TDDC76 – Programmering och datastrukturer

Pekare, abstrakta datatyper och speciella medlemsfunktioner

Klas Arvidsson, Eric Ekström

Institutionen för datavetenskap

LINKÖPINGS UNIVERSITET

Allmän info

Datastrukturer och algoritmer

- Datastrukturer och algoritmer ingår i kursen.
- Öppna rätt OpenDSA kursinstans från kurshemsidan!
- https://www.ida.liu.se/~TDDC76/2022/info/opendsa.sv.shtml
- Krävs för UPG3 1.5hp. Se det som kursens ”hemtenta”.
- OBS: List-labben ställer krav på effektivitet!
  - insättning i konstant tid, kopiering i linjär tid, effektiv sortering mm.

Agenda

1. Pekare
2. Exempel: operatorer för avreferering
3. Exempel: dynamisk datastruktur (en länkad lista)
4. Exempel: abstrakt datatyp (för lista)
5. Speciella medlemsfunktioner
6. Exempel: speciella medlemsfunktioner (för lista)

LINKÖPINGS UNIVERSITET

Varför behövs dynamiskt minne?

Fundera på:

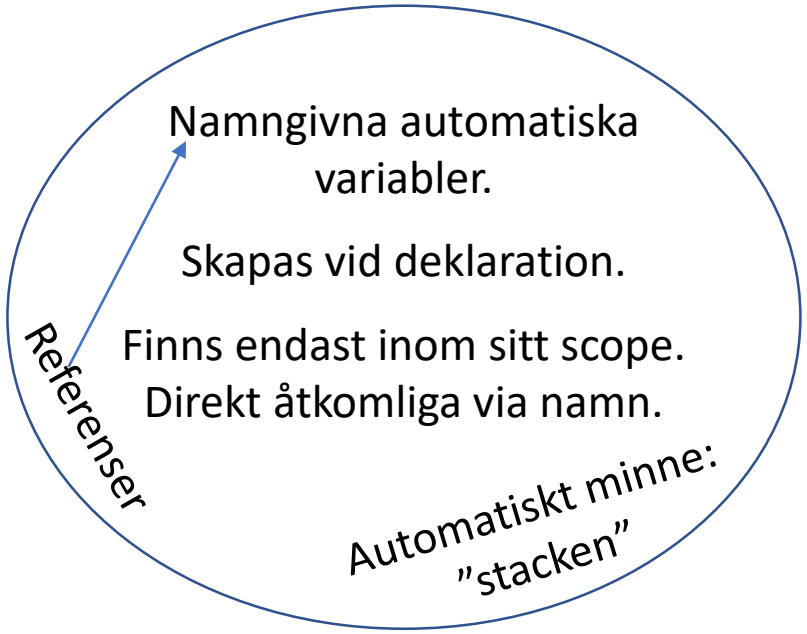
- Skriv en loop som varje iteration skapar en extra variabel i programmet. Efter N iterationer ska det alltså finnas N variabler att använda.
- Skriv en funktion som tar emot ett värde och en (kanske full) behållare. Funktionen ska garantera att värdet finns i behållaren när funktionen är klar.

Du får bara använda vanliga variabler och arrayer vars storlek bestäms vid kompilering.

Alla datatyper vars storlek kan ändras under programkörning använder någonstans dynamiskt minne.

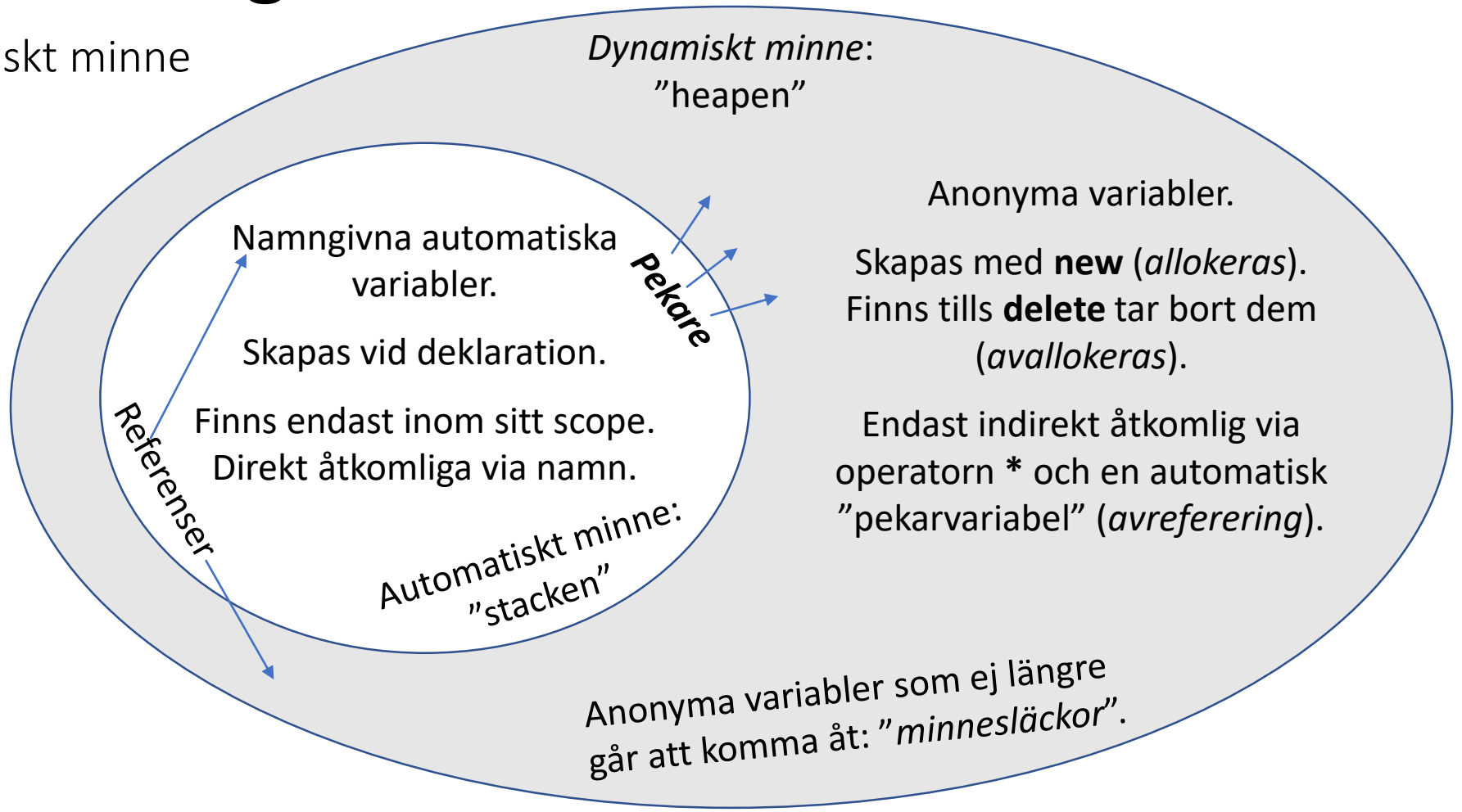
Terminologi

Detta känner vi redan till



Terminologi

Dynamiskt minne



Pekare

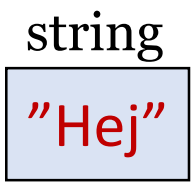
Analogier

- En pekare är en variabel som innehåller en adress
  - Adressen kan leda till en variabel
- Tänk adressen till ett hus
  - Får du adressen kan du hitta till huset
  - Ändrar du något på *huset* kommer ändringen vara där när jag också besöker adressen
  - Ändrar du på *adressen* kan du komma till … annat hus? rivningstomt? mitt i atlanten?
  - Tappar du bort adressen hittar du aldrig till huset igen

Allokera anonyma objekt i minnet med new

Analogi: Blås upp en heliumballong utomhus och släpp den!
Vi kommer inte att få tag på ballongen igen.

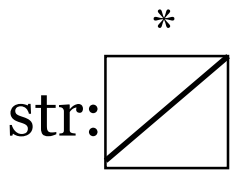
```
str = new string{”Hej”};
```



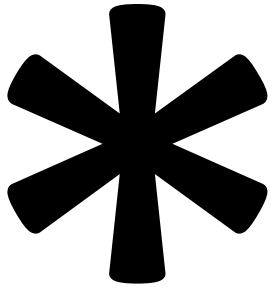
Skapa en pekarvariabel med * som datatyp

Analogi: Hållare för exakt ett ballongsnöre.

```
string* str{};
str = new string{”Hej”};
```



Asterisk – förtydligande



Asterisk efter datatyp

Skapar en pekarvariabel

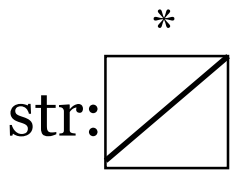
Asterix

En kul typ

En pekarvariabel lagrar en adress, på adressen finns ett objekt av datatypen

Analogi: Ballongen med dess text sitter i andra änden av snöret.

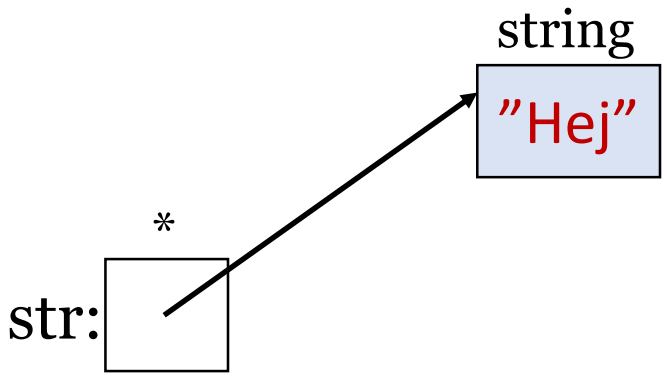
```
string* str{};
str = new string{”Hej”};
```



Spara adressen till ett anonymt objekt

Analogi: Fäst ett snöre i snörhållaren.
Nu kan vi få tag i ballongen så länge hållaren har kvar snöret.

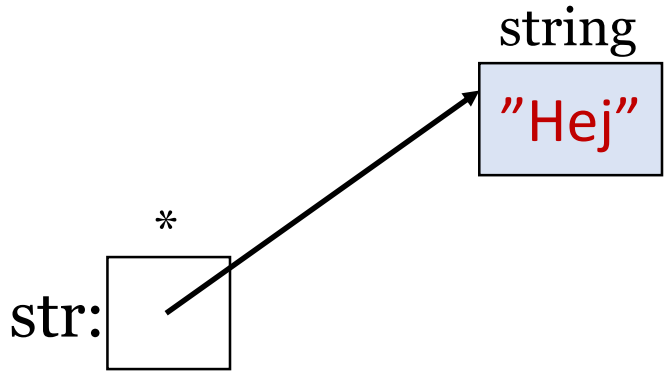
```
string* str{};
str = new string{”Hej”};
```



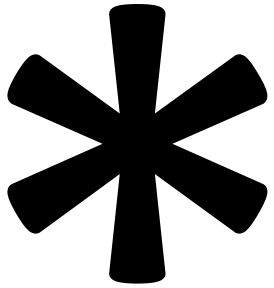
Avreferera ett anonymt objekt med *

Analogi: Hala in ballongen via snöret.

```
string* str{};
str = new string{”Hej”};
*str = ”Hallå”;
```



Asterisk – förtydligande



Asterisk före pekare

Avrefererar (”går till”) objekt i minnet

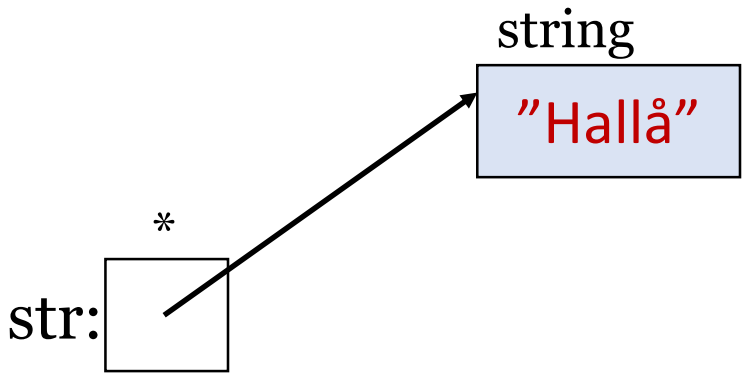
Asterix

Pekar på något

Avreferera ett anonymt objekt med *

Analogi: Hala in ballongen via snöret.
Nu kan du uppdatera texten.

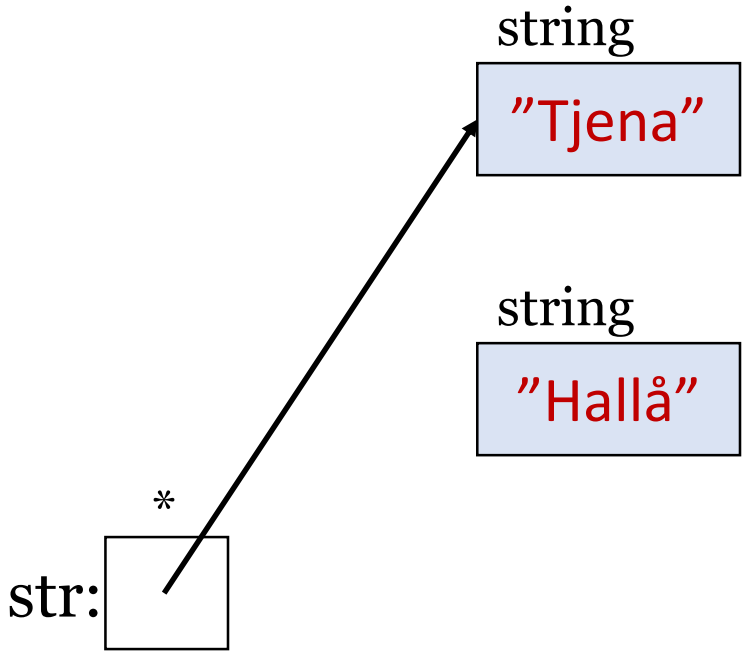
```
string* str{};
str = new string{”Hej”};
*str = ”Hallå”;
```



Minnesläcka!

Analogi: Släpp snöret och sätt ett annat snöre i hållaren.
Den tappade ballongen svävar iväg och går ej nå mer!

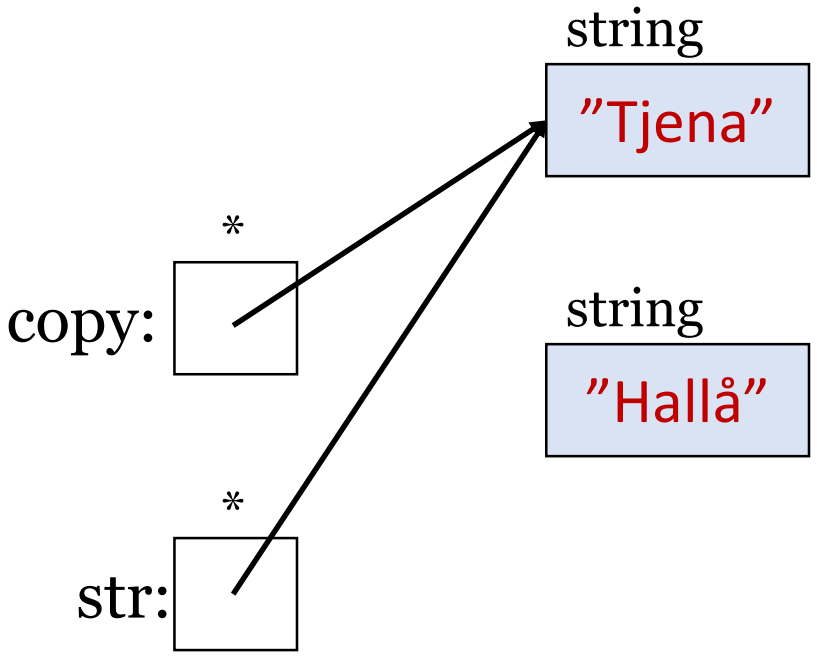
```
string* str{};
str = new string{”Hej”};
*str = ”Hallå”;
str = new string{”Tjena”};
```



Kopiering av pekare!

Analogi: Fäst snöret i en andra(!) snörhållare.

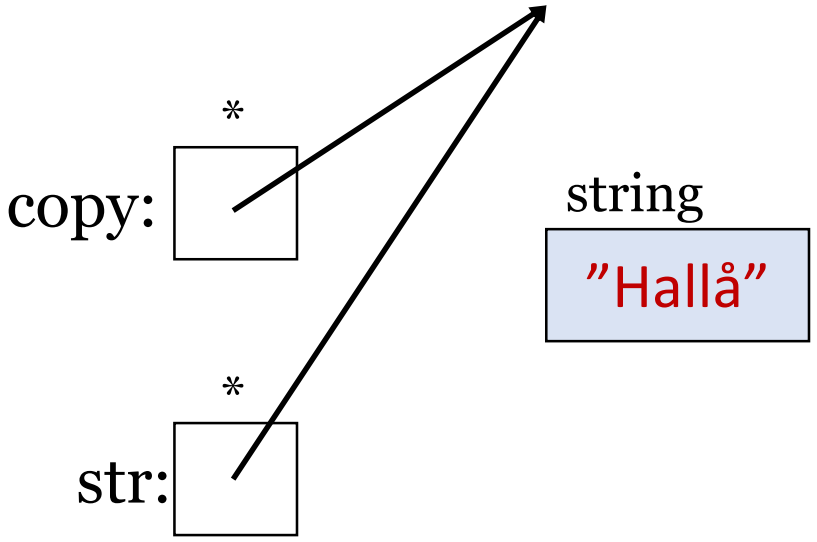
```
string* str{};
str = new string{”Hej”};
*str = ”Hallå”;
str = new string{”Tjena”};
string* copy{ str };
```



Avallokera ett anonymt objekt med delete

Analogi: Hala in ballongen via snöret. Knyt loss den och ta vara på heliumet så det kan återanvändas.

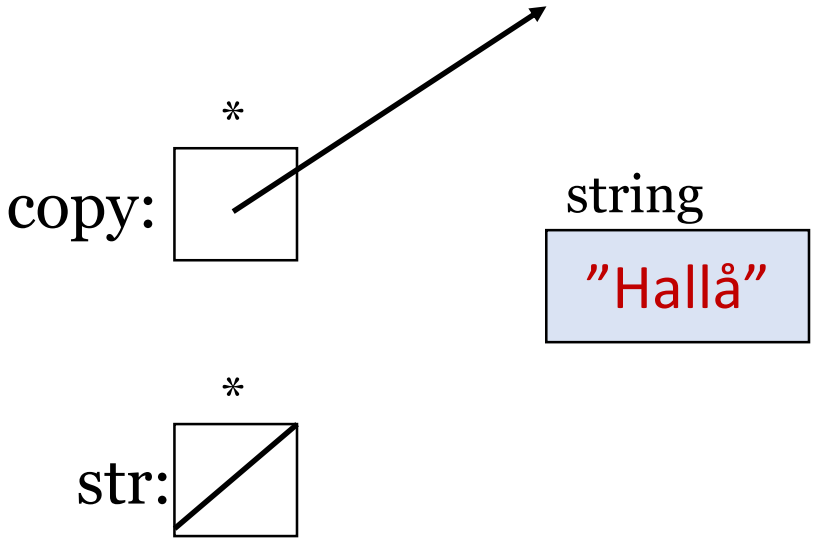
```
string* str{};
str = new string{”Hej”};
*str = ”Hallå”;
str = new string{”Tjena”};
string* copy{ str };
delete str;
```



Markera att en pekare är tom med nullptr

Analogi: Släpp snöret och visa att hållaren är tom.

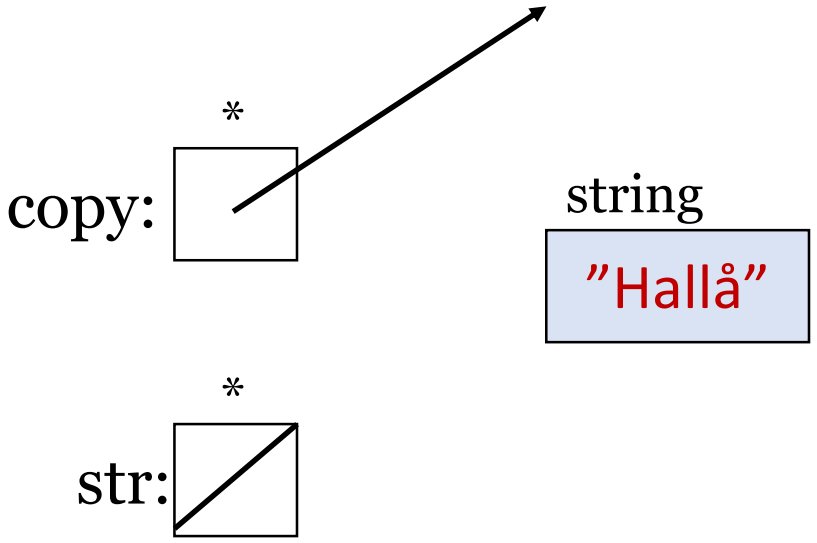
```
string* str{};
str = new string{”Hej”};
*str = ”Hallå”;
str = new string{”Tjena”};
string* copy{ str };
delete str;
str = nullptr;
```



Vad ska vi göra med ”copy”?

Analogi: Vi håller i ett snöre utan ballong!

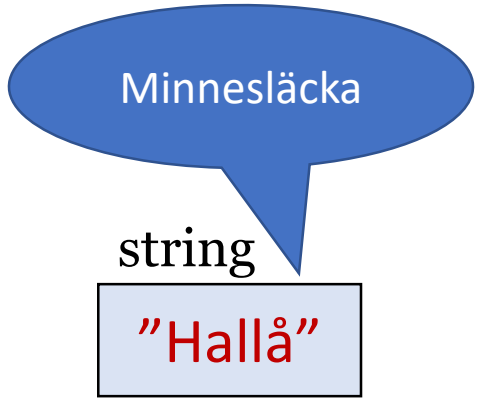
```
string* str{};
str = new string{”Hej”};
*str = ”Hallå”;
str = new string{”Tjena”};
string* copy{ str };
delete str;
str = nullptr;
delete copy; // ?
copy = nullptr; // ?
```



Programmeraren har ansvar för minnet i C++

Analogi: Du måste se till att inget helium går till spillo. Om du släpper en ballong så svävar den iväg upp i atmosfären och spricker.

I andra programspråk hanteras heliumballonger alltid inomhus så att det blir lätt att samla upp tappade ballonger. I C++ är vi däremot alltid utomhus men använder automatik i språket och kunniga programmerare för att inte tappa ballonger.



Agenda

1. Pekare
2. Exempel: operatorer för avreferering
3. Exempel: dynamisk datastruktur (en länkad lista)
4. Exempel: abstrakt datatyp (för lista)
5. Speciella medlemsfunktioner
6. Exempel: speciella medlemsfunktioner (för lista)

LINKÖPINGS UNIVERSITET

Pekare

Vi behöver en klass som exempel

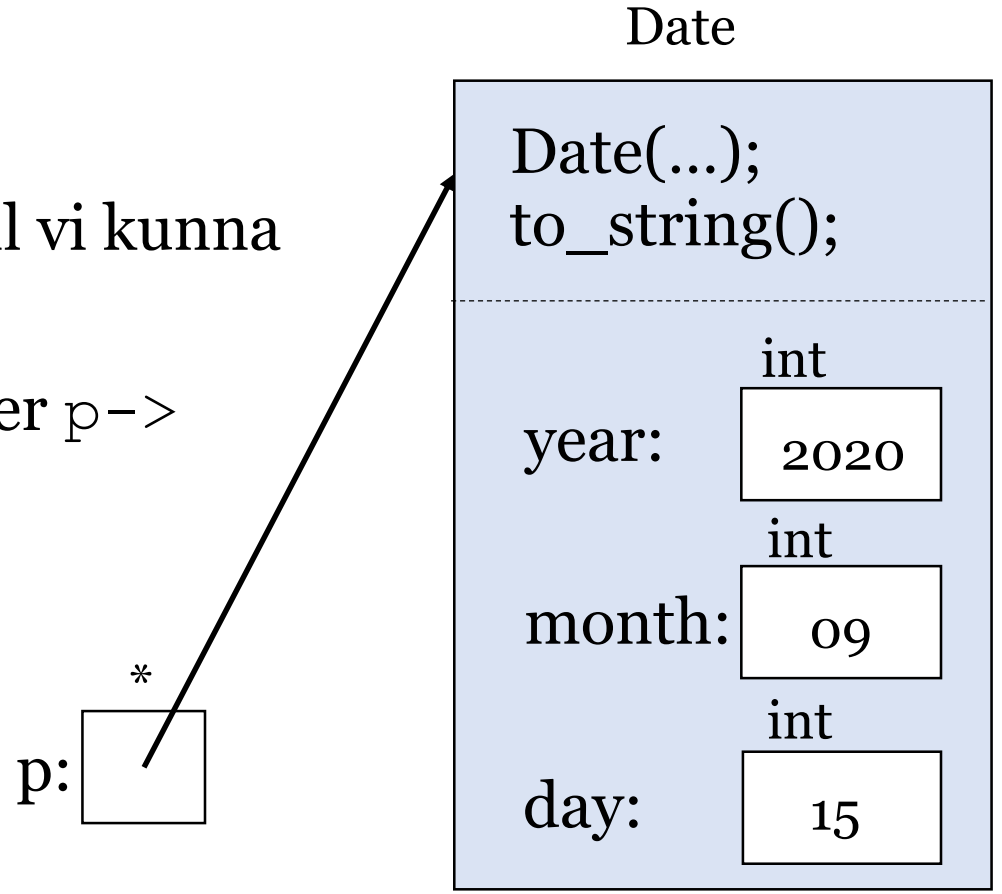
```
class Date
{
public:
  Date( int y = 1970, int m = 1, int d = 1 );

  std::string to_string() const;

private:
  int year;
  int month;
  int day
};
```

Pekare

Pekar-operatorer

- Säg att pekaren pekar på en klass, då vill vi kunna komma åt dess medlemmar
- Avreferera och välj medlem: `(*p).` eller `p->`

```
Date* p{ new Date{2020, 9, 15} };

cout << p->to_string() << endl;
cout << (*p).to_string() << endl;
```

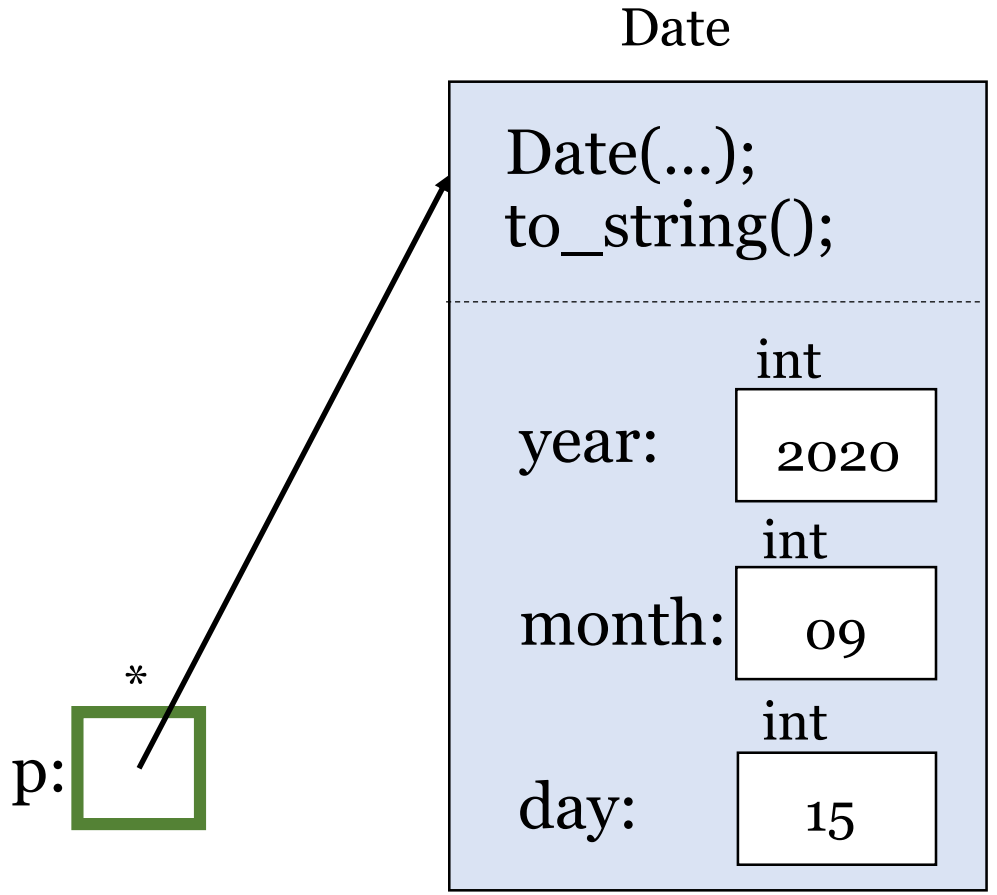


Pekare

Pekar-operatorer

- Avreferering steg för steg

```
(*p).year
```

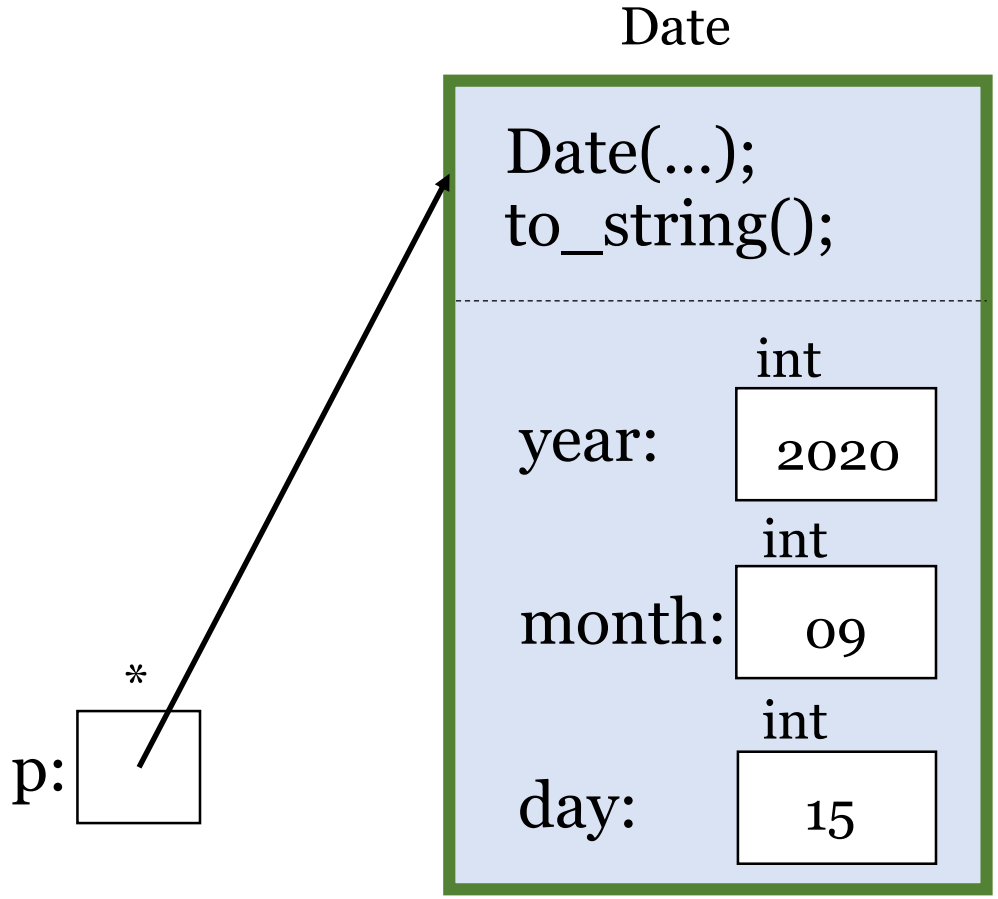


Pekare

Pekar-operatorer

- Avreferering steg för steg

```
(*p).year
```

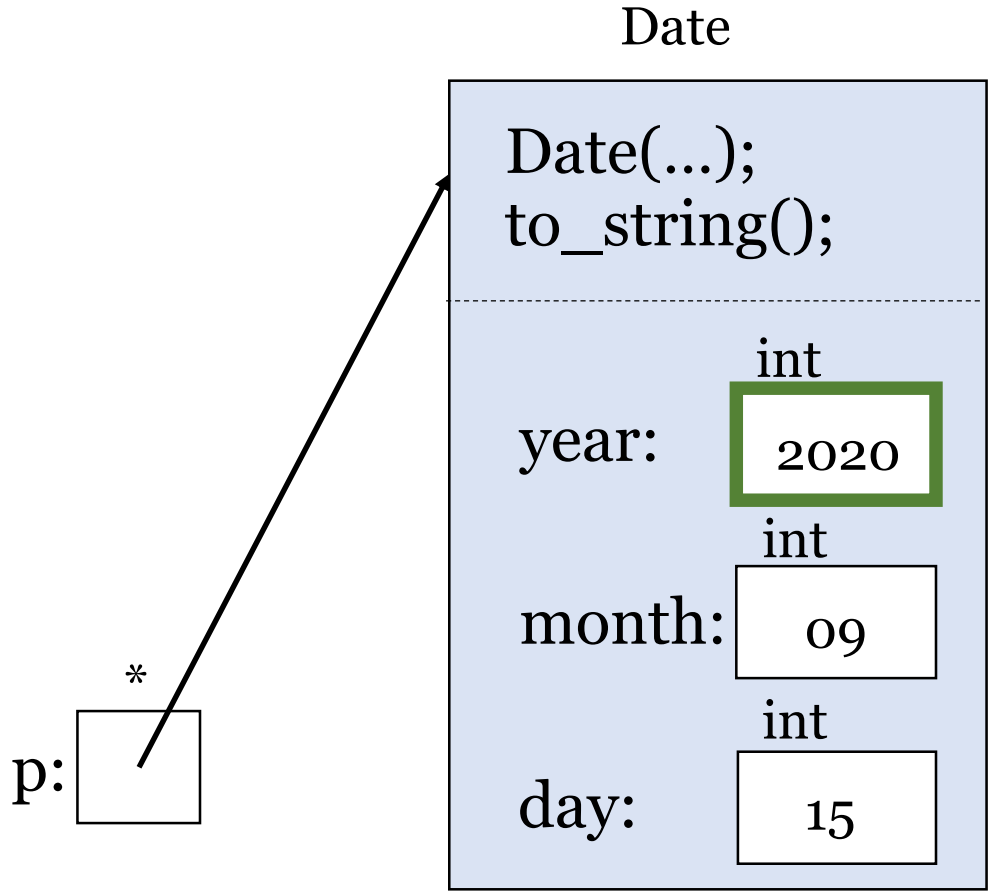


Pekare

Pekar-operatorer

- Avreferering steg för steg

```
(*p).year
```

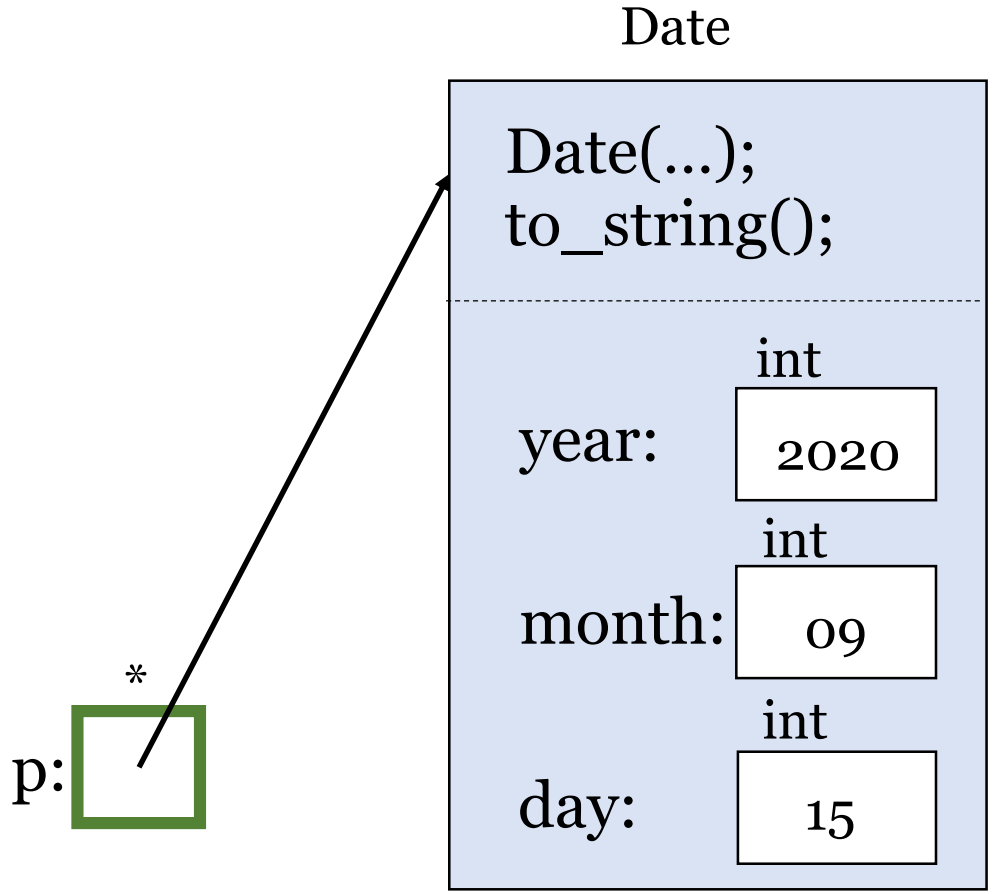


Pekare

Pekar-operatorer

- Piloperatorn steg för steg

```
p->year
```

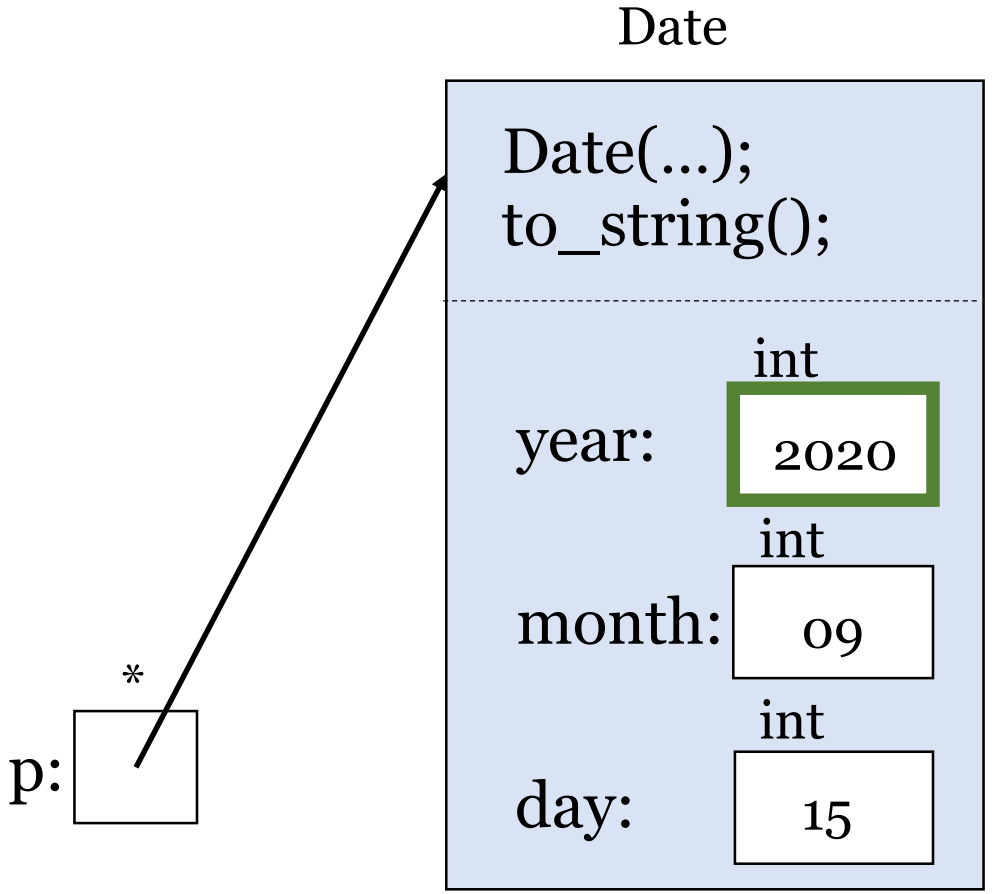


Pekare

Pekar-operatorer

- Piloperatorn steg för steg

```
p->year
```

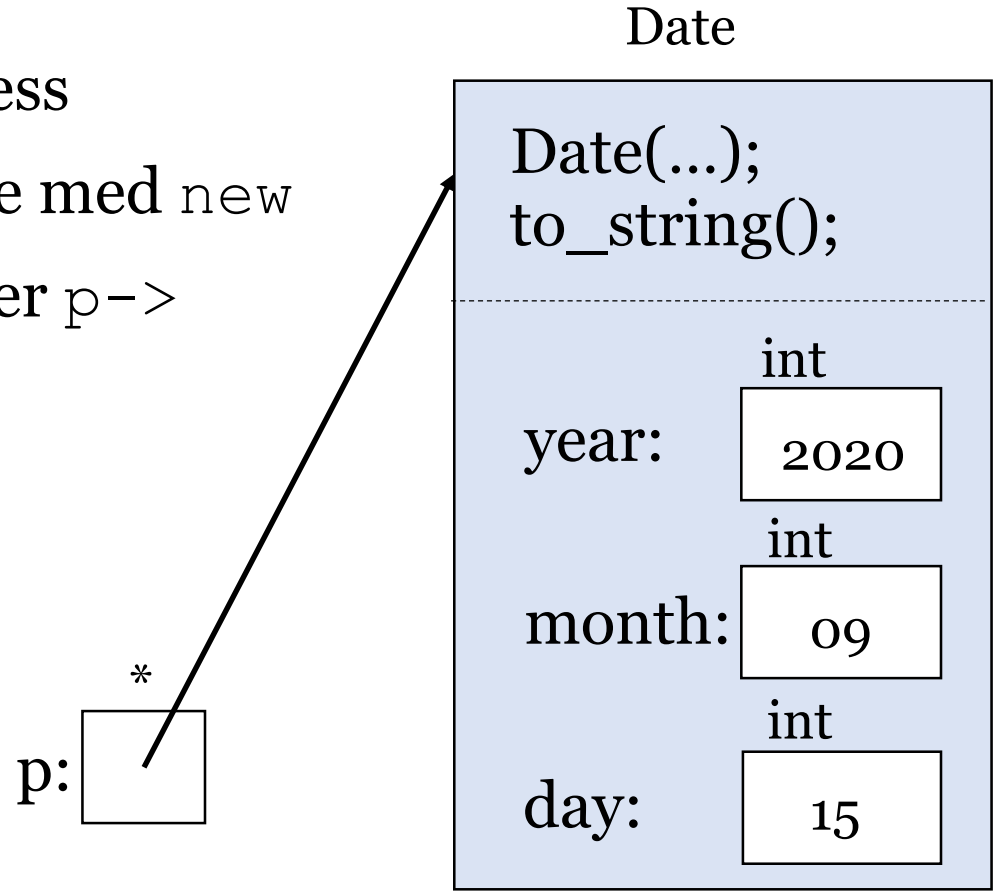


Pekare, sammanfattning

Pekar-operatorer

- *Separat* variabel som innehåller en adress
- Kan peka ut *anonyma* variabler skapade med `new`
- Avreferera och välj medlem: `(*p).` eller `p->`

```
Date * p{new Date{2020, 9, 15}};
cout << p->to_string() << endl;
cout << (*p).to_string() << endl;
```

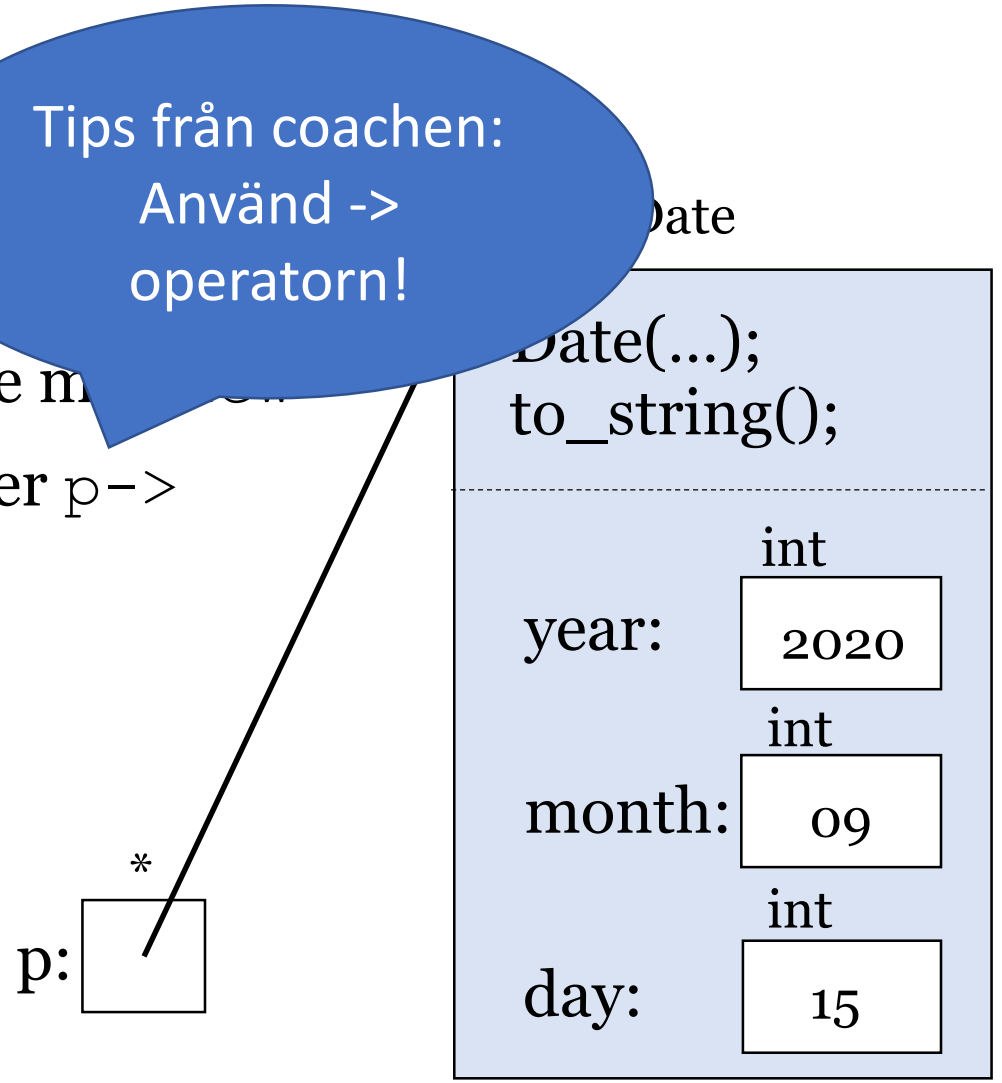


Pekare, sammanfattning

Pekar-operatorer

- *Separat* variabel som innehåller en a
- Kan peka ut *anonyma* variabler skapade m
- Avreferera och välj medlem: `(*p).` eller `p->`

```
Date * p{new Date{2020, 9, 15}};
cout << p->to_string() << endl;
cout << (*p).to_string() << endl;
```



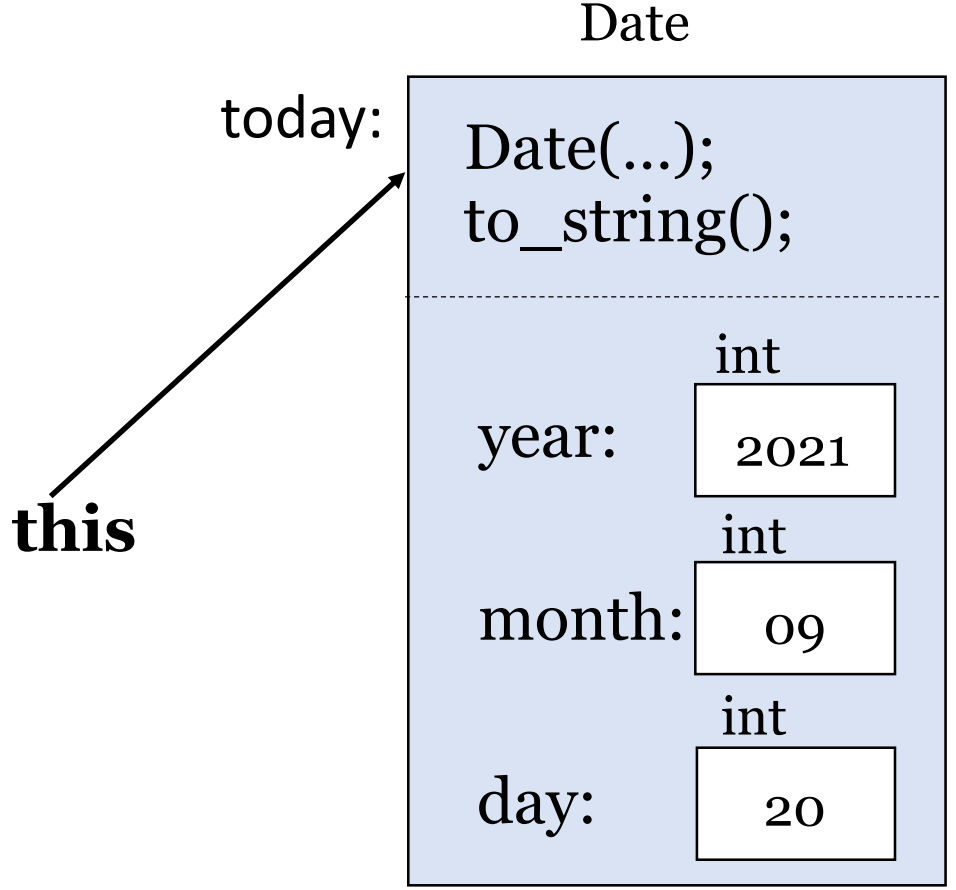
Pekare

this – en pekare till instansen(objektet) ”vi är i”

```
Date today{2021, 9, 20};
cout << today.increment();
```

```
Date& Date::increment()
{
  ++day;
  return *this;
}
```

Använd sparsamt



Agenda

1. Pekare
2. Exempel: operatorer för avreferering
3. Exempel: dynamisk datastruktur (en länkad lista)
4. Exempel: abstrakt datatyp (för lista)
5. Speciella medlemsfunktioner
6. Exempel: speciella medlemsfunktioner (för lista)

LINKÖPINGS UNIVERSITET

Dynamiska datastrukturer

Vad är en dynamisk datastruktur?

- En dynamisk datastruktur kan ändra storlek under programkörningen
  - Vi får mer data som vi vill fylla ut strukturen med, från till exempel inmatning från användaren.
- Vi har redan använt en dynamisk datastruktur, `std::string`
  - Vi behöver inte, innan programmet körs, definiera hur lång den ska vara. Ex:

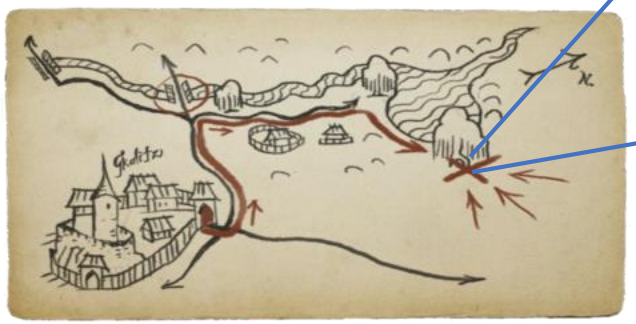
```
string name{};
cin >> name;
name.append(” is awsome!”);
```

Dynamiska datastrukturer

- Hur mycket ska den som använder en datastruktur behöva tänka på?
  => Abstrakta datatyper: vad som går att göra, inte hur det går till.
- Hur ska en viss datastruktur implementeras effektivt?
  => Beror på vad vi vill kunna göra med strukturen.

Skattjakt

En rolig aktivitet för 5-åringar (och äldre)

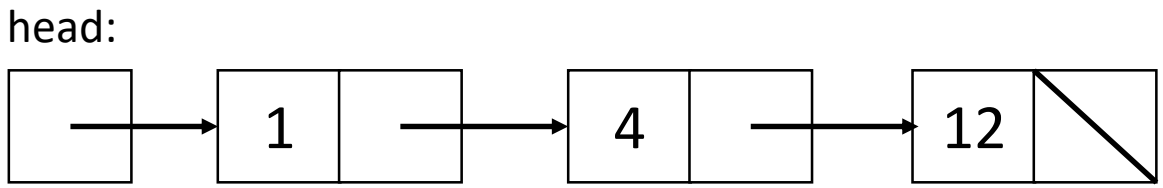


…?

Enkellänkad lista

En dynamisk datastruktur under huven

- En enkellänkad lista består av noder

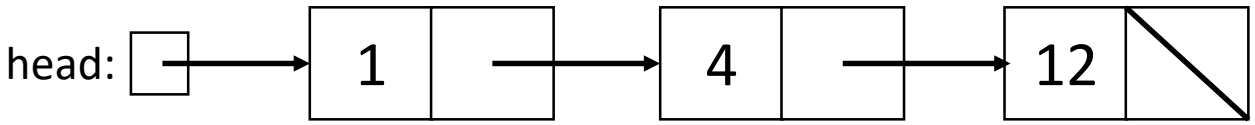


Enkellänkad lista

En dynamisk datastruktur under huven

Egenskaper:

- Det är effektivt att gå framåt från en nod till nästa
- Det går inte att gå bakåt
- Det är effektivt att skjuta in nya noder efter den nod vi tittar på

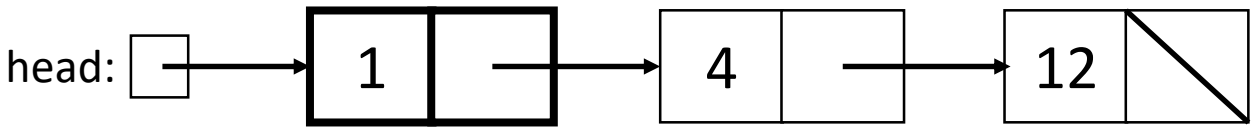


Enkellänkad lista

En dynamisk datastruktur under huven

- En **nod** är en klass som innehåller två saker
  - Ett värde av någon datatyp
  - En pekare till nästa nod i listan

```
class Node{
public:
  int data{};
  Node * next{};
};
```

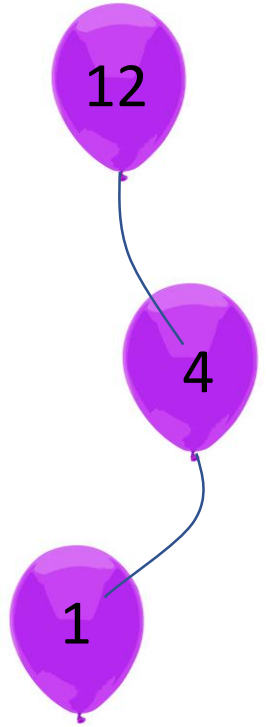


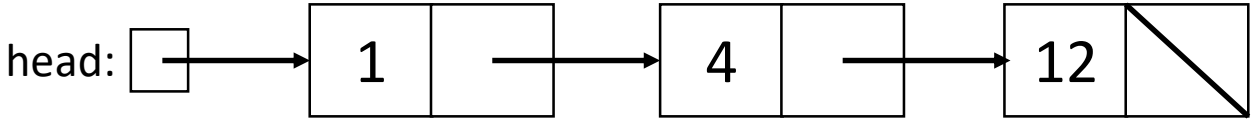
Enkellänkad lista

En dynamisk datastruktur under huven

Analogi:

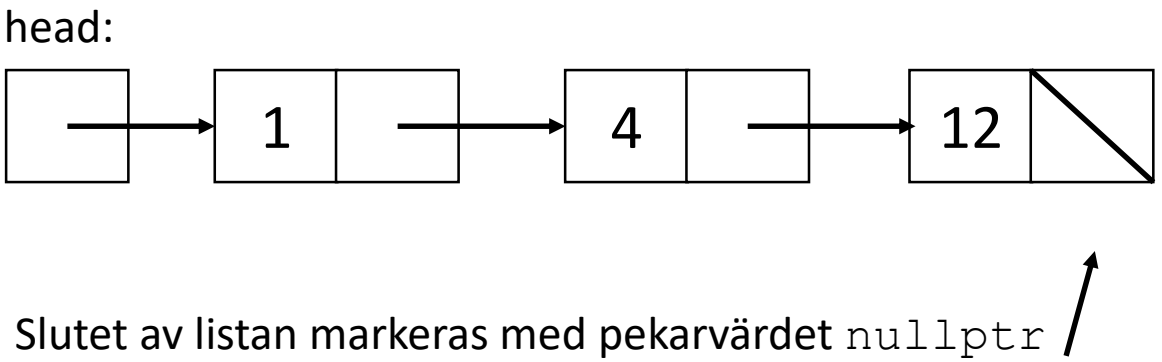
- Varje nod är en heliumballong
- På varje ballong finns skriven text
- På varje ballong finns en tejpbit som håller snöret till nästa ballong
- Vi har en tejpbit (head) för att hålla fast första ballongen





Enkellänkad lista

En dynamisk datastruktur under huven



Enkellänkad lista

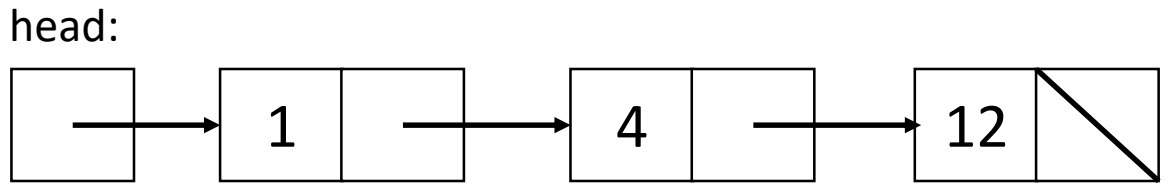
En dynamisk datastruktur under huven

- För att skapa en lista behöver vi allokera minne (new)
  - Analogi: Blåsa upp nya heliumballonger.
- Om vi allokerat minne behöver vi se till att det inte uppstår minnesläckor (att vi ”tappar bort” minne vi allokerat)
  - Analogi: Garantera att allt helium kan återanvändas.

Enkellänkad lista

En dynamisk datastruktur under huven

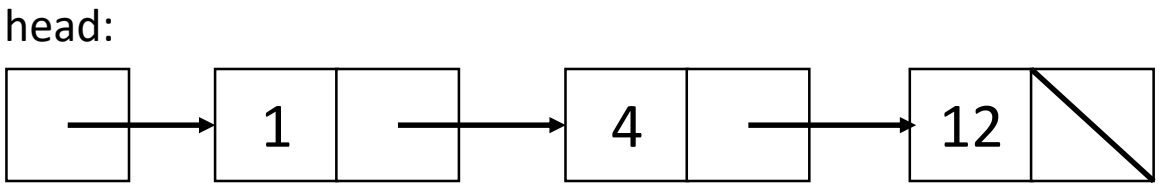
- Problem Minnesläcka #1:
- Vi vill ta bort siffran 4 från listan
  - Vi behöver ta bort noden som siffran 4 ligger i



Enkellänkad lista

En dynamisk datastruktur under huven

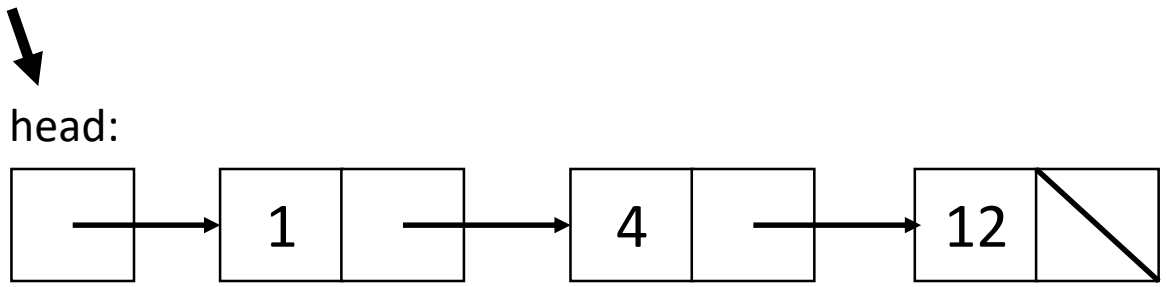
- Problem Minnesläcka #1:
- Vi vill ta bort heltalet 4 från listan
  - Hitta noden med heltalet 4 i sig.



Enkellänkad lista

En dynamisk datastruktur under huven

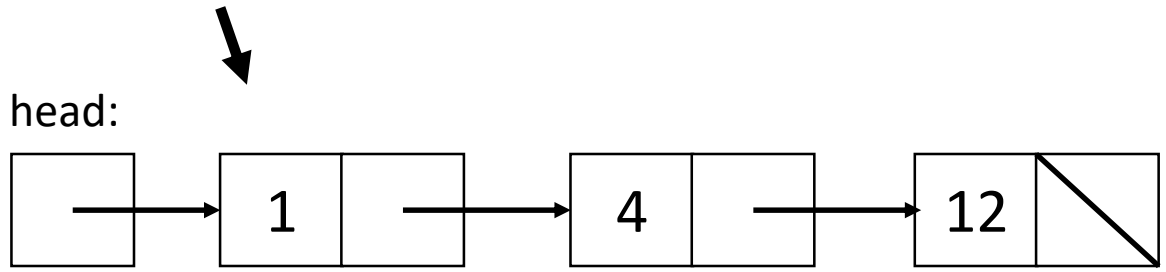
- Problem Minnesläcka #1:
- Vi vill ta bort heltalet 4 från listan
  - Hitta noden med heltalet 4 i sig.



Enkellänkad lista

En dynamisk datastruktur under huven

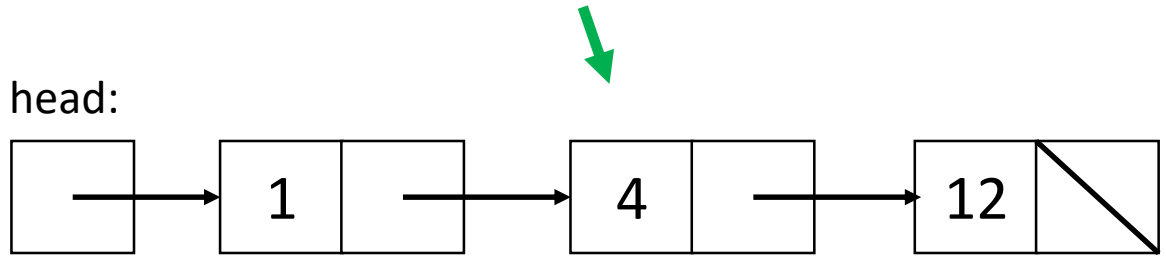
- Problem Minnesläcka #1:
- Vi vill ta bort heltalet 4 från listan
  - Hitta noden med heltalet 4 i sig.



Enkellänkad lista

En dynamisk datastruktur under huven

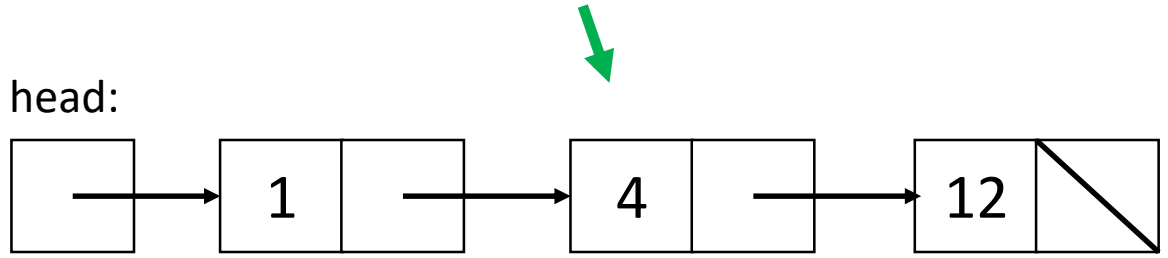
- Problem Minnesläcka #1:
- Vi vill ta bort heltalet 4 från listan
  - Hitta noden med heltalet 4 i sig.



Enkellänkad lista

En dynamisk datastruktur under huven

- Problem Minnesläcka #1:
- Vi vill ta bort heltalet 4 från listan
  - Hitta noden med heltalet 4 i sig.
  - Ta bort noden (utför delete på pekaren till noden)



Enkellänkad lista

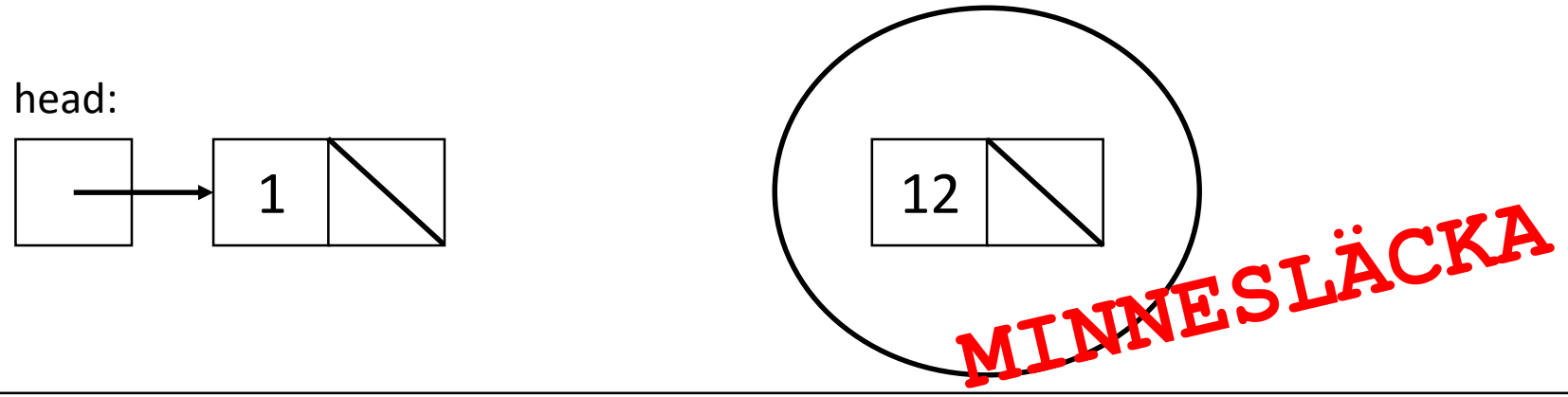
En dynamisk datastruktur under huven

- Problem Minnesläcka #1:
- Vi vill ta bort heltalet 4 från listan
  - Hitta noden med heltalet 4 i sig.
  - Ta bort noden (utför delete på pekaren till noden)
  - Ersätt pekaren med nullptr

Enkellänkad lista

En dynamisk datastruktur under huven

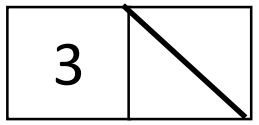
- Problem Minnesläcka #1:
- Vi vill ta bort heltalet 4 från listan
  - Hitta noden med heltalet 4 i sig.
  - Ta bort noden (utför delete på pekaren till noden)

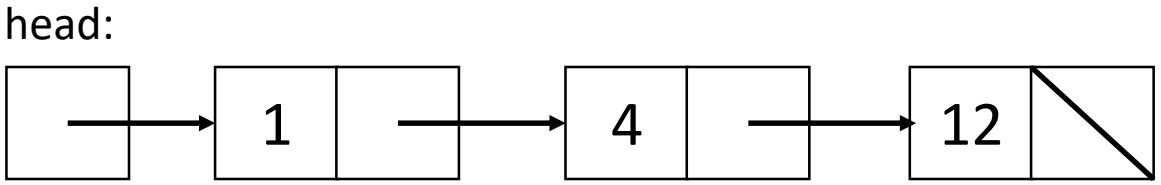


Enkellänkad lista

En dynamisk datastruktur under huven

- Problem Minnesläcka #2:
- Vi vill stoppa in heltalet 3 sorterat i listan
  - Hitta platsen där 3 där ska stoppas in

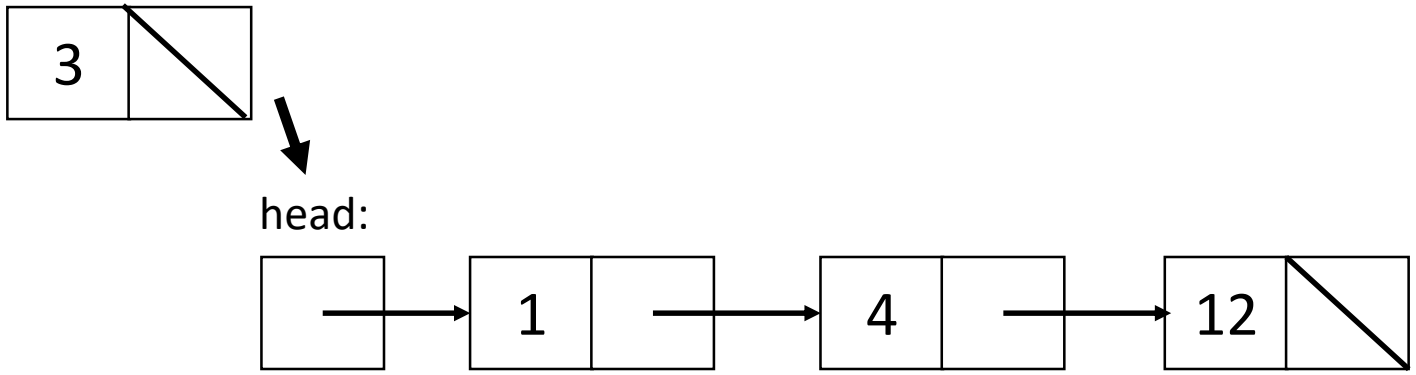




Enkellänkad lista

En dynamisk datastruktur under huven

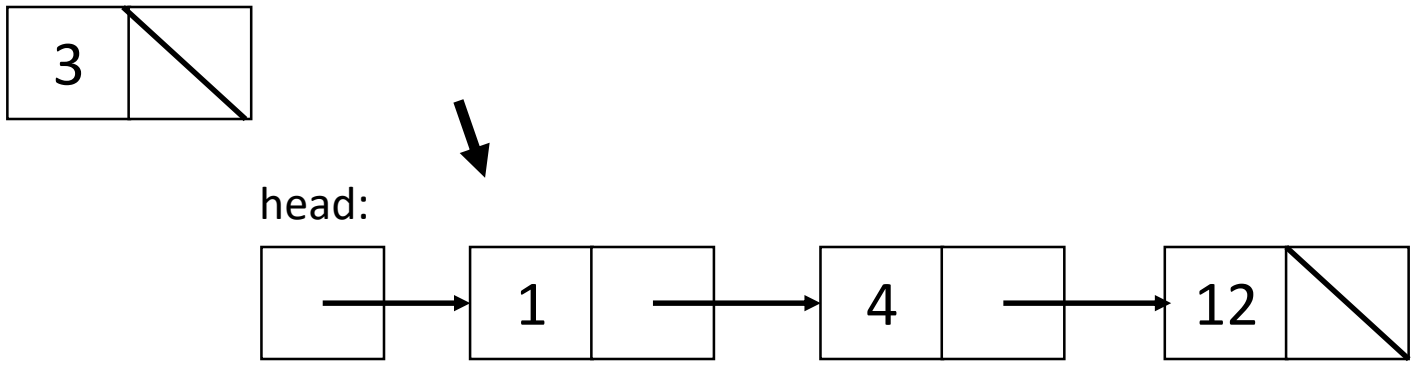
- Problem Minnesläcka #2:
- Vi vill stoppa in heltalet 3 sorterat i listan
  - Hitta platsen där 3 där ska stoppas in



Enkellänkad lista

En dynamisk datastruktur under huven

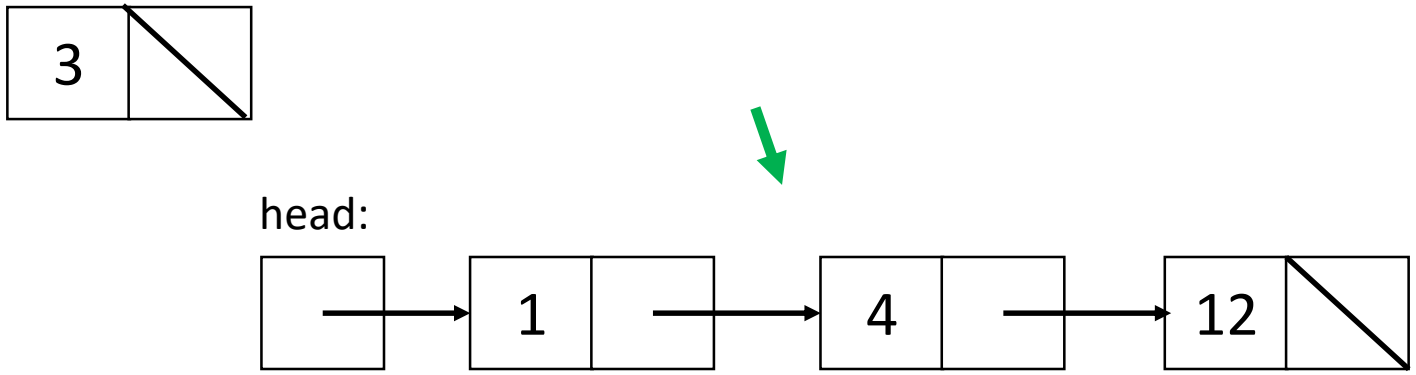
- Problem Minnesläcka #2:
- Vi vill stoppa in heltalet 3 sorterat i listan
  - Hitta platsen där 3 där ska stoppas in



Enkellänkad lista

En dynamisk datastruktur under huven

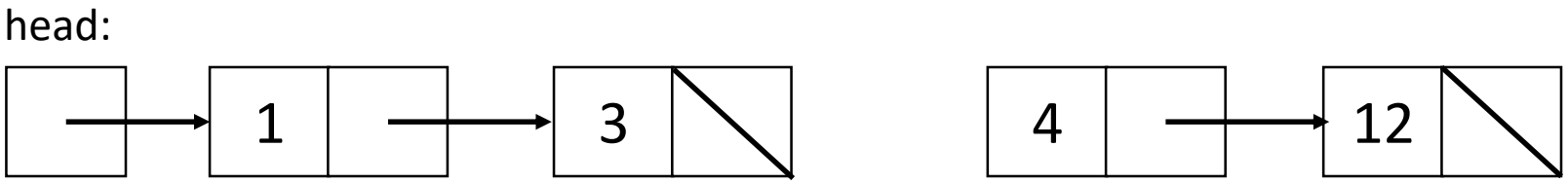
- Problem Minnesläcka #2:
- Vi vill stoppa in heltalet 3 sorterat i listan
  - Hitta platsen där 3 där ska stoppas in



Enkellänkad lista

En dynamisk datastruktur under huven

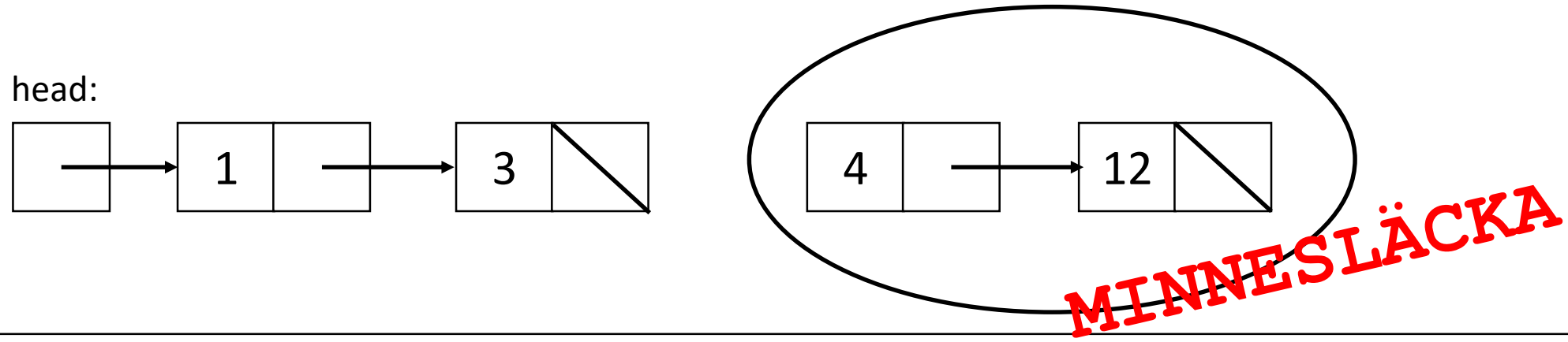
- Problem Minnesläcka #2:
- Vi vill stoppa in heltalet 3 sorterat i listan
  - Hitta platsen där 3 där ska stoppas in
  - Next-pekaren i noden med heltalet 1 ska peka på den nya noden



Enkellänkad lista

En dynamisk datastruktur under huven

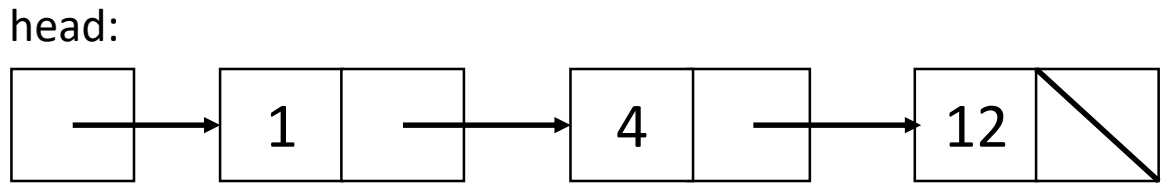
- Problem Minnesläcka #2:
- Vi vill stoppa in heltalet 3 sorterat i listan
  - Hitta platsen där 3 där ska stoppas in
  - Next-pekaren i noden med heltalet 1 ska peka på den nya noden



Enkellänkad lista

En dynamisk datastruktur under huven

- Säg att vi vill destruera (ta bort minnet av) en lista
- Vad händer om vi bara tar bort den översta nod-pekaren?



Enkellänkad lista

En dynamisk datastruktur under huven

- Säg att vi vill destruera (ta bort minnet av) en lista
- Vad händer om vi bara tar bort den översta nod-pekaren?
- delete head;

Enkellänkad lista

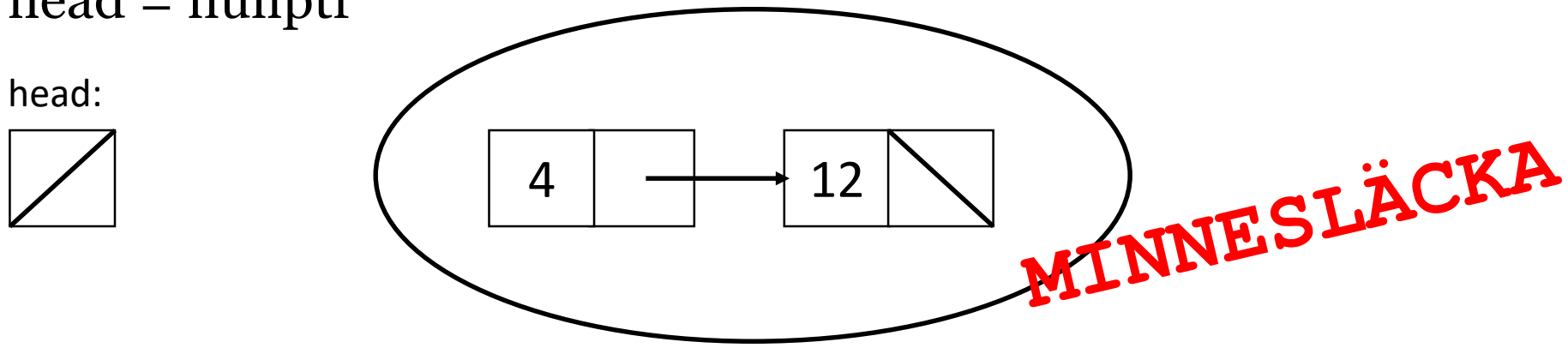
En dynamisk datastruktur under huven

- Säg att vi vill destruera (ta bort minnet av) en lista
- Vad händer om vi bara tar bort den översta nod-pekaren?
- delete head
- head = nullptr

Enkellänkad lista

En dynamisk datastruktur under huven

- Säg att vi vill destruera (ta bort minnet av) en lista
- Vad händer om vi bara tar bort den översta nod-pekaren?
- delete head
- head = nullptr



Agenda

1. Pekare
2. Exempel: operatorer för avreferering
3. Exempel: dynamisk datastruktur (en länkad lista)
4. Exempel: abstrakt datatyp (för lista)
5. Speciella medlemsfunktioner
6. Exempel: speciella medlemsfunktioner (för lista)

LINKÖPINGS UNIVERSITET

Abstrakta datatyper

Dynamiska datastrukturer som är enkla och säkra att använda

- En dynamisk datastruktur kan vara utmanande att implementera. Det är mycket att hålla reda på. Hur varje operation ska gå till, specialfall, initiering, minneshantering, minnesläckor, djup kopiering, korrekthet, effektivitet, destruering.
- För att *använda* en datastruktur ska man inte behöva hålla reda på något av ovan. Det ska räcka veta *vad* man kan använda den till.
- C++ ger oss verktygen att abstrahera datastrukturen så den som implementerar den löser alla problem och den som använder den bara kör på!

Dynamisk datastruktur => Abstrakt datatyp?

- Hur mycket ska den som använder en datastruktur behöva tänka på?
  ⇒ Abstrakta datatyper: vad som går att göra, inte hur det går till.
- Hur ska en viss datastruktur implementeras effektivt?
  ⇒ Beror på vad vi vill kunna göra med strukturen.

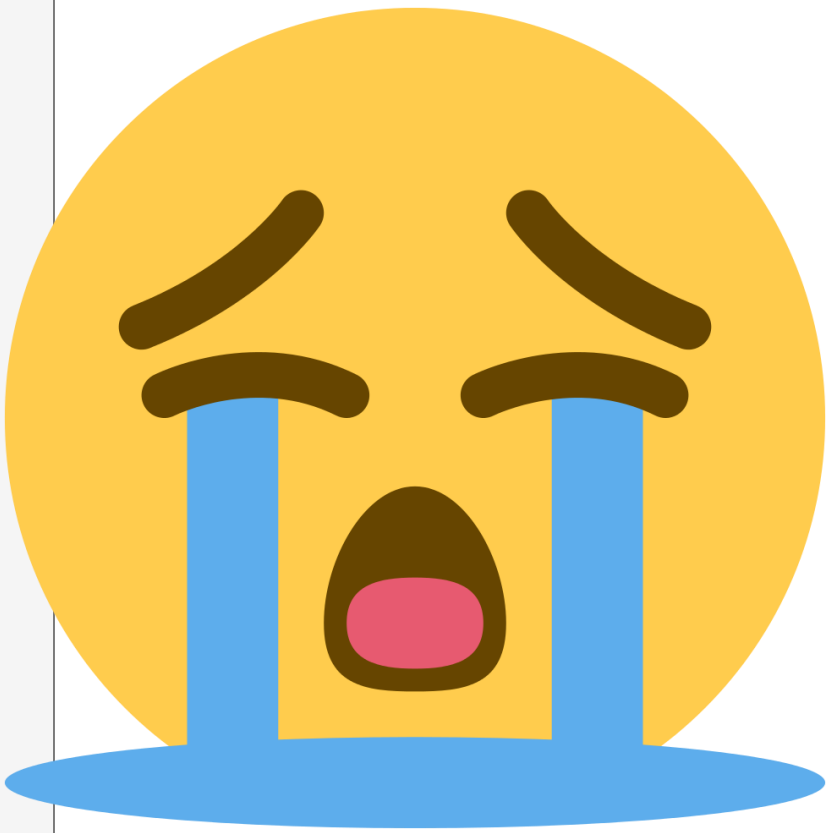
Vi har sett tre exempel på att det är väldigt noga i vilken ordning vi uppdaterar pekare för att inte tappa bort noder.

Det här är inte en datastruktur tillräckligt enkel att använda i ett program – misstag kommer att uppstå.

Enkellänkad lista – för svårt att använda korrekt!

En dynamisk datastruktur under huven

```
int main()
{
  Node* list{};

  list = new Node{12, nullptr};
  list = new Node{4, list};
  list = new Node{1, list};

  delete list->next->next;
  delete list->next;
  delete list;
  list = nullptr;
}
```



Abstrakt datatyp, enkel att använda korrekt!

En dynamisk datastruktur under huven

```
int main()
{
  List list{};

  list.insert(12);
  list.insert(4);
  list.insert(1);

  List copy{ list };

  copy.insert(3);
}
```

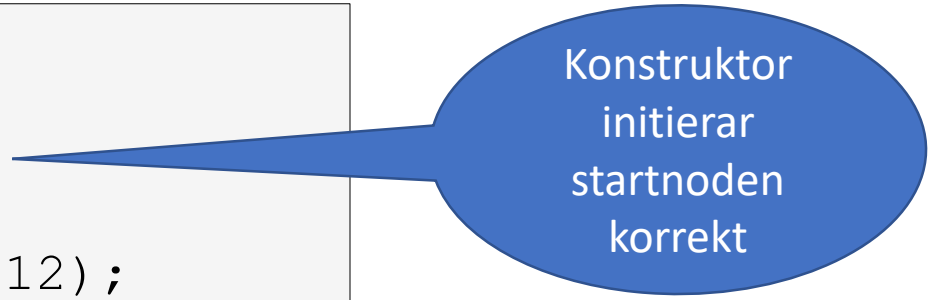
Dynamisk datastruktur => Abstrakt datatyp?

- Hur förvandlar vi vår dynamiska datastruktur till att även vara en abstrakt datatyp?
  - Dölj allt som har med noder att göra privat i en klass
  - Lägg till en konstruktor som initerar head-pekaren rätt
  - Lägg till medlemsfunktioner som gör alla besvärliga pekaroperationer
  - Lägg till alla speciella medlemsfunktioner för att alla pekare ska kopieras rätt i dessa situationer och allt minne ska avallokeras rätt i destruktorn

Abstrakt datatyp, enkel att använda korrekt!

En dynamisk datastruktur under huven

```
int main()
{
  List list{};

  list.insert(12);
  list.insert(4);
  list.insert(1);

  List copy{ list };

  copy.insert(3);
}
```



Abstrakt datatyp, enkel att använda korrekt!

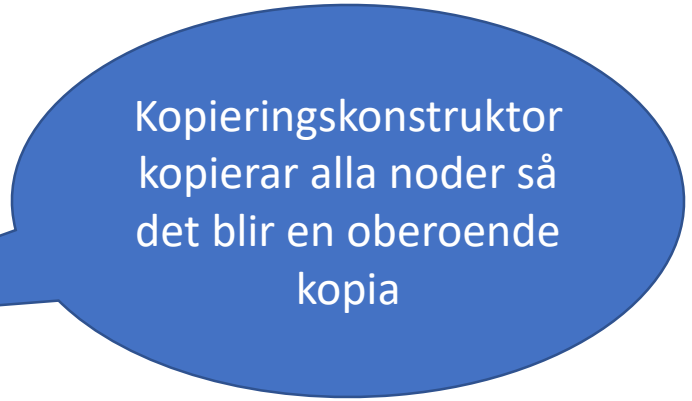
En dynamisk datastruktur under huven

```
int main()
{
  List list{};

  list.insert(12);
  list.insert(4);
  list.insert(1);

  List copy{ list };

  copy.insert(3);
}
```

Abstrakt datatyp, enkel att använda korrekt!

En dynamisk datastruktur under huven

```
int main()
{
  List list{};

  list.insert(12);
  list.insert(4);
  list.insert(1);

  List copy{ list };

  copy.insert(3);
}
```



Abstrakt datatyp, enkel att använda korrekt!

En dynamisk datastruktur under huven

```
int main()
{
  List list{};

  list.insert(12);
  list.insert(4);
  list.insert(1);

  List copy{ list };

  copy.insert(3);
}
```

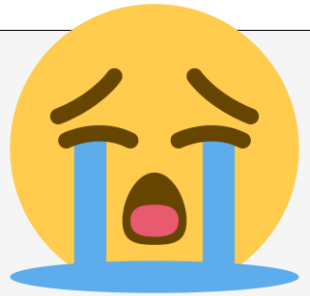
När objekten list och copy går ur scope körs destruktorn och avallokerar minne för alla noder

Inre klass

Dölj allt som har med noder att göra privat i en klass

- Det ska inte gå att från huvudprogrammet använda listans interna noder!
- Skriv listan så att dess användare tvingas att använda listans publika gränssnitt.

```
int main()
{
  Node* list{};

  list = new Node{12, nullptr};
  list = new Node{4, list};
  list = new Node{1, list};
}
```



```
int main()
{
  List list{};
  list.insert(12);
  List copy{ list };
  copy.insert(3);
}
```

Inre klass

Dölj allt som har med noder att göra privat i en klass

```
class List
{
private:
   struct Node
   {
      ...
   };
};
```

```
int main()
{
   Node* n{}; // Fel: ”Node” finns inte

   List::Node* ln{}; // Fel: ”Node” är privat
}
```

Inre klass

Dölj allt som har med noder att göra privat i en klass

```
class List
{
private:
  class Node
  {
  public:
    Node();
  };
};
```

```
List::Node::Node()
{
}
```

Inre klasser fungerar som vanligt, men vid deklarationer och definitioner måste hela ”sökvägen” till deklarationer i klassen anges!

Agenda

1. Pekare
2. Exempel: operatorer för avreferering
3. Exempel: dynamisk datastruktur (en länkad lista)
4. Exempel: abstrakt datatyp (för lista)
5. Speciella medlemsfunktioner
6. Exempel: speciella medlemsfunktioner (för lista)

LINKÖPINGS UNIVERSITET

Agenda

5 Speciella medlemsfunktioner

1. Konstruktor och destruktor
2. Kopieringskonstruktor
3. Operator för kopieringstilldelning
4. Flyttkonstruktor
5. Operator för flytt-tilldelning

LINKÖPINGS UNIVERSITET

Konstruktorn – körs när objekt skapas

En speciell medlemsfunktion

- Konstruktorn namnges som klassen.
- Deklaration:

```
Student();
```

- Defintion:

```
Student() : datamedlemsinitiering{}
{
 //Här kan det hända grejer
}
```

Konstruktorn – skapas automatiskt

En speciell medlemsfunktion

- Skapas automatiskt om du inte deklarerar någon alls.
- Motsvarande egna definition:

```
Student() = default;
```

Destruktorn – körs när objekt försvinner ur scope

En speciell medlemsfunktion

- Destruktorn namnges som klassen med tilde framför.
- Deklaration:

```
~Student();
```

- Defintion:

```
~Student()
{
 //Här kan det hända grejer
}
```

Destruktorn – skapas automatiskt

En speciell medlemsfunktion

- Skapas automatiskt om du inte deklarerar någon alls.
- Motsvarande egna definition:

```
~Student() = default;
```

Idiomet RAII i C++

Automatisk resurshanteringshjälp

- Resurser allokeras ofta i konstruktorn enligt idiomet RAII (Resource Acquisition Is Initialization) som knyter resursens livstid till objektets livstid för att kunna automatisera uppstädning mha destruktorn.
- Resurser allokeras även under objektets livstid.
- I destruktorn tas alla resurser som behöver stängas, avslutas eller avallokeras om hand på säkert sätt.
- Destruktorn anropas aldrig explicit, kompilatorn genererar automatiskt alla nödvändiga anrop.

Destruktorn – anropas automatiskt

En speciell medlemsfunktion

```
class Student
{
public:

  Student(string n, int a)
  : name{n}, age{a}
  {}

  ~Student();

private:
  string name;
  int age;

};
```

```
int main()
{
  Student * st{ new Student{”Oskar”, 7} };

  delete st; // destruktorn anropas automatiskt
}
```

```
int main()
{
  Student st{”Oskar”, 7};

  return 0; // destruktorn anropas automatiskt
}
```

Vem tar bort anonyma variabler i objekt?

Hur använder vi destruktorn?

- Om programmeraren som använder klassen måste *känna till* hur klassen fungerar och *komma ihåg* ta bort klassens minne uppstår snabbt buggar, minnesläckor och underhållssvårigheter (koden för borttagning på alla ställen klassen används).
- Klassens *destruktor* körs automatiskt när objektet tas bort för att programmeraren av klassen ska kunna ta sitt ansvar och programmeraren som använder klassen ska slippa komma ihåg anrop för uppstädning!

➔Ansvaret för borttagning måste ligga i klassens destruktor!

Agenda

5 Speciella medlemsfunktioner

1 Konstruktor och destruktor

2 Kopieringskonstruktor

3 Operator för kopieringstilldelning

4 Flyttkonstruktor

5 Operator för flytt-tilldelning

LINKÖPINGS UNIVERSITET

Kopieringskonstruktorn – för kopiering till *nytt objekt*

En speciell medlemsfunktion

- Skapas automatiskt om du inte deklarerar någon.
- Motsvarande egna definition:

```
Student(Student const& rhs) = default;
```

- Om du inte vill ha någon alls:

```
Student(Student const& rhs) = delete;
```

- Egen defintion:

```
Student(Student const& rhs) : datamedlemsinitieriingslista{}
{
 //Här kan det hända grejer
}
```

Kopieringskonstruktorn – när anropas den?

En speciell medlemsfunktion

```
class Student
{
public:

  Student(string n, int a)
  : name{n}, age{a}
  {}

  Student(Student const&);

private:
  string name;
  int age;

};
```

```
void grade(Student param) // kopiering
{
   Student copy{ param }; // kopiering
   Student copy = param ; // kopiering (!)
   …
}
```

```
int main()
{
  Student st{”Oskar”, 7};

  grade(st);
}
```

Agenda

5 Speciella medlemsfunktioner

1 Konstruktor och destruktor

2 Kopieringskonstruktor

3 Operator för kopieringstilldelning

4 Flyttkonstruktor

5 Operator för flytt-tilldelning

Operator för kopieringstilldelning av *befintligt* objekt

En speciell medlemsfunktion

- Skapas automatiskt om du inte deklarerar någon.
- Motsvarande egna definition:

```
Student& operator=(Student const& rhs) = default;
```

- Om du inte vill ha någon alls:

```
Student& operator=(Student const& rhs) = delete;
```

- Egen defintion:

```
Student& operator=(Student const& rhs)
{
 //Här kan det hända grejer
}
```

Operator för kopieringstilldelning – när anropas den?

En speciell medlemsfunktion

```
class Student
{
public:

  Student(string n, int a)
  : name{n}, age{a}
  {}

  Student& operator=(Student const&);

private:
  string name;
  int age;

};
```

```
int main()
{
  Student oskar{"Oskar", 7};
  Student klas{"Klas", 5};

  oskar = klas; // kopieringstilldelning
}
```

Agenda

5 Speciella medlemsfunktioner

1. Konstruktor och destruktor
2. Kopieringskonstruktor
3. Operator för kopieringstilldelning
4. Flyttkonstruktor
5. Operator för flytt-tilldelning

LINKÖPINGS UNIVERSITET

Flyttkonstruktorn – för *flytt* till *nytt objekt*

En speciell medlemsfunktion

- Skapas automatiskt om du inte deklarerar någon.
- Används automatiskt när rhs inte behöver bevaras.
- Motsvarande egna definition:

```
Student(Student && rhs) = default;
```

- Om du inte vill ha någon alls:

```
Student(Student && rhs) = delete;
```

- Egen defintion:

```
Student(Student && rhs) : datamedlemsinitieriingslista{}
{
 //Här kan det hända grejer
}
```

Kopieringskonstruktorn – när anropas den?

En speciell medlemsfunktion

- Anropas automatiskt när kompilatorn vill eliminera kopiering

```
class Student
{
public:

   Student(string n, int a)
   : name{n}, age{a}
   {}

   Student(Student &&);

private:
   string name;
   int age;

};
```

```
void grade(Student param) // flytt? (beror på argument)
{
    Student copy{ std::move( param ) }; // flytt
    …
}
```

std::move tvingar fram flytt, bra vid testning

```
int main()
{
   grade(Student{"Oskar", 7}); // temporärt objekt
}
```

Temporärskapade objektet (objekt utan namn) finns inte kvar efter anropet och behöver därför inte bevaras

Agenda

5 Speciella medlemsfunktioner

1 Konstruktor och destruktor

2 Kopieringskonstruktor

3 Operator för kopieringstilldelning

4 Flyttkonstruktor

5 Operator för flytt-tilldelning

Operator för *flytt* till *befintligt* objekt

En speciell medlemsfunktion

- Skapas automatiskt om du inte deklarerar någon.
- Används automatiskt när rhs inte behöver bevaras.
- Motsvarande egna definition:

```
Student& operator=(Student && rhs) = default;
```

- Om du inte vill ha någon alls:

```
Student& operator=(Student && rhs) = delete;
```

- Egen defintion:

```
Student& operator=(Student && rhs)
{
 //Här kan det hända grejer
}
```

Operator för flytt-tilldelning – när anropas den?

En speciell medlemsfunktion

- Anropas automatiskt när kompilatorn vill eliminera kopiering

```
class Student
{
public:

  Student(string n, int a)
  : name{n}, age{a}
  {}

  Student& operator=(Student &&);

private:
  string name;
  int age;

};
```

```
int main()
{
  Student oskar{"Oskar", 7};
  Student klas{"Klas", 5};

  oskar = std::move(klas); // flytt

  klas = (klas + knowledge); // flytt
}
```

std::move tvingar fram flytt, bra vid testning

Sammansatta objektet finns inte kvar efter anropet och behöver därför inte bevaras

Agenda

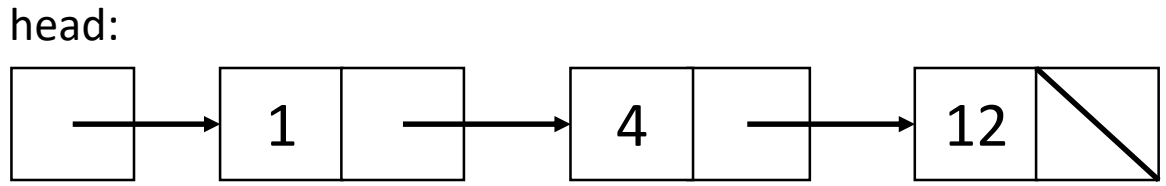
1. Pekare
2. Exempel: operatorer för avreferering
3. Exempel: dynamisk datastruktur (en länkad lista)
4. Exempel: abstrakt datatyp (för lista)
5. Speciella medlemsfunktioner
6. Exempel: speciella medlemsfunktioner (för lista)

LINKÖPINGS UNIVERSITET

Destruktorn

En speciell medlemsfunktion

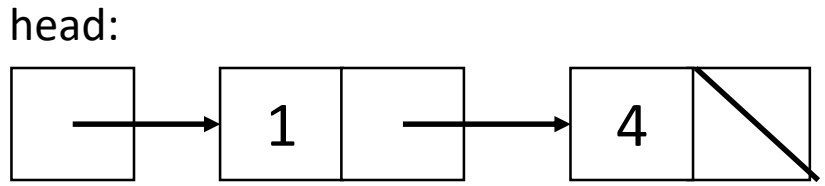
- Säg att vi vill destruera (ta bort minnet av) en lista
- Vi måste ta bort listan i rätt ordning – börja att ta bort längst bak och arbeta oss mot head



Destruktorn

En speciell medlemsfunktion

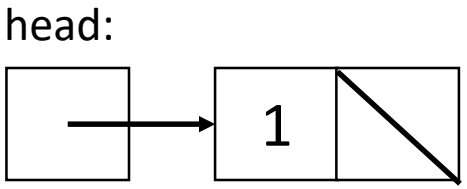
- Säg att vi vill destruera (ta bort minnet av) en lista
- Vi måste ta bort listan i rätt ordning – börja att ta bort längst bak och arbeta oss mot head



Destruktorn

En speciell medlemsfunktion

- Säg att vi vill destruera (ta bort minnet av) en lista
- Vi måste ta bort listan i rätt ordning – börja att ta bort längst bak och arbeta oss mot head

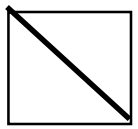


Destruktorn

En speciell medlemsfunktion

- Säg att vi vill destruera (ta bort minnet av) en lista
- Vi måste ta bort listan i rätt ordning – börja att ta bort längst bak och arbeta oss mot head

head:



Destruktorn

En speciell medlemsfunktion

- Sanning om destruktorn:
  - Vi behöver bara implementera den om vi har en resurs som kräver speciell hantering för att återlämna minne som allokerats
  - I andra fall kan kompilatorn själv generera en destruktor som gör jobbet åt oss

Kopieringskonstruktorn

En speciell medlemsfunktion

- Kompilatorn genererar rudimentärt stöd för kopiering av objekt utan att vi gör något speciellt

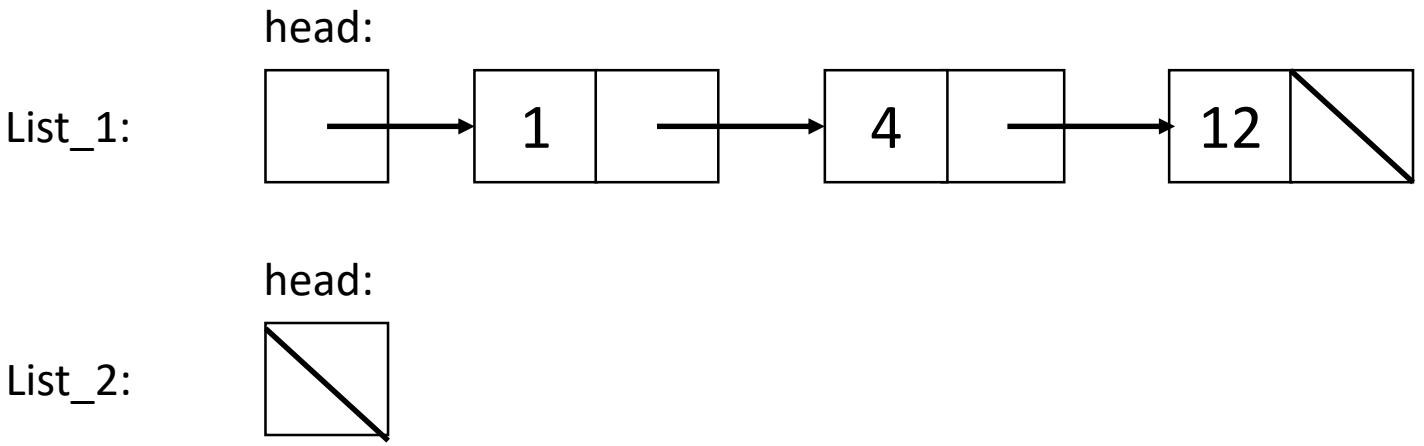
```
int main()
{
  List list_1 {1, 4, 12};
  List list_2 {list_1};
}
```

- Här används en kopieringskonstruktor
- Den automatgenererade varianten kommer kopiera värdet av varje datamedlem. Detta ger problem med pekare…

Kopieringskonstruktorn

En speciell medlemsfunktion

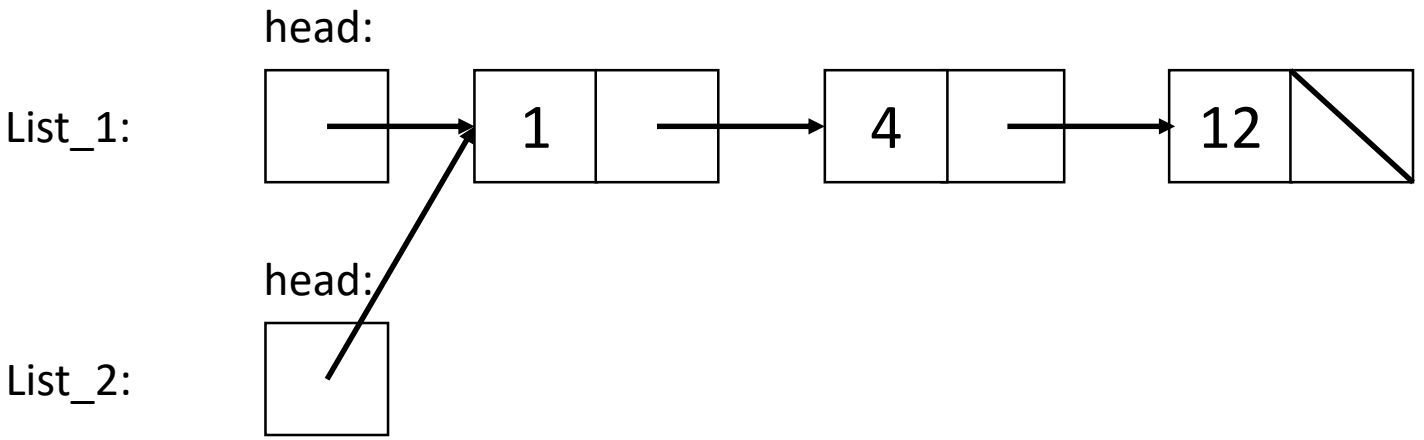
- I list har vi en datamedlem head, som är en nod-pekare



Kopieringskonstruktorn

En speciell medlemsfunktion

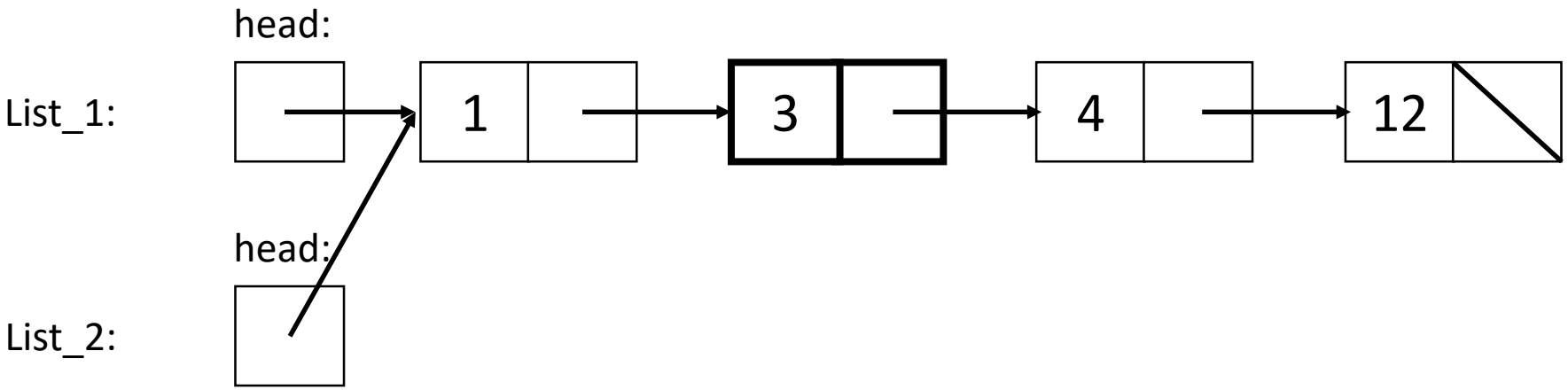
- Den automatgenererade kopieringskonstruktorn kommer skapa en *ytlig* kopia



Kopieringskonstruktorn

En speciell medlemsfunktion

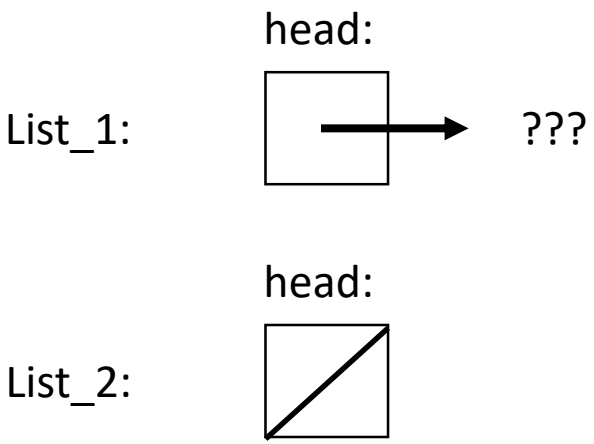
- Problem 1:
  Ändringar i List_1 dyker upp i den andra – är det verkligen en kopia då?



Kopieringskonstruktorn

En speciell medlemsfunktion

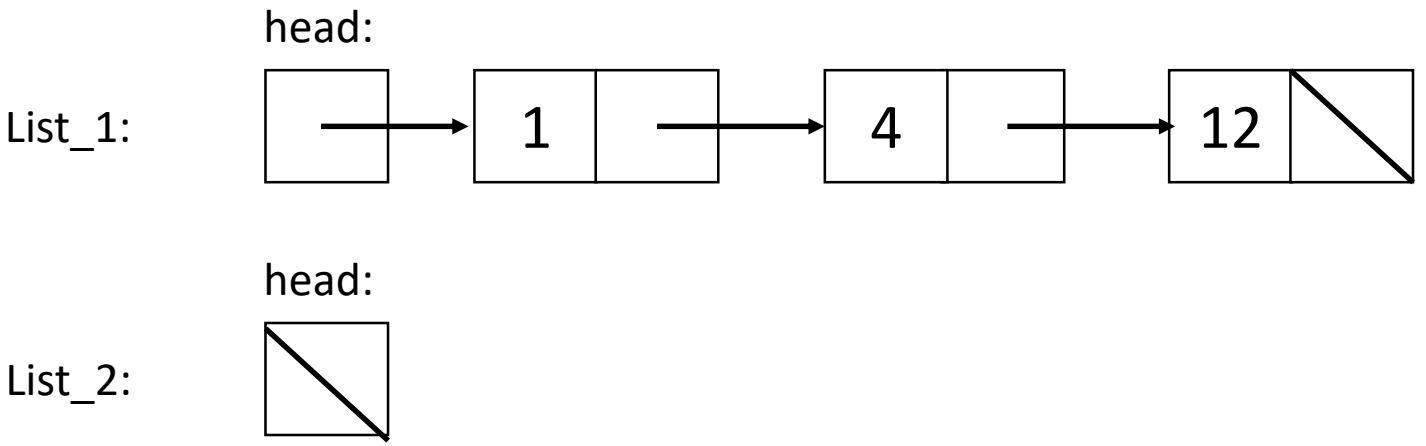
- Problem 2:
  Hur fungerar List_1 när destruktorn för List_2 körts?



Kopieringskonstruktorn

En speciell medlemsfunktion

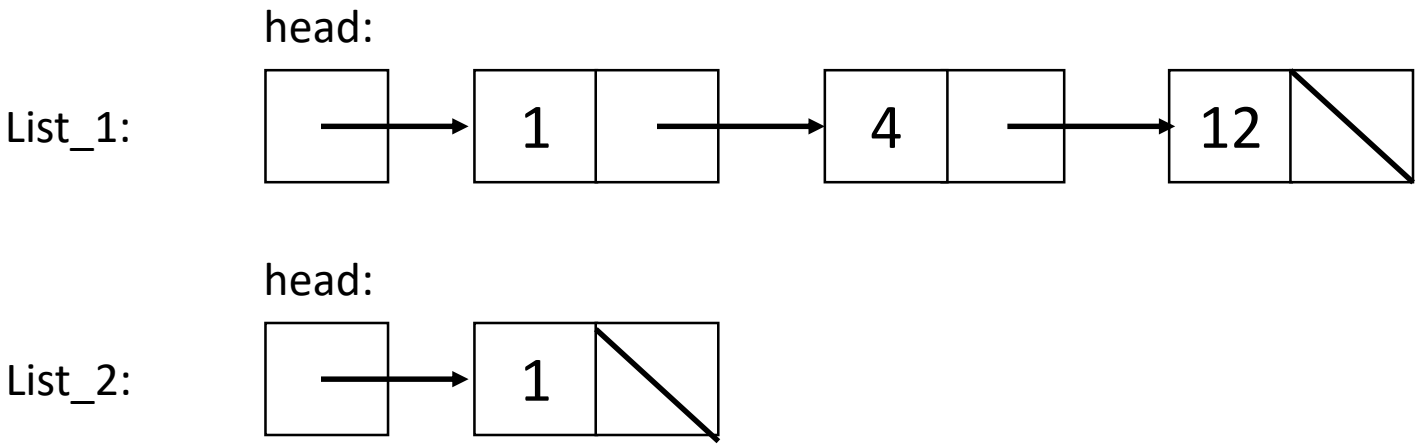
- Vad vill vi ska hända? Vi vill skapa en *djup kopia*
- En djup kopia innebär att vi kopierar varje nod i List_1 till List_2, dvs bygger upp en hel lista igen



Kopieringskonstruktorn

En speciell medlemsfunktion

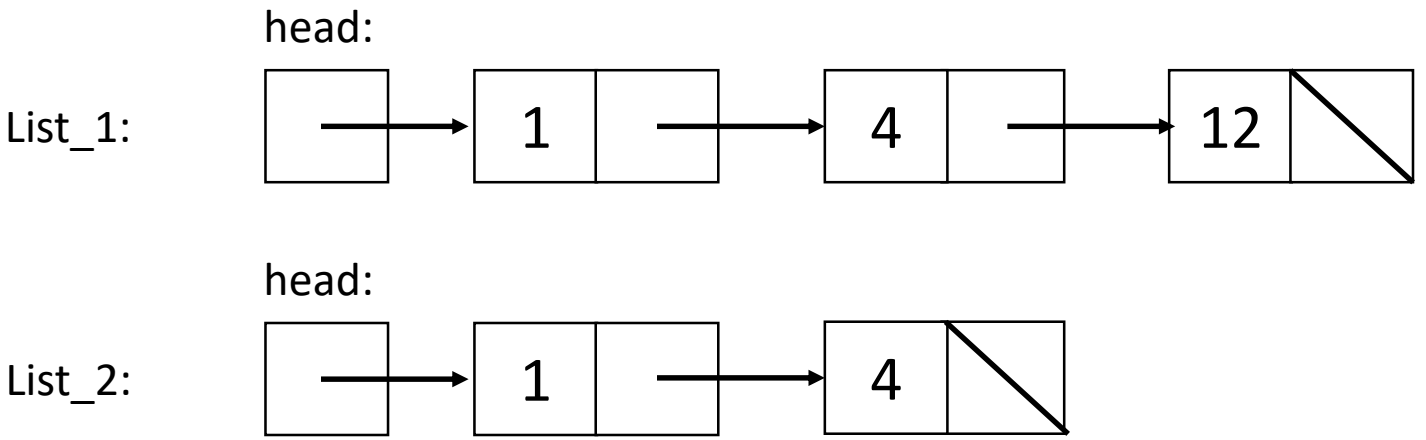
- Vad vill vi ska hända? Vi vill skapa en *djup kopia*
- En djup kopia innebär att vi kopierar varje nod i List_1 till List_2, dvs bygger upp en hel lista igen



Kopieringskonstruktorn

En speciell medlemsfunktion

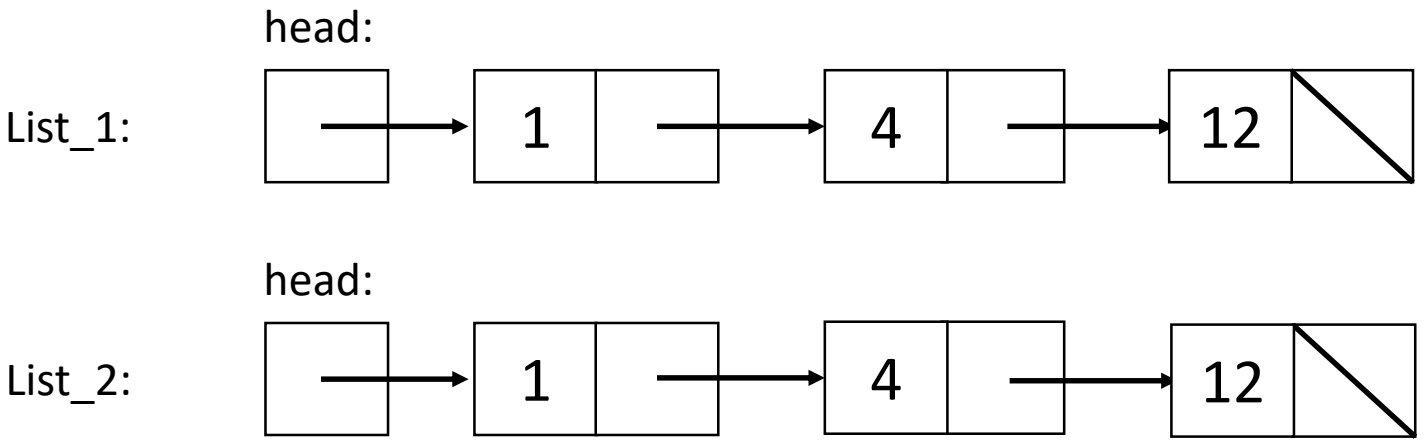
- Vad vill vi ska hända? Vi vill skapa en *djup kopia*
- En djup kopia innebär att vi kopierar varje nod i List_1 till List_2, dvs bygger upp en hel lista igen



Kopieringskonstruktorn

En speciell medlemsfunktion

- Vad vill vi ska hända? Vi vill skapa en *djup kopia*
- En djup kopia innebär att vi kopierar varje nod i List_1 till List_2, dvs bygger upp en hel lista igen



Operatorn för kopieringstilldelning

En speciell medlemsfunktion

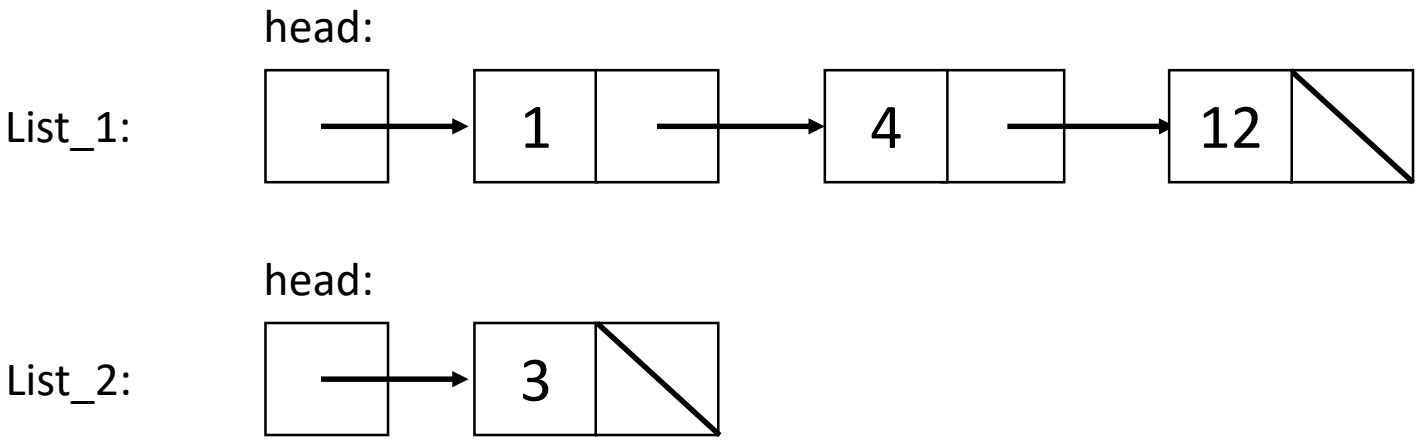
- Problemet med ytlig kopiering uppstår även vid tilldelning
- Här används kopieringstilldelning
- Den enda skillnaden är att vid tilldelning är det ett gammalt objekt som skrivs över

```
int main()
{
  List list_1 {1, 4, 12};

  List list_2 {3};

  list_1 = list_2;

  return 0;
}
```

Operatorn för kopieringstilldelning

En speciell medlemsfunktion

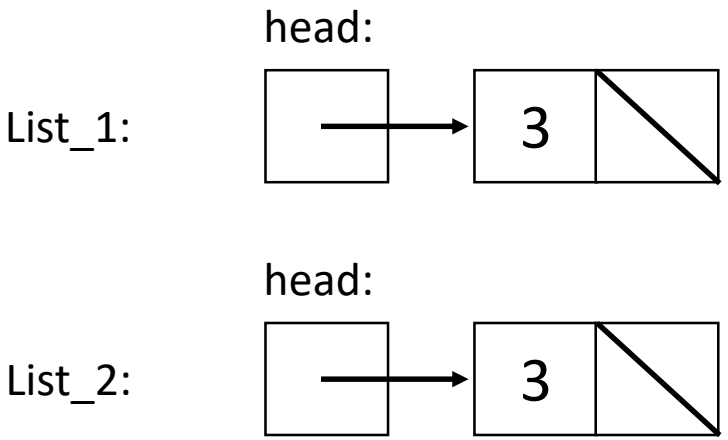
- Före ”list_1 = list_2;”



Operatorn för kopieringstilldelning

En speciell medlemsfunktion

- Efter ”list_1 = list_2;”



Kopieringskonstruktorn och tilldelningsoperatorn

Speciella medlemsfunktioner

- För att djup kopiering ska ske behöver vi själva implementera det
- Vi behöver skapa två saker:
  - kopieringkonstruktor

```
List::List(List const & rhs)
: //Här kan vi kopiera rhs
{
  //eller här
}
```

  - kopieringtilldelningsoperator

```
List & List::operator=(List const & rhs)
{
  //Här kan vi kopiera rhs och ta bort *this
  return *this;
}
```

Kopieringskonstruktorn och tilldelningsoperatorn

Speciella medlemsfunktioner

- För att djup kopiering ska ske behöver vi själva implementera det
- Vi behöver skapa två saker:
  - kopieringkonstruktor (initiering av *tomt* objekt)

```
List::List(List const & rhs)
: //Här kan vi kopiera rhs
{
  //eller här
}
```

  - rhs är den som kopieras från (skickas in i konstruktorn)

  - kopieringtilldelningsoperator (överskrivning av *existerande* objekt)

```
List & List::operator=(List const & rhs)
{
  //Här kan vi kopiera rhs och ta bort *this
  return *this;
}
```

  - rhs är den som kopieras från (till höger om tilldelningen)

Kopieringskonstruktorn och tilldelningsoperatorn

Speciella medlemsfunktioner

- För att djup kopiering ska ske behöver vi själva implementera det
- Vi behöver skapa två saker:
  - kopieringkonstruktor

```
List::List(List const & rhs)
: //Här kan vi kopiera rhs
{
  //eller här
}
```

  - kopieringtilldelningsoperator

```
List & List::operator=(List const & rhs)
{
  //Här kan vi kopiera rhs och ta bort *this
  return *this;
}
```

```
int main()
{
  List list_1 {1, 4, 12};

  List list_2 {list_1};

  list_1 = list_2;

  return 0;
}
```

Kopieringskonstruktorn och tilldelningsoperatorn

Speciella medlemsfunktioner

- För att djup kopiering ska ske behöver vi själva implementera det
- Vi behöver skapa två saker:
  - kopieringkonstruktor

```
List::List(List const & rhs)
: //Här kan vi kopiera rhs
{
  //eller här
}
```

  - kopieringtilldelningsoperator

```
List & List::operator=(List const & rhs)
{
  //Här kan vi kopiera rhs och ta bort *this
  return *this;
}
```

```
int main()
{
  List list_1 {1, 4, 12};

  List list_2 {list_1};

  list_1 = list_2;

  return 0;
}
```

Kopieringskonstruktorn och tilldelningsoperatorn

Speciella medlemsfunktioner

- För att djup kopiering ska ske behöver vi själva implementera det
- Vi behöver skapa två saker:
  - kopieringkonstruktor

```
List::List(List const & rhs)
: //Här kan vi kopiera rhs
{
  //eller här
}
```

  - kopieringtilldelningsoperator

```
List & List::operator=(List const & rhs)
{
  //Här kan vi kopiera rhs och ta bort *this
  return *this;
}
```

```
int main()
{
  List list_1 {1, 4, 12};

  List list_2 {list_1};

  list_1 = list_2;

  return 0;
}
```

Operatorn för tilldelning

Speciella medlemsfunktioner

- För tilldelningsoperatorn finns ett användbart mönster att använda

```
List& List::operator=(List const& rhs)
{
  // Kopiera rhs med kopieringskonstruktor

  // byt plats på kopians och this-objektets
  // headpekare (och ev andra datamedlemmar)

  // Njut när destruktorn tar hand om kopian
  // som nu har this-objektets gamla lista
  return *this;
}
```

```
int main()
{
  List list_1 {1, 4, 12};

  List list_2 {list_1};

  list_1 = list_2;

  return 0;
}
```

Operatorn för tilldelning

Speciella medlemsfunktioner

- Används inte mönstret måste vi se upp för självtilldelning

```
List& List::operator=(List const& rhs)
{
  if ( this == &rhs )
  {
    // avallokera gamla listan i detta objekt
    // och gör djup kopia av rhs
  }
  return *this;
}
```

```
int main()
{
  List list_1 {1, 4, 12};

  List list_2 {list_1};

  list_1 = list_1;

  return 0;
}
```

Flyttsemantik

Speciella medlemsfunktioner

- Kompilatorn kan hitta många ställen där objekt behöver kopieras
- Detta kan ta mycket kraft och känns onödigt ibland
- Därför infördes flyttsemantik i C++11!
- Målet är att kompilatorn kan se när ett temporärt objekt ska kopieras och istället flytta innehållet
  - Temporärt innebär att objektet kommer dö snart…

Flyttsemantik

Speciella medlemsfunktioner

- Antag att jag har ett hus.
- Om jag vill ge dig ett likadant hus som mitt gör C++ så här:
  - Djup kopiering: Går igenom del för del av mitt hus, och bygger upp en kopia hos dig.
- Antag nu att jag med säkerhet inte behöver mitt hus mer, då fortsätter C++ med:
  - Destruering: Alla delar av mitt hus rivs och återbrukas.

Flyttsemantik

Speciella medlemsfunktioner

- Om vi från början VET att jag inte behöver mitt hus mer är det mycket enklare att du direkt adressändrar till mitt hus. Vi slipper hela byggprocessen och rivningsprocessen!

- Adressändring (***flytt***) är mycket effektivare även i C++!

Flyttkonstruktor och flytt-tilldelning

Speciella medlemsfunktioner

- För att flytt ska ske behöver vi själva implementera det med hjälp av:
  - flyttkonstruktor (flytta till nytt tomt objekt)

```
List::List(List && rhs)
: //Här kan rhs flytta in
{
  //eller här
}
```

```
#include <utility>
int main()
{
  List list_1 {1, 4, 12};
  List list_2 {move(list_1)};
  return 0;
}
```

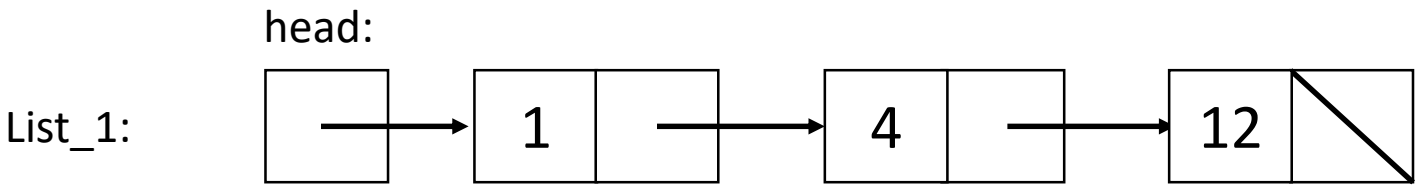
  - flytttilldelningsoperator (byt plats med befintligt objekt)

```
List & List::operator=(List && rhs)
{
  //Här kan *this flytta ut och rhs flytta in
  return *this;
}
```

Flyttkonstruktorn

En speciell medlemsfunktion

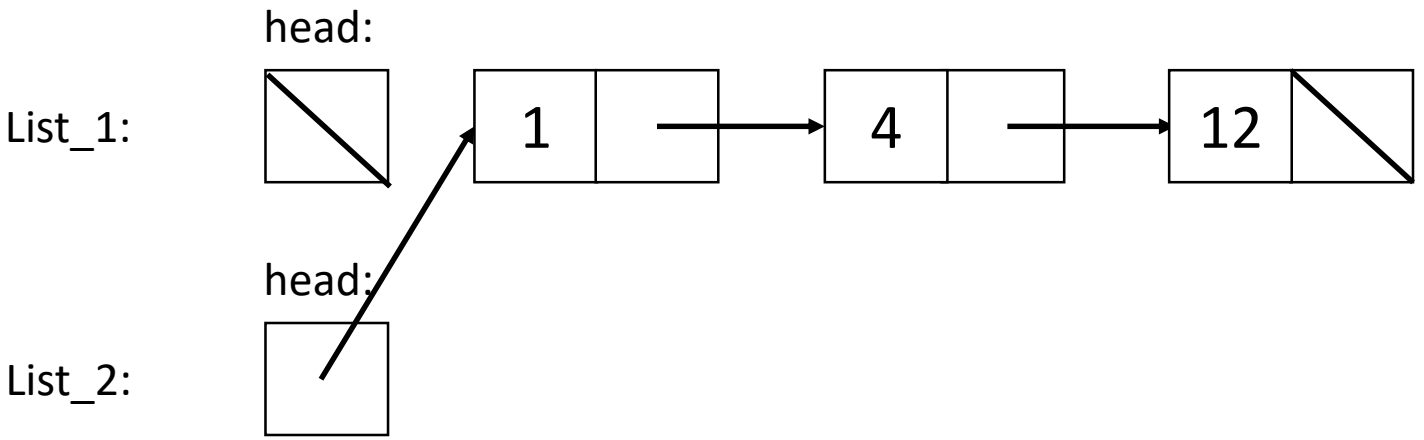
- Före ”List list_2{ std::move(list_1) };”



Flyttkonstruktorn

En speciell medlemsfunktion

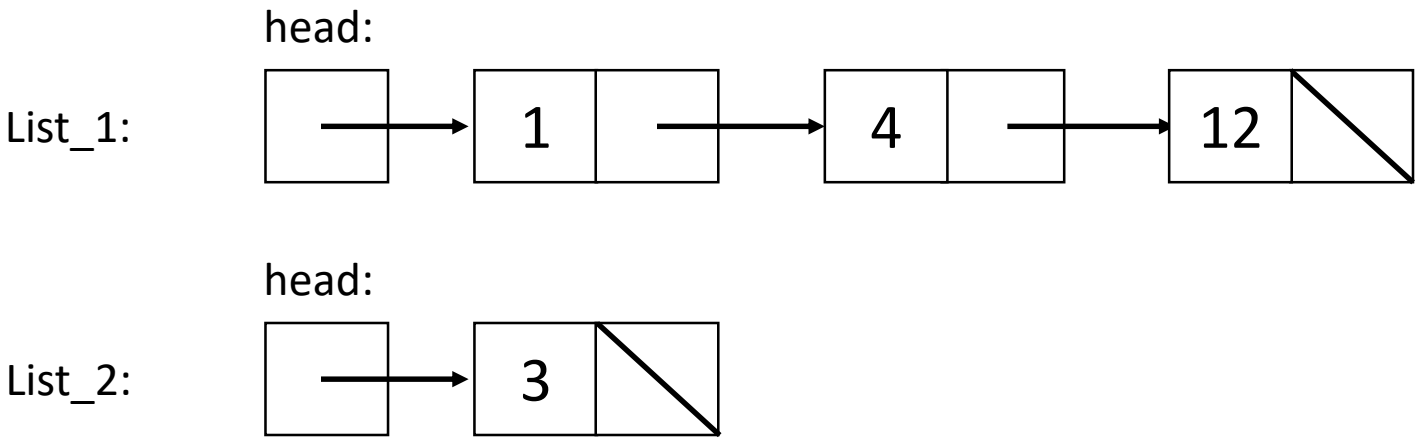
- Före ”List list_2{ std::move(list_1) };”



Operatorn för flytt-tilldelning

En speciell medlemsfunktion

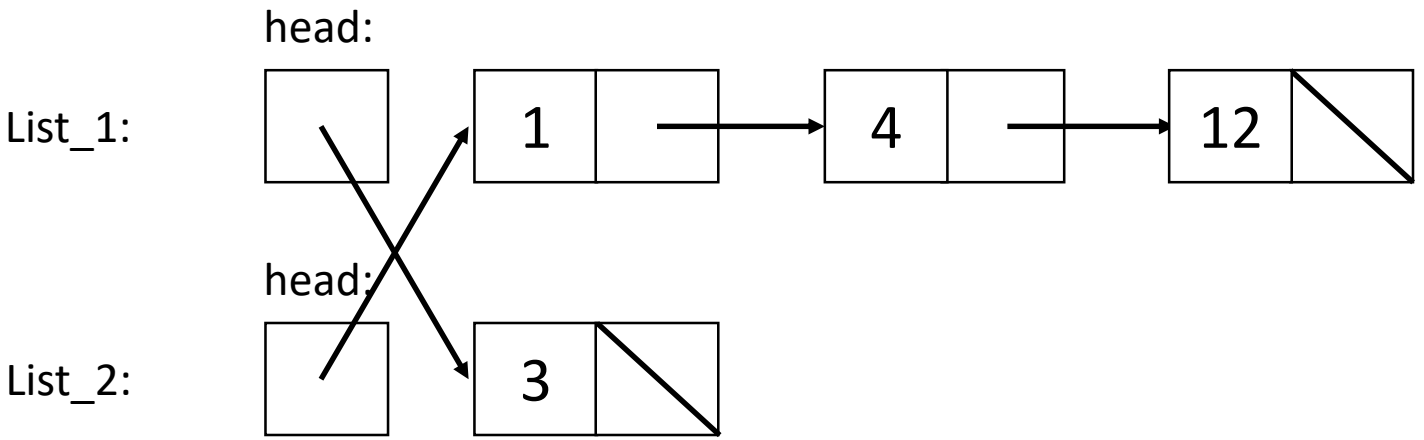
- Före ”list_1 = std::move(list_2);”



Operatorn för flytt-tilldelning

En speciell medlemsfunktion

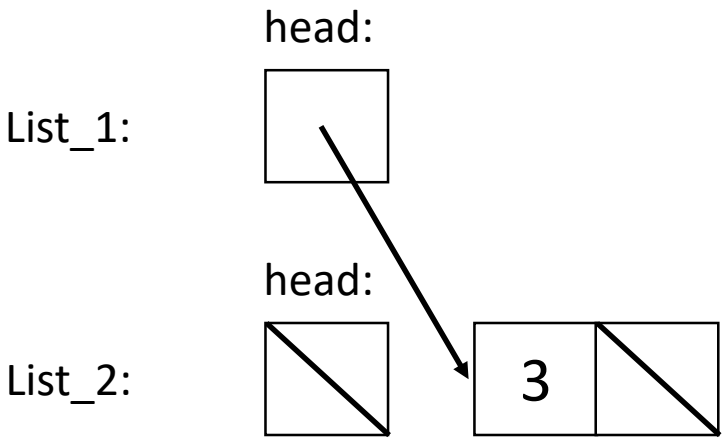
- Efter ”list_1 = std::move(list_2);”



Operatorn för flytt-tilldelning

En speciell medlemsfunktion

- Efter ”list_1 = std::move(list_2);” (alternativ)



Flyttkonstruktor och flytt-tilldelning

Speciella medlemsfunktioner

- För att testa flytt kan vi använda oss av funktionen `std::move` från `<utility>`
- Normalt optimerar en modern kompilator bort både många kopieringar och flytter. Vill vi se hur det ser ut utan optimering kan vi använda kompileringsflaggan

  `-fno-elide-constructors`

```
int main()
{
  List list_1 {1, 4, 12};
  List list_2 {move(list_1)};
  return 0;
}
```

www.liu.se

LINKÖPINGS UNIVERSITET