TDDI82 - Objektorienterad problemlösning

Christoffer Holm

Institutionen för datavetenskap

LINKÖPING UNIVERSITY

1. Kursupplägg
2. STL
3. Behållare
4. Associativa behållare
5. Iteratorer

LINKÖPING UNIVERSITY

1 Kursupplägg
2 STL
3 Behållare
4 Associativa behållare
5 Iteratorer

LINKÖPING UNIVERSITY

Kursupplägg

Mål

- C++: Standardbiblioteket
- C++: Mallar
- Projekt: Lösa och formulera problem med objektorientering
- Projekt: Skapa objektorienterad design
- Etik: Redgöra och analysera etiska aspekter inom datateknik

Kursupplägg

Kursmoment

- Labbserie
- Etikseminarier
- Projekt
- Tentamen

Kursupplägg

Kursmoment

- Labbserie
  - 2 laborationer
  - ungefär 2 veckor
  - Möjlighet för bonus på tentan
  - Storseminarie - Föreläsning + labbpass
- Etikseminarier
- Projekt
- Tentamen

Kursupplägg

Kursmoment

- Labbserie
- Etikseminarier
  - 2 seminarier
  - Separat lärare (se kurshemsida)
- Projekt
- Tentamen

Kursupplägg

Kursmoment

- Labbserie
- Etikseminarier
- Projekt
  - Objektorienterat projekt i C++
  - Oftast ett spel, men kan vara annat också
  - 4 personer per grupp
  - Verktyg såsom git och make
- Tentamen

Kursupplägg

Kursmoment

- Labbserie
- Etikseminarier
- Projekt
- Tentamen
  - 2 mall uppgifter
  - 2 STL uppgifter
  - Måste klara minst en av varje
  - Betyg baseras på tid *och* antal lösta uppgifter
  - Visst överlapp med laborationer

Kursupplägg

Planering

- Denna kurs har högre tempo än TDIU20
- Planera er tid: utnyttja ingenjörsprofessionalismen
- Tänk på att projektet är det stora arbetet i kursen

Kursupplägg

Laborationer

- Textredigering
- Smartpekare

Kursupplägg

Laborationer

- Textredigering
  - Lambda funktioner
  - STL Algoritmer
  - Databehållare
- Smartpekare

Kursupplägg

Laborationer

- Textredigering
  - Lambda funktioner
  - STL Algoritmer
  - Databehållare
- Smartpekare
  - Klassmallar
  - Funktionsmallar
  - Minneshantering

Kursupplägg

Ändringar

- Projektmomentet har omstrukturerats
- Mer tid allokerat för projektet
- Omstrukturering i labserien
- Ny labb för mallar!

Kursupplägg

Laborationer

Registrera er i WebReg så snart som möjligt!

1 Kursupplägg
2 STL
3 Behållare
4 Associativa behållare
5 Iteratorer

LINKÖPING UNIVERSITY

STL

Vad är programmering?

1. Läs in data

Varje program tar emot data på något vis: från filer, terminaler, musklick, nätverk etc.

STL

Vad är programmering?

1. Läs in data
2. Lagra data

Denna data lagras på något vis i programmet, t.ex. i variabler, arrayer, vektorer, länkade listor etc.

STL

Vad är programmering?

1. Läs in data
2. Lagra data
3. Processera data

Program tar data och gör någonting med den: en beräkning, förbereder ett kommando till en databas, skriver ett meddelanden som ska skickas över nätverk, simulerar en spelvärld etc.

STL

Vad är programmering?

1. Läs in data
2. Lagra data
3. Processera data
4. Returnera data

Den färdigprocesserade datan måste sedan kommuniceras till någonting utanför programmet, t.ex. till en terminal, en skärm, en fil etc.

STL

Vad är programmering?

1. Läs in data
2. Lagra data
3. Processera data
4. Returnera data

Steg 1 och steg 4 har vi hittils i utbildningen skött med `iostream` paketet. Vi kommer se – senare i kursen – andra sätt att sköta dessa steg på.

STL

Vad är programmering?

1. Läs in data
2. Lagra data
3. Processera data
4. Returnera data

Steg 2 och steg 3 kan såklart också skötas på olika sätt, och i den första labben i denna kurs ska vi lära oss mer om dessa steg.

STL

Standardbibliotek

- Moduler och komponenter som kommer paketerat med språket
- Varje kompilator har en implementation
- Allting i standardbiblioteket är skrivet i C++
- Används för att förenkla många operationer

STL

Standardbiblioteket i C++ heter **STL**:

Standard **T**emplate **L**ibrary

STL

Varför lära sig STL?

- STL ger *alla* C++ programmerare en gemensam bas
- används för att undvika att uppfinna hjulet på nytt
- fokusera på övergripande problem istället för detaljer
- ger tillgång till effektiva och stabila lösningar

1 Kursupplägg
2 STL
3 Behållare
4 Associativa behållare
5 Iteratorer

Behållare

Hur kan vi lagra data?

- I tidigare kurser har vi lärt oss om statiska arrayer (fält), dynamiska arrayer (`std::vector`) och länkade listor.
- Alla dessa har olika för- och nackdelar och det är viktigt att vi lär oss när man ska använda vad.
- STL har ett antal olika sätt att lagra och strukturera datamängder på.

Behållare

Sekventiella behållare

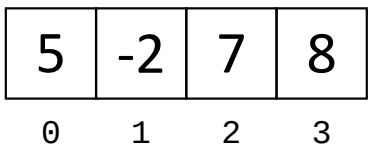
Sekventiella behållare:

- Strukturerarar data i en *sekvens*
- Varje element har ett *index*
- Hittils i utbildningen har vi bara sett hur man lagrar saker sekventiellt
- Både arrayer och länkade listor är sekventiella behållare
- Men de fungerar helt annorlunda

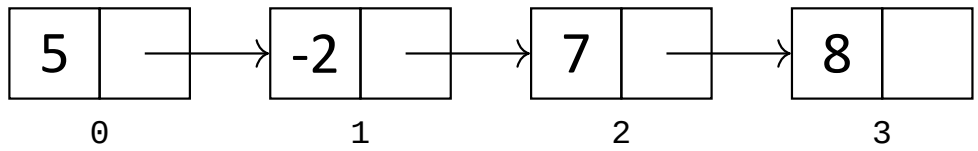
Behållare

Sekventiella behållare

Array



Länkad lista




```
Both contains: 5, -2, 7, 8
```

Behållare

Sekventiella behållare

- Alla element i en array sparas sekventiellt i minnet.
- Detta betyder att om vi vet addressen av första elementet, så kan vi hitta elementet med index `i` genom en enkel beräkning: `*(first + i)`, där `first` är en pekare till första elementet.
- Dock innebär det också att om vi vill stoppa in nya värden i en array så måste vi flytta alla efterföljande element för att skapa en lucka som det nya värdet kan ligga i.

Behållare

Sekventiella behållare

- Varje element i en länkad lista pekar på nästa element.
- Detta betyder att om vi vill stoppa in ett nytt element i listan så kan vi bara ändra pekaren på föregående nod så att den pekar på den nya noden (och ser till att den nya noden pekar på nästa element).
- Dock är det knepigt att hitta element av index `i` eftersom att vi måste starta från början av den länkade listan och följa varje länk tills vi når det sökta elementet. D.v.s. det finns ingen genväg.

Behållare

Sekventiella behållare

Array

+ Lätt att hitta element baserat på index.
– Svårt att stoppa in nya värden mellan element.

Länkad lista

+ Lätt att stoppa in nya värden mellan element.
– Svårt att hitta element baserat på index.

Behållare

Sekventiella behållare

- Som exemplet med array och länkad lista förhoppningsvis demonstrerar så finns det olika för- och nackdelar med olika sätt att lagra data.
- I tidigare kurser har vi implementerat t.ex. länkade listor själva, men i den här kursen använder vi istället redan existerande lösningar från standardbiblioteket (STL).

Behållare

Sekventiella behållare i STL

- `std::array` – Statisk array
- `std::vector` – Dynamisk array
- `std::list` – länkad lista

Behållare

`std::array`

```
 1 #include <array>
 2 #include <iostream>
 3
 4 int main()
 5 {
 6   std::array<int, 5> numbers { };
 7   for (unsigned i { 0 }; i < numbers.size(); ++i)
 8   {
 9     std::cin >> numbers[i];
10     numbers[i] = 2 * numbers[i];
11   }
12 }
```

Behållare

`std::array`

En array innehåller alltid element av *samma* datatyp

```
1  #include <array>
2  #include <iostream>
3
4  int main()
5  {
6    std::array<int, 5> numbers { };
7    for (unsigned i { 0 }; i < numbers.size(); ++i)
8    {
9      std::cin >> numbers[i];
10     numbers[i] = 2 * numbers[i];
11   }
12 }
```

Behållare

`std::array`

Storleken av en statisk array är *konstant* och måste vara känd av kompilatorn

```
 1 #include <array>
 2 #include <iostream>
 3
 4 int main()
 5 {
 6   std::array<int, 5> numbers { };
 7   for (unsigned i { 0 }; i < numbers.size(); ++i)
 8   {
 9     std::cin >> numbers[i];
10     numbers[i] = 2 * numbers[i];
11   }
12 }
```

Behållare

`std::array`

Eftersom att en statisk array aldrig kan ändra storlek så kan man endast läsa...

```
 1 #include <array>
 2 #include <iostream>
 3
 4 int main()
 5 {
 6   std::array<int, 5> numbers { };
 7   for (unsigned i { 0 }; i < numbers.size(); ++i)
 8   {
 9     std::cin >> numbers[i];
10     numbers[i] = 2 * numbers[i];
11   }
12 }
```

Behållare

`std::array`

... och ändra de olika elementen, men aldrig ta bort eller lägga till något.

```
 1 #include <array>
 2 #include <iostream>
 3
 4 int main()
 5 {
 6   std::array<int, 5> numbers { };
 7   for (unsigned i { 0 }; i < numbers.size(); ++i)
 8   {
 9     std::cin >> numbers[i];
10     numbers[i] = 2 * numbers[i];
11   }
12 }
```

Behållare

`std::array`

I det här exemplet vet inte kompilatorn vad `size` är eftersom at det läses in under *körningen* av programmet...

```
 1 #include <array>
 2 #include <iostream>
 3
 4 int main()
 5 {
 6   int size { };
 7   std::cin >> size;
 8
 9   std::array<int, size> numbers { };
10 }
```

Behållare

`std::array`

... därför är detta exempel inte tillåtet, eftersom att kompilatorn måste veta storleken av den statiska arrayen *innan* programmet kör.

```
1  #include <array>
2  #include <iostream>
3
4  int main()
5  {
6    int size { };
7    std::cin >> size;
8
9    std::array<int, size> numbers { };
10 }
```

Otillåtet

Behållare

`std::array`

+ Det mest effektiva sättet att lagra data på.
+ Vi vet mängden element under körning.
– Fixerat antal element, kan inte ta bort eller lägga till.
– Vi måste veta exakta storleken under kompilering.

Behållare

`std::array`

+ Det mest effektiva sättet att lagra data på.
+ Vi vet mängden element under körning.
– Fixerat antal element, kan inte ta bort eller lägga till.
– Vi måste veta exakta storleken under kompilering.

Vi löser dessa problem m.h.a. `std::vector`!

Behållare

`std::array`

+ Det mest effektiva sättet att lagra data på.
+ Vi vet mängden element under körning.
– Fixerat antal element, kan inte ta bort eller lägga till.
– Vi måste veta exakta storleken under kompilering.

Vi löser dessa problem m.h.a. `std::vector`!
Men `std::vector` har vissa problem också...

Behållare

`std::vector`

```
 1 #include <vector>
 2 #include <iostream>
 3 int main()
 4 {
 5   int size { };
 6   std::cin >> size;
 7   std::vector<int> numbers(size);
 8   for (unsigned i { 0 }; i < numbers.size(); ++i)
 9   {
10     std::cin >> numbers[i];
11     numbers[i] = 2 * numbers[i];
12   }
13   numbers.push_back(12);
14 }
```

Behållare

`std::vector`

Likt `std::array` kan vi bestämma storleken av en `std::vector`…

```
 1 #include <vector>
 2 #include <iostream>
 3 int main()
 4 {
 5   int size { };
 6   std::cin >> size;
 7   std::vector<int> numbers(size);
 8   for (unsigned i { 0 }; i < numbers.size(); ++i)
 9   {
10     std::cin >> numbers[i];
11     numbers[i] = 2 * numbers[i];
12   }
13   numbers.push_back(12);
14 }
```

Behållare

`std::vector`

...men `std::vector` kräver inte att storleken är känd under kompilering...

```
1  #include <vector>
2  #include <iostream>
3  int main()
4  {
5    int size { };
6    std::cin >> size;
7    std::vector<int> numbers(size);
8    for (unsigned i { 0 }; i < numbers.size(); ++i)
9    {
10     std::cin >> numbers[i];
11     numbers[i] = 2 * numbers[i];
12   }
13   numbers.push_back(12);
14 }
```

Behållare

`std::vector`

...och med `std::vector` kan vi utöka storleken m.h.a. t.ex. `push_back()`!

```
 1 #include <vector>
 2 #include <iostream>
 3 int main()
 4 {
 5   int size { };
 6   std::cin >> size;
 7   std::vector<int> numbers(size);
 8   for (unsigned i { 0 }; i < numbers.size(); ++i)
 9   {
10     std::cin >> numbers[i];
11     numbers[i] = 2 * numbers[i];
12   }
13   numbers.push_back(12);
14 }
```

Behållare

`std::vector`

- Vi har redan lärt oss om `std::vector` tidigare, så detaljerna utelämnas i denna föreläsning.
- Däremot kan det vara intressant att observera att `std::vector` är väldigt nära besläktad med `std::array`.
- I utbyte mot att `std::vector` är något dyrar i termer av både minnesanvändning och tidsåtgång så får vi ett mer flexibelt sätt att lagra data på.

Behållare

`std::vector`

+ Kan lägga till och ta bort element dynamiskt
+ Otroligt billigt att hitta ett element av specifik index
– Dyrt att lägga till eller ta bort saker (förutom i slutet)
– Är något dyrare att skapa jämfört med `std::array`

Behållare

`std::list`

```
#include <list>
#include <iostream>

int main()
{
  // skapa en tom lista
  std::list<int> numbers { };

  int front { };
  int back { };

  // läs in två tal i taget
  while (std::cin >> front >> back)
  {
    // lägg ena i början och andra i slutet
    numbers.push_front(front);
    numbers.push_back(back);
  }
}
```

Behållare

`std::list`

- `std::list` är standardbibliotekets implementation av en (dubbel[1])länkad lista.
- Med denna behållare är det mycket billigare att lägga till eller ta bort element jämfört med `std::vector` (med undantag för i slutet: där är de ungefär lika snabba).
- Men det är **mycket** dyrare och svårare att hitta element baserat på index.

[1] Dubbellänkad betyder att det finns en pekare framåt och en bakåt i varje nod, vilket tillåter insättning både före och efter noder.

Behållare

`std::list`

- \+ Kan lägga till och ta bort element dynamiskt
- \+ Samma kostnad oavsett vart i listan man lägger till eller tar bort element.
- – Det är svårt och dyrt att hitta element baserat på index.
- – Kan bara ta **ett** steg åt gången (mer om detta senare!)

Behållare

Sammanfattning av sekventiella behållare

Välj `std::array` om:

- Vi vet antalet element innan programmet ens kör.
- Vi *aldrig* behöver lägga till eller ta bort något.

Välj `std::vector` om:

- Det räcker att lägga till eller ta bort element i slutet.
- Vi behöver kunna hitta element baserat på index.

Välj `std::list` om:

- Vi behöver lägga till element vart som helst i sekvensen.
- Vi behöver *inte* hitta element på specifika index.

Om du är osäker: välj `std::vector`!

1 Kursupplägg
2 STL
3 Behållare
4 Associativa behållare
5 Iteratorer

Associativa behållare

Allting måste inte lagras som en sekvens

- Hittils har vi pratat om *sekventiella* behållare.
- Men det finns många tillfällen då lagring baserat på sekvenser och index inte är det som behövs.
- Tänk t.ex. på ordböcker. Vi säger inte att man ”slår upp ordet med index 3789”, istället säger vi att man ”slår upp ordet *programmering*”.
- P.g.a. denna typ av problem vore det trevligt om vi kunde *associera* värden med s.k. *nycklar*. I exemplet ovan skulle våra ord vara *nycklar* och deras översättningar vara *värden*.

Associativa behållare

`std::map`

När man deklarerar en map så måste man ange två saker...

```
1 std::map<std::string, std::string> table { };
```

Associativa behållare

`std::map`

...vilken datatyp nycklarna har...

```
1 std::map<std::string, std::string> table { };
```

Associativa behållare

`std::map`

...och vilken datatyp värdena har!

```
1 std::map<std::string, std::string> table { };
```

Associativa behållare

Vi kan tänka oss en `std::map` som en tabell med två kolumner: nyckel och värde, där varje rad motsvarar en *association*. Initialt är den tom.

`std::map`

```
1 std::map<std::string, std::string> table { };
```

Nyckel	Värde

Associativa behållare

`std::map`

Vi kan lägga till associationer genom att tilldela ett värde till en nyckel, vilket är annorlunda från sekventiella behållare där vi måste lägga till ett element innan vi kan använda `operator[]` för att komma åt det. Detta är för att `std::map` fungerar lite annorlunda. Notera att `operator[]` tar en nyckel (en sträng i det här fallet) och returnerar värdet.

```
1 table["kort"] = "short";
```

Nyckel	Värde

Associativa behållare

`std::map`

Det `operator[]` anropet faktiskt gör är att kontrollera huruvida nyckeln redan finns. Om den finns returnerar den motsvarande värde, men om den inte finns (vilket är vårt nuvarande fall), så lägger den till nyckeln associerat med *standardvärdet* för värdetypen (i detta fall tomma strängen) i tabellen. Efter det finns nyckeln i tabellen och då returneras dess associerade värde...

```
1 table["kort"] = "short";
```

Nyckel	Värde
”kort”	””

Associativa behållare

`std::map`

...som vi då kan tilldela värdet `"short"` och på så vis har vi lagt in associationen `"kort"` med `short"`. I detta fall antar vi att tabellen innehåller associationen svenska ord till engelska ord.

```
1 table["kort"] = "short";
```

Nyckel	Värde
”kort”	”short”

Associativa behållare

På samma sätt som innan kan vi fortsätta lägga in nya värden i vår tabell.

`std::map`

```
1  table["bräde"] = "board";
```

Nyckel	Värde
”kort”	”short”

Associativa behållare

Notera att vi håller nycklarna sorterade, detta är för att vi ska kunna loopa igenom dem i en specifik ordning (se ”Iteratorer” kapitlet).

`std::map`

```
1 table["bräde"] = "board";
```

Nyckel	Värde
”bräde”	”board”
”kort”	”short”

Associativa behållare

`std::map`

Om vi vill ändra ett redan existerande värde så gör vi på exakt samma sätt. I detta fall insåg vi att `"kort"` refererade till spelkort, inte en längd, så vi vill ändra det.

```
1 table["kort"] = "card";
```

Nyckel	Värde
”bräde”	”board”
”kort”	”short”

Associativa behållare

`std::map`

Eftersom att vi använder nycklarna för att hitta specifika värden så betyder det att varje nyckel lagras exakt *en gång* per map, så om vi tilldelar samma nyckel en gång till så uppdaterar vi ett befintligt värde, snarare än att lägga in ett nytt.

```
1 table["kort"] = "card";
```

Nyckel	Värde
”bräde”	”board”
”kort”	”card”

Associativa behållare

`std::map`

Vi kan ju såklart också läsa värdet m.h.a. `operator[]` utan att göra någon ändring på det. I detta fall kommer utskriften bli `board`.

```
1 std::cout << table["bräde"] << std::endl;
```

Nyckel	Värde
”bräde”	”board”
”kort”	”card”

Associativa behållare

Vi kan också ta bort värden m.h.a. `erase()` funktionen, som tar emot nyckeln.

`std::map`

```
1 table.erase("bräde");
```

Nyckel	Värde
”bräde”	”board”
”kort”	”card”

Associativa behållare

Detta anrop tar bort associationen helt från tabellen.

`std::map`

```
1 table.erase("bräde");
```

Nyckel	Värde
”kort”	”short”

Associativa behållare

`std::map`

Det finns jättemycket mer man kan göra med `std::map`, se cppreference:
`https://en.cppreference.com/w/cpp/container/map`

Associativa behållare

`std::set`

En `std::set` motsvarar en matematisk mängd där alla element är *unika*. Man kan också tänka sig att det är en `std::map` där det inte förekommer några värden, endast nycklar.

```
1 std::set<int> set{};
```

Associativa behållare

`std::set`

Vad detta innebär rent konkret är att varje värde antingen finns eller inte finns i mängden. Denna behållare är användbar om vi vill just hålla koll på huruvida ett värde är aktuellt (d.v.s. lagrat i behållaren) eller inte.

```
1 std::set<int> set{};
```

{}

Associativa behållare

`std::set`

Vi använder `insert()` för att stoppa in värden...

```
1 set.insert(4);
```

{}

Associativa behållare

`std::set`

...om värdet inte redan finns i mängden så läggs det till...

```
1 set.insert(4);
```

{4}

Associativa behållare

`std::set`

...men om vi försöker igen...

```
1 set.insert(4); // försöker igen
```

{4}

Associativa behållare

`std::set`

...så ser vi att ingenting händer, för värdet ligger redan i behållaren!

```
1 set.insert(4); // försöker igen -> ingen skillnad
```

{4}

Associativa behållare

`std::set`

Vi lägger till några fler värden. Notera framförallt att dessa värden är *sorterade*, detta av samma anledning som för `std::map`, att det ska finnas en bestämd ordning vi itererar igenom (relevant snart).

```
1 set.insert(3);
```

{4}

Associativa behållare

`std::set`

Vi lägger till några fler värden. Notera framförallt att dessa värden är *sorterade*, detta av samma anledning som för `std::map`, att det ska finnas en bestämd ordning vi itererar igenom (relevant snart).

```
1 set.insert(3);
```

{3, 4}

Associativa behållare

`std::set`

Vi lägger till några fler värden. Notera framförallt att dessa värden är *sorterade*, detta av samma anledning som för `std::map`, att det ska finnas en bestämd ordning vi itererar igenom (relevant snart).

```
1 set.insert(5);
```

{3, 4}

Associativa behållare

`std::set`

Vi lägger till några fler värden. Notera framförallt att dessa värden är *sorterade*, detta av samma anledning som för `std::map`, att det ska finnas en bestämd ordning vi itererar igenom (relevant snart).

```
1 set.insert(5);
```

{3, 4, 5}

Associativa behållare

`std::set`

Vi lägger till några fler värden. Notera framförallt att dessa värden är *sorterade*, detta av samma anledning som för `std::map`, att det ska finnas en bestämd ordning vi itererar igenom (relevant snart).

```
1 set.insert(1);
```

{3, 4, 5}

Associativa behållare

`std::set`

Vi lägger till några fler värden. Notera framförallt att dessa värden är *sorterade*, detta av samma anledning som för `std::map`, att det ska finnas en bestämd ordning vi itererar igenom (relevant snart).

```
1 set.insert(1);
```

{1, 3, 4, 5}

Associativa behållare

`std::set`

Vi lägger till några fler värden. Notera framförallt att dessa värden är *sorterade*, detta av samma anledning som för `std::map`, att det ska finnas en bestämd ordning vi itererar igenom (relevant snart).

```
1 set.insert(2);
```

{1, 3, 4, 5}

Associativa behållare

`std::set`

Vi lägger till några fler värden. Notera framförallt att dessa värden är *sorterade*, detta av samma anledning som för `std::map`, att det ska finnas en bestämd ordning vi itererar igenom (relevant snart).

```
1 set.insert(2);
```

{1, 2, 3, 4, 5}

Associativa behållare

`std::set`

Vi kan såklart också ta bort värden, detta sker på ett liknande sätt som för `std::map` m.h.a. funktionen `erase()`…

```
1 set.erase(3);
```

{1, 2, 3, 4, 5}

Associativa behållare

`std::set`

Där värdet tas bort om det fanns i mängden. Om värdet inte finns så händer ingenting.

```
1 set.erase(3);
```

{1, 2, 4, 5}

Associativa behållare

`std::set`

För att ta reda på om ett element finns i behållaren använder vi `count()` som räknar antalet förekomster av det givna värdet i mängden. Här finns det inte 3 alls, så `count(3)` returnerar `0`...

```
1 set.count(3) == 0;
```

{1, 2, 4, 5}

Associativa behållare

`std::set`

...men om värdet finns (t.ex. 4) så förekommer det exakt en gång, så då returnerar `count(4)` värdet `1`.

```
1 set.count(4) == 1;
```

{1, 2, 4, 5}

Associativa behållare

Viktiga krav

För både `std::map` och `std::set` gäller det att:

- *nycklarna* går att jämföra m.h.a. `operator==`
- *nycklarna* går att sortera m.h.a. `operator<`
- Både *nycklarna* och *värdena* (för `std::map`) går att kopiera.

Associativa behållare

Sammanfattning

- Om vi vill lagra unika (och sorterade) värden, använder vi `std::set`.
- Om vi behöver associera värden med andra specifika värde, använder vi `std::map`.
- Annars räcker sekventiella behållare.

1 Kursupplägg
2 STL
3 Behållare
4 Associativa behållare
5 Iteratorer

Iteratorer

Iteration

- Vi vill kunna loopa igenom våra behållare
- Vore trevligt om vi kan göra det generellt
- Problemet är att alla behållare har inte samma sätt att komma åt element
- Därför måste vi tänka om för att kunna loopa igenom behållare på ett generellt sätt

Iteratorer

Iteration

Det går jättebra att itererar genom `std::vector` och `std::array` m.h.a. en s.k. index-baserad loop. Detta funkar eftersom att vi kan komma åt element via deras index.

```
1  int main()
2  {
3    vector<int> v {1, 2, 3};
4    for (unsigned i{0}; i < v.size(); ++i)
5    {
6      cout << v[i] << endl;
7    }
8  }
```

Iteratorer

Iteration

Däremot så fungerar det **inte** för `std::list` eftersom att vi inte kommer åt element via index. Det fungerar inte heller för `std::set` eller `std::map` eftersom att de inte är sekventiella ö.h.t. Finns det något sätt att loopa igenom *alla* behållare på ett enhetligt sätt?

```
1  int main()
2  {
3    list<int> v {1, 2, 3};
4    for (unsigned i{0}; i < v.size(); ++i)
5    {
6      cout << v[i] << endl; // fungerar ej
7    }
8  }
```

Iteratorer

Iteration

Svaret är ja! Det finns. Denna typ av loop kallas en range-baserad (eller intervall-baserad) loop. Denna loop går igenom elementen en i taget (ordningen specificeras av behållaren). För varje iteration lagras det nuvarande elementet i variabeln e.

```
1  int main()
2  {
3    vector<int> v {1, 2, 3};
4    for (int e : v)
5    {
6      cout << e << endl;
7    }
8  }
```

Iteratorer

Iteration

Som vi ser här så fungerar denna loop även för `std::list`. Den fungerar också för `std::set`, samt `std::map` (där varje element är en `std::pair<NyckelTyp, VärdeTyp>`).
Det finns fler behållare än vi diskuterat i kursen, och även dessa fungerar med range-baserade loopar.

```
1  int main()
2  {
3    list<int> v {1, 2, 3};
4    for (int e : v)
5    {
6      cout << e << endl; // fungerar
7    }
8  }
```

Iteratorer

Range-baserad for-loop

- Man kan skapa egna behållare
- Hur är det då möjligt att få denna generalla loop att fungera för den egen-skapade behållaren?
- Det är här iteratorer kommer in

Iteratorer

Range-baserad for-loop

```
1 for (int e : v)
2 {
3   cout << e << endl;
4 }
```

Iteratorer

Range-baserad for-loop

```
1 using iterator = std::vector<int>::iterator;
2
3 for (iterator it{v.begin()}; it != v.end(); ++it)
4 {
5   cout << *it << endl;
6 }
```

Iteratorer

Range-baserad for-loop

En behållare går att loopa igenom om:

- Det finns en inre klass som heter `iterator`
- Det finns medlemsfunktioner `begin()` och `end()` som returnerar `iterator` objekt
- `iterator` har definierat `operator++`, `operator*`, `operator==` och `operator!=`

Iteratorer

Iteratorer

- Iteratorer är generaliserade pekare
- Ett generellt sätt att iterera över alla behållare på samma sätt
- Pekar på ett element i behållaren
- Möjligt att komma åt elementet med `operator*`
- Gå till nästa element med `operator++`

Iteratorer

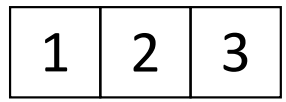
Iteratorer

```
1 vector<int> v{1,2,3};
```

Iteratorer

Iteratorer

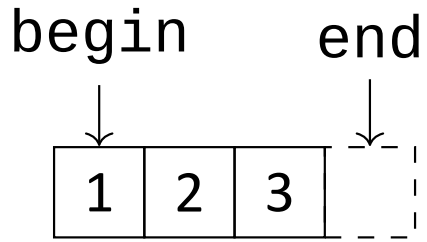
```
1 vector<int> v{1,2,3};
```



Iteratorer

Iteratorer

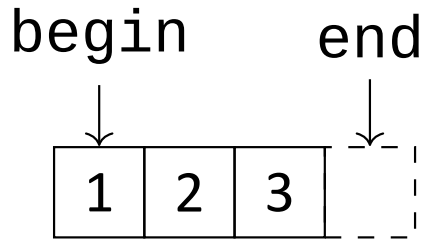
```
1 vector<int> v{1,2,3};
```



Iteratorer

Iteratorer

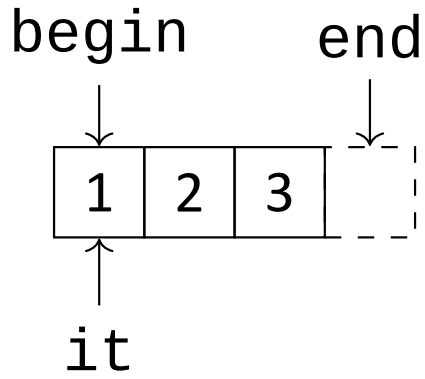
```
1 vector<int>::iterator it{v.begin()};
```



Iteratorer

Iteratorer

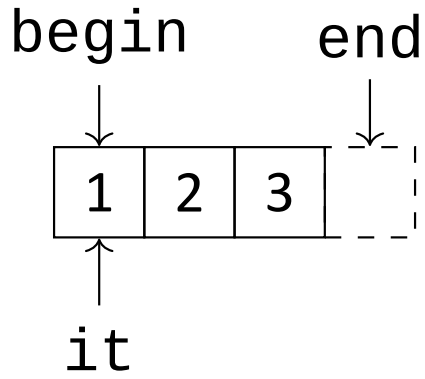
```
1 vector<int>::iterator it{v.begin()};
```



Iteratorer

Iteratorer

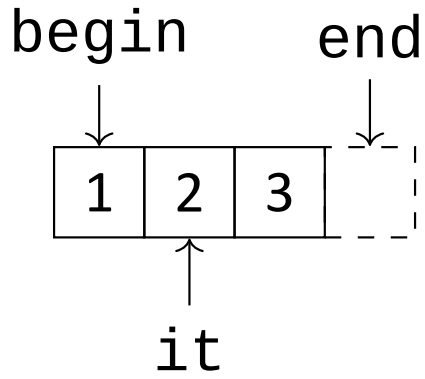
```
1 ++it;
```



Iteratorer

Iteratorer

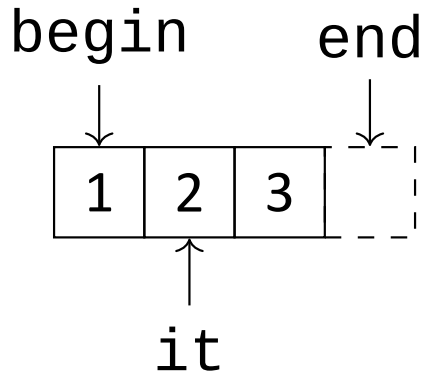
```
1 ++it;
```



Iteratorer

Iteratorer

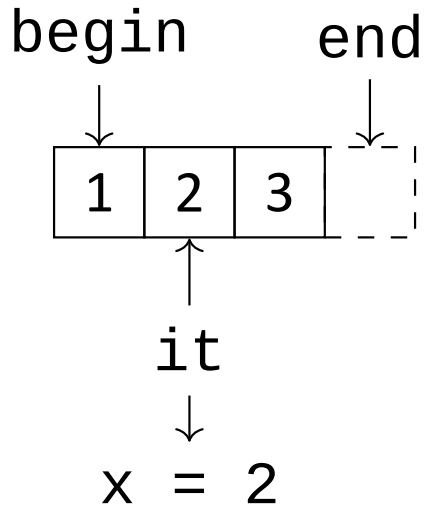
```
1 int x{*it};
```



Iteratorer

Iteratorer

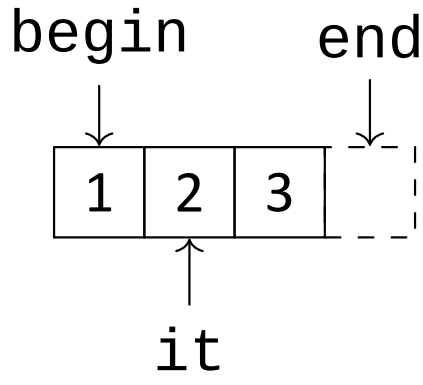
```
1 int x{*it};
```



Iteratorer

Iteratorer

```
1 int x{*it};
```

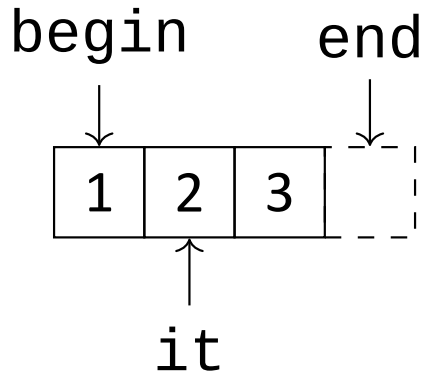


x = 2

Iteratorer

Iteratorer

```
1 ++it;
```

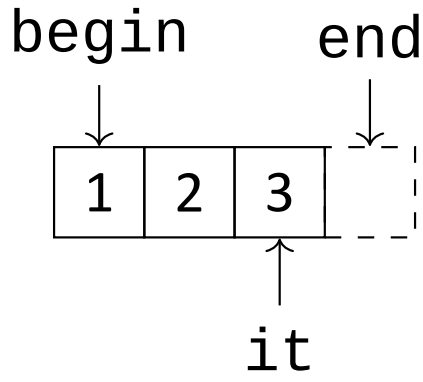


x = 2

Iteratorer

Iteratorer

```
1 ++it;
```

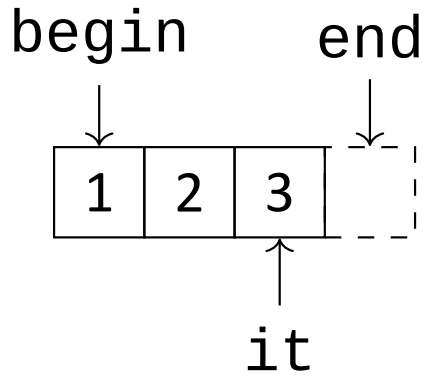


x = 2

Iteratorer

Iteratorer

```
1 *it = 4;
```

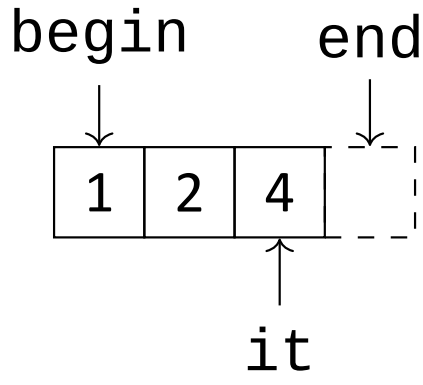


x = 2

Iteratorer

Iteratorer

```
1 *it = 4;
```

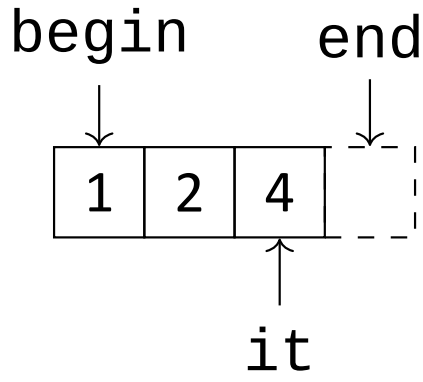


x = 2

Iteratorer

Iteratorer

```
1 ++it;
```

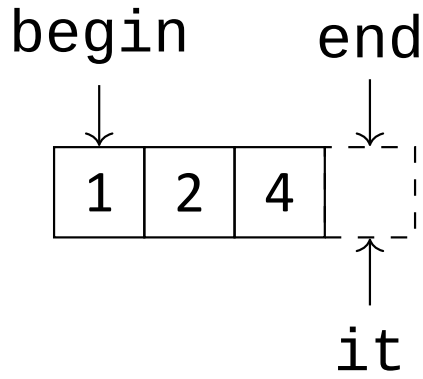


x = 2

Iteratorer

Iteratorer

```
1 ++it;
```

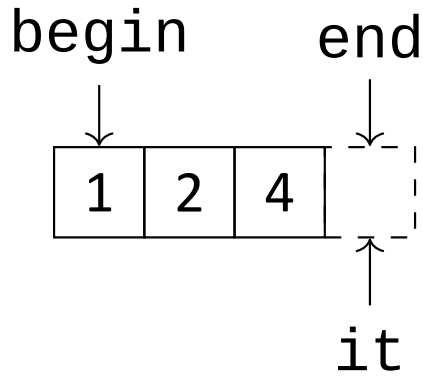


x = 2

Iteratorer

Iteratorer

```
1 it == v.end();
```



x = 2

Iteratorer

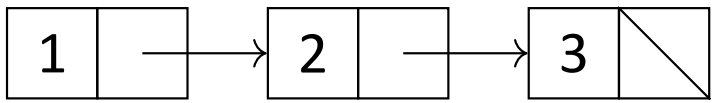
Iteratorer

```
1 list<int> l{1,2,3};
```

Iteratorer

Iteratorer

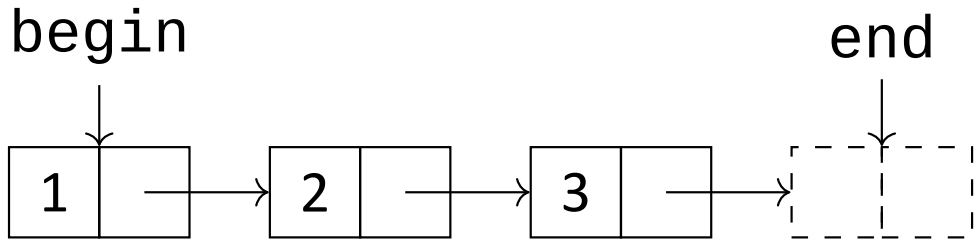
```
1 list<int> l{1,2,3};
```



Iteratorer

Iteratorer

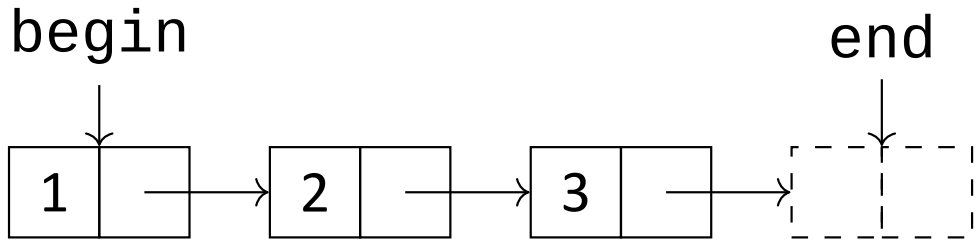
```
1 list<int> l{1,2,3};
```



Iteratorer

Iteratorer

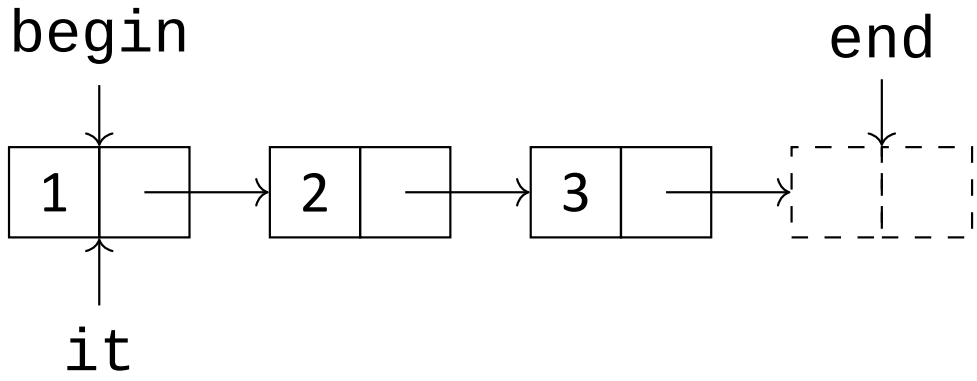
```
1 list<int>::iterator it{l.begin()};
```



Iteratorer

Iteratorer

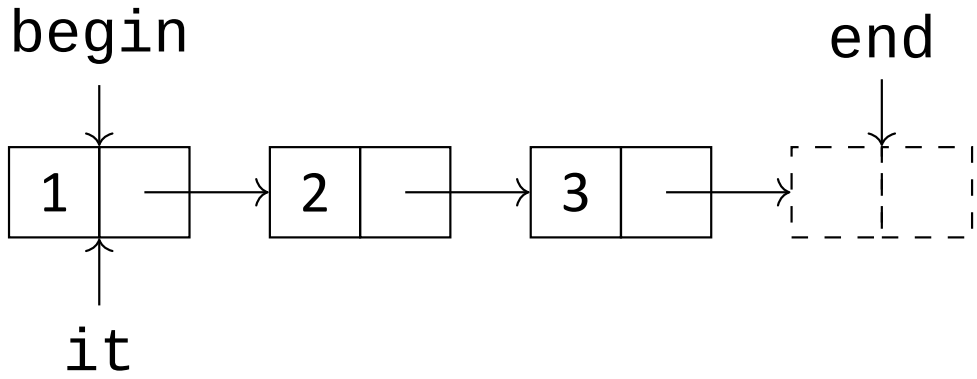
```
1 list<int>::iterator it{l.begin()};
```



Iteratorer

Iteratorer

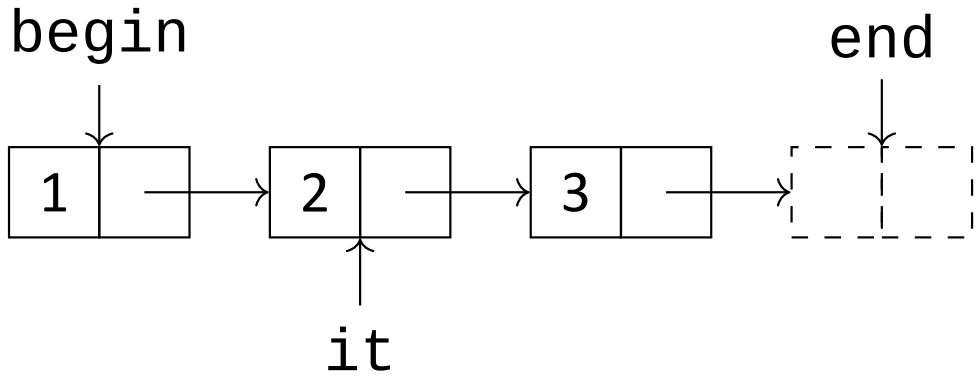
```
1 ++it;
```



Iteratorer

Iteratorer

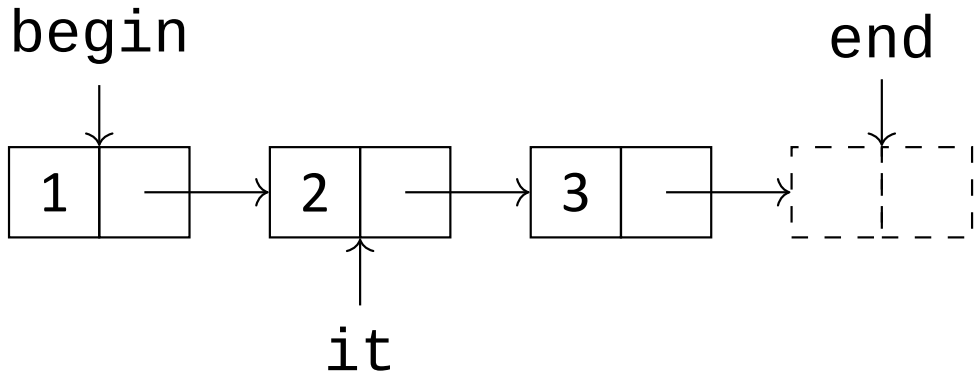
```
1 ++it;
```



Iteratorer

Iteratorer

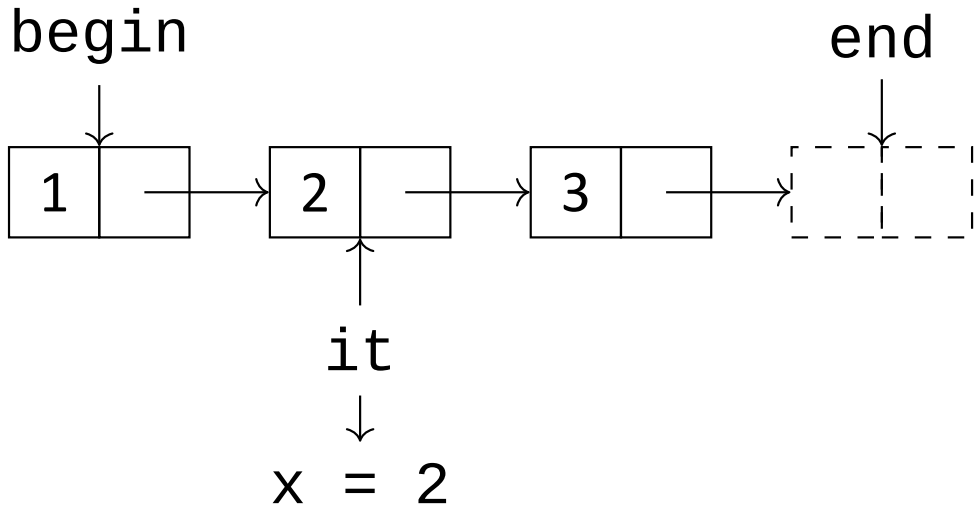
```
1 int x{*it};
```



Iteratorer

Iteratorer

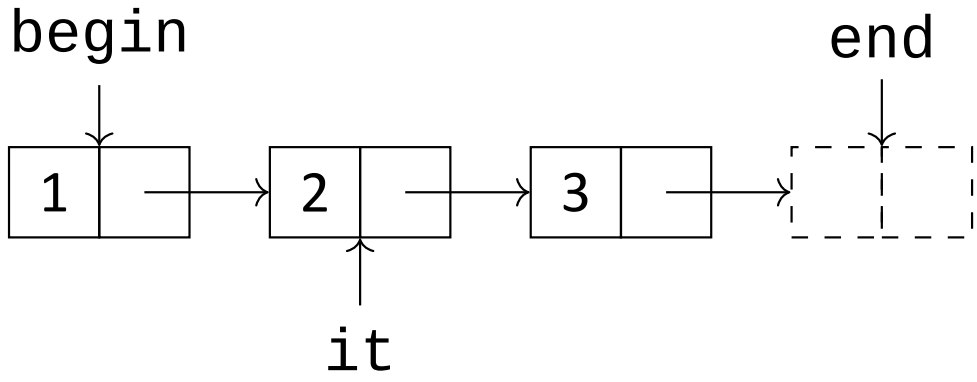
```
1 int x{*it};
```



Iteratorer

Iteratorer

```
1 int x{*it};
```

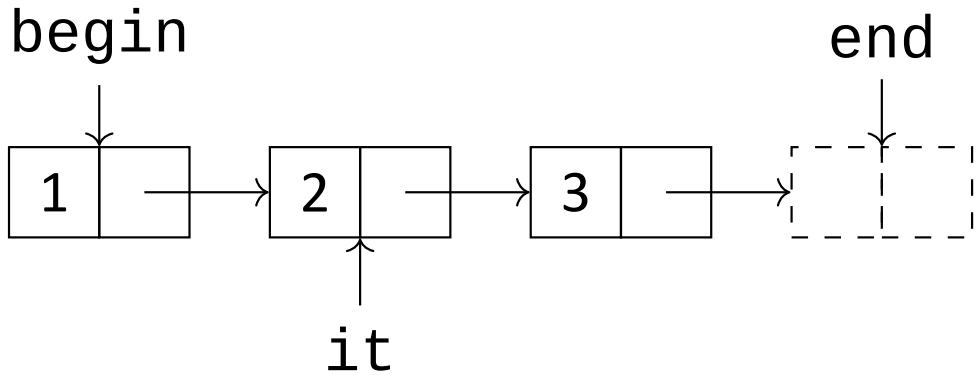


x = 2

Iteratorer

Iteratorer

```
1 ++it;
```

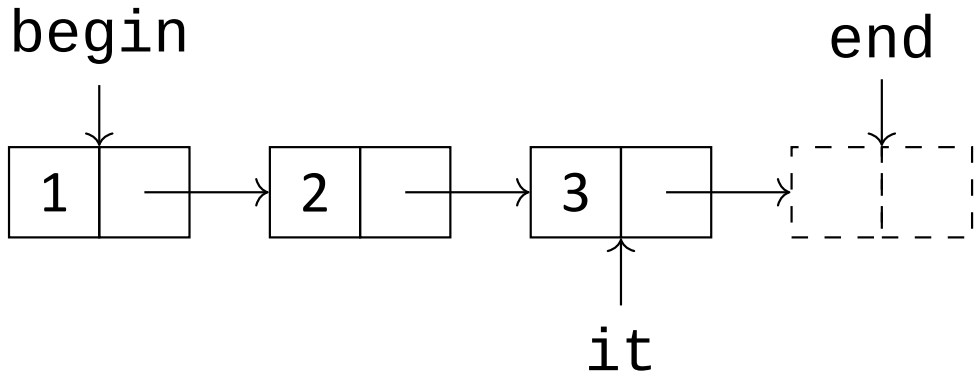


x = 2

Iteratorer

Iteratorer

```
1 ++it;
```

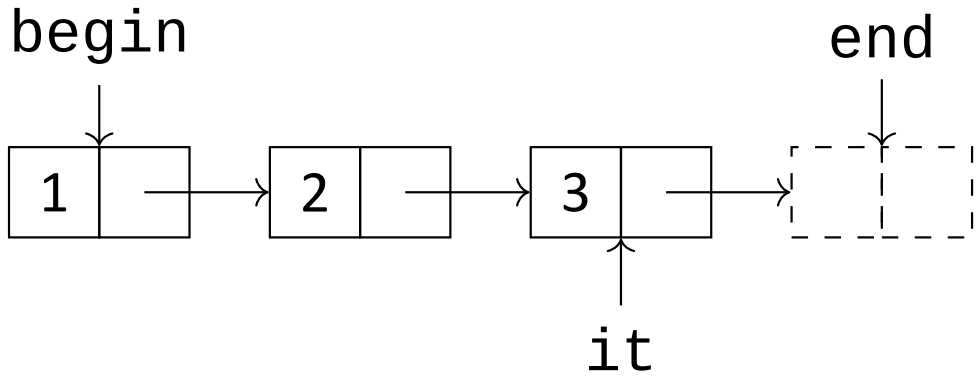


x = 2

Iteratorer

Iteratorer

```
1 *it = 4;
```

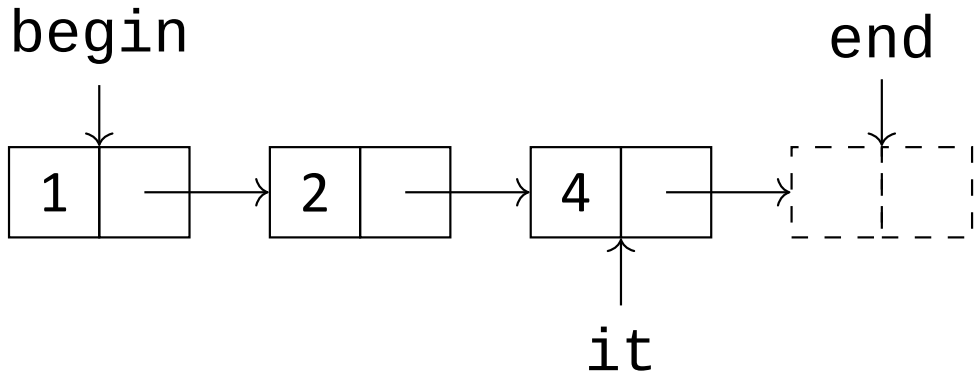


x = 2

Iteratorer

Iteratorer

```
1 *it = 4;
```

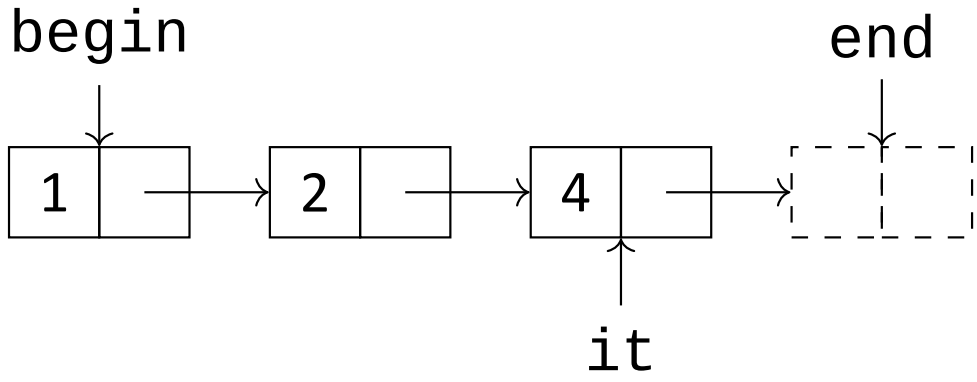


x = 2

Iteratorer

Iteratorer

```
1 ++it;
```

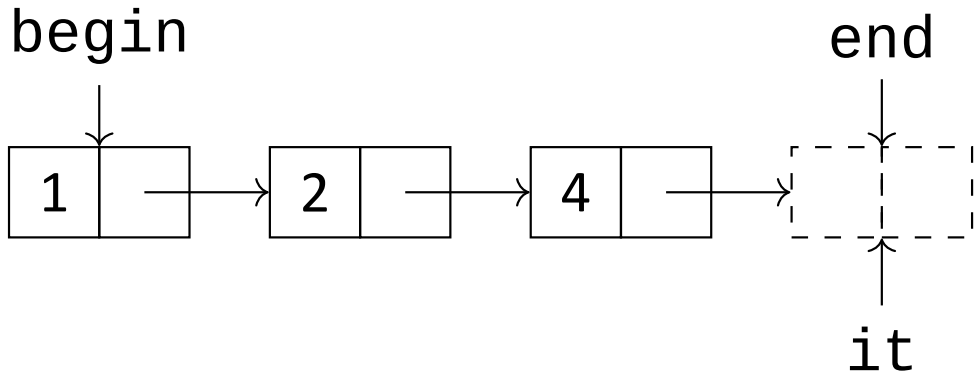


x = 2

Iteratorer

Iteratorer

```
1 ++it;
```

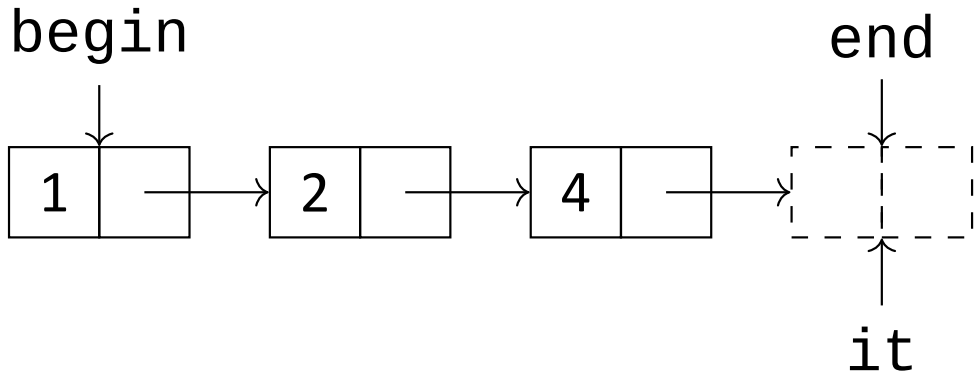


x = 2

Iteratorer

Iteratorer

```
1 it == v.end();
```



x = 2

Iteratorer

Iteratorer

- Det är viktigt att `end`-iteratorn inte pekar på det sista elementet
- Annars kommer vi missa sista elementet i behållaren eftersom att loopen avslutas då `it == v.end()`
- Den måste unikt kunna identifiera att nu är iterationen slut
- Därför tänker vi att den pekar på elementet efter sista

Iteratorer

Iterator kategorier

Det finns olika typer av iteratorer, dessa är:

- Input
  - Kan läsa befintliga värden i behållaren
- Output
- Forward
- Bidirectional
- Random Access

Iteratorer

Iterator kategorier

Det finns olika typer av iteratorer, dessa är:

- Input
- Output
  - Kan lägga till nya värden i behållaren
- Forward
- Bidirectional
- Random Access

Iteratorer

Iterator kategorier

Det finns olika typer av iteratorer, dessa är:

- Input
- Output
- Forward
  - Kan läsa/skriva över befintliga värden i behållaren
  - Kan stega framåt i behållaren
  - Är även en Input iterator
- Bidirectional
- Random Access

Iteratorer

Iterator kategorier

Det finns olika typer av iteratorer, dessa är:

- Input
- Output
- Forward
- Bidirectional
  - Kan stega bakåt i behållaren
  - Är även en Forward iterator
- Random Access

Iteratorer

Iterator kategorier

Det finns olika typer av iteratorer, dessa är:

- Input
- Output
- Forward
- Bidirectional
- Random Access
  - Kan hoppa till godtyckligt element i behållaren
  - Är även en Bidirectional iterator

Iteratorer

Iterator kategorier

Nedan tabell visar vilka operationer som är möjliga för de olika kategorierna

	Kategori				
Operationer	**Input**	**Output**	**Forward**	**Bidirectional**	**Random Access**
==, !=	✓	✓	✓	✓	✓
*, ->	Läsa	Skriva	Läsa/Skriva	Läsa/Skriva	Läsa/Skriva
++	✓	✓	✓	✓	✓
	-	-	-	✓	✓
+, +=, -, -=	-	-	-	-	✓
<, <=, >, >=	-	-	-	-	✓
i[n]	-	-	-	-	✓

www.liu.se

LINKÖPING UNIVERSITY