TDIU20

Pekare och dynamiskt minne

Eric Ekström

Institutionen för datavetenskap

LINKÖPING UNIVERSITY

Problem - Godtyckligt mycket minne

Kan vi skriva

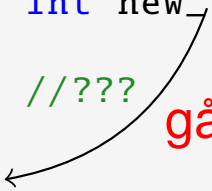
- en loop som varje iteration skapar en ny variabel? Efter N iterationer ska det finnas N variabler att använda.
- en funktion som lägger till ett värde i en (kanske full) behållare? Funktionen ska garantera att värdet läggs till.

Vi får bara använda vanliga variabler och språkkonstruktioner. (Inga komponenter från standardbibliotek)

Problem - Godtyckligt mycket minne

Försök:

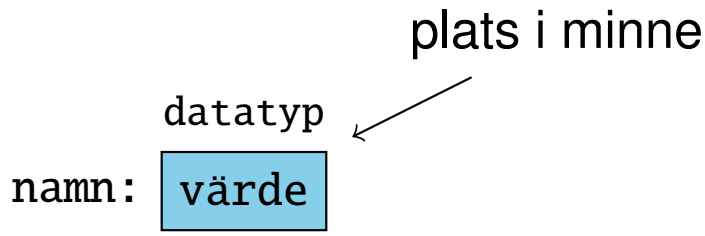
```
for (int i {}; i < N; ++i)
{

  int new_var {};

  //???

}

// Inga variabler kvar
```



1 Repetition
2 Pekare
3 Dynamiskt minne
4 Länkade strukturer
5 Objektorienterade tankar
6 Speciella medlemsfunktioner
7 Övrigt
8 Livekod

Repetition: Variabler

- Automatiska namngivna variabler
- En låda för att lagra ett värde
  - Namn
  - Värde
  - Datatyp
  - Plats i minne



- Kompilatorn sköter allt åt oss (allokera, avallokera, tolka minnet)
- Sparas på en exekveringsstack (“stacken”).

TODO

Det kändes tomt här. Ville ha mer att prata om.

1 Repetition
2 Pekare
3 Dynamiskt minne
4 Länkade strukturer
5 Objektorienterade tankar
6 Speciella medlemsfunktioner
7 Övrigt
8 Livekod

Pekare

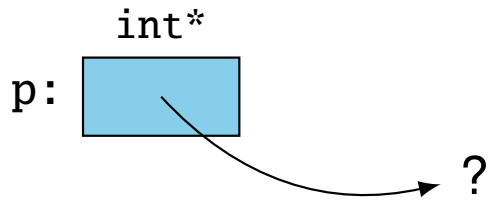
En pekare är en variabel som innehåller en address.

- Dvs en variabel som pekare ut en annan plats i minne.

Deklareras med en * och vilken datatyp den pekar på.

- “p är en pekare till ett heltal.”

```
int* p {?};
```



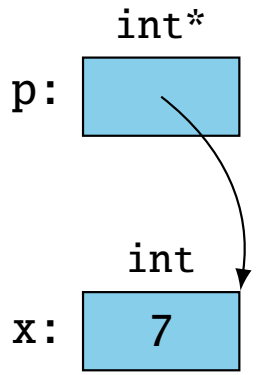
Pekare - Adressoperatorn

Vi kan hitta adressen av en variabel med `operator&`.

- “Adressoperatorn”
- Inte samma som referens! Samma tecken till två olika koncept.

```
int x {7};
int* p {&x};
cout << &x << " " << p << endl;
```

```
$ ./a.out
0x7ffe4 0x7ffe4
```



(Används sällan. Undvik i labb!)

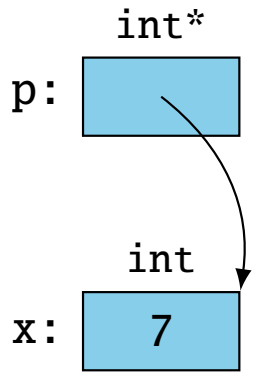
Pekare - Avreferering

Vi kan hitta värdet på adressen av en pekare med `operator*`.

- Avrefereringsoperatorn
- “följ pilen till värdet”
- Inte samma som multiplikation eller pekardeklaration!

```
int x {7};
int* p {&x};
cout << x << " " << *p << endl;
```

```
$ ./a.out
7 7
```



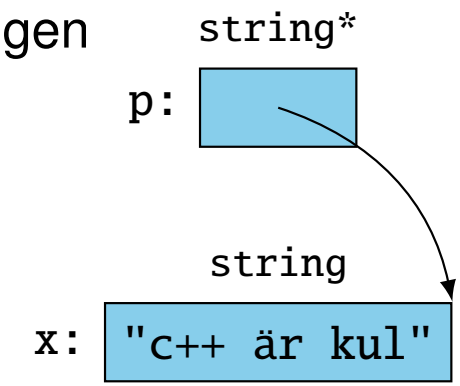
Pekare - Avreferering

Vi kan komma åt en medlem på det utpekade objektet med `operator->`.

- piloperatorn
- samma som `operator*` men smidigare.
- "följ pilen till värdet" - bokstavligen

```
std::string s {"c++ är kul"};
std::string* p {&s};
cout << "'" << *p << " ";
cout << (*p).length() << " ";
cout << p->length() << endl;
```

```
$ ./a.out
c++ är kul 10 10
```



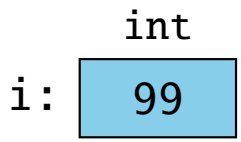
Pekare - this

När vi implementerar en medlemsfunktion i vår klass har vi automatiskt tillgång till pekaren `this` som innehåller adressen av aktuellt objekt.

- Vi behöver inte använda this. När vi anger namnet på en medlem `x` är det förstått att vi menar `this->x`.
- Om det finns en lokal variabler med samma namn skuggar den memdlemen. Byt namn på någon av dem!
- Om vi behöver referera till objektet själv är det okej att använda `*this`.

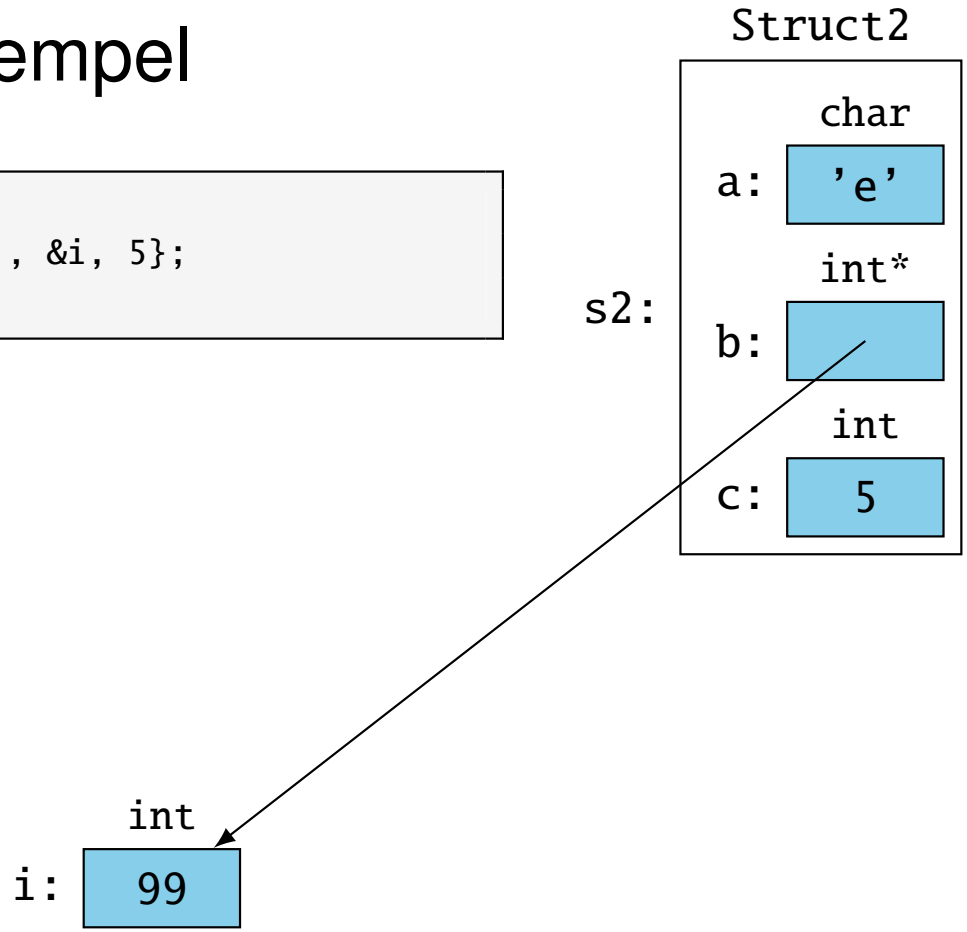
Pekare - Exempel

```
int i {99};
```



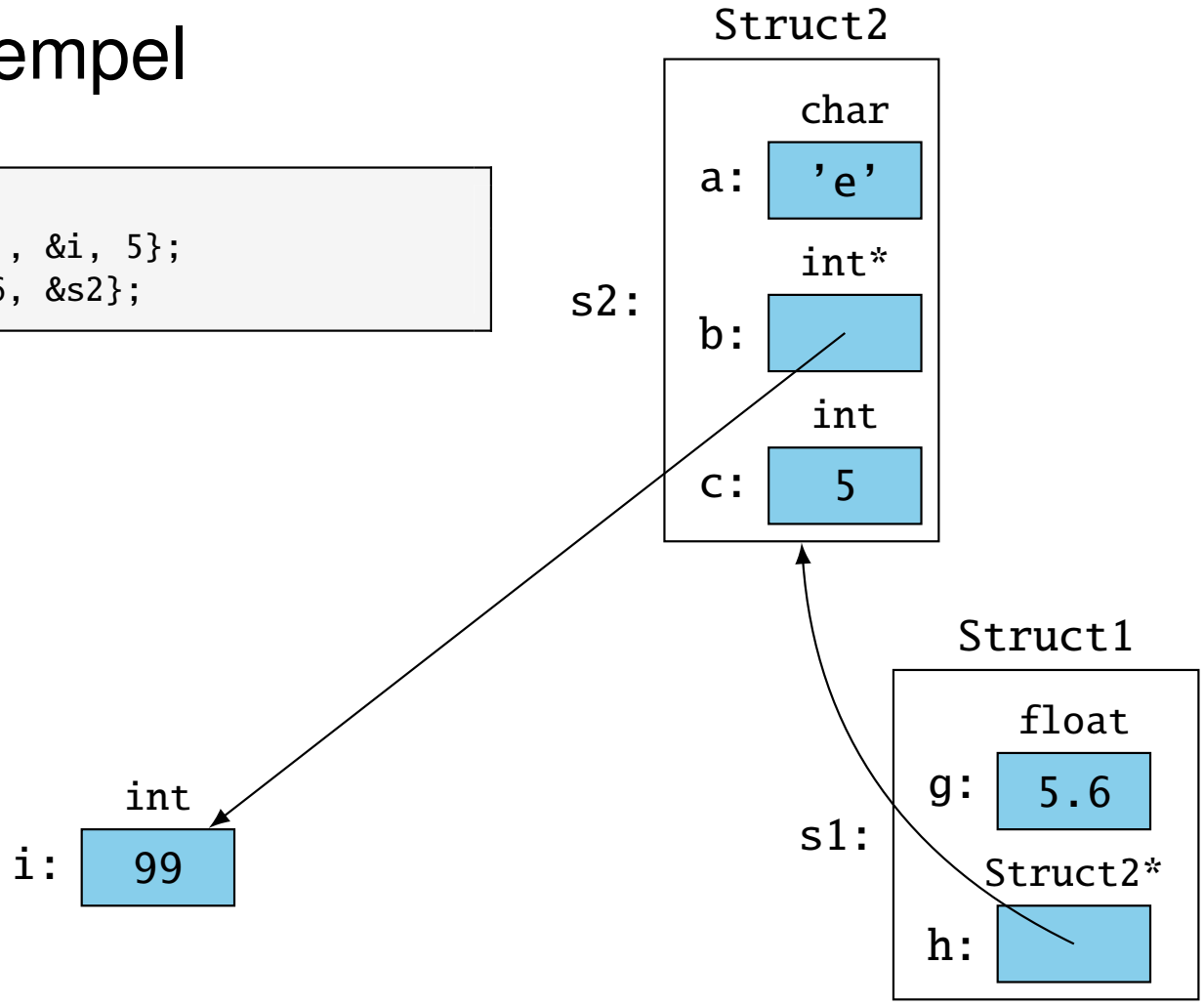
Pekare - Exempel

```
int i {99};
Struct2 s2 {'e', &i, 5};
```



Pekare - Exempel

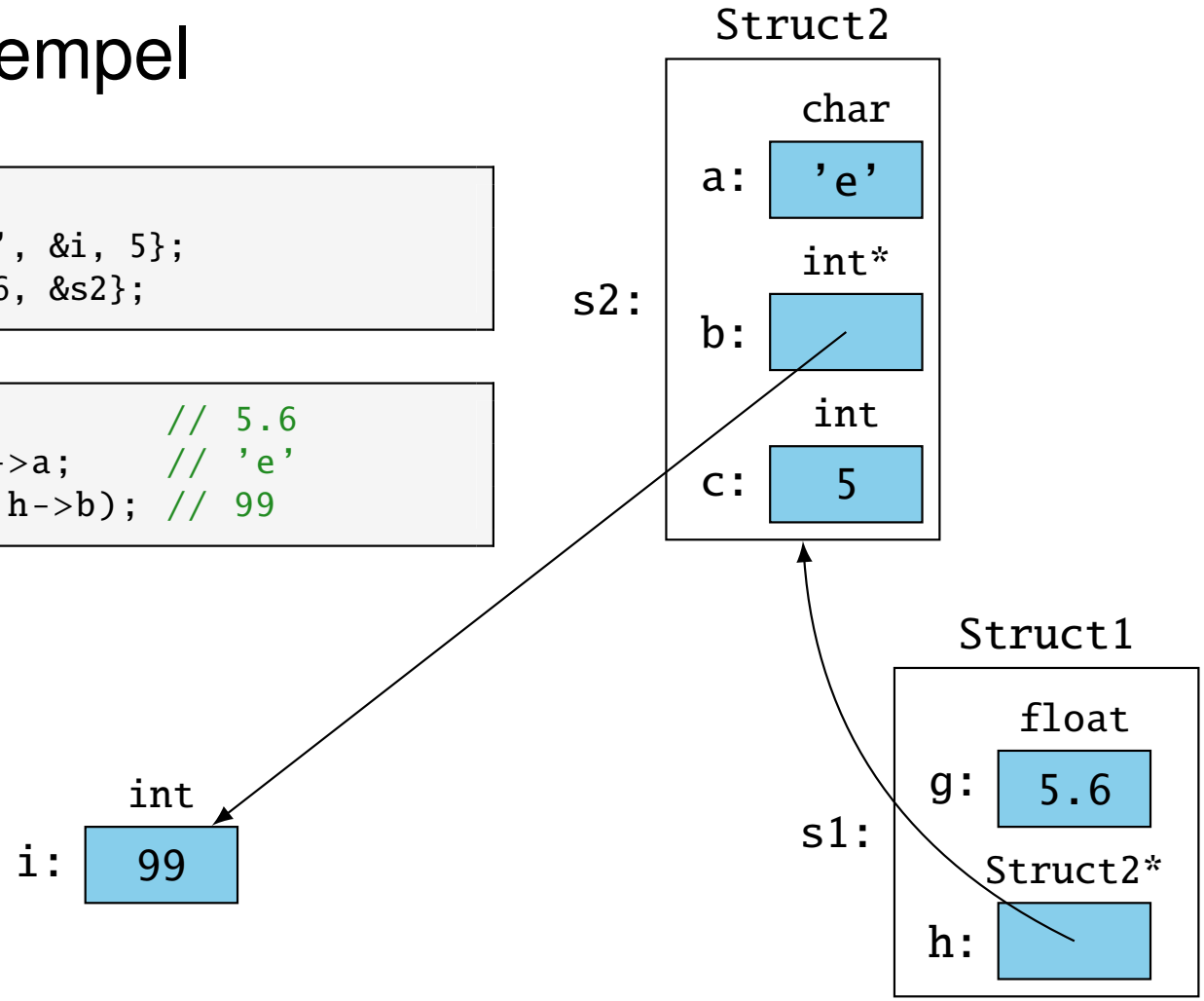
```
int i {99};
Struct2 s2 {'e', &i, 5};
Struct1 s1 {5.6, &s2};
```



Pekare - Exempel

```
int i {99};
Struct2 s2 {'e', &i, 5};
Struct1 s1 {5.6, &s2};
```

```
cout << s1.g;        // 5.6
cout << s1.h->a;     // 'e'
cout << *(s1.h->b); // 99
```



1 Repetition
2 Pekare
3 Dynamiskt minne
4 Länkade strukturer
5 Objektorienterade tankar
6 Speciella medlemsfunktioner
7 Övrigt
8 Livekod

Dynamiskt minne - Heap

Det finns olika ställen att spara variabler under programkörning.

Stack

- automatiska variabler
- Kompilatorn sköter allokering och avallokering
- lokalt scope
- variabler refereras till med namn

Heap

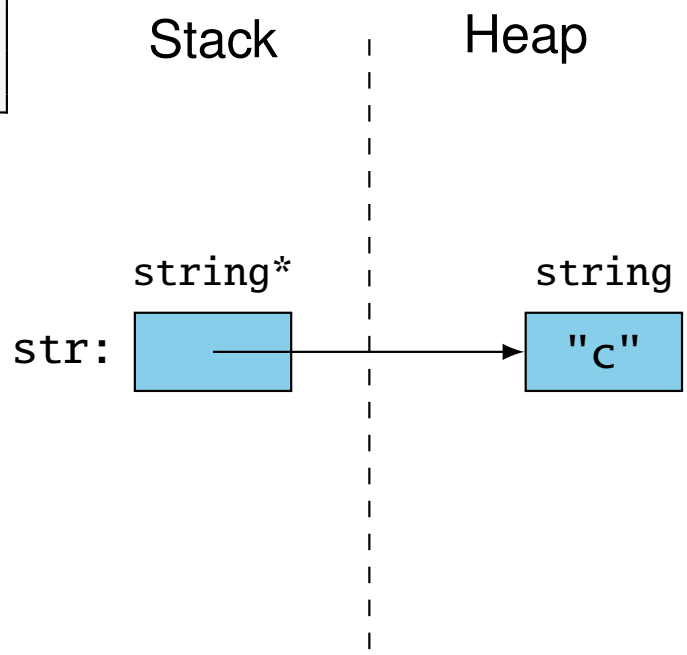
- Anonyma variabler
- Programmeraren sköter allokering och avallokering
- globalt scope
- variabler refereras till med adress

Dynamiskt minne - new

Vi kan allokera dynamiskt minne med nyckelordet `new`.

- Ger en pekare till det nyligen skapade minnet på heapen.

```
string* str {};
str = new string{"c"};
```



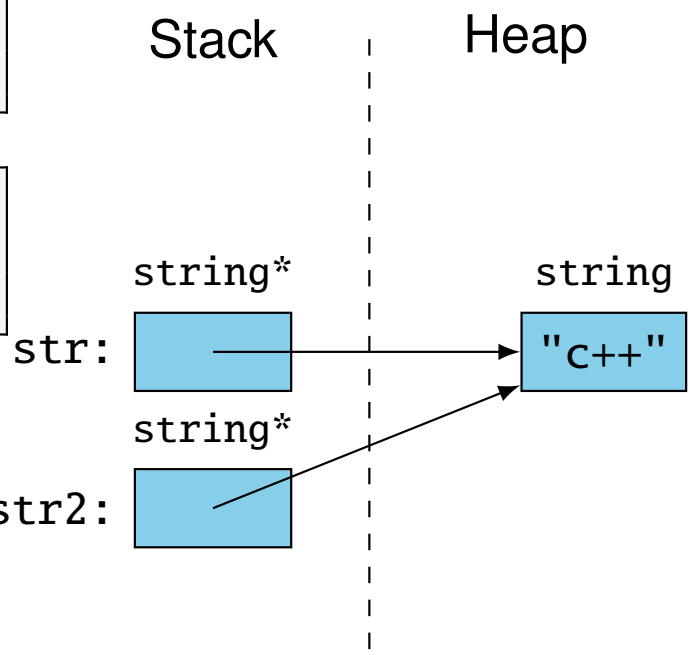
Dynamiskt minne - new

Vi kan allokera dynamiskt minne med nyckelordet `new`.

- Ger en pekare till det nyligen skapade minnet på heapen.

```
string* str {};
str = new string{"c"};
```

```
string* str2 {str};
*str = "c++";
cout << *str2;
```



Dynamiskt minne - delete

Vi kan avallokera minne med nyckelordet `delete`.

- Tar en pekare till dynamiskt allokerat minne.

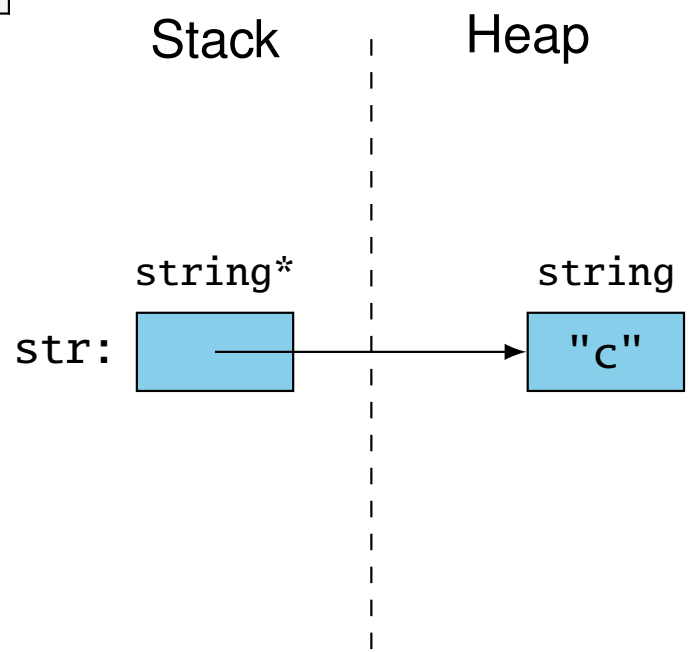
```
string* str {};
str = new string{"c"};

delete str;
str = nullptr;
```

- Vad händer om vi ger `delete` en pekare till stacken?
- Vad händer om vi ger `delete` minne vi inte äger?

Dynamiskt minne - Minnesläcka

Vad händer om vi tappar bort adressen till vår variabel?

```
string* str {};
str = new string{"c"};
```

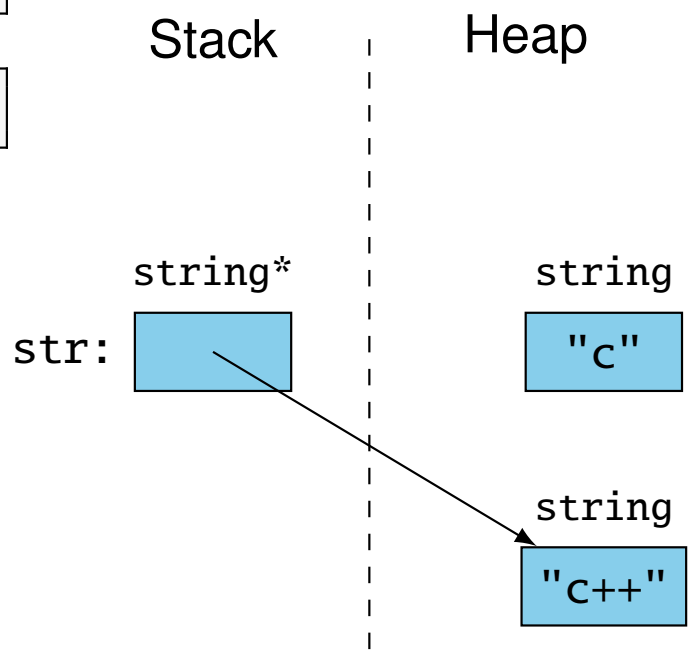


Dynamiskt minne - Minnesläcka

Vad händer om vi tappar bort adressen till vår variabel?

```
string* str {};
str = new string{"c"};
```

```
str = new string{"c++"};
```

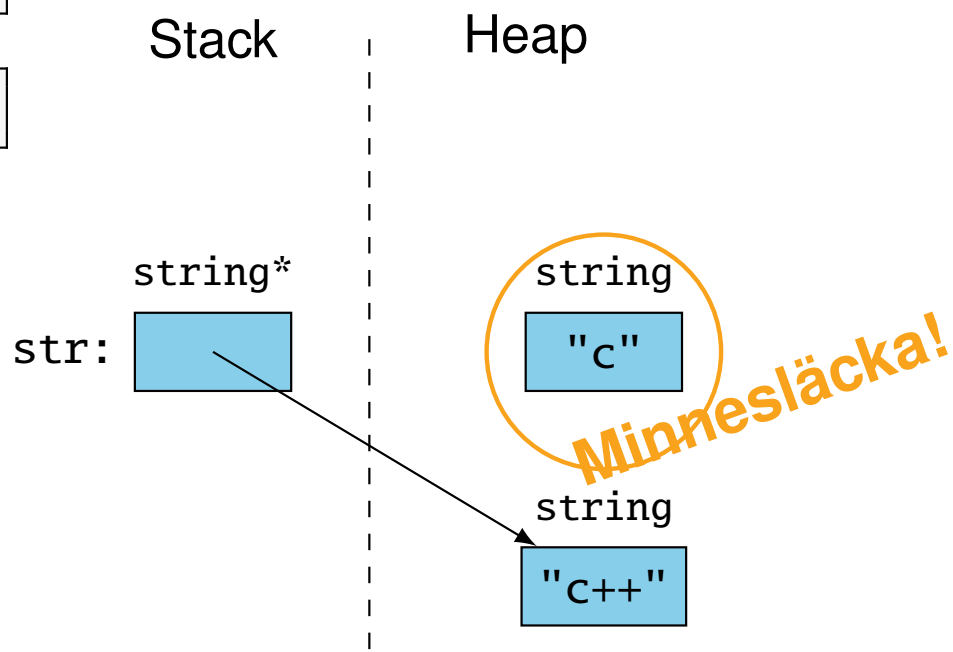


Dynamiskt minne - Minnesläcka

Vad händer om vi tappar bort adressen till vår variabel?

```
string* str {};
str = new string{"c"};
```

```
str = new string{"c++"};
```



Dynamiskt minne - Minnesläcka

Valgrind kan hitta minnesläckor i kod.

```
$ g++ main.cc
$ valgrind ./a.out
```

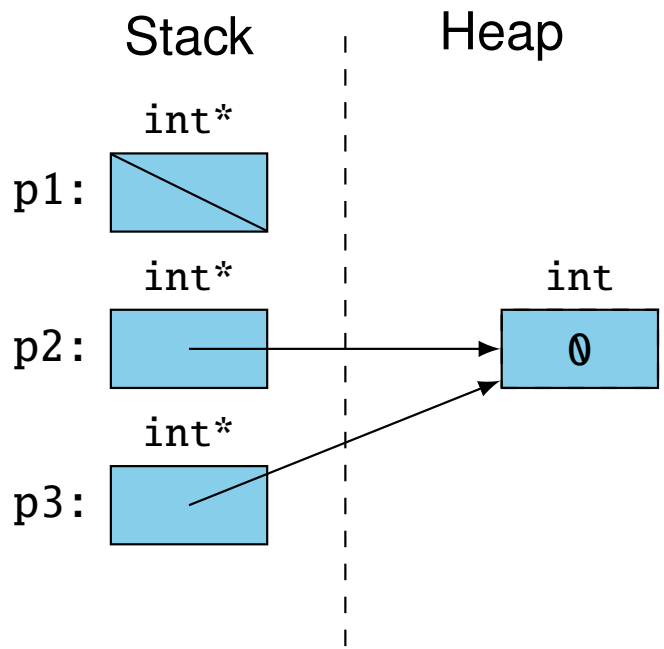
```
==19528== HEAP SUMMARY:
==19528==     in use at exit: 0 bytes in 0 blocks
==19528== All heap blocks were freed -- no leaks are possible
```

```
==19650== HEAP SUMMARY:
==19650==     in use at exit: 4 bytes in 1 blocks
==19650== LEAK SUMMARY:
==19650==    definitely lost: 4 bytes in 1 blocks
==19650==    indirectly lost: 0 bytes in 0 blocks
==19650==      possibly lost: 0 bytes in 0 blocks
==19650==    still reachable: 0 bytes in 0 blocks
==19650==         suppressed: 0 bytes in 0 blocks
```

Dynamiskt minne - Åtkomstfel

Vad händer om vi försöker komma åt minne som inte är vårt?

```
int* p1 {};
int* p2 { new int{} };
int* p3 { p2 };
```



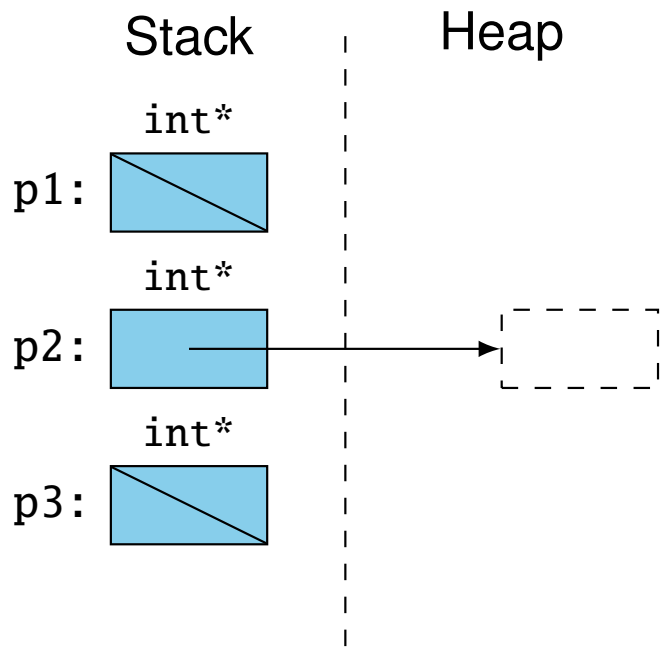
Dynamiskt minne - Åtkomstfel

Vad händer om vi försöker komma åt minne som inte är vårt?

```
int* p1 {};
int* p2 { new int{} };
int* p3 { p2 };
```

```
delete p3;
p3 = nullptr;

cout << *p1; // nullptr
cout << *p2; // borttaget minne
```

```
$ ./a.out
Segmentation fault (core dumped)
```



1 Repetition
2 Pekare
3 Dynamiskt minne
4 Länkade strukturer
5 Objektorienterade tankar
6 Speciella medlemsfunktioner
7 Övrigt
8 Livekod

LINKÖPING UNIVERSITY

Länkade strukturer

“Kan vi skriva en loop som varje iteration skapar en ny variabel? Efter N iterationer ska det finnas N variabler att använda.”

```
for (int i {}; i < N; ++i)
{

  new int{};

}
```

Länkade strukturer

“Kan vi skriva en loop som varje iteration skapar en ny variabel? Efter N iterationer ska det finnas N variabler att använda.”

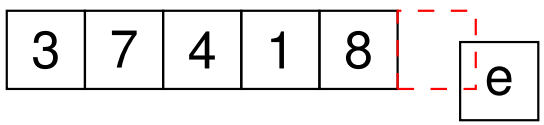
```
for (int i {}; i < N; ++i)
{

  new int{};

}
```

Typ...

Länkade strukturer - Sekventiell lista

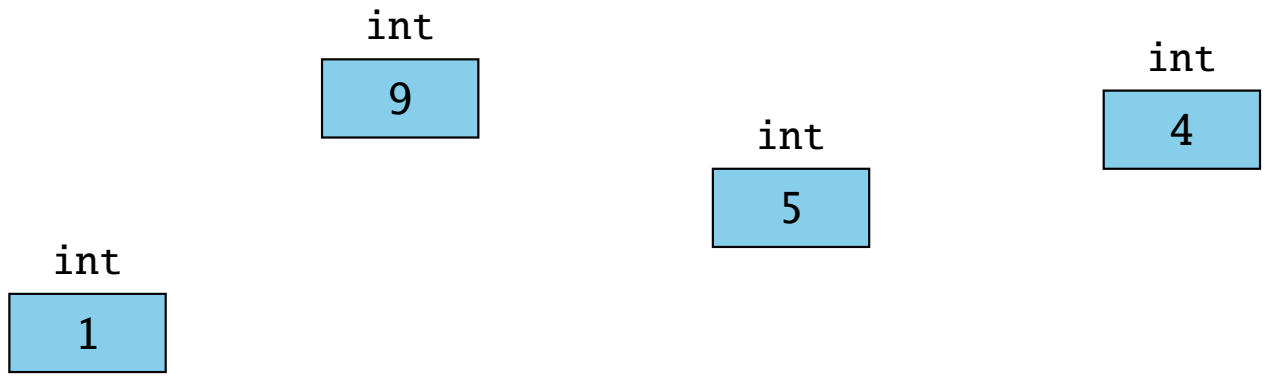
Vi skulle kunna spara alla värden direkt efter varandra på heapen.

- Lätt att komma åt värden (För fjärde värdet, stega tre steg från första)
- Hur vet vi att det finns plats?



Länkade strukturer

Datan får sparas på olika platser i minne istället.

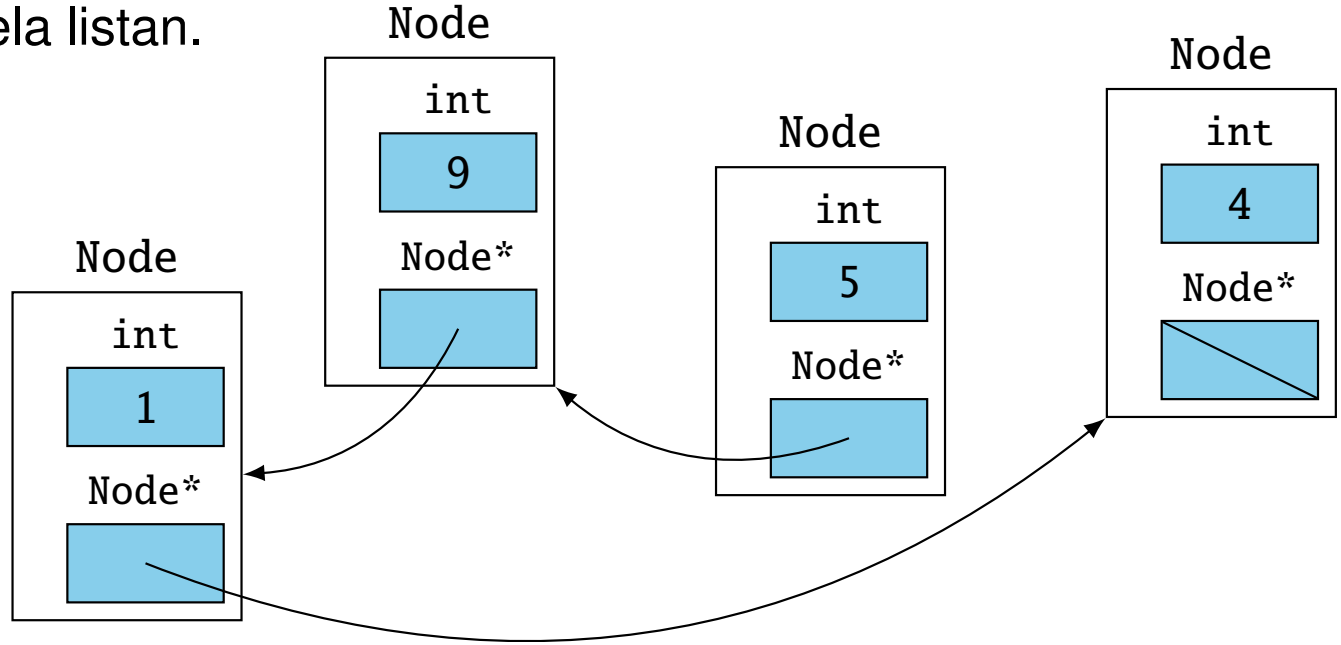


Men i vilken ordning ligger de? Hur kommer vi åt vår data?

Länkade strukturer

Varje box får spara en pekare till nästa box.

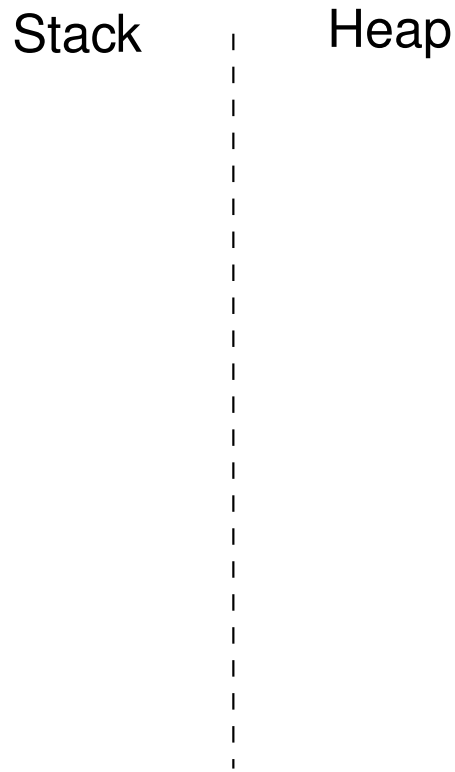
- Vi kan hitta nästa värde genom att besöka värdet innan.
- Vi behöver endast en pekare till första värdet för att ha koll på hela listan.



Länkade strukturer - Nodestrukt

Vi behöver en datastruktur för att representera våra noder.

```
struct Node
{
  int value;
  Node* next;
};
```

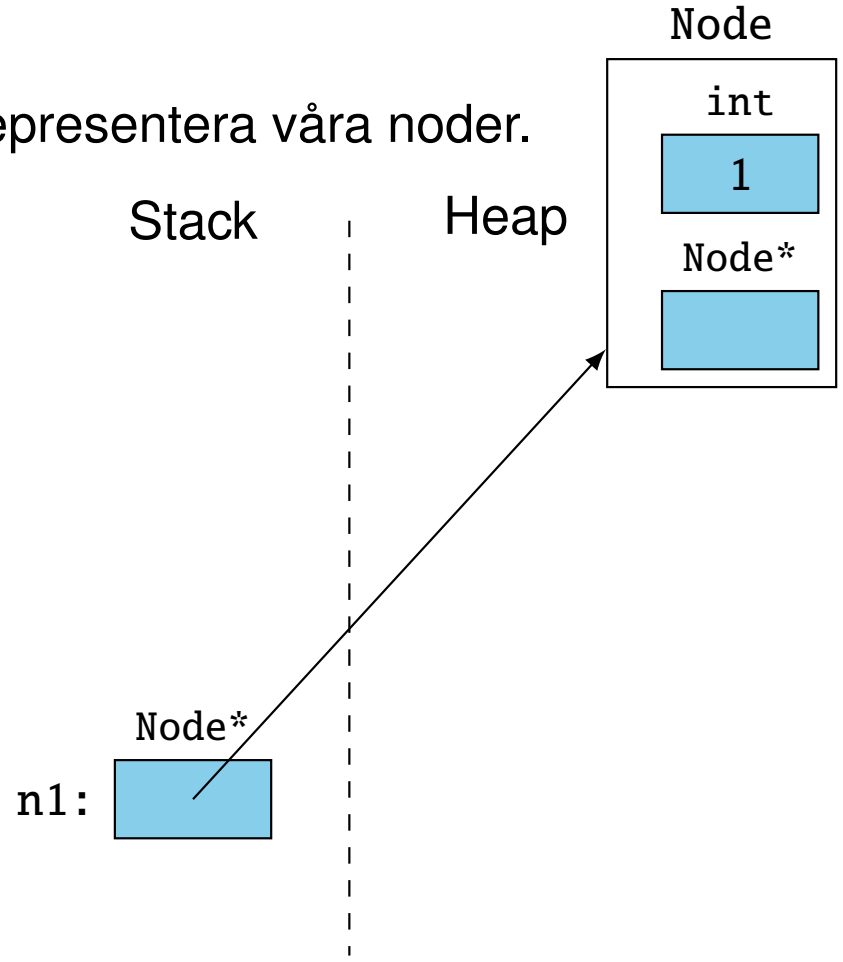


Länkade strukturer - Nodestrukt

Vi behöver en datastruktur för att representera våra noder.

```
struct Node
{
  int value;
  Node* next;
};
```

```
Node* n1 { new Node{1, nullptr} };
```



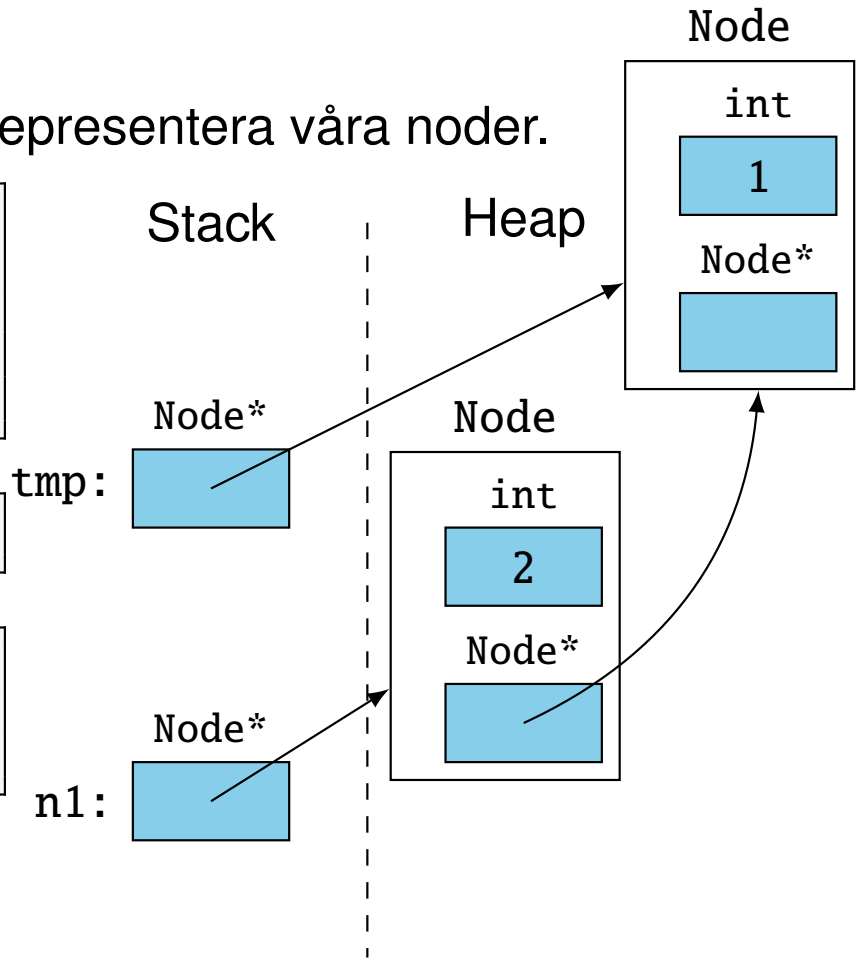
Länkade strukturer - Nodestrukt

Vi behöver en datastruktur för att representera våra noder.

```
struct Node
{
  int value;
  Node* next;
};
```

```
Node* n1 { new Node{1, nullptr} };
```

```
Node* tmp {n1};
n1 = new Node{2, nullptr};
n1->next = tmp;
```



Länkade strukturer - Nodestrukt

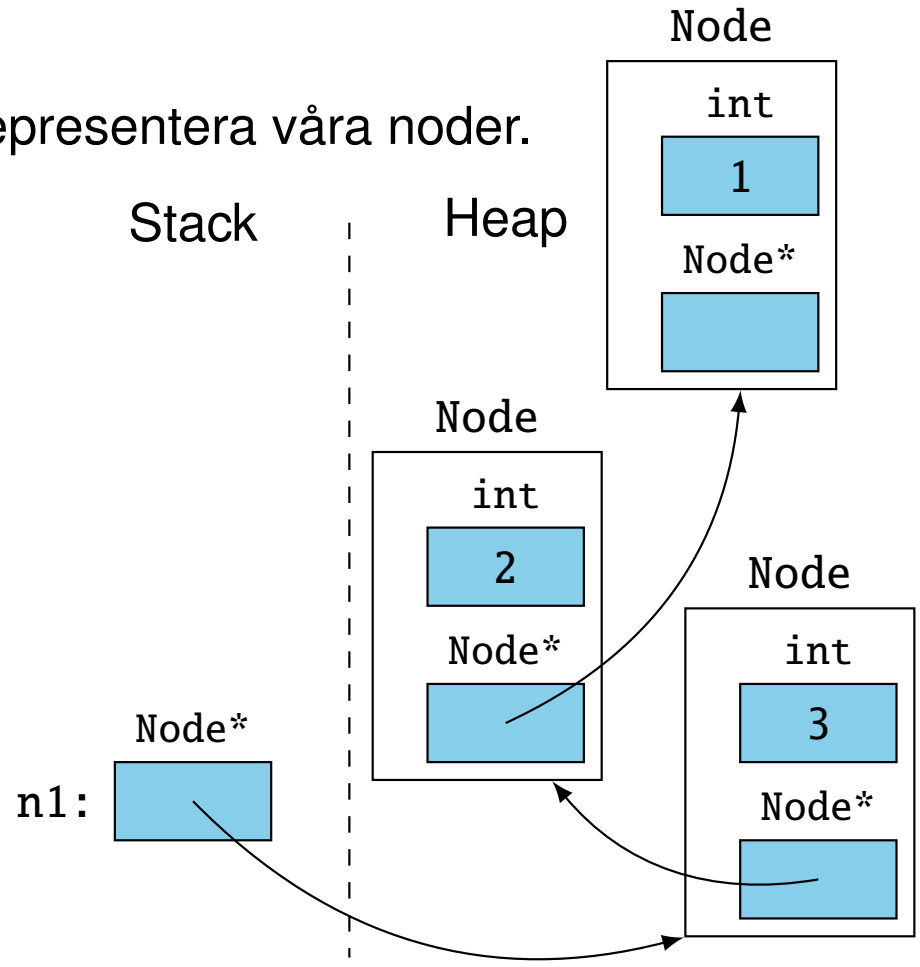
Vi behöver en datastruktur för att representera våra noder.

```
struct Node
{
  int value;
  Node* next;
};
```

```
Node* n1 { new Node{1, nullptr} };
```

```
Node* tmp {n1};
n1 = new Node{2, nullptr};
n1->next = tmp;
```

```
n1 = new Node{3, n1};
```



1 Repetition
2 Pekare
3 Dynamiskt minne
4 Länkade strukturer
5 Objektorienterade tankar
6 Speciella medlemsfunktioner
7 Övrigt
8 Livekod

LINKÖPING UNIVERSITY

Objektorienterade tankar - Abstrakt datatyp

```
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;
}
```

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

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




} // list går ur scope
```

Objektorienterade tankar - Abstrakt datatyp

- För att använda en datastruktur ska man inte behöva veta hur den är implementerad.
  (Jämför: vet du hur std::vector är implementerad?)
- En abstrakt datatyp är en beskrivning av hur en datatyp används, sett från användarens perspektiv.
  (Tänk publika delen av h-filen för en klass.)

Objektorienterade tankar - 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 första pekaren rätt
- Lägg till medlemsfunktioner som gör alla besvärliga pekaroperationer

Objektorienterade tankar - Inre klass

- En inre klass är en klass med begränsad synlighet.
- Node syns nu endast innuti List.
- Medlemmarna i Node är inte medlemmar i List.
- Vid deklaration och definition måste hela "sökvägen" anges.

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

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

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

Objektorienterade tankar - RAII

“Resource acquisition is initialization”

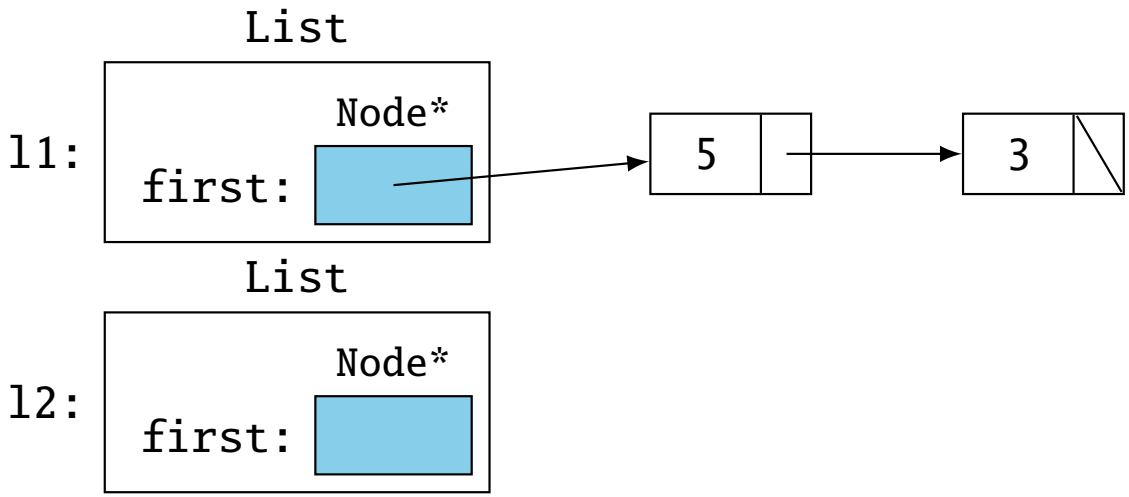
- En resurs som ett objekt behöver allokeras i konstruktorn.
  - anonyma variabler, filer, databasanslutningar, sockets, etc
- Objektet släpper resursen när objektet destrueras.
- Objektet ser till att oväntade fel inte hindrar resursen från att släppas korrekt.
- Resursens livstid är kopplad till objektets.

1 Repetition
2 Pekare
3 Dynamiskt minne
4 Länkade strukturer
5 Objektorienterade tankar
6 Speciella medlemsfunktioner
7 Övrigt
8 Livekod

Speciella medlemsfunktioner

Hur vet vi att vår list-klass kopieras korrekt?

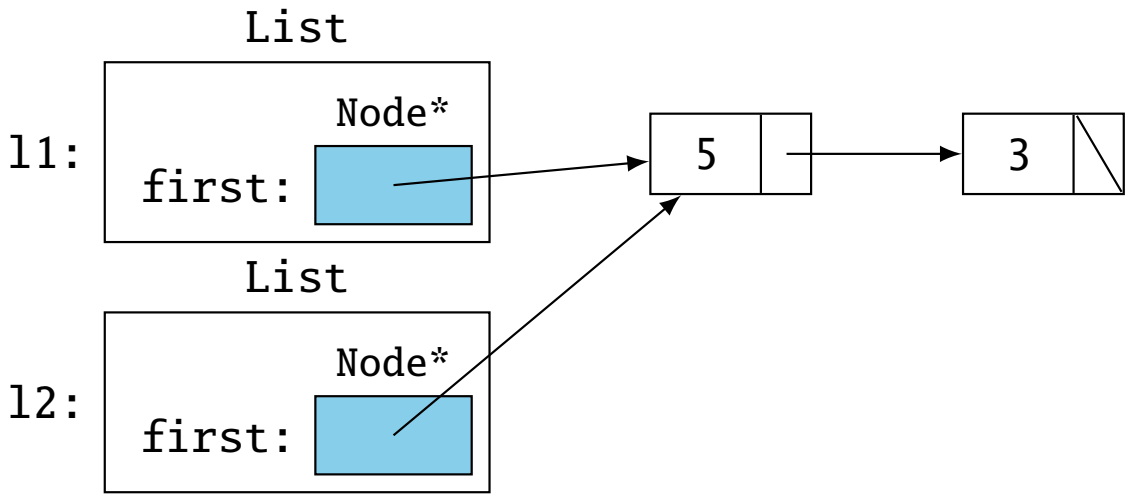
```
List l1 {};
l1.insert(3);
l1.insert(5);

List l2 {l1};
l2.insert(7); // ???
```



Speciella medlemsfunktioner

Hur vet vi att vår list-klass kopieras korrekt?

```
List l1 {};
l1.insert(3);
l1.insert(5);

List l2 {l1};
l2.insert(7); // ???
```



Speciella medlemsfunktioner

Kompilatorn generera ett antal medlemsfunktioner åt oss.

Kopieringskonstruktorn	`Foo(Foo const& other)`
Kopieringstilldelningsoperatorn	`Foo& operator=(Foo const& rhs)`
Destrukturn	`~Foo()`
Flyttkonstruktorn	`Foo(Foo && other)`
Flytttilldelningsoperatorn	`Foo& operator=(Foo && rhs)`

Dessa är de speciella medlemsfunktionerna.

Speciella medlemsfunktioner

När behöver vi själva implementera dessa?

- Om objektet hanterar en unik resurs
  Tex en fil eller ström
- Om objektet inte får kopieras
- Om vi behöver göra något speciellt när objektet tas bort
- Generellt: om objektet ansvarar för dynamiskt minne

Speciella medlemsfunktioner

Kopieringskonstruktor

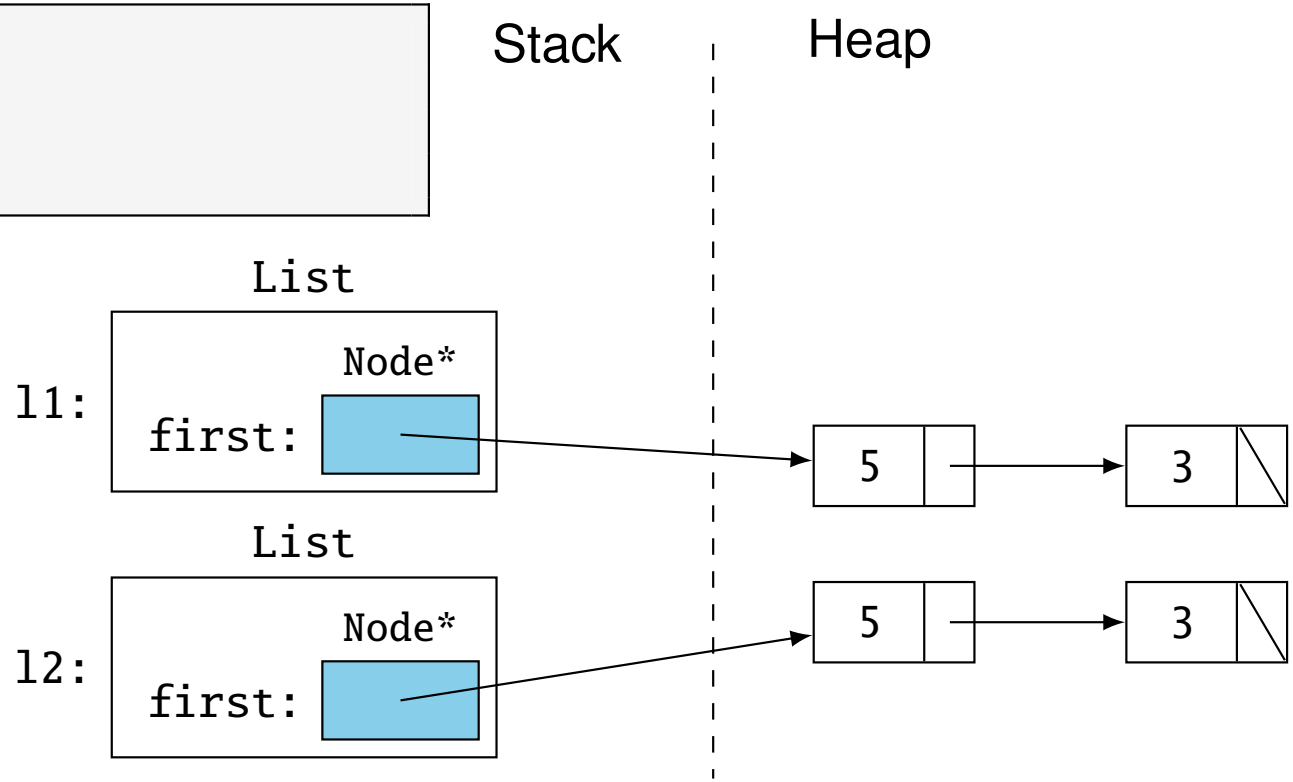
```
Foo(Foo const& other)
```

- Skapa ett nytt objekt utifrån ett annat objekt av samma typ.
- I fallet av länkade strukturer bör denna göra en djup kopia av det andra objektet.
- Anropas automatiskt i många lägen
  - Vid definition av nya variabler
  - När objekt returneras
  - Vid parameteröverföring

Speciella medlemsfunktioner

Kopieringskonstruktur

```
List l1 {};
l1.insert(3);
l1.insert(5);
List l2 {l1};
```



Speciella medlemsfunktioner

Kopieringstilldelningsoperator

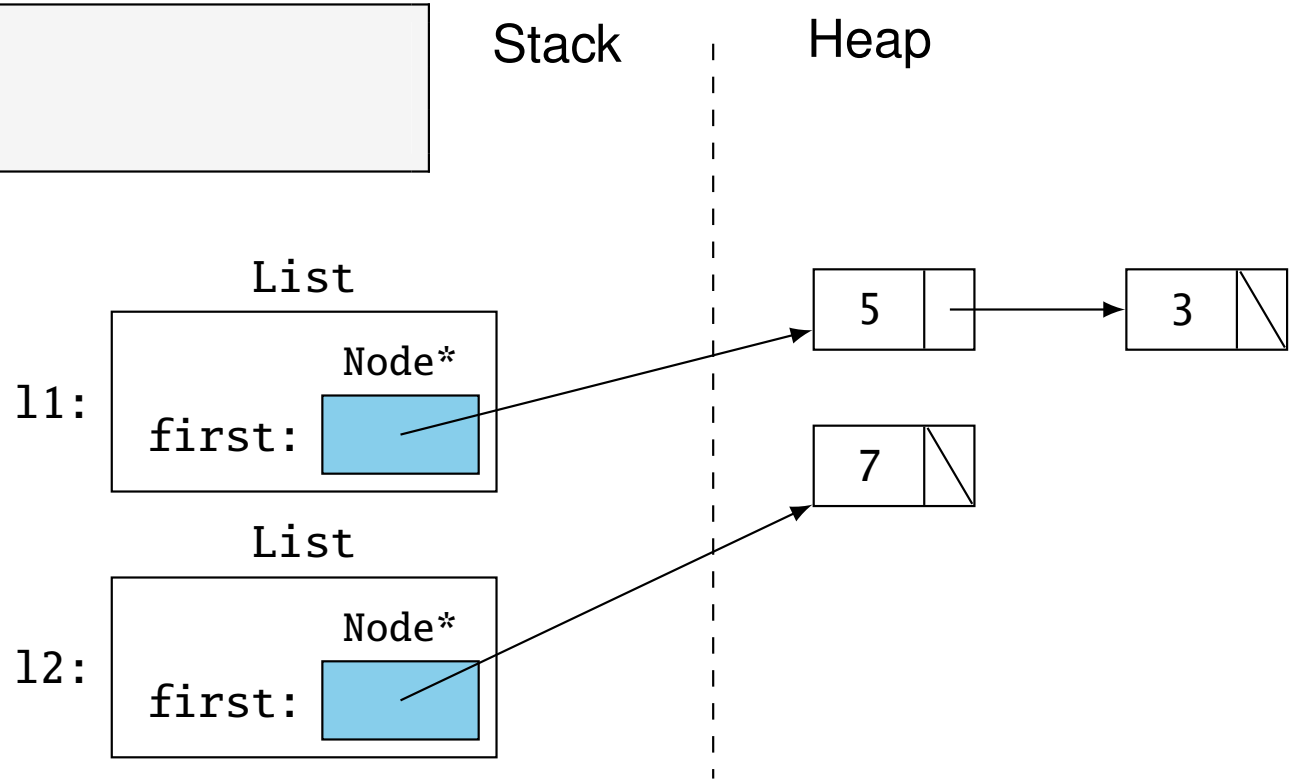
```
Foo& operator=(Foo const& other)
```

- Ersätt ett existerande objekt med innehållet från ett annat objekt av samma typ.
- I fallet av länkade strukturer bör denna göra en djup kopia av det andra objektet.
- Samma funktionallitet som kopieringskonstruktorn men anropas i andra situationer.
- returvärdet är en referens till sig själv

Speciella medlemsfunktioner

Kopieringstilldelningsoperator

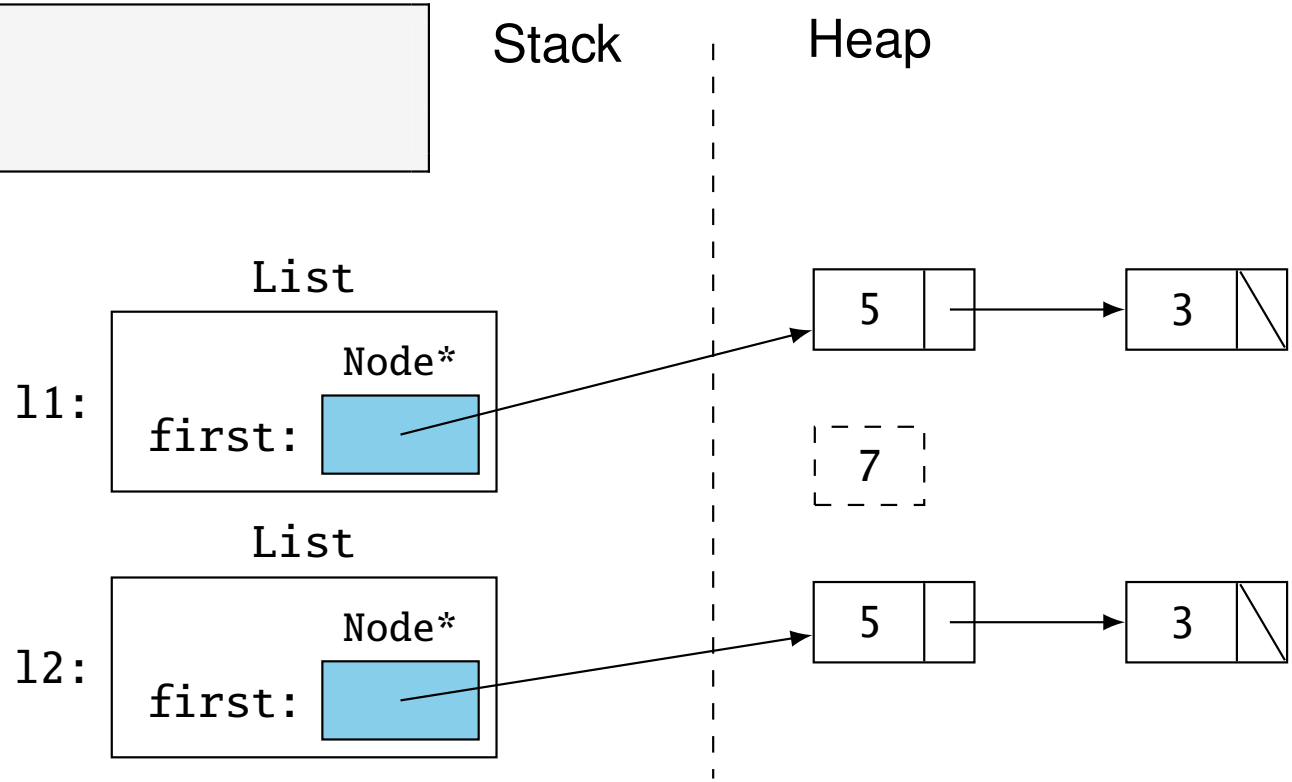
```
List l1 {3, 5};
List l2 {7};
l2 = l1;
```



Speciella medlemsfunktioner

Kopieringstilldelningsoperator

```
List l1 {3, 5};
List l2 {7};
l2 = l1;
```



Speciella medlemsfunktioner

Destruktor

```
~Foo()
```

- Körs när ett objekt ska tas bort.
  Om delete anropas eller objektet går ur scope.
- Bör ansvara för att ta bort allt minne som objektet har allokerat.
- Ska aldrig anropas explicit.

Speciella medlemsfunktioner

Flyttkonstruktor

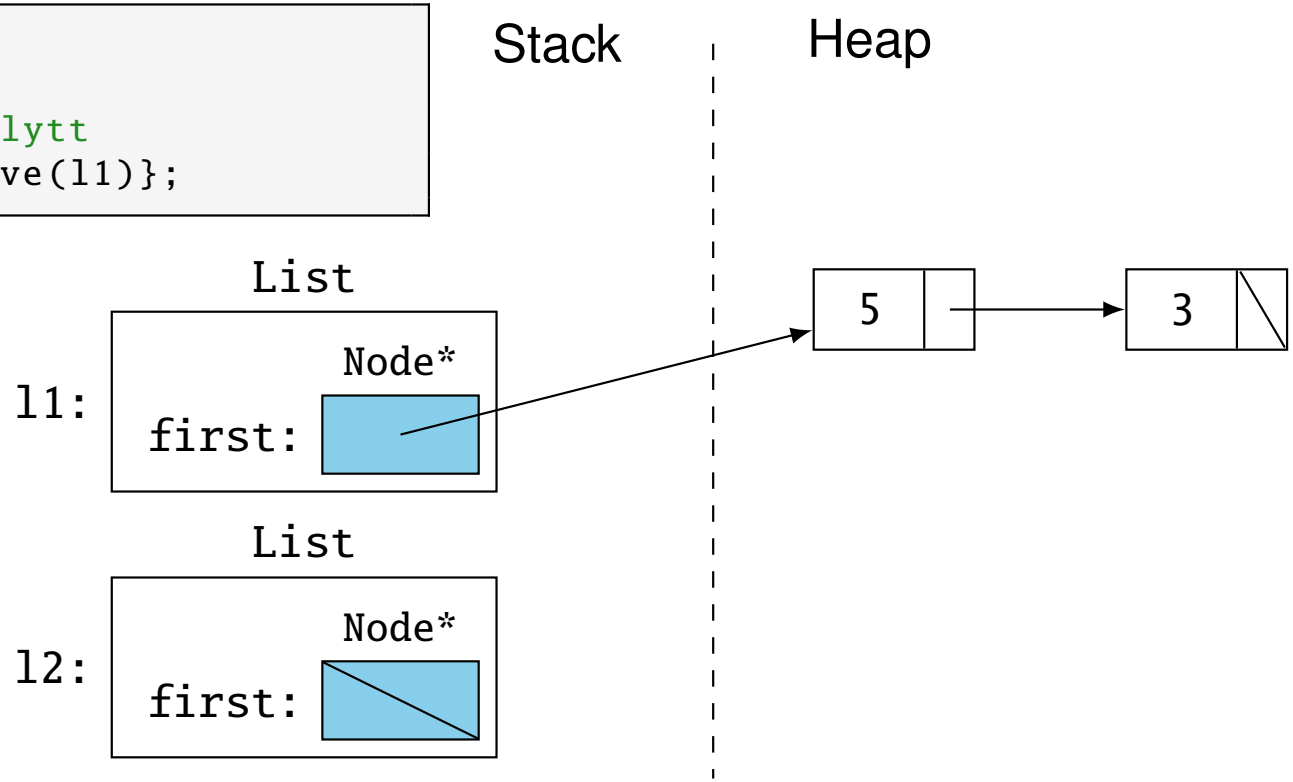
```
Foo(Foo && other)
```

- Skapa ett nytt objekt genom att “sno” innehållet från ett annat objekt av samma typ.
- Används när man vill ha en kopia av ett döende objekt.
- Anropas av kompilatorn för att optimera.
  Bör sällan anropas explicit.

Speciella medlemsfunktioner

Flyttkonstruktor

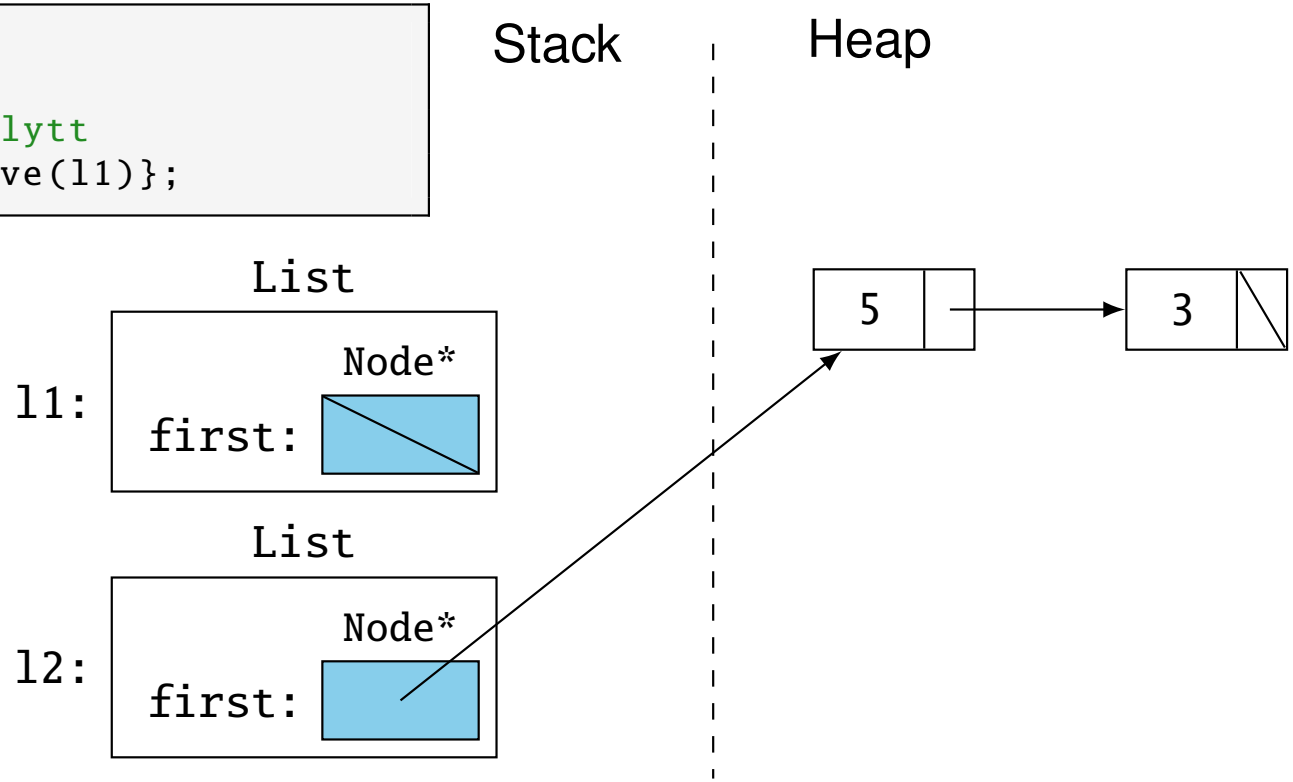
```
List l1 {3, 5};

// tvinga fram flytt
List l2 {std::move(l1)};
```



Speciella medlemsfunktioner

Flyttkonstruktor

```
List l1 {3, 5};

// tvinga fram flytt
List l2 {std::move(l1)};
```



Speciella medlemsfunktioner

Flyttilldelningsoperator

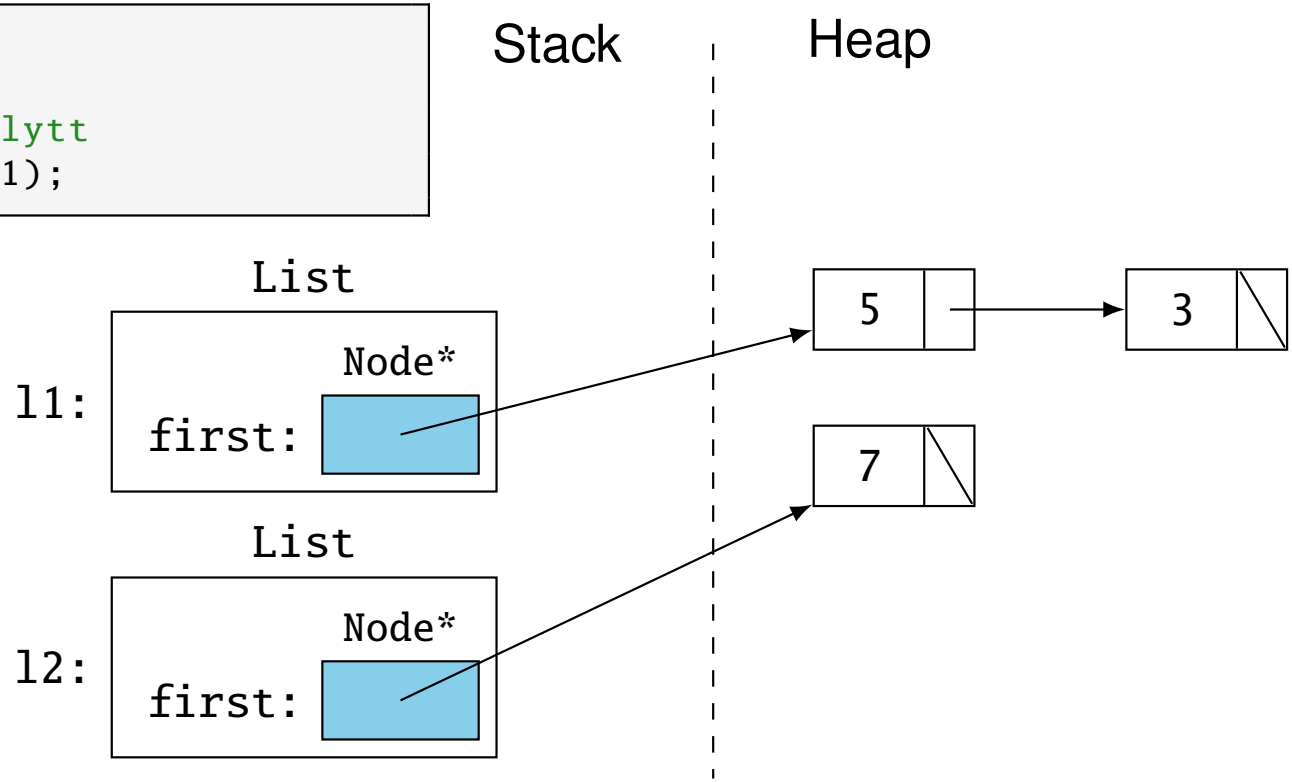
```
Foo& operator=(Foo && other)
```

- Ersätt ett existerande objekt genom att “sno” innehållet från ett annat objekt av samma typ.
- Används när man vill ha en kopia av ett döende objekt.
- Anropas av kompilatorn för att optimera.
  Bör sällan anropas explicit.

Speciella medlemsfunktioner

Flyttilldelningsoperator

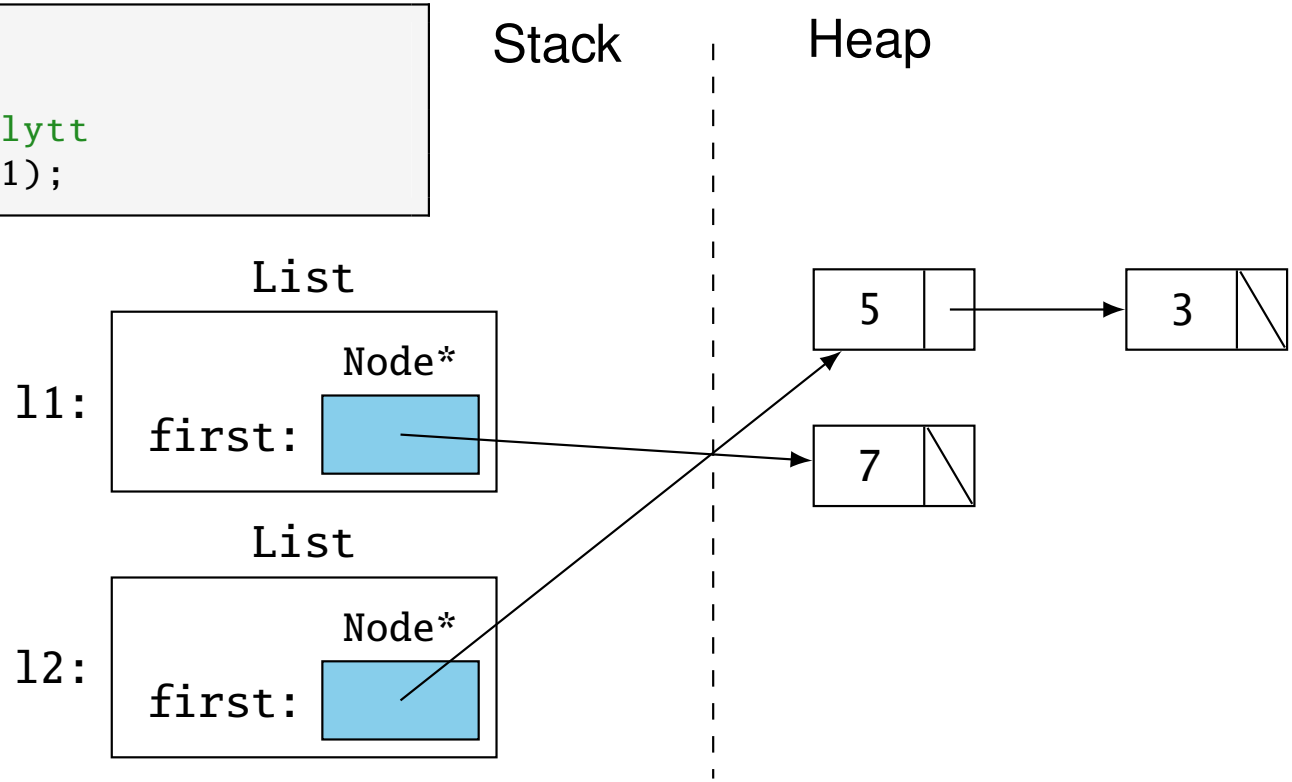
```
List l1 {3, 5};
List l2 {7};
// tvinga fram flytt
l2 = std::move(l1);
```



Speciella medlemsfunktioner

Flyttilldelningsoperator

```
List l1 {3, 5};
List l2 {7};
// tvinga fram flytt
l2 = std::move(l1);
```



Speciella medlemsfunktioner - Rule of five

- Om en av de fem speciella medlemsfunktionerna behöver implementeras så bör man implementera alla fem.
- Om de inte behövs ska man inte implementera någon.

Om man inte implementerar alla fem finns det risk att kompilatorn anropar en av de default-genererade vilket ger problem.

Speciella medlemsfunktioner

- Om vi vill säga åt kompilatorn att använda dess standardimplementation kan vi använda nyckelordet `default`.

```
List(List const& other) = default;
~List() = default;
```

- Om vi vill säga åt kompilatorn att inte generera någon standardimplementation kan vi använda nyckelordet `delete`.

```
List(List const& other) = delete;
List& operator=(List const& other) = delete;
```

1 Repetition
2 Pekare
3 Dynamiskt minne
4 Länkade strukturer
5 Objektorienterade tankar
6 Speciella medlemsfunktioner
7 Övrigt
8 Livekod

std::initializer_list

- Skapas när vi skriver en lista inom måsvingar.
  `{1, 1, 2, 3, 5, 8}`
- Kan stegas genom med en for-loop
- Användbar för att ta emot godtyckligt många argument i en konstruktor.

```
List::List(std::initializer_list<int> list)
{
  for (auto v : list)
  {
    // ...
  }
}
```

Catch - SECTION

Hur fungerar en SECTION i Catch?

```
TEST_CASE("ett test")
{
  int i {4};
  SECTION("sektion 1")
  {
    cout << "s1: " << i << endl;
    i = 7;
  }
  SECTION("sektion 2")
  {
    cout << "s2: " << i << endl;
  }
  cout << "ute: " << i << endl;
}
```

Catch - SECTION

Hur fungerar en SECTION i Catch?

```
TEST_CASE("ett test")
{
  int i {4};
  SECTION("sektion 1")
  {
    cout << "s1: " << i << endl;
    i = 7;
  }
  SECTION("sektion 2")
  {
    cout << "s2: " << i << endl;
  }
  cout << "ute: " << i << endl;
}
```

```
$ ./a.out
s1: 4
ute: 7
s2: 4
ute: 4
===============================================
testcases: 1 | 1 passed
assertions: - none -
```

1 Repetition
2 Pekare
3 Dynamiskt minne
4 Länkade strukturer
5 Objektorienterade tankar
6 Speciella medlemsfunktioner
7 Övrigt
8 Livekod

LINKÖPING UNIVERSITY

www.liu.se

LINKÖPING UNIVERSITY