

C++

- bred användning inom programvaruindustrin
 - tillgängligt överallt
 - generellt
 - effektiv kod
 - god kontroll över minnesanvändning
- ny standard 2011 – C++11
 - mindre revision planerad till senare delen av 2014 – C++14
 - större revision planerad till 2017 – C++1y
 - stort fokus på ytterligare generalisering och förenklad användning

Ett första programexempel

```
// Det klassiska C-programmet i C++-tappning
```

```
#include <iostream>
```

```
int main()
{
    std::cout << "Hello, world" << std::endl;

    return 0;
}
```

- stort standardbibliotek
 - åtkomst genom inkludering av de delar som programmet utnyttjar
 - alla standardbiblioteksnamn kapslas i standardnamnrymden `std`
- Översättningsmodell:

källkod -> preprocessor -> kompilator -> länkare -> körbart program (a.out)

```
g++ hello.cc
```

Imperativ programmering (Fö 1–2)

Imperativ programmering är i många språk grunden för kodning av algoritmer – även i objektorienterade programmeringsspråk.

- *variabler* – används för att lagra värden

```
int x{ 0 };          int x = 0;
```

- deklaras med *namn* och *datatyp* – *deklarationssats*
- datatypen bestämmer *värdemängd* och tillåtna *operationer*
- en *konstant* kan ses som en variabel som måste *initieras* och sedan inte kan ändras
- även variabler och konstanter av enkel typ kallas *objekt* i C++

- *tilldelning* – den grundläggande imperativa operationen

```
x = 1;
```

- ger en variabel ett nytt värde – *tilldelningsuttryck*, *tilldelningssats*
- inte tillåtet för konstanter
- *sats* – den minsta fristående konstruktionen i ett program
- de tre *strukturprimitiverna* – med dessa kan alla slags sekventiella program konstrueras
 - *sekvens* { sammansatt sats; block }
 - *val* (if, switch)
 - *repetition* (while, do, for)
- *funktion* – grundläggande konstruktion för att strukturera och återanvända programkod
- *in- och utmatning* – typisk *standardbiblioteksprogramvara*

Enkel in- och utmatning – strömbiblioteket

```
#include <iostream>
#include <iomanip>
```

```
using namespace std;
```

```
// öppna standardnamnrymden std
```

```
int main()
{
    int i;

    cout << "Skriv ett heltal < 1000: ";

    cin >> i;

    cout << "Det var " << setw(3) << i << endl;
}
```

- strömbiblioteket kan hantera in- och utmatning av alla grundläggande datatyper samt strängar
- standardströmmar – `cin` och `cout` samt `cerr` och `clog` för speciella syften
- funktioner för in- och utmatning – de formaterande operatorerna `>>` och `<<`
- olika möjligheter att formatera utskrifter – *manipulatorer*

Programstruktur

Ett enkelt C++-program består av en eller flera funktioner.

```
#include <iostream>
using namespace std;

int max(int x, int y)
{
    return (x > y ? x : y);
}

int main()
{
    int i, j;

    cout << "Skriv två heltal: ";
    cin >> i >> j;

    auto m = max(i, j);

    cout << m << " var störst.\n";

    return 0;
}
```

Uppdelning på filer – "file as module" (forts.)

```
// Fil: test.cc

#include <iostream>
#include "max.h"

using namespace std;

int main()
{
    ...
    auto m = max(i, j);
    ...
}
```

- inkluderingsfilerna `iostream` och `max.h` innehåller det som behövs för att kompilera `max.cc`
- för att skapa det körbara programmet behövs även definitionen för `max` – kompilering:

```
g++ test.cc max.cc          eller stegvis:    g++ -c test.cc
g++ -c max.cc              g++ -c max.cc
g++ test.o max.o           g++ test.o max.o
```

Uppdelning på filer – "file as module"

Program kan delas upp på flera filer, typiskt inkluderingsfiler med datatypdefinitioner och funktionsdeklarationer (*.h) tillhörande implementeringsfiler (*.cc, *.cpp, ...) med exempelvis funktionsdefinitioner

```
// Fil: max.h
#ifndef MAX_H
#define MAX_H

int max(int, int);

#endif
```

```
// Fil: max.cc
#include "max.h"

int max(int x, int y)
{
    return (x > y ? x : y);
}
```

- inkluderingsfilen `max.h` innehåller det som behövs för att kompilera `max.cc`
- en *inkluderingsgard* ska alltid finnas i en inkluderingsfil – skyddar mot upprepad inkludering
 - en *deklaration* kan förekommer flera gånger – genererar ingen kod
 - en *definition* som förekommer mer än en gång ger kompileringfel (gäller ej mallar, template)

Datatyper i C++

Variabler och *konstanter* används för att hantera värden – fungerar som behållare för värden.

Deklareras med namn och en datatyp – traditionell syntax:

```
char c;                // teckenvariabel c
int i = 1;             // heltalsvariabel i som initieras till 1
const double PI = 3.14; // flyttalskonstant PI
```

Alternativ syntax i C++11 för att initiera – den mest generella syntaxen:

```
int i{1};
const double PI{3.14};
```

Ytterligare en variant från C++98 – ibland måste denna användas för klasser med lite speciella konstruktörer:

```
int i(1);
const double PI(3.14);
```

Två huvudkategorier av datatyper:

- *grundläggande datatyper* ("fundamental types")
 - exempelvis **int**, **double**, **char**, **bool** och lite speciella **void**
- *sammansatta datatyper* ("compound types")
 - exempelvis *klasstyper* (**struct**, **class**), *fält* ("array"), *pekare*, *referenser* och *funktioner*

Variabel- och konstantdeklARATIONER

```

const double PI{3.1415}; // konstant på filnivå – "global" (undvik)
void print_area(double r) // funktionsparameter – initieras av argumentet
{
    const string text{"Areal = "}; // lokal konstant
    cout << text << PI * r * r << endl;
}
int main()
{
    double radius; // lokal variabel
    cout << "Ge en cirkelradie: ";
    cin >> radius;
    print_area(radius); // funktionsanrop, parameteröverföring
}

```

- variabler som deklaras på filnivå initieras alltid – numeriska variabler nollställs automatiskt om ingen initierare anges
- lokala variabler initieras inte – `radius` har ett odefinierat initialvärde

Anm. Datamedlemmar i klasser återkommer vi till.

Datatypning

Typlikhet och typomvandling är viktiga egenskaper hos ett programmeringsspråk.

- C++ är ett starkt typat språk
- varje objekt/uttryck har en specifik typ, som ej kan ändras

"Uppluckring" – värden av grundläggande typ kan typomvandlas automatiskt i många sammanhang:

- "conversions" – omvandlingar som kan leda till problem om man inte är medveten om vad man gör (**int** → **short int**)
- "promotions" – säkra omvandlingar (**short int** → **int**)

Uttrycklig typomvandling kan göras med *typomvandlingsoperatorer*:

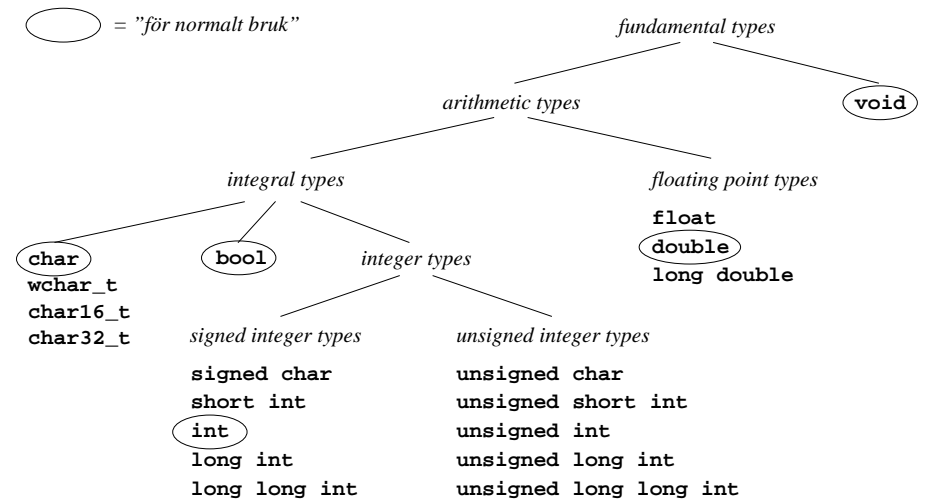
```

static_cast<till-typ>(x) till relaterad typ (t.ex. mellan olika aritmetiska typer)
reinterpret_cast<till-typ>(x) till ej relaterad typ (t.ex. heltal till pekare)
dynamic_cast<till-typ>(x) inom en polymorf klasshierarki (t.ex. från basklasspekare till subclasspekare)
const_cast<till-typ>(x) tar bort const-het

```

- dessa kan inte användas hur som helst – kompilatorm kontrollerar!

Översikt grundläggande typer



Uttryck och operatorer

Möjligheterna att konstruera uttryck är omfattande. Några enkla exempel

<code>x = 21</code>	tilldelningsuttryck
<code>2 * PI * r</code>	multiplikation, sammansatt uttryck
<code>-x</code>	negation
<code>x < y</code>	relationsuttryck
<code>line[21]</code>	indexeringsuttryck (t.ex. <code>vector</code> , <code>fält</code>)
<code>sin(pi)</code>	funktionsanrop
<code>name.length</code>	medlemsval (class , struct , union)
<code>p->length</code>	medlemsval när ett klassobjekt refereras av pekare
<code>&x</code>	adresstagning
<code>*p</code>	avreferering av pekare
<code>static_cast<int>(x)</code>	typomvandling

Sammanfatta uttryck – prioritet – beräkningsordning

Då sammansatta uttryck beräknas inverkar flera regler i språket.

- alla operatörer är inordnade i en prioritetshierarki (17-19 nivåer)
- en operatör med högre prioritet har företräde jämfört med en operatör med lägre prioritet

```
a + b * c          beräknas      a + (b * c)
```

- då operatorerna i ett sammansatt uttryck har samma prioritet gäller den aktuella nivåns beräkningsriktning, *associativitet* (höger → vänster eller vänster → höger)

```
a + b - c + d      beräknas      ((a + b) - c) + d
```

```
x = y = z = 0      beräknas      x = (y = (z = 0))
```

- parenteser kan användas för att ge en annan beräkningsordning

```
(a + b) * c
```

- beräkningsordningen för *operander* är i de flesta fall ospecificerad, liksom beräkningsordningen för funktionsargument

```
x + y - fun()
```

- se upp med funktioner som har sidoeffekter (fun kanske ändrar på x och y i det fördolda)

Satser

Det finns ett flertal olika slags satser:

- sammansatt sats, blocksats – {...}
- deklarasatser
- uttryckssatser – *uttryck*;

```
{
    int i;           // sammansatt sats
    int j{0};       // deklarasatser

    i = j + 1;      // uttryckssats (tilldelningsuttryck följt av semikolon)
    ...
}
```

- selektionssatser – **if**, **switch**
- repetitionssatser (iterationssatser) – **while**, **do**, **for**
- hoppssatser – **return**, **break**, **continue**, **goto**

If-satsen

Villkorsstyrt val av alternativa programavsnitt.

```
int main()
{
    int x, y;

    cout << "Ge två heltal: ";
    cin >> x >> y;

    cout << "Det största värdet är ";

    if (x > y)
        cout << x << endl;
    else
        cout << y << endl;
}
```

- *styrvillkoret* mellan parenteserna beräknas först, **true** eller **false**
- om värdet är **true** utförs satsen mellan **if** och **else**
- om värdet är **false** utförs satsen efter **else**
- **else**-grenen är valfri (används endast vid behov) – enkel **if**-sats
- om uttrycket inte är ett **bool**-uttryck kan automatisk typomvandling komma att ske

```
if (x = y) ...           // en klassiskt felskrivning
```

Blocksats – om mer än en sats ska finnas där syntaxen säger "sats"

```
if (top > 0)
{
    top = top - 1;
    return stack[top];
}
else
{
    cout << "Stacken är tom\n";
    throw stack_error;           // ett undantag kastas – en passande hanterare letas upp
}
```

- satserna inom ett block utgör en *sekvens* – de utförs i den ordning de skrivits
- behovet av blocksatsen gäller för samtliga styrsatser
- man bör alltid använda block i villkors- och iterationssatser

Switch-satsen

Ett *mångförgrenat* val styrt av ett *heltal*uttryck.

```
int main()
{
    char c;

    cout << "Välj j = JA, n = NEJ: ";
    cin >> c;

    switch (c)
    {
        case 'j':
        case 'J':
            cout << "Du har valt JA";
            break;
        case 'n':
        case 'N':
            cout << "Du har valt NEJ";
            break;
        default:
            cout << "Fel val!";
            break;
    }
}
```

Switch-satsen, forts.

- *styruttrycket* beräknas
 - ska vara ett **int**-kompatibelt uttryck – **char** omvandlas automatiskt till **int**
- om det finns en **case** med samma värde fortsätter exekveringen efter denna **case**
- om ingen **case** passar utförs **default**
 - **default** behöver inte placeras sist men det är mest logiskt så
- **break** avbryter exekveringen av **switch**, liksom om exekveringen når avslutande '}'
- varning för "fall-through"!

While-satsen

While-satsen är en *förtestad*, *villkorsstyrd* repetition.

```
int main()
{
    int sum{0};
    int x;

    while (cin >> x)
    {
        sum = sum + x;
    }

    cout << "summa = " << sum << endl;
}
```

- styruttrycket beräknas först
 - om **true** utförs satsen/satsblocket, varefter styruttrycket åter beräknas
 - **while**-satsen avbryts då styruttrycket blir **false**

Do-satsen

Do-satsen är en *eftertestad*, *villkorsstyrd* repetition.

```
int main()
{
    int x;

    do
    {
        cout << "Skriv ett heltal: ";
        cin >> x;
        cout << "Du skrev " << x << endl;
    }
    while (x > 0);
}
```

- satsen/satsblocket mellan **do** och **while** utförs
- styruttrycket mellan parenteserna efter **while** beräknas
 - om styruttrycket är *falskt* avbryts **do**-satsen och satsen efter utförs
 - om styruttrycket är *sant*, upprepas från första punkten ovan

For-satsen

Den "vanliga" for-satsen är en *förtestad, villkorsstyrd* repetition.

```
int main()
{
    int sum{0};

    for (int i = 1; i <= 10; ++i)
    {
        sum = sum + i;
    }

    cout << "Summan av 1, 2,... 10 är " << sum << endl;
}
```

- *initieringssatsen* beräknas – variabeln *i* deklaras och nollställs
- *styrvillkoret* beräknas – om *falskt* avslutas **for**-satsen
- satsen/satserna som hör till **for**-satsen utförs
- sista uttrycket i **for**-satsens styrdel utförs – *i* räknas upp med 1 – varefter styrvillkoret åter beräknas
- konstruktionen av styrningen ovan kan kategoriseras som en "räknarstyrd repetition"

Intervallstyrd for-sats

Om det man vill iterera över är någon form av **intervall** – *container, fält, initierarlista, ...*

```
int x[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 }; // fält med 9 initierade element

for (auto i : x) // motsvarar: for (int i : x)
{
    cout << i << '\n'; // i är en kopia av ett element i x
}

for (auto& i : x) // motsvarar: for (int& i : x)
{
    i = 2 * i; // i är en referens till ett element i x
}
```

Om *x* är

- ett *fält* (som ovan) kommer **for**-satsen att stega från första till (förbi) sista elementet
- av *klass* ska medlemsfunktionerna *begin()* och *end()* finnas som returnerar pekare/iteratorer till första resp. förbi sista elementet
- av annan typ ska det finnas vanliga funktioner *begin()* och *end()* som returnerar pekare/ iteratorer till första resp. förbi sista elementet

Semantiskt hanteras styrvariabeln *i* som en lokal variabel *inuti* satsdelen

- återskapas och initieras *i* varje varv
- nödvändigt för fallet då *i* är en *referens* – en referens måste alltid initieras då den skapas och kan inte tilldelas nytt värde

Hoppsatser

Det finns fyra former av hoppsatser i C++:

- **goto**
 - innebär hopp till ett namngivet läge i programmet
 - undantagshantering (exception) har eliminerat en klassisk användning av **goto**
- **return**
 - returnerar från funktion
 - **return** uttryck för funktioner som ska returnera ett objekt
 - enbart **return** för funktioner som returnerar **void**
 - en funktion som når sitt slut utan att passera en **return**-sats returnerar automatiskt – fatalt om ett värde ska returneras!
- **break**
 - avbryter **for**, **while**, **do** och **switch**
 - exekveringen fortsätter efter satsen ifråga
 - nödvändig i **switch**
 - använd med omdöme i repetitionssatser – bör medföra en påtaglig förenkling
- **continue**
 - medför att hopp till slutet av satsdelen i en **for**, **while** eller **do**
 - för en **while**- eller **do**-sats innebär det att exekveringen hoppar till styruttrycket
 - för en **for**-sats innebär det att exekveringen hoppar till det tredje delen i styrhuvudet och sedan till styruttrycket
 - använd med omdöme

Funktionsdefinition och funktionsdeklaration

- Funktionsdefinition

```
double mean(double x, double y)
{
    return ((x + y) / 2);
}
```

- kod genereras
- det får bara finnas en definition

- Funktionsdeklarationer

```
double mean(double x, double y);

double mean(double, double);
```

- ingen kod genereras – anger bara att *mean* ska vara namnet på en funktion med en viss signatur
- det får finnas flera sådana deklarationer
- kan användas av compilatorn för att kontrollera funktionsanrop
- definitionen (i kompilerad form) måste finnas tillgänglig vid länknigen

Parameteröverföring

Värdeöverföring – "call by value" – argumentets värde kopieras till parametern

```
int max(int x, int y)
{
    return (x > y ? x : y);
}
```

Referensöverföring – "call by reference" – parametern kommer att referera till (vara ett annat namn för) argumentet
– lvalue reference

```
void swap(int& x, int& y)
{
    const int old_x{x};
    x = y;
    y = old_x;
}
```

– rvalue reference – parameterns innehåll kan "stjälas" (flyttsemantik), argumentet nollställs

```
void fun(vector<int>&& v);
```

Överföring som konstant referens – för att undvika onödig kopiering av objekt av komplicerad typ

```
void print(const vector<int>& v);
```

Defaultargument för funktionsparametrar

Används om inget motsvarande argument ges i ett anrop.

```
int increment(int value, int amount = 1)
{
    return value + amount;
}
```

Anrop:

```
cout << increment(5, 2) << endl;    // 7
cout << increment(7) << endl;      // 8
```

Överlagring av funktioner

Flera funktioner kan ges samma namn.

```
void print(int value);
void print(double value);
void print(const string& s);
```

Kompilatorn väljer den överlagring som passar argumenttypen bäst.

```
print(4711);    // 4711 är int
```

Överlagring som alternativ till defaultargument (se exemplet på föregående sida)

```
int increment(int value)
{
    return value + 1;
}

int increment(int value, int amount)
{
    return value + amount;
}
```

- två nästan identiska definitioner – i detta fall är defaultargument att föredra

Sammansatta datatyper

Sammansatta datatyper kan konstrueras på många sätt – några exempel:

- fält ("array") är indexerade datatyper bestående av objekt av samma typ – använd hellre `std::vector` eller `std::array`

```
int a[100]                                Typen för a justeras ibland till int*
```

- pekare till objekt av en viss typ

```
int* p
```

- referens till objekt av en viss typ – den med ett & kallas lvalue-referens – vi väntar med rvalue-referens (&&)

```
int& r
```

- klass (**class** och **struct**) som består av objekt av olika typ, även funktioner

```
struct point
{
    int x;
    int y;
};
```

- funktioner har parametrar av olika typer och returnerar **void** (inget) eller ett objekt av en viss typ

```
int max(int x, int y)                    Typ: int (int, int)
```

Sammansatta datatyper kan kombineras på olika sätt för att konstruera komplexa datatyper och datastrukturer.

Pekare – *

Pekarobjekt kan innehålla minnesadresser till andra objekt.

```
int i{4711}; // Heltalsvariabel
int* p{&i}; // Pekare till heltal – adressoperatorm
```

Användning:

```
cout << i << endl; // 4711
cout << p << " -> " << *p; // 0xeffff828 -> 4711
```

Med adressoperatorm & kan man bl.a. få adressen till

- en variabel eller konstant, ett element i ett fält, en komponent i en post, ...

Avrefereringsoperatorm * är dess "invers", "avrefererar" en pekare

- uttrycket ger det objekt som en pekare/adress anger

Pekare är viktiga komponenter för att skapa dynamiska datastrukturer – mer om det senare.

Referenser

Det finns två slags referenser i C++ – *lvalue-referenser* (&) och *rvalue-referenser* (&&).

```
int i{4711};
int& ir{i}; // referens till heltal – ir blir ett alternativt namn på i
```

En referens måste *alltid* initieras då den definieras och kan sedan inte ändras.

Möjlig användning (kanske inte så realistiskt):

```
ir = 11147; // i princip tilldelning av i
cout << ir << "==" << i << endl; // implicit avreferering
```

En form av *alias* – främsta användning är som typ för funktionsparametrar och funktionsreturvärden.

```
void add(int& x, int y)
{
    x = x + y;
}

add(i, 21);
```

Undvik jämförelse med pekare – skillnaderna är stora, likheterna små!

Poststrukturer – struct

Den grundläggande användningen av **struct** är att definiera, framför allt enkla, *posttyper* (inga medlemmar som behöver skyddas)

```
struct Person
{
    string name;
    int age;
};

Person p{ "Per Post", 21 }; // initiering med initierarlista
```

Punktoperatorm används för att referera till komponenter:

```
p.age = p.age + 1;
cout << p.name << " är " << p.age << " år.\n";
```

Det kommer mera längre fram ...

Containrar – vector

Containrar är datastrukturer för att lagra element av en viss typ – array, deque, forward_list, list och vector är sekventiella containrar.

- **vector** har dynamisk kapacitet som utökas vid behov av operationen `push_back()`, som sätter in sist.
- indexierbar
- har iteratorer – objekt som kan pekar på element i en vector och "vet" hur det tar sig till nästa/föregående element

```
#include <vector>

vector<int> v; // initialt tom
int x;

while (cin >> x)
    v.push_back(x); // sätt in sist – kapaciteteten ökas vid behov

for (unsigned i = 0; i < v.size(); ++i) // stegning med traditionell indexering
    v[i] = v[i] + 1;

for (auto it = v.begin(); it != v.end(); ++it) // stegning med iterator – fungerar som pekare
    *it = *it + 1;

for (auto x : v) // intervallstyrd – elementens värden ska inte ändras
    cout << x << '\n';

for (auto& x: v) // intervallstyrd – elementens värden ska ändras
    x = 2 * x + 1;
```


Exempel på operationer på vector

```
vector<int> v1;           // defaultinitiering - v1 är tom från början
vector<int> v2(100);     // initierar v2 med 100 defaultinitierade element av typen int - () måste användas här!
vector<int> v3{v2};     // kopieringsinitiering - initierar v3 som en kopia av v2

v.front()              // returnerar det första elementet i v
v.back()               // returnerar det sista elementet i v
v[i]                   // returnerar elementet i indexposition i
v.pop_back()           // tar bort sista elementet i v, returnerar inget
v.insert(it, x)        // sätter in x direkt före iteratorpositionen it, returnerar iterator till det nya elementet
v.erase(it)            // raderar elementet i iteratorpositionen it, returnerar iterator till elementet som följer efter det raderade elementet
v.clear()              // raderar samtliga element i v
v.empty()              // returnerar true om v är tom
v.size()               // returnerar antalet element som lagras i v (storleken)
v1 = v2                // tilldelning, innehållet i v1 ersätts av en kopia av innehållet i v2
v1 == v2               // returnerar true om innehållet i v1 är lika med innehållet i v2; != för att jämföra med avseende på olikhet
v1 < v2                // returnerar true om elementen i v1 är lexikografiskt mindre än v2; <=, > och >= för analoga jämförelser
v1.swap(v2)           // byter innehåll på v1 och v2
swap(v1, v2)          // byter innehåll på v1 och v2
begin(v)               // returnerar iterator till första elementet i v eller, om v är tom, en "förbi-slutet-iterator"
end(v)                 // returnerar en "förbi-slutet-iterator"
```

En del funktioner, som insert och erase, finns i flera varianter.

Exempel: vector med Person-element

```
#include <vector>
#include "Person.h"
using namespace std;

int main()
{
    vector<Person> pv;
    Person p;

    while (getline(cin, p.name, '\n')) // Inmatningsformat: förnamn efternamn ålder
    {
        cin >> p.age >> ws; // ws för att nästa getline ska börja läsa från rätt plats i indata
        pv.push_back(p);
    }

    for (auto x : pv)
    {
        cout << x.name << " är " << x.age << " år.\n";
    }

    return 0;
}
```

Länkade datastrukturer – länkad lista: struct + pekare

```
struct List_Node
{
    std::string name;
    int age;
    List_Node* next;
};
```

Dynamisk minnestilldelning görs med operatoren **new**:

```
List_Node* p{ new List_Node{"", 0, nullptr} }; // objektet listinitieras (endast för "aggregate")
```

Operatorm -> används för åtkomst till medlemmar:

```
p->name = "Per Post";
p->age = 21; // Samma som (*p).age men det används normalt inte
```

Minne skapat med **new** ska återlämnas med **delete**:

```
delete p; // pekaren p oförändrad
p = nullptr; // om p lever vidare bör den alltid tilldelas ett nytt pekarvärde
```

Anm. Ett "aggregate" är antingen ett *fält* eller en **struct** med endast publika datamedlemmar, inga egendefinierade konstruktörer, inga basklasser, inga virtuella funktioner och inga initierare för datamedlemmarna (NSDMI; tillåtet i C++14). *Fö 3–6.*

Aliasdeklaration

Ett sätt att definiera enkla namn för sammansatta typer.

```
struct List_node
{
    std::string name;
    int age;
    List_Node* next;
};

using List = List_Node*; // aliasdeklaration - att slippa asterisken och få ett enkelt namn: List

List head{nullptr}; // deklaration av personlista - head har typ Person*
```

Det enkla namnet **List** har ofta klara fördelar framför det sammansatta typnamnet **List_Node***.

```
void insert(List& list, const string& name, int age); // i.s.f. List_Node*&
```

Ett alternativ till aliasdeklarationen ovan är **typedef**-deklaration (före C++11):

```
typedef List_Node* List;
```

Sätta in sist i lista

```

void append(List& list, const string& name, int age)
{
    auto p = new List_Node{name, age, nullptr};    // typ List_Node*

    if (list == nullptr) // tom lista
    {
        list = p;
        return;
    }

    // Det finns minst en nod i listan, sök upp den sista noden och länka in den nya noden efter den sista

    auto last = list;                            // typ List_Node*

    while (last->next != nullptr)
    {
        last = last->next;
    }

    last->next = p;
}

```

Användning av strömbiblioteket †

Det finns flera inkluderingar som hör till strömbiblioteket, t.ex.

```

#include <iostream>
- strömklasserna istream, ostream och iostream
- de fördefinierade standardströmmarna cin, cout, cerr och clog
- manipulatorer utan parametrar, t.ex. endl, noskipws, oct

#include <iosfwd>
- i stället för <iostream> om man bara använder typdefinitioner i <iostream>
- man kan deklarerat exempelvis funktionsparametrar av typ ostream& men inte använda t.ex. cin och cout

#include <fstream>
- strömklasserna ifstream, ofstream och fstream
- för att hantera filer som strömmar
- fstream inkluderar i sin tur istream

#include <sstream>
- strängströmsklasserna istringstream och ostringstream

#include <iomanip>
- parametriserade manipulatorer, t.ex. setw(10), setprecision(2)

```

Strömbiblioteket †

Ett klassbaserat bibliotek för att läsa och skriva strömmar (sekvenser av element av en viss typ, t.ex. tecken).

- Strömtyper (klasser)
 - inströmmar – istream, ifstream
 - utströmmar – ostream, ofstream
 - in- och utströmmar – istream, fstream
 - strängströmmar – istringstream, ostringstream
 - datatyper för att deklarerat strömvariabler som kan kopplas till filer för att läsa/skriva sådana
- Operationer för
 - att öppna och stänga strömmar
 - att läsa och skriva strömmar (text, binärt)
 - att testa strömmars tillstånd (filslut, feltillstånd)
 - manipulera textströmmar (in- och utmatningsformat, sätta in radslut, strängslut, etc.)

† Sidorna 37–46 går inte igenom på föreläsning – ingår för kort sammanfattning av grundläggande egenskaper hos strömmar.

Skriv- och läsoperatorerna << och >> †

Kan användas på standardströmmarna, filströmmar och strängströmmar för formaterad in- och utmatning.

```
cout << x;
```

- används för att *skriva* grundläggande datatyper och strängar på utströmmar
- omvandling från intern representation till textrepresentation

```
cin >> x;
```

- används för att *läsa* grundläggande datatyper och strängar från inströmmar
- läsning sker på "fritt format"
 - inledande "vita tecken" läses förbi
 - läser sedan tecken så länge tecknet kan ingå i värdet för den typ av variabel inläsningen sker till
 - om inget tecken kan läsas sätts en feflagga (failbit) i strömmen och strömmen läses för vidare läsning (kan läsas upp)
- omvandling från text till interna representation

Funktioner för att läsa och skriva tecken †

```
char c;
is.get(c);
    - läs nästa tecken (ingen förbiläsning av blanka) från inströmmen is till variabeln c
is.putback(c);
    - lägg tillbaka tecknet i variabeln c i inströmmen is
c = is.peek();
    - hämta nästa tecken i inströmmen is utan att ta ut det
is.ignore(255, '\n');
    - läs förbi maximalt 255 tecken i inströmmen is eller till radslut om det inträffar dessförinnan
os.put('A');
    - skriv tecknet 'A' i utströmmen os
```

Formatering †

Används för att ange önskat dataformat vid utskrifter. Några exempel:

```
os.width(n);
    - sätter fältvidden för nästa, men endast nästa, utskrift på utströmmen os till n teckens bredd
os.fill(c);
    - sätter utfyllnadstecknet till c för os (mellanrumstecken är standard)
os.precision(p);
    - sätter önskad precision (antalet decimaler för flyttal) till p för strömmen os
```

För var och en av dessa finns en motsvarande parameteriserad *manipulator*.

Funktioner för att läsa och skriva strängar †

```
string s;
getline(cin, s);
```

C-strängar

```
char buf[80];
is.read(buf, 80);
    - läs 80 tecken från is och lagra i buf
os.write(buf, 20);
    - skriv de 20 första tecknen i buf på os
is.getline(buf, 80);
    - läs is till radslut, eller max 80 tecken (eventuella inledande vita tecken läses normalt förbi)
is.getline(buf, 80, ';');
    - läs is tills ett semikolon dyker upp, dock max 80 tecken
```

Manipulatorer †

Det finns två kategorier av manipulatorer.

- Utan parametrar – direkt tillgängliga genom inkludering av `iostream`, exempel:


```
ws      endl      right    oct      dec      hex
```
- Parameteriserade – kräver att även `omanip` inkluderas, exempel:

```
setbase(b)
setw(n)
setfill(c)
setprecision(p)
```

Används med läs- och skrivoperatörer, t.ex.:

```
cin >> ws >> x; // läs förbi eventuella vita tecken innan x läses
cout << setw(20) << right << x << endl; // x ska skrivas ut högerjusterat i fält om 20 positioner
```

Vissa manipulatorers effekt gäller endast för närmast efterföljande strömoperation.

Statusinformation för strömmar †

Operationer för att kontrollera och hantera tillstånd hos strömmar.

`!s` – returnerar sant om strömmen `s` är i fel tillstånd

```
if (!cin)
{
    cerr << "Fel i cin" << endl;
    exit(1);
}
```

`s.eof()` **true** om filslut inträffat för strömmen `s`

`s.fail()` **true** om ett mindre allvarligt fel inträffat för strömmen `s` – `s` ska kunna återställas med `s.clear()`

`s.bad()` **true** om ett allvarligt fel inträffat för strömmen `s`

`s.good()` **true** om inga fel inträffat för strömmen `s`

`s.clear()` återställer strömmen `s` till normaltillstånd – ska kunna göras efter `s.fail()`

Inströmmar kan fås att kasta undantag då något går fel, i stället för att tyst gå i lås:

```
cin.exceptions(ios_base::failbit | ios_base::badbit);
```

Efter detta kommer undantaget `ios_base::failure` att kastas om man t.ex. matar in något som ej stämmer med vad som förväntas.

Filströmmar †

Används för att läsa och skriva filer på sekundärminne.

- `ifstream` – för filströmmas som enbart ska *läsas*
- `ofstream` – för filströmmas som enbart ska *skrivas*
- `fstream` – för filströmmas som ska både *läsas* och *skrivas*
- operationer
 - öppna filström
 - stänga filström
 - positionering i filström
 - allmänna strömoperationer för läsning och skrivning
- inkludera `<fstream>`

Öppna och stänga filströmmar †

Två möjligheter att öppna en fil.

- öppna filen direkt i samband med att ett filströmobjekt deklaras.

```
ifstream input("DATA.TXT");

if (!input)
{
    cout << "filen DATA.TXT kunde inte öppnas\n";
    return 1;
}
```

- deklarerar först filströmmen och öppnar sedan filen med operationen `open()`

```
ifstream input;
...

input.open("DATA.TXT");
```

En filström stängs med operationen `close()`

```
input.close();
```