C++ rautaisannos

Kolme tapaa sanoa, että tulostukseen käytetään standardikirjaston `iostream`-osassa määriteltyä, nimiavaruuden `std` oliota `cout`:

```
#include <iostream>
main(){
 using namespace std;
 cout<<"toimii"<<endl;
}
```

```
#include <iostream>
 main(){

 std::cout<<"toimii\n";
}
```

Kolmas tapa on `using std::cout;`, ja sen jälkeen `cout` riittää. Koodi `\n` vaihtaa riviä, `endl` sekä vaihtaa riviä että tyhjentää tulostuspuskurin (tulostaa heti, ei "sopivammalla hetkellä").
preprosessoridirektiivit, koodia prosessoidaan jo ennen käännöstä:

```
#include <iostream> // hae kirjaston osa iostream
```

Tärkeitä standardikirjaston osia:

```
#include <iostream>   // perustulostusta ja -lukua
#include <fstream>    // tiedostojen käsittelyä
#include <iomanip>    // tulostuksen muotoilua
#include <string>     // merkkijonot
#include <cmath>      // matikkaa (sin, cos ...)
```

Kaksi include-tapaa, erona hakuprosessi:

```
#include <iostream>   // haku "standardipaikasta"
                      // varmin standardikirjaston osille
                      // ei aina tiedosto
#include "mydefs.hpp" // laaja haku, mm. työhakemisto
                      // vaarana löytää väärä tiedosto
```

Kommentit:

```
// kommentti (C++-tyyliä)
/* kommentti (C-tyyliä) parempi pitkissä,
   monirivisissä
   kommenteissa */
```

Muuttujatyyppejä: (** = tärkeä, * = hyvä tietää - = harvinaisuus)

```
kokonaisluvut: short(-) int (**) long (*) long long (-)
  reaaliluvut: float (*) double (**) long double (-)
       merkki: char (**)
   totuusarvo: bool (*)
    ei mitään: void (**)
```

Numeerisissa laskuissa `double` on eniten käytetty laskentatarkkuus. Jos funktio ei palauta mitään arvoa, vain tekee jotain, sen tyyppi on `void`.

Lisämääreitä:

```
const int i=13; // vakio i on 13
                // et voi muuttaa sitä myöhemmin
unsigned j;     // etumerkitön kokonaisluku
                // varo laskutoimituksissa, jos tulos voi olla
                     negatiivinen!
size_t koko;    // käytetään kertomaan kokoja, kuin unsigned
```

Käytä tyyppiä `const` aina kun se on mahdollista.
Yksinkertaisia tyyppimuunnoksia (kääntäjä tekee jotkut automaattisesti, mutta silloin niitä on vaikea löytää koodista)

```
int i;
double x=2.345;
i = static_cast<int>(x); // C++ nykytyyli, i on 2
i = (int) x; // sama C-tyylisenä
kääntäjä tekee usein edellisestä jälkimmäisen.
```

C-tyylinen muunnos on lyhyt, mutta vaikea löytää koodista, etkä monimutkaisemmissa tietotyypeissä tiedä mitä se yrittää tehdä ("luota minuun ja tee mitä käsken"). Laskuoperaattorit

```
+ - * /  % (on jakojäännös)

++i;  // kasvatetaan muuttujan i arvoa yhdellä
      // ja käytetään sitä arvoa
i++;  // käytetään muuttujan i arvoa
      // ja kasvatetaan sitä *sen jälkeen* yhdellä
```

Vertailuoperaattorit

```
==      yhtä kuin               esim. (2==3) on false
!=      eri kuin                esim. (2!=3) on true
<       pienempi kuin
<=      pienempi tai yhtä suuri kuin
>       suurempi kuin
>=      suurempi tai yhtä suuri kuin
```

Johdettuja operaattoreita

```
vector<double> x(10); // x on 10 reaaliluvun vector-kontti
              // indeksi alkaa nollasta
              // olemassa on siis x[0],x[1],...,x[9]
x[i]=12;      // i:s alkio on 12
              // i:llä on jokin arvo tähän tultaessa
x.at(i) = 12; // kuten x[i], mutta testattuna
              // että i:s alkio on kontin sisällä
```

```
*  on osoitin eli pointteri
int* a;  // a on muistiosoite, jossa on kokonaisluku
int *a ; // sama
         // lue: ‘‘a osoittaa kokonaislukuun’’
         // tai ‘‘kokonaisluvun muistipaikka on a’’
int *a,*b,*c; // sama monelle
```

```
&  on viittaus
int b ; // b on kokonaisluku
fun(int& b) // funktioon menee viittaus kokonaislukuun b
```

if-ehtolause

```
if(i<10)
  cout< "i < 10"<<endl;
else if(i<15)
  cout << " 10<= i < 15"<<endl;
else {
  cerr << "virhe, i >= 15"<<endl;
  exit(1);
}
```

for-silmukka; (muuttuja `i` on olemassa vain silmukan sisällä)

```
vector<int> x(10);
for (int i=0;i<10;i++){
   x[i] = i;
}
```

while-silmukka

```
vector<double> x(20);
int i=19;
while (i!=0) {
  x[i] = 1.0/i;
  std::cout << i<<" "<<x[i]<<std::endl;
  i--;
}
```

do-while-silmukka

```
i = 10
do {
    cout<<i<<endl;
    --i;
} while (i>0);
```

switch-rakenne

```
switch (m)  // toimii vain kokonaislukutyyppisille m
{
  case 0:
    // jatka tekemättä mitään
  case 1: case 2:
    break;  // lopeta ohjelma
  case 3:
  {
    // aaltosulut kun useampia rivejä
    break;
  }
  default:
    // oletustoiminto
    break;
}
```

Luokka (*class*) on luettelo ominaisuuksistä, joita sen ilmentymillä, olioilla (*object*), voi olla, sekä menetelmistä (metodeista), jotka ovat funktioita joilla ominaisuuksiin vaikutetaan. Jos luokka on veturi, niin Tuomas on yksi veturi, eli veturi-luokan olio eli "Tuomas on veturi". Tavallisempi luokan nimeäminen olisi isoilla kirjaimilla, tyyliin VeturiLuokka.

```
#include <iostream>
#include <string>

typedef std::string color;
typedef std::string island;
typedef bool working;

class veturi {
  color vari;
  working tilattu;
  static island saari;
public:
  std:: string nimi;
  int teho;
  void maalaa(color);
  void katso() const;
  island sijainti() const { return saari;};
  void sijoita(island) ;
  friend void tilaa(veturi &);
  void savuttaa() {std::cout<<"Veturi "<<nimi<<" savuttaa
      mustaa savua\n";}
  veturi() { std::cout<<"Muodostin 1 rakentaa nimettömän
      veturin\n";
    nimi = "Rautahepo";  teho = 0; tilattu = false;}
  veturi(std::string name){std::cout<<"Muodostin 2 rakentaa
      veturin nimeltään "<<name<<std::endl;
    nimi=name; vari ="harmaa"; teho = 5; tilattu = false;}
  ~veturi(){std::cout<<"Laitan veturin "<<this->nimi<<"
      laatikkoon"<<std::endl;}
};


class muuveturi: public veturi { // perii luokan veturi
  std::string omistaja;
public:
  friend void tilaa(muuveturi &);
  muuveturi(std::string a,std::string b):  veturi(a), omistaja
      {b}  {}; // alustuslista
  void savuttaa() {std::cout<<"Veturi "<<nimi<<" savuttaa
      sinistä savua\n";}
  std::string lue_omistaja() const { return omistaja;};
};
```

```
void veturi::maalaa(color col) {
  vari = col ;
}

void veturi::katso() const {
  if(vari=="harmaa")
    {
      std::cout<< this->nimi <<"-veturi on maalaamaton ja
          vielä "<<vari<<std::endl;
    }
  else
    {
      std::cout<< this->nimi <<"-veturi on "<<vari<<std::endl;
    }
}

void veturi::sijoita(island place){
  saari=place;
}

void tilaa(veturi& engine){
  if(engine.tilattu) {
    std::cout <<"Olen pahoillani, veturi "<<engine.nimi<<" on
        jo varattu :("<<std::endl;
    return;
  }
  engine.tilattu = true;
  std::cout<<"Veturi "<<engine.nimi<<" tilattu!\n";
}
void tilaa(muuveturi& engine,bool suhteita){
  if(suhteita)
    {
      std::cout<<"Ahaa, teillä on suhteita ...";
      tilaa(static_cast<veturi&>(engine));
    }
  else
    {
      std::cout<<"Olen pahoillani, Herra "<<engine.
          lue_omistaja()<<"n veturia "<<engine.nimi<<" voi
          tilata vain suhteilla!\n";
    }
}


island veturi::saari;

int main(){
  using namespace std;
```

```
  veturi Tuomas, Pekka("Pekka");

  cout<<"Tuomaksen nimi on viela "<<Tuomas.nimi<<endl;
  cout<<"Pekka on jo nimeltaan  "<<Pekka.nimi<<endl;
  Tuomas.nimi = "Tuomas";  // public, voidaan asettaa suoraan
  cout<<"Tuomas nimettiin nimella "<<Tuomas.nimi<<endl;
  Tuomas.teho = 10;         // public, voidaan asettaa suoraan
  Tuomas.maalaa("vihrea");  // private, tehtävä menetelmällä
  Tuomas.sijoita("Sodor");  // static, kaikki veturit menevät
      samaan paikkaan
  Tuomas.katso();
  Pekka.katso();
  Pekka.maalaa("punainen");
  cout<<"Nyt ";
  Pekka.katso();
  cout<<"Tuomas ja Pekka ovat "<<Pekka.sijainti()<<"in
      saarella"<<endl;
  cout<<"Tuomaksen  teho on "<<Tuomas.teho<<endl;
  cout<<"Pekan teho on "<<Pekka.teho<<endl;
  Pekka.savuttaa();

  muuveturi Santtu("Santtu","Herttua");
  Santtu.maalaa("harmaa") ; // veturilta peritty ominaisuus
  Santtu.lue_omistaja(); // uusi ominaisuus, jota muilla
      vetureilla ei ole
  Santtu.katso();
  cout<<"Santtu on paikassa nimeltä "<<Santtu.sijainti()<<endl
      ; //
  Santtu.savuttaa();

  tilaa(Tuomas);
  tilaa(Tuomas);

  tilaa(Santtu,false);
  tilaa(Santtu,true);


  cout<<"Lopetan leikin talta illalta.\n";
  // ohjelma päättyy ja kutsuu hajottimia
  return 0;
}
```

Tulostus:

```
Muodostin 1 rakentaa nimettömän veturin
Muodostin 2 rakentaa veturin nimeltään Pekka
Tuomaksen nimi on viela Rautahepo
Pekka on jo nimeltaan  Pekka
Tuomas nimettiin nimella Tuomas
Tuomas-veturi on vihrea
Pekka-veturi on maalaamaton ja vielä harmaa
Nyt Pekka-veturi on punainen
Tuomas ja Pekka ovat Sodorin saarella
Tuomaksen  teho on 10
Pekan teho on 5
Veturi Pekka savuttaa mustaa savua
Muodostin 2 rakentaa veturin nimeltään Santtu
Santtu-veturin omistaa Herttua
Santtu-veturi on maalaamaton ja vielä harmaa
Santtu on paikassa nimeltä Sodor
Veturi Santtu savuttaa sinistä savua
Veturi Tuomas tilattu!
Olen pahoillani, veturi Tuomas on jo varattu :(
Olen pahoillani, Herra Herttuan veturia Santtu voi tilata vain suhteilla!
Ahaa, teillä on suhteita ...Veturi Santtu tilattu!
Lopetan leikin talta illalta.
Laitan veturin Santtu laatikkoon
Laitan veturin Pekka laatikkoon
Laitan veturin Tuomas laatikkoon
```

Esimerkin veturi-luokan selostus:

- `typedef std::string color;` määrittelee, että keksimäni tyyppinimi `color` on merkkijono. Helpottaa lukemista ja myöhemmin on helppo muuttaa mieltä jos `std::string` ei miellytäkään.
- `Class:` Oletuksena kaikki luokan jäsenet (*members*) ovat yksityisiä (*private*), ellei toisin sanota (*public*). `Struct:` Oletuksena kaikki on julkista (*public*). Muita eroja ei ole.
- Staattiset jäsenet (*static*): jäsenestä `saari` ei ole joka oliolle omaa kopiota, vaan yksi yhteinen. Staattinen jäsen ei ole varsinaisesti luokan osa, siksi se pitää ottaa käyttöön erikseen rivillä `island veturi::saari;`
- Luokassa `veturi` on joukko menetelmiä (*method*; `maalaa()`,`katso()` jne.),joilla jäsenille annetaan arvoja ja niitä tutkitaan. Menetelmät määritellään joko samantien luokan yhteydessä tai erikseen kuten funktio `void veturi::maalaa(color col)`

- Veturilla on yksityinen väri, eikä sitä saa muuttaa muuten kuin julkisella maalaa()-metodilla. Tieto väristä on siis suojattu (*data encapsulation*).
- Menetelmän `color katso() const;` ei muuta mitään, siksi perässä on `const`.
- Menetelmä `tilaa()` ei kuulu luokkaan, vaan on luotettu ystävä `friend void tilaa()` joka saa luvan käsitellä tilauksia.
- Veturin menetelmään `tilaa()` lähetetään **viittaus** veturiin, `tilaa(veturi& engine)`, koska sen pitää muuttaa tietyn veturi-olion arvoa `tilattu`. Jos funktioon lähetetään olio, siitä tehdään funktiossa vain pian hajotettava kopio. Toinen toimiva tapa olisi lähettää veturin osoite, `tilaa(veturi* pengine)`, jolloin voisi asettaa `(*pengine).tilattu = true;` tai nätimmin `pengine->tilattu = true;`
- Muodostimet (*constructors*) `veturi()` ja `veturi(std::string name)` antavat kaksi tapaa luoda veturi: ilman argumenttia käytössä on edellinen ja string-argumentilla jälkimmäinen. Rivillä `veturi Tuomas, Pekka("Pekka");` kutsutaan Tuomas veturille `veturi()` muodostinta, joka panee sen nimeksi "Rautahepo"ja tehoksi nolla. Pekka veturi sen sijaan tehdään argumentilla "Pekka", jonka perusteella muodostin on `veturi(std::string name)` ja nimeksi asettuu "Pekka", väriksi "harmaa"ja tehoksi 5. Jos itse tekee muodostimen, niin kääntäjä ei tee ns. oletusmuodostinta.
- Luokan hajotinta (*destructor*) `~veturi()` kutsutaan kun oliota ei enää tarvita. Jos sitä ei ole kääntäjä tekee sen. Hajotinta harvoin kutsutaan itse.
- Veturin nimi on yleinen (*public*), joten sen voi antaa suoraan, kuten `Tuomas.nimi="Tuomas";`
- Koska veturi-luokan jäsen `saari` on staattinen ja rivi `Tuomas.sijoita("Sodor");` asettaa saareksi "Sodor", niin kaikki veturit ovat sen jälkeen samalla Sodorin saarella, myös Pekka ja Santtu. Saari on oletusarvoisesti yksityinen (*private*), joten vetureita ei voi siirtää saarelta toiselle ilman julkista `sijoita()` menetelmää.
- Luokka `muuveturi` perii kaiken luokasta `veturi`, eli "muuveturi on veturi". Lisäksi sillä on `omistaja`.

Veturi-luokassa ei ole näkyvää kopionmuodostinta, mutta kääntäjä tekee sen:

```
veturi Jori(Pekka);
```

tekisi Pekasta täydellisen kopion Jori.

Myös veturin asettaminen toiseksi toimii, Pekasta tulee Tuomas kun

```
Pekka = Tuomas;
```

Luokan voi **periä**, eli jokin luokka voi periä toisen luokan ominaisuudet. Tämä auttaa kierrättämään olemassaolevaa koodia. Esimerkissä luokka `muuveturi` perii luokan `veturi` kaikkinensa. Perintä on "is-a", ("on") kuten "hauki on kala"ja hauella on perittynä *kaikki* kalojen ominaisuudet. Pidä varasi, jos määrittelet `lintu`-luokalle ominaisuuden `lentää`, ja annat `pingviini`-luokan periä `lintu`-luokan, niin myös pingviinit lentävät.

Perintään liittyy myös **virtuaaliset menetelmät (funktiot)**. Virtuaalinen on "melkein olemassaoleva" menetelmä, jota käytetään jos ei muusta tiedetä. Tämä liittyy läheisesti *polymorfismiin*.