorit (`.*` ja `->*`) ovat käytettävissä vain C++:ssa. Operaattoria `.*` *ei* voi ylikuormata.

Presedenssitaso 5

Op	Kuvaus	Syntaksi
`*`	kertaa	*lauseke*`*`*lauseke*
`/`	jako	*lauseke*`/`*lauseke*
`%`	modulo	*lauseke*`%`*lauseke*

Presedenssitaso 6

Op	Kuvaus	Syntaksi
`+`	plus	*lauseke*`+`*lauseke*
`-`	miinus	*lauseke*`-`*lauseke*

Presedenssitaso 7

Op	Kuvaus	Syntaksi
`<<`	siirto vasemmalle	*v-arvo*`<<`*lauseke*
`>>`	siirto oikealle	*v-arvo*`>>`*lauseke*

Siirto(shift)-operaattorit siirtävät bitittäin vasemmanpuoleista operandiansa oikeanpuoleisen operandin ilmaiseman kokonaislukumäärän vasemmalle tai oikealle. Vapautuvat bittipaikat täytetään nollilla.
Jos esim. `char a`$= 01101110_2$, niin `a << 4`$= 11100000_2$ ja `a >> 3`$= 00001101_2$.

Presedenssitaso 8

Op	Kuvaus	Syntaksi
`<`	pienempi	*lauseke*`<`*lauseke*
`<=`	pienempi tai yhtäsuuri	*lauseke*`<=`*lauseke*
`>`	suurempi	*lauseke*`>`*lauseke*
`>=`	suurempi tai yhtäsuuri	*lauseke*`>=`*lauseke*

Presedenssitaso 9

Op	Kuvaus	Syntaksi
`==`	yhtäsuuri	*lauseke*`==`*lauseke*
`!=`	erisuuri	*lauseke*`!=`*lauseke*

Presedenssitaso 10

Op	Kuvaus	Syntaksi
`&`	bitittäinen ja	*lauseke*`&`*lauseke*

Presedenssitaso 11

Op	Kuvaus	Syntaksi
`^`	yksinomainen tai	*lauseke*`^`*lauseke*

Presedenssitaso 12

Op	Kuvaus	Syntaksi
`|`	bitittäinen tai	*lauseke*`|`*lauseke*

Jos `char a`$= 10011011_2$ ja `char b`$= 01010001_2$, niin
`a & b` $= 00010001_2$,
`a ^ b` $= 11001010_2$ ja
`a | b` $= 11011011_2$.

Presedenssitaso 13

Op	Kuvaus	Syntaksi
`&&`	looginen ja	*lauseke*`&&`*lauseke*

Presedenssitaso 14

Op	Kuvaus	Syntaksi
`||`	looginen tai	*lauseke*`||`*lauseke*

Loogisten toimintojen (`&&, ||, !` ja vertailujen) arvo on 1, jos lauseke on tosi, muutoin 0.

Presedenssitaso 15

Op	Kuvaus	Syntaksi
`?:`	aritmeettinen jos	*lauseke*`?`*lauseke*`:`*lauseke*

Aritmeettinen jos (ehdollinen lauseke, conditional expression) operaattori on oikeastaan trinäärinen. Ehdollisen lausekkeen arvo on toinen *lauseke*, jos ensimmäinen *lauseke*$\neq 0$ (tosi), muulloin viimeinen *lauseke*. Esim. `max = a > b ? a : b;`.

Presedenssitaso 16

Op	Kuvaus	Syntaksi
`=`	sijoitus	*v-arvo*`=`*lauseke*
`*=`	kerto ja sijoitus	*v-arvo*`*=`*lauseke*
`/=`	jako ja sijoitus	*v-arvo*`/=`*lauseke*
`%=`	modulo ja sijoitus	*v-arvo*`%=`*lauseke*
`+=`	plus ja sijoitus	*v-arvo*`+=`*lauseke*
`-=`	miinus ja sijoitus	*v-arvo*`-=`*lauseke*
`&=`	ja ja sijoitus	*v-arvo*`&=`*lauseke*
`^=`	yks. tai ja sijoitus	*v-arvo*`^=`*lauseke*
`|=`	tai ja sijoitus	*v-arvo*`|=`*lauseke*
`<<=`	vasen siirto ja sijoitus	*v-arvo*`<<=`*lauseke*
`>>=`	oikea siirto ja sijoitus	*v-arvo*`>>=`*lauseke*

Kaikilla C:n lausekkeilla on arvo: siis myös sijoituslausekkeella. Ilmaisun *v* `=` *e* arvo on se arvo, joka sijoitetaan muuttujaan *v*. Esim.

```
// Merkkijonon kopiointi
void strcpy( char *dest, char *src )
{
  while( *dest++ = *src++ );
}
```

Yhdistetty sijoituslauseke *v op*= *e* on (lähes) yhtenevä lausekkeen *v* = *v op e* kanssa. Esim.

```
// Kertoma
int fact( int n )
{
  int f = 1;
  for( int i = 2; i <= n; i++ )
    f *= i; // sama kuin f = f*i;
  return f;
}
```

Huomaa kuitenkin että esim.

```
*p++ += 5;
```

on sama kuin

```
*p = *p + 5; p++;
```

eikä ole yhtenevä lauseen

```
*p++ = *p++ + 5;
```

kanssa. Jälkimmäisen tyyppinen ilmaisu ei ole hyvin määritelty (voi antaa eri kääntäjillä eri tuloksen), sillä (useimmissa tapauksissa) C (C++) ei yksilöi lausekkeissa esiintyvien alilausekkeiden ajallista suoritusjärjestystä.

Presedenssitaso 17

Op	Kuvaus	Syntaksi
,	pilkku	*lauseke*,*lauseke*

Sekvenssointioperaattori (,) yhdistää kaksi lauseketta yhdeksi. Lausekkeiden arvon määrittämisjärjestys on vasemmalta oikealle, joten yhdistetyn lausekkeen arvo on oikeanpuoleisen operandin arvo.

```
void f1( int );
void f2( int, int );
int i, j;
j = ( i = 2, i+1 ); // j == 3

for( i = 0, j = 1; i < 4; i++, j++ )
{ .... }

f1( ( i, j++ ) );   // yksi argumentti
f2( i, j++ );       // kaksi argumenttia
```

Määrittämisjärjestys

C takaa, että operaattorien

`,  &&  ||`

vasemmanpuoleisen operandin arvo määritetään ennen oikeanpuoleista operandia.

`&&` Jos loogisen `ja`-operaattorin vasemmanpuoleinen operandi on 0, niin oikeanpuoleista operandin arvoa *ei* määritetä lainkaan. Toiminnon arvo on 0 (epätosi).

```
for( i = 5; i < 5 && *cp++; i++ )
// toimintoa *cp++ ei tehdä koskaan
```

`||` Jos loogisen tai operaattorin vasemmanpuoleinen operandi on $\neq 0$, niin oikeanpuoleista operandin arvoa *ei* määritetä lainkaan. Toiminnon arvo on 1 (tosi).

```
for( i = 0; i < 5 || *cp++; i++ )
// toiminto *cp++ tehdään vasta kun i=5
```

3.2 Käskyt (lauseet, statements)

Syntaksi (lauseoppi)

käsky:

esittely
{*käskylista*$_{\text{opt}}$}
lauseke$_{\text{opt}}$;
`if(` *lauseke* `)` *käsky*
`if(` *lauseke* `)` *käsky* `else` *käsky*
`switch(` *lauseke* `)` *käsky*
`while(` *lauseke* `)` *käsky*
`do` *käsky* `while(` *lauseke* `);`
`for(` *käsky lauseke*$_{\text{opt}}$`;` *lauseke*$_{\text{opt}}$ `)` *käsky*
`case` *vakiolauseke* `:` *käsky*
`default :` *käsky*
`break ;`
`continue ;`
`return` *lauseke*$_{\text{opt}}$ `;`
`goto` *tunnus* `;`
tunnus `:` *käsky*

käskylista:

käsky
käsky käskylista

Huom.

- Alaindeksi $_{opt}$ tarkoittaa, että ko. käskylista tai lauseke ei ole pakollinen.
- Esittely on käsky.
- C:ssä ei ole sijoituskäskyä; sijoitukset käsitellään lausekkeina.
- C:ssä ei ole aliohjelmankutsukäskyä (FORTRANin `CALL`); funktion kutsu on lauseke.
- C:ssä ei liioin ole I/O käskyjä (FORTRANin `READ, WRITE`); I/O hoidetaan funktioilla.

If

Ilmaisussa `if(` *lauseke* `)` *käsky* suoritetaan *käsky*, mikäli *lauseke* $\neq 0$. Esim. koodi

```
if( a ) ...
```

on yhtenevä koodin

```
if( a != 0 ) ...
```

kanssa.

Break

Käsky `break;` siirtää suorituksen ulos parhaillaan suoritettavasta silmukasta tai `switch`-rakenteesta. Esim.

```
int i = 0;
while( 1 ) {
  if( i++ > 4 )
    break;
}
// Suoritus jatkuu tästä break:in jälkeen
```

Continue

Käsky `continue;` siirtää suorituksen parhaillaan suoritettavan silmukan loppuun. Esim.

```
for( i = 0; i < 100; i++ ) {
  if( i == 5 ) continue;
  // Näitä käskyjä ei suoriteta,
  // jos i == 5. Suoritus jatkuu
  // lisäyksestä i++
  ...
}
```

Do-while

Rakenteessa `do-while` silmukointiehto testataan silmukan lopussa. Esim.

```
i = 5;
do {
   // Suoritetaan vähintään kerran
   ...
} while( i < 4 );
```

4. Funktiot ja tiedostot

- Useimmiten ohjelma kannattaa jakaa useisiin tiedostoihin (moduleihin).
- Useasta tiedostosta koostuvassa ohjelmassa täytyy tunnusten ja tyyppien olla yksikäsitteisiä —niitä on käytettävä aivan kuin ohjelma olisi kirjoitettu yhdeksi tiedostoksi.
- Yksikäsitteisyyden ylläpitoa voidaan auttaa kirjoittamalla eri modulien tarvitsema tyyppi-informaatio (esittelyt) otsikkotiedostoiksi, jotka liitetään `#include`-direktiiveillä käännettävään tiedostoon.

4.1 Linkitys

Pääsääntöisesti tunnuksen, joka ei ole jonkin funktion paikallinen tunnus, täytyy jokaisessa erikseen käännettävässä ohjelman modulissa viitata samaan tyyppiin, arvoon, funktioon tai olioon. Esim.

```
// file1.c
int a = 1;
int f() { ... }

// file2.c
extern int a;
int f();
void g() { a = f(); }
```

- Tiedostossa `file2.c` käytetyt `a` ja `f()` on määritelty tiedostossa `file1.c`.
- Avainsana `extern` tunnuksen `a` esittelyssä ilmoittaa, että kyseessä ei ole määrittely.
- Jos `extern`-esiteltävä olio alustetaan, kyseessä on määrittely: `extern`-atribuutti jätetään huomiotta.
- Jokainen ohjelman olio on määriteltävä täsmälleen kerran.

Jos ohjelma koostuu esim. seuraavista tiedostoista

```
// file1.c
int a = 1;
int b = 1;
extern int c;

// file2.c
int a;
extern double b;
extern int c;
```

niin siinä on kolme virhettä:

- Tunnus `a` on määritelty kahdesti.
- Tunnus `b` on esitelty kahdesti ristiriitaisin tyypein.
- Tunnusta `c` ei ole määritelty lainkaan.

Nimi voidaan tehdä modulin sisäiseksi `static`-atribuutilla. Esim. koodi

```
// file1.c
static int a = 5;
static int f() { ... }

//file2.c
static int a = 6;
static int f() { ... }
```
on oikein. Kummassakin tiedostossa on oma muuttuja `a` ja oma funktio `f()`.

4.2 Otsikkotiedostot (headers)

Otsikkotiedostossa voi olla

Tyyppimäärittely	`struct point{int x,y;};`
Funktion esittely	`int strlen(const char*);`
Inline-funktion määr.	`inline void inci(){i++;}`
Datan esittely	`extern int a;`
Vakiomäärittely	`const float pi = 3.1415;`
Numerointi	`enum bool{false, true};`
Include-direktiivi	`#include <stdio.h>`
Makromäärittely	`#define begin {`
Kommentti	`/* Tähän on tultu */`

sillä tämänkaltaisten tunnusten useampikertainen määrittely on sallittu.

Otsikkotiedostossa ei pitäisi koskaan olla

Tavallinen funktiomäär.	`void inci(){i++;}`
Datamäärittely	`int a;`

4.3 Funktiot

Taulukko argumenttina

- Jos taulukon nimi esiintyy funktion argumenttina, niin funktio saa osoittimen taulukon ensimmäiseen alkioon.
- Taulukon koko ei ole käytettävissä kutsuttavassa funktiossa. Yksiulotteisten taulukoiden (vektoreiden) kanssa voidaan menetellä, kuten:
  - Merkkijonotaulukot päätetään `'\0'`:aan, joten niiden koko on helposti laskettavissa.
  - Taulukon nimen lisäksi argumenttina voisi olla taulukon koko.

Moniulotteiset taulukot

- Ilmoitetaan (ensimmäistä lukuunottamatta) dimensiot eksaktisti, esim.

```
void huuhaa( int m[][5], int dim1 )
{ ... }
```

- Annetaan argumentteina myös dimensiot, ja lasketaan itse alkion paikka, esim.

```
void huuhaa( int **m, int dim1, int dim2 )
{
  int i, j;
  // Alkio m[i][j]
  cout << ((int *)m)[i*dim2 + j] << "\n";
}
int a[50][10];
huuhaa( a, 50, 10 );
```

  Huom. Määrittely `m[][]` on laiton.

- Useimmiten (ks. *Numerical Recipes in C*) kannattaa käyttää osoitintaulukoita, esim.

```
char *day[] = {
  "mon", "tue", "wed", "thu",
  "fri", "sat", "sun" };
```

- C++:ssa voi määritellä uusia tietotyyppejä, esim.

```
struct iarray2 {
  int dim1, dim2;
  int **a;
  iarray2( int d1, int d2 );
  int* operator[] (int i)
    return a[i];
};
iarray2::iarray2( int d1, int d2 )
{
  dim1 = d1; dim2 = d2;
  a = new int*[d1];
  for( int i = 0; i < d1; i++ )
    a[i] = new int[d2];
}
```

Komentoriviargumentit

`main()`-funktion prototyyppi on

```
int main( int argc, char *argv[] );
```

- Jos ohjelma on käynnistetty komennolla

  ohjelma arg_1 arg_2 ... arg_n

  niin
  - `argc` on komentorivillä olevien sanojen lukumäärä $(n+1)$.
  - `argv[i]` osoittaa (`i+1`):nnen sanan alkuun.

```
// Kaiutus
int main( int cnt, char *words[] )
{
  cout << words[1] << "\n";
  return 0;
}
```

- `main()` palauttaa käyttöjärjestelmälle statuskoodin: 0 on OK, $\neq 0$ tarkoittaa virhettä.

Oletusargumentit (C++)

Funktion argumenteille voidaan antaa oletusarvot. Esim.

```
void huuhaa( int i, int j = 10 );
void f()
{
  huuhaa( 5 );   // huuhaa( 5, 10 )
  huuhaa( 5, 10 );
  huuhaa( 5, 6);
}
```

Oletusarvoja voi olla vain loppupään argumenteilla, esim.

```
void huuhaa( int i = 0, int j );
```

on laiton.

Yksilöimättömät argumentit

Joillakin funktioilla voi olla argumentteja, joiden lukumäärää ja tyyppiä ei yksilöidä. Esim. funktiolla `printf()` kirjoitetaan näyttöön muotoiltua tulostusta, kuten

```
printf( "Laskun %d + %d tulos on %d\n",
        i, j, i + j );
```

Tämän funktion prototyyppi on

```
int printf( const char*, ... );
```

Tiedostossa `stdarg.h` on määritelty apuneuvoja tälläisen argumenttilistan käsittelyyn (`va_list, va_start, va_arg, va_end`).

Osoittimet funktioihin

Funktiota voidaan kutsua myös asettamalla osoitin ko. funktioon. Esim.

```
void huuhaa( char *str ) {...}
void (*fptr)( char * );
void f()
{
  fptr = &huuhaa;
  (*fptr)( "Heippa!" );
}
```

- Funktio-osoittimille määritellään argumenttityypit täsmälleen samoin kuin funktioillekin.
- Funktion osoitteenotossa `&`-operaattori on tarpeeton, esim. `fptr = huuhaa;` on ok. Samoin osoitinta ei kutsussa tarvitse dereferenssoida, esim. `fptr( "Heippa!" )` riittää.
- Sijoitettaessa arvoja funktio-osoittimille täytyy argumenttien ja funktion tyyppien olla täysin yhteensopivia.

```
void (*pf)( char* );
void f1( char* );
int f2( char* );
void f3( int* );
void f()
{
  pf = &f1;  // ok
  pf = &f2;  // väärä paluutyyppi
  pf = &f3;  // väärä argumenttityyppi

  (*pf)( "qwerty" );  // ok
  (*pf)( 1 );        // väärä argumentti
}
```

Funktio-osoittimia määriteltäessä kannattaa usein käyttää `typedef`-mekanismia:

```
typedef void (*PF)( char * );
PF pf;
```

Funktio-osoitintaulukot

Ohjelman logiikka on joskus helpompi toteuttaa osoitintaulukoilla kuin esim. `switch`-rakenteella. Esim.

```
typedef void (*PF)( char * );

void do_this( char* );
void do_that( char* );
void so_on( char* );

PF do_it[] = {
  &do_this, &do_that, &so_on
};

void dispatch( char *str, int code )
{
  (*do_it[code])( str );
}
```

Käyttämällä funktio-osoittimia funktion argumentteina voidaan kirjoittaa monimuotoisia, yleiskäyttöisiä, funktioita. Esim.

```
// Kuplalajittelu
typedef int (*CFT)( void*, void* );
void sort(void *base, unsigned n,
          unsigned sz, CFT cmp )
{
  for( int i = 0; i < n - 1; i++ )
    for( int j = n - 1; i < j; j-- ) {
      char *pj = (char *)base + j*sz; // b[j]
      char *pj1 = pj - sz;           // b[j-1]
      if( (*cmp)( pj, pj1 ) < 0 ) {
        // vaihda b[j] ja b[j-1]
        for( int k = 0; k < sz; k++ ) {
          char temp = pj[k];
          pj[k] = pj1[k];
          pj1[k] = temp;
        }
      }
    }
}
```

4.4 Makrot

Makromäärittely on muotoa

- `#define` *nimi teksti*
  - Määrittelyä seuraavassa ohjelmatekstissä jokainen esiintymä *nimi* korvataan merkkijonolla *teksti*.
  - Makroteksti *teksti* käsittää koko loogisen rivin lopun.
  - Merkki `\` rivin lopussa ilmoittaa, että looginen rivi jatkuu seuraavalla fyysisellä rivillä.
- `#define` $nimi(p_1, p_2, \ldots, p_n)$ *teksti*
  - Määrittelyä seuraavassa ohjelmatekstissä jokainen esiintymä $nimi(a_1, a_2, \ldots, a_n)$ korvataan *teksti*llä siten, että jonossa *teksti* esiintyvät jonot p_i korvataan todellisilla parametreilla a_i.

Esim.

```
#define PI 3.14159  /* Vakio pi */
#define swap(a, b) {int z = a; a = b; b = z;}
```
Koodi
```
p = PI;
swap( x, y );
```
on täsmälleen sama kuin
```
p = 3.14159  /* Vakio pi */;
{int z = x; x = y; y = z;};
```

- *Esikääntäjä* (*preprosessor*) käsittelee makrot ennen käännöstä; kääntäjä ei näe alkuperäisiä makromäärittelyjä eikä -kutsuja.
- Vakioiden määrittelyssä kannattaa useimmiten käyttää `const`-määrittelyä.
- C++:ssa argumentilliset makrot voidaan useimmiten määritellä `inline`-funktioina.

Ehdollinen käännös
Rakenteella
if-rivi
käännetään, jos *if-rivi*n ehto on tosi; optionaalinen
elif-rivi; optionaalinen
käännetään, jos *if-rivi*n ehto on epätosi ja *elif-rivi*n ehto tosi; optionaalinen
else-rivi; optionaalinen
käännetään, jos *if-rivi*n ja sitä mahdollisesti seuraavien *elif-rivi*en ehdot ovat epätosia; optionaalinen
endif-rivi
voidaan valita käännettävä teksti.
if-rivi on jokin seuraavista

- `#if` *vakiolauseke*
  Rivin ehto on tosi, jos *vakiolauseke*$\neq 0$.
- `#ifdef` *tunnus*
  Rivin ehto on tosi, jos *tunnus* on määritelty `#define`-direktiivillä.
- `#ifndef` *tunnus*
  Rivin ehto on tosi, jos *tunnus*ta ei ole määritelty.

Esim.
```
#define A 2
#if A==2     // rivin ehto on tosi
 ...         // käännetään
#endif
#define foo
#ifdef foo   // rivin ehto on tosi
 ...         // käännetään
#endif
#ifndef foo  // rivin ehto on epätosi
 ...         // ei käännetä
#endif
#undef foo   // poistaa määrittelyn
#ifndef foo  // rivin ehto on tosi
 ...         // käännetään
#endif
```

elif-rivi on muotoa
`#elif` *vakiolauseke*
Rivin ehto on tosi, jos *vakiolauseke*$\neq 0$.
else-rivi on
`#else`
endif-rivi on
`#endif`

5. Tietorakenteet

Tietorakenteet organisoivat samankaltaisten olioiden muodostaman tietojoukon. Tämä järjestys voidaan saada aikaan monin tavoin, esim.

- Keräämällä oliot taulukkoon.
- Liittämällä olioihin tieto (C:ssä osoitin) niihin relaatiossa olevista olioista.

Tietorakenteet voidaan jakaa esim. luokkiin:

- Lineaariset rakenteet: jokaiselle alkiolle (*element*) on määritelty edeltäjä (*predecessor*), seuraaja (*successor*) tai molemmat.
- Epälineaariset rakenteet: alkiolla voi olla useampia edeltäjiä tai seuraajia.

Tietorakenteet ovat useimmiten dynaamisia.

5.1 Pino (stack)

- Pino on ns. *LIFO*-rakenne (*Last In First Out*): viimeksi pinoon lisätty alkio poistetaan siitä ensimmäisenä.
- Perustoiminnot kohdistuvat pinon *huippuun* (*top*):

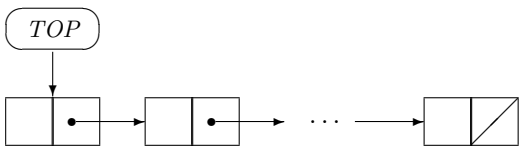
  lisäys (push) Työnnetään pinon päällimmäiseksi uusi alkio.

  poisto (pop) Poistetaan pinon päällimmäinen alkio.
- C:ssä pino voidaan toteuttaa siten, että
  - jokaisessa pinon alkiossa on osoitin, *linkki*, pinossa seuraavana olevaan alkioon,
  - pinon pohjimmaisessa alkiossa on osoitin 0 (`NULL`),
  - talletetaan päällimmäisen alkion osoite johonkin osoitinmuuttujaan, jota sekä *lisäys*- että *poisto*toiminnot käsittelevät.

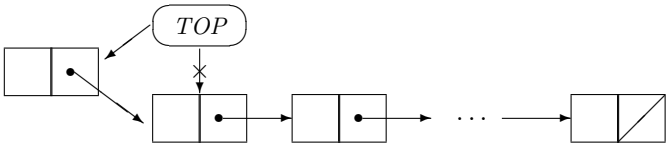
Graafinen esitys

Merkitään symbolilla  `NULL`-linkkiä.

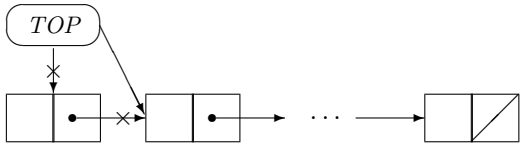
Pino



Lisäys (push)



Poisto (pop)



Koodiluonnos

Tietueet

```
struct node{
  ... // alkion data
  node *next;
};
node *TOP = NULL;
```

Lisäys

```
void push( node *newnode )
{
  newnode->next = TOP;
  TOP = newnode;
}
```

Poisto

```
node *pop()
{
  node *pt = TOP;
  if( TOP ) TOP = TOP->next;
  return pt;
}
```

5.2 Jono (queue)

- Jono on ns. *FIFO*–rakenne (*First In First Out*): ensimmäiseksi jonoon lisätty alkio poistetaan siitä ensimmäisenä.
- Perustoiminnot kohdistuvat jonon päihin:

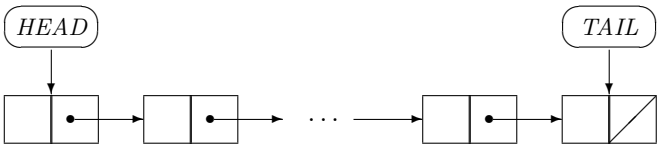
  Lisäys (enqueue) Lisätään jonon häntään (*tail*) uusi alkio.

  Poisto (dequeue) Poistetaan jonon kärjestä (*head*) alkio.

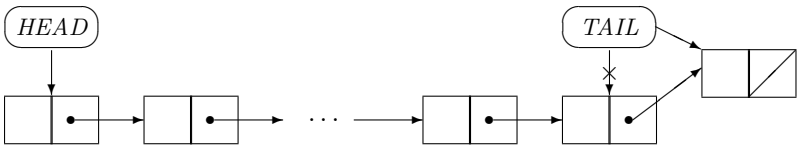
- C:ssä jono voidaan toteuttaa siten, että
  - jokainen jonon alkio on linkitetty jonon seuraavaan (järjestys kärjestä häntään) alkioon.
  - talletetaan osoittimet jonon ensimmäiseen ja viimeiseen alkioon joihinkin osoitinmuuttujiin.

Graafinen esitys

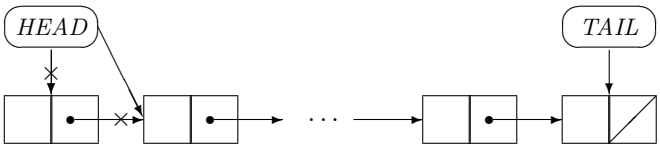
Jono



Lisäys (enqueue)



Poisto (dequeue)



Koodiluonnos

Alustus

```
node *HEAD = NULL;
node *TAIL = NULL;
```

Lisäys

```
void enqueue( node *newnode )
{ newnode->next = NULL;
  if( HEAD )          // Jono tyhjä?
    TAIL->next = newnode;
  else                // Ensimmäinen alkio
    HEAD = newnode;
  TAIL = newnode;
}
```

Poisto

```
node *dequeue()
{  node *pt = HEAD;
  if( HEAD ) HEAD = HEAD->next;
  return pt;
}
```

Jonoja käytetään usein puskureitten toteutukseen:

- Ohjelman jokin rutiini lukee tietoa, jota jokin toinen ohjelman rutiini käsittelee.
- Monesti on epäkäytännöllistä tai mahdotonta hoitaa luku ja käsittely synkronisesti, esim.
  - Ohjelma, joka tulostaa tiedoston n viimeistä riviä: lukematta tiedostoa läpi on mahdotonta tietää, milloin ollaan n:ksi viimeisellä rivillä.
  - Pääteohjelma: koneeseen ulkomaailmasta saapuvien merkkien tulohetkeä ei voida ennustaa ja merkit saattavat seurata toisiaan niin nopeasti, että pääteohjelma ei ehdi kirjoittaa niitä synkronisesti näytölle.
- Asynkronisuusongelma voidaan hoitaa siten, että
  - Tuleva tieto lisätään puskurijonon loppuun.
  - Käsittelyrutiini poimii tietoa puskurin alusta.
- Puskurit toteutetaan yleensä *rengaslistoina*: jonoina, joiden viimeinen alkio on linkitetty ensimmäiseen.
- Mikäli puskurin koko on muuttumaton, kannattaa harkita jonon toteuttamista taulukkona, esim.
  - Tiedoston n viimeistä riviä: n on ohjelman suorituksen aikana vakio, joskin vaihtelee suorituskerrasta toiseen. Sopiva tietorakenne voisi olla luettuihin riveihin osoittavien n alkion vektori.
  - Pääteohjelma: Varataan puskuriksi tarpeeksi suuri merkkitaulukko.

```
// Merkkipuskurin luonnos
static char *buff = new char[SIZE];
static int nc = 0;  // Merkkilaskuri
static int HEAD = 0;
static int TAIL = 0;
// Lisäys
int put( char c )
{ if( nc == SIZE )
    return -1;      // Puskuri täynnä
  if( TAIL == SIZE )
    TAIL = 0;       // Kierto renkaaksi
  buff[TAIL++] = c;
  return ++nc;
}
// Poisto
int get()
{ if( !nc )
    return -1;      // Puskuri tyhjä
  nc--;
  char c = buff[HEAD++];
  if( HEAD == SIZE )
    HEAD = 0;       // Kierto renkaaksi
  return c;
}
```

5.3 Kaksoislinkitetty lista

- Kaksoislinkitetyssä listassa jokaisessa alkiossa (solmussa) on tieto välittömästä edeltäjästä ja välittömästä seuraajasta.
- Perustoiminnot voivat kohdistua mihin tahansa listan alkioon:

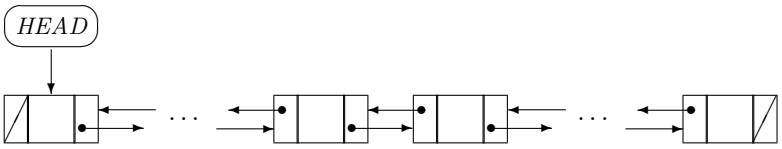
  lisäys (insert) Lisätään uusi alkio (solmu, node) jonkin alkion välittömäksi seuraajaksi (tai edeltäjäksi).

poisto (delete) Poistetaan alkio (solmu, node).

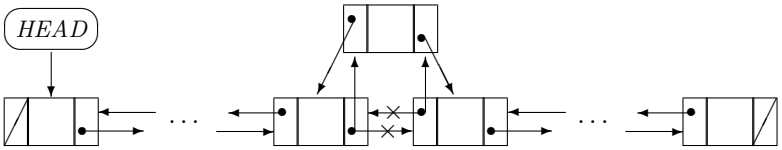
- C:ssä kaksoislinkitetty lista voidaan toteuttaa siten, että
  - jokaisessa listan alkiossa on linkki listan seuraavaan ja edelliseen alkioon.
  - talletetaan osoitin listan ensimmäiseen alkioon johonkin osoitinmuuttujaan.

Graafinen esitys

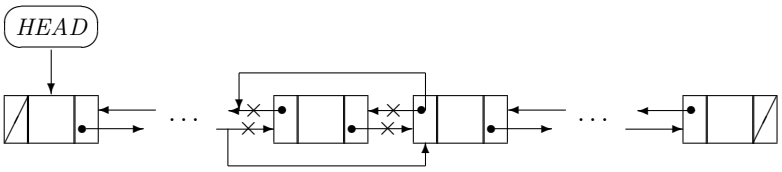
Kaksoislinkitetty lista



Lisäys (insert)



Poisto (delete)



Koodiluonnos

Tietueet

```
struct node {
  ...  // Alkion data
  node *prev;
  node *next;
};

node *HEAD = NULL;
```

Lisäys

- `void append( node *curr, node *app)` lisää alkion `app` alkion `curr` seuraajaksi.
- `void prepend( node *curr, node *prep)` lisää alkion `prep` alkion `curr` edeltäjäksi.
- Oletetaan, että `curr == NULL` tarkoittaa tyhjää listaa.

```
// Lisäys alkion seuraajaksi
void append( node *curr, node *app )
{
  if( curr ) { // Ei-tyhjä lista
    app->prev = curr;
    app->next = curr->next;
    if( curr->next )
      curr->next->prev = app;
    curr->next = app;
  }
  else {       // Tyhjä lista
    app->prev = app->next = NULL;
    HEAD = app;
  }
}
// Lisäys alkion edeltäjäksi
void prepend( node *curr, node *prep )
{
  if( curr ) {  // Ei-tyhjä lists
    prep->next = curr;
    prep->prev = curr->prev;
    if( curr->prev )
        curr->prev->next = prep;
    else        // Listan 1. alkio
        HEAD = prep;
    curr->prev = prep;
  }
  else {        // Tyhjä lista
    prep->prev = prep->next = NULL;
    HEAD = prep;
  }
}
```

Poisto (delete)

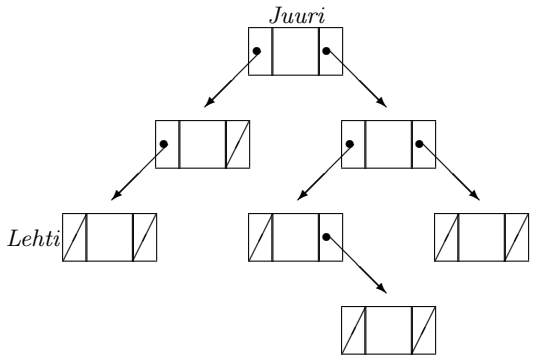
```
void remove( node *curr )
{
  if( curr->prev )
    curr->prev->next = curr->next;
  else          // Listan 1. alkio
    HEAD = curr->next;

  if( curr->next )
    curr->next->prev = curr->prev;
}
```

5.4 Binääripuu

- Binääripuun jokaisella alkiolla on joko
  - vasen tai oikea (tai molemmat) *lapsi* (*child*). Tällöin alkio on alipuun *juuri* (*root*).
  - tai ei yhtään lasta. Tälläinen alkio on nimeltään *lehti* (*leaf*).
- Binääripuun jokaisella alkiolla, lukuunottamatta koko puun juurta, on välitön edeltäjä: *vanhempi* (*parent*).
- Lehtiä lukuunottamatta jokaisella alkiolla on yksi tai kaksi välitöntä seuraajaa.
- Saman vanhemman lapsia sanotaan *sisaruksiksi* (*sibling*).
- Jos jostakin alkiosta päästään toiseen alkioon seuraamalla vanhempi-lapsi-ketjua, sanotaan, että alkioita yhdistää *polku* (*path*). Polun *pituus* on sillä olevien alkioiden lukumäärä.
- Binääripuun jokaiseen alkioon on polku puun juuresta. Alkion *korkeus* on ko. polun pituus. Binääripuun korkeus on pisimmän polun pituus.
- Polun yhdistämät alkiot ovat edeltäjä-seuraaja-suhteessa keskenään.
- Koska kahdesta alkiosta ei toisen tarvitse olla välttämättä toisen edeltäjä (tai seuraaja), binääripuu on *epälineaarinen* rakenne.

Graafinen esitys



Puun läpikäynti (traverse)

Binääripuu voidaan käydä läpi kolmella tavalla:

inorder vasen alipuu, alkio, oikea alipuu

1. Käy läpi vasen alipuu inorder-järjestyksessä.
2. Käsittele alkio.
3. Käy läpi oikea alipuu inorder-järjestyksessä.

preorder alkio, vasen alipuu, oikea alipuu

1. Käsittele alkio
2. Käy läpi vasen alipuu preorder-järjestyksessä.
3. Käy läpi oikea alipuu preorder-järjestyksessä.

postorder vasen alipuu, oikea alipuu, alkio

1. Käy läpi vasen alipuu postorder-järjestyksessä.
2. käy läpi oikea alipuu postorder-järjestyksessä.
3. Käsittele alkio

Esim. Sanojen frekvenssien lasku

- Oletetaan, että on käytettävissä funktio `int getword( char *word, int max )`, joka

  - lukee standardisyöttöä,
  - erottelee syöttövirrasta sanan kerrallaan `max` pituiseen merkkivektoriin `word` ja
  - palauttaa arvon `LETTER`, jos sana muodostuu kirjaimista.

- Kirjoitetaan funktio `tnode *tree( tnode *root, const char *word )` joka rakentaa alkiosta `root` alkavan binääripuun siten, että
  - alkion datana on luettu sana ja sen esiintymiskerta,
  - jokaisen alkion vasemmat jälkeläiset edeltävät aakkosellisesti alkion sanaa ja vastaavasti oikeat jälkeläiset seuraavat sitä.

Otsikkotiedosto

```
// wcount.h
struct tnode {
  char *word;    // Osoitin alkion sanaan
  int count;     // Esiintymien lukumäärä
  tnode *left;   // Vasen lapsi
  tnode *right;  // Oikea lapsi
};

// Vakiot
#define MAXWORD  20
#define LETTER   'A'
#define DIGIT    '1'

// Prototyypit
tnode *tree( tnode*, const char* );
void treeprint( const tnode* );
int getword( char*, int );
```

Tree

```
// tree.cpp
#include <stdlib.h>
#include <string.h>
#include "wcount.h"
tnode *tree( tnode *p, const char *w )
{
  int cmp;
  if( p == NULL ) {  // Uusi alkio
    p = new tnode;   // Rakenna uusi alkio
    p->word = new char[strlen( w ) + 1];
    strcpy( p->word, w );
    p->count = 1;
    p->left = p->right = NULL;
  }
  else if( (cmp = strcmp( w, p->word )) == 0 )
    p->count++;      // Toistuva sana
  else if( cmp < 0 ) // Laskeudu vasempaan
    p->left = tree( p->left, w );
  else               // Laskeudu oikeaan
    p->right = tree( p->right, w );
  return p;
}
```

- Kirjoitetaan funktio `void treeprint( const tnode *root )`, joka käy läpi puun inorder-järjestyksessä tulostaen sanat ja niiden frekvensit.

```
// print.cpp
#include <stdio.h>
#include "wcount.h"
void treeprint( const tnode *p )
{
  if( p ) {
    treeprint( p->left );  // Vasen alipuu
    printf( "%4d %s\n", p->count, p->word );
    treeprint( p->right ); // Oikea alipuu
  }
}
```

- Kerätään aiemmat funktiot toimivaksi ohjelmaksi.

```
// wcount.cpp
#include <stdio.h>
#include "wcount.h"
main()
{
  tnode *root = NULL;
  char word[MAXWORD];
  int t;

  while((t = getword( word, MAXWORD )) != EOF)
    if( t == LETTER )
      root = tree( root, word );

  treeprint( root );
}
```

Huomioita (binääri)puista

- Binääripuita käytetään usein tiedon järjestelyyn (lajitteluun, sorting) ja vastaavasti järjestetyn tiedon hakuun. Tällöin puhutaan ns. *hakupuista* (*search trees*).
- Mikäli binääripuun kaikkien lehtien korkeus on suunnilleen sama —puu on *tasapainossa* (*balanced*)— voidaan tietty alkio löytää tai todeta, ettei ko. tietoa ole, $\log_2 n$ toiminnolla, kun n on alkioiden lukumäärä.
- Huonoimmassa tapauksessa, kun puu muodostuu ainoastaan joko vasemman- tai oikeanpuoleisista lapsista, haku vaatii n toimintoa.
- On olemassa ns. *tasapainotettuja* (*balanced*) puurakenteita (*B-trees*), jotka takaavat, että hakuun käytetään korkeintaan $\log_t n$ toimintoa.

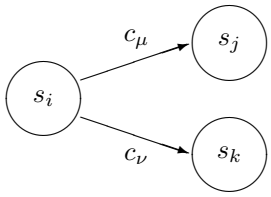
5.5 Äärelliset automaatit

Äärellinen automaatti koostuu

1. äärellisestä joukosta $S = \{s_1, s_2, \ldots, s_n\}$ tiloja s_i,
2. äärellisestä joukosta $C = \{c_1, c_2, \ldots, c_m\}$ syöttömerkkejä c_i ja

3. kuvauksesta $t : S \times C \mapsto S$.

Äärellistä automaattia voidaan pitää koneena, joka lukiessaan tilassa s_i merkin c_j siirtyy tilaan $s_k = t(s_i, c_j)$. Yleensä kone on ennen käynnistystä, ts. ennen kuin se on lukenut ainoaakaan merkkiä, tietyssä määrätyssä aloitustilassa. Koneen toiminta pysähtyy sen joutuessa johonkin ns. terminaalitilaan f, $t(f, C) = \emptyset$.
Graafisesti tiloja ja siirtymiä on tapana esittää kuten



Esim. Syöttötiedoston sanojen ja rivien lasku. Luokitellaan merkit tyyppeihin
`enum Chartype{white, newline, character, eof};`
missä

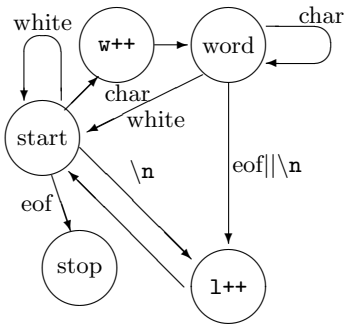
`white` on jokin *whitespace*-merkki (`' '`, `'\t'`,...),

`newline` on merkki `\n`,

`character` on mikä tahansa muu merkki ja

`eof` tarkoittaa tiedoston loppua.

Oletetaan, että on olemassa funktio `Chartype nextchar()`, joka lukee syöttötiedostoa merkin kerrallaan ja palauttaa arvonaan lukemansa merkin tyypin.
Sanojen ja rivien lasku voitaisiin hoitaa esim. äärellisellä automaatilla



Automaattia vastaava C++:n luokka voisi olla seuraavanlainen

```
class Counter {
  enum State{ start, word, stop };
  State state;
  int lines;
  int words;
public:
  void Reset() {
    state = start;
    lines = words = 0;
  }
  Counter() { Reset(); }
  bool Count( Chartype type );
  int Lines() { return lines; }
  int Words() { return words; }
};
bool Counter::Count( Chartype type )
{
  switch( state ) {
  case start:
    switch( type ) {
    case eof:
      state = stop;
      return false;
    case character:
      words++;
      state = word;
      return true;
    case newline:
      lines++;
    case white:
      return true;
    }
  case word:
    switch( type ) {
    case eof:
    case newline:
      lines++;
    case white:
      state = start;
    case character:
      return true;
    }
  case stop:
    return false;
  }
}
```

Kutsuvassa ohjelmassa olisi tällöin silmukka

```
  Counter counter;
  while( counter.Count( nextchar() ) );
```

6. Mallipohjat (templates)

6.1 Konttiluokat (container classes)

Kontit (containers) ovat luokkia, jotka

- varastoivat jotakin toista tyyppiä olevia olioita.
- toteutetaan tietorakenteina, esim. pinoina, listoina, binääripuina, jne.

Riippumatta varastoitavan olion tyypistä itse tietorakenne ja siihen liittyvät metodit ovat samoja.
Esim. kokonaislukuja sisällään pitävä pino

```
class stacki {
  int* v;
  int* p;
  int sz;
public:
  stacki( int s )
    { v = p = new int[sz = s]; }
  ~stacki() { delete[] v; }

  void push( int i ) { *p++ = i; }
  int pop() { return *--p; }
  int size() { return p - v; }
};
```

on nimeä ja oliotyyppejä lukuun ottamatta täsmälleen samanlainen kuin luokan

```
  class huu{ ... };
```

olioiden pino

```
class stackh {
  huu* v;
  huu* p;
  int sz;
public:
  stackh( int s )
    { v = p = new huu[sz = s]; }
  ~stackh() { delete[] v; }

  void push( huu i ) { *p++ = i; }
  huu pop() { return *--p; }
  int size() { return p - v; }
};
```

Koska C++ on *tyypittävä (type checking)* kieli, ei ole mahdollista kirjoittaa sellaista pinoluokkaa, johon voitaisiin tallettaa *mitä tahansa* olioita.
Eräs mahdollisuus välttyä saman koodin useampikertaiselta kirjoittamiselta on toteuttaa *geneerinen* konttiluokka siten, että talletettavina olioina ovatkin osoittimet. Esim.

```
class gstack {
  void** v;
  void** p;
  int sz;
public:
  gstack( int s )
    { v = p = new void*[sz = s]; }
  ~gstack() { delete[] v; }

  void push( void* i ) { *p++ = i; }
  void* pop() { return *--p; }
  int size() { return p - v; }
};
```

Kokonaislukujen (tai paremminkin kokonaislukuosoittimien) pino saadaan johtamalla geneerisestä luokasta, kuten

```
class istack : public gstack {
public:
  void push( int* i ) { gstack::push( i ); }
  int* pop() { return (int*)gstack::pop(); }
};
```

tai käyttämällä alkuperäistä geneeristä pinoa kuten

```
gstack intps( 100 );
int i = 5;
intps.push( &i );
int j = *(int*)intps.pop();
```

6.2 Luokkien mallipohjat

Saman koodin useampikertainen kirjoittaminen voidaan välttää myös antamalla kääntäjälle *malli* siitä, miten jotkut luokat tulisi toteuttaa. Esim.

```
template<class T>
class stack {
  T* v;
  T* p;
  int sz;
public:
  stack( int s )
    { v = p = new T[sz = s]; }
  ~stack() { delete[] v; }

  void push( T i ) { *p++ = i; }
  T pop() { return *--p; }
  int size() { return p - v; }
};
```

Tämä mallipohja *ei ole* vielä mikään luokka. Kääntäjä muodostaa, *instantioi*, luokan vasta tarvittaessa. Esim. määrittely

```
stack<int> is( 100 );
```

tai

```
typedef stack<int> istack;
```

saa kääntäjän muodostamaan luokan

```
class stack<int> {
  int* v;
  int* p;
  int sz;
public:
  stack( int s )
    { v = p = new int[sz = s]; }
  ~stack() { delete[] v; }

  void push( int i ) { *p++ = i; }
  int pop() { return *--p; }
  int size() { return p - v; }
};
```

Malliluokkien metodit voidaan kirjoittaa myös luokkien ulkopuolelle. Esim.

```
template<class T>
class stack {
```

```
  T* v;
  T* p;
  int sz;
public:
  stack( int s );
  ~stack() { delete[] v; }

  void push( T i );
  T pop();
  int size() { return p - v; }
};
template<class T>
stack<T>::stack( int s )
  { v = p = new T[sz = s]; }

template<class T>
void stack<T>::push( T i )
  { *p++ = i; }

template<class T>
T stack<T>::pop()
  { return *--p; }
```

Huom. Jotta kääntäjä osaisi toteuttaa muodostettavan luokan, on mallipohjan samoin kuin siihen liittyvien metodien mallien oltava kääntäjän näkyvissä. Siksi ne on tapana kirjoittaa `.h`-otsikkotiedostoihin.

6.3 Esimerkki: Järjestetty kaksoislinkitetty lista

Kirjoitetaan geneerinen lista, jonka alkiot on järjestetty jonkin avaimen mukaan.
Listan alkioiden kantaluokkana voisi olla luokka

```
class DoubleLink {
  friend class DoubleLinkList;
  DoubleLink* sucP;       // Seuraaja
  DoubleLink* preP;       // Edeltäjä
public:
  DoubleLink()
  { sucP = preP = 0; }
  virtual ~DoubleLink() {}
};
```

Huom. Tästä luokasta johdetaan varsinaisia dataa sisältäviä alkioita, joihin itse listassa viitataan osoittimilla. Jotta näihin alkioihin kohdistetut tuhoamistoiminnot kohdentuisivat todellisiin alkioihin, on hajotin `~DoubleLink()` määritelty virtuaaliseksi.

Itse listan kantaluokkana voisi olla luokka

```
class DoubleLinkList {
protected:
  DoubleLink* headP;   // Listan kärki
  DoubleLink* tailP;   // Listan häntä
  DoubleLink* currP;   // Nykyinen alkio
  int cnt;             // Alkioitten lukumäärä
public:
  DoubleLinkList()
  {  headP = tailP = currP = 0;
     cnt = 0;
  }
  ~DoubleLinkList();
  void append( DoubleLink* link );
  void prepend( DoubleLink* link );
  DoubleLink* remove();
  DoubleLink* first() { return currP = headP; }
  DoubleLink* last() { return currP = tailP; }
  DoubleLink* next()
  { return  currP ? currP = currP->sucP : 0; }
  DoubleLink* prev()
  { return currP ? currP = currP->preP : 0; }
};
```

Listan toiminnot kohdistuvat *nykyiseen* alkioon, johon viittaa osoitin `currP`:

- `void append( DoubleLink* link)` lisää alkion nykyisen alkion seuraajaksi.
- `void prepend( DoubleLink* link)` lisää alkion nykyisen alkion edeltäjäksi.
- `DoubleLink* remove()` poistaa listasta nykyisen alkion ja palauttaa osoittimen siihen.

Johdetaan dataa sisältävä alkio kantaluokasta `DoubleLink`. Koska tarkoitus on kirjoittaa monikäyttöinen lista, oletetaan, että

- alkion data on geneeristä tyyppiä `T`.
- data voidaan alustaa tyyppiä `C` olevalla datalla. Tätä varten täytyy olla olemassa muodostin `T( C data )`.
- alkioiden järjestelemiseen käytettävä data on tyyppiä `K`. Tällöin tarvitaan tyyppimuunnos `T::operator K()` ja vertailumetodi `int T::compare( K key )`.

Esim. Jos alkion data on tyyppiä

```
class Pair {
  char *str;
  int v;
public:
  Pair( const char* s )
  { str = new char[strlen( s ) + 1];
    strcpy( str, s ); v = 0;
  }
  ~Pair() { delete[] str; }
  int compare( const char* s )
  { return strcmp( str, s ); }
  operator const char*() { return str; }
  int& value() { return v; }
};
```

niin tyyppi `T` on `Pair`, tyyppiä `C` vastaa `const char*` ja järjestelyavaimen tyyppiä `K` vastaa myöskin tyyppi `const char*`.
Dataa sisältävän alkion malli on nyt

```
template<class T, class C, class K>
         class SList;
template<class T, class C, class K>
```

```
class KeyLink : public DoubleLink {
  friend class SList<T, C, K>;
  T data;
public:
  KeyLink( C d ) : data( d ) {}
  int compare( K key )
  { return data.compare( key );}
  K key() { return (K)data; }
};
```

Huom. Koska ystävyys ei ole periytyvä ominaisuus ja koska johdettavan listan on päästävä käsittelemään suoraan mm. alkion linkkejä, on johdetussa alkiossa määriteltävä johdettava lista eksplisiittisesti ystäväksi.

Järjestetyt listat muodostetaan mallipohjasta

```
template<class T, class C, class K>
class SList : public DoubleLinkList {
  int compare( K key );
public;
  int seek( K key );
  void insert( C d );
  int remove();
  int first():
  int last();
  int next();
  int prev();
  T& current();
};
```

- `int compare( K key )` vertaa nykyistä alkiota avaimeen `key`.
- `int seek(K key)` etsii listasta alkiota, jonka avain on `key` ja löydettyään sellaisen palauttaa arvonaan luvun 1. Jos ko. avainta ei löydy palautetaan 0 ja nykytietueeksi jää sellainen alkio, jonka seuraajaksi kyseisen avaimen omaava alkio tulisi.
- `void insert( C d )` lisää nykyisen alkion seuraajaksi alkion, jonka datana on `d`.
- `int remove()` poistaa listasta nykyalkion. Jos nykyalkiota ei ole (`currP == 0`), palautetaan 0.
- metodit `int first()`, `int last()`, `int next()` ja `int prev()` toimivat kuten kantaluokassakin paitsi, että ne palauttavat arvonaan 0:n listan loppuessa.
- `T& current()` palauttaa viittauksen nykyiseen alkioon.

Esimerkiksi metodien `int compare( K key )`, `int prev()` ja `void insert( C d )` mallit ovat

```
template<class T, class C, class K>
int
SList<T, C, K>::compare( K key )
{
  return
    ((KeyLink<T, C, K>*)currP)->compare( key );
}

template<class T, class C, class K>
int
SList<T, C, K>::prev()
{
  return DoubleLinkList::prev() != 0;
}

template<class T, class C, class K>
void
SList<T, C, K>::insert( C d )
{
  append(new KeyLink<T, C, K>( d ));
}
```

Metodin `int seek( K key )` malli voisi olla

```
template<class T, class C, class K>
int
SList<T, C, K>::seek( K key )
{
  if( !currP && !first() )
    return 0;

  int cmp = compare( key );

  if( !cmp )
    return 1;

  if( cmp > 0 ) {
    while( prev() ) {
      cmp = compare( key );
      if( !cmp )
        return 1;
      if( cmp < 0 )
        break;
    }
    return 0;
  }
  while( next() ) {
    cmp = compare( key );
    if( !cmp )
      return 1;
    if( cmp > 0 ) {
      prev();
      return 0;
    }
  }
  last();
  return 0;
}
```

Aiemmin esillä olleiden `Pair`-olioiden lista voidaan nyt tehdä yksinkertaisesti esim. `typedef`-määrittelyn avulla, kuten

```
typedef SList<Pair, const char*, const char*>
        PairList;
PairList plist;
 ...
char* p = "Heippa";
if( plist.seek( p ) )
  plist.current().value()++;
```

```
else
  plist.insert( p );
 ...
```

Huom. Jotkut kääntäjät eivät osaa implisiittisesti muodostaa `PairList` luokan käyttämiä `KeyLink<Pair, const char*, const char*>` luokkia. Ohjelmoija voi muodostaa ne eksplisiittisesti esim. määrittelyllä

```
typedef KeyLink<Pair, const char*, const char*>
        KeyPair;
```

6.4 Funktiomallit

Jäsenfunktioiden lisäksi voidaan tehdä myös globaaleja mallifunktioita. Esim. esittely
`template<class T> void swap( T& x, T&y );`
kertoo kääntäjälle, että koodissa

```
int a = 5;
int b = 6;
swap( a, b );
```

sen tulee kutsua funktiota `void swap( int&, int& )`. Jotta kääntäjä osaisi tehdä kyseisen funktion koodin, täytyy sille antaa malli

```
template<class T>
void swap( T& x, T& y )
{
  T z = x;
  x = y;
  y = z;
}
```

7. Standardikirjasto

Ohjelmankehitysympäristössä

- on joukko kirjastoituja funktioita.
- on joukko otsikkotiedostoja, joissa on kirjastofunktioiden prototyypit (esittelyt) ja tarvittavien vakioiden, tietotyyppien ja olioiden määrittelyt.
- kirjastofunktioista suuri osa on ISO-C-standardin mukaisia ja siten siirrettävissä käyttöympäristöstä toiseen.
- jotkin kirjastofunktiot ovat käyttöjärjestelmä- tai konekohtaisia.

7.1 I/O

- Käyttöjärjestelmä (ja standardikirjaston funktiot) puskuroi raa'an (raw) levy- ja konsolidatan koneen muistiin.
- Ohjelmoijan kannalta syötettävä/tulostettava data näkyy yhtenäisenä datavirtana (*stream*).
- Puskuri- ja datavirtatyypit on määritelty tiedostossa `stdio.h`. Datavirtatyyppi on `FILE`. Ohjelmoija käyttää (yleensä) osoittimia (`FILE*`) datavirtaolioihin.
- C++:ssa on käytettävissä tiedostossa `iostream.h` määriteltyjä korkeamman tason luokkia ja olioita (esim. `cin, cout, ...`).
- On myös mahdollista käsitellä I/O:ta alhaisen tason funktioilla (esim. levysektoreittainen luku/kirjoitus). Ohjelmasta tulee tällöin useinkin kone/käyttöjärjestelmäkohtainen.

Konsoli-I/O

`main()`-funktion käynnistyessä on käytettävissä kolme `FILE*`-osoitinta:

`stdin` Standardisyöttö, näppäimistö.

`stdout` Standarditulostus, näyttö.

`stderr` Standardivirhe, näyttö.

Useat kirjastofunktiot operoivat näihin datavirtoihin.

Merkki- ja merkkijono-konsoli-I/O

`int getchar()` Palauttaa standardisyötöstä lukemansa merkin. Tiedoston loppuessa palautetaan `EOF`.

`int putchar(char)` Kirjoittaa argumenttinsa standarditulostukseen.

`char *gets(char *buff)` Lukee standardisyötöstä puskuriin *buff* telanpalautukseen päättyvän merkkijonon. Telanpalautusta ei oteta mukaan. Merkkijono päätetään `'\0'`:lla. Arvona on *buff*.

`int puts(char *buff)` Kirjoittaa puskurissa *buff* '\0':lla päätetyn merkkijonon standarditulostukseen.

Formatoitu konsoli-I/O

`int printf(const char *format,...)` Formatoitu tulostus.

`int scanf(const char *format,...)` Formatoitu syöttö.

Esim.

```
#include <stdio.h>
main()
{
  float x, y;
  printf( "Kaksi lukua: " );
  scanf( "%f%f", &x, &y );
  printf( "Lukujen %f ja %f summa on %f\n",
           x, y, x + y );
}
```

Tiedostot

Tiedostojen käsittely

`FILE *fopen(const char *nimi,const char *mod);` Tiedoston *nimi* avaus. *mod* on merkeistä `r` (read), `w` (write), `a` (append) ja `+` (update) koostuva merkkijono.

`int fclose( FILE *file );` Tiedoston sulku.

`int remove( const char *nimi);` Tiedoston tuhoaminen.

`int fseek( FILE *file, long kohta, int miten );` Seuraava I/O *kohta* tavun päästä tiedoston alusta (*miten*=0), lopusta (*miten*=2) tai nykykohdasta (*miten*=1).

Tiedosto-I/O

`int fgetc( FILE *file );` Palauttaa seuraavan merkin datavirrassa *file*.

`int fputc( int c, FILE *file );` Kirjoittaa merkin *c* tiedostoon *file*.

`char *fgets( char *s, int n, FILE *file );` Lukee puskuriin *s* merkkiin '\n' päättyvän, korkeintaan $n-1$ merkkiä pitkän, merkkijonon.

`int fputs( const char *s, FILE *file );` Kirjoittaa puskurista *s* merkkijonon datavirtaan *file*.

`int fprintf(FILE *file, const char *form, ... );` Kuten `printf()`.

`int fscanf( FILE *file, const char *form, ... );` Kuten `scanf()`.

Formatoimaton (binäärinen) I/O

`long fread(void *b,long k,long n,FILE *file);` Lukee puskuriin *b* *n* kappaletta *k* tavun suuruista oliota.

`long fwrite(const void *b,long k,long n, FILE *file);` Kirjoittaa puskurista *b* *n* kappaletta *k* tavun suuruista oliota.

Formatoitu I/O merkkijonoon/jonosta

`int sprintf( char *b, const char *fm, ...);` Kuten `printf()`, mutta tulostus puskuriin *b*.

`int sscanf(const char *b,const char *fm,...);` Kuten `scanf()`, mutta luku puskurista *b*.

7.2 Merkkijonot

Prototyypit tiedostossa `string.h`.

`int strlen( const char *s );` Merkkijonon pituus.

`char *strcpy( char *dest, const char *src );` Merkkijonon kopiointi.

`char *strcat( char *dest, const char *src );` Liittää jonon *src* jonon *dest* perään.

`int strcmp( const char *s1, const char *s2 );` Merkkijonojen vertailu. Arvona < 0, jos *s1* pienempi kuin *s2*, > 0, jos *s1* suurempi kuin *s2*, ja 0, jos merkkijonot ovat samoja.

`void *memcpy(void *dst,const void *sr,long n);` Kopioi *n* tavua.

7.3 Standardirutiinit

Prototyypit tiedostossa `stdlib.h`.

`double atof( const char *s );` Muuntaa merkkijonon *s* liukuluvuksi.

`int atoi( const char *s );` Muuntaa merkkijonon *s* kokonaisluvuksi.

`char *getenv( const char *nimi );` Palauttaa osoittimen ympäristömuuttujan *nimi* arvoon. (WINDOWSissa ympäristömuuttujat asetetaan esim. `SET` *nimi=arvo* tyyppisillä komennoilla)

`int putenv( const char *jono );` Asettaa ympäristöön *jono*n. (WINDOWSissa *jono* voi olla esim. `"PATH=C:\BIN"`)

`void qsort( void *base, long n, long w,` int(*cmp)(const void*, const void*)); Quicksort-lajittelu.

7.4 Muistinhallinta

Prototyypit tiedostossa `alloc.h`.

`void *malloc( long koko );`

`void *calloc( long kpl, long koko );`

`void *realloc( void *block, long koko );`
Muuttaa aiemmin varatun muistiblokin *block* kokoa.

`void free( void *block );`

7.5 Prosessikontrolli

Prototyypit tiedostossa `process.h`.

`void exit( int status );` Lopettaa ohjelman ja palauttaa arvon *status* kutsuvalle prosessille. Tiedostot suljetaan ennen ohjelman lopetusta.

`void abort();` Hätälopetus.

`int execl( char *path, char *arg0, char *arg1,..., char *argN, NULL );` Lataa kutsuvan prosessin päälle ja suorittaa ohjelman *path*, jonka komentoriviargumenteiksi tulevat merkkijonot *arg0* (yleensä merkkijonon *path* kopio), *arg1*,...,*argN*. Mikäli kutsu onnistuu, `execl()` ei koskaan palaa. Tästä funktiosta on lukuisia argumenttien esitystapojen suhteen poikkeavia muunnelmia.

Lapsiprosessit (child processes)

`int spawnl( char *path, char *arg0, char *arg1,..., char *argN, NULL );` Kuten `execl()`, mutta ohjelmaa ei ladata kutsuvan prosessin päälle. `spawnl()` palauttaa suoritetun ohjelman status-koodin. Tästä funktiosta on myös lukuisia muunnelmia.

`int fork();` Luo uuden prosessin, joka on oleellisesti kutsuvan prosessin kopio. Ohjelmoijan kannalta `fork()` palaa *kahdesti*: arvolla 0 luotuun lapsiprosessiin ja ns. lapsiprosessi-ID:llä kutsuvaan prosessiin. Lapsiprosessi voi esim. kutsua myöhemmin funktiota `execl()`.

UNIXissa, samoin kuin monissa muissa ns. moniprosessikäyttöjärjestelmissä `fork()` on tärkein uusien prosessien luontimekanismi. Esim. UNIXissa jokainen prosessi lukuunottamatta prosessia 1 (`init`) luodaan `fork()`:lla.
Esim.

```
main()
{
  int child_ID, stat;
  if( !(child_ID = fork()) ) {
     cout << "Lapsiprosessissa\n";
     sleep( 20 ); // Nukutetaan lasta 20s
                  // tuutilullaa
     cout << "Lopetetaan lapsiprosessi\n";
     exit( 0 );
  }
  cout << "Luotiin lapsiprosessi " << child_ID
       << "\n";
  wait( &stat );  // Odota lapsiprosessin
                  // päättymistä
  cout << "Lapsiprosessi päättyi\n";
}
```

7.6 Kontit

Standardikirjaston kontit on määritelty nimiavaruudessa `std`. Ne voidaan jakaa kahteen ryhmään:

- sekventiaaliset kontit, esim. vektorit.
- assosiatiiviset kontit, joissa olioihin voidaan viitata *avaimella*.

Kontteihin talletetaan objektien kopioita. Siksi objekteille täytyy olla määriteltynä

- sijoitusoperaattori, esim.

  ```
  X& X::operator=( const X& x )
  { ...  return *this; }
  ```

- kopiomuodostin, esim.

  ```
  X::X( const X& x ) { ... }
  ```

Jokainen kontti määrittelee joukon tyyppinimiä, mm.

```
template<class T, ....>
class Cont {
public:
  typedef T value_type;
  typedef ... reference;
  typedef ... const_reference;
  typedef ... size_type;
  typedef ... iterator;
  typedef ... const_iterator;
  typedef ... reverse_iterator;
  typedef ... const_reverse_iterator;
  ...
};
```

Tässä

- `value_type` on konttiin talletettavan tiedon tyyppi.
- `reference` ja `const_reference` käyttäytyvät kuten tyypit `T&` ja `const T&`.
- `size_type` on lukumäärälaskuriksi sopiva kokonaislukutyyppi.
- `iterator` ja `const_iterator` käyttäytyvät kuten `T*` ja `const T*`. Näiden tyyppisille objekteille on määritelty mm. inkrementointioperaattori `++`.
- `reverse_iterator` ja `const_reverse_iterator` tyypit käyttäytyvät kuten `iterator` ja `const_iterator`, mutta niiden tyyppiset objektit tottelevat dekrementointioperaatiota `--`.

Tyyppimäärittelyjen avulla on mahdollista kirjoittaa geneerisiä ohjelmia, esim.

```
template<class C>
typename C::value_type sum( const C& c )
{
  typename C::value_type s = 0;
  typename C::const_iterator p = c.begin();
  while( p != c.end() ) {
    s += *p;
    ++p;
  }
  return s;
}
```

Huom. Avainsana `typename` kertoo kääntäjälle, että seuraava tunnus tarkoittaa tyyppiä. Normaalisti resoluutio-operaattorilla viitataan luokan jäseniin: `X::c` tarkoittaa luokan `X` jäsentä `c`.

Jokainen kontti määrittelee *iteraattorit*

```
template<class T, ...>
class Cont{
public:
  ...
  iterator begin();
  const_iterator begin() const;
  iterator end();
  const_iterator end() const;
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;
  ...
};
```

Metodin

- `begin()` palauttama olio viittaa kontin ensimmäiseen alkioon.
- `end()` palauttaman olion voidaan ajatella viittaavan viimeistä alkiota seuraavaan alkioon.
- `rbegin()` palauttama olio viittaa viimeiseen alkioon.
- `rend()` palauttama olio viittaa ensimmäistä edeltävään alkioon.

Jokaisessa kontissa on myös määritelty metodi `size_type size()`, joka kertoo konttiin talletettujen olioiden lukumäärän.

Vector (<vector>)

Standardikirjaston määrittelemässä vektorissa `vector` voidaan alkioita käsitellä mielivaltaisessa järjestyksessä:

```
template <class T>
class vector {
public:
  ...
  reference operator[]( size_type n );
  const_reference
    operator[]( size_type n ) const;
  refrence at( size_type n );
  const_reference at( size_type n ) const;
  reference front();
  const_reference front() const;
  reference back();
  const_reference back() const;
  ...
};
```

Kontin `vector`

- indeksointioperaattorit `operator[]` eivät tarkista indeksiä. Käyttäjän on huolehdittava siitä, että indeksi pysyy sallituissa rajoissa, ts. välillä 0 – alkioiden lukumäärä - 1.
- metodit `at()` tarkistavat indeksin laillisuuden. Mikäli indeksi on laiton, ne heittävät `out_of_range`-poikkeuksen (ks. Poikkeukset).
- indeksointioperaattorit ovat nopeampia kuin `at()`-metodit.
- metodit `front()` ja `back()` palauttavat referenssin vektorin ensimmäiseen ja viimeiseen alkioon.
- aksessointioperaatioiden `[]` ja `at()` argumentti on kokonaislukutyyppiä `size_type`, kun taas mm. metodi `begin()` palauttaa jonkin iteraattorityypin. Siispä lausekkeen `c[c.begin()]` kaltaiset ilmaisut ovat laittomia.

Luokka `vector` määrittelee mm. muodostimet

```
template <class T> class vector{
public:
  ...
  vector();
  vector( size_type n );
  vector( const vector& v );
  ...
};
```

Muodostin

- `vector()` luo vektorin, jonka koko riippuu implementoinnista.
- `vector( size_type n)` luo `n` alkioisen vektorin.
- `vector( const vector&v)` luo kopion vektorista `v`.

Luokan `vector` objekteja voidaan käyttää myös pinon tavoin:

```
template <class T> class vector{
public:
  ...
  void push_back( const T& x );
  void pop_back();
  ...
};
```

Metodi

- `push_back()` lisää alkion vektorin loppuun vektorin koon kasvaessa yhdellä.
- `pop_back()` poistaa vektorin viimeisen alkion, joten vektorin koko pienenee yhdellä.

Alkioita on myös mahdollista lisätä ja poistaa vektorin keskeltä:

```
template <class T> class vector{
public:
  ...
  iterator insert( iterator p, const T& x );
  void insert( iterator p, size_type n,
               const T& x );
  iterator erase( iterator p );
  iterator erase( iterator first,
                  iterator last );
  void clear();
  ...
};
```
Metodit

- `insert( iterator p, ... )` lisäävät alkioita iteraattorin `p` osoittaman alkion eteen.
- `erase( iterator ... )` poistavat iteraattorin viittaaman alkion tai kaikki iteraattorien väliin jäävät alkiot.

Metodi `clear()` tyhjentää koko vektorin.

List (<list>)

Lisäysten ja poistojen kannalta listarakenne on optimaalinen (ks. Kaksoislinkitetty lista). Tälläisen rakenteen toteuttaa standardikirjaston `list`-kontti. Se tarjoaa kaikki edellä mainitut `vector`-kontin metodit lukuunottamatta indeksointioperaatioita, joita ei ole mahdollista implementoida tehokkaiksi.
Listojen manipulointiin on tarjolla metodit
```
template <class T> class list{
public:
  ...
  void splice( iterator pos, list& x );
  void splice( iterator pos, list& x,
               iterator p );
  void splice( iterator pos, list& x,
               iterator first, iterator last );
  void merge( list& x );
  void sort();
  ...
};
```
Operaatio

- `splice( iterator pos, list&x )` siirtää listan `x` alkiot iteraattorin `pos` osoittaman alkion edeltäjiksi.
- `splice( iterator pos, list&x, iterator p )` siirtään listasta `x` iteraattorin `p` osoittaman alkion iteraattorin `pos` osoittaman alkion eteen.
- `splice( iterator pos, list&x, iterator first, iterator last )` siirtää listasta `x` iteraattorien `first` ja `last` osoittamien alkioiden väliin jäävät alkiot (`*first` mukaanlukien) kohdan `pos` eteen.
- `merge()` yhdistää kaksi *järjestettyä* listaa.
- `sort()` järjestää listan.

Jotta metodit `sort()` ja `merge()` toimisivat, täytyy alkioiden olla vertailtavissa. Vertailumetodina käytetään oletuksena operaattoria `<`. Käyttäjä voi myös määritellä jonkin muun vertailumenetelmän.
Erikoisesti listan kärjen käsittelyyn on metodit
```
template <class T> class list{
public:
  ...
  reference front();
  const_reference front() const;
  void push_front( const T& x );
  void pop_front();
  ...
};
```
Nämä toimivat vastaavien `back`-operaatioiden tavoin, mutta kohdistuvat listan kärkeen.

Deque (<deque>)

Luokka
```
template <class T> class deque{
public:
   // tyypit
   // vektorioperaatiot
   // front-operaatiot
};
```
on

- optimoitu silmälläpitäen ensimmäisen ja viimeisen alkion lisäystä ja poistoa.
- indeksoinnin suhteen lähes yhtä tehokas kuin vektori.
- vektorin lailla tehoton lisättäessä ja poistettaessa alkioita rakenteen keskeltä.

Stack (<stack>)

Pinorakenne `stack` on yleensä toteutettu adaptoimalla joko vektori tai `deque`-rakenne, kuten
```
template <class T> class stack{
protected:
  deque<T> c;
public:
  ...
  value_type& top() { return c.back(); }
  void push( const value_type& x )
    { c.push_back( x ); }
  void pop() { c.pop_back(); }
};
```

Queue (<queue>)

Jonorakenne on toteutettu pinorakenteen tavoin adaptoimalla, kuten
```
template <class T> class queue{
protected:
  deque<T> c;
public:
  ...
  value_type& front() { return c.front(); }
  value_type& back() { return c.back(); }
```

```
  void push( const value_type& x )
    { c.push_back( x ); }
  void pop() { c.pop_front(); }
};
```

Map (<map>)

Assosiatiiviseen rakenteeseen `map` talletetaan *avain-arvo*-pareja. Siksi siinä on uusia typpimäärittelyjä ja muutamat entiset määrittely poikkeavat edellä esitetystä, mm.

```
template <class Key, class T, class Cmp>
class map{
public:
  typedef Key key_type;
  typedef T mapped_type;
  typedef pair<const Key, T> value_type;
  typedef Cmp key_compare;
  ...
};
```

Tässä mallipohjan parametri `Cmp` on avainten vertailussa käytettävän olion tyyppi. Vertailtaessa kutsutaan ko. objektin funktiokutsumetodia `bool operator()( const Key& k1, const Key& k2)`.

Usein vertailuobjektiksi soveltuu standardikirjaston tiedostossa `<functional>` suurinpiirtein seuraavasti määritellyn mallipohjan

```
template<class T> class less{
public:
  bool operator()( const T& t1, const T& t2 )
    { return t1 < t2; }
};
```

mukaan instantioitu tyyppi `less<Key>`.

Avaimet ja arvot talletetaan pareina

```
template <class T1, class T2> class pair {
public:
  typedef T1 first_type;
  typedef T2 second_type;
  T1 first;
  T2 second;
  pair();
  pair( const T1& x, const T2& y );
};
```

Indeksointi tapahtuu avaimen perusteella

```
template <class Key, class T, class Cmp>
class map{
public:
  ...
  mapped_type& operator[]( const key_type& k );
  ...
};
```

Mikäli avainta ei löydy, talletetaan rakenteeseen uusi alkio ja palautetaan referenssi tähän.

Indeksointioperaattorin lisäksi `map`-rakenteesta voidaan etsiä avaimia mm. metodeilla

```
template <class Key, class T, class Cmp>
class map{
public:
  ...
  iterator find( const key_type& k );
  size_type count( const key_type& k );
  iterator lower_bound( const key_type& k );
  iterator upper_bound( const key_type& k );
  ...
};
```

Funktio

- `find( k )` palauttaa iteraattorin pariin, jonka ensimmäisenä jäsenenä on avain `k`. Mikäli tätä avainta ei löydy, palautetaan iteraattori `end()`.
- `count( k )` laskee niiden elementtien lukumäärän, joiden avaimena on `k`.
- `lower_bound( k )` palauttaa iteraattorin ensimmäiseen pariin, jonka avaimena on `k`.
- `upper_bound( k )` palauttaa iteraattorin ensimmäiseen pariin, jonka avain on suurempi kuin `k`.

Rakenteessa `map` voi kukin avain esiintyä vain yhden kerran. Tämän variaatio `multimap` sallii saman avaimen useamman kertaisen talletuksen.

Set (<set>)

Joukko `set` eroaa rakenteesta `map` ainoastaan siinä, että siihen talletetaan vain avaimia, joihin ei liity mitään arvoja:

```
template <class Key, class Cmp>
class set{
public:
  // Kuten map, paitsi että
  typedef Key value_type;
  typedef Cmp value_compare;
  // ei indeksointioperaattoria []
};
```

Myös tästä kontista on variaatio `multiset`, johon saman avaimen voi tallettaa useamman kerran.

Merkkijonot (<string>)

Merkkijonojen tallennukseen on kontti

```
template <class Ch>
class basic_string{
public:
  // tyypit kuten muissakin konteissa
  ...
};
```

Parametri `Ch` ilmoittaa merkkien esitykseen käytettävän tyypin. Normaali merkkijono on toteutettu määrittelyllä

```
typedef basic_string<char> string;
```

Merkkijono voidaan alustaa merkkitaulukoilla ja C:n merkkijonoilla muodostimia

```
template <class Ch>
class basic_string{
public:
  ...
  basic_string( const Ch* p, size_type n );
  basic_string( const Ch* p );
  ...
};
```

käyttäen.
Merkkijonoille on määritelty sijoitusoperaatiot

```
template <class Ch>
class basic_string{
public:
  ...
  basic_string& operator=( const basic_string& );
  basic_string& operator=( const Ch* p );
  basic_string& operator=( Ch c );
  basic_string& assign( const basic_string& );
  basic_string& assign( const Ch* s,
                        size_type n );
  basic_string& assign( const Ch* s );
  ...
};
```

Merkkijonoja voidaan verrata keskenään ja C:n merkkijonojen kanssa mm. metodeilla

```
template <class Ch>
class basic_string{
public:
  ...
  int compare( const basic_string& );
  int compare( const Ch* s );
  ..
};
```

Merkkijonojoihin voidaan lisätä ja niistä voidaan poistaa sekä merkkejä että merkkijonoja. Näiden lisäksi operaattori `+` on ylikuormattu liittämään merkkijonoja tosiinsa. Merkkijonoista voidaan etsiä merkkejä ja alimerkkijonoja.

8. Luokat (C++)

- Luokat ovat tietueita, joiden jäsenet ovat ilman eri määrittelyä yksityisiä, ts. vain ko. luokan ilmentymät (oliot) voivat viitata niihin. `struct X { ... }` on sama kuin `class X{ public: ... }`.
- Luokkien (ja tietueiden) jäseninä voi olla funktioita (metodeja).
- Luokissa, tai paremminkin niiden ilmentymissä (olioissa), on siis pakattu yhteen data ja dataa käsittelevät metodit.

8.1 Jäsenfunktiot

Jäsenfunktio voi viitata suoraan sen olion jäseniin, jonka jäsenenä funktiota kutsutaan:

```
class X {
  int m;
public:
  int givem() { return m; }
};

void f( X aa, X bb )
{
  int a = aa.givem();
  int b = bb.givem();
  ...
}
```

Ensimmäinen metodin `givem()` kutsu viittaa jäseneen `aa.m` ja jälkimmäinen jäseneen `bb.m`.

- Tosiasiassa jäsenfunktioita ei luoda erikseen jokaiselle luokan oliolle: jokaisesta jäsenfunktiosta on olemassa vain yksi ainoa kopio. Sen sijaan jokaisesta datajäsenestä tulee jokaiseen olioon oma kopio.
- Jotta luokan `X` jäsenfunktio tietäisi, minkä olion jäsenenä sitä on kutsuttu, välittyy sille näkymätön argumentti `X* const this`.
- `this` on alustettu osoittamaan siihen luokan `X` olioon, jonka jäsenenä ko. funktiota kutsutaan.
- Muuttuja `this` on myös ohjelmoijan käytettävissä: yo. esimerkin metodi `givem()` voidaan yhtenevästi toteuttaa kuten: `int givem() { return this->m; }`.
- Viitattaessa olion jäseniin muuttujan `this` käyttö on kuitenkin tarpeetonta.

Pääasiallisesti muuttujaa `this` tarvitaan kirjoitettaessa sellaisia jäsenfunktioita, jotka käsittelevät suoraan osoittimia. Esim. funktio, joka lisää alkion kaksoislinkitettyyn listaan:

```
class dlink {
  dlink *prev;
  dlink *next;
  ...
```

```
public:
  void append( dlink* );
  ...
};
void dlink::append( dlink *p )
{                   //  Sama kuin
  p->next = next;   // p->next = this->next
  p->prev = this;
  next->prev = p;   // this->next->prev = p
  next = p;         // this->next = p
}
dlink *head;

void f( dlink *a, dlink *b )
{
  ...
  head->append( a );
  head->append( b );
  ...
}
```

Huom. Suojauksen yksikkö on *luokka*, ei yksityinen luokan olio. Siksi jäsenfunktio `dlink::append()` voi viitata saman luokan eri olioiden yksityisiin jäseniin kuten esim. lauseessa `p->prev = this;`.

8.2 Muodostimet (constructors)

Luokan olio alustetaan *muodostin*funktiolla määrittelyn yhteydessä.

- Muodostimella on sama nimi kuin luokalla.
- Jos ohjelmoija määrittelee muodostimen/t, kaikki luokan oliot alustetaan tällä/näillä muodostime(i)lla.
- Jos ohjelmoija ei määrittele muodostinta, tekee kääntäjä oletusmuodostimen.

```
class date {
  ...
  date( int, int, int );
};
date today(27,2,2006);           // OK
date tomorrow = date(28,2,2006); // OK
date yesterday;                  // Virhe!
```

Usein on hyödyllistä kirjoittaa useita muodostimia.

```
class date {
  int day, month, year;
public:
  ...
  date(int, int, int);  // pp, kk, vv
  date(int, int);       // pp, kk, tämä vv
  date(int);            // pp, tämä kk ja vv
  date();               // Oletus: tänään
  date(const char *);   // merkkijono
};

date today(27);
date xmas("25 jouluk. 2002");
date now;
```

Olio voidaan alustaa myös sijoittamalla siihen jokin toinen luokan olio:

```
date d = today;  // Bitittäinen kopio
```

Luokkaoliot jäseninä

Tarkastellaan luokkia

```
class table {
  int no_of_elems;
  int *els;
  ...
public:
  table( int sz )
    { els = new int[no_of_elems = sz]; }
  ...
};
class classdef {
  table members;
  int no_of_members;
  ...
public:
  classdef( int sz );
  ...
};
```

Muodostin `classdef(int)` on kirjoitettava siten, että luokkajäsen alustetaan muodostimella `table(int)`:

```
classdef::classdef( int sz ) :
  members( sz )
{
  no_of_members = sz;
  ...
}
```

- Luokan jäsenten muodostimien kutsut kirjoitetaan luokan muodostimen *määritelmään* (ei esittelyyn) ennen sen runkoa ja erotetaan kaksoispisteellä (`:`) varsinaisesta otsikko-osasta.
- Jos useampia jäseniä joudutaan alustamaan eksplisiittisillä muodostimilla, kirjoitetaan näiden kutsut peräkkäin pilkulla toisistaan erotettuna.
- Jäsenten muodostimet suoritetaan luokkamäärittelyssä esiintyvässä järjestyksessä ja ennen luokan muodostimen runkoa.

```
class classdef {
  table members;
  table friends;
  int no_of_members;
  ...
public:
  classdef( int );
  ...
};
classdef::classdef( int sz ) :
  friends( sz ), no_of_members( sz ),
  members( sz )
{
  ...
}
```

8.3 Hajottimet (destructors)

Ohjelman suorituksen siirtyessä pois muuttujan vaikutusalueelta tai hävitettäessä muuttuja `delete`-operaattorilla tuhoamiseen sovelletaan argumentitonta *hajotin*funktiota.

- Luokan `X` hajotin on `~X()`.
- Mikäli ohjelmoija ei kirjoita hajotinta, tekee kääntäjä automaattisesti oletushajottimen. Ohjelmoijan on yleensä tarpeen antaa eksplisiittinen hajotin esimerkiksi silloin kun
  - luokan oliot varaavat dynaamista muistia. Hajottimen tehtävänä on vapauttaa tämä muisti.
  - luokan oliot puskuroivat levylle kirjoitettavaa tietoa. Olioiden tuhoutuessa on tämä tieto kirjoitettava levylle (puskureiden *huuhtominen, flushing*).

```
class ch_stack {  // Merkkipino
  int size;
  char *top;
  char *s;
public:
  ch_stack(int sz)
    {top = s = new char[size = sz];}
  ~ch_stack() {delete[] s;}
  void push(char c) {*top++ = c;}
  char pop() {return *--top;}
};
void f()
{
  ch_stack s1(100);
  ch_stack s2(200);
  s1.push('a');
  s2.push(s1.pop());
  cout << s2.pop() << '\n';
}  // s1 ja s2 tuhotaan ~ch_stack():lla
```

8.4 Ystävät (friends)

Oletetaan, että

- on määritelty luokat `matrix` ja `vector`.
- näille on määritelty täydellinen joukko jäsenfunktioita, mm.
  - funktiot `elem()`, joilla viitataan matriisin ja vektorin alkioihin.
  - funktiot `rows()` ja `cols()`, jotka antavat olioiden dimensiot.
- luokkien eksplisiittinen dataesitys on kätketty (`private`).

Tehtävänä on kirjoittaa funktio, joka kertoo vektorin matriisilla.

Ratkaisu 1.

```
vector mult(const matrix& m, const vector& v)
{
  int d = m.rows();
  vector r(d);
  for(int i = 0; i < d; i++) {
    r.elem(i) = 0;
    for(int j = 0; j < m.cols(); j++)
       r.elem(i) += m.elem(i,j)*v.elem(j);
  }
  return r;
}
```

Koska funktioita `elem()` kutsutaan `m.rows()*(1+3*m.cols())` kertaa, tämä ratkaisu on erittäin tehoton varsinkin, jos nämä tarkastavat indeksien arvot.

Ratkaisu 2.

Tehokkaampi ratkaisu saadaan, jos funktio `mult()` voi viitata suoraan olioiden jäseniin.

- Funktion `mult()` tulisi olla sekä luokan `matrix` että luokan `vector` jäsen. Tämä on kiellettyä: C++ jäsen voi kuulua vain yhteen luokkaan.
- Poistetaan `private`-suojaukset luokkien datalta. Tällöin muutkin funktiot kuin `mult()` voisivat käsitellä suoraan olioiden dataa mahdollisesti vahingollisin seurauksin.
- Ei-jäsenfunktio voi olla luokan *ystävä* (*friend*). Ystäväfunktiot voivat viitata luokan yksityisiin jäseniin.

```
class matrix;
class vector {
  float *v;
  int col;
  ...
  friend vector mult(const matrix&,
                     const vector&);
};
class matrix {
  vector *v;
  int row, col;
  ...
  friend vector mult(const matrix&,
                     const vector&);
};
vector mult(const matrix& m, const vector& v )
{ vector r(m.row);
  for( int i = 0; i < m.row; i++ ) {
    r.v[i] = 0;
    for( int j = 0; j < m.col; j++ )
      r.v[i] += m.v[i][j]*v.v[j];
  }
  return r;
}
```

Luokan jäsenfunktio voi olla jonkin toisen luokan ystävä:

```
class x {
  ...
  void f();
```

```
};
class y {
  ...
  friend void x::f();
};
```
Jos luokan (y) *jokainen* jäsenfunktio on jonkin toisen luokan (x) ystävä, voidaan tämä ilmoittaa lyhennysmerkinnällä:
```
class x {
  ...
  friend class y;
};
```

8.5 Staattiset jäsenet

- Normaalisti luokan jokaisella oliolla on omat kopiot luokan datajäsenistä.
- Joskus saattaa olla mielekkäämpää jakaa jokin datajäsen kaikkien olioiden kesken. Esim. ikkunointiohjelmassa
  - jokainen ikkuna voisi olla luokan `window` olio.
  - jokaisen ikkunan on tiedettävä jotain näyttöruudusta, joten
  - näyttöruutua vastaavan luokan (`screen`) olio on `window`-luokan luonnollinen jäsen.
  - näyttöruutu on jokaiselle ikkunalle sama.
- `static`-atribuutilla esitellyt luokan jäsenet ovat yhteisiä kaikille luokan olioille: niitä on ohjelmassa täsmälleen yksi kappale.

```
class window {
  ...
  static screen scr;
};
```
Jos julkisiin (`public`) jäseniin halutaan viitata luokan ulkopuolelta, on käytettävä luokkaerottelu-operaattoria (`::`):
```
int pxls = window::scr.hor_size();
```
Jäsenen esittely staattiseksi on tosiaankin vain esittely. Itse olio on määriteltävä jossakin kohtaa ohjelmassa:
```
screen window::scr(800,600);
```

8.6 Osoittimet jäseniin

On mahdollista ottaa luokan jäsenen osoite.

- Jos `X` on luokka ja `y` on sen jäsen, niin jäsenen `y` osoite on `&X::y`.
- Muuttuja, joka on tyyppiä "osoitin luokan `X` jäseneen", esitellään rakenteella `X::*`.
- Osoittimet *luokan* jäseniin eivät ole todellisia muistiosoitteita (vaan ne ovat siirtymiä, offsetteja, tietueen alusta); niitä ei pidä sotkea *olioiden* jäseniin viittaaviin osoittimiin.
- Luokan jäsenosoittimet dereferensoidaan operaattoreilla `.*` ja `->*`.

```
struct cl {
  char *val;
  void show( int x )
    { cout << val << x << '\n'; }
  cl( char *v ) { val = v; }
};
typedef void (cl::*PMFI)(int);
void f()
{
  cl z1("z1 ");
  cl z2("z2 ");
  cl *p = &z2;
  PFMI pf = &cl::show;
  z1.show(1);
  (z1.*pf)(2);
  z2.show(3);
  (p->*pf)(4);
}
```

9. Johdetut luokat (C++)

- Käsitteet, jotka C++::ssa esitetään luokkina, ovat yleensä relaatiossa keskenään.
- Käsitteet muodostavat hierarkisen systeemin. Esim.
  - Ympyrä ja kolmio ovat molemmat kuvioita: kuvion käsite on niille molemmille yhteinen.
  - C++::ssa näitä voisivat esittää luokat `circle` ja `triangle`, joille luokka `shape` on yhteinen.
- Luokasta voidaan *johtaa* (*derive*) toinen luokka.
- Johdettu luokka *perii* (*inherits*) kantaluokan ominaisuudet: data- ja funktiojäsenet.

9.1 Luokkien johto

Tarkastellaan luokkaa, joka esittää työntekijää:

```
class employee {
  char *name;
  short age;
  short department;
  int salary;
public:
  employee *next;
  ...
};
```

- Järjestetään työntekijäoliot linkitetyksi listaksi.
- Linkki `next` on julkinen, jotta listoja voitaisiin käsitellä olioiden ulkopuolelta.

Päällikkö on työntekijä, jonka alaisena on jokin ryhmä:

```
class manager : public employee {
  employee *group;
  short level;
  ...
};
```

- `manager` on johdettu luokka.
- `employee` on kantaluokka.
- Kaikki kantaluokan jäsenet ovat myös johdetun luokan jäseniä.
- Johdetussa `manager`-luokassa on myös omia jäseniä.
- Atribuutti `public` kantaluokan edellä ilmoittaa, että kaikki kantaluokan julkiset jäsenet (esim. `next`) ovat myös johdetun luokan julkisia jäseniä.
- Jos johdetulla luokalla on julkinen kantaluokka, voidaan tyyppiä "osoitin johdettuun luokkaan"oleva muuttuja sijoittaa ilman tyyppimuunnosta tyyppiä "osoitin kantaluokkaan"olevaan muuttujaan. Tämä *ei* ole kääntäen voimassa: jokainen kantaluokan olio ei ole johdetun luokan olio.

```
void f()
{ manager mm;
  employee *pe = &mm;  // OK
  employee ee;
  manager *pm = &ee;   // Virhe!
  manager *pm = (manager *)&ee; // OK, mutta
  pm->level = 2;       // on katastrofi!
  pm = (manager *)pe;  // OK, pm osoittaa
                       // manager mm:ään
  pm->level = 1;       // OK
}
```

9.2 Jäsenfunktiot

Tarkastellaan luokkia `employee` ja `manager`.
Lisätään jäsenfunktiot, jotka tulostavat ko. olioiden tiedot.
Koska luokassa `manager` on omia `employee`-luokkaan kuulumattomia datajäseniä, tarvitsee tämä oman tulostusfunktion.

```
class employee {
  char *name;
  ...
public:
  employee *next;
  void print();
  ...
};
class manager : public employee {
  ...
public:
  void print();
  ...
};
```

Viittausoikeudet

- Johdetun luokan metodit *eivät* voi viitata kantaluokan yksityisiin jäseniin.
- Jos kantaluokka on yksityinen (`class manager:employee{...};`), kantaluokan julkisiakaan jäseniniä ei voi dereferensoida johdetun luokan olioista käsin.
- Jos kantaluokka on julkinen, voidaan kantaluokan julkisiin jäseniin viitatata johdetun luokan olioitten kautta.

```
void manager::print()
{
  employee::print();  // Tulosta employee-tieto
  // Luokkaerottelu on välttämätön: pelkkä
  // print() viittaisi tähän metodiin.
  ...                 // Tulosta manager-tieto
}
```

Suojatut jäsenet

Luokan jäsenet voivat olla joko yksityisiä (`private`), julkisia (`public`) tai *suojattuja* (`protected`). Johdetun luokan metodit saavat viitata julkisen kantaluokan suojattuihin jäseniin.

```
class employee {
protected:
  char *name;
  ...
public:
  void print();
  ...
};
class manager : public employee {
  ...
public:
  void print();
  ...
};
void manager::print()
{ cout << name ...;  // OK }
```

9.3 Muodostimet ja hajottimet

Tarkastellaan luokkia `employee` ja `manager`.

```
class employee {
  ...
public:
  ...
  static employee *list;
  employee( char *n, int d );
};
class manager : public employee {
  ...
public:
  ...
  manager( char *n, int l, int d );
};
```

Oletetaan, että kaikki työntekijät linkitetään listaksi. Staattinen, ts. kaikkien `employee`- ja siitä johdettujen tyyppisten olioiden yhteinen jäsen `list` viittaa tähän listaan.

Koska kantaluokan `employee` muodostin tarvitsee argumentteja, on johdetun luokan muodostin määriteltävä siten, että kannan muodostimelle annetaan oikeat parametrit. Esim.

```
employee::employee( char *n, int d )
  : name( n ), department( d )
{
  next = list;
  list = this;
}
manager::manager( char *n, int l, int d )
  : employee( n, d ), level( l ), group( 0 )
{}
```

Muodostimien suoritusjärjestys on: ensin kanta, sitten jäsenet ja viimeksi luokka itse. Tuhoamisjärjestys on päinvastainen. Koska hajottimilla ei ole argumentteja, ei johdetun luokan hajottimen tarvitse kutsua kantaluokan hajotinta eksplisiittisesti.

9.4 Luokkahierarkiat

Johdettu luokka voi olla jonkin toisen luokan kanta. Esim.

```
class employee { ... };
class manager : public employee {
  ...
};
class director : public manager {
  ...
};
```

Tälläinen luokkahierarkia muodostaa puurakenteen. Hierarkia voi olla myös monimutkaisempi, sillä luokalla voi olla useampia kantaluokkia:

```
class temp { ... };
class secretary : public employee {
  ...
};
class tsec
  : public temp, public secretary {
  ...
};
class consultant
  : public temp, public manager {
  ...
};
```

9.5 Virtuaaliset funktiot

Tarkastellaan edellisten lukujen luokkia `employee` ja `manager`. Linkitetään näiden luokkien oliot listaksi osoitinjäsenellä `employee *next`.

- Koska linkki on tyyppiä "osoitin kantaluokan (`employee`) olioon", voidaan myös johdetun luokan (`manager`) oliot liittää automaattisesti tähän listaan.
- Koska listan rakentaminen tapahtuu ohjelman ajoaikana, kääntäjä ei ole tietoinen listan olioiden todellisesta tyypistä. Kääntäjän mielestä kaikki listan alkiot ovat tyyppiä `employee*`.
- Oletetaan, että tehtävänä on tulostaa listan alkioiden tiedot. Nyt esim. `next->print()` tarkoittaa metodia `employee::print()`, joten `manager`-luokalle yksilöityjen tietojen tulostaminen ei tällä tavoin onnistu.

Kun ohjelman on ajoaikana päätettävä olioihin sovellettavasta metodista, on tarjolla seuraavat ratkaisut:

Ratkaisu 1.
Lisätään luokkaan jäsen, joka ilmoittaa objektin tyypin:

```
struct employee {
  enum empl_type {M, E};
  empl_type type;
  ...
};
void print( employee *e )
{
  switch( e->type ) {
    case 'E':
      e->print();
      break;
    case 'M':
      ((manager *)e)->print();
```

```
      break;
   }
}
```
Tämä ratkaisu ei ole ideaalinen, sillä

- oliota käsittelevien funktioiden on oltava tietoisia tyyppi-informaation olemassaolosta: on muistettava testata tyyppi.
- oliota käsittelevien funktioiden on muistettava testata (yleensä) kaikki mahdolliset tyypin arvot.
- jos luokkien esitystä muutetaan, tämä muutos heijastuu myös luokkien ulkopuolisiin funktioihin.
- tyyppitesteillä on tapana hajaantua sinne tänne ohjelmaan: päivitettäessä ohjelmaa on vaikea etsiä kaikkia testejä.

Menetetään datan ja sitä käsittelevien metodien kapseloinnista saatu etu.

Ratkaisu 2.
Normaalisti kääntäjä sitoo metodit olioihin käännösaikana staattisesti (*static binding*). On myös mahdollista sitoa metodit ja oliot ajoaikana dynaamisesti (*run time binding*) määrittelemällä metodit *virtuaalisiksi*:
```
class employee {
  ...
public:
  ...
  static employee *list;
  virtual void print();
  void print_list();
};
class manager : public employee {
  ...
public:
  void print();
};
void employee::print() { ... }
void manager::print() { ... }
```
Listan alkioiden tiedot voidaan tulostaa jäsenfunktiolla
```
void employee::print_list()
{
  for( eployee *p = list; p; p = p->next )
    p->print();
}
```
Nyt lausekkeessa `p->print()` metodiksi `print()` valitaan joko `employee::print()` tai `manager::print()` sen mukaan, minkä tyyppiseen olioon `p` osoittaa.
Kääntäjä toteuttaa virtuaaliset funktiot siten, että virtuaalisia funktioita sisältävien luokkien olioihin lisätään näkymätön osoitin.

Abstraktit luokat

Joskus kannattaa johtaa luokkia sellaisista kantaluokista, joiden olioita sellaisinaan ei voi olla olemassa. Tarkastellaan esimerkkinä geometrisia muotoja esittäviä luokkia `triangle` ja `circle`:

- Molemmille luokille tarvitaan esim. metodit `rotate()` ja `draw()`.
- Metodit `rotate()` ja `draw()` ovat eri luokille erilaiset.
- Halutaan käsitellä muotoja yhtenäisesti, esim. halutaan rotatoida listaksi linkitettyjen olioiden muodostama kuvio.
- Tällöin kannattaa määritellä esim. luokka `shape`, josta johdetaan luokat `triangle` ja `circle`, ja jonka jäsenfunktiot `rotate()` ja `draw()` ovat virtuaalisia.

Luokan `shape` olio sellaisenaan ei ole mielekäs.
Luokka, jolla ei voi olla olioita, kannattaa määritellä *abstraktiksi*. Tämä tehdään esittelemällä jokin(jotkin) jäsenfunktio(t) *puhtaasti virtuaalisiksi* (*pure virtual*) funktioiksi:
```
class shape {
  ...
public:
  virtual void rotate( int ) = 0;
  virtual void draw() = 0;
  ...
};
```
Nyt määritelmä
```
shape s;
```
ei ole sallittu.
Abstraktia luokkaa voi käyttää ainoastaan jonkin toisen luokan kantana:
```
class circle : public shape {
  int radius;
public:
  void rotate( int ) {}
  void draw();
  ...
};
```
Puhdas virtuaalinen funktio, jota ei määritellä johdetussa luokassa, jää puhtaaksi virtuaaliseksi funktioksi ja johdettu luokka abstraktiksi.

10. Operaattorien ylikuormaus

Operaattoreille voidaan C++:ssa antaa uusia merkityksiä. Määrittelemällä luokkaolioihin operoivat operaattorit voidaan joskus saada aikaan tavanomaisempi ja mukavampi merkintätapa. Esim.

```
class complex {
  double re, im;
public:
  complex( double r, double i )
    { re = r; im = i; }
  friend complex operator+(complex, complex);
  friend complex operator*(complex, complex);
};
void f()
{ complex a = complex( 1.2, 3 );
  complex b = complex( 1, 2.4 );
  complex c = b;
  a = b + c;
  b = b + c*a;
  c = a*b + complex( 1, 2 );
}
```

10.1 Operaattorifunktiot

Uusia operaattoreita ei voida määritellä. Vanhat operaattorit

```
+   -   *   /   %   ^   &   |   ~   !
=   <   >   +=  -=  *=  /=  %=  ^=  &=
|=  <<  >>  <<= >>= ==  !=  <=  >=  &&
||  ++  --  ,   ->  []  ()  new delete
```

voidaan ylikuormata.

- Operaattorit ovat oikeastaan yksi- (unaari) tai kaksi- (binääri) argumenttisia funktioita.
- Operaattorifunktion nimi on avainsana `operator`, jota seuraa itse operaattori, esim. `operator<<`.
- Operaattorifunktio voidaan määritellä ja sitä voidaan kutsua aivan samoin kuin muitakin funktioita.

Esim.

```
void f( complex a, complex b )
{
  complex c = a + b; // Lyhyesti tai
  // eksplisiittisesti:
  complex d = operator+(a, b);
}
```

- Ylikuormauksessa operaattorit säilyttävät presedenssinsä ja assosiatiivisuutensa.
- Operaattori voidaan ylikuormata ainoastaan siten, että C++:n kielioppi säilyy ennallaan, esim. ei voida määritellä unaarista operaattoria `%`.

Binäärinen operaattori

- Jos `@` on binäärinen operaattori, se voidaan määritellä joko
  - yksiargumenttisena jäsenfunktiona tai
  - kaksiargumenttisena globaalina funktiona.
- Lauseke `aa@bb` tulkitaan vastaavasti joko
  - lausekkeeksi `aa.operator@(bb)` tai
  - lausekkeeksi `operator@(aa,bb)`.
- Jos molemmat on määritelty päätellään tulkinta operandien tyypeistä.

Unaarinen operaattori

- Jos `@` on unaarinen prefix operaattori, se voidaan määritellä joko
  - argumentittomana jäsenfunktiona tai
  - yksiargumenttisena globaalina funktiona.
- Lauseke `@aa` tulkitaan vastaavasti joko
  - lausekkeeksi `aa.operator@()` tai
  - lausekkeeksi `operator@(aa)`.
- Jos molemmat on määritelty päätellään tulkinta operandin tyypistä.
- Vastaava on voimassa myös postfix operaattoreille.

Esim.

```
class X {
  ...
  // Jäsenfunktiot
  X *operator&();    // prefix unaarinen
  X operator&(X);    // binäärinen
  X operator++();    // prefix unaarinen
  X operator++(int); // postfix unaarinen
  X operator&(X,X);  // Väärin!
  X operator/();     // Väärin!
};
// Globaalit funktiot
X operator-(X);        // prefix unaarinen
X operator-(X,X);      // binäärinen
X operator--(X&);      // prefix unaarinen
X operator--(X&, int); // postfix unaarinen
X operator-();         // Väärin!
X operator-(X,X,X);    // Väärin!
X operator%(X);        // Väärin!
```

Huom. Jos operandit ovat kooltaan suuria, kannattaa harkita viittauksien käyttöä operaattorifunktioiden argumenttina (ja mahdollisesti myös paluuarvona), jotta vältyttäisiin ylenmääräiseltä kopioinnilta.

Jos halutaan ylikuormata operaattorit `operator=`, `operator[]`, `operator()` tai `operator->` on ne määriteltävä ei-staattisiksi jäsenfunktioiksi. Tällöin niiden ensimmäinen operandi on välttämättä v-arvo (lvalue).

Seuraavilla operaattoreilla on luokkaolioihin sovellettuna ennalta annettu merkitys:

`=`	bitittäinen sijoitus
`&`	olion osoite
`,`	sekvenssointi

Ohjelmoija voi kätkeä nämä merkitykset joko

- esittelemällä ko. operaattorit luokkien yksityisiksi jäseniksi tai
- ylikuormaamalla ne.

10.2 Käyttäjän määrittelemät tyyppimuunnokset

Tarkastellaan luokkaa `complex`. Jotta esim. yhteenlasku voitaisiin ilmaista luonnollisella tavalla, pitäisi luokalla olla ystäväfunktiot eri operandityypeille:

```
class complex {
  double re, im;
public:
  complex( double r, double i )
    { re = r; im = i; }
  friend complex operator+(complex, complex);
  friend complex operator+(complex, double);
  friend complex operator+(double, complex);
  // Vastaavat ystävät muille aritmeettisille
  // toiminnoille
};
void f() {
  complex a(1, 2), b(2.4, 3);
  a = a + 2.5;   // Kääntäjä sovittaa float- ja
  b = b + a + 1; // int-argumentit double:ksi
}
```

Muodostimet ja tyyppimuunnokset

Lukuisten ystäväfunktioiden määrittely voidaan välttää määrittelemällä sopivia muodostimia:

```
class complex {
  ...
  complex(double r) { re = r; im = 0; }
  friend complex operator+(complex, complex);
};
void f()
{
  // Eksplisiittinen muodostin
  complex z1 = complex( 5 ); // complex(double)
  // Implisiittinen muodostin
  complex z2 = 23;  // complex(double)
  z1 = z1+2; // z1 = operator+(z1,complex(2))
}
```

Muunnosoperaattorit

Muodostimien puutteita ovat mm.

- Koska perustyypit eivät ole luokkia, ei voi olla olemassa implisiittistä muunnosta käyttäjän määrittelemästä tyypistä perustyyppiin.
- Ei ole mahdollista määritellä muunnosta uudesta tyypistä ennestään olemassaolevaan tyyppiin muuttamatta sen määrittelyä.

Nämä puutteet voidaan korjata määrittelemällä *muunnosoperaattori*: jos `X` on luokka ja `T` jokin toinen tyyppi, niin jäsenfunktio `X::operator T()` määrittelee muunnoksen tyypistä `X` tyyppiin `T`.
Tarkastellaan esimerkkinä luokkaa, jonka oliot ovat pieniä kokonaislukuja välillä 0–63.

```
class tiny {
  char v;
  void assign( int i )
    { if( i > 63 || i < 0 )
        error("Ei sallituissa rajoissa" );
      v = i & 63;
    }
public:
  tiny( int i ) { assign( i ); }
  tiny( const tiny& t ) { v = t.v; }
  tiny& operator=(const tiny& t)
    { v = t.v; return *this; }
  tiny& operator=(int i)
    { assign( i ); return *this; }
  operator int() { return v; }
};
```

Luokan `tiny` olioita voidaan nyt vapaasti sekoittaa aritmeettisissa lausekkeissa kokonaislukujen kanssa.

```
main()
{
  tiny c1 = 2;
  tiny c2 = 62;
  tiny c3 = c2 - c1;  // c3 = 60
  tiny c4 = c3;     // Ei arvon tarkistusta
  int i = c1 + c2;  // i = 64
  c1 = c2 + 2*c1;   // Liian suuri
  c2 = c1 - i;      // Ei sallituissa rajoissa
  c3 = c2;          // Ei arvon tarkistusta
}
```

10.3 Sijoitus ja alustus

Luokkaolioihin kohdistuvan sijoitusoperaattorin (=) oletusmääritelmä on bitittäinen kopiointi. Tämä ei aina ole tarkoituksenmukaista, esim.

```
struct string {
  char *p;
  int size;   // vektorin p koko
  string( int sz )
    { p = new char[size = sz]; }
  string( const char *s )
    { p = new char[size = strlen( s ) + 1];
      strcpy( p, s ); }
  ~string() { delete[] p; }
};
void f()
{ string s0( "Heippa!" );
  string s1( 10 );
  string s2( 20 );
  s1 = s2;  // alkuperäinen s1.p unohtuu
  // Hajottimet tuhoavat s2.p:n kahdesti!
}
```

Parempi ratkaisu on sijoitusoperaattorin ylikuormaus:

```
struct string {
  char *p;
  int size;   // vektorin p koko
  string( int sz )
    { p = new char[size = sz]; }
  string( const char *s )
    { p = new char[size = strlen( s ) + 1];
```

```
      strcpy( p, s ); }
  ~string() { delete [] p; }
  string& operator=( const string& );
};
string& string::operator=( string& s )
{
  if( this != &s ) {  // s = s?
    delete[] p;
    p = new char[size = s.size];
    strcpy( p, s.p );
  }
  return *this;
}
```

Kuitenkaan käyttäjän määrittelemää sijoitusoperaattoria *ei* sovelleta alustuksissa:

```
void f()
{
  string s1( 10 );
  string s2 = s1;  // Alustus, ei sijoitus
  // Hajottimet tuhoavat kaksi
  // string-oliota ja saman
  // vektorin kahdesti!
}
```

Sijoitus ja alustus ovat eri toimintoja!

Tämän kaltaisten ongelmien välttämiseksi on syytä määritellä ns. *kopiomuodostin* (*copy constructor*).

```
struct string {
  char *p;
  int size;   // vektorin p koko
  string( int sz )
    { p = new char[size = sz]; }
  string( const char *s )
    { p = new char[size = strlen( s ) + 1];
      strcpy( p, s ); }
  string( const string& );  // Kopiomuodostin
  ~string() { delete[] p; }
  string& operator=( const string& );
};
string::string( string& s )
{
  p = new char[size = s.size];
  strcpy( p, s.p );
}
void f()
{
  string s1( 20 );
  string s2 = s1;  // Kopiomuodostin
}
```

Olio kopioidaan, joko bitittäin tai ohjelmoijan määrittelemää kopiomuodostinta soveltaen, kolmessa tapauksessa:

- alustettaessa määriteltäviä muuttujia.
- välitettäessä argumentteja funktiolle. Tällöin muodolliset argumentit alustetaan täsmälleen samallatavoin kuin muuttujia määriteltäessä.
- palautettaessa funktion arvo.

Esim.

```
string g( string arg )
{
  return arg;
}
main()
{
  string s = "Huuhaa!";
  s = g( s );
}
```

- Kutsussa `g( s )` funktion `g` muodollinen argumentti `arg` alustetaan kopiomuodostimella `string::string( string&)`. Alustuksen merkitys on sama kuin lauseen `string arg = s;`.
- Kun funktio `g` suorittaa käskyn `return arg;`, muodostetaan kopioimalla (`string::string( string& )`) väliaikainen muuttuja, joka sitten kutsuvassa ohjelmassa sijoitetaan (`string& string::operator=( string& )`) olioon `s`. Tämä väliaikainen muuttuja tuhoutuu automaattisesti hajottimella (`string::~string()`).

Kolmen suuren laki

Jos luokka tarvitsee hajottimen tai kopiomuodostimen tai sijoitusoperaattorin, se tarvitsee ne kaikki!

10.4 Indeksointi

Binäärisen `operator[]`-funktion tavallisin tarkoitus on määritellä luokkaolioille indeksointi. Toinen argumentti (ensimmäinen on luokkaolio itse), indeksi, voi olla mitä tahansa tyyppiä. Tarkastellaan esimerkkinä assosiatiivista taulukkoa, jossa merkkijonoihin liitetään numeerinen arvo.

```
class assoc {
  struct pair {
    char *name;
    int val; };
  pair *vec;
  int max;
  int free;
  assoc( const assoc& );
  assoc& operator=( const assoc& );
public:
  assoc( int );
  int& operator[]( const char * );
  void print_all();
};
```

- Luokkamäärittelyn sisällä voidaan määritellä luokkia. Näiden nimien käyttöön pätevät samat säännöt kuin jäseniin. Tässä `pair` on luokan `assoc` yksityinen tyyppi.
- Esittelemällä kopiomuodostin ja sijoitusoperaattori yksityisiksi jäseniksi estetään `assoc`-olioiden kopiointi.
- Muuttuja `max` ilmoittaa vektorin `vec` alkioiden lukumäärän ja `free` on ensimmäisen käyttämättömän alkion indeksi tässä vektorissa.

Määritellään muodostin:
```
assoc::assoc( int s )
{
  max = ( s < 16 ) ? s : 16;
  free = 0;
  vec = new pair[max];
}
```
Määritellään indeksointi siten, että tuloksena on joko ennestään talletettuun merkkijonoon liitetty kokonaisluku tai talletetaan uusi merkkijono ja annetaan tarvittaessa taulukon `vec` kasvaa:
```
int& assoc::operator[]( const char *p )
{ pair *pp;
  for( pp = vec + free - 1; vec <= pp; pp-- )
    if( strcmp( p, pp->name ) == 0 )
      return pp->val;
  if( free == max ) {  // Ylivuoto
    pair *nvec = new pair[2*max];
    for( int i = 0; i < max; i++ )
      nvec[i] = vec[i];
    delete[] vec;
    vec = nvec;
    max = 2*max;
  }
  pp = vec + free++;
  pp->name = new char[strlen( p ) + 1];
  strcpy( pp->name, p );
  pp->val = 0;          // Alkuarvo
  return pp->val;
}
```
Koska `assoc`-olioiden data on kätketty, tarvitaan tulostamiseen julkinen metodi:
```
void assoc::print_all()
{
  for( int i = 0; i < free; i++ )
    cout << vec[i].name << ": "
         << vec[i].val << '\n';
}
```
Kirjoitetaan ohjelma, joka laskee syötettyjen sanojen frekvenssin:
```
main()
{
  const MAX = 256;
  char buf[MAX];
  assoc vec(128);
  while( cin >> buff )
    vec[buff]++;
  vec.print_all();
}
```

11. Poikkeukset (Exceptions)

11.1 Virhetilanteet

Tässä virhe

- tarkoittaa tilannetta, johon jouduttuaan funktio tai metodi ei voi toteuttaa sille annettua tehtävää.
- ei tarkoita ohjelmointivirhettä (ainakaan kyseisessä funktiossa).
- on tilanne, johon voidaan varautua.

Tarkastellaan esimerkkinä pinoa
```
class stack {  // Merkkipino
  int size;
  int *top;
  int *s;
public:
  stack(int sz)
    {top = s = new char[size = sz]; }
  ~stack() {delete[] s;}
  void push(int i) { *top++ = i; }
  int pop();
  int count() const { return top - s; }
};
```
Metodi `pop()` voi tahtomattaan joutua kutsutuksi pinon ollessa tyhjä johtuen esim.

- ohjelmointivirheestä sitä kutsuvissa funktioissa.
- ajoaikaisista olosuhteista kuten virheellisistä syöttötiedoista.

Huomattakoon, että

- useimmat tämän kaltaiset virhetilanteet olisivat vältettävissä, mikäli ennen metodien kutsua tarkistettaisiin, onko ko. operaatio toteutettavissa.
- esim. ennen `pop()`-operaatiota tulisi `count()`-metodilla tarkastaa, että pino ei ole tyhjä.
- ylenmääräisten tarkistusten tekiminen on työlästä.
- negatiivisen tuloksen antaneen tarkistuksen seurauksena saattaa tarkistava funktio joutua tilanteeseen, jota se ei hallitse.

Eräs tapa on lisätä olioihin atribuutti, virhekoodi, kertomaan onnistuiko operaatio vai ei. Esim.
```
class stack {
  ...
  bool ok;
public:
  stack( int sz ) : ok( true )
    {top = s = new char[size = sz]; }
  ...
  bool isOK() const { return ok; }
};
int stack::pop()
{
  if( top == s ) {
```

```
    ok = false;
    return 0;
  }
  return *--top;
}
```

Näin menetellen

- operaatiot eivät virhetilanteessakaan johda katastrofiin.
- esim. `pop()` toimii kaikissa tilanteissa: osoitin `top` pysyy aina sallituissa rajoissa.
- kutsujan vastuulle jää virhekoodin tarkistus.
- ohjelman logiikasta saattaa tulla monimutkainen, jos virhe syntyy syvällä sisäkkäisissä funktiokutsuissa ja virheen voi hoitaa (tai todeta että mitään ei ole tehtävissä) vain jokin ulommaisista funktioista.

Esimerkiksi

```
void huu( stack& s )
{
  ...
  int i = s.pop();
  if( !s.isOK() )
    return;
  ...
}
void haa( stack& s )
{
  ...
  huu( s );
  if( !s.isOK() )
    return;
  ...
}
void foo()
{
  ...
  stack x( 100 );
  ...
  haa( x );
  if( !x.isOK() ) {
    ...    // hoida virhe
  }
  ...
}
```

Ei-lokaaliset hypyt (C)

Eräs ongelma on siis hallittu paluu virhetilanteeseen joutuneesta funktiosta useamman toisiaan kutsuvan funktion kautta. Tämä voidaan toteuttaa esimerkiksi *ei-lokaalisena hyppynä (nonlocal* `goto`*)*:

- kohta, johon halutaan palata, merkitään kutsumalla funktiota `int setjmp(jmp_buf jmpb)`.
- välittömästi palatessaan funktion `setjmp` arvona on 0.
- pidetään `jmp_buf`-muuttuja tallessa, esim. globaalina muuttujana.
- kutsuttaessa myöhemmin funktiota `void longjmp(jmp_buf jmpb, int v)` funktio `setjmp` palaa uudemman kerran arvonaan tällä kertaa `v`.

Esim.

```
jmp_buff jmpb;
...
void huu( stack& s )
{
  int i = s.pop();
  if( !s.isOK() )
    longjmp( jmpb, 1 );
  ...
}
void haa( stack& s )
{
  ...
  huu( s );
}
void foo()
{
  stack x( 100 );
  int v;
  ...
  v = setjmp( jmpb );
  if( v != 0 ) {
    ...     // hoida virhe
  }
  ...       // normaali prosessointi
  haa( s );
  ...
}
```

11.2 Poikkeukset (C++)

`longjmp` aiheuttaa välittömän hypyn aiemmin merkittyyn kohtaan. Tällöin jäävät mm. funktiossa määritellyt objektit tuhoamatta, niiden hajoittimia ei koskaan kutsuta. Sen vuoksi C++:ssa ei pitäisi koskaan käyttää `setjmp`/`longjmp`-mekanismia, vaan

- antaa virhetilanteeseen joutuneen metodin/funktion *heittää* (*throw*) (virhe)objektin.
- antaa sen metodin, joka pystyy virhetilanteen käsittelemään, *yrittää* (*try*) sellaisia metodeja tai sellaisia metodeja kutsuvia metodeja, joiden se epäilee mahdollisesti joutuvan virheeseen.
- *siepata* (*catch*) yrityksen seurauksena heitetyt objektit.

Esim.

```
class underflow {...};

class stack {
  ...
public:
  stack( int sz )
    {top = s = new char[size = sz]; }
  ...
```

```
  int pop() throw( underflow );
};
int stack::pop() throw( underflow )
{
  if( top == s )
    throw underflow();
  return *--top;
}
```
Nähdään, että

- heitettävän objektin tyyppi voi olla mikä tahansa, myös perustyypit (`int`, `char`, `char*`, ...) kelpaavat.
- heitettävien objektien tyypit kannattaa kuitenkin nimetä niin, että ne ovat virhettä kuvaavia.
- esiteltäessä ja määriteltäessä sellaista funktiota tai metodia, joka voi heittää virheen, on generoituvan koodin tehostamiseksi suositeltavaa ilmoittaa siitä luettelemalla avainsanan `throw` jälkeen suluissa niiden objektien tyypit, joita ko. funktio mahdollisesti heittää.
- esiteltäessä funktio lisämääreellä `throw()` kääntäjä olettaa että kyseinen funktio ei heitä virhettä.
- esittelysäännöt ovat ohjeellisia. Ne eivät esim. takaa, että funktio ei heitä virhettä tai että se voi heittää vain luetellun tyyppisiä virheitä.
- virhe heitetään käskyllä `throw`, jota seuraa heitettävä objekti. Esimerkissä,

  `throw underflow();`,

  objekti luodaan oletusmuodostinta soveltaen vasta heitettäessä.

Funktioita yritetään ja virheita siepataan, kuten
```
void huu( stack& s )
{
  ...
  int i = s.pop();
  ...
}
void haa( stack& s )
{
  ...
  huu( s );
  ...
}
void foo()
{
  stack x( 100 );

  try {
    haa( x );
    ...
  }
  catch( underflow& err ) {
    ...   // hoidetaan virhe
  }
  ...
}
```
Menetellään siten, että

- siinä funktiossa/metodissa, joka pystyy virhetilanteen käsittelemään, kiedotaan se osa koodista, jonka virheistä ollaan kiinnostuneita, `try`-lohkoon:

  `try { ... }`

- `try`-lohkosta heitetty virhe siepataan sitä seuraavalla `catch`-käskyllä.
- `catch`-lauseen syntaksi ja semantiikka on sama kuin funktion: toimitaan kuten `try`-lohkosta oltaisiin kutsuttu `catch`-nimistä funktiota argumentin ollessa heitetty objekti.
- `catch`-lohkosta voidaan edelleen heittää virhe. Tämä voidaan siepata jossakin ylemmän tason `try`/`catch`-konstruktiossa.

Huomattakoon, että

- `catch`-lohkoja voi olla useita peräkkäin. Tällöin sieppauksen hoitaa se `catch`, jonka argumentti ensimmäisenä sovittuu heitetyn objektin tyyppiin. Käytössä ovat normaalit automaattiset konversiot: `int`$\longrightarrow$`float`$\longrightarrow\ \cdots$, osoitin/referenssi johdetun luokan objektiin $\longrightarrow$ osoitin/referenssi kantaluokan objektiin.
- peräkkäisistä `catch`-lohkoista toteutetaan korkeintaan yksi.
- vain `try`-lohkosta peräisin olevat poikkeukset ovat siepattavissa.
- sieppaamatta jääneet virheoliot etenevät aina ulommille kutsuville metodeille, lopulta `main`-funktioon ja siitä käyttöjärjestelmälle.

Poikkeusmekanismin eduista verrattuna esim. virhekoodeihin mainittakoon, että

- ohjelmointi on helpompaa, koska ei ole tarpeen jatkuvasti testata objektien validiteettia.
- ohjelmat ovat (yleensä) nopeampia, koska ainoastaan virheen sattuessa joudutaan tekemään jotain poikkeavaa.
- ohjelmat ovat helpommin luettavissa ja ymmärrettävissä, koska virheenkäsittelyn logiikka on erotettu muusta logiikasta.

Esim.: epäonnistuvat objektit
Usein metodin tulosta ei kuvaa pelkästään sen paluuarvo. Esimerkiksi etsintöjä suorittavien metodien olisi syytä myös ilmoittaa, onnistuiko haku vai ei. Tälläisistä metodeista kannattaa joskus palauttaa ns. *epäonnistuvia* (*fallible*) objekteja. Epäonnistuvan objektin mallipohja voisi olla vaikkapa

```
template<class T>
class Fallible {
public:
  Fallible() : fail( true ) {}
  Fallible( T t ) : fail( false ), v( t ) {}

  int ok() const { return !fail; }
  operator T() const { return v; }
private:
  bool fail;
  T v;
};
```

Esimerkiksi kokonaisluku-merkkijono-pareja sisältävästä `Tree`-rakenteesta merkkijonoon liittyvää kokonaislukua etsivä metodi olisi skemaattisesti

```
Fallible<int>
Tree::seek( const char* s )
{
  ...
  if( ... ) {
    i = ...      // löytyi
    return Fallible<int>( i );
  }
  else
    return Fallible<int>();  // ei löytynyt
}
```

Paluuarvoa voidaan käyttää aivan kuin olisi palautettu kokonaisluku:

```
Tree t;
...
int i = t.seek( "huu" );
```

Kutsujan tulisi kuitenkin testata `ok()`-metodia käyttäen paluuarvon validiteetti. Monesti voidaan kuitenkin pitää virheenä haun epäonnistumista. Muutetaan sen vuoksi luokan `Fallible` määritelmää hieman:

```
class Failed { ... };

template<class T>
class Fallible {
public:
  ...
  operator T() const throw( Failed )
    { if(fail) throw Failed(); else return v; }
private:
  bool fail;
  T v;
};
```

Kietomalla nyt `seek`-metodia kutsuva koodi `try`-pakettiin ja sieppaamalla `Failed`-tyyppiset objektit voidaan epäonnistuneet haut käsitellä virhetilanteina.

12. FORTRAN, C ja C++

12.1 Linkitysmääritteet

C++-kielinen ja jonkin muun kielinen koodi voidaan linkittää yhteen *linkitysmääritteillä* (*linkage specification*)

`extern "`*kieli*`" {`*esittelylista*`}`

tai

`extern "`*kieli*`"` *esittely*

- *Kieli* on jokin ohjemointikieli.
- Käytettävissä olevat kielet riippuvat kääntäjän implementaatiosta. Ainoastaan `"C"` ja `"C++"` määritteet kuuluvat standardiin.
- Standardi-C-kirjastoon liittyvissä otsikkotiedostoissa mm. funktioiden prototyypit ympäröidään automaattisesti määritteellä `extern "C" {...}` silloin, kun niitä käytetään C++-ympäristössä.

Linkitysmääritteet toimivat kumpaankin suuntaan: C++-ohjelmasta voidaan kutsua jollakin muulla kielellä kirjoitettua funktiota ja muun kielisestä ohjelmasta voidaan kutsua C++-funktiota.

Esim.

```
extern "FORTRAN"
float foo( const float& x )
{ return 5*x; }

...

  REAL A, Z
  ...
  A = 2
  Z = FOO( A )
  ...
```

Huom. Useimmat C++-implementaatiot eivät tunne määritettä `"FORTRAN"`. Toisaalta esim. UNIX-ympäristössä `FORTRAN`in ja C:n kutsusekvenssit samoin kuin globaalien tunnusten generointi (alleviivauksen liittäminen tunnuksen eteen) ovat ekvivalentteja.

12.2 FORTRAN ja C++

`FORTRAN`issa argumentit välitetään *call by reference*-mekanismilla. Vastaavissa C++-funktiossa argumentit esitellään siten joko referensseinä, kuten

```
extern "FORTRAN"
float foo( const float& x )
{ return 5*x; }
```

tai osoittimina, kuten

```
extern "FORTRAN"
float foo( const float* x )
{ return 5* *x; }
```

`FORTRAN`in funktioita vastaavat C++:n vastaavan tyyppiset funktiot (`REAL` $\longleftrightarrow$ `float`, `INTEGER` $\longleftrightarrow$ `int`, ...) ja `SUBROUTINE`-tyyppisiä aliohjelmia `void`-tyyppiset funktiot.

Taulukot

`FORTRAN`- ja C++-funktioiden välillä yksiulotteiset taulukot voidaan siirtää yksinkertaisesti käyttäen taulukon nimeä. Esim.

```
extern "FORTRAN"
void foo( const float*, const int& d, float& );
...
const int dim = 20;
float* vec = new float[dim];
float res;
...
vec[0] = 3.5; // FORTRANissa vec( 1 )
foo( vec, dim, res );
...
C FORTRAN-KOODI

  SUBROUTINE FOO( V, K, R)
  DIMENSION V(K)
  REAL S
  ...
  S = V(1)
C V(1) on taulukon 1. alkio (3.5)
  ...
  R = ...
  RETURN
  END
```

Useampiulotteisten taulukkojen kyseessä ollen on muistettava, että taulukot talletetaan muistiin siten, että

- `FORTRAN`issa vasemman puoleiset indeksit juoksevat nopeimmin:

$$a_{111}a_{211}\dots a_{k11}a_{121}a_{221}\dots a_{kmn}.$$

- C++:ssa (C:ssä) oikean puoleiset indeksit juoksevat nopeimmin:

$$a_{000}a_{001}\dots a_{00,n-1}a_{010}a_{011}\dots a_{k-1,m-1,n-1}.$$

Siten esim. `FORTRAN`in `V(I,J)` viittaa C++:n vastaavan taulukon `v` elementtiin `v[j-1][i-1]`.
Eräs mahdollinen tapa välittää useampiulotteisia taulukoita on transponoida ne. Huomattavasti tehokkaampaa on kuitenkin luoda C++:n taulukkoluokkia, joissa indeksointi tehdään `FORTRAN`in tapaan.
Esim. Kaksiulotteisen `FORTRAN`-tyyppisen taulukon mallipohja

```
template<class T>
class Array2D {
public:
  Array2D( int r, int c ) : r_( r ), c_( c )
    { v_ = new T[r*c]; }
  ~Array2D() { delete[] v_; }

  T& operator()( int i, int j )
    { return v_[j*r_ + i]; }
  T* array() { return v_; }
  operator T*() { return v_; }
private:
  const int r_; // rivien lkm
  const int c_; // sarakkeiden lkm
  T* v_;        // elementit
};
```

Tätä mallipohjaa voidaan käyttää, kuten

```
extern "FORTRAN"
float foo( float*, const int&, const int& );
...
Array2D<float> arr(5, 10);
...
arr(3,0) = 4.5;
...
float z = foo( arr, 5, 10);
..
```

Viittauksia taulukon alkioihin voitaisiin nopeuttaa laskemalla valmiiksi ja tallettamalla luokan jäsentaulukkoon konstruoinnin yhteydessä siirtymäindeksit `j*r_`.

13. Assembly-kielistä

13.1 Symbolinen konekieli (assembly)

Suorittimen ymmärtämät *konekieliset* käskyt esiintyvät koneen muistissa muotoa `toimintokoodi` `operandi` olevina binäärilukuina. Assemblykielessä

- esitetään toimintokoodit *mnemonisina* symboleina, esim. konekielistä toimintoa `0xA1` vastaa mnemoninen koodi `mov`.
- mahdollisista operandeista, jotka yleensä ovat joko muistiosoitteita tai prosesorin rekistereitä, voidaan käyttää symbolisia nimiä.

Esim.

```
mov ax,[x]
```

tarkoittaa toimintoa "siirrä osoitteessa `x` oleva 16-bittinen luku rekisteriin `ax`".
Assembleri on ohjelma, joka korvaa symbolit niitten todellisilla arvoilla, ts. tekee konekielisen ohjelman. Yhtä assemblykielistä käskyä vastaa yksi konekielinen käsky.
Symbolisella konekielellä kirjoitetaan

- lyhyitä ohjelmia, koska lähtö- ja konekoodin välillä on yksikäsitteinen vastaavuus.
- ohjelmat, joiden muistin käyttö halutaan minimoida. Korkean tason kielten kääntäjät eivät aina osaa optimoida koodin kokoa.
- reaaliaikaisia kontrollisovellutuksia. Korkean tason kielissä (tai käyttöjärjestelmissä) ei aina ole mekanismeja tai kääntäjän tekemä konekielinen koodi on liian hidas poikkeustilanteitten käsittelyyn.
- ohjelmat, jotka käsittelevät suoraan koneen kovoa. Korkean tason kielet ovat koneesta riippumattomia; niissä ei ole tukea esim. PC:n VGA-videokortin ohjaamiseen. (Tähän tarkoitukseen on toki olemassa aliohjelmakirjastoja, joiden funktioita voidaan kutsua C-kielisestä ohjelmasta).

Yleensä ohjelmat tulisi kirjoittaa

- korkean tason kielillä.
- koneriippumattomiksi, jos mahdollista.
- siten, että koneriippuvat osat on eristetty yhteen paikkaan.

Symbolisella konekielellä ohjelmoitaessa on aina muistettava, että koodi riippuu sekä prosessorista että assembleristä.

13.2 TASM:n syntaksi (lauseoppi)

Tässä kurssissa käytettävä assembleri on BORLANDin TASM. Haluttaessa TASM saadaan ymmärtämään myös MicroSoftin assemblerin (MASM) syntaksia.
Esitetään alustavasti muutamia tärkeimpiä kohtia TASMin syntaksista.

Kokonaisluvut

xxxx	desimaaliluku	*x*=0,1,...,9
xxxx`B`	binääriluku	*x*=0,1
xxxx`H`	heksaluku	*x*=0,1,...,9,`A`,...,`F`

Myös pienet kirjaimet (`b`, `h`, `a`,...,`f`) kelpaavat.

Merkki- ja merkkijonovakiot
ovat muotoa '*merkit*'.

Nimet (tunnukset)

- alkavat kirjaimella tai merkillä `_`, `?`, `$` tai `@`.
- pienet kirjaimet muunnetaan isoiksi, ellei käytetä `/ML` tai `/MX` optioita.
- on olemassa varattuja nimiä.
- nimi tarkoittaa (yleensä) muistiosoitetta tai vakiota.

Käskyt
Käskyjä voi olla vain yksi/rivi. Niiden muoto on

[*nimi*:] *mnem* [;*kommentti*]

Tässä

- [] tarkoittaa optionaalista.
- *mnem* on mnemoninen toimintokoodi.

Yleensä

- *nimi* aloitetaan 1. sarakkeelta.
- *mnem* aloitetaan 1. tabulaattoripositiosta.

Kommentit
Puolipisteen (`;`) jälkeinen rivin loppu tulkitaan kommentiksi.

14. 8086:n ohjelmointi ja osoitusmoodit

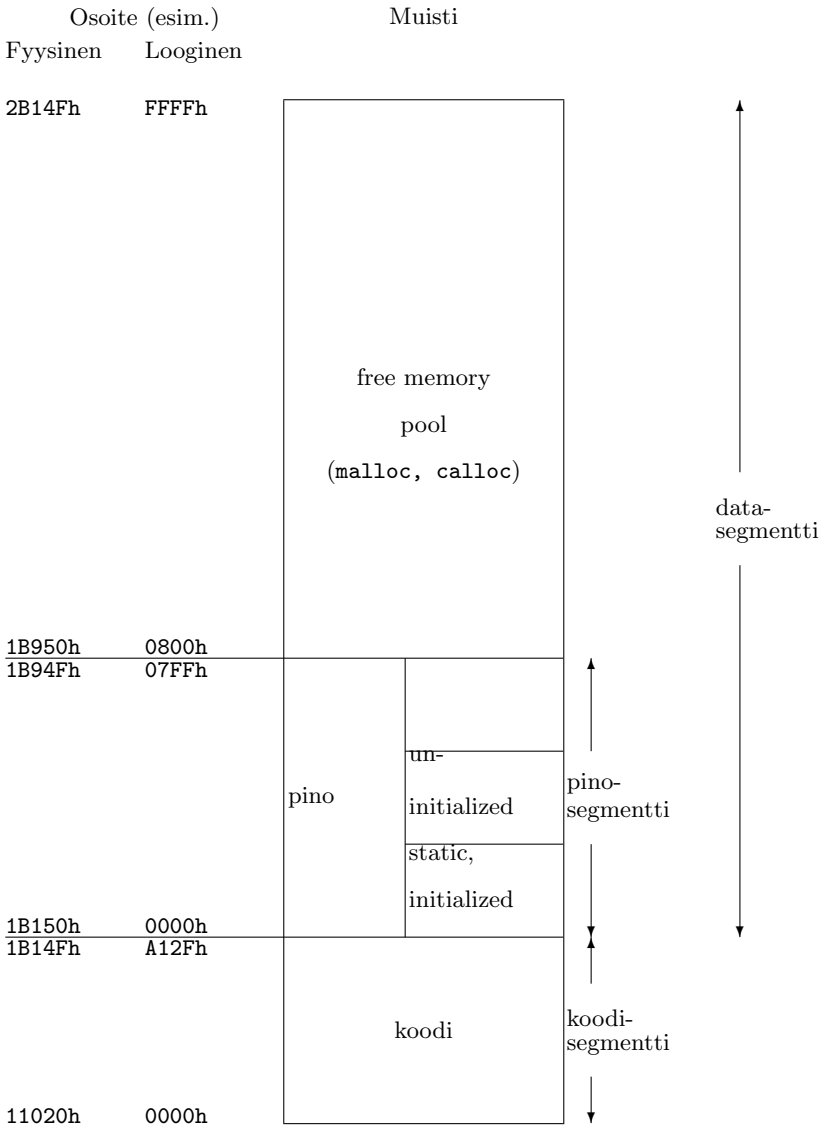
14.1 Muistin segmentointi

Ohjelmoija jakaa ohjelman käyttämän muistin loogisiin segmentteihin

Segmentti	Pääasiallinen käyttö	TASM nimi
Koodisegmentti (code segment)	ohjelman koodi	`.CODE`
Datasegmentti (data segment)	data	`.DATA`
Pinosegmentti (stack segment)	prosessorin pino	`.STACK`

- Kukin segmentti on korkeintaan 64kB.
- Saman tyyppisiä segmenttejä voi olla useita.
- Segmentit voivat olla osittain tai kokonaan päällekkäisiä

Esim. BORLAND C:n käyttämä segmentointi (pieni malli):



Huom. Ohjelmoija ei yleensä määrää fyysisiä osoitteita, vaan

- link-editori (ja ohjelmoija) määräävät suhteelliset osoitteet (*offset*) ohjelman sisällä.
- lataaja sijoittaa ohjelman vapaaseen paikkaan muistissa, ts. määrää kantaosoitteen (yo. esimerkissä `11020h`).

Kyseessä on ns. *vapaastisijoittuva* (*relocatable*) ohjelma. Useimmiten ohjelmoijan ei tarvitse huolehtia suhteellisistakaan osoitteista; assembleri laskee automaattisesti esim. konekoodin osoitteet.

14.2 8086:n datan esitysmuodot

Pienin osoitettava muistiyksikkö on 1B = 8bit. Muistin jokaisella tavulla on oma osoitteensa. Yksittäisen tavun (tai suuremman yksikön) sisällä bitit numeroidaan lähtien vähiten merkitsevästä bitistä, jonka järjestysnumero on 0.

ASCII-merkit
1B = 8bit

Kokonaisluvut

Koko	Merkki	Arvoalue
1B	etumerkitön	0–255_{10}
1B	etumerkillinen	-128_{10}–$+127_{10}$
2B	etumerkitön	0–65535_{10}
2B	etumerkillinen	-32768_{10}–$+32767_{10}$
4B	...	...
...	...	...

Useampitavuinen kokonaisluku talletetaan muistiin siten, että vähemmän merkitsevä tavu on pienemmässä osoitteessa. Esim. luku
1025_{10} = `401h` = `0100 0000 0001B`
on muistissa järjestyksessä

`01h`	`04h`

Jos tavun `01h` osoite on a, niin tavun `04h` osoite on $a+1$. Useampitavuisen kokonaisluvun osoite on sen *vähiten* merkitsevän tavun osoite. Tätä esitystä sanotaan *little endian*-järjestykseksi. (On olemassa myös prosessoreita, jotka käyttävät *big endian*-esitystä, esim. Motorola).

Muistiosoitteet
8086:n fyysinen osoite on 20 bittinen. Muistiin talletetut osoitteet ovat

Koko	Merkitys
1B	siirtymä jostakin kantaosoitteesta
2B	siirtymä jostakin kantaosoitteesta
4B	2B siirtymä ja 2B kanta

fyysinen osoite = 16×kanta + siirtymä

Osoitteet talletetaan muistiin kuten useampi tavuiset kokonaisluvut.

Käskykoodit
Käskykoodit talletetaan muistiin järjestyksessä

- 1–2B toiminto-osa, jota seuraa
- 0–4B operandiosa.

14.3 80xxx:n rekisterit

Koska suoritin pystyy käsittelemään dataa pääasiassa vain omassa sisäisessä muistissaan, rekistereissä, on mikroprosessorin ohjelmointi enimmältään

- tiedon siirtoa muistista rekistereihin
- tiedon siirtoa rekistereistä muistiin
- tiedon siirtoa rekistereistä rekistereihin
- rekisterien sisällön käsittelyä aritmeettisilla ja loogisilla toiminnoilla.

Rekistereiden käyttö
80xxx:n rekisterit eivät ole täysin symmetrisiä: tietyille rekistereille on varattu tietty käyttötarkoitus:

`eax, ax, al, ah`
`ebx, bx, bl, bh`
`ecx, cx, cl, ch`
`edx, dx, dl, dh`
	aritmeettiset ja loogiset toiminnot

`ebx`	taulukkojen kantaosoitteet
`ecx, cx, cl`	silmukkalaskurit
`dx`	I/O-portin osoitus
`esp, ebp, esi, edi`	taulukkoindeksi
`ecs, eds, ess, ees`	segmenttien kantaosoitteet
`ip`	seuraavaksi suoritettavan käskyn osoite koodisegmentissä

14.4 Osoitteitus

Käytetään käskyjen ja osoitusmoodien kuvaamiseen seuraavia merkintöjä:

expr jokin kokonaisluvun tulokseksi antava lauseke.

r jokin rekisterin nimi (`ax, bx, al, ...`).

[***expr***] sen muistipaikan sisältö, jonka osoite on *expr*.

[***r***] rekisterin *r* sisältö.

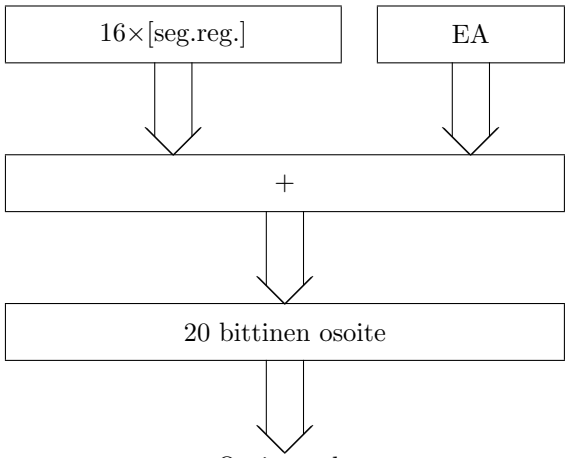
dest*←*sour *sour*:n ilmaisema data siirretään *dest*:n ilmaisemaan kohteeseen.

Esim.

[`sp`]	rekisterin `sp` sisältö
[[`sp`]]	sen muistipaikan sisältö, jonka osoite on rekisterissä `sp`
[[`sp`]]←[`ax`]	`sp`:n osoittamaan kohtaan muistissa viedään `ax`-rekisterin sisältö

Muistiviittaus (memory reference) ja osoitteen muodostus
Toimintoja, jotka tarvitsevat dataa, sanotaan muistiviittaustoiminnoiksi. Jos tarvittava data on muistissa, laskee prosessori ensin ns. *efektiivisen osoitteen* (*Effective*

Address, EA) ja lisää tähän 16×jonkin segmenttirekisterin sisällön.



Esim. seuraava suoritettava käsky otetaan osoitteesta 16×[cs]+[ip].
Käytettävä segmenttirekisteri riippuu siitä, millä tavoin EA on muodostettu.

14.5 Osoitusmoodit

8086 suoritin voi viitata dataan seitsemällä eri tavalla, osoitusmoodilla.

1. **Välitön (immediate)**
   Operandi on osa käskykoodia; seuraa toimintokoodia.
   TASM syntaksi: *expr*
   Esim.
   `mov ax, 1010H`
   Efekti: [ax]←1010H

2. **Rekisteri (register)**
   Operandi on rekisterissä.
   TASM syntaksi: *r*
   Esim.
   `mov ax,bx`
   Efekti: [ax]←[bx]

3. **Suora (direct)**
   Operandi on käskykoodin operandiosan ilmoittamassa muistipaikassa.
   TASM syntaksi: *segm*:[*expr*]
   Oletussegmentti on `ds`.
   EA = *expr*
   Esim.
   `mov ax,[1010H]`
   Efekti: [ax]←[1010H]

4. **Epäsuora rekisterin kautta (register indirect)**
   Operandi on rekisterin ilmoittamassa osoitteessa.
   TASM syntaksi: [*r*]
   Oletussegmentti riippuu *r*:stä.
   EA = [*r*]
   Esim.
   `mov ax,[bx]`
   Efekti: [ax]←[[bx]]

5. **Epäsuora rekisterin kautta ja siirtymä (register indirect with displacement)**
   Operandi on osoitteessa, joka saadaan kun rekisterin sisältöön lisätään käskykoodin ns. disp-kentän sisältö.
   TASM syntaksi: *expr*[*r*] tai [*r*+*expr*]
   EA = *expr*+[*r*]
   Esim.
   `mov ax,[bx+1010H]`
   Efekti: [ax]←[1010H+[bx]]

6. **Epäsuora kanta- ja indeksirekisterin kautta (register indirect with base and index register)**
   TASM syntaksi: [r_1][r_2] tai [r_1+r_2]
   r_1 = kantarekisteri = `bx`, `bp`
   r_2 = indeksirekisteri = `si`, `di`
   EA = [r_1]+[r_2]
   Esim.
   `mov [bx+si],ax`
   Efekti: [[bx]+[si]]←[ax]

7. **Epäsuora kanta- ja indeksirekisterin kautta ja siirtymä (register indirect with base + index + constant)**
   Operandin osoite on kanta- ja indeksirekisterien ja käskykoodin disp-kentän sisältöjen summa.
   TASM syntaksi: *expr*[r_1][r_2] tai [*expr*+r_1+r_2]
   r_1 = `bx, bp`
   r_2 = `si, di`
   EA = *expr*+[r_1]+[r_2]
   Esim.
   `mov 1010H[bx][di],ax`
   Efekti: [1010H+[bx]+[di]]← [ax]

16 tapaa ilmaista muistiosoite

Lauseke	Moodi	Lauseke	Moodi
[*expr*]	3	[bp+*expr*]	5
[bx]	4	[bx+*expr*]	5
[si]	4	[si+*expr*]	5
[di]	4	[di+*expr*]	5
[bx+si]	6	[bx+si+*expr*]	7
[bx+di]	6	[bx+di+*expr*]	7
[bp+si]	6	[bp+si+*expr*]	7
[bp+di]	6	[bp+di+*expr*]	7

14.6 Liput (flags)

8086:n *lippurekisterin* yksittäiset bitit, liput, on jaettu kahteen ryhmään:

tilaliput (status flags) ilmoittavat, miten viimeksi suoritettu aritmeettinen tai looginen toiminto päättyi.

kontrolliliput (controll flags) määräävät prosessorin toimintamoodin tietyissä toiminnoissa.

Ohjelmoija ei yleensä suoraan aseta tilalippuja. Ne saavat arvonsa automaattisesti tiettyjen käskyjen (esim.

add) seurauksena. Samoin niiden arvoihin viitataan yleensä epäsuorasti ehdollisilla haarautumiskäskyillä. Prosessorin käynnistyessä kontrolliliput saavat tietyt oletusarvot. Nämä arvot voidaan muuttaa eksplisiittisillä käskyillä.

15. 8086:n ohjelmointi

15.1 TASM-assembleri

Esim.

```
; Kokonaislukujen yhteenlasku
; Vastaa C-ohjelmaa
;   int x = 10,
;       y = 5,
;       z;
;       z = x + y;
;
        DOSSEG          ; DOSin segmentointi
        .MODEL SMALL    ; Pieni malli
        PUBLIC lasku    ; Tämä nimi tunnetaan
                        ; muissakin tiedostoissa
        .DATA
x       DW     10       ; x:n ja
y       DW     5        ; y:n alkuarvot
z       DW     ?        ; Paikka z:lle
; Vaihdetaan koodisegmenttiin
        .CODE
lasku   PROC
        mov    ax,[x]   ; Siirrä x ax:ään
        add    ax,[y]   ; Lisää y
        mov    [z],ax   ; Talleta tulos
        ret             ; Lopeta
lasku   ENDP
        END
```

Osoitelaskuri

- TASM pitää yllä kullekin segmentille omaa osoitelaskuria.
- Osoitelaskurin arvo on se osoite tässä segmentissä, johon parhaillaan assembloitava (tai jollei assembleri ole tekemässä koodia, niin seuraavaksi assembloitava) käsky tai data sijoittuu.
- Osoitelaskurin arvoon voidaan viitata symbolilla `$`.
- `ORG` *direktiivillä* voidaan osoitelaskurille antaa arvo.

Esim.

```
       .DATA
       ORG    $+10  ; Jätä tilaa 10 luvulle
```

Osoitteet

- Jokaista käskyä voi edeltää osoite, joka on muotoa *tunnus*:
- Osoitteen (tunnuksen) arvona on osoitelaskurin sen hetkinen arvo. Esim.

```
          .CODE
          ORG     120H
alku:     mov     ax,bx  ; alku  = 120H
loppu:    mov     cx,bx  ; loppu = 122H
```

- Osoite voidaan määritellä vain `cs`-rekisteriin liittyvissä segmenteissä.

- Osoitteen arvoa ei saa määrittelyn jälkeen muuttaa.
- Osoite voi olla myös yksinään käskyä edeltävällä rivillä.

Direktiivit

- Direktiivit antavat ohjeita assemblerille, joten ne vaikuttavat vain assemblauksen aikana.
- Direktiivit on tapana kirjoittaa ISOILLA kirjaimilla.

Osoitelaskurin ohjaus
`ORG` *expr*
Asetetaan osoitelaskurin arvo.
`EVEN`

Siirtää osoitelaskurin seuraavan sanan rajalle (parilliseen osoitteeseen).

Segmentointi
`.DATA`
Assemblaus datasegmenttiin.
`.CODE`

Assemblaus koodisegmenttiin.
`.STACk` *koko*

Varataan tilaa pinosegmentille *koko* tavua.
`DOSSEG`

DOSin käyttämä segmentointi
`.MODEL` *muistimalli*

Käytettävä *muistimalli*: `TINY`, `SMALL` (oletus), `MEDIUM`, `COMPACT`, `LARGE` tai `HUGE`.

Proseduurimäärittelyt

nimi	`PROC`	*etäisyys*
	⋮	ohjelman käskyt
nimi	`ENDP`	

- `PROC` ja `ENDP` direktiivien ainoa vaikutus koodiin on se, miten niiden välissä mahdollisesti esiintyvä paluukäsky (`ret`) koodataan.
- Jos *etäisyys* on `NEAR` (oletus), niin `ret` käyttää kaksitavuista paluuosoitetta ja neljätavuista, jos se on `FAR`.
- Muutoin näiden direktiivien tehtävänä on auttaa ohjelmoijaa kirjoittamaan hyvin jäsenneltyä (rakenteellista) koodia.

Datamäärittelyt
Datamäärittelydirektiiveillä voidaan assemblata dataa tai varata tilaa datalle osoitelaskurin ilmoittamaan kohtaan muistissa. Muistiosoitteille voidaan samalla antaa symboliset nimet.
[*nimi*] `DB` *alkuarvo,...*

Tavujen assemblaus. *Alkuarvo* voi olla

`'a'`	ASCII merkki
`'`*merkkijono*`'`	ASCII merkkijono
`16`	8-bittinen kokonaisluku
`?`	määräämätön
n`DUP(`*alkuarvo*`)`	*n* kpl *alkuarvo*ja
n`DUP(?)`	tilaa *n*:lle tavulle

[*nimi*] `DW` *alkuarvo,...*

Sanojen assemblaus. *Alkuarvo* kuten `DB` direktiivillä, mutta 16-bittinen.
[*nimi*] `DD` *alkuarvo,...*

Kaksoissanojen assemblaus.

Symbolit
nimi `=` *expr*

- *nimi* esittää 16 bittistä kokonaislukua, jonka arvo on *expr*.
- *nimi* voidaan määritellä uudelleen.

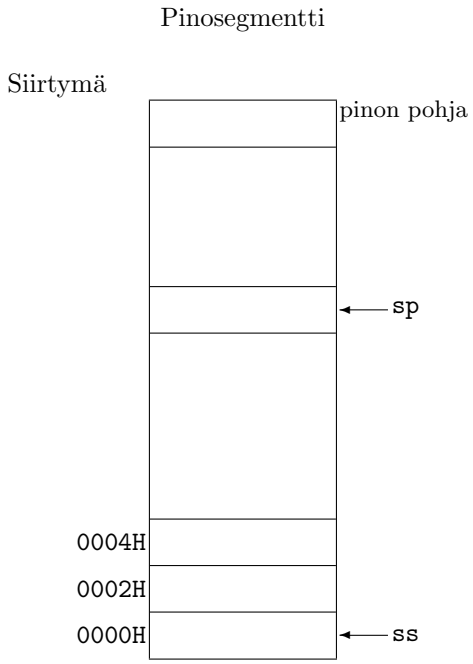
nimi `EQU` *expr*
expr on

- 16 bittinen kokonaisluku.
- symboli.
- merkkijono.

Esim.
```
ten    EQU   10
ptr    EQU   [bp]
clear  EQU   xor ax,ax
```
Nimeä ei voi määritellä uudelleen.
nimi `LABEL` *tyyppi*

- *nimen* arvo on osoitelaskurin tämän hetkinen arvo.
- *tyyppi* on `BYTE`, `WORD`, `DWORD`, `NEAR` tai `FAR`.

15.2 Pino ja aliohjelmat



- `ss` osoittaa pinosegmentin alkuun.
- `sp` osoittaa pinon huippua (top).
- pino kasvaa alaspäin, kohti pienempiä osoitteita.

Pinotoiminnot
Pinoa käsitellään vain sanan mittaisina yksikköinä.
`push` (datan lisäys)

Siirretään rekisterin tai muistipaikan sisältö pinon huipulle.
Esim.
`push ax`
Efekti: [sp]←[sp]-2, [[sp]]←[ax]
`pop` (datan poisto)

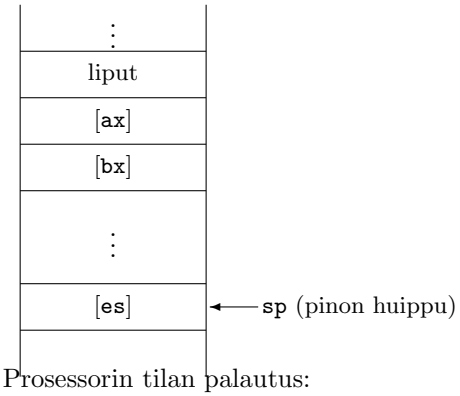
Siirretään pinon huipulla oleva data rekisteriin tai muistipaikkaan.
Esim.
`pop ax`
Efekti: [ax]←[[sp]], [sp]←[sp]+2

Huom. `sp` osoittaa aina viimeksi huipulle siirrettyyn sanaan.

Pinoa käytetään mm. datan väliaikaiseen talletukseen.
Esim. Prosessorin tilan talletus:

```
pushf          ; Talleta liput
push  ax       ; Talleta rekisterit
push  bx
push  cx
push  dx
push  bp
push  di
push  si
push  ds
push  es
```

Pinon sisältö on nyt



Esim. Prosessorin tilan palautus:

```
pop  es        ; Palauta rekisterit
pop  ds        ; Huom.\ käänteinen
pop  si        ; järjestys verrattuna
pop  di        ; talletukseen
pop  bp
pop  dx
pop  cx
pop  bx
pop  ax
popf           ; Palauta liput
```

Huom. Ennen tilan palautusta on huolehdittava siitä, että `sp` osoittaa siihen kohtaan, mihin se jäi tilaa talletettaessa.

Esim. `ax` ja `bx` rekisterien sisällön vaihto:

```
push  ax       ; ax talteen
mov   ax,bx    ; ax:lle uusi arvo
pop   bx       ; bx:lle ax:n entinen arvo
```

Huom. On olemassa käsky, `xchg`, jolla voidaan suoraan vaihtaa rekisterien (tai muistipaikan ja rekisterin) sisällöt keskenään.

Pinon dataan voidaan viitata muillakin kuin `push` ja `pop` toiminnoilla, ts. kasvattamatta tai pienentämättä pinoa: Koska `bp` rekisteriä käytettäessä oletussegmentti on `ss`, tavallisin menetelmä on siirtää `sp`:n sisältö `bp`:hen. Esim. pinon toiseksi päällimmäisin sana saadaan `ax` rekisteriin toiminnoilla

```
mov  bp,sp
mov  ax,2[bp]
```

Aliohjelmat
Eksplisiittisten pinotoimintojen (`push`, `pop`, `mov`, ...) sen sisältöön ja kokoon vaikuttavat implisiittisesti myös aliohjelmien kutsut ja niistä paluu.
Aliohjelmaan siirrytään käskyllä:

`call` *a*
Efekti: [sp]←[sp]-2, [[sp]]←[ip], [ip]←EA
Seuraavan käskyn osoite työnnetään pinoon ja suoritus jatkuu operandin *a* ilmoittamasta osoitteesta.
Aliohjelmasta palataan käskyllä:

`ret`
Efekti: [ip]←[[sp]], [sp]←[sp]+2

Suoritus jatkuu pinon päällimmäisenä olevasta osoitteesta.
Koska 8086 operoi pinoon implisiittisesti, sisäisesti, sanotaan, että se on ns. *pinokone* (*stack machine*). Mainframe- ja RISC-prosessorit ovat yleensä ns. *rekisterikoneita*: mm. aliohjelmien paluuosoitteet tallettuvat johonkin suorittimen rekisteriin.
Esim.

```
; 32 bittisten kokonaislukujen yhteenlasku
; Operandit rekisteriparessa ax:bx ja cx:dx
; Tulos rekisteriparissa ax:bx
sum32   PROC
        add  ax,cx ; Vähiten merkitsevät sanat
        adc  bx,dx ; Eniten merkitsevät+carry
        ret
sum32   ENDP
        ...
        mov  ax,10
        mov  bx,1  ; [ax:bx]=65546
        mov  cx,25
        mov  dx,2  ; [cx:dx]=131097
        call sum32 ; [ax:bx]=196643
        ...
```

Huom. Ennen `ret` käskyä on huolehdittava siitä, että `sp` osoittaa samaan kohtaan, kuin ohjelmaan tultaessa.

Aliohjelmille voidaan välittää parametreja mm.

- rekistereissä (kuten yo. esimerkissä)
- rekistereissä osoitin parametrialueelle
- pinossa

15.3 PC C:n ja assemblykielen liittymä

Kutsuvalle C-ohjelmalle palautetaan funktion arvot seuraavasti:

Funktion tyyppi	Arvo rekisterissä
`char`	`al`
`int`	`ax`
`long`	`ax:dx` (low:high)
osoitin (`*`)	`ax`

Aliohjelmalle parametrit välitetään pinossa. Kutsuva ohjelma työntää argumentit pinoon alkaen oikenpuoleisimmasta argumentista siten, että:

Argumentin tyyppi	Tilanvaraus pinossa
`char`	laajennetaan `int` tyyppiseksi
`int`	sana
`long`	ensin eniten merkitsevä ja sitten vähiten merkitsevä
osoitin (`*`)	sana

Osoitin on joko siirtymä (offset) datasegmentissä tai koodisegmentissä, mikäli kyseessä on osoitin funktioon. `main()` ohjelman käynnistyessä on [ss]=[ds]=[es].

Huom. PC C lisää ulkoisten tunnusten alkuun alleviivauksen (_).

Esim.

```
int i;
long l;
char c, *cp;
...
f( i, l, c, cp );
```

Aliohjelmaan `_f` tultaessa pinon sisältö on

Osoite		
	⋮	
[sp]+10	cp	
[sp]+8	c	
[sp]+6	l-high	
[sp]+4	l-low	
[sp]+2	i	
[sp]	paluuosoite	←sp

Palattaessa C:n aliohjelmista täytyy segmenttirekistereillä ja rekistereillä `si`, `di` ja `bp` olla alkuperäiset arvot.
Esim.

```
        PUBLIC   _f
_f      PROC     NEAR
        push     si     ; Talleta säilytettävät
        push     di     ; rekisterit pinoon
        push     bp
        mov      bp,sp
;
;   Pinon sisältö on nyt
;    Osoite       Sisältö
;    [bp]+16        cp
;    [bp]+14        c
;    [bp]+12        l-high
;    [bp]+10        l-low
;    [bp]+8         i
;    [bp]+6         paluuosoite
;    [bp]+4         si
;    [bp]+2         di
;    [bp]           bp
;
        ...
        mov      bx,16[bp] ; Siirrä cp bx:ään
        mov      al,[bx]   ; cp:n osoittama
                           ; merkki al:ään
        ...
        mov      sp,bp
        pop      bp        ; Palauta rekisterit
        pop      di
        pop      si
        ret
_f      ENDP
```

15.4 DOS:n funktiot

DOS käyttöjärjestelmä sisältää lukuisia ohjelmoijan käytettävissä olevia funktioita. Niillä voidaan käsitellä mm.

- laite I/O (näppäimistö, näyttö, ...)
- tiedostoja
- hakemistoja
- muistia
- muut (esim. CTRL-Break:n tarkistus, levyn vapaa tila, ...)

Funktioita käytetään seuraavasti:

1. Jos funktio tarvitsee argumentteja, siirrä ne yksilöityihin rekistereihin.
2. Siirrä funktion tunnusnumero `ah` rekisteriin.
3. Anna käsky `int 21H`.
4. Monet funktiot palauttavat virhekoodin: jos `CF`=0, niin OK, muutoin `ax` rekisteri sisältää tarkemman virhekoodin.
5. DOSin funktiot säilyttävät rekisterit.

Esim.

```
        .DOSSEG
        .DATA
msg     DB      'Heippa$'
        .CODE
_terve  PROC
        mov     dx, OFFSET msg
        mov     ah, 9
        int     21H
        ret
_terve  ENDP
        END
```

Huom. Direktiivi `OFFSET` laskee operandinsa (tässä `msg`) siirtymän operandin sisältävässä segmentissä (tässä `.DATA` eli `ds`). Muodostuva osoitusmoodi on välitön.

15.5 8086:n tärkeimmät käskyt

Datan siirtokäskyt

`mov`
[*r*]←[*r*], [*r*]←[*mem*], [*mem*]←[*r*], [*r*]←immed. data
`push`

`pushf`
[pino]←[*r*], [pino]←[*mem*]
`pop`

`popf`
[*r*]←[pino], [*mem*]←[pino]
`lahf`

[`ah`]←liput
`sahf`

liput←[`ah`]
`xchg`

[*mem*/*r*]↔[*mem*/*r*], [`ax`]↔[*r*]

Aritmeettiset käskyt

`add`
[mem_1/r_1]← [mem_1/r_1]+[mem_2/r_2]
[mem_1/r_1]← [mem_1/r_1]+immed. data
`adc`

kuten `add` + [`C`]
`inc`

[*mem*/*r*]←[*mem*/*r*]+1
`sub`

[mem_1/r_1]← [mem_1/r_1]-[mem_2/r_2]
[mem_1/r_1]← [mem_1/r_1]-immed. data
`sbb`

kuten `sub` - [`C`]
`dec`

[*mem*/*r*]←[*mem*/*r*]-1
`neg`

[*mem*/*r*]← -[*mem*/*r*]

Vertailukäskyt

`cmp`
[mem_1/r_1]-[mem_2/r_2]
[mem_1/r_1]-immed. data
asetetaan ainoastaan liput

Loogiset käskyt

`and`
`or`
`xor`
[mem_1/r_1]← [mem_1/r_1] *op* [mem_2/r_2]
[mem_1/r_1]← [mem_1/r_1] *op* immed. data
`not`

[*mem*/*r*]←[*mem*/*r*]:n 1:n komplementti
`test`

[mem_1/r_1] AND [mem_2/r_2]
[mem_1/r_1] AND immed. data
asetetaan ainoastaan liput

Ehdottomat hyppykäskyt

`call` *d*
pino←paluuosoite, [`ip`]←*d*
`jmp` *d*

kuten `call`, mutta paluuosoitetta ei talleteta.
`ret`

[`ip`]←pino

Ehdolliset hyppykäskyt

`j`*cond d*

- hypätään osoitteeseen *d*, mikäli *cond* on tosi.
- osoite *d* voi olla korkeintaan 127 tavun päässä ko. käskystä.

- katso *cond* koodit taulukosta.

Esim.
`jnz d`
hyppää osoitteeseen `d`, mikäli `Z`-lippu on 0.
Esim.

```
; Datan siirto
size    EQU     2000
        .DATA
sour    DW   size DUP(?)     ; Lähde
dest    DW   size DUP(?)     ; Kohde
        .CODE
        mov  si, OFFSET sour ; Indeksi-
        mov  di, OFFSET dest ; rekisterit
        mov  cx, size        ; Laskuri
move:   mov  ax, [si]        ; Siirrä sana
        mov  [di], ax
        inc  si              ; Indeksit
        inc  si              ; seuraavaan
        inc  di              ; sanaan
        inc  di
        dec  cx              ; Data loppu?
        jnz  move
```

`loop` *d*
[cx]←[cx]-1, jos cx≠0, niin hyppää *d*:hen.
`loope` *d*

`loopz` *d*
kuten `loop` ja jos [ZF]=1, niin hyppää.
`loopne` *d*

`loopnz` *d*
kuten `loope`, mutta ehtona [ZF]=0.

I/O-käskyt

`in ax,` *p*
`in al,` *p*
[ax/al]←portissa *p* oleva data.
0≤*p*≤255
`in ax, dx`

`in al, dx`
[ax/al]←portissa, jonka osoite on [dx], oleva data.
`out` *p*, `ax`

`out` *p*, `al`
`out dx, ax`
`out dx, al`
käänteinen `in`-käskylle.

Siirto- ja rotaatiokäskyt

`rcl` *a*, `1`
rotatoi (kierrä) carryn kautta vasemmalle 1 bitti.
`rcr`

kuten `rcl`, mutta oikealle.
`rol`

rotatoi (kierrä) vasemmalle, korkein bitti `CF`:ään.
`ror`

kuten `rol`, mutta oikealle.

`shr`

siirrä oikealle ja ylimpään bittiin 0.
`shl`

`sal`
siirrä vasemmalle ja alimpaan bittiin 0.
`sar`

siirrä oikealle ja säilytä ylin bitti.
Toinen muoto:

op a, `cl`
`cl`:ssä siirtojen lukumäärä.

Blokkitoiminnot

`movs`
siirrä tavu tai sana osoitteesta `ds:si` osoitteeseen `es:di`.
$[\texttt{si}] \leftarrow [\texttt{si}] \pm \begin{cases} 1, & \text{b} \\ 2, & \text{w} \end{cases}$
$[\texttt{di}] \leftarrow [\texttt{di}] \pm \begin{cases} 1, & \text{b} \\ 2, & \text{w} \end{cases}$
+, jos [DF]=0,
-, jos [DF]=1
`stos`

[ax] tai [al] osoitteeseen `es:di`.
$[\texttt{di}] \leftarrow [\texttt{di}] \pm \begin{cases} 1, & \text{b} \\ 2, & \text{w} \end{cases}$
`rep`

jos näitä *primitiivejä* (`movs`, `stos`, ...) edeltää käsky `rep`, niin toistetaan niin kauan kuin [cx]≠0 ja [cx]←[cx]-1.

Prosessorin kontrollikäskyt

`clc`
nollaa carry.
`stc`

aseta carry.
`cmc`

komplementoi carry.
`cld`

nollaa `DF`.
`std`

aseta `DF`.
`cli`

nollaa `IF`.
`sti`

aseta `IF`.
`nop`

ei toimintoa.

16. Keskeytykset (interrupts)

16.1 Keskeytyskäsite

Keskeytyksen tarkoitus on kertoa prosessorille, että syystä tai toisesta sen on lykättävä normaalia prosessointia ja siirryttävä palvelemaan keskeytyksen aiheuttajaa.
Esim. Lämpötilan säätö

Mikroon on kytketty anturi, joka antaa signaalin kun lämpötila ylittää tai alittaa tietyt kriittiset rajat ja servo, joka tietokoneen käskystä vääntää venttiiliä auki tai kiinni.

- Säätöjä tehdään harvoin⇒prosessori on suurimman osan ajasta käytettävissä muuhun tarkoitukseen.
- Kriittisten rajojen ylitystä ei voida ennakoida⇒ei voida kirjoittaa muita ohjelmia siten, että ne tietyin väliajoin korjaavat säätöä.
- Rajojen ylitys vaatii välitöntä korjausta⇒säädöt on hoidettava kesken muuta prosessointia.

Ohjelmoidaan mikro siten, että anturin antaessa signaalin keskeytetään normaali prosessointi, hoidetaan säätö ja jatketaan normaalia prosessointia.
Keskeytyksiä käytetään

- ennakoimattomien tapausten käsittelyyn.
- välitöntä vastausta vaativaan prosessointiin.

16.2 Keskeytystyypit

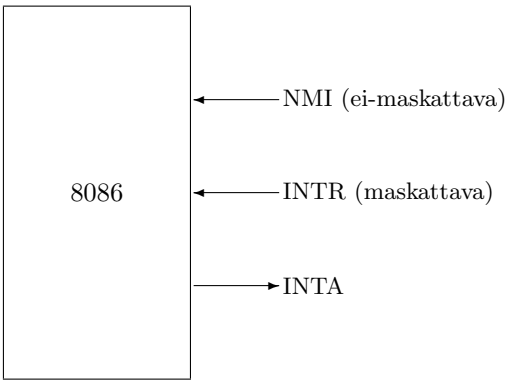
Maskattavat keskeytykset (maskable interrupts)
Voidaan kieltää tai sallia asettamalla keskeytysmaski

`sti` asettaa `IF`:n, sallii keskeytykset.

`cli` resetoi (nollaa) `IF`:n, kieltää maskattavat keskeytykset.

Ei-maskattavat keskeytykset (Non Maskable Interrupts, NMI)
Ei voida kieltää.

Ulkoiset keskeytykset
NMI- ja INTR-keskeytykset.



Sisäiset keskeytykset
`int` n
tuottaa n-tyyppisen keskeytyksen; ei voida kieltää.

16.3 Keskeytysmekanismit

INTR-keskeytys (maskattava)

1. Ulkoinen laite asettaa signaalin INTR-langalle.
2. Prosessori vastaa INTA-signaalilla.
3. Ulkoinen laite asettaa osoiteväylälle 8-bittisen luvun n.
4. Prosessori lukee osoitteesta $4n$ sanan *offs* ja osoitteesta $4n + 2$ sanan *seg*.
5. Prosessori työntää pinoon statussanan,
6. asettaa [`IF`]←0 (kieltää keskeytykset),
7. työntää pinoon `cs` ja `ip` rekisterit ja
8. jatkaa suoritusta osoitteesta *seg*:*offs*.

n on ns. keskeytystyyppi. Osoitteessa $4n$ olevaa *seg*:*offs* osoitetta sanotaan keskeytysvektoriksi.

Sisäinen keskeytys

1–3 Käsky `int` n generoi tyypin n.

4–8 Kuten edellä.

Keskeytetyn ohjelman jatkaminen
`iret`
palauttaa pinosta `ip`:n, `cs`:n ja statuksen.

16.4 Keskeytyspalveluohjelmat

Keskeytyspalveluohjelmien tulee noudattaa sekvenssiä

1. Salli maskattavat keskeytykset, ellei ole mitään erikoista syytä niiden kieltämiseen.
2. Talleta prosessorin tila pinoon.
3. Mikäli pino on liian pieni, vaihda uusi pino.
4. Varmista, että segmenttirekistereillä (`ds` ja `es`) on oikeat arvot.
5. Suorita keskeytyspalvelut.
6. Palauta prosessorin tila (mukaan lukien alkuperäinen pino, jos tarpeen).
7. `iret`.

Keskeytyspalveluohjelmaa kirjoitettaessa on huomioitava

- Mikäli keskeytyspalvelu käyttää aliohjelmia, jotka voivat olla samanaikaisesti muidenkin ohjelmien käytössä, on nämä kirjoitettava ns. *monikäyntiohjelmiksi* tai varmistuttava siitä, että näitä ohjelmia ei voida keskeyttää.

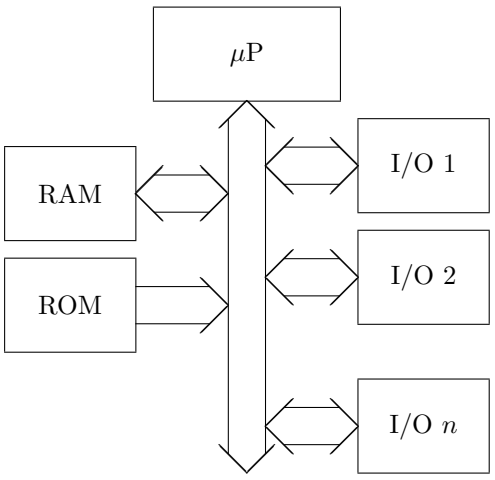
- Useimmat DOSin funktiot *eivät ole* monikäyntiohjelmia.
- Monikäynti- (*re-entrant*)-ohjelma on sellainen, että se voi olla samalla hetkellä useamman käyttäjän suoritettavana (esim. C:n rekursiiviset funktiot).
- Mikäli keskeytyspalveluohjelma käsittelee globaalia dataa, on varmistuttava tämän datan integriteetistä.

C-ympäristössä varsinainen palveluohjelma voidaan kirjoittaa C-kielellä. Jotta ohjelman suoritus saataisiin siirtymään palveluohjelmaan ja vastaavasti takaisin keskeytettyyn ohjelmaan, tarvitaan

1. Prologi, joka tallettaa rekisterit ja kutsuu C-ohjelmaa.
2. Epilogi, joka palauttaa rekisterit ja palaa keskeytettyyn ohjelmaan.
3. Ohjelma, joka asettaa keskeytysvektorin osoittamaan prologiin (TURBO C:ssä `void setvect( int, void interrupt(*)() )`).

16.5 Ulkoiset keskeytykset

Kytketään systeemiin useita ulkoisia laitteita:

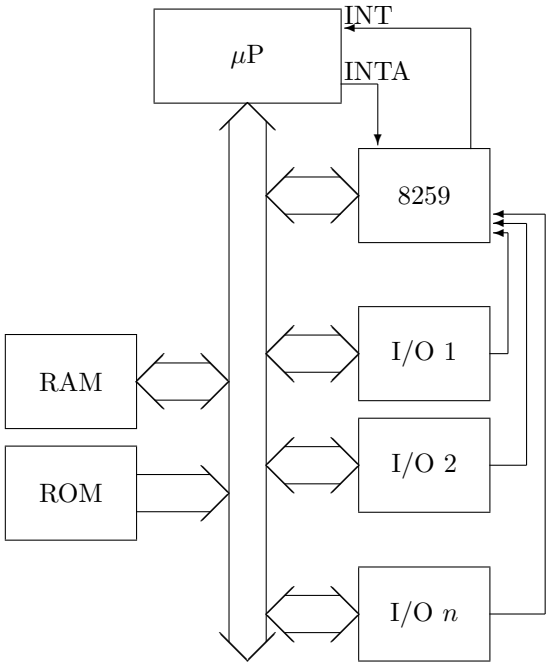


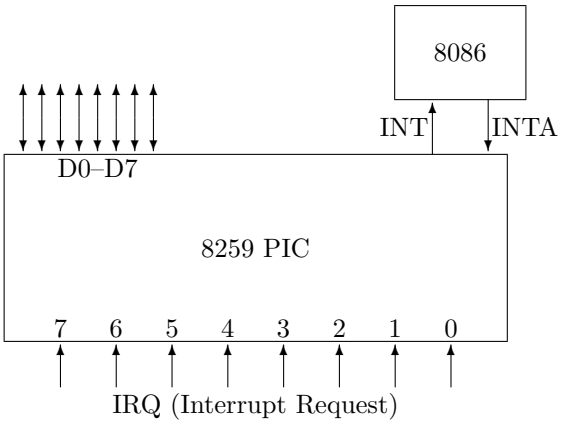
Näitä laitteita voidaan palvella

pollaamalla: prosessori "kysyy" kultakin laitteelta vuoronperään, tarvitseeko se palvelua.

keskeyttämällä: prosessori suorittaa normaalia ohjelmaa ja palvelua tarvitseva laite aiheuttaa keskeytyksen.

Koska 8086 suorittimessa on vain yksi INTR-sisäänmeno, tarvitaan ulkoisten keskeytysten erotteluun ja priorisointiin ylimääräinen mikropiiri. IBM PC:ssä (ja klooneissa) nämä hoidetaan 8259 PIC- (Programmable Interrupt Controller-) piirillä.





Keskeytyssekvenssi on

1. Ulkoinen laite asettaa signaalin IRQ-langalle.
2. PIC asettaa INT-signaalin.
3. CPU vastaa INTA-signaalilla.
4. PIC asettaa dataväylälle luvun $b + i$, missä b on PIC:iin ohjelmoitu kantaosoite (DOSissa 8) ja i on IRQ-langan numero.
5. CPU suorittaa keskeytyspalvelun. Tänä aikana alempiprioriteettinen (IRQ $\geq i$) keskeytys ei ole sallittu.
6. CPU kuittaa keskeytyksen kirjoittamalla PIC:n rekisteriin. Tämä vapauttaa alempiprioriteettiset keskeytykset.
7. CPU palaa keskeytettyyn ohjelmaan (`iret`).

8259:n ohjelmoinnista

Alustus Kirjoitetaan 4 ns. ICW:tä (Initialization Command Word) PIC.n rekistereihin. Alustus tehdään (normaalisti) konetta käynnistettäessä.

Operointi

Keskeytysten maskaus (OCW1, Operation Command Word)
Kirjoitetaan PIC;n rekisteriin (PC:ssä portti `21H`) 8-bittinen tavu:

- bitti = 0, vastaava IRQ sallittu.
- bitti = 1, vastaava IRQ kielletty.

Tämä rekisteri voidaan myös lukea.

Kuittaus (OCW2)
Kirjoitetaan PIC:n rekisteriin (PC:ssä portti `20H`) 8-bittinen tavu `20H`.

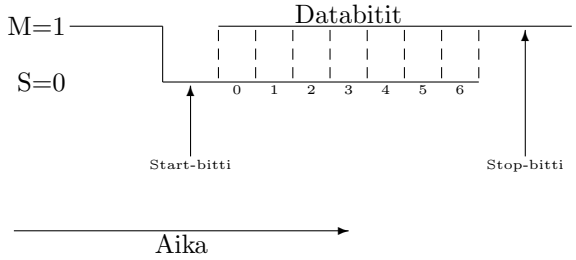
16.6 Sarjamuotoinen I/O

Sarjamuotoinen I/O on eräs tärkeimmistä PC:n ja ulkoisen maailman välisistä tiedonsiirtotavoista. Tiedonsiirtoon liittyviä käsitteitä ovat mm.

rinnakkaisliitäntä Siirretään kerrallaan (8- tai 16-bittinen) sana.

sarjaliitäntä Sanan bitit siirretään peräkkäin.

asynkroninen sarjaliitäntä



M=mark, S=space

Tavallisimmin 1 start-bitti, 8 data-bittiä ja 1 stop-bitti.

baudimäärä (*Baud rate*) Siirrettävien bittien (start+data+stop) määrä sekunnissa. Tyypillisesti 300–19200 baud.

start-bitti Kun vastaanottaja tunnistaa start-bitin, se käynnistää oman baudigeneraattorinsa käymään sovitulla nopeudella.

data-bitti Baudigeneraattorin antaessa pulssin vastaanottaja tunnistelee datalankaa ja määrää bitin arvon (0 tai 1).

stop-bitti Kun sovittu määrä data-bittejä on lähetetty, siirtää lähettäjä signaalin mark-tasolle, jotta vastaanottaja voisi tunnistaa seuraavan start-bitin.

synkroninen sarjaliitäntä Lähetetään samanaikaisesti sekä data-bitti että tahdistava kellopulssi.

pariteetti Voidaan sopia, että välitettävässä (tavussa tai) sanassa 1 data-bittien määrä on parillinen tai pariton. Tämä toteutetaan lisäämällä ylimääräinen 0- tai 1-bitti viimeisen data-bitin ja stop-bitin väliin. Yleensä data-bittien (varsinainen data + pariteetti) kokonaislukumäärä on 8.

protokolla Sopimus tiedonsiirtotavasta. Teknisellä tasolla se käsittää mm. sopimukset

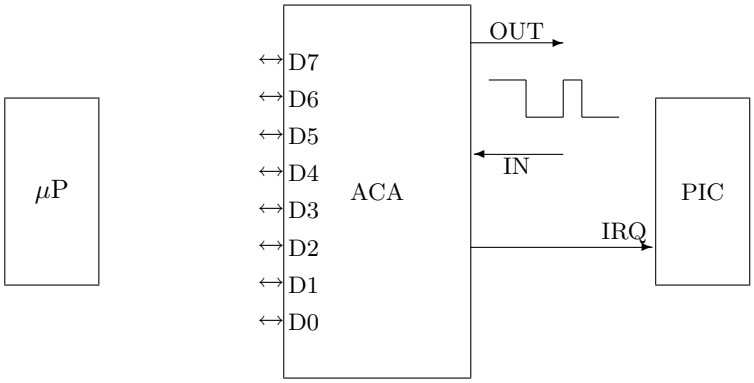
- bittien koodauksesta,

- sarja/rinnakkais-liikenteestä,
- synkronisesta/asynkronisesta liitännästä,
- ...

Tekninen taso (esim. RS232 standardi) hoidetaan (yleensä) kovolla.

Esim. IBM PC:ssä on piiri (Asynchronous Communications Adapter), joka

- muuntaa sisäisen rinnakkaismuotoisen tiedon sarjamuotoiseksi.
- generoi start- ja stop-bitit.
- tarkistaa mahdollisen pariteetin.
- generoi tarvittaessa keskeytyspyynnön.



17. Graafiset käyttäjäliittymät (GUI)

17.1 Asiakas-palvelinmalli

Lähes kaikki (X, Windows, PM, ...) nykyiset graafiset käyttäjäliittymät (*graphical user interface, GUI*) perustuvat asiakas-palvelin(*client-server*)-ohjelmointimalliin:

- *asiakas* on varsinainen sovellusohjelma.
- *palvelin* on ohjelmisto, joka huolehtii kommunikoinnista (näyttö, näppäimistö, hiiri, ...) käyttäjän kanssa.

Sovellusohjelma, asiakas, kytkeytyy palvelimeen API:n (application program interface) kautta:

- asiakkaan pyynnöt (`avaa ikkuna`, `kirjoita ikkunaan`, `sulje ikkuna`, ...) välitetään palvelimelle tavallisesti funktiokutsuilla.
- palvelin ilmoittaa ympäristön tapahtumista asiakkaalle tai vaatii asiakkaalta tiettyjä toimenpiteitä *viestien* (*messages*) välityksellä.

Fyysisesti palvelin suorittuu käyttäjän koneessa. Asiakas sen sijaan voi olla jopa toisessa koneessa (X). Tällöin palvelinkoneena voi olla pelkkä älykäs pääte (X-pääte). Monissa tapauksissa kannattaa asiakas-palvelinmallia soveltaa myös sovellusohjelmaan. Eräs luonnollinen jako on

- asiakas on sovelluksen käyttäjäliittymästä riippumaton osa.
- palvelimen muodostaa sovelluksen ja GUI:n liittymäpinta.

Tämä jako mahdollistaa suhteellisen helpon siirrettävyyden GUI-ympäristöstä toiseen. Mikäli käyttöjärjestelmä sallii, kannattaa asiakas- ja palvelinosat suorittaa eri prosesseina tai säikeinä (*thread*). Tällöin asiakas voi käyttäjän näppäinten painallusten ja hiiren liikuttelun aikana tehdä esim. pitkiä laskutoimituksia.

17.2 Ikkunointi

Lähes jokainen ruudulla näkyvä olio on ikkuna:

- ikkunan *asiakasalue*, ts. alue johon asiakasohjelma kirjoittaa tai piirtää, on ikkuna.
- ikkunan kehys koostuu useista ikkunoista: vierityspalkit (*scroll bar*), minimointi- ja maksimointinappulat, systeemimenu, menupalkki, otsikkopalkki, ..., ovat kukin omia ikkunoitaan.
- ikonit ovat ikkunoita.

Käyttäjä ei välttämättä edes tiedosta kaikkia ikkunoita: asiakasohjelma voi esim. jakaa asiakasalueensa ikkunoihin, joiden kehykset ovat näkymättömiä.

Ikkunat ovat hierarkisessa vanhempi–lapsi-relaatiossa toisiinsa nähden. Lapset eivät voi esim. siirtyä vanhempiensa asiakasalueen ulkopuolelle.
Ikkunointisysteemit noudattavat olioperustaista ohjelmointimallia: jokainen ikkuna on jonkin abstraktin ikkunaluokan olio.

- Ikkunointisysteemeissä on yleensä joukko *esirekisteröityjä* ikkunaluokkia. Esim. ikkunan kehykset, vierityspalkit, nappulat (*push buttons*) ja dialogi-ikkunat (*dialog boxes*) ovat systeemiin sisään rakennettuja ikkunaluokkia.
- Pääsääntöisesti asiakkaan on luotava, *rekisteröitävä* ikkunointisysteemiin, sellaiset ikkunaluokat, jonka olioihin se aikoo piirtää.

Ikkunointisysteemit eivät kuitenkaan toteuta kaikkia OOP:n ominaisuuksia: mm. ikkunaluokkia ei voi johtaa perimällä aikaisemmista luokista. Kaupallisesti on saatavana C++-kirjastoja (esim. Borlandin Object Windows Library, OWL, tulee kääntäjän mukana), joissa on useita valmiita C++:lla toteutettuja ikkunaluokkia. Näistä ohjelmoija voi tietenkin johtaa omia luokkiaan.
Rekisteröitäessä ilmoitetaan luokan ikkunoiden yleiset ominaisuudet, esim. minimoitaessa näytettävä ikoni. Ikkunaluokan olion muodostamista sanotaan ikkunan luonniksi. Ikkuna saa luotaessa luokan ominaisuudet. Sen lisäksi voidaan määritellä kullekin ikkunalle sille ominaisia piirteitä, kuten ikkunan koko, otsikkopalkissa näkyvä ikkunan nimi, jne.
Ikkunointisysteemin kannalta asiakas, sovellus, on myös luokka ja suorittuva sovellus asiakasluokan olio. On täysin sallittua muodostaa sama sovellus useampaan kertaan, ts. samasta ohjelmasta voi olla useampi kopio suorittumassa samaan aikaan.

17.3 Tapahtumaohjaus ja viestit

Perinteisen komentoriviliittymäisen (*command line interface, CLI*) ohjelman suoritus etenee ohjelmaan kirjoitetun logiikan mukaisesti. Kärjistäen voisi sanoa, että CLI-käyttäjä joutuu toimimaan ohjelman ehdoilla.
GUI-ohjelmat toimivat CLI-ohjelmiin verrattuina tavallaan täsmälleen päinvastoin: käyttäjän kunkin hetkiset toimenpiteet vaikuttavat ohjelman suoritukseen eli ohjelma toimii käyttäjän ehdoilla.
Käyttäjäehtoisuus on ikkunointisysteemeissä toteutettu ns. tapahtumaohjauksella (*event driving*):

- jokainen käyttäjän toimi (näppäimen painallus, hiiren liike, ...) saa ikkunointisysteemin muodostamaan jonkin tapahtuman.
- sovelluksen on periaatteessa joka hetki oltava valmiina reagoimaan kyseisiin tapahtumiin.

Ikkunointisysteemi tiedottaa asiakkaalleen tapahtumista viestien muodossa. Nämä viestit päätyvät (enimmäkseen) loppujen lopuksi ns. *ikkunaproseduurille*.
Jokaiseen ikkunaan liittyy ikkunaproseduuri, jonka tehtävänä on vastata viesteihin. Esirekisteröityjen

ikkunaluokkien ikkunaproseduurit ovat valmiina ikkunointisysteemissä. Muiden luokkien ikkunaproseduurit on ohjelmoijan kirjoitettava (ohjelmoija voi halutessaan kirjoittaa myös esirekisteröityjen luokkien proseduurit). GUI:n ohjelmointi onkin pääasiassa näiden proseduurien kirjoittamista.
Viestit voidaan jakaa kahteen ryhmään:

- *Synkroniset* viestit ovat yleensä palvelimen tekemiä (kyselyt, ilmoitukset minimoinnista, maksimoinnista, sulkemisesta, ...). Palvelin *lähettää* (*send*) viestin asiakkaalleen (ikkunaproseduurille) ja jää odottamaan asiakkaansa vastetta.
- *Asynkroniset* viestit aiheutuvat yleensä käyttäjän toimenpiteistä (näppäimistö, hiiren liike, hiiren nappulat, ikkunan tai sen osan paljastuminen, ...). Palvelin *postaa* (*post*) viestin asiakkaalleen, mutta ei odota vastausta.

Myös sovellus voi tuottaa itselleen tai muille sovelluksille tarkoitettuja viestejä. Tätä varten APIssa on postaus- ja lähetyskutsut. Viestien lopullinen kohde on yleensä jokin ikkuna (ikkunaproseduuri).
Viestien asynkroninen käsittely toteutetaan ns. *viestijonon* (*message queue*) avulla:

- Ensimmäisinä toimenpiteinään asiakas luo itselleen viestijonon (moniajosysteemeissä mahdollisesti useampiakin jonoja). Joissakin ikkunointijärjestelmissä (esim. Windows) palvelin tekee jonon automaattisesti.
- Sovellus aloittaa ns. *viestisilmukan* (*message loop*) suorituksen:
  - sovellus poimii jonosta ensimmäisen viestin (on mahdollista poimia myös muualtakin kuin jonon kärjestä).
  - mahdollisesti muokkaa viestin sopivampaan muotoon.
  - palauttaa viestin palvelimelle, joka lähettää sen kohdeikkunalle.

  Sovellus pysyy viestisilmukassa niin kauan kuin viimeinenkin sovelluksen ikkuna on hengissä.

17.4 Windows-ympäristö

Kirjoitetaan ohjelma, joka avaa ikkunan ja tulostaa siihen tekstin "Heippa!".

```
// heippa.cpp
#include <windows.h>

LRESULT CALLBACK _export
winProc( HWND hWnd, UINT iMsg,
         WPARAM wPar, LPARAM lPar );

static char szAppName[] = "Eka Win ohjelma";

int PASCAL
WinMain( HINSTANCE hInst, HINSTANCE hPrevInst,
         LPSTR lpszCmdLine, int nCmdShow )
{
  HWND hWnd;
  MSG msg;
  WNDCLASS wndClass;
  if( !hPrevInst ) {
    wndClass.style = CS_HREDRAW | CS_VREDRAW;
    wndClass.lpfnWndProc = winProc;
    wndClass.cbClsExtra = 0;
    wndClass.cbWndExtra = 0;
    wndClass.hInstance = hInst;
    wndClass.hIcon = LoadIcon( NULL,
                      IDI_APPLICATION );
    wndClass.hCursor = LoadCursor( NULL,
                        IDC_ARROW );
    wndClass.hbrBackground =
            GetStockObject( WHITE_BRUSH );
    wndClass.lpszMenuName = 0;
    wndClass.lpszClassName = szAppName;

    RegisterClass( &wndClass );
  }
  hWnd = CreateWindow(
       szAppName,          // Ikkunaluokka
       "Ohjelma Heippa",   // Ikkunan otsikko
        WS_OVERLAPPEDWINDOW,// Ikkunatyyli
        CW_USEDEFAULT,     // x koordinaatti
        CW_USEDEFAULT,     // y koordinaatti
        CW_USEDEFAULT,     // Leveys
        CW_USEDEFAULT,     // Korkeus
        0,                 // Ikkunan vanhempi
        0,                 // Menu
        hInst,             // Olio, instanssi
        0                  // Luontiparametrit
       );

  ShowWindow( hWnd, nCmdShow );
  UpdateWindow( hWnd );

  while( GetMessage( &msg, 0, 0, 0 ) ) {
    TranslateMessage( &msg );
    DispatchMessage( &msg );
  }
  return msg.wParam;
}
LRESULT CALLBACK _export
winProc( HWND hWnd, UINT iMsg,
         WPARAM wPar, LPARAM lPar )
{
  HDC hdc;
  PAINTSTRUCT ps;
  RECT rect;
  switch( iMsg ) {
    case WM_PAINT:
      hdc = BeginPaint( hWnd, &ps );
      GetClientRect( hWnd, &rect );
      DrawText( hdc, "Heippa!", -1, &rect,
         DT_SINGLELINE | DT_CENTER |
```

```
          DT_VCENTER );
        EndPaint( hWnd, &ps );
        return 0;
      case WM_DESTROY:
        PostQuitMessage( 0 );
        return 0;
      default:
        return DefWindowProc( hWnd, iMsg,
                              wPar, lPar );
    }
}
```

Huomioita:

1. Vaikka monet tiedostossa `windows.h` määritellyistä tyyppinimistä ovat pelkästään perustyyppien synonyymejä (esim. `typedef long LRESULT;`), on suositeltavampaa käyttää näitä synonyyminimiä. Näin menetellen sovelluksen lähdekoodia ei tarvitse muuttaa vaikka tulevissa Windows-versioissa datatyypit muuttuisivatkin.

2. BORLAND C:n avainsana `_export` kertoo kääntäjälle ja link-editorille, että kyseistä funktiota aiotaan kutsua sovelluksen ulkopuolelta.

3. Windows-ympäristössä pääohjelman nimi on `WinMain` (vastaten CLI-pääohjelmaa `main`). Windows kutsuu pääohjelmaa PASCALin kutsusekvenssillä: siksi atribuutti `PASCAL`.

4. Samasta sovelluksesta voi yhtäaikaa olla suoritettavana useampia olioita. Pääohjelman argumentti `hInst` on viittaus tähän olioon ja argumentti `hPrevInst` viittaus edelliseen olioon.

5. Pääohjelman argumentteina on myös komentorivi (`lpszCmdLine`) ja tieto siitä näytetäänkö sovelluksen pääikkuna vai ei (`nCmdShow`).

6. Ikkunoihin viitataan käyttäen ns. *ripaa* (*handle*, `HWND`).

7. Viestit ovat `MSG`-tyyppisiä tietueita.

8. Ikkunaluokan määrää `WNDCLASS`-tyyppinen tietue. Ikkunaluokka määrää mm. käytettävän ikonin, kursorin ja taustan tyylin. Tietueen jäsen `lpfnWndProc` on osoitin luokan ikkunaproseduuriin. Luokkaan viitataan nimellä, johon osoittaa jäsen `lpszClassName`.

9. Ikkunaluokka rekisteröidään Windowsiin APIlla `RegisterClass( WNDCLASS* )`. Sovelluksen käyttämät luokat rekisteröidään vain yhteen kertaan. Jos asiakas on muodostettu jo aikaisemmin, ei rekisteröintiä tehdä.

10. Ikkunaluokan oliot luodaan APIlla `CreateWindow`. Luonnin yhteydessä voidaan ilmoittaa esim. ikkunan koko ja paikka ruudulla (vakio `CW_DEFAULT` sallii Windowsin valita sopivat arvot).

11. Luotu ikkuna ei vielä näy kuvaruudulla. API `ShowWindow` argumentista riippuen joko tekee ikkunasta näkyvän tai piilottaa sen.

12. Jotta ikkunaan saataisiin haluttu sisältö, on sovelluksen *maalattava* (*paint*) se. API `UpdateWindow` saa Windowsin lähettämään ikkunaproseduurille maalausviestin.

13. Windows luo automaattisesti jokaiselle sovelluksen oliolle viestijonon. Moniajokäyttöjärjestelmiin liittyvissä ikkunointisysteemeissä asiakkaan on itse luotava viestijono (tätä varten systeemissä on API), sillä asiakas voi koostua useammasta prosessista tai säikeestä, joissa kussakin voi olla (tai olla olematta) oma viestinkäsittelynsä.

14. Lopun aikaansa pääohjelma viettää viestisilmukassa. API `GetMessage` poimii viestijonosta kärjessä olevan viestin. API `TranslateMessage` muuntaa tietyt näppäinten painalluksista aiheutuneet viestit sopivaan muotoon. API `DispatchMessage` palauttaa viestin takaisin Windowsille, joka puolestaan lähettää sen viestissä ilmoitetulle ikkunalle. Funktio `GetMessage` palauttaa nollasta poikkeavan arvon aina, kun poimittu viesti ei ole `WM_QUIT`.

15. Windows lähettää viestit ikkunalle kutsumalla kyseisen ikkunaluokan ikkunaproseduuria. Määrättyyn ikkunaan viitataan rivalla (parametri `hWnd`). Viesti on koodattu `UINT`-tyyppiseksi kokonaisluvuksi (parametri `iMsg`). Loppujen ikkunaproseduurin parametrien (`wPar` ja `lPar`) merkitys riippuu viestistä.

16. Ikkunaproseduurin on vastattava kaikkiin viesteihin. Koska erilaisia viestejä on ≈200, ei voida vaatia, että jokainen ikkunaproseduuri hoitaisi jokaisen mahdollisen viestin. Windowsissa onkin API `DefWindowProc`, joka vastaa viesteihin oletustoimilla (yleensä ei tee mitään). Esimerkkiohjelmamme vastaa itse kahteen viestiin: `WM_PAINT` ja `WM_DESTROY`.

17. Ikkuna saa maalausviestin `WM_PAINT` aina kun ikkunan jokin osa pitää maalata uudelleen tai sovellus itse pyytää ko. viestiä (API `UpdateWindow`).

18. Ikkuna maalataan (ts. kirjoitetaan tekstiä tai piirretään kuvia) käyttäen API-funktioita. C:n standardikirjaston funktioilla (`printf`, `puts`, ...) ei voi kirjoittaa Windows-ikkunoihin sen paremmin kuin lukea (`scanf`, `gets`, ...) käyttäjän syötettäkään. API `DrawText` on eräs lukuisista tekstin tulostusfunktioista. Näiden maalausfunktioiden kohteena on tyyppiä `HDC` oleva *laiteyhteys* (*device context*). Laiteyhteys liittyy johonkin fyysiseen laitteeseen (kuvaruutu, kirjoitin, ...). Tässä esimerkissä API `BeginPaint` liittää yhteyden `hdc` maalausta kaipaavaan ikkunaan `hWnd`. Yhteyskäsitteen etuna on se, että täsmälleen sama maalauskäsittely toimii eri

laitteille: ikkuna voidaan esim. tulostaa kirjoittimelle vain vaihtamalla `WM_PAINT`-käsittelyssä käytettyä laiteyhteyttä. Maalauksen jälkeen on laiteyhteys vapautettava. Tässä esimerkissä API `EndPaint` palauttaa ko. ikkunaan liittyvän yhteyden takaisin Windowsille.

19. Asiakasikkunan suorakaide saadaan selville APIlla `GetClientRect`.

20. Esim. näpättäessä hiirellä kaksi kertaa ikkunan vasemmassa yläkulmassa olevaan systeemivalikkoon tai valitsemalla sieltä `sulje`-optio saa ikkuna viestin `WM_DESTROY`. Tässä vaiheessa ikkunaproseduurin on tarkoitus tehdä ikkunan sulkeutumisesta aiheutuvat toimenpiteet (esim. kysyä käyttäjältä, onko hän tosissaan). Jos kyseessä on sovelluksen pääikkuna, käyttäjä ilmeisestikin haluaa lopettaa koko sovelluksen. Tämä saadaan aikaiseksi postaamalla viestijonoon `WM_QUIT`-viesti APIlla `PostQuitMessage`.

18. C++:n I/O-toiminnot

Syöttö- ja tulostustoiminnot eivät kuulu C++:n määrittelyyn. C++:ssa on kuitenkin mahdollista käyttää ANSI C-standardin mukaisia I/O-kirjastofunktioita (kuten `printf()`, `scanf()`, jne.) ja C++:lla itsellään toteutettua `iostream`-luokkakirjastoa.

18.1 Syöttö- ja tulostusvirran käsittely

Luokkakirjasto `iostream` määrittelee joukon toimintoja, joilla voidaan hoitaa C++:n sisäisten muuttujatyyppien (`char`, `int`, jne.) syöttö ja tulostus. Lisäksi osa näistä operaatioista saadaan toimimaan myös (ohjelmoijan määrittelemille) luokkatyypeille.

Syöttötoimintoja varten on olemassa `istream`-luokka ja tulostustoimintoja varten `ostream`-luokka. Luokka `iostream` on johdettu näistä molemmista, joten se mahdollistaa sekä syöttö- että tulostustoiminnot. Luokan `ostream` tulostustoiminto on toteutettu operaattorilla `<<`.

Esim. `cout` on `ostream`-luokan olio, joka on sidottu tulostusvirtaan (standard output). Tulostus tulostusvirtaan tapahtuu (kuten jo hyvin tiedetään) esimerkiksi:

```
int i=2;
cout << i;
```

Vastaavasti `cin` on `istream`-luokan olio, joka on sidottu syöttövirtaan (standard input). Esim. luku syöttövirrasta:

```
int i;
cin >> i;
```

Lisäksi on olemassa `ostream`-luokan oliot `cerr` ja `clog`, jotka on sidottu tulostusvirtaan. Niitä käytetään pääasiassa virhetilanteissa tulostukseen. Niiden toiminta riippuu käyttöjärjestelmästä.

Luokkien `istream` ja `ostream` metodeja

`istream& istream::get(char& c)` Lukee yhden merkin argumenttiin `c`.

`int istream::get()` Palauttaa luetun merkin.

`istream& istream::get(char* s, int n, char c='\n')` Lukee korkeintaan `n-1` merkkiä merkkitaulukkoon `s`. Loppumerkkinä `c`. Jättää loppumerkin merkkijonoon.

`istream& istream::getline(char* s, int n, char c='\n')` Sama kuin edellinen, mutta ei jätä loppumerkkiä luettuun merkkijonoon.

`int istream::gcount()` Palauttaa viimeksi `getline()`:lla luettujen merkkien lukumäärän.

`int istream::peak()` Palauttaa syöttövirrassa seuraavan olevan merkin, mutta ei poista sitä (peek = kurkistaa).

`istream& istream::putback(char)` Lisää merkin takaisin syöttövirtaan.

`int ostream::width(int i)` Asettaa tulostuskentän leveydeksi `i`:n ilmoittaman määrän merkkejä ja palauttaa arvonaan edellisen leveyden.

`ostream& ostream::precision(int i)` Asettaa tulostustarkkuuden eli kertoo montako numeroa tulee desimaalipisteen jälkeen. Uusi asetus on voimassa toistaiseksi.

`ostream& ostream::write(const char* s, int n)` Kirjoittaa tulostusvirtaan merkkijonosta `s` korkeintaan `n` merkkiä.

18.2 Tiedostojen I/O-toiminnot

Otsikkotiedostossa `fstream.h` on määritelty tiedostojen käsittelyyn tarkoitetut luokat `ifstream` ja `ofstream`. Esim.

```
ofstream ulos("tulos.txt");
ifstream sisaan("data.txt");
```

avaa tiedoston `tulos.txt` kirjoitusta varten ja liittää olion `ulos` ko. tiedostoon sekä avaa tiedoston `data.txt` lukua varten ja liittää olion `sisaan` ko. tiedostoon. Luokka `ofstream` on johdettu `ostream` luokasta, joten `ostream`-luokan julkiset metodit ovat myös käytettävissä. Vastaavasti `ifstream`-luokka on johdettu `istream`-luokasta. Lisäksi `oftream`- ja `ifstream`-luokilla on myös omia metodeja: Metodi `eof()` palauttaa luvun, joka on erisuuri kuin nolla, kun on päästy tiedoston loppuun. Metodi `fail()` palautusarvo on erisuuri kuin nolla, jos edellinen I/O-toiminto epäonnistui.

18.3 Operaattoreiden << ja >> ylikuormaus

Esim.

```
 // String.h
class String {
  ...
  friend
    ostream& operator<<(ostream& os,
                        const String& s);
  friend
    istream& operator>>(istream& is, String& s);
private:
  char p[81];
};
// String.cpp
ostream& operator<<(ostream& os, const String& s)
{
  return os << s.p;
}
istream& operator>>(istream& is, String& s)
{
  return is >> s.p;
}
// main.cpp
  ...
  ifstream ifs("syote");
  if (ifs.fail()) {
    cout << "tiedoston syote avaus ei onnistu\n";
    exit(1);
  }
  ofstream ofs("tuloste");
  if (ofs.fail()) {
    cout<<"tiedoston tuloste avaus ei onnistu\n";
    exit(1);
  }
  String a;
  cout << "anna merkkijono\n";
  cin >> a;
  cout << "annoit merkkijonon " << a << endl;
  ofs << a << endl;
  cout << "merkkijono " << a
       << " kirjoitettiin tiedostoon tuloste\n";
  ifs >> a;
  cout << "tiedotosta syote luettiin merkkijono "
       << a << endl;
  ...
```

Yo. esimerkin otsikkotiedostossa `String.h` esitellään globaalit kaksioperandiset operaattorit << ja >>, joilla tulostetaan ja luetaan `String`-tyyppisiä olioita. Koska `ofstream`- ja `ifstream`-luokat on johdettu `ostream`- ja `istream`-luokista, ei `String`-tyyppisten olioiden tiedostoon kirjoittamiseen ja tiedostosta lukemiseen tarvitse ylikuormata operaattoreita << ja >> uudelleen.

19. Tapaustutkimus I: Monen fermionin järjestelmät

Kvanttimekaniikan mukaan potentiaalin sitoman hiukkasen energia ε on kvantittunut:

- ε voi saada arvoksensa vain jonkun diskreeteistä (systeemistä riippuvista) arvoista $\varepsilon_0, \varepsilon_1, \varepsilon_2, \ldots$.
- kutakin sallittua energiaa ε_i vastaa yksi tai useampi *kvanttitila* ψ_k. Kvanttitilat erotetaan toisistaan usein *kvanttilukujen* k avulla.
- kvanttitiloja on yleensä numeroituvasti ääretön määrä, joskus harvoin äärellinen määrä.
- jos energiaan ε_i liittyy täsmälleen yksi kvanttitila ψ_k, sanotaan energiatilan ε_i olevan *degeneroitumaton*, muussa tapauksessa *degeneroitunut*.

Esim. Yksiulotteisessa harmonisessa potentiaalissa

$$V = \frac{1}{2}\, m\omega^2 x^2$$

sallitut energiatilat

$$\varepsilon_n = \hbar\omega \left(n + \frac{1}{2} \right), \; n = 0, 1, 2, \ldots$$

ovat degeneroitumattomia. Kolmiulotteisen harmonisen oskillaattorin energiatilat

$$\varepsilon_{n_1 n_2 n_3} = \hbar\omega(n_1 + n_2 + n_3 + \frac{3}{2}), \; n_i = 0, 1, 2, \ldots$$

ovat enimmäkseen degeneroituneita, sillä samaa energiaa vastaa yleensä useampi kvanttilukujen kombinaatio ($\varepsilon_{300} = \varepsilon_{210} = \varepsilon_{111}$) ja kvanttilukujen kolmikon (n_1, n_2, n_3) ja kvanttitilojen välillä on yksikäsitteinen vastaavuus.

19.1 Monen hiukkasen tilat

Tarkastellaan ulkoisen potentiaalin sitomaa N identtisen hiukkasen systeemiä. Mikäli hiukkaset eivät vuorovaikuta keskenään, kukin niistä asettuu johonkin yhden hiukkasen kvanttitilaan muodostaen N hiukkasen kvanttitilan. Hiukkasia on kahdenlaisia:

bosonit voivat miehittää yksihiukkastiloja rajoituksetta (niiden monihiukkastilan on oltava symmetrinen hiukkasten vaihdon suhteen). Tälläisiä ovat esim. fotonit ja ^{4}He-atomit.

fermionit voivat miehittää yksihiukkastilan vain yhdesti (Paulin kieltosääntö, monihiukkastila on antisymmetrinen hiukkasten vaihdon suhteen). Tälläisiä ovat esim. elektronit ja ^{3}He-atomit.

Keskenään vuorovaikuttavien hiukkasten monihiukkastila on vuorovaikuttamattomien monihiukkastilojen lineaarikombinaatio.

19.2 Fermionijärjestelmien mallintaminen

Yksihiukkastilat

- Numeroidaan yksihiukkastilat esim. $0, 1, 2, \ldots$.
- Vaikka tiloja yleensä onkin ääretön määrä joudutaan käytännössä rajoittumaan äärelliseen, usein pieneenkin ykshiukkastilojen joukkoon.
- Yksihiukkastilat voidaan esittää kokonaislukuina, esim. 8 bittiset etumerkittömät kokonaisluvut riittävät 255 kvanttitilan kuvaukseen.

Kannattaa kuitenkin varautua lukualueen muutokseen määrittelemällä esimerkiksi tyyppiekvivalenssi

```
typedef unsigned char SingleParticleQN;
```

- Kun koodissa käytetään tyyppinimeä `SingleParticleQN` lukualueen muutoksista selvitään pelkästään `typedef`-määrittelyä korjaamalla.
- Yleensä yksihiukkastilojen lukumäärää joudutaan rajoittamaan vielä huomattavasti pienemmäksi kuin käytetty kokonaislukutyyppi sallisi.

Miten itse yksihiukkastiloja kannattaa esittää, rippuu hyvin paljon probleemasta. Joka tapauksessa ainakin energia on jokaiseen tilaan liittyvä ominaisuus. Menetellään siten, että esim. luokasta

```
class SingleParticleState {
public:
  SingleParticleState( Float e ) : e_( e ) {}
  virtual ~SingleParticleState() {}

  Float energy() const { return e_; }
private:
  Float e_;
};
```

johdetaan aktuaaliseen fysikaaliseen tilanteeseen sopiva luokka. Tulevia lukualueen muutoksia ennakoiden luokassa on käytetty tyyppiekvivalenssia

```
typedef double Float;
```

Kvanttiluvut ja yksihiukkastilat on sidottava toisiinsa. Tämä voidaan toteuttaa vaikkapa standardikirjaston assosiaatioluokalla `map` tai yksinkertaisesti vektorina indeksinään kvanttiluvut.

Monihiukkastilat

Monihiukkastilojen esitys riippuu myös usein todellisesta fysikaalisesta tilanteesta. Jokaisen monihiukkastilan on joka tapauksessa tiedettävä mitkä yksihiukkastilat ovat miehitettyjä. Monihiukkastilojen kantaluokkana voisi olla vaikkapa

```
class ManyParticleState {
public:
  ManyParticleState() :
    snglOcc_( new SingleParticleQN[nPart_] ){}
  virtual ~ManyParticleState()
    { delete[] snglOcc_; }

  SingleParticleQN* snglOcc() const
    { return snglOcc_; }
  int partNum() const { return nPart_; }
```

```
  private:
    SingleParticleQN* snglOcc_;
    static int nPart_;
    static int qMax_;
  };
```

Huomattakoon, että

- hiukkasten lukumäärän ollessa N (koodissa `nPart_`) ja käytössä olevien yksihiukkastilojen määrän ollessa M (koodissa `qMax_`) kaikkien mahdollisten monihiukkastilojen lukumäärä on $\left(\begin{array}{c} M \\ N \end{array} \right)$, yleensä suuri luku.
- monihiukkastiloja tarvitaan suuri määrä.
- koska jokaisessa käsiteltävässä monihiukkastilassa on sama määrä hiukkasia, voidaan tilan säästämiseksi tallettaa hiukkasluku staattiseen jäsenmuuttujaan.

19.3 Tilat ja kvanttiluvut

Yksihiukkastilat

Yksihiukkastilojen luonnin yhteydessä lienee syytä määrittää myös kuvaus kvanttilukujen ja tilojen välillä. Koska tämän kuvauksen toteutus rippuu hyvin paljon ympäristöstä, esim. käytössä olevista kirjastoista, kirjoitetaan tässä vaiheessa vain

```
  class QNumMap {
  public:
    virtual ~QNumMap() {}

    virtual const SingleParticleState*
      mapQN( SingleParticleQN ) = 0;
  };
```

Luokka `QNumMap` on siis abstrakti.

Monihiukkastilat

Koska meillä on nyt käytössämme kvanttilukujen ja yksihiukkastilojen välinen kuvaus voimme lisätä monihiukkastilojen kantaluokkaan jäsenet

```
  class ManyParticleState {
  public:
    ...
    Float energy() const;
    ...
  private:
    ...
    static const QNumMap* qNMap_;
  };
  ...
  Float ManyParticleState::energy() const
  {
    Float e = 0.0;
    for( int i = 0; i < nPart_; i++ )
      e += qNMap_->mapQN(snglOcc_[i])->energy();
    return e;
  }
```

Monihiukkastilojen ehdot

Usein kaikki $\left(\begin{array}{c} M \\ N \end{array} \right)$ monihiukkastilaa (M yksihiukkastilojen ja N hiukkasten lukumäärä) eivät ole fysikaalisesti kelvollisia, esim.

- säilymislait, kuten kokonaisliikemäärä, kulmaliikemäärä jne., asettavat ehtoja sallituille monihiukkastiloille.
- halutaan rajoittaa monihiukkastilojen lukumäärää vaikkapa asettamalla yläraja monihiukkastilan energialle.

Implementoidaan ehdot luokan

```
  class StateAcceptance {
  public:
    StateAcceptance( int n, const QNumMap* m ) :
      nPart_( n ), qNMap_( m ) {}

    virtual bool
      accept( const SingleParticleQN* q )
        { return true; }
  protected:
    const int nPart_;
    const QNumMap* qNMap_;
  };
```

metodilla `accept()`. Tässä

- oletuksena hyväksytään kaikki ehdokkaat. Johdetuissa luokissa voidaan tätä oletusta muuttaa.
- jäsenet `nPart_` (hiukkasluku) ja `qNMap_` (kvanttiluku $\mapsto$yksihiukkastila) on otettu mukaan, koska erittäin todennäköisesti näitä tarvitaan silloin kun halutaan poiketa oletuskäytännöstä.

19.4 Virheiden hausta

Koodia kirjoitettaessa kannatta varautua virheiden metsästykseen. Usein hyödylliseksi osoittautuu olioiden sisällön tulostus. Lisätään tätä varten luokkiimme metodit

```
  class SingleParticleState {
  public:
    ...
  #ifdef SET_DEBUG_
    virtual void dump() = 0;
  #endif
    ...
```

Yllä

- tulostusfunktio `dump()` on jätetty puhtaasti virtuaaliseksi, koska tällä abstraktilla tasolla ei ole mitään tietoa yksihiukkastilojen luonteesta,
- virheenhaku on käärityt `#ifdef`- ja `#endif`- direktiivien väliin, jolloin virheenhakukoodi saadaan mukaan tai pois käännöksestä yksinkertaisesti asettamalla sopivassa otsikkotiedostossa joko

```
  #define SET_DEBUG_
```

tai

```
#undef SET_DEBUG_
```

Lisätään vastaavasti monihiukkastiloihin metodi

```
class ManyParticleState {
public:
  ...
#ifdef SET_DEBUG_
  virtual void dump() const;
#endif
  ...
};
...
#ifdef SET_DEBUG_
void ManyPartcleState::dump() const
{
  for( int i = 0; i < nPart_; i++ )
    qNMap_->mapQN( snglOcc_[i] )->dump();
}
#endif
```

Tässäkin `dump()` on jätetty virtuaaliseksi, koska todennäköisesti halutaan tilanteeseen sopiva tulostus.

19.5 Statistista mekaniikkaa

Olkoon A jokin systeemin ominaisuus ja A_i tämän monihiukkastilassa i saama arvo. Jos esim. A on kokonaisenergia E, niin vuorovaikuttamattomien hiukkasten tapauksessa

$$E_i = \sum_{n \in K_i} \epsilon_n,$$

missä kvanttiluvut K_i ilmoittavat monihiukkastilassa i miehitetyt yksihiukkastilat. Statistisen mekaniikan mukaan mitattaessa suuretta A saadaan (kanonisessa joukossa) tulokseksi *odotusarvo*

$$\langle A \rangle = \frac{1}{Z} \sum_i A_i e^{-\beta E_i},$$

missä E_i on monihiukkastilan i energia, suure

$$\beta = \frac{1}{k_B T}$$

on kääntäen verrannollinen lämpötilaan T ja summaus käy yli kaikkien monihiukkastilojen. Suuretta

$$Z = \sum_i e^{-\beta E_i}$$

kutsutaan *tilasummaksi* tai *partitiofunktioksi*.

Numeerisen tarkkuuden säilyttämiseksi, varsinkin matalissa lämpötiloissa, on syytä kirjoittaa odotusarvon lausekkeessa esiintyvä osoittaja Q muotoon

$$\begin{aligned} Q &= \sum_i A_i e^{-\beta E_i} \\ &= e^{-\beta E_0} \left(A_0 + \sum_{i>0} A_i e^{-\beta(E_i - E_0)} \right). \end{aligned}$$

Tässä on ajateltu monihiukkastilat numeroiduksi siten, että $i = 0, 1, 2 \ldots$ ja että tila 0 on systeemin *perustila*, ts.

$$E_0 < E_i \quad \forall i > 0.$$

Kirjoitetaan nimittäjä samoin muotoon

$$Z = e^{-\beta E_0} \left(1 + \sum_{i>0} e^{-\beta(E_i - E_0)} \right).$$

Nyt

$$\langle A \rangle = \frac{A_0 + \sum_{i>0} A_i e^{-\beta(E_i - E_0)}}{1 + \sum_{i>0} e^{-\beta(E_i - E_0)}},$$

joten myös äärimmäisen matalien lämpötilojen käsittely on tarkkaa, erikoisesti

$$\langle A \rangle|_{T=0} = A_0,$$

sillä

$$\lim_{T \to 0} e^{-\beta(E_i - E_0)} = 0.$$

Huom. Edellä esitetty soveltuu degeneroitumattoman perustilan tapaukseen. Menettely on kuitenkin helposti yleistettävissä degeneroituneen perustilan käsittelyyn, ts. systeemiin jossa

$$E_0 = E_1 = \cdots = E_m \text{ ja } E_0 < E_i \ \forall i > m.$$

Toteutus

Modifioidaan hiukan aikaisempaa monihiukkastilaamme:

- muutetaan tilan kokonaisenergian evaluoiva metodi virtuaaliseksi, jolloin luokka soveltuu myös vuorovaikuttavien hiukkasten käsittelyyn.
- siirrettävien argumenttien lukumäärän vähentämiseksi lisätään metodi, jolla saadaan selville kvanttilukujen ja yksihiukkastilojen välinen kuvaus.

Luokka `ManyParticleState` näyttäisi nyt tälläiseltä

```
class ManyParticleState {
public:
  ...
  const QNumMap* map() const
    { return qNMap_; }
  virtual Float energy() const;
  ...
private:
  ...
  static const QNumMap* qNMap_;
};
```

Implementoidaan statistiikka esim. luokalla

```
class Statistics {
public:
  Statistics( const ManyParticleState** s,
              int n );

  Float partition( Float t ) const;
  Float energy() const;
```

```
  ...
private:
  const ManyParticleState** s_;
  int n_;
  const QNumMap* m_;
  int g0_;     // ground state
  Float e0_;   // its energy
  Float t_;
  Float z_;
  ...
};
```

Muodostimessa on tarkoitus etsiä perustila, esim.

```
Statistics::Statistics(
    const ManyParticleState** s, int n ) :
  s_( s ), n_( n ), m_( s[0]->map() )
{
  g0_ = 0;
  e0_ = s_[0]->energy();
  for( int i = 1; i < n_; i++ )
    if( s_[i]->energy() < e_0 ) {
      e0_ = s_[i]->energy();
      g0_ = i;
    }
}
```

Suureiden odotusarvot on ajateltu laskettaviksi siten, että ensin evaluoidaan partitiofunktio:

```
Float Statistics::partition( Float t ) const
{
  t_ = t;
  z_ = 1.0;
  if( t > 0.0 ) {
    Float b = 1.0/t;
    Float d;
    for( int i = 0; i < n_; i++ )
      if( i != g0_ ) {
        d = s_[i]->energy();
        z_ += exp( -b*(d - e0_) );
      }
  }
  return z_;
}
```

Esim. energian kyseessä ollen odotusarvo olisi

```
Float Statistics::energy() const
{
  Float e = e0_;
  if( t_ > 0.0 ) {
    Float b = 1.0/t_;
    Float d;
    for( int i = 0; i < n_; i++ )
      if( i != g0_ ) {
        d = s_[i]->energy();
        e += d*exp( -b*(d - e0_) );
      }
  }
  return e/z_;
}
```

Huom. Numeerisen stabiilisuuden takia saattaa olla järkevämpää verrata lämpötilaa `t` koneen esitystarkkuuden suuruusluokkaa olevaan lukuun kuin nollaan.

20. Tapaustutkimus II: Lineaarialgebra

Olio-ohjelmoinnissa luonnollinen tapa implementoida lineaarialgebra on kirjoittaa

- lineaarialgebran entiteettejä, vektoreita ja matriiseja, vastaavat luokat ja
- luokkiin jäsenmetodit tai globaalit funktiot toteuttamaan lineaarialgebran operaatioita.

Sellaiset operaatiot kuin matriisien yhteen- ja kertolasku on suhteellisen helppo koodata. Sen sijaan esim. matriisin kääntö- ja diagonalisointioperaatioita ei yleensä kannata itse kirjoittaa, koska näiden toteuttamiseksi on saatavana valmiina lukuisia hyviä aliohjelmakirjastoja.

Suurin osa numeerisista valmiskirjastoista on FORTRAN-kielisiä. Eräs erinomainen (ja ilmainen) kirjasto on LAPACK (http://www.cs.colorado.edu/~lapack/): LAPACK provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as reordering of the Schur factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision.

Tarkastellaan esimerkkinä reaalisten symmetristen matriisien diagonalisointia. Yksinkertaisen tarkkuuden ohjelman listaus alkaa näin

```
 SUBROUTINE SSYEV(JOBZ, UPLO, N, A, LDA,
 +                W, WORK, LWORK, INFO )
*
*  -- LAPACK driver routine (version 3.0) --
*     Univ. of Tennessee,
*     Univ. of California Berkeley, ....
*
*     .. Scalar Arguments ..
      CHARACTER  JOBZ, UPLO
      INTEGER    INFO, LDA, LWORK, N
*     ..
*     .. Array Arguments ..
      REAL       A( LDA, * ), W( * ), WORK( * )
*     ..
*
*  Purpose
*  =======
*
*  SSYEV computes all eigenvalues and,
*  optionally, eigenvectors of a
*  real symmetric matrix A.
*
*  Arguments
*  =========
*
*  JOBZ    (input) CHARACTER*1
*          = 'N':  Compute eigenvalues only;
```

```
*          = 'V':  Compute eigenvalues and
*                  eigenvectors.
*
*  UPLO    (input) CHARACTER*1
*          = 'U':  Upper triangle of A is stored;
*          = 'L':  Lower triangle of A is stored.
*
*  N       (input) INTEGER
*          The order of the matrix A.  N >= 0.
*
*  A       (input/output) REAL array,
*          dimension (LDA, N)
*          On entry, the symmetric matrix A.
*          If UPLO = 'U', the leading N-by-N
*          upper triangular part of A contains the
*          upper triangular part of the matrix A.
*          If UPLO = 'L', the leading N-by-N
*          lower triangular part of A contains
*          the lower triangular part
*          of the matrix A.
*          On exit, if JOBZ = 'V',
*          then if INFO = 0, A contains the
*          orthonormal eigenvectors of
*          the matrix A.
*          If JOBZ = 'N', then on exit
*          the lower triangle (if UPLO='L')
*          or the upper triangle (if UPLO='U')
*          of A, including the
*          diagonal, is destroyed.
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A.
*          LDA >= max(1,N).
*
*  W       (output) REAL array, dimension (N)
*          If INFO = 0, the eigenvalues
*          in ascending order.
*
*  WORK    (workspace/output) REAL array,
*          dimension (LWORK)
*          On exit, if INFO = 0, WORK(1)
*          returns the optimal LWORK.
*
*  LWORK   (input) INTEGER
*          The length of the array WORK.
*          LWORK >= max(1,3*N-1).
*          For optimal efficiency,
*          LWORK >= (NB+2)*N,
*          where NB is the blocksize for
*          SSYTRD returned by ILAENV.
*
*          If LWORK = -1, then a workspace
*          query is assumed; the routine
*          only calculates the optimal size
*          of the WORK array, returns
*          this value as the first entry
*          of the WORK array, and no error
*          message related to LWORK is issued
*          by XERBLA.
*
*  INFO    (output) INTEGER
*          = 0:  successful exit
*          < 0:  if INFO = -i,
*                the i-th argument had
*                an illegal value
*          > 0:  if INFO = i,
*                the algorithm failed
*                to converge;
*                i off-diagonal elements
*                of an intermediate tridiagonal
*                form did not converge to zero.
*
*  =====================================
```

Kaksinkertaisen tarkkuuden rutiinissa `DSYEV` taulukot ovat tyyppiä `DOUBLE PRECISION`, mutta muutoin rutiini on samanlainen kuin `SSYEV`.

Ensimmäinen tehtävä on kirjoittaa aliohjelmien `SSYEV` ja `DSYEV` C-kutsuja vastaavat esittelyt. Nyt

- FORTRAN-aliohjelmiin välitetään parametrit pääsääntöisesti referensseinä,
- kutsumekanismi noudattaa (pääsääntöisesti) C:n mekanismia,
- kutsuissa on kääntäjäkohtaisia eroja.

Esimerkiksi MS Visual-ympäristössä

- FORTRANin `CHARACTER`-tyyppiset merkkijonot välitetään siten, että aliohjelmalle annetaan osoitin merkkitaulukkoon ja sitä seuraten parametrilistassa merkkijonon pituus *by value*-mekanismilla,
- FORTRANissa kutsutut aliohjelmat siivoavat itse pinon, ts. poistavat siitä välitetyt parametrit kun taas C- ja C++-ohjelmissa siivous on kutsujan vastuulla. Tätä varten Visual C++:aan on lisätty avainsana `__stdcall`. FORTRAN-kieliset (samoin kuin WINDOWSin API-funktiot) on esiteltävä tätä avainsanaa käyttäen.

Näin päädymme esittelyihin

```
extern "C" {
void __stdcall SSYEV( const unsigned char& jobZ,
                      int lz,
                      const unsigned char& upLo,
                      int lu,
                      const int& n, float* a,
                      const int& lda,
                      float* w, float* work,
                      int& lwork, int& info );
void __stdcall DSYEV( const unsigned char& jobZ,
                      int lz,
                      const unsigned char& upLo,
                      int lu,
                      const int& n, double* a,
                      const int& lda,
                      double* w, double* work,
                      int& lwork, int& info );
}
```

Toteutetaan tällä kertaa matriisin diagonalisointi matriisiluokan ulkopuolisena metodina. Käytetään hyväksi aiemmin esillä ollutta `Array2D`-luokkaa, jonka instanssien `Array2D<float>` ja `Array2D<double>` tyyppisiä olioita voidaan nyt suoraan käyttää diagonalisointirutiinien argumentteina. Koska nämä rutiinit tarvitsevat työtaulukkoja ja muita käyttäjän kannalta epäoleellisia parametrejä, kirjoitetaan diagonalisoinnista huolehtiva luokka, esim.

```
template<class Field>
class SymmEigVal_ {
public:
  SymmEigVal_( int n ) :
      n_( n ), work_( 0 ), za_( 'V' ), ua_( 'U' ) {}
  virtual ~SymmEigVal_() { delete[] work_; }
  void diagonalize( Array2D<Field>& a, Field* eig );
protected:
  const int n_;
  Field* work_;
  int nw_;
  int info_;
  const unsigned char za_;
  const unsigned char ua_;
  virtual void fcall_( Field* a, Field* w,
                       Field* work ) = 0;
};
```

Koska FORTRAN-aliohjelmien nimet riippuvat laskentatarkkuudesta, toteutetaan varsinainen kutsu vasta yo. luokasta johdetuissa spesialisoiduissa luokissa. Tässä vaiheessa implementoidaan `diagonalize`-metodi kuten

```
template< class Field>
void
SymmEigVal_<Field>::diagonalize( Array2D<Field>& a,
                                 Field* eig )
{
  if( !work_ ) {
    nw_ = -1;
    Field d;
    fcall_( a, eig, &d );
    nw_ = d;
    work_ = new Field[nw_];
  }
  fcall_( a, eig, work_ );
}
```

Nyt voimme spesialisoida lukutyyppeihin liittyvät luokat

```
template<class Field> class SymmEigVal;

template<>
class SymmEigVal<float> :
      public SymmEigVal_<float> {
public:
  SymmEigVal( int n ) : SymmEigVal_<float>( n ) {}
private:
  void fcall_( float* a, float* w, float* work )
    { SSYEV( za_, 1, ua_, 1, n_, a, n_, w, work,
             nw_, info_ );
    }
};
template<>
class SymmEigVal<double> :
      public SymmEigVal_<double> {
public:
  SymmEigVal( int n ) : SymmEigVal_<double>( n ) {}
private:
  void fcall_( double* a, double* w, double* work )
    { DSYEV( za_, 1, ua_, 1, n_, a, n_, w, work,
             nw_, info_ );
    }
};
```