

TDIU20

Pekare och dynamiskt minne

Eric Ekström

Institutionen för datavetenskap

Handout version

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.

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
}
```

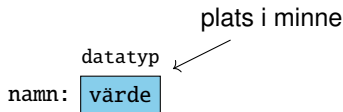
går ur scope

- 1 Repetition
- 2 Pekare
- 3 Dynamiskt minne
- 4 Länkade strukturer
- 5 Objektorienterade tankar
- 6 Speciella medlemsfunktioner

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").

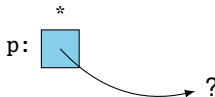


- 1 Repetition
- 2 **Pekare**
- 3 Dynamiskt minne
- 4 Länkade strukturer
- 5 Objektorienterade tankar
- 6 Speciella medlemsfunktioner

Pekare

En pekare är en variabel som innehåller en address.

```
int* p;
```



- 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.”

Pekare - Adressoperatorn

Vi kan hitta adressen av en variabel med operator&.

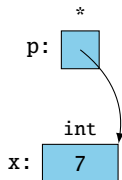
```
int x {7};  
int* p {&x};  
cout << &x << " " << p << endl;
```

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

Adressoperatorn

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

(Används sällan. Undvik i labb!)



Pekare - Avreferering

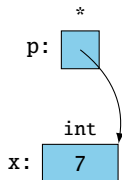
Vi kan hitta värdet på en adress med operator*.

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

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

Avrefereringsoperatören

- "följ pilen till värdet"
- Inte samma som multiplikation eller pekardeklaration!

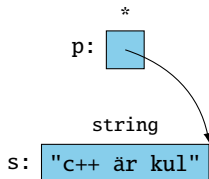


Pekare - Avreferering

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

```
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
```



Piloperatorn

- Samma som `operator*`, fast smidigare.
- "följ pilen till värdet" - bokstavligen

Pekare - this

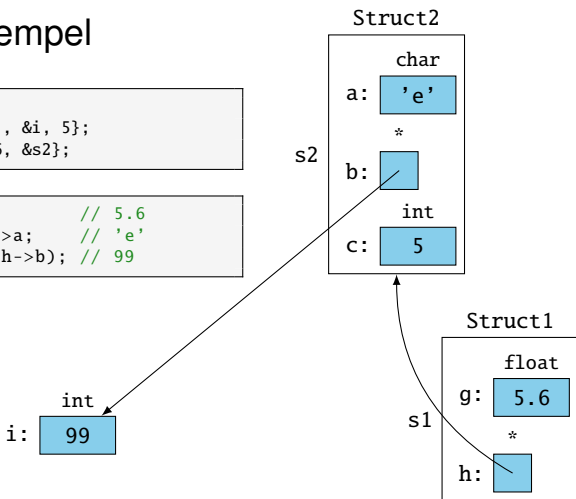
När vi implementerar en medlemsfunktion i vår klass har vi automatiskt tillgång till pekaren `this` som innehåller adressen av aktuellt objekt.

- 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 variabel med samma namn skuggar den medlemmen. Byt namn på någon av dem!
- Om vi behöver referera till objektet själv är det okej att använda `*this`.

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

Dynamiskt minne - Heap

Programmets minne är delat i två.

Stack-minne

- Automatiska variabler
- Variabler refereras till med namn
- Kompilatorn sköter allokering och avallokering
- Lokalt scope

Heap-minne

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

Dynamiskt minne - new

Vi kan allokera dynamiskt minne med nyckelordet `new`.

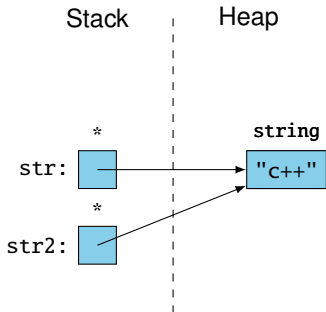
- Ger en pekare till det nyligen skapade minnet på heapen.

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

```
string* str2 {str};
```

```
*str2 = "c++";
```

```
cout << *str;
```



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?

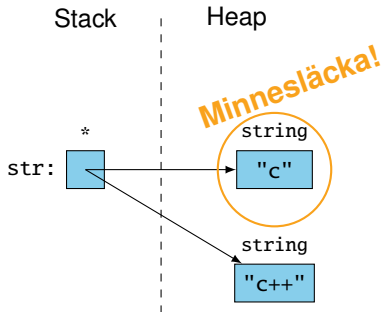
Vad händer om vi ger `delete` minne vi inte äger?

Dynamiskt minne - Minnesläcka

Vad händer om vi tappar bort adressen till vår variabel?

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

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



Dynamiskt minne - Minnesläcka

Valgrind kan hitta minnesläckor i kod.

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

```
==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
```

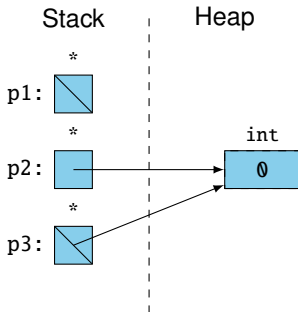
Dynamiskt minne - Åtkomstfel

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

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

```
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

Länkade strukturer

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

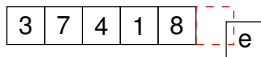
```
for (int i {}; i < N; ++i)
{
    new int{};
}
```

Typ...

Länkade strukturer - Sekventiell lista

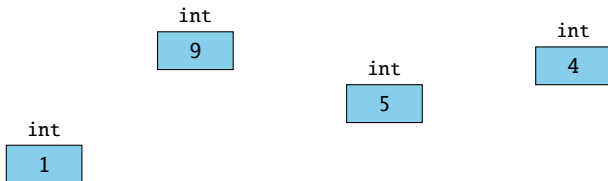
Vi skulle kunna spara alla värden direkt efter varandra på heapen.

- 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.

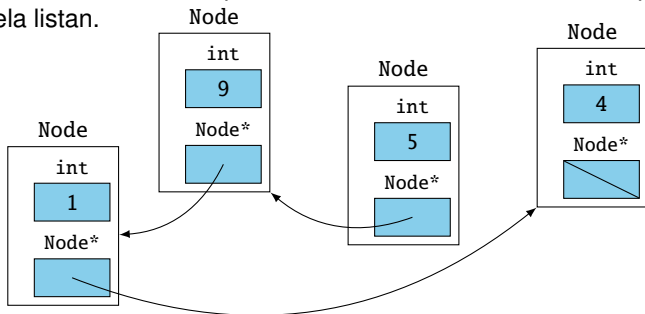


Men i vilken ordning ligger de? Hur kommer vi åt vår data?

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

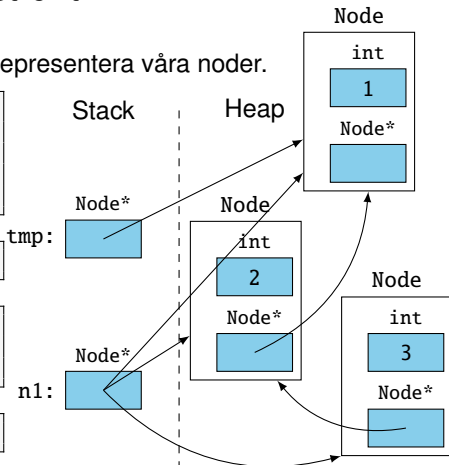
Vi behöver en datastruktur för att representera våra noder.

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

```
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

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

Objektorienterade tankar - Ansvar

“Objektet ansvarar för alla resurser den allokerar”

- Alla resurser 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.

Objektorienterade tankar - Inre klass

```
class List
{
public:

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

- 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.

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

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

- 1 Repetition
- 2 Pekare
- 3 Dynamiskt minne
- 4 Länkade strukturer
- 5 Objektorienterade tankar
- 6 Speciella medlemsfunktioner

Speciella medlemsfunktioner

En klassdeklaration.

```
class List
{
public:
    void insert(int i);

private:

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

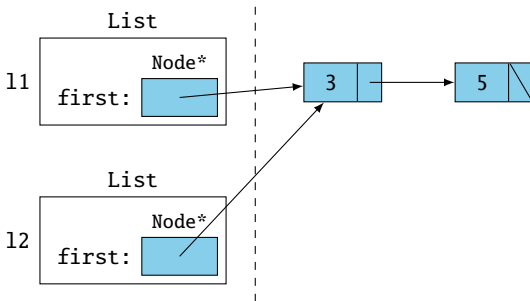
    Node* first;
};
```

Speciella medlemsfunktioner

Vad händer om vi försöker kopiera listan?

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

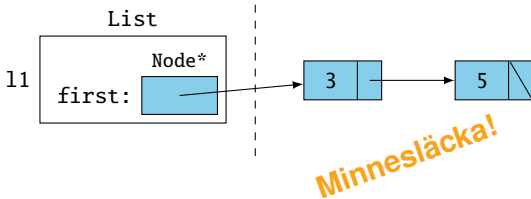
    List l2 {l1};
}
```



Speciella medlemsfunktioner

Vad händer när listan går ur scope?

```
int main()
{
    List l1 {};
    l1.insert(3);
    l1.insert(5);
} //???
```



Speciella medlemsfunktioner

Kompilatorn genererar ett antal medlemsfunktioner åt oss.

Kopieringskonstruktor	<code>X(X const& other)</code>
Kopieringstilldelningsoperator	<code>X& operator=(X const& rhs)</code>
Destruktur	<code>~X()</code>
Flyttkonstruktor	<code>X(X && other)</code>
Flytttilldelningsoperator	<code>X& operator=(X && rhs)</code>

Dessa är de speciella medlemsfunktionerna.

Speciella medlemsfunktioner

När behöver vi själva implementera dessa?

- Om objektet hanterar en unik resurs
Tex dynamiskt minne eller en fil
- Om objektet inte får kopieras
- Om vi behöver göra något speciellt när objektet tas bort

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

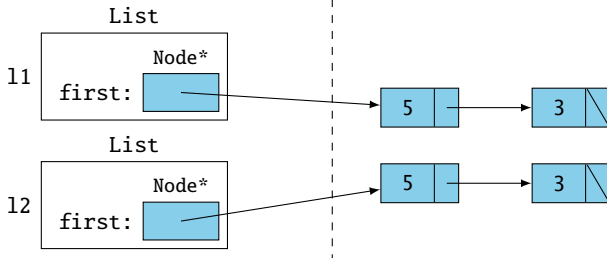
Speciella medlemsfunktioner

Kopieringskonstruktör

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

Stack

Heap



Speciella medlemsfunktioner

Kopieringstilldelningsoperator

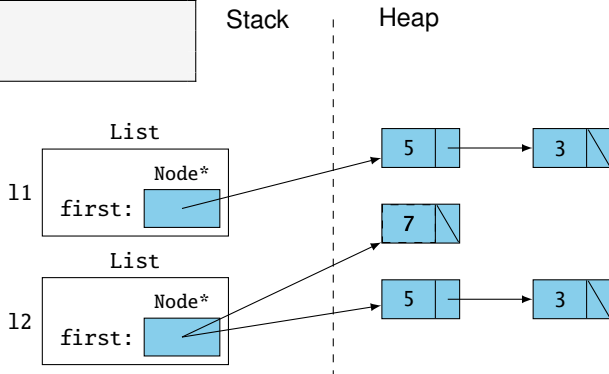
```
Foo& operator=(Foo const& other)
```

- 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

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

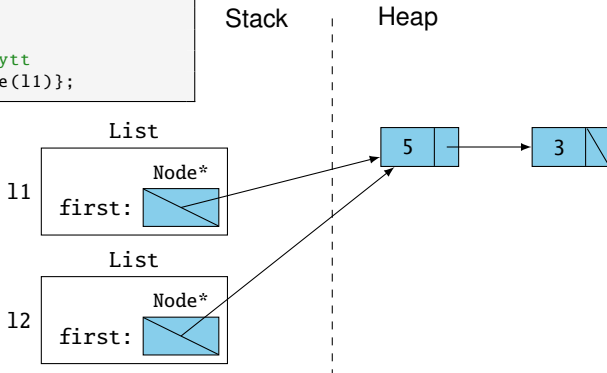
```
Foo(Foo && other)
```

- 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

Flyttkonstruktör

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



Speciella medlemsfunktioner

Flyttilldelningsoperator

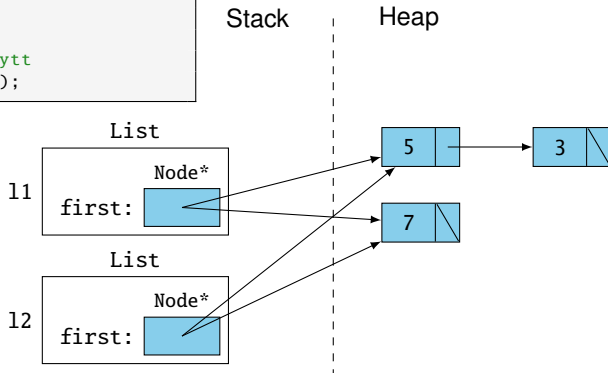
```
Foo& operator=(Foo && other)
```

- 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.

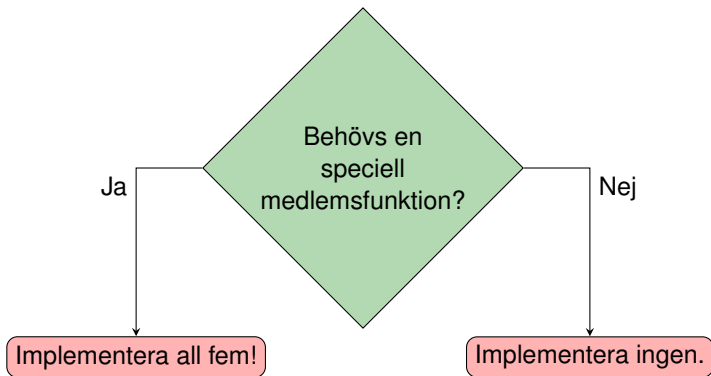
Speciella medlemsfunktioner

Flyttilldelningsoperator

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



Speciella medlemsfunktioner - Rule of five



Speciella medlemsfunktioner

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

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

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

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

www.ida.liu.se/~TDIU20/