TDIU20 Föreläsning 4-5

Klas Arvidsson / Eric Ekström

LINKÖPINGS UNIVERSITET

1. Påverka

Kritiska händelser

- Vi är nyfikna på hur ni upplever kursen.
- Saker som positivt eller negativt avvikit från det du förväntat.
- Du behöver inte fundera på detta nu, men jag vill gärna du slänger iväg en rad om du skulle bli positivt överraskad av något du inte förväntat, eller om du blir sur över något du väntat skulle lösas smidigare. Och hur hade du väntat det skulle fungera?

Berätta vad du tycker direkt

- Om vi så snabbt som möjligt får veta vad som avviker från förväntan kan vi utveckla kursen till er fördel.
- Berätta direkt, vänta inte tills enbart nästa års studenter får glädje av eventuell förändring.
- Positiv respons gör oss mer engagerade för er och hjälper oss veta vad vi lyckats med.
- Negativ respons hjälper oss åtgärda problem för att göra kursen bättre för er.

2. Automatiska namngivna variabler

Konceptuell bild av hur de fungerar

Direkt åtkomst via namnet

Vad vi kan göra och inte

Referenser

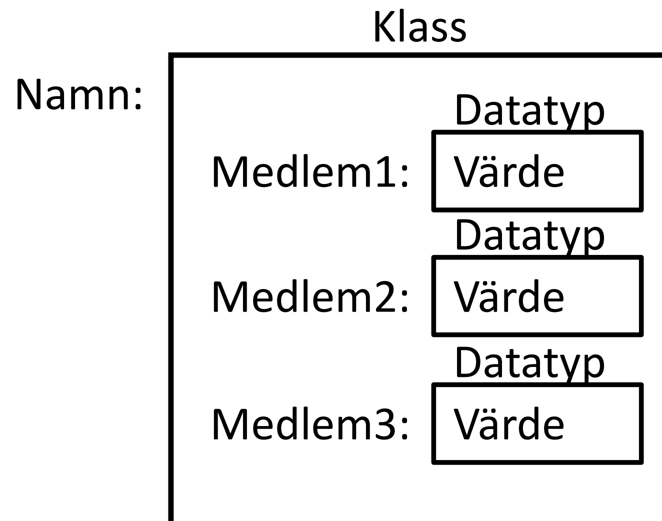
Repetition: Vad är en variabel

- Automatiska namngivna variabler
- ”Låda” för att lagra ett värde i minnet
  - Namn
  - Värde
  - Datatyp

- Kompilatorn automatiskt det som behövs för att varje variabel får sin egen plats i minnet och att bitmönstret där tolkas enligt datatypen vi valt
- En (exekverings)stack används för att organisera minnesanvändningen för automatiska variabler

Repetition: Objekt och instans

- Variabel av klasstyp. Mer saker i lådan helt enkelt.

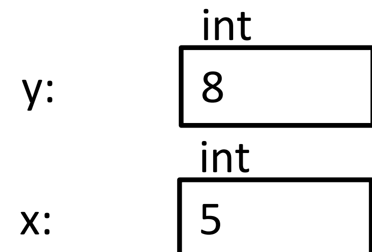


Konceptuellt: Hur hanteras variabler?

```
2:void swap_if(int& a, int& b, bool cond) {
    if ( cond ) {
3:      int c{a};
        a = b;
        b = c;
4:  }
5:}
6:void print(int x) {
    cout << x << endl;
7:}
  int main() {
1:  int x{5}, y{8};
    swap_if(x, y, x < y);
    print(x);
8:}
```

Konceptuellt: Hur hanteras variabler?

```
2:void swap_if(int& a, int& b, bool cond) {
    if ( cond ) {
3:      int c{a};
        a = b;
        b = c;
4:  }
5:}
6:void print(int x) {
    cout << x << endl;
7:}
  int main() {
1:  int x{5}, y{8};
    swap_if(x, y, x < y);
    print(x);
8:}
```



Konceptuellt: Hur hanteras variabler?

```
2:void swap_if(int& a, int& b, bool cond) {
    if ( cond ) {
3:      int c{a};
        a = b;
        b = c;
4:    }
5:}
6:void print(int x) {
    cout << x << endl;
7:}
  int main() {
1:  int x{5}, y{8};
    swap_if(x, y, x < y);
    print(x);
8:}
```

	type	value
cond:	bool	true
y, b:	int	8
x, a:	int	5

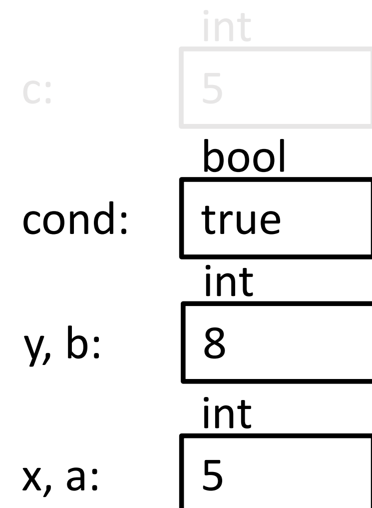
Konceptuellt: Hur hanteras variabler?

```
2:void swap_if(int& a, int& b, bool cond) {
    if ( cond ) {
3:      int c{a};
      a = b;
      b = c;
4:  }
5:}
6:void print(int x) {
    cout << x << endl;
7:}
  int main() {
1:  int x{5}, y{8};
    swap_if(x, y, x < y);
    print(x);
8:}
```

	type	value
c:	int	5
cond:	bool	true
y, b:	int	8
x, a:	int	5

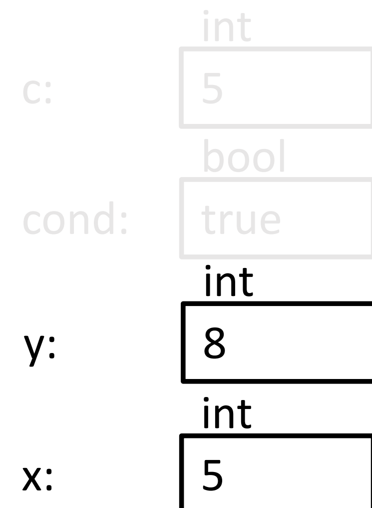
Konceptuellt: Hur hanteras variabler?

```
2:void swap_if(int& a, int& b, bool cond) {
    if ( cond ) {
3:      int c{a};
        a = b;
        b = c;
4:  }
5:}
6:void print(int x) {
    cout << x << endl;
7:}
  int main() {
1:  int x{5}, y{8};
    swap_if(x, y, x < y);
    print(x);
8:}
```



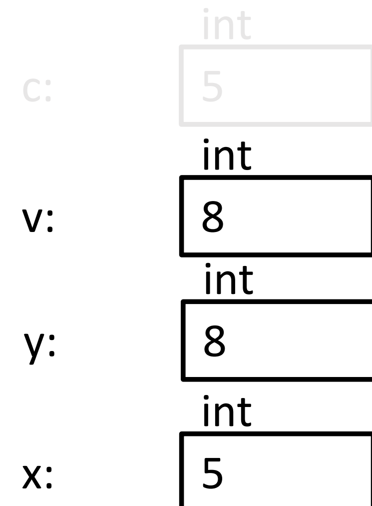
Konceptuellt: Hur hanteras variabler?

```
2:void swap_if(int& a, int& b, bool cond) {
    if ( cond ) {
3:      int c{a};
        a = b;
        b = c;
4:  }
5:}
6:void print(int x) {
    cout << x << endl;
7:}
  int main() {
1:  int x{5}, y{8};
    swap_if(x, y, x < y);
    print(x);
8:}
```



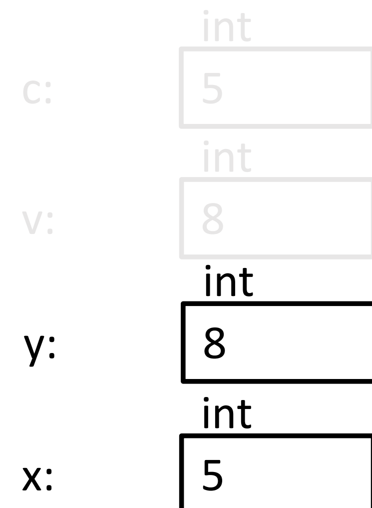
Konceptuellt: Hur hanteras variabler?

```
2:void swap_if(int& a, int& b, bool cond) {
    if ( cond ) {
3:      int c{a};
        a = b;
        b = c;
4:  }
5:}
6:void print(int v) {
    cout << v << endl;
7:}
  int main() {
1:  int x{5}, y{8};
    swap_if(x, y, x < y);
    print(x);
8:}
```



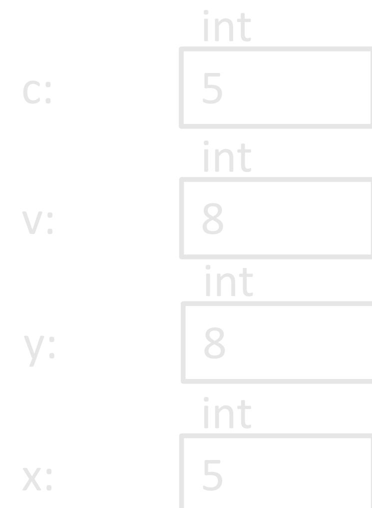
Konceptuellt: Hur hanteras variabler?

```
2:void swap_if(int& a, int& b, bool cond) {
    if ( cond ) {
3:      int c{a};
        a = b;
        b = c;
4:    }
5:}
6:void print(int v) {
    cout << v << endl;
7:}
  int main() {
1:  int x{5}, y{8};
    swap_if(x, y, x < y);
    print(x);
8:}
```



Konceptuellt: Hur hanteras variabler?

```
2:void swap_if(int& a, int& b, bool cond) {
    if ( cond ) {
3:      int c{a};
        a = b;
        b = c;
4:  }
5:}
6:void print(int v) {
    cout << v << endl;
7:}
  int main() {
1:  int x{5}, y{8};
    swap_if(x, y, x < y);
    print(x);
8:}
```



Begränsningar

- Kan vi referera till en variabel längre ned?
- Kan vi referera till en variabel högre upp?

```
Book& new_book(string const& title)
{
  // title refererar till variabel längre ned
  Book c{title};
  return c;
}
int main()
{
  string cpp{"C++"};
  Book& b{ new_book(cpp) };
  // b refererar till variabel högre upp, går detta bra?
  print("Vad händer nu?");
}
```

Men om vi vill ha en variabel kvar då?

- Med automatiska namngivna variabler:
  - No can do.
  - Too bad.
  - Do not compute.
  - Impossible.

- Vi behöver en annan modell:
  - Dynamiskt minne!

3. Dynamiskt minne

Pekare: en automatisk namngiven variabel för att hålla reda på ”namnet” (adressen) av en anonym indirekt variabel

`new, delete, nullptr, *, ->, &`

Konceptuell bild av hur det fungerar

Krav på användning, ansvar, ägarskap

Minnesläckor

Bud Lawson

- 1937, föddes i Philadephia USA
- 1964, uppfann **pekaren** i högnivåspråk
  - tidigare kunde en post bara innehålla data (text, tal)
  - såg möjligheterna med att en post kan peka ut en annan post
  - beskrev hur detta kunde lösas i språket PL/I
- 1971, kom till Linköping
  - Datasaab
  - Professor telekommunikation och datorsystem, LiTH
- 1983, grundade **Institutionen för datavetenskap** med Eric Sandewall
- 2000, ”Computer Pioneer Award” av IEEE för **pekaren**
- 2019, inrättade ”Lawson stipendium”

Källor:

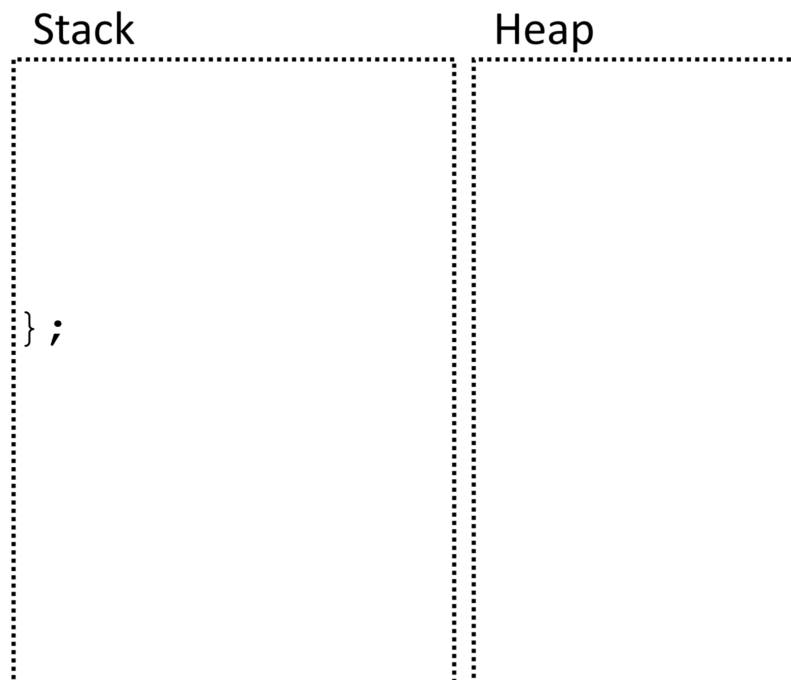
https://liu.se/nyhet/bud-lawson-artikel

https://en.wikipedia.org/wiki/Harold_Lawson

Pekare

- Vanlig namngiven variabel (här x).
- Håller reda på anonym variabel (adress).

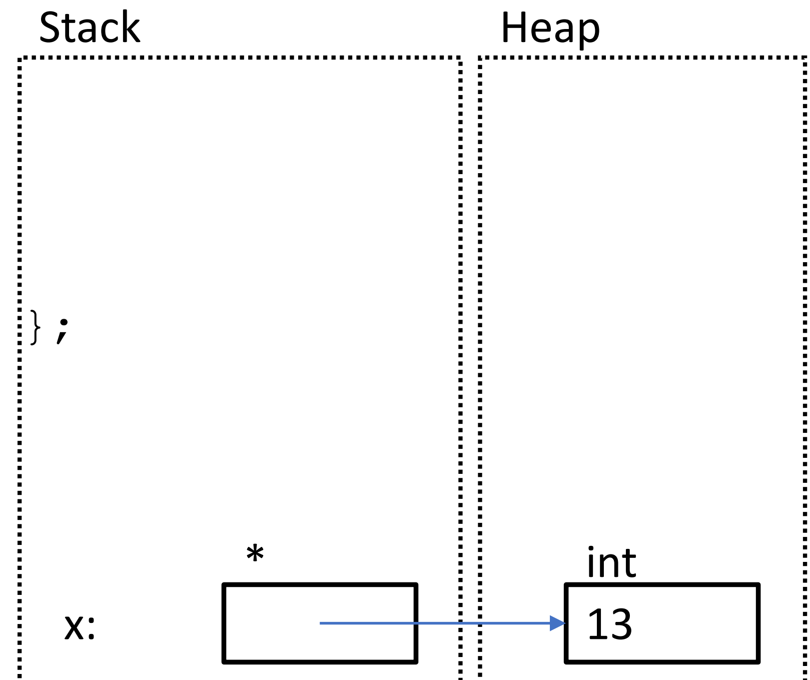
```
   int main()
   {
1:   int* x{ new int{13} };
2:   delete x;
3:   float* pi{ new float{3.14} };
4:   x = nullptr;
   }
```



Dynamiskt minne med new…

- ”new” skapar en anonym variabel på heapen
- Dess ”namn” (adressen) lagras i x

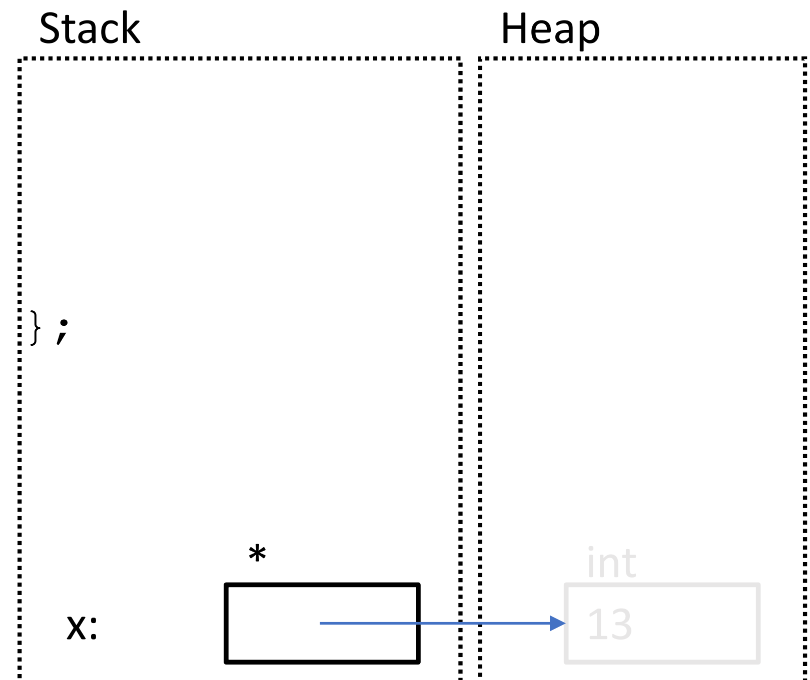
```
   int main()
   {
1:   int* x{ new int{13} };
2:   delete x;
3:   float* pi{ new float{3.14} };
4:   x = nullptr;
5:}
```



… och delete

- ”delete” tar bort en anonym variabel från heapen
- Adressen (”Namnet”) som ska bort hämtas från variabeln x

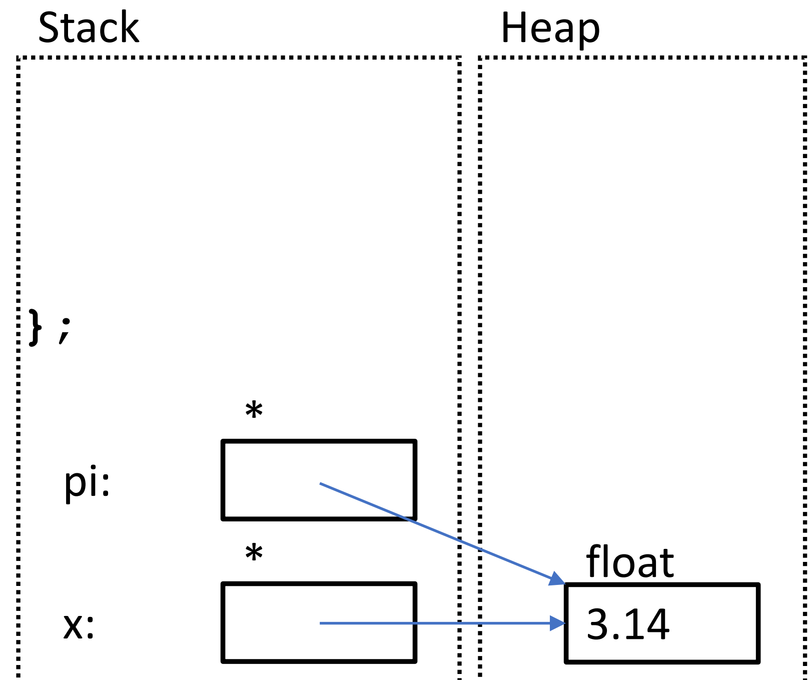
```
   int main()
   {
1:   int* x{ new int{13} };
2:   delete x;
3:   float* pi{ new float{3.14} };
4:   x = nullptr;
5:}
```



Ogiltiga pekare kan uppstå...

- Heapens minnesutrymme återanvänds.
- Vad pekar nu x på för heltalsvärde?

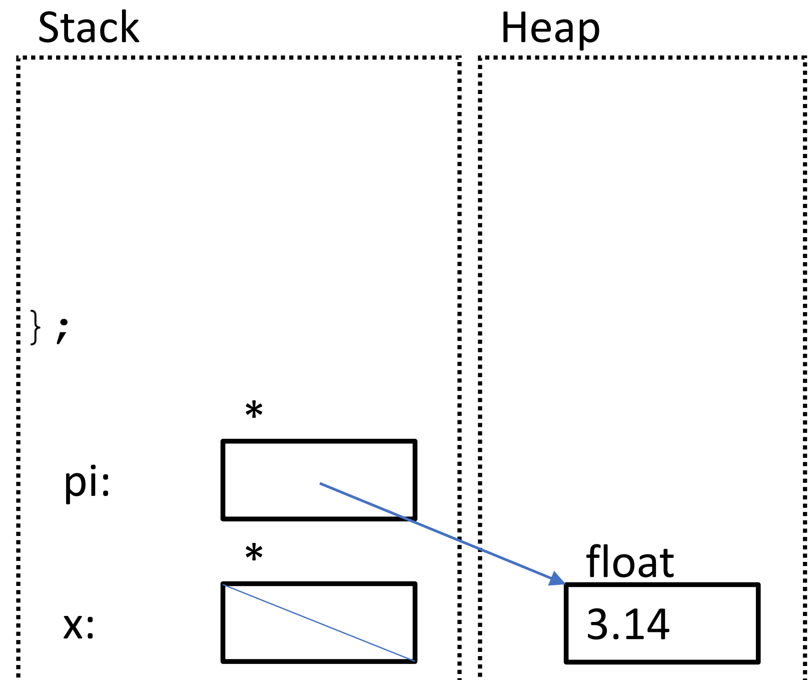
```
   int main()
   {
1:   int* x{ new int{13} };
2:   delete x;
3:   float* pi{ new float{3.14} };
4:   x = nullptr;
5:}
```



… och åtgärdas med nullptr

- ”nullptr” används när en pekarvariabel är ”tom”
- Då håller den inte reda på någon anonym variabel än (eller längre)

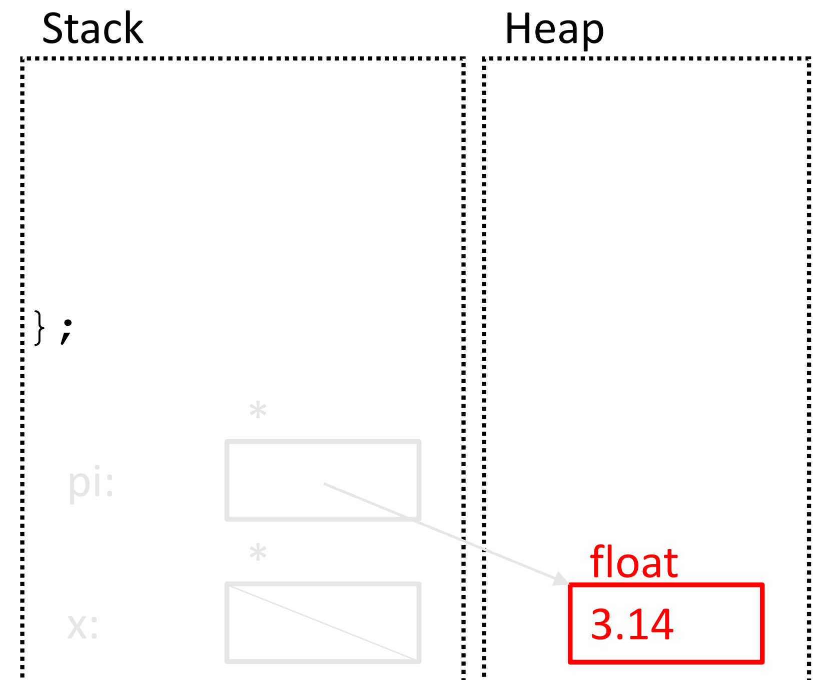
```
   int main()
   {
1:   int* x{ new int{13} };
2:   delete x;
3:   float* pi{ new float{3.14} };
4:   x = nullptr;
5:}
```



Minnesläcka

- Nu har vi tappat bort ”namnet” på den anonyma variabeln med värdet 3.14 och den tas aldrig bort.
- Detta kallas minnesläcka.

```
   int main()
   {
1:   int* x{ new int{13} };
2:   delete x;
3:   float* pi{ new float{3.14} };
4:   x = nullptr;
5: }
```



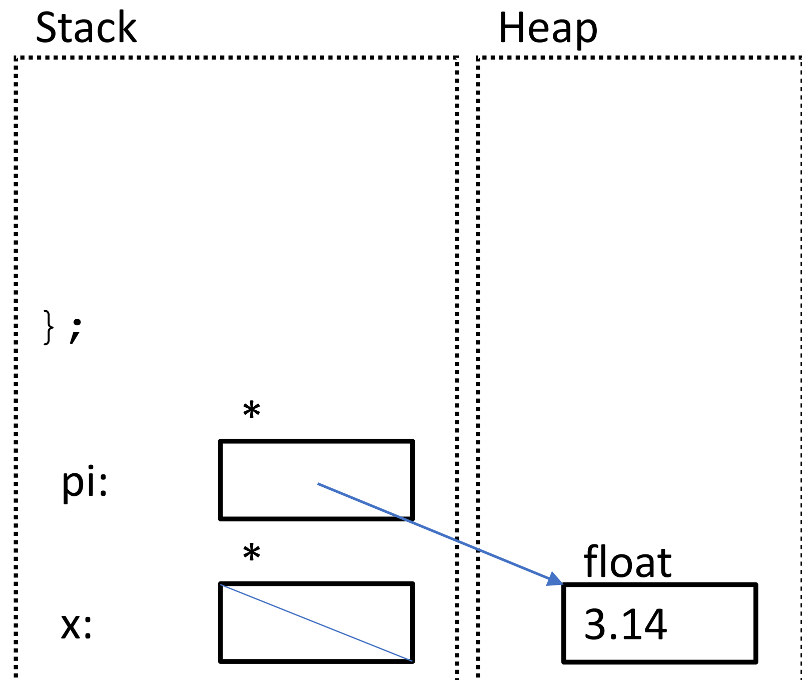
Åtkomst av anonym variabel

- Åtkomst är inte automatisk, vi måste avreferera med (operator* eller operator->)
- Operator*
  - ”Gå till” (eller ”namnge”) den anonyma variabeln
- Operator->
  - ”Gå till” (eller ”namnge”) en datamedlem i den anonyma variabeln
  - Ekvivalent med användning av (*) följt av .
  - (-> anropas rekursivt på returvärdet vid överlagring)

Avreferering

- Skriver ut 3.14

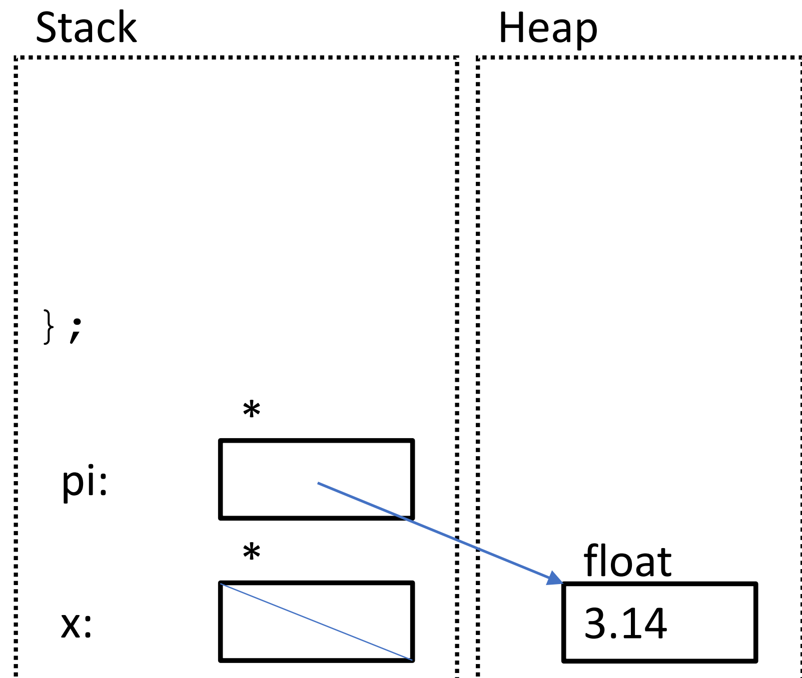
```
int main()
{
  int* x{ new int{13} };
  delete x;
  float* pi{ new float{3.14} };
  x = nullptr;
  cout << *pi << endl;
  cout << *x << endl;
}
```



Avreferering

- Programmet kraschar med ”segmentation fault” eftersom x inte pekar på något just nu

```
int main()
{
  int* x{ new int{13} };
  delete x;
  float* pi{ new float{3.14} };
  x = nullptr;
  cout << *pi << endl;
  cout << *x << endl;
}
```



Åtkomst av anonymt objekt

```
struct Node {
  int x;
  Node* n;
};
int main() {
  struct Node* top{ nullptr };
  top = new Node{21, top};
  top = new Node{34, top};
  cout << (*top).x << top->n->x << endl; // 3421
  *top = Node{89, nullptr}; // ??
};
```

Objektets självpekare

- När du implementerar medlemsfunktioner i din klass så har du automatiskt tillgång till pekare ”this” som innehåller adressen till aktuell instans (pekare till objektet funktionen anropades på).
- Men:
  - Du behöver inte använda ”this”. När du namnger en datamedlem ”x” är det implicit att du menar ”this->x”.
  - Om du har en lokal variabel eller parameter ”x” som skuggar din datamedlem ”x”, byt hellre namn på endera ”x” så det blir tydligare var som är vad.
- Undantag:
  - När du behöver en referens till objektet självt, antingen för att använda en operator, eller för att returnera, är det okej att använda *this.

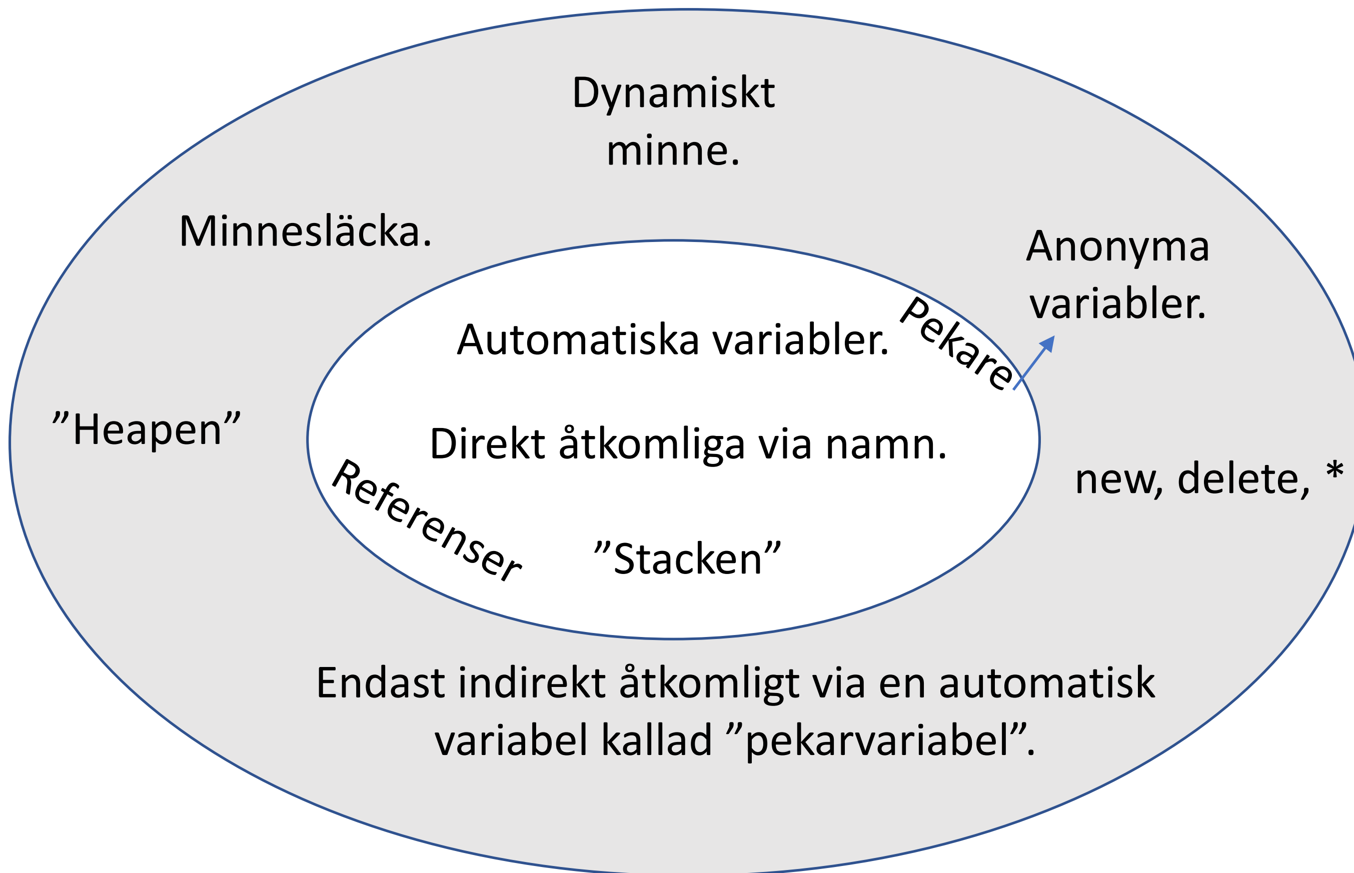
Adressoperatorn & (Undvik den!)

- Det går att ta fram adressen av vanliga automatiska variabler med operator&
- Men:
  - I 1:a hand: En automatisk variabel har ett namn, använd det.
  - I 2:a hand: Skapa en referens istället för att ta fram adressen.
  - I 3:e hand: tänk på att det är omöjligt att avgöra om en pekarvariabel hänvisar till en automatisk eller anonym variabel. Automatiska variabler får du inte köra ”delete” på. Anonyma variabler ska du köra ”delete” på. Blanda inte!
  - & är användbar: När du vill undersöka om ett objekt ligger på samma plats i minnet som ett annat objekt (dvs det är samma objekt)

Ansvar, användning, ägarskap

- Anonyma variabler
  - Finns kvar tills du tar bort dem.
  - Ska alltid tas bort så fort du inte behöver dem mer.
  - Ska aldrig tas bort flera gånger.
- Använd en tydlig ordning för anonyma variabler
  - Bestäm vem som ”äger” varje anonym variabel (ofta en klass)
  - Den som skapar(new) ansvarar för att ta bort(delete)
- Språkstöd finns
  - Destruktor (nästa föreläsning, ska användas i lab)
  - std::unique_ptr (eller liknande, används ej i lab)

Sammanfattning



4. Inre klasser

Inre klassen medlem till och kommer åt yttre klassen

Yttre klassen ej medlem av inre

Prova ange fullt scope om deklarationer inte fungerar

- `void Yttre::Inre::metod(void)`

Inre klass

- En klass definierad inuti en annan:

Outer.h

```
class Outer
{
public:
  Outer();

private:
  class In_Private
  {
  public:
    In_Private();
  };
};
```

Outer.cc

```
Outer::Outer()
{
  // can only access Outer members
}

Outer::In_Private::In_Private()
{
  // is a member of Outer
  // can access all Outer members
}
```

5. Objektorienterade tankar

Komposition

Aggregation

Privata hjälpklasser som implementationsdetalj kollegan inte ska känna till

Komposition och Aggregation

- Med ett objektorienterat tankesätt bygger vi programmet av moduler.
- En modul kan behöva andra moduler eller hjälpklasser för att lösa sin uppgift.
- Är när en klass har en eller flera datamedlem av klasstyp.
- Vi säger att klassen ”har” eller ”består av” en annan klass.
  - En Person har ett Hjärta
  - En Bil består av ett Chassi, en Motor och 4 Hjul

Komposition och Aggregation

- Begreppen är snarlika och gränsen ibland flytande.
- Komposition:
  - Beståndsdelen skapas med helheten och försvinner med helheten:
  - En Person har ett Hjärta
- Aggregation:
  - Beståndsdelen skapas och försvinner efter behov:
  - En Person har Kläder

Aggregat

- En klass eller struct som samlar ett antal datamedlemmar utan att använda inkapsling.

```
class Constants
{
public:
  double pi{3.14};
  double e{2.72};
}
struct Association
{
  string key;
  string value;
};
std::pair
std::tuple
```

Hjälpklasser

- Ibland behövs en hjälpklass för att bygga upp helheten.
- En hjälpklass är starkt knuten till sin huvudklass och är ofta inte meningsfull utan huvudklassen.
- I dessa lägen är det lämpligt att använda en inre klass.

Hjälpklasser och inkapsling

- Ofta ska användande kollega vare sig kunna pilla med eller behöva förstå den inre klassen.
  - Gör inre klassen privat i huvudklassen.
  - Se noga till att inre klassen inte syns bland något som är publikt.
  - Eftersom hela inre klassen är privat behöver vi inte vara så noga med inre klassens inkapsling.
- När kollegan ***ska*** kunna använda hjälpklassen (t.ex. en Iterator)
  - Gör inre klassen publik i huvudklassen.
  - Var noga med även den inre klassens inkapsling. Till och med konstruktorn ska vara privat om du vill ha kontroll på att endast huvudklassen (du) kan skapa hjälpklassobjekt.

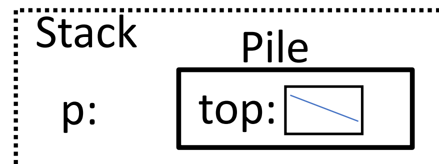
6. Exempel

Vi bygger en länkad stapel

En stapel

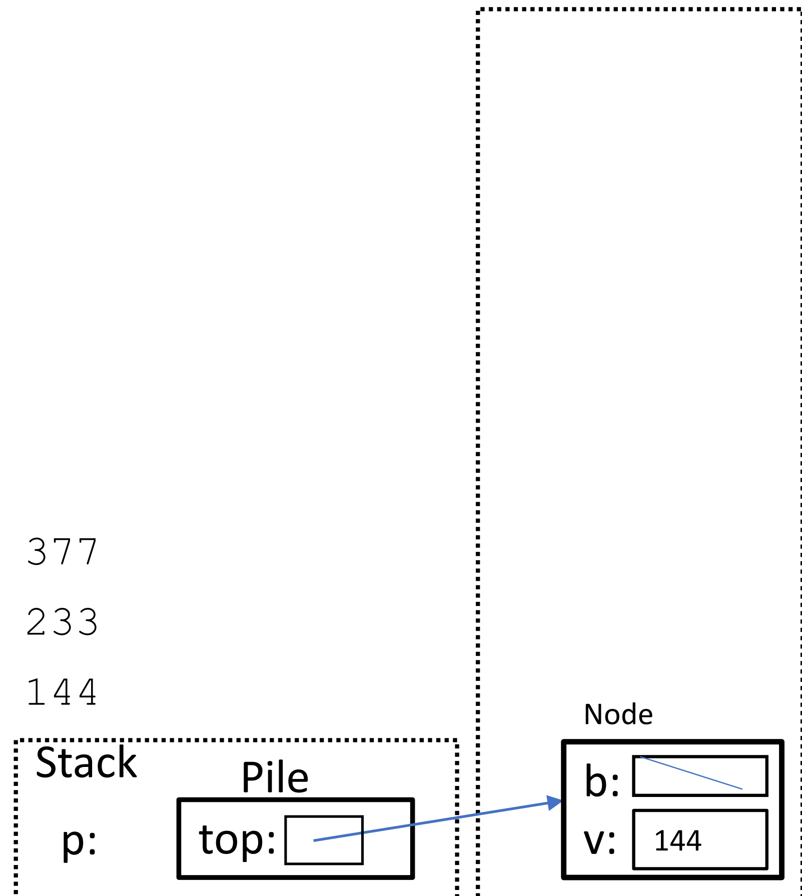
Heap

```
  int main() {
1:  Pile p;

2:  p.push(144);
3:  p.push(233);
4:  p.push(377);

5:  cout << p.pop() << endl; // 377
6:  cout << p.pop() << endl; // 233
7:  cout << p.pop() << endl; // 144
  }
```



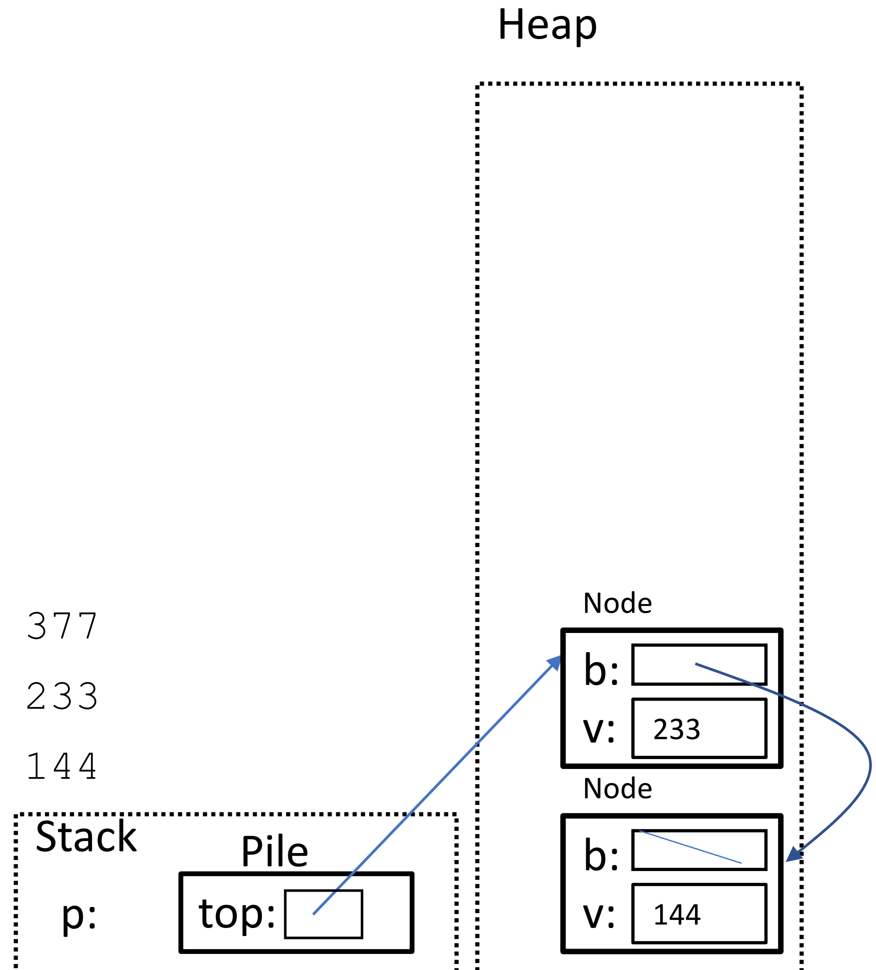
En stapel

```
  int main() {
1:  Pile p;

2:  p.push(144);
3:  p.push(233);
4:  p.push(377);

5:  cout << p.pop() << endl; // 377
6:  cout << p.pop() << endl; // 233
7:  cout << p.pop() << endl; // 144
  }
```



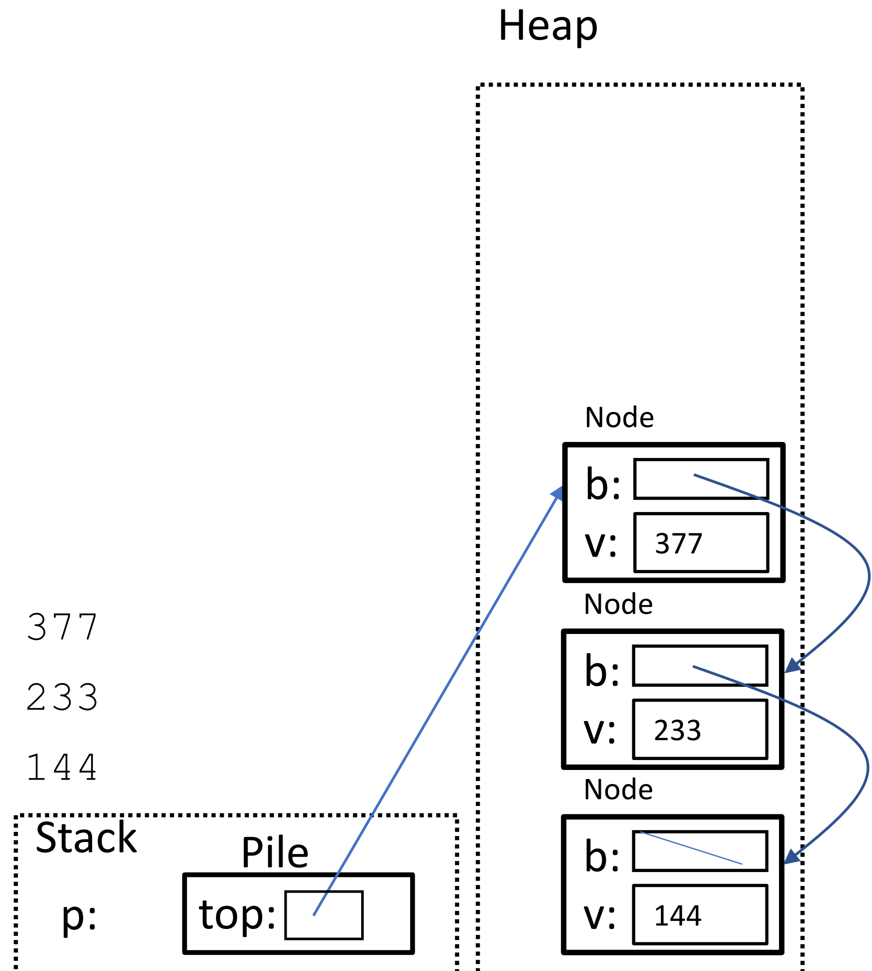
En stapel

```
  int main() {
1:  Pile p;

2:  p.push(144);
3:  p.push(233);
4:  p.push(377);

5:  cout << p.pop() << endl; // 377
6:  cout << p.pop() << endl; // 233
7:  cout << p.pop() << endl; // 144
  }
```



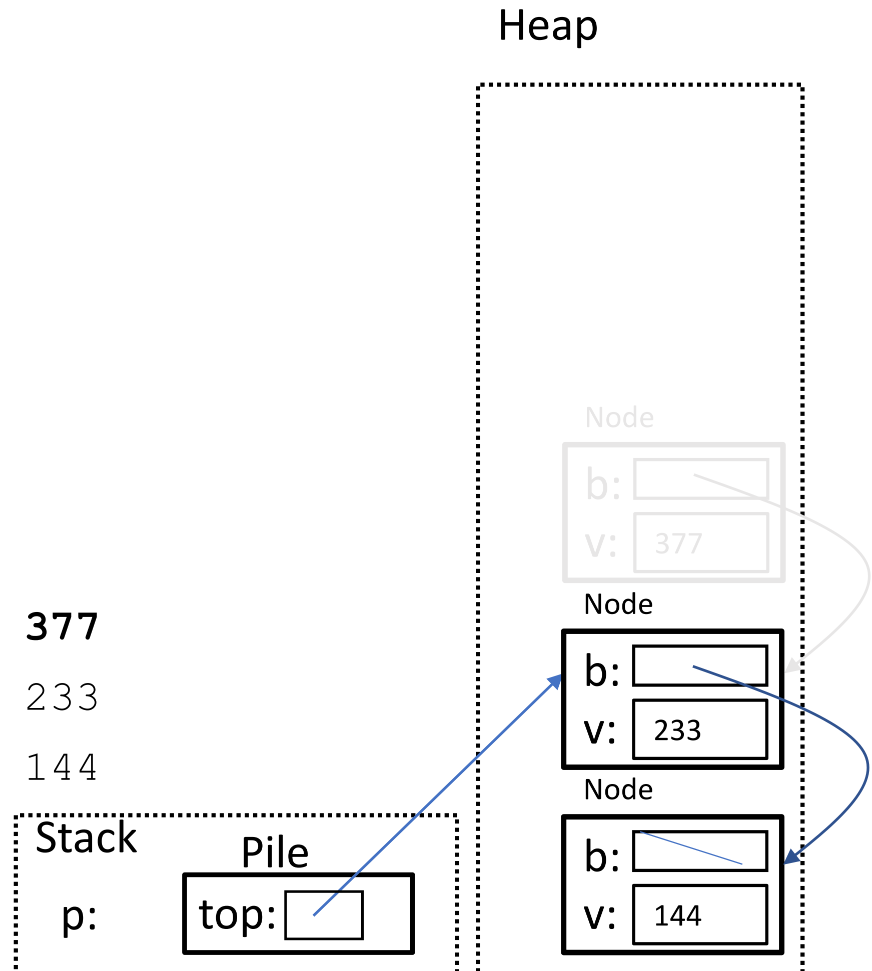
En stapel

```
  int main() {
1:   Pile p;

2:   p.push(144);
3:   p.push(233);
4:   p.push(377);

5:   cout << p.pop() << endl; // 377
6:   cout << p.pop() << endl; // 233
7:   cout << p.pop() << endl; // 144
  }
```



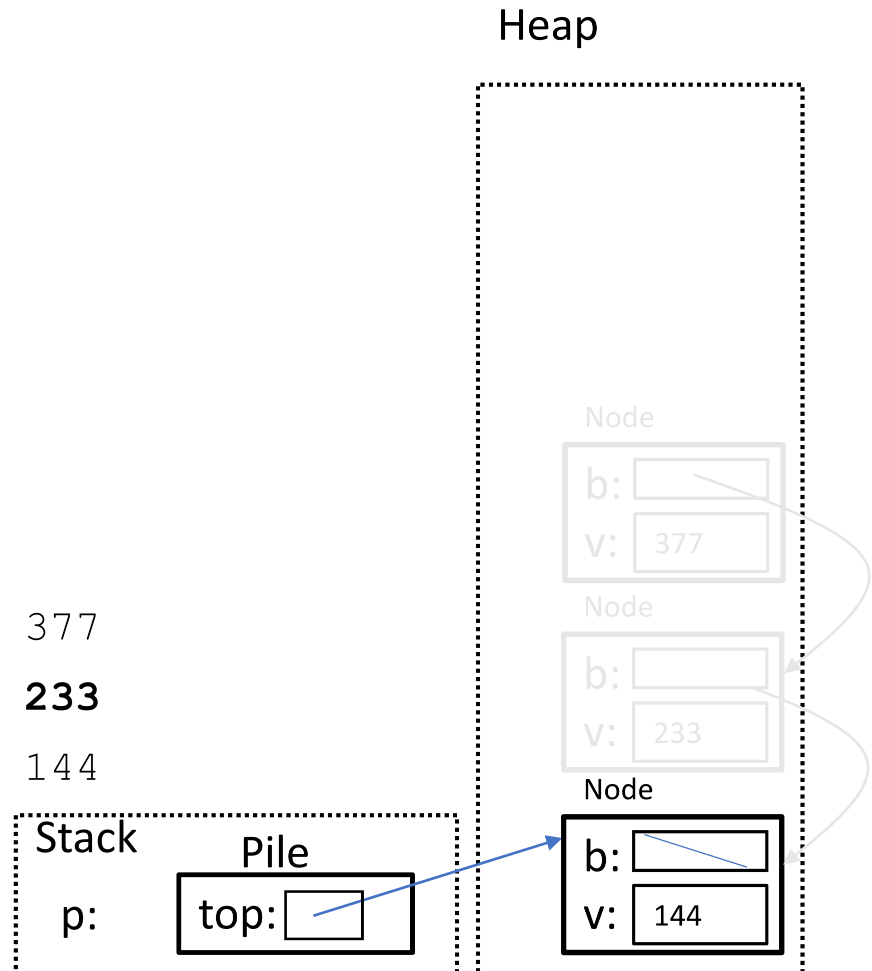
En stapel

```
  int main() {
1:  Pile p;

2:  p.push(144);
3:  p.push(233);
4:  p.push(377);

5:  cout << p.pop() << endl; // 377
6:  cout << p.pop() << endl; // 233
7:  cout << p.pop() << endl; // 144
  }
```



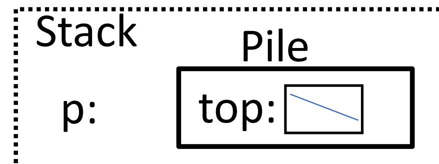
En stapel

```
  int main() {
1:  Pile p;

2:  p.push(144);
3:  p.push(233);
4:  p.push(377);

5:  cout << p.pop() << endl; // 377
6:  cout << p.pop() << endl; // 233
7:  cout << p.pop() << endl; // 144
  }
```

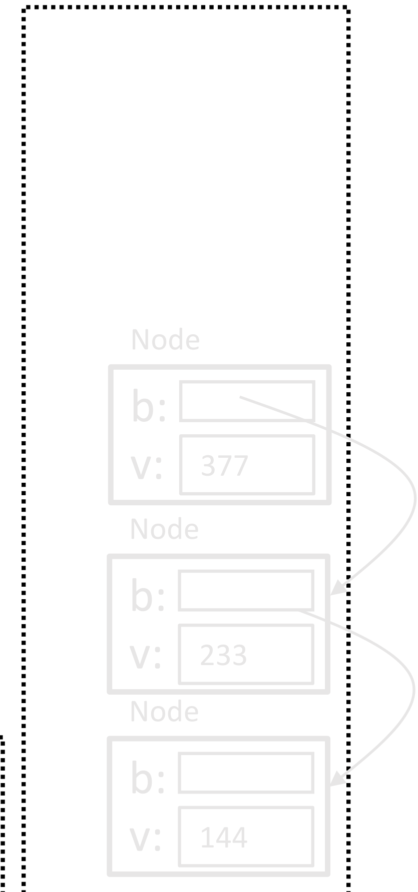


En stapel

```
  int main() {
1:  Pile p;

2:  p.push(144);
3:  p.push(233);
4:  p.push(377);

5:  cout << p.pop() << endl; // 377
6:  cout << p.pop() << endl; // 233
7:  cout << p.pop() << endl; // 144
  }
```



Heap



En stapel

```
class Pile
{
public:
  Pile();
  void push(int i);
  int pop();
private:
  struct Node
  {
    int value;
    Node* below;
  };
  Node* top;
};
```

En stapel

```
Pile::Pile() : top{nullptr}
{}

void Pile::push(int i)
{
  top = new Node{i, top};
}
```

```
// ordningen är viktig!
int Pile::pop()
{
  int v{ top->value };
  Node* removed{ top };
  top = top->below;
  delete removed;
  return v;
}
```

7. Speciella medlemsfunktioner

Konstruktor (känner vi redan till)

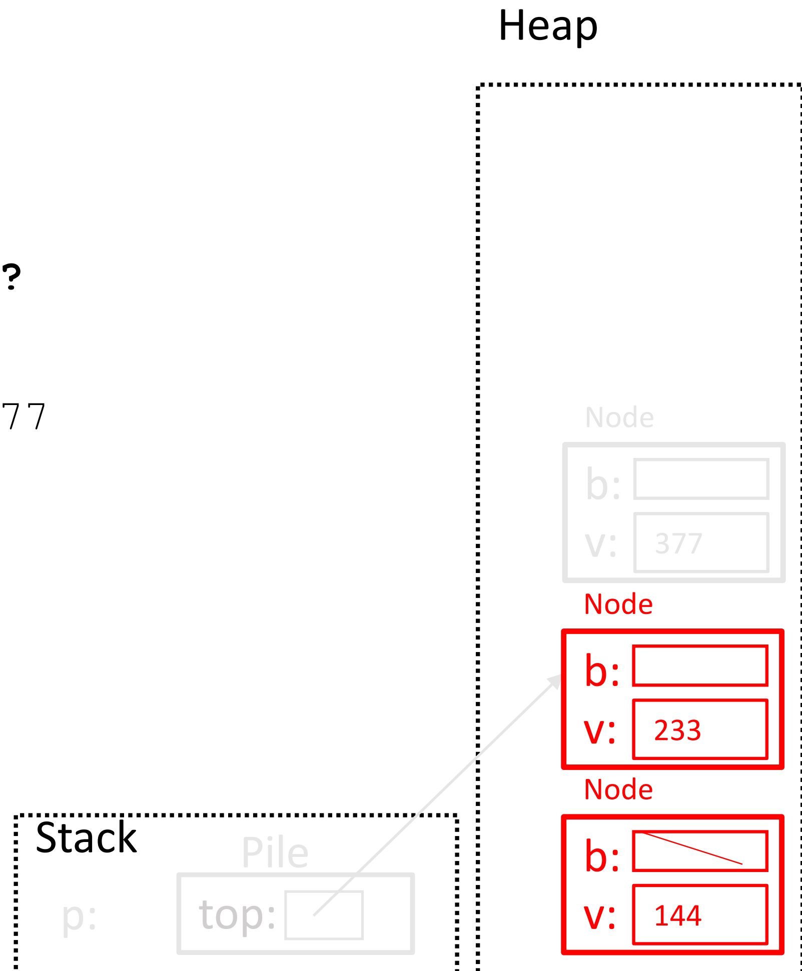
`std::initializer_list`

Destruktor (motsatsen till konstruktor)

`= default`

En stapel

```
int main()
{
  Pile p{144, 233, 377}; // Hur?

  cout << p.pop() << endl; // 377

} // Minnesläcka!
```



std::initializer_list

- Inkludera <initializer_list>
- https://en.cppreference.com/w/cpp/utility/initializer_list
- Skapas när du skriver en lista inom klamrar i din kod:

```
{5, 8, 13, 21, 34}
```

- Kan stegas igenom med ”range-based-for”:

```
for ( auto v : list )
{
  // use item v
}
```

Konstruktor för brace-enclosed-list

```
#include <initializer_list>

Pile::Pile(std::initializer_list<int> const& l)
 : top{nullptr}
{
  for ( int i : l )
  {
    push(i);
  }
}
```

Destruktor

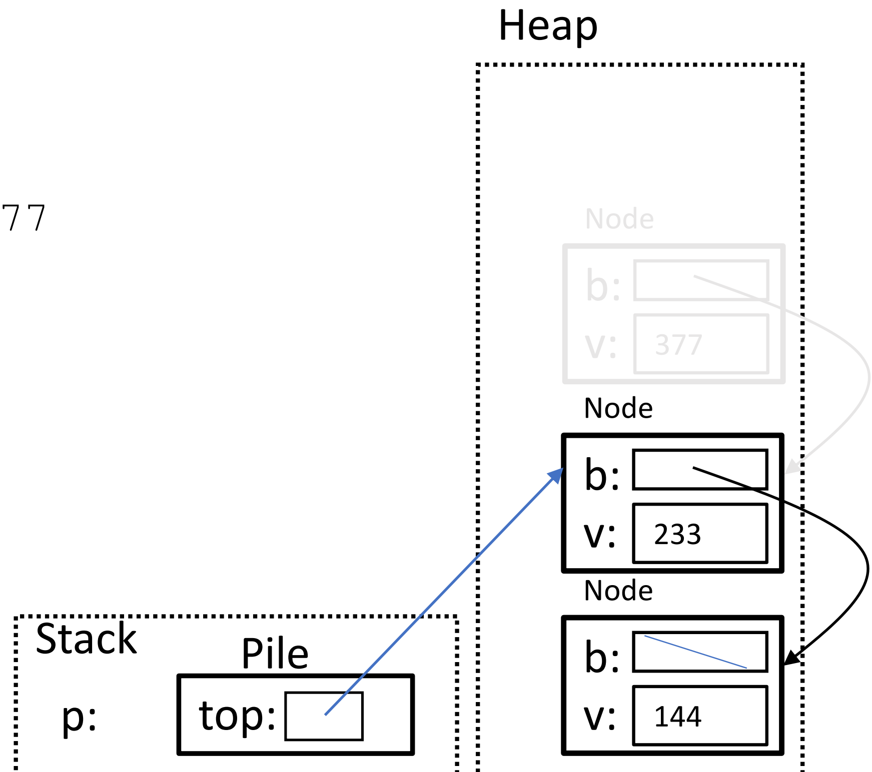
- Samma namn som klassen med prefixet ~
- Saknar returvärde
- Anropas aldrig explicit (dvs inte av dig)
- Anropas alltid automatiskt när objekt avförs från exekveringsstacken *eller* minnesheapen
  - Stack: vid `return` eller `}` eller `throw`
  - Heap: vid `delete`
- Låter dig utföra extra instruktioner precis innan dina objekt försvinner

Destruktor för stapeln

```
Pile::~Pile()
{
  // while nodes remaining on top
  {
     // save pointer to top node
     // remove top node from pile
     // delete memory allocated for top node
  }
}
```

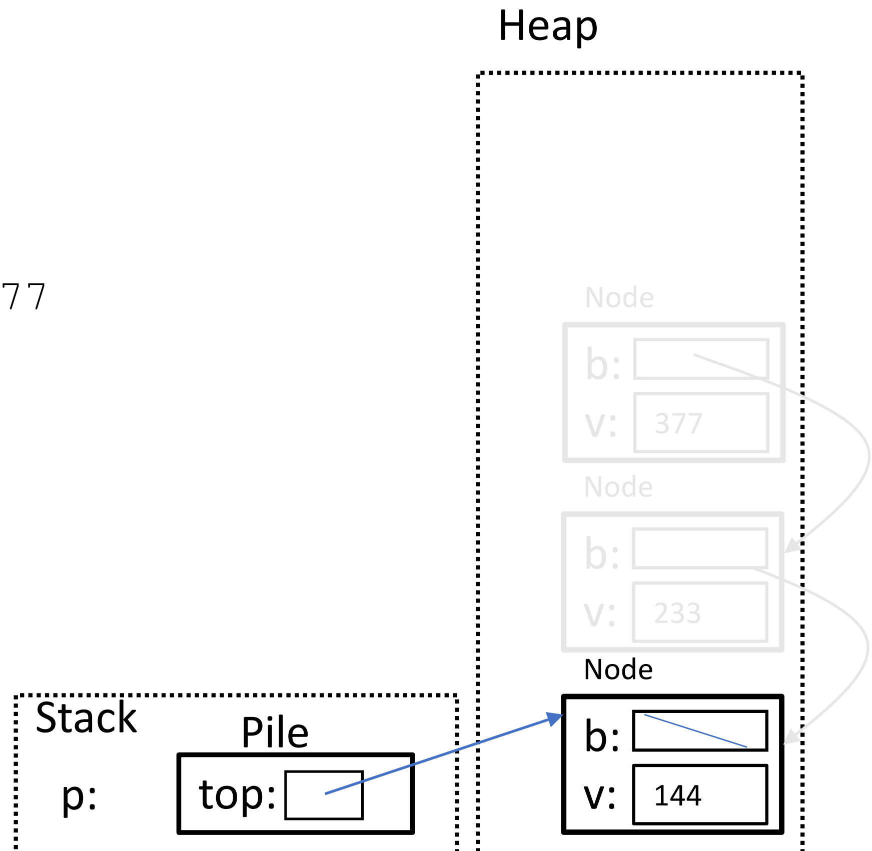
Destruering (utgångsläge)

```
int main()
{
  Pile p{144, 233, 377};

  cout << p.pop() << endl; // 377

} // Destruktor!
```



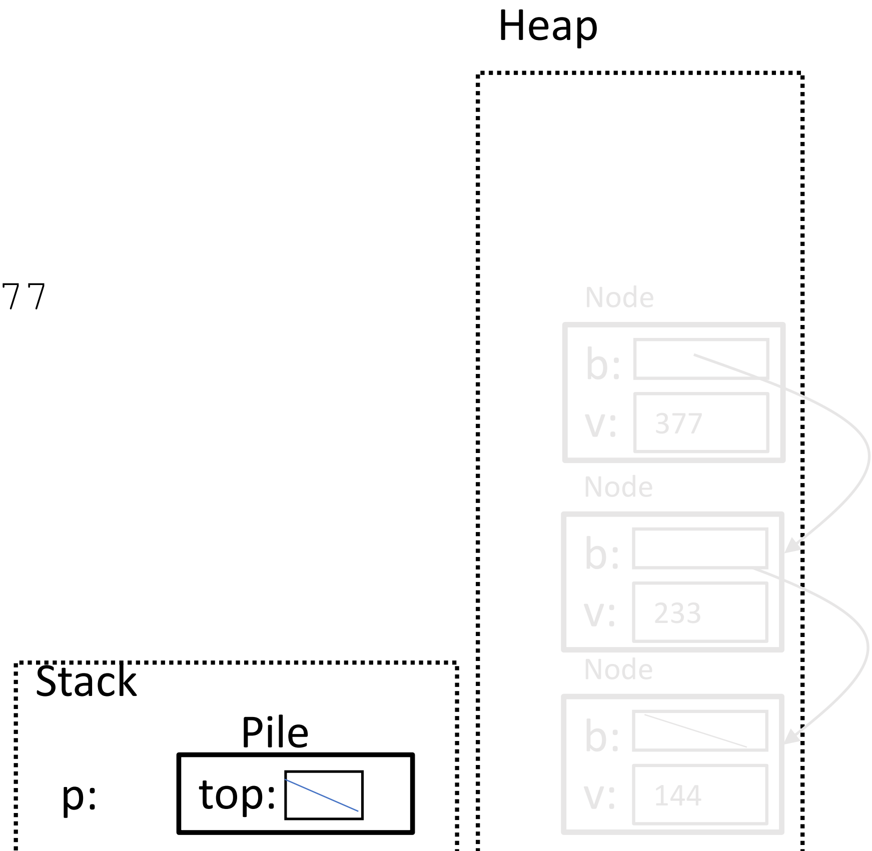
Destruering (halvvägs)

```
int main()
{
  Pile p{144, 233, 377};

  cout << p.pop() << endl; // 377

} // Destruktor!
```



Destruering (klar)

```
int main()
{
  Pile p{144, 233, 377};

  cout << p.pop() << endl; // 377

} // Destruktor!
```



Alternativ destruktor för stapeln

```
Pile::Node::~Node()
{
  // delete memory allocated for below node
}
Pile::pop()
{
  // revisit to fix bug caused by above destructor
}
Pile::~Pile()
{
  // delete memory allocated for top node
}
```

Repetition: Ansvar, användning, ägarskap

- Anonyma variabler
  - Finns kvar tills du tar bort dem.
  - Ska alltid tas bort så fort du inte behöver dem mer.
  - Ska aldrig tas bort flera gånger.
- Använd en tydlig ordning för anonyma variabler
  - Bestäm vem som ”äger” varje anonym variabel (ofta en klass)
  - Den som skapar(new) ansvarar för att ta bort(delete)
- Språkstöd finns
  - Destruktor (ska användas i lab)
  - std::unique_ptr (eller liknande, används ej i lab)

Ägarskap och ansvar i vår stapel

- I `Pile` har vi en pekare `top`
- I `Node` har vi en pekare `below`
- Alternativ 1, klassen Pile ansvarar
  - `Pile::push` är den som gör `new`
  - `Pile::pop` är den som gör `delete`
  - `Pile::~Pile` är den som tömmer hela stapeln
- Alternativ 2, varje instans ansvarar för sin pekare
  - `Pile` ansvarar bara för `top`
  - `Pile::~Pile` tar bort `top` (och då körs `Node::~Node`)
  - `Node` ansvarar för `below`
  - `Node::~Node` tar bort `below` (rekursivt)

Standard{kon,de}struktor

- Om du inte deklarerar någon konstruktor så kommer kompilatorn att definiera en **tom** åt dig.
- Om du inte deklarerar någon konstruktor så kommer kompilatorn att definiera en **tom** åt dig.
- Du kan explicit ange att du vill ha kompilatorns **tomma** standard{kon,de}struktor med ” = default”

```
class X {
public:
  X() = default;
 ~X() = default;
};
```

8. Speciella medlemsfunktioner

Kopieringskonstruktor (hjälp för dig att skapa djupa kopior)

Kopieringstilldelning (kopiera och byt plats)

`= delete`

Flyttkonstruktor (hjälp för dig att undvika onödiga kopior)

Flytttilldeling (bara byt plats)

Rule of [0..6[

Språkautomatik

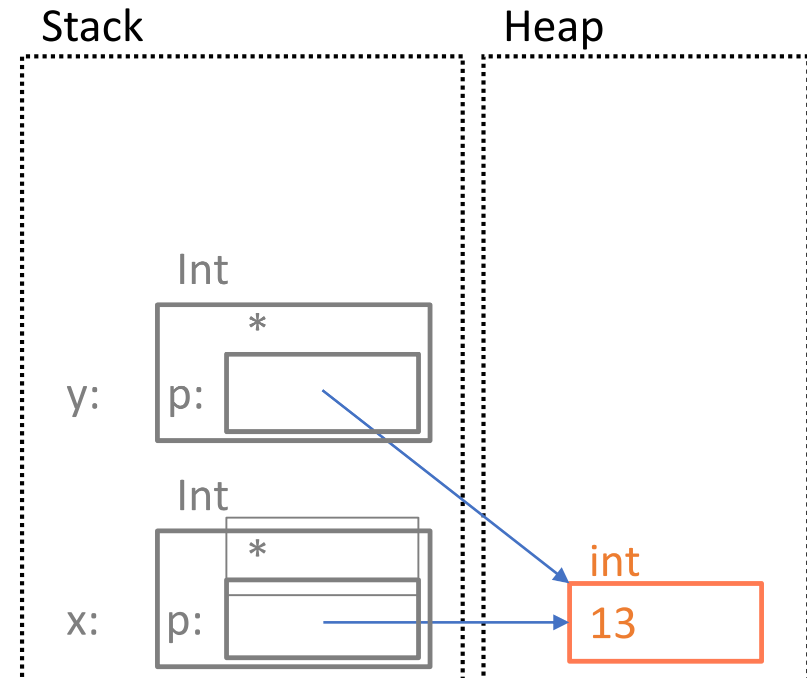
- Kompilatorn gör ett ***försök*** att definiera hur dina klassobjekt kopieras tills du själv deklarerar antingen kopieringskonstruktor eller tilldelningsoperator
- Standard metod för tilldelning, parameteröverföring och returvärde i C++ är **kopiering**. Metoderna för kopiering anropas **ofta**.
- Kompilatorn är inte oändligt smart, den vet inte om anonyma variabler och kan inte kopiera dem korrekt (de finns ju inte förrän programmet körs!)
- Använder du anonyma variabler måste du själv implementera hur kopiering ska gå till.

Språkautomatik, grund kopia

- Vad händer om ”Int” är en klass och kompilatorns standard-destruktorer körs?

```
int main()
{
  Int x{ new int{13} };
  Int y{x}; // kopiering
} // destruktorer!
```

```
// tom för y

// tom för x
```



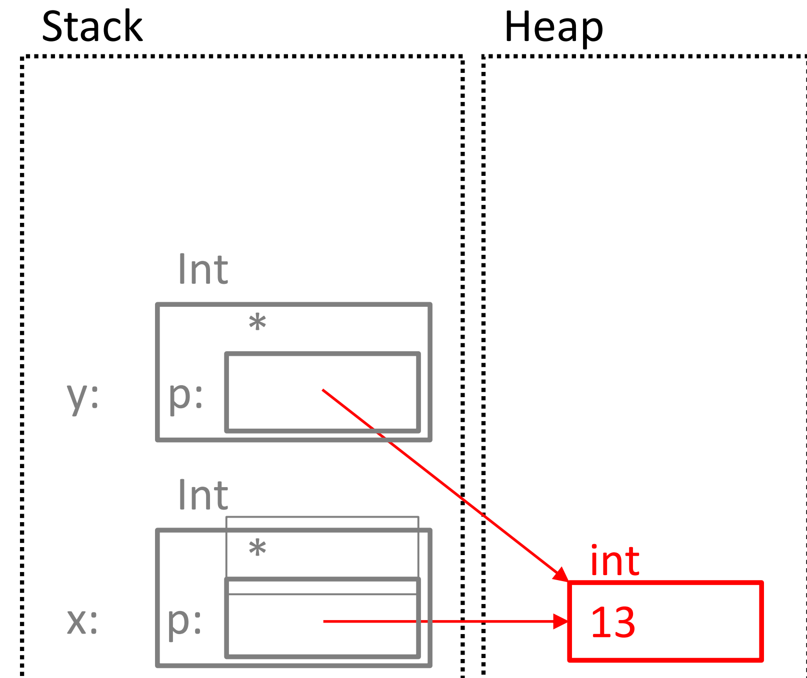
Språkautomatik, grund kopia

- Vad händer om ”Int” är en klass och ”delete” körs *på rätt sätt* i destruktorn?

```
int main()
{
  Int x{ new int{13} };
  Int y{x}; // kopiering
} // destruktorer!
```

delete y.p

delete x.p



Programmerare, djup kopia

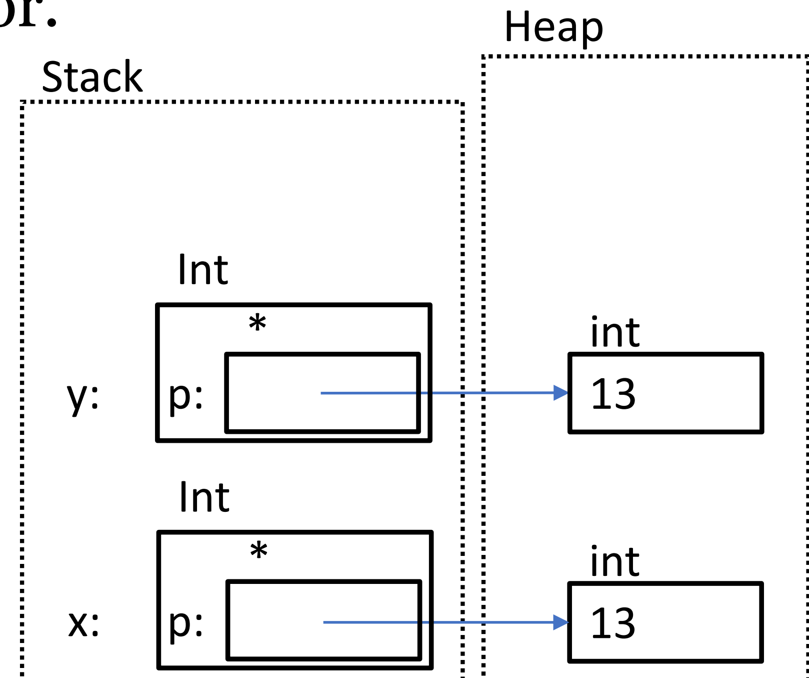
- Som programmerare måste du se till att kopiera hela *strukturen* av anonyma variabler, annars *kan du inte* skriva en korrekt destruktor.

```
int main()
{
  Int x{new int{13}};
  Int y{x}; // djup kopia
} // destruktorer!
```

new int{x.value()}

delete y.p

delete x.p



Mål: alltid unika instanser

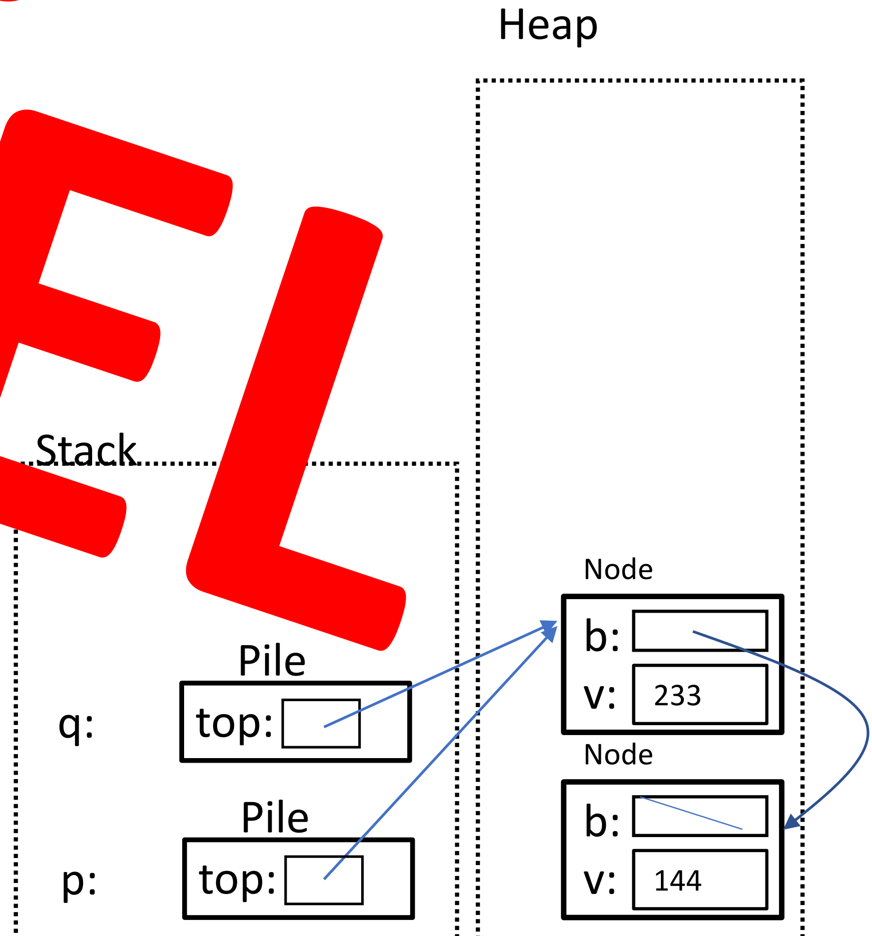
- Som programmerare måste du se till att alla instanser som skapas av din klass är unika. En instans ska aldrig peka ut minne som ägs av en annan instans.
  - Du kan bygga in extra logik i din klass för att hantera just situationen att instanser kan dela minne, men det är inte en del av denna kurs.
- Destruktorn ska alltid vara trygg i att den bara tar bort saker som hör till ”sin” instans (som går ur scope eller ”försvinner”).
- För att vi ska kunna uppnå målet måste vi kunna kontrollera hur objekt kopieras (det sker normalt implicit och helt automatiskt).
- Vi behöver mer språkautomatik för att språkautomatiken ska fungera. C++ ger oss ”hooks” för att köra kod vi skriver vid varje typ av kopiering som sker.
- **Kompilatorn känner inte till anonyma variabler och kan aldrig hantera(kopiera) dem automatiskt.**

Kopieringskonstruktor

- Kopiera befintligt objekt till nyskapat (tomt) objekt
- Din målsättning: Skapa djup kopia
- Deklaration:
  - `Pile::Pile(Pile const& p);`
- Anropas automatiskt:
  - Vid definition av nya variabler: `Pile p{other_pile};`
  - Fortfarande ny variabel: `Pile p = other_pile;`
  - När objekt returneras: `Pile fun();`
  - Vid parameteröverföring: `void fun(Pile p);`
- Notera här en viktig anledning till ”`const&`” parametrar!

Automatisk kopieringskonstruktor

```
int main()
{
  Pile p;
  p.push(144);
  p.push(233);

  Pile q{p};
}
```

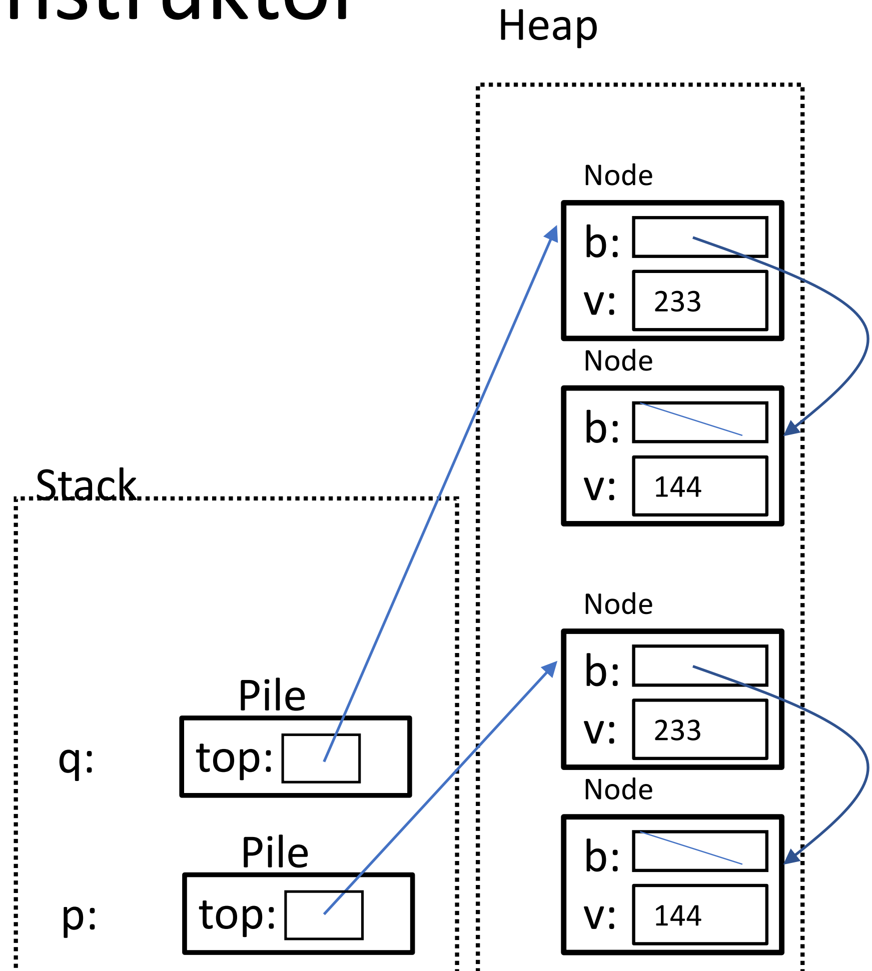


Fel resultat. Objekten p och q pekar ut samma stapel. De är inte självständiga oberoende objekt. Pop eller destruktor för den ena förstör den andra.

Korrekt kopieringskonstruktor

```
int main()
{
  Pile p;
  p.push(144);
  p.push(233);

  Pile q{p};
}
```

Rätt resultat. Objekten p och q har inget gemensamt. Hela strukturen med alla anonyma variabler är kopierad.

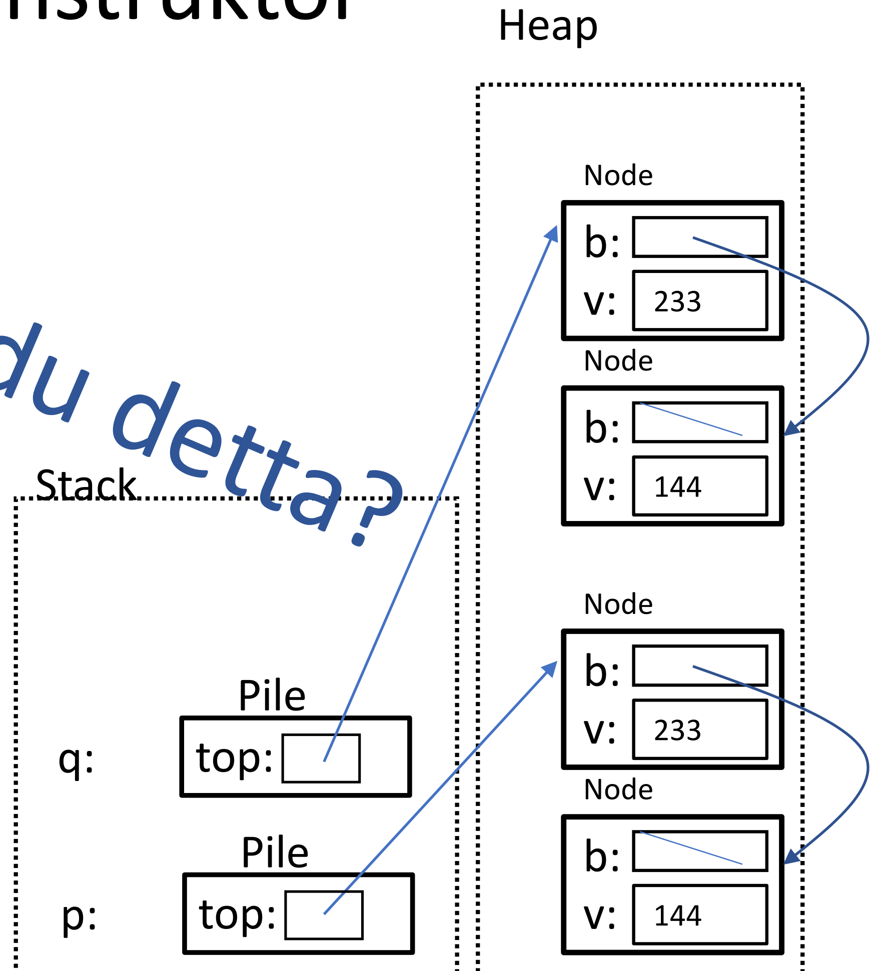


Korrekt kopieringskonstruktor

```
int main()
{
  Pile p;
  p.push(144);
  p.push(233);

  Pile q{p};
}
```

Hur testar du detta?

Rätt resultat. Objekten p och q har inget gemensamt. Hela strukturen med alla anonyma variabler är kopierad.



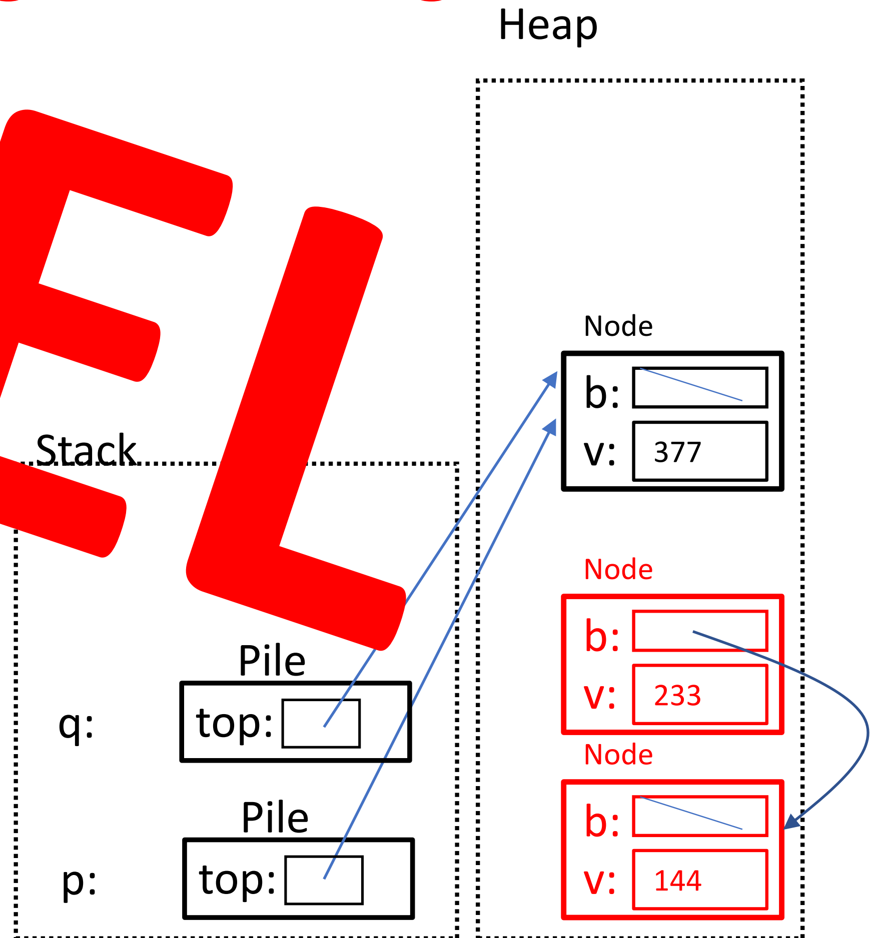
Kopieringstilldelning

- Kopiera befintligt objekt till *existerande* objekt
- Din målsättning: Skapa djup kopia
- Deklaration:
  - `Pile& Pile::operator=(Pile const& rhs);`
- Anropas:
  - Vid tilldelning: `some_pile = other_pile;`
  - OBS! Ofta sker tilldelning i samband med att ett nytt objekt skapas. Då körs istället kopieringskonstruktorn direkt.
- Idiom: Kopiera och byt plats
- Gardera mot självtilldelning

Automatisk kopieringstilldelning

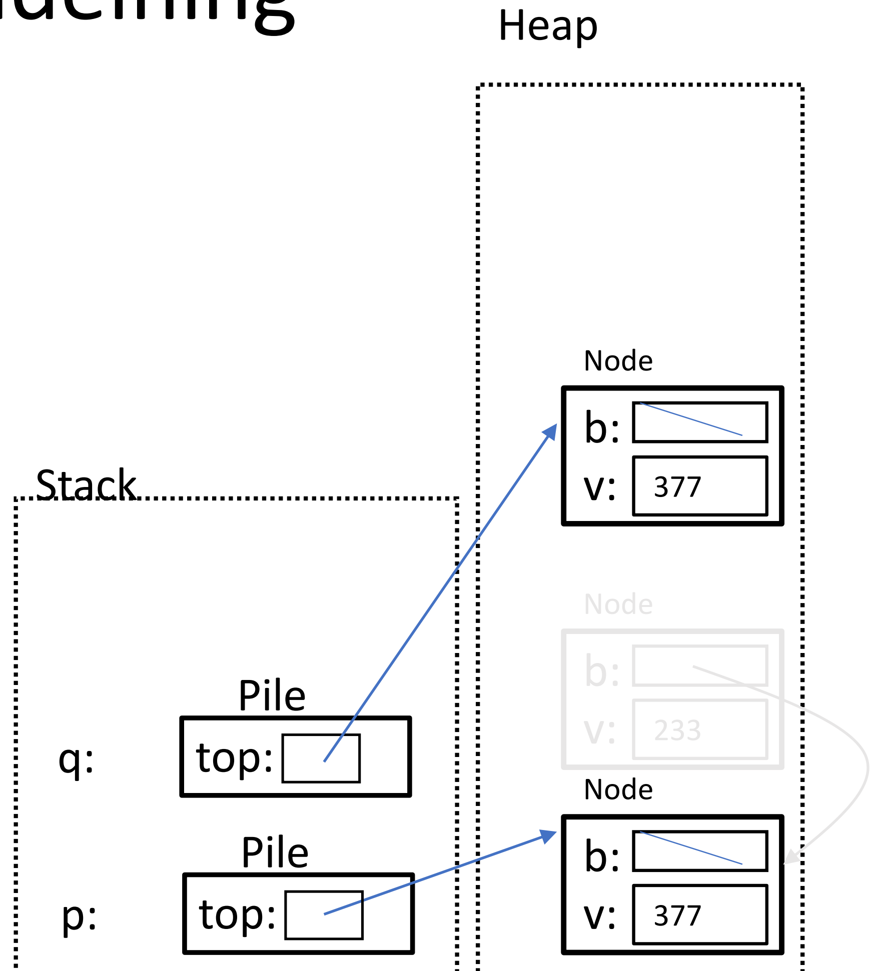
```
int main()
{
  Pile p{144, 233};
  Pile q{377};
  p = q;
}
```

Fel resultat. Objekten p och q pekar ut samma stapel. De är inte självständiga oberoende objekt. Pop eller destruktor för den ena förstör den andra. Dessutom tappade vi bott de anonyma variabler q höll reda på. Minnesläcka.



Korrekt kopieringstilldelning

```
int main()
{
  Pile p{144, 233};
  Pile q{377};
  p = q;
}
```



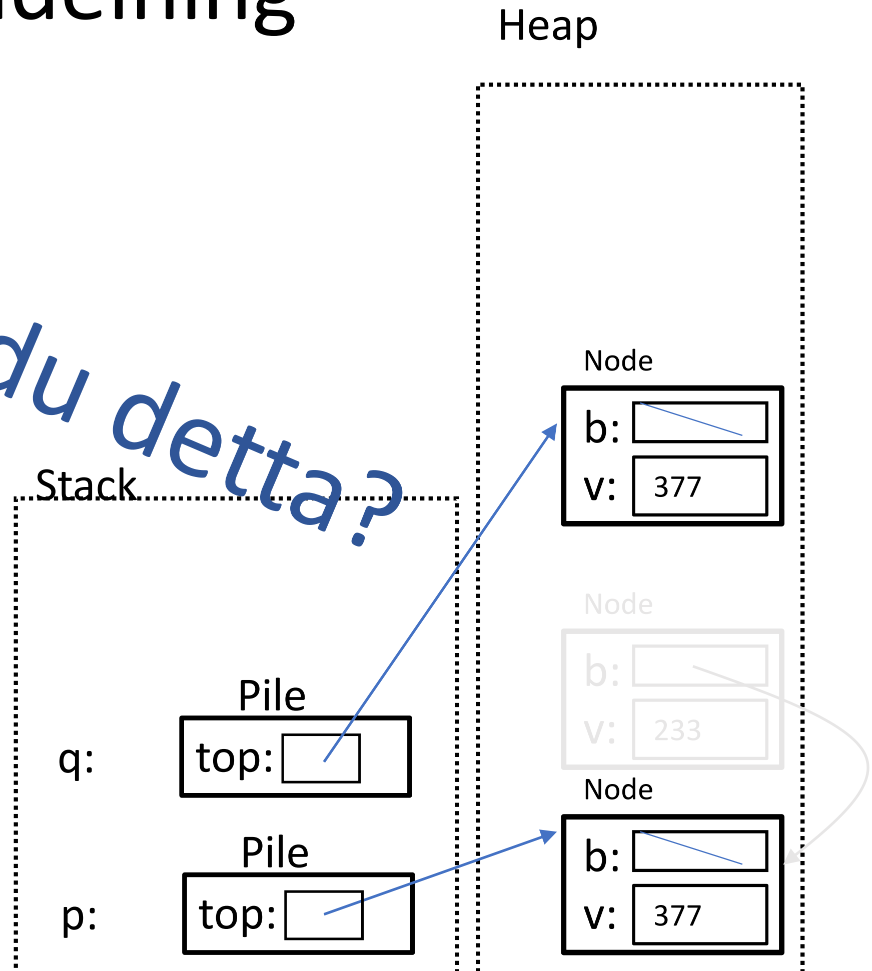
Rätt resultat. Objekten p och q har inget gemensamt. Hela strukturen med alla anonyma variabler är kopierad. Gamla innehållet i p är borttaget.

Korrekt kopieringstilldelning

```
int main()
{
  Pile p{144, 233};
  Pile q{377};
  p = q;
}
```

Hur testar du detta?

Rätt resultat. Objekten p och q har inget gemensamt. Hela strukturen med alla anonyma variabler är kopierad. Gamla innehållet i p är borttaget.



Gardera mot självtilldelning

```
Klasstyp& Klasstyp::operator=(Klasstyp const& rhs) {
  // undersök om objekten ligger på samma adress
  if ( this != &rhs ) {
    …
  }
  return *this;
}
Pile& select_highest(Pile& p, Pile& q) { if ( … ) return p; else … }
int main() {
  Pile a{21, 34};
  Pile b{55};
  a = select_highest(a, b); // a = a;
}
```

Idiom: Kopiera och byt plats

```
Klasstyp& Klasstyp::operator=(Klasstyp const& rhs)
{
  Klasstyp djup_kopia{rhs}; // Kopieringskonstruktorn
  swap(datamedlem1, djup_kopia.datamedlem1);
  swap(datamedlem2, djup_kopia.datamedlem2);
  …
  swap(datamedlemN, djup_kopia.datamedlemN);
  return *this; // Destruktor för djup_kopia
}
```

Idiomet fungerar korrekt även vid självtilldelning!

Hur ofta skapas kopior? (i onödan?)

```
Pile select(Pile a, Pile b) {
  if ( a > b )
    return a;
  else
    return b;
}
int main() {
  Pile p{1,1,2,3,5};
  Pile q{8,13,21};
  p = select(p, Pile{34,55}) + q;
}
```

Hemövning!
Hur tar du reda på när kopior skapas?
Hur många kopieringar sker här?
Hur många är ”onödiga” eftersom det som kopieras ändå inte behövs mer?

Vill du titta i min C++-bok?

- C++ approacher:
  - **Kopiera boken, ge kopian till dig (standard)**
  - Låna ut boken (referens)
  - Lägg boken på utsatt plats, berätta var (pekare)

- Föreställ dig nu att vi vet att jag är på väg att kasta min bok för att jag inte längre behöver den.
  - Kan du komma på något sätt att förenkla standardapproachen när du har denna information?

Ibland är det lättare att flytta

```
Car grandpa_life() { return Mustang; }
Car your_car = Toyota;
your_car = grandpa_life();
```

- C++ standardapproach:
  - Bygg en replika av Mustangen (djup kopiering i operator=)
  - Kör Toyotan till skroten (destruktor)
  - Kör original-Mustangen till skroten (destruktor)
- *Effektivare att bara byta ägare i bilregistret!*

När är det lättare att flytta?

- När ett *tillfälligt* objekt ska kopieras.
    1. Vi utnyttjar att objektet är tillfälligt genom att plocka ut det vi vill ha kvar istället för att kopiera det. Har vi skräp vi inte vill ha kvar kan vi stoppa in skräpet i det tillfälliga objektet.
    2. Det tillfälliga objektets destruktor körs.
- *Kompilatorn identifierar tillfälliga objekt helt automatiskt och anropar våra flyttfunktioner när det går. Vi behöver bara implementera dem!*

```
Pile p = pile_a + pile_b;
```

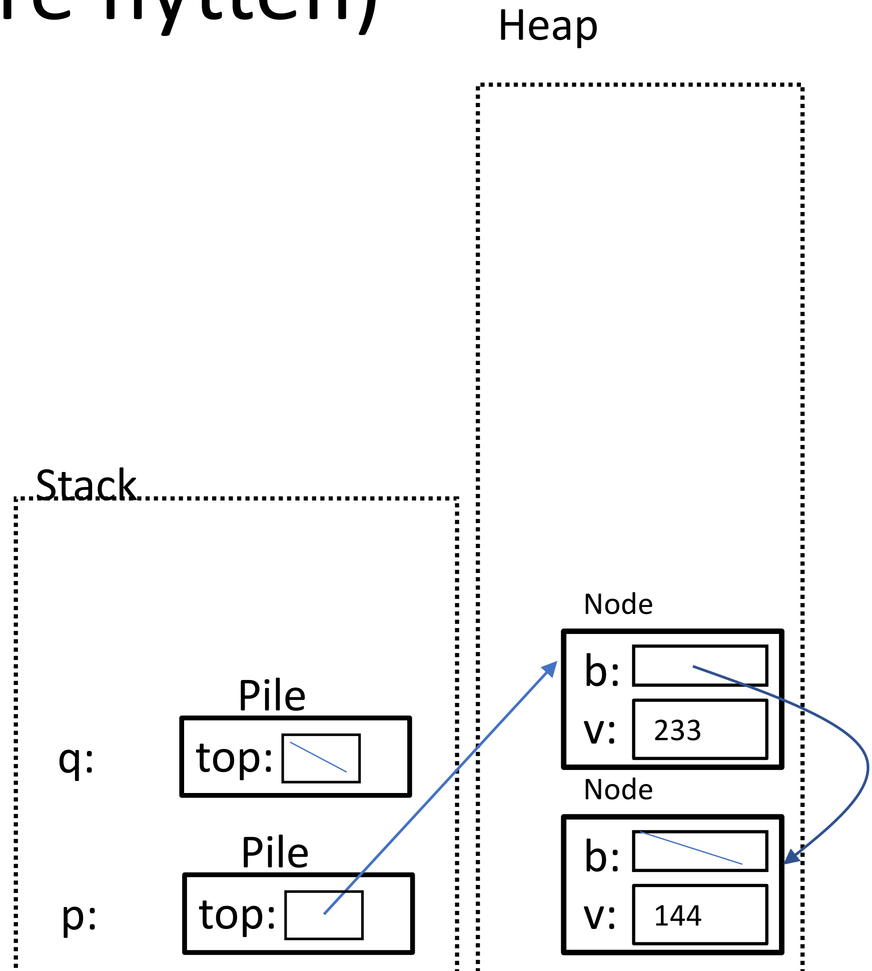
Flyttkonstruktor

- Flytta tillfälligt objekt till nyskapat (tomt) objekt
- Det tillfälliga objektet kommer inte användas mer
- Byt plats!
- Deklaration:
  - `Pile::Pile(Pile && p);`
- Anropas:
  - Vid definition av nya variabler och parameteröverföring där högerled eller argument är tillfälliga temporärobjekt.
  - Kompilatorn gör rätt anrop helt automatiskt.

Flyttkonstruktion (före flytten)

```
int main()
{
  Pile p;
  p.push(144);
  p.push(233);

  Pile q{std::move(p)};
}
```

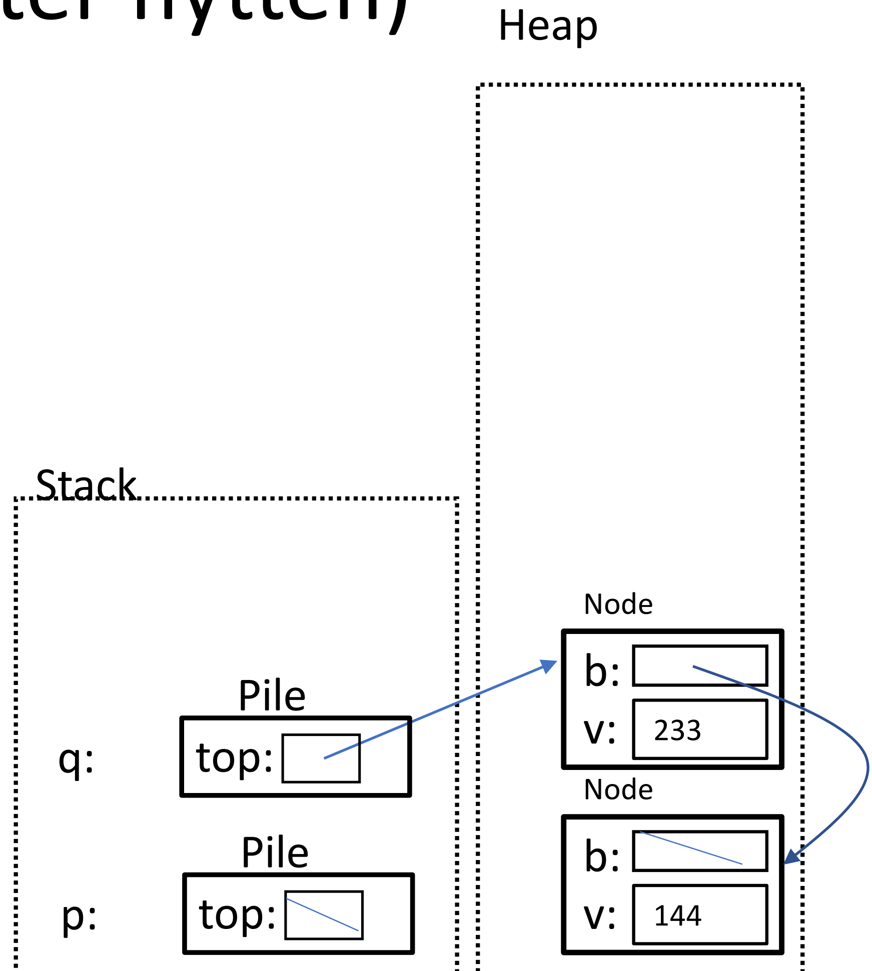
Obs: I exemplet ovan tvingar vi av praktiska skäl fram att flytt anropas istället för kopieringskonstruktion genom att använda std::move.



Flyttkonstruktion (efter flytten)

```
int main()
{
  Pile p;
  p.push(144);
  p.push(233);

  Pile q{std::move(p)};
}
```

Rätt resultat. Objekten q pekar ut den stapel som fanns i p. Objektet p pekar inte längre på något så destruktorn kan köras korrekt.



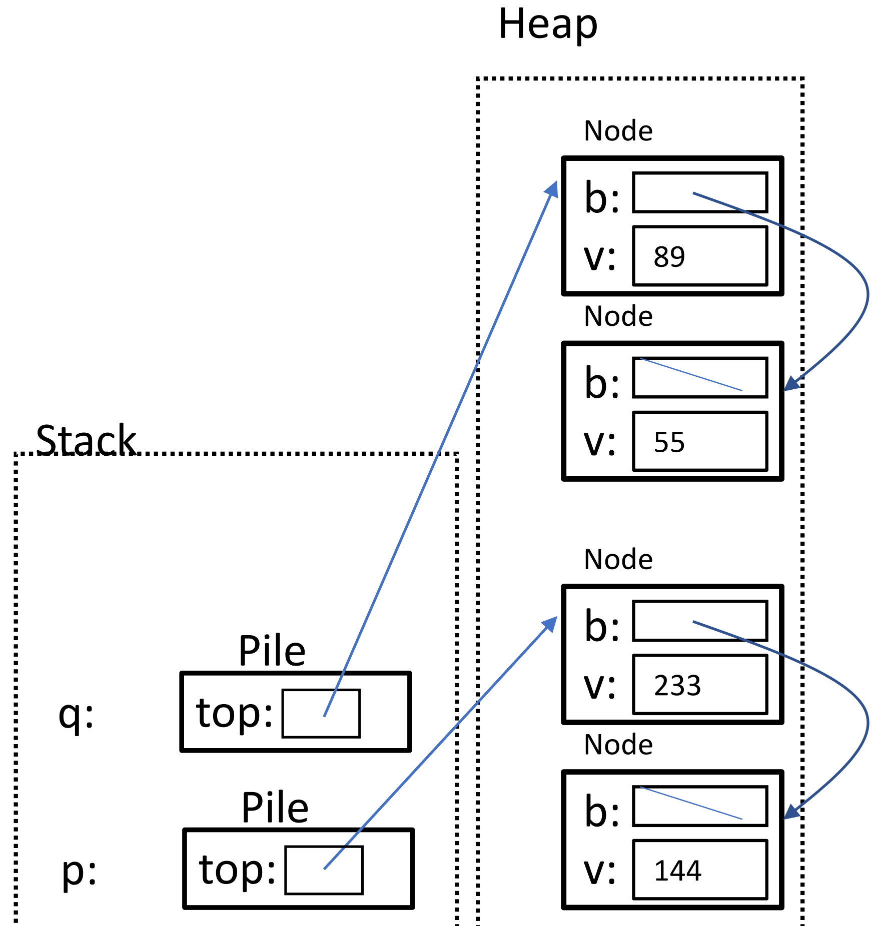
Flytt-tilldelning

- Flytta *tillfälligt* objekt till existerande objekt
- Det tillfälliga objektet kommer inte användas mer
- Byt plats!
- Deklaration:
    - `Pile& Pile::operator=(Pile && rhs);`
- Anropas:
    - Vid tilldelning: `some_pile = other_pile;`
    - Vid överskrivning av befintliga variabler där högerled är tillfälliga temporärobjekt.
    - Kompilatorn gör rätt anrop helt automatiskt.

Korrekt flytt-tilldelning (före flytten)

```
int main()
{
  Pile p{144, 233};
  Pile q{55, 89};

  q = std::move(p);
}
```

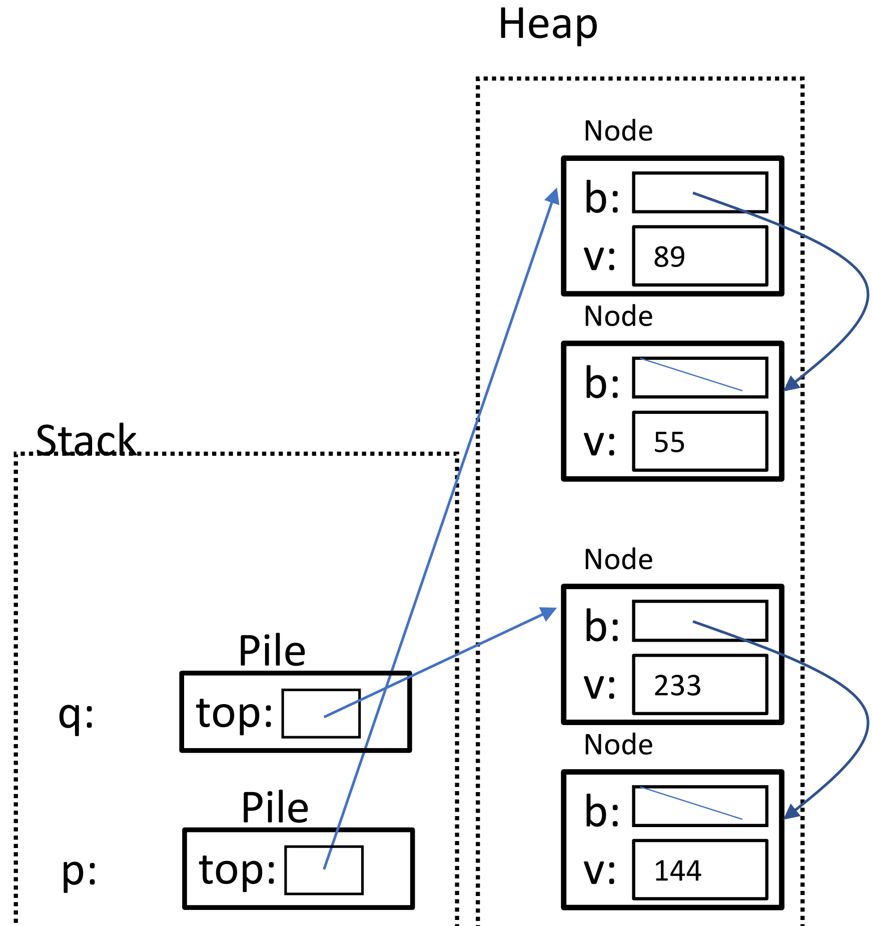
Obs: I exemplet ovan tvingar vi av praktiska skäl fram att flytt anropas istället för kopieringstilldelning genom att använda std::move.



Korrekt flytt-tilldelning (efter flytten)

```
int main()
{
  Pile p{144, 233};
  Pile q{55, 89};

  q = std::move(p);
}
```

Rätt resultat. Objektet p pekar nu ut det som fanns i q och objektet q pekar ut det som fanns i p. Destruktorn för p kommer därför ta hand om det som fanns i q korrekt.



Ta bort automatiska implementationer

- Kompilatorn skapar och använder automatiskt kopieringskonstruktor och kopieringstilldelning.
- Om dessa är felaktiga måste du:
  - Antingen ta bort dem explicit med = *delete*
  - eller implementera korrekta versioner
- Om du varken vill ha en egen version eller kompilatorns standardversion ska du ta bort dem.
  - Lämpligt när kompilatorns standardversioner är felaktiga och du inte behöver kunna kopiera objekt, eller ännu inte vet om kopiering behövs.

```
class Pile {
public:
  Pile(Pile const& p) = delete;
};
```

Rule of [0, 6[

- Det finns många idiom ”Rule of” som skapts allt eftersom C++-standarden utvecklats. För er gäller:
- Rule of 0
  - Om du har en klass med enbart automatiska variabler och utan resurser som måste öppnas eller stängas, implementera inga speciella medlemsfunktioner.
- Rule of 5
  - Om ”Rule of 0” inte passar, implementera alla speciella medlemsfunktioner.

(Konstruktor ej inräknad. Du ska *alltid* ha minst en konstruktor med medlemsinitieringslista för initiering av *alla* datamedlemmar. –Weffc++)

RAII

- Resourse Aquisition Is Initialization

- Vi initierar resurser i konstruktorn
  - allokera anonyma variabler
  - öppna nätverksanslutning
  - öppna filer
  - anslut till databasen
  - anslut till servern

- Vi avvecklar resurser i destruktorn
  - avallokera anonyma variabler
  - stäng nätverksanslutning
  - spara och stäng öppna filer
  - stäng databaskopplingen
  - koppla från servern

9. Iteratorer

Koncept

Implementation

Exempel

Iterator: Koncept

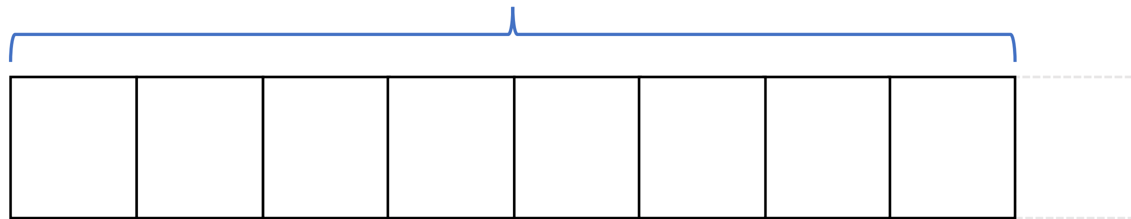
- Vad behövs för att stega igenom en datamängd?
  - Ett sätt att hitta början
  - Ett sätt att avgöra när mängden är slut
  - Ett sätt att gå från en position till nästa
  - Ett sätt att hämta ut eller stoppa in värden på aktuell position
- Hur implementerar vi det generiskt?
  - Det ska se likadant ut oavsett hur datamängden är implementerad.

Iterator: C++ implementation

- Behållaren implementerar
  - Pile::iterator (klass för iterering)
  - iterator Pile::begin()
  - iterator Pile::end()
- Iteratorklassen implementerar
  - iterator::operator!=
  - iterator::operator++ (preinkrement)
  - iterator::operator* (avreferering, ”gå till”)

Iterator: C++ implementation

container (ex: std::vector v{8};)



iterator
it

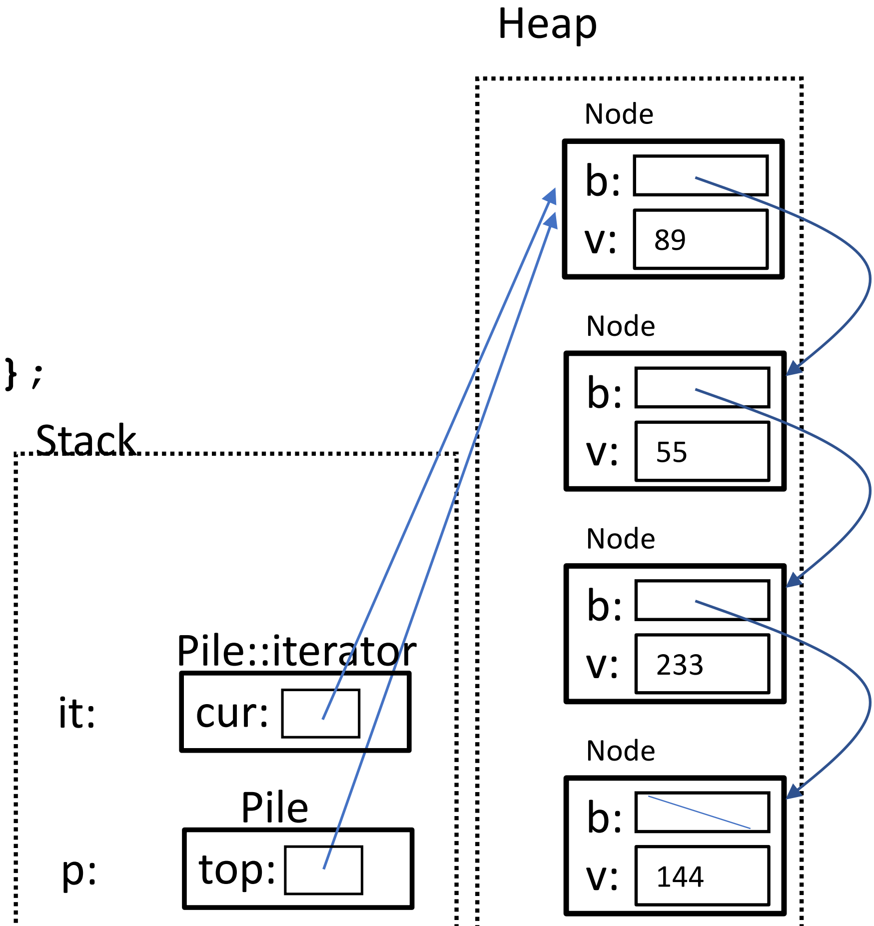
it = v.begin()
it != end // okay
*it // okay
++it // okay

iterator
end

end = v.end()
it != end // okay
*it // undefined
++it // undefined

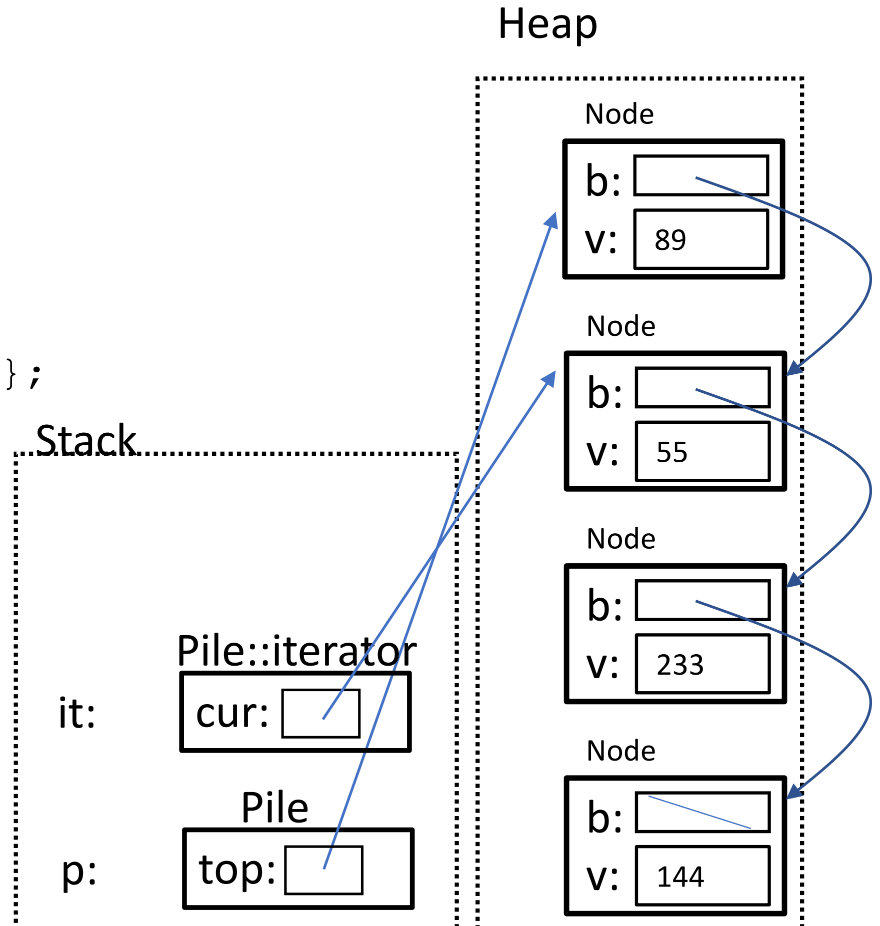
Iterator för stapel

```
int main()
{
  Pile p{144, 233, 55, 89};

  Pile::iterator it{ p.begin() };
  for ( ; it != p.end(); ++it )
  {
    int& value = *it; // 89
    cout << value << endl;
  }
}
```



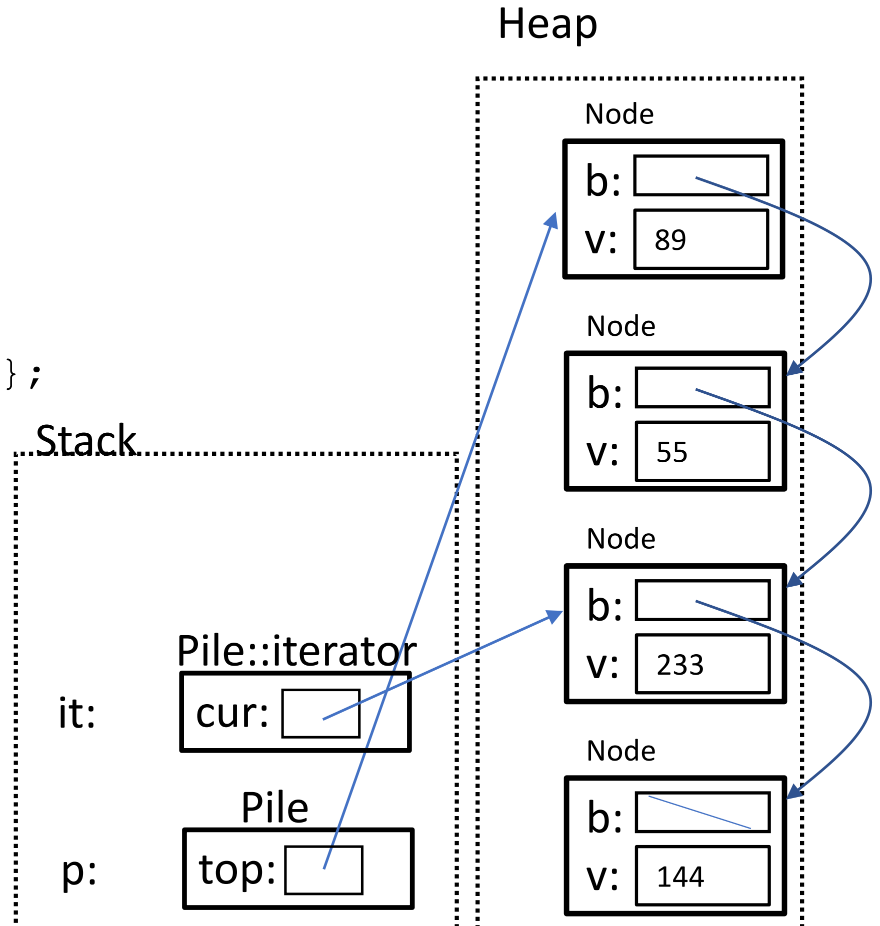
Iterator för stapel

```
int main()
{
  Pile p{144, 233, 55, 89};

  Pile::iterator it{ p.begin() };
  for ( ; it != p.end(); ++it )
  {
    int& value = *it; // 55
    cout << value << endl;
  }
}
```



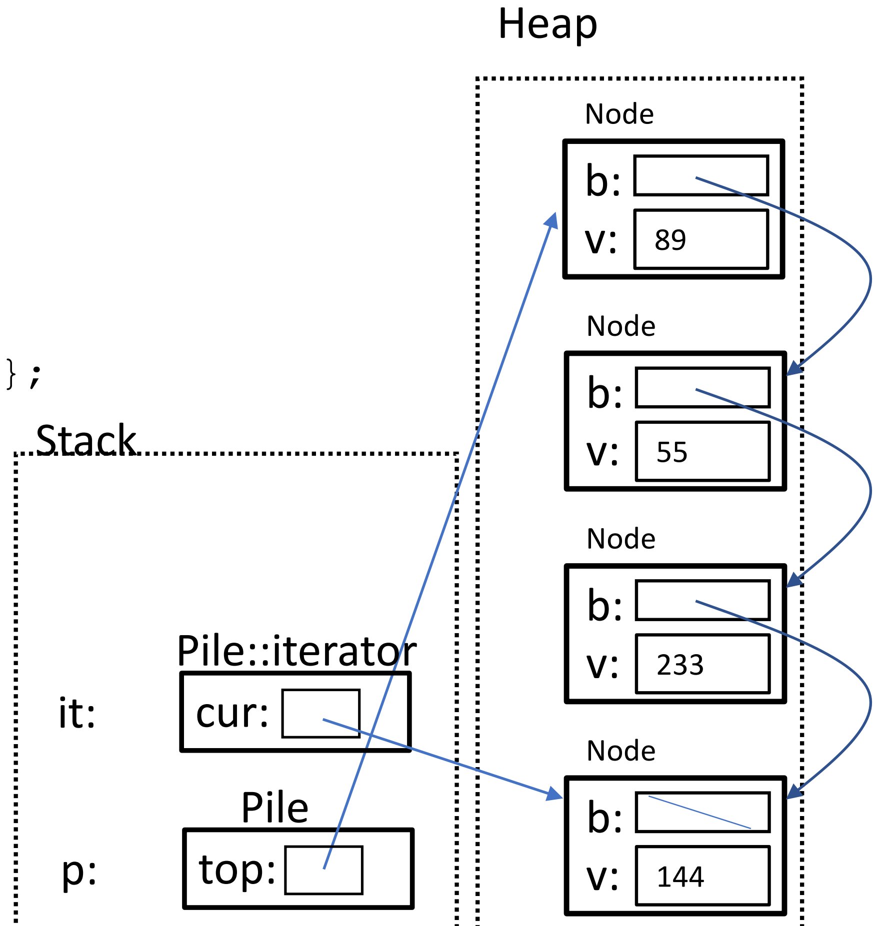
Iterator för stapel

```
int main()
{
  Pile p{144, 233, 55, 89};

  Pile::iterator it{ p.begin() };
  for ( ; it != p.end(); ++it )
  {
    int& value = *it; // 233
    cout << value << endl;
  }
}
```



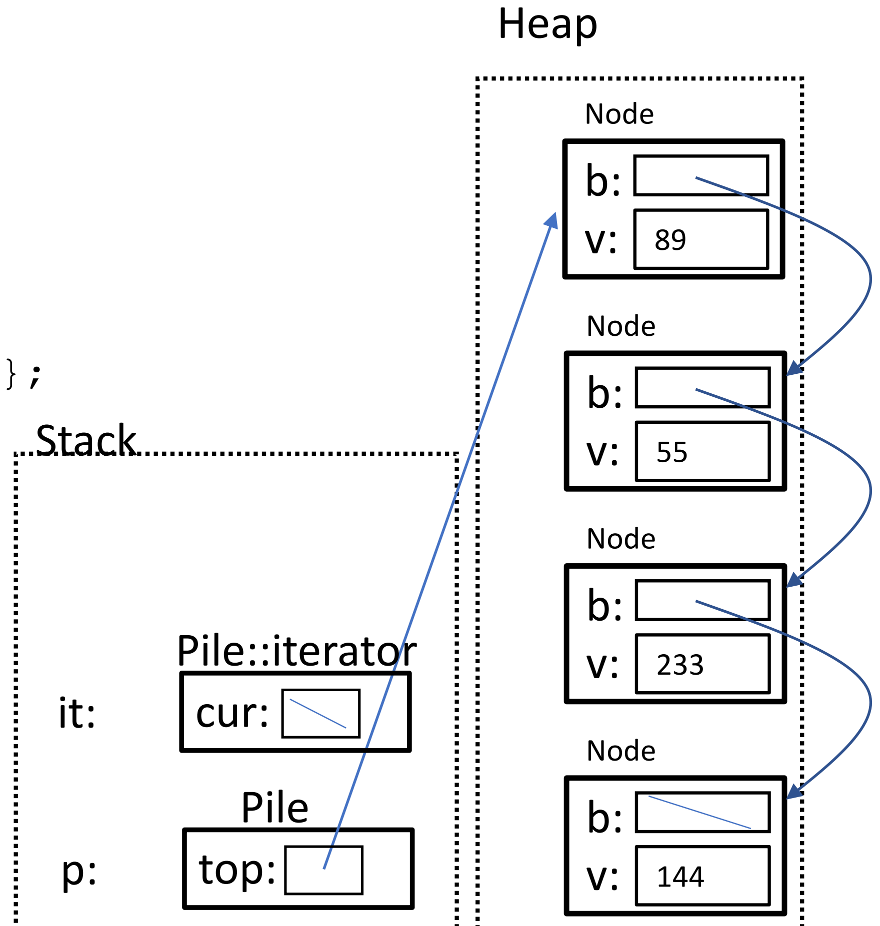
Iterator för stapel

```
int main()
{
  Pile p{144, 233, 55, 89};

  Pile::iterator it{ p.begin() };
  for ( ; it != p.end(); ++it )
  {
    int& value = *it; // 144
    cout << value << endl;
  }
}
```



Iterator för stapel

```
int main()
{
  Pile p{144, 233, 55, 89};

  Pile::iterator it{ p.begin() };
  for ( ; it != p.end(); ++it )
  {
    int& value = *it;
    cout << value << endl;
  }
}
```



Range-for

```
int main()
{
  Pile p{144, 233, 55, 89};

  Pile::iterator it{p.begin()};
  Pile::iterator end{p.end()};
  for ( ; it != end; ++it )
  {
    int& value = *it;
    cout << value << endl;
  }
}
```

```
int main()
{
  Pile p{144, 233, 55, 89};

  // Ekvivalent!
  for ( int& value : p )
  {
    cout << value << endl;
  }
  // kompilatorn översätter
  // ovan loop så det blir
  // som koden till vänster!
}
```

https://en.cppreference.com/w/cpp/language/range-for

Slut för idag.

På återseende.

Pekare – Ballong-analogi

Repetition av avsnitt ”3. dynamiskt minne” ovan.

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.

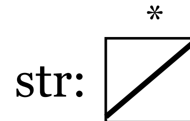
```
str =  new string{”Hej”};
```

string
”Hej”

Skapa en pekarvariabel med * som datatyp

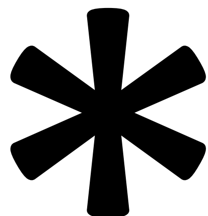
Analogi: Hållare för exakt ett ballongsnöre.

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



LINKÖPINGS UNIVERSITET

Asterisk – förtydligande



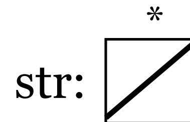
Asterisk efter datatyp
Skapar en pekarvariabel

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.

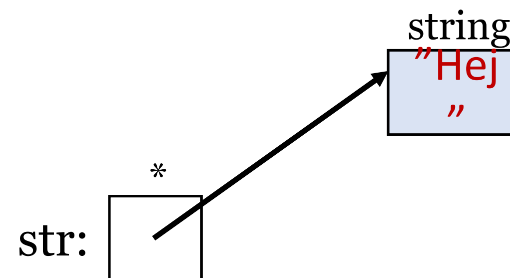
```
string* str{};
str = new string{"Hej"};
```



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"};
```

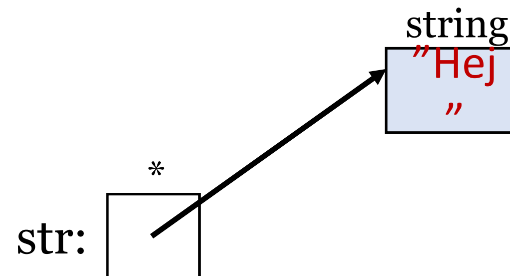


LINKÖPINGS UNIVERSITET

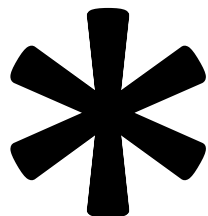
Avreferera ett anonymt objekt med *

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

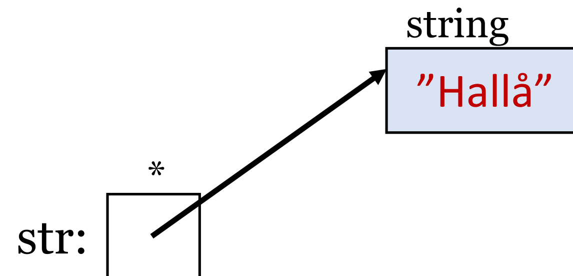
Asterix

Pekar på något

Avreferera ett anonymt objekt med *

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

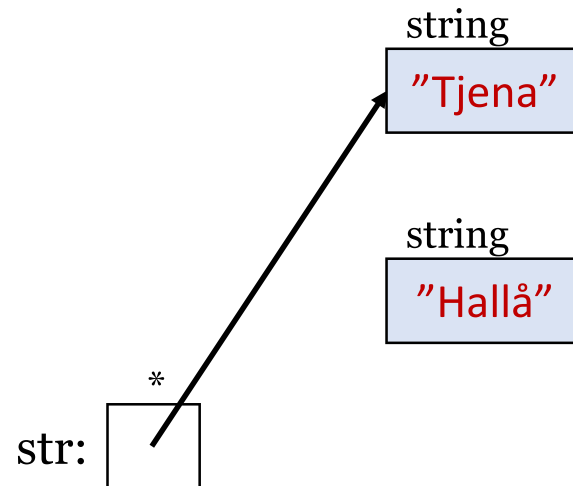
```
string* str{};
str = new string{"Hej"};
*str = "Hallå";
```



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"};
```

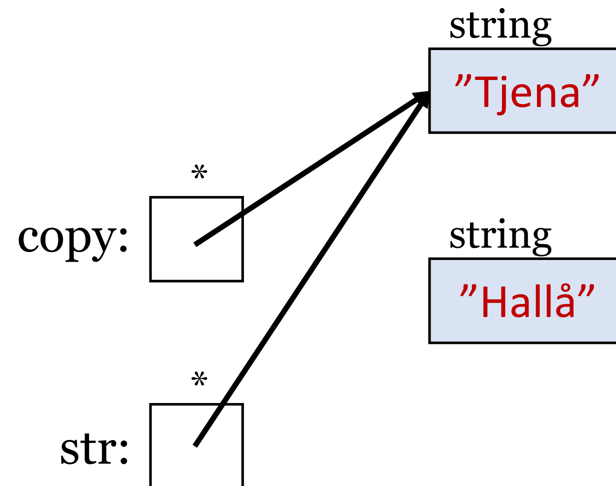


LINKÖPINGS
UNIVERSITET

Kopiering av pekare! (NYTT)

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

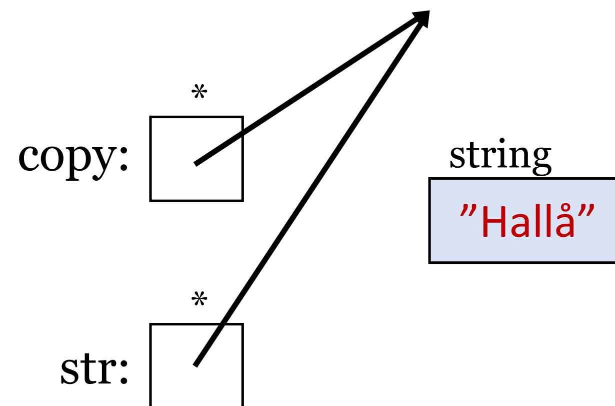
```
string* str{};
str = new string{"Hej"};
*str = "Hallå";
str = new string{"Tjena"};
string* copy{ str };
```



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;
```

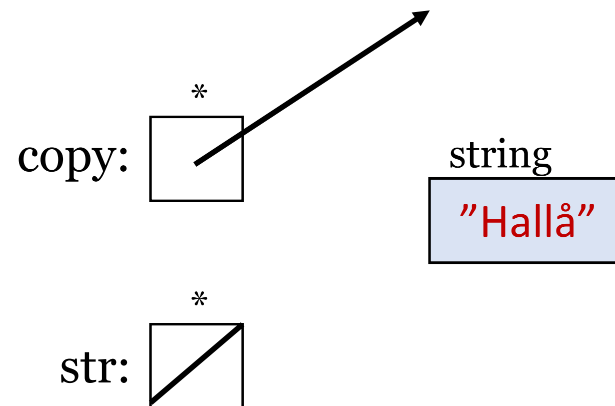


LINKÖPINGS
UNIVERSITET

Markera att en pekare är tom med nullptr

Analogi: Släpp snöret och visa att hållaren är tom.

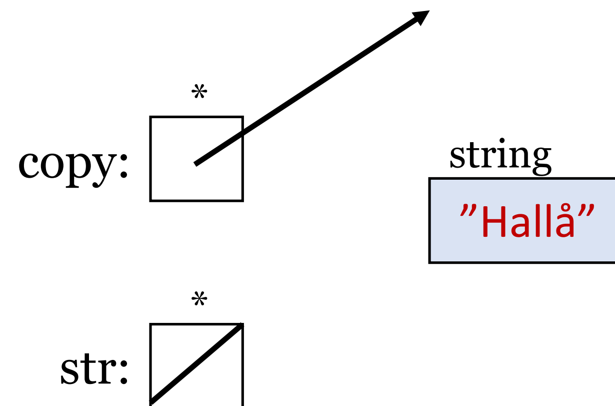
```
string* str{};
str = new string{"Hej"};
*str = "Hallå";
str = new string{"Tjena"};
string* copy{ str };
delete str;
str = nullptr;
```



Vad ska vi göra med ”copy”? (NYTT)

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;
// (yes) delete copy?
// (no)  copy = nullptr?
```

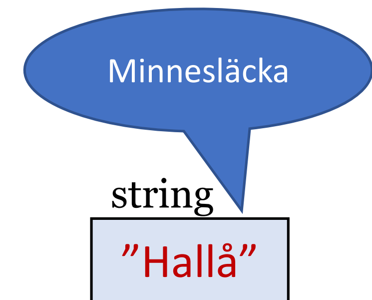


Programmeraren har ansvar för minnet i C++

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.

Vi tittar på språkautomatiken nästa föreläsning. I miniprojektet ska du bortse från minnesläckor till dess du blir ombedd hantera dem.



Pekare

Alternativ analogi

- 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