TDDI82 - Objektorienterad problemlösning

Storseminarium – standardbibliotek

Christoffer Holm

Institutionen för datavetenskap

LINKÖPING UNIVERSITY

1. Upplägg av storseminarium
2. Behållare
3. Algoritmer
4. Modifierande algoritmer
5. Lambdafunktioner
6. Övningsuppgifter

1 Upplägg av storseminarium
2 Behållare
3 Algoritmer
4 Modifierande algoritmer
5 Lambdafunktioner
6 Övningsuppgifter

LINKÖPING UNIVERSITY

Upplägg av storseminarium

Introduktion

- Storseminariet är en kombination mellan föreläsning och labbpass
- Vi kommer upprepa följande upplägg:
  1. Ni löser en uppgift
  2. Diskussion i halvklass
  3. Genomgång av material
- Under tiden som ni löser uppgifter kommer lärare och assistenter gå runt i salen och hjälpa till.
- Om ni blir klara innan utsatt tid: prata med en assistent, de kan ge er utmaningar!

Upplägg av storseminarium

Ungefärlig tidslinje

- 13:20 - 13:40: Uppgift #1
- 14:00 - 14:30: Uppgift #2 + 15 min rast, välj själv när
- 15:00 - 15:40: Uppgift #3 + 15 min rast, välj själv när
- 16:15 - 17:00: Övningsuppgifter

Sluttiderna är då ni ska vara tillbaka i salen för då börjar genomgången.

1 Upplägg av storseminarium
2 Behållare
3 Algoritmer
4 Modifierande algoritmer
5 Lambdafunktioner
6 Övningsuppgifter

LINKÖPING UNIVERSITY

Behållare

Uppgift #1

Skriv ett program som läser in heltal från `std::cin` till en lämplig behållare. Användaren avslutar inmatningen genom att trycka `ctrl+D`.
Skriv sedan ut alla *unika* heltal som användaren matade in. Notera att dessa ska skrivas ut i *sorterad* ordning (i stigande ordning). Inga dubletter av värden får förekomma i utskriften, trots att det potentiellt förekommer dubletter i inmatningen.
Tips: Ditt val av behållare kommer ha en stor påverkan på komplexiteten av din lösning, så fundera noggrannt.

Behållare

Lösning: Uppgift #1 (1/3)

```
1  #include <iostream>
2  #include <vector>
3
4  void remove_duplicates(std::vector<int>& values)
5  {
6    for (int i { 0 }; i < values.size(); ++i)
7    {
8      for (int j { i + 1 }; j < values.size();)
9      {
10       if (values[i] == values[j])
11       {
12         values.erase(values.begin() + j);
13       }
14       else
15       {
16         ++j;
17       }
18     }
19   }
```

Behållare

Lösning: Uppgift #1 (2/3)

```
1  void sort(std::vector<int>& values)
2  {
3    for (int i { 0 }; i < values.size(); ++i)
4    {
5      for (int j { 1 }; j < values.size() - i; ++j)
6      {
7        if (values[j - 1] > values[j])
8        {
9          std::swap(values[j - 1], values[j]);
10       }
11     }
12   }
```

Behållare

Lösning: Uppgift #1 (3/3)

```
1  int main()
2  {
3    std::vector<int> values { };
4
5    int value { };
6    while (std::cin >> value)
7    {
8      values.push_back(value);
9    }
10
11   remove_duplicates(values);
12   sort(values);
13
14   for (int value : values)
15   {
16     std::cout << value << std::endl;
17   }
18 }
```

Behållare

Alternativ lösning!

Låt oss testa en annan behållare!

Behållare

Lösning: Uppgift #1 med `std::set`

```
1  #include <iostream>
2  #include <set>
3
4  int main()
5  {
6    std::set<int> values { };
7
8    int value { };
9    while (std::cin >> value)
10   {
11     values.insert(value);
12   }
13
14   for (int value : values)
15   {
16     std::cout << value << std::endl;
17   }
18 }
```

Behållare

Diskussion: Uppgift #1

- Eftersom att `std::set` garanterar att element är unika så är den högst lämplig för detta problem.
- Om vi inte använder `std::set` behöver vi skriva kod som letar efter och tar bort dubletter själva.
- `std::set` garanterar också att den itereras i sorterad ordning. När en annan behållare väljs måste vi sortera behållaren innan vi själva skriver ut allting.
- **Men:** i fallen då vi inte kan välja `std::set` så vore det toppen om vi kunde förenkla de stegen också...

Algoritmer

Processera data

- I förra föreläsningen nämnde vi att processering av data var ett av de viktigaste stegen.
- Vi har nu sett att i vissa fall går det att lösa processering genom att välja lämpliga behållare.
- Men det är inte alltid garanterat att det finns en behållare som löser det problem vi har.
- Hittils har vi då använt loopar och andra konstruktioner för att implementera logik.
- Men vissa kodstycken skulle man då behöva skriva om och om igen... Det vill vi undvika!

Algoritmer

STL Algoritmer

- I STL finns det 100+ s.k. algoritmer
- Dessa är funktioner som löser vanliga problem utan att vi måste skriva lösningen för detta från början varje gång. Vissa problem är också såpass svårlösta att det är viktigt att språket tillhandahåller en färdigskriven lösning åt programmeraren.
- Exempel på problem som algoritmer kan lösa: sortering, summering av behållare, hitta delmängder, blanda behållare o.s.v.

Algoritmer

STL Algoritmer

- Ett annat viktigt syfte med algoritmer är att skapa ett gemensamt språk för alla C++ programmerare.
- Idéen är att om du använder en egenskriven lösning för ett problem så tar det längre tid för en annan programmerare att förstå vad du har gjort jämfört med om du bara anropar en lämplig STL algoritm.
- På nästa slide kommer ett exempel som förhoppningsvis illustrerar detta.

Algoritmer

Vad gör detta program?

```
1  std::vector<int> values { /* ... */ };
2
3  int x { 0 };
4  int y { 1 };
5
6  for (auto it = values.begin(); it != values.end(); ++it)
7  {
8    if (*it == y)
9    {
10     ++x;
11   }
12 }
13
14 std::cout << x << std::endl;
```

Algoritmer

Svar: Koden räknar antalet ettor som förekommer i `values`

Algoritmer

Jämför med:

```
1 std::vector<int> values { /* ... */ };
2 int x { std::count(values.begin(), values.end(), 1) };
3 std::cout << x << std::endl;
```

Algoritmer

`std::count()`

- Målsättningen med denna kurs är att du ska kunna tillräckligt mycket om algoritmer för att snabbt och enkelt förstå vad som pågår i andra exemplet.
- Men för att du ska kunna förstå det fullt ut måste vi först lära oss exakt hur dessa algoritmer används.
- Notera t.ex. att algoritmen `std::count()` tar emot *iteratorer* för behållaren den vill räkna antalet ettor i.

Algoritmer

Noteringar om algoritmer

- Algoritmer är funktioner som opererar på sekvenser av data (t.ex. behållare)
- Iteratorer tillhandahåller ett gemensamt gränssnitt för hur sekvenser av data itereras och tillåter oss att uttrycka delsekvenser.
- Därför jobbar *alla* algoritmer med *iteratorer* (snarare än behållare).
- Detta ger ett antal fördelar:
    1. Det begränsar oss inte till att använda just behållare.
    2. Vi kan låta algoritmer operera på delsekvenser av behållare, genom att ange andra intervall än just från `.begin()` till `.end()`.
    3. Det tillåter oss att endast oroa oss för egenskaper hos iteratorer, snarare än egenskaper hos behållarna och dess iteratorer.
- Dock medför denna design att algoritmerna är något svårare att lära sig jämfört med motsvarande funktionalitet i andra språk.

Algoritmer

Noteringar om algoritmer

- Algoritmerna är i regel namngedda på ett sätt som gör det (förhoppningsvis) uppenbart för läsaren av koden vad funktionen faktiskt gör.
- Med vana blir det därför oerhört snabbt för läsaren att förstå vad som sker när algoritmer används istället för handskrivna kodstycken.
- Algoritmerna i standardbiblioteket är skrivna för att vara minst lika effektiva (eller effektivare) jämfört med motsvarande handskrivna alternativ.
- Därför finns det (oftast) ingen fördel med att välja handskrivna alternativ över algoritmer (när en lämplig algoritm finns).

Algoritmer

Vad gör denna kod?

```
1  std::vector<int> values { /* ... */ };
2
3  int y { 1 };
4  auto it = values.begin();
5
6  while (it != values.end())
7  {
8    if (*it == y)
9    {
10     break;
11   }
12   ++it;
13 }
14
15 // vad är it här?
```

Algoritmer

Förklaring

- Koden hittar den *första* förekomsten av `1` i `values` behållaren.
- Specifikt hittar den *platsen* (representerad som en iterator) i behållaren där den första `1`:an ligger.
- Om det inte finns någon `1`:a så blir iteratorn lika med `values.end()`, vilket alltså tillåter oss att använda koden för två olika syften:
    1. Vi kan använda kodstycket för att hitta huruvida ett värde förekommer i behållaren överhuvudtaget (genom att jämföra iteratorn med `values.end()`).
    2. Kodstycket kan också användas för att hitta *vart* den första förekomsten av det sökta värdet förekommer.

Algoritmer

Jämför förra exemplet med:

```
1 std::vector<int> values { /* ... */ };
2 auto it = std::find(values.begin(), values.end(), 1);
3
4 // vad är it här?
```

Algoritmer

cppreference.com

Öppna följande sida: `https://en.cppreference.com/w/cpp/algorithm`

Algoritmer

cppreference.com

- Länken på föregående slide leder till algoritm-sidan på cppreference. Här listas *alla* algoritmer som finns i C++.
- **Tips:** För att se C++17 algoritmerna tydligare (vilket är de som är relevanta för just den här kursen), kan du använda drop-down menyn ”Standard revision” och välja ”C++17”

Algoritmer

Uppgift #2

Läs in en behållare av heltal från `std::cin` (precis på samma sätt som i förra uppgiften). När användaren är klar med sin inmatning trycker denne `ctrl+D`.
Använd sedan lämpliga algoritmer för att hitta det största värdet i behållaren och skriv sedan ut hur många gånger det största värdet förekom i behållaren.
Notera: Poängen med uppgiften är att söka upp lämpliga algoritmer på cppreference.

Algoritmer

Lösning: Uppgift #2

```
1  #include <algorithm>
2  #include <iostream>
3  #include <vector>
4
5  int main()
6  {
7    std::vector<int> values { };
8
9    int value { };
10   while (std::cin >> value)
11   {
12     values.push_back(value);
13   }
14
15   auto it = std::max_element(values.begin(), values.end());
16   std::cout << std::count(it, values.end(), *it) << std::endl;
17 }
```

Algoritmer

Diskussion Uppgift #2

- Notera att det även funkar att göra `std::sort` och sen kolla på sista värdet för att få det största.
- En viktig detalj man kan notera med `std::max_element` är att denne returnerar en iterator till det största värdet, snarare än värdet i sig. Detta görs av flera skäl:
  - Om elementet inte finns returneras end-iteratorn snarare än att vi behöver hantera fel
  - Det är mer generellt för att nu kan vi även använda den för att hitta *vart* det största elementet finns och vad det har för värde.
- Du kan även se i lösningsförslaget att vi använder det faktumet att `std::max_element` returnerar *den första* förekomsten av det största värdet. Specifikt använder vi detta för att begränsa vilken del av behållaren vi räknar i, eftersom att det första värdet garanterat inte finns före `it`.

1 Upplägg av storseminarium
2 Behållare
3 Algoritmer
4 Modifierande algoritmer
5 Lambdafunktioner
6 Övningsuppgifter

LINKÖPING UNIVERSITY

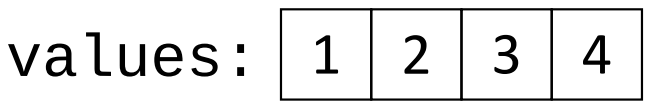
Modifierande algoritmer

- Hittils har vi bara kollat på algoritmer som tar reda på information om sekvenser.
- Men självklart finns det många situationer där vi vill kunna köra algoritmer som ändrar på datan.
- Dessa algoritm kallas *modifierande algoritmer*.

Modifierande algoritmer

`std::fill`

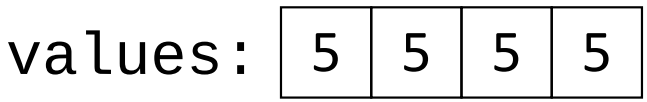
```
1 std::vector<int> values { 1, 2, 3, 4 };
2 std::fill(values.begin(), values.end(), 5);
```



Modifierande algoritmer

`std::fill`

```
1 std::vector<int> values { 1, 2, 3, 4 };
2 std::fill(values.begin(), values.end(), 5);
```



Modifierande algoritmer

`std::fill`

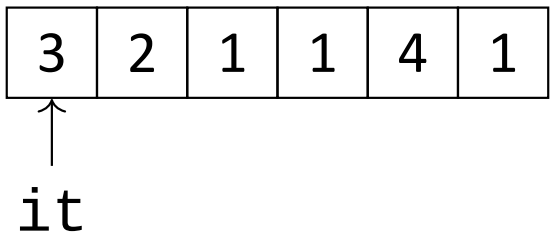
- `std::fill` skriver över alla värden i en sekvens med ett specifikt värde.
- Detta är den mest grundläggande formen av modifiering som kan göras.
- Observera att denna algoritm modifierar *redan befintliga* värde.
- Om behållaren är tom gör `std::fill` ingenting.

Modifierande algoritmer

Borttagning av värden

```
1  std::vector<int> v { 3, 2, 1, 1, 4, 1 };
2  for (auto it { v.begin() }; it != v.end(); ++it)
3  {
4    if (*it == 1)
5    {
6      v.erase(it);
7    }
8  }
```

Första är inte en etta, så vi stegar vidare!

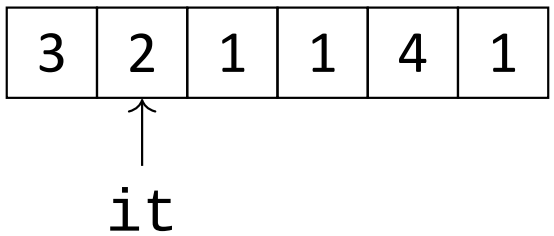


Modifierande algoritmer

Borttagning av värden

```
1  std::vector<int> v { 3, 2, 1, 1, 4, 1 };
2  for (auto it { v.begin() }; it != v.end(); ++it)
3  {
4    if (*it == 1)
5    {
6      v.erase(it);
7    }
8  }
```

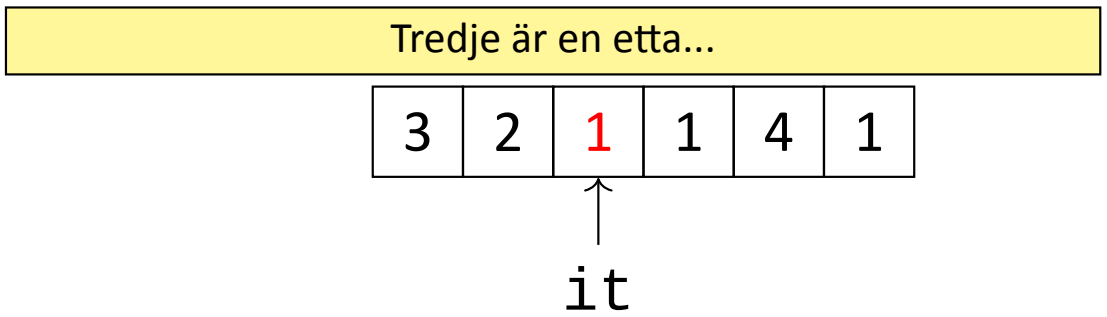
Andra är inte heller en etta, så vi stegar till nästa!



Modifierande algoritmer

Borttagning av värden

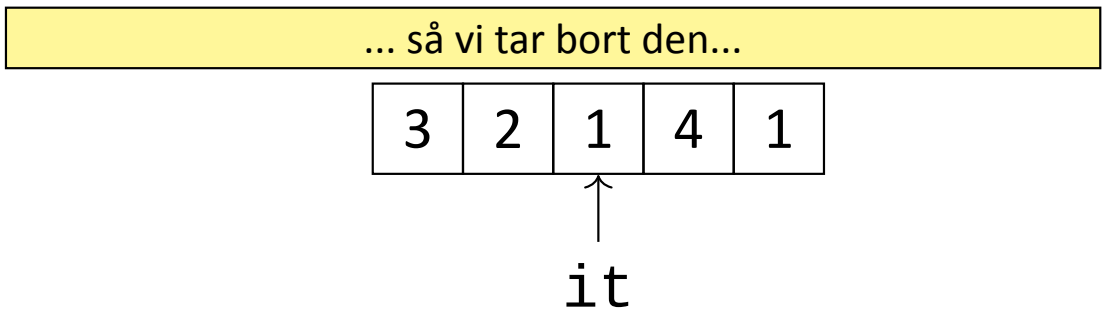
```
1  std::vector<int> v { 3, 2, 1, 1, 4, 1 };
2  for (auto it { v.begin() }; it != v.end(); ++it)
3  {
4    if (*it == 1)
5    {
6      v.erase(it);
7    }
8  }
```



Modifierande algoritmer

Borttagning av värden

```
1  std::vector<int> v { 3, 2, 1, 1, 4, 1 };
2  for (auto it { v.begin() }; it != v.end(); ++it)
3  {
4    if (*it == 1)
5    {
6      v.erase(it);
7    }
8  }
```

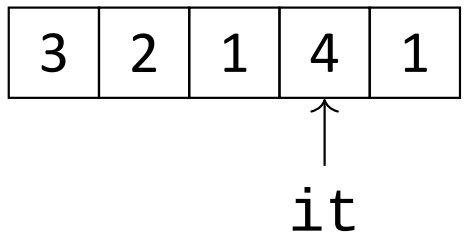


Modifierande algoritmer

Borttagning av värden

```
1  std::vector<int> v { 3, 2, 1, 1, 4, 1 };
2  for (auto it { v.begin() }; it != v.end(); ++it)
3  {
4    if (*it == 1)
5    {
6      v.erase(it);
7    }
8  }
```

... och stegar sedan vidare, utan att kolla på nästa etta?!

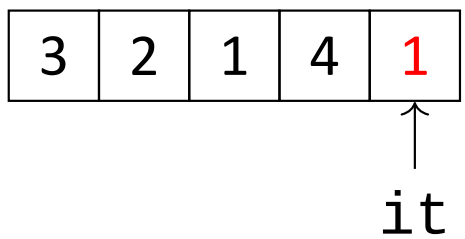


Modifierande algoritmer

Borttagning av värden

```
1  std::vector<int> v { 3, 2, 1, 1, 4, 1 };
2  for (auto it { v.begin() }; it != v.end(); ++it)
3  {
4    if (*it == 1)
5    {
6      v.erase(it);
7    }
8  }
```

Men den här ettan hittar den? Varför?



Modifierande algoritmer

Förklaring

- På föregående slide försökte vi skriva en loop som tar bort värden.
- Men som vi noterade blev det lite fel, den missade vissa ettor men inte alla.
- Detta beror på att när vi faktiskt tar bort värden så måste `std::vector` fylla igen hålet som uppstår, så att vi ersätter värdet som tas bort med nästa...
- Men i slutet av varje iteration stegar vi iteratorn. Det resulterar i att elementet som ersatte det borttagna blir aldrig kontrollerat.

Modifierande algoritmer

Lösning

```
1  std::vector<int> v { 3, 2, 1, 1, 4, 1 };
2  for (auto it { v.begin() }; it != v.end();)
3  {
4    if (*it == 1)
5    {
6      it = v.erase(it);
7    }
8    else
9    {
10     ++it;
11   }
12 }
```

Modifierande algoritmer

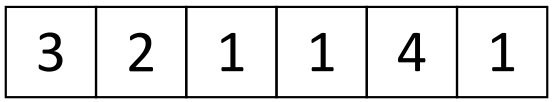
Lösning: förklaring

- På föregående slide löser vi problemet med borttagning.
- Lösningen innebär att vi endast stegar om ett element *inte* togs bort.
- Vi löser också andra problem som inte har diskuterats genom att fånga returvärdet av `v.erase()` anropet och ersätter `it` med det.
- Dessa problem dyker inte upp för alla behållare, men för t.ex. länkade listor skulle det bli problematiskt om iteratorn fortfarande pekade på noden som togs bort (vi skulle tappa bort resten av listan helt enkelt). Därför returnerar `v.erase()` iteratorn till *nästa* plats.
- Så vi kan helt enkelt se det som att borttagning av ett element också är en stegning.
- Men detta är ganska invecklat, mycket detaljer att tänka på...
- Kan vi inte använda något från standardbiblioteket för att lösa detta?

Modifierande algoritmer

Borttagning med algoritm

```
1 std::vector<int> v { 3, 2, 1, 1, 4, 1 };
2 auto it = std::remove(v.begin(), v.end(), 1);
3 v.erase(it, v.end());
```



Modifierande algoritmer

Borttagning med algoritm

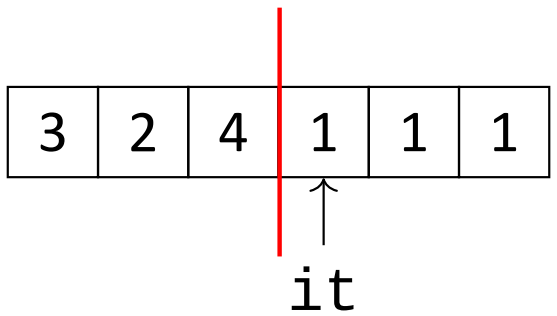
```
1 std::vector<int> v { 3, 2, 1, 1, 4, 1 };
2 auto it = std::remove(v.begin(), v.end(), 1);
3 v.erase(it, v.end());
```

3	2	4	1	1	1

Modifierande algoritmer

Borttagning med algoritm

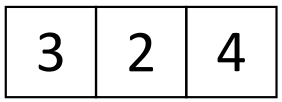
```
1 std::vector<int> v { 3, 2, 1, 1, 4, 1 };
2 auto it = std::remove(v.begin(), v.end(), 1);
3 v.erase(it, v.end());
```



Modifierande algoritmer

Borttagning med algoritm

```
1 std::vector<int> v { 3, 2, 1, 1, 4, 1 };
2 auto it = std::remove(v.begin(), v.end(), 1);
3 v.erase(it, v.end());
```



Modifierande algoritmer

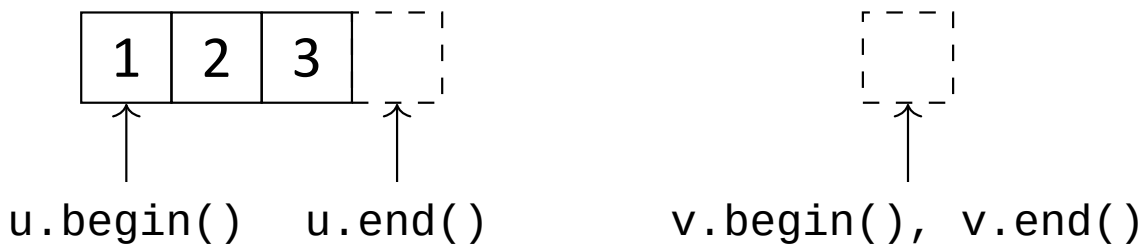
Borttagning med algoritm: noteringar

- Algoritmer har endast tillgång till iteratorer.
- Iteratorer håller endast koll på befintliga platser.
- Detta innebär att iteratorer kan inte *ta bort* värden, endast modifiera dem.
- Så en algoritm kan aldrig ta bort värden.
- **Men**: en algoritm kan flytta runt värden inuti en behållare.
- På detta viset kan en algoritm strukturera om behållaren så att det blir väldigt lätt för oss att be behållaren ta bort värdena, vilket är så alla algoritmer som ”tar bort” värden fungerar.
- Detta kallas *remove + erase idiomet* och är väldigt vanlig inom standardbiblioteket.
- Så om borttagning fungerar annorlunda, hur fungerar det då att ”lägga till” nya värden med hjälp av en algoritm?

Modifierande algoritmer

Kopiering till annan behållare

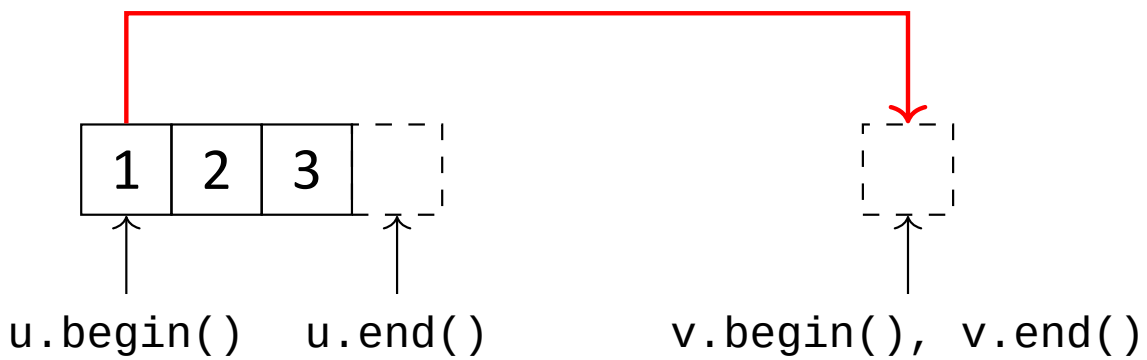
```
1 std::vector<int> u { 1, 2, 3 };
2
3 std::vector<int> v { };
4
5 std::copy(u.begin(), u.end(), v.begin());
```



Modifierande algoritmer

Kopiering till annan behållare

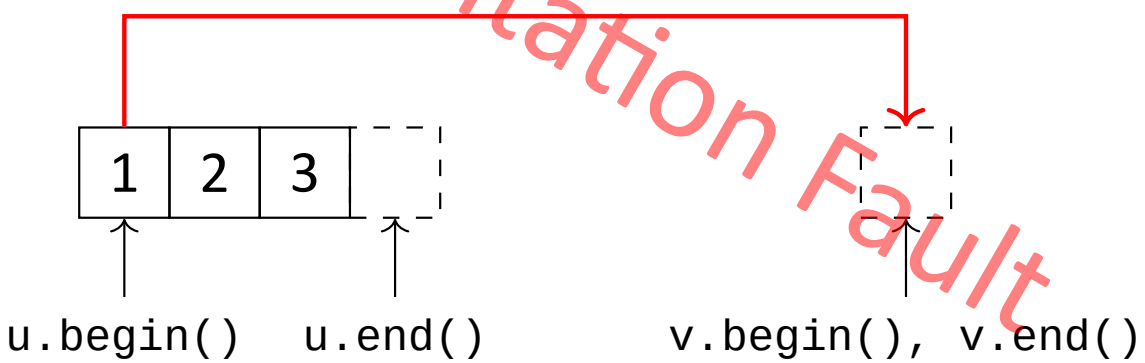
```
1 std::vector<int> u { 1, 2, 3 };
2
3 std::vector<int> v { };
4
5 std::copy(u.begin(), u.end(), v.begin());
```



Modifierande algoritmer

Kopiering till annan behållare

```
1 std::vector<int> u { 1, 2, 3 };
2
3 std::vector<int> v { };
4
5 std::copy(u.begin(), u.end(), v.begin());
```

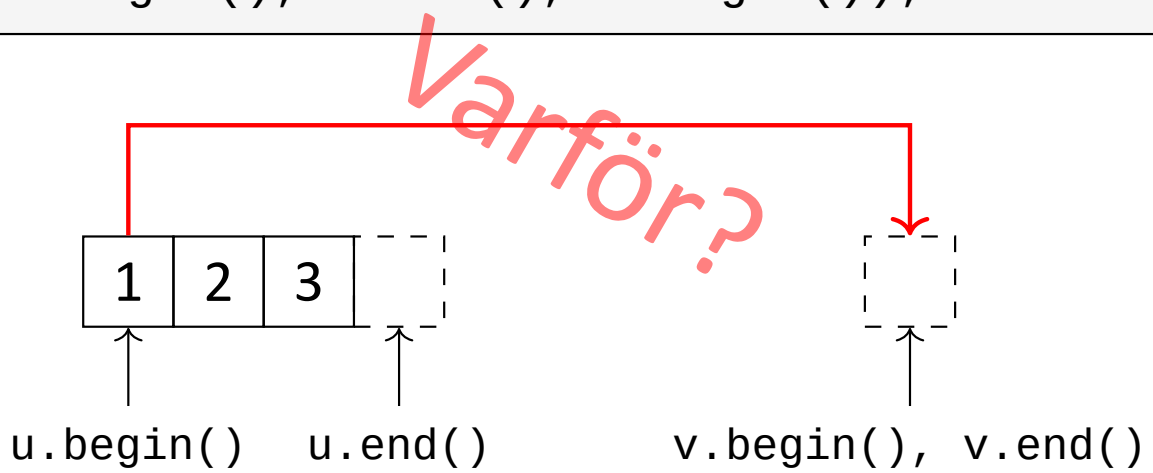


Segmentation Fault

Modifierande algoritmer

Kopiering till annan behållare

```
1 std::vector<int> u { 1, 2, 3 };
2
3 std::vector<int> v { };
4
5 std::copy(u.begin(), u.end(), v.begin());
```



Modifierande algoritmer

Varför blev det fel?

- Algoritmer har endast tillgång till iteratorer...
- ... och iteratorer kan endast komma åt befintliga värden.
- De första två iteratorerna som skickas till `std::copy` är alla värden som vi vill kopiera.
- Den tredje iteratorn är vart de kopierade värdena ska placeras. Notera att vi *inte* anger slutet av intervallet som värden ska kopieras till. Detta är för att algoritmen antar att det finns tillräckligt med utrymme dit vi kopierar till.
- Men detta antagande bryter vi på föregående slide, eftersom att `v` är *tom* (d.v.s. det finns *inga* platser att skriva till) medan `u` innehåller tre element. D.v.s. att vi försöker skriva över tre element i den *tomma* vektorn `v`.

Modifierande algoritmer

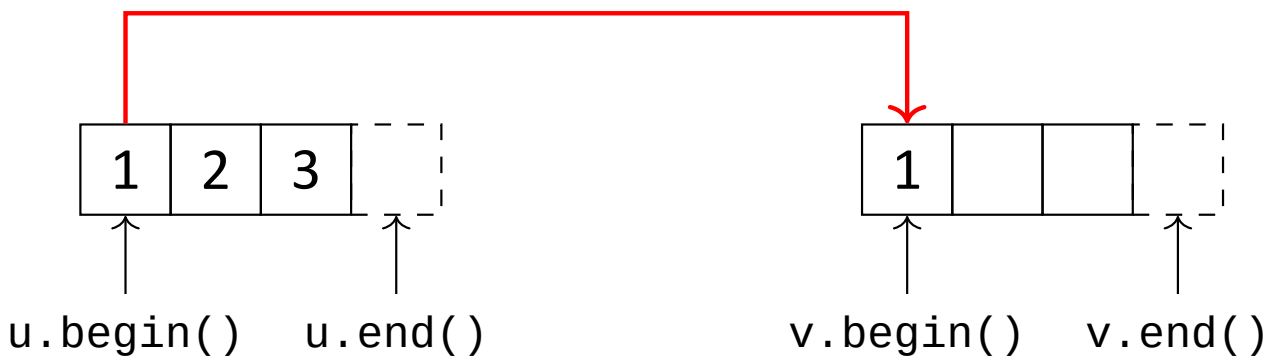
Lösning

```
1 std::vector<int> u { 1, 2, 3 };
2 // se till att `v` har lika många platser som `u`
3 std::vector<int> v(u.size());
4
5 std::copy(u.begin(), u.end(), v.begin());
```

Modifierande algoritmer

Lösning

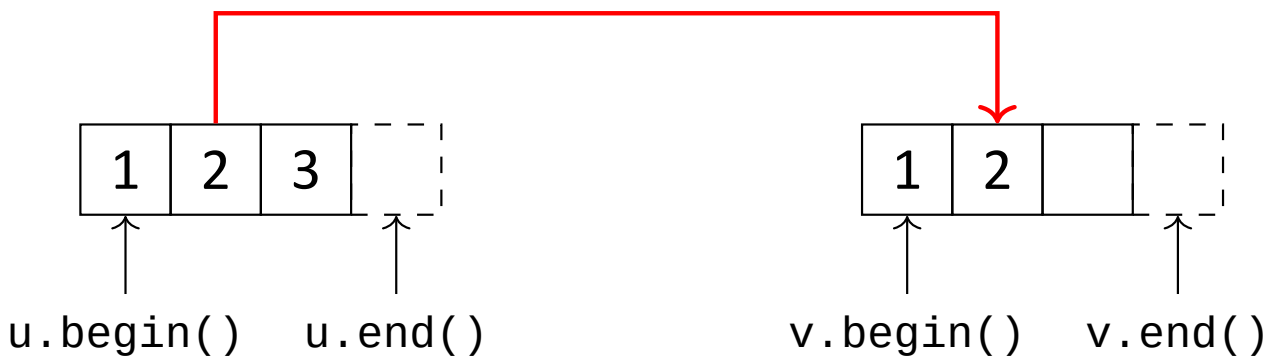
```
1 std::vector<int> u { 1, 2, 3 };
2 // se till att `v` har lika många platser som `u`
3 std::vector<int> v(u.size());
4
5 std::copy(u.begin(), u.end(), v.begin());
```



Modifierande algoritmer

Lösning

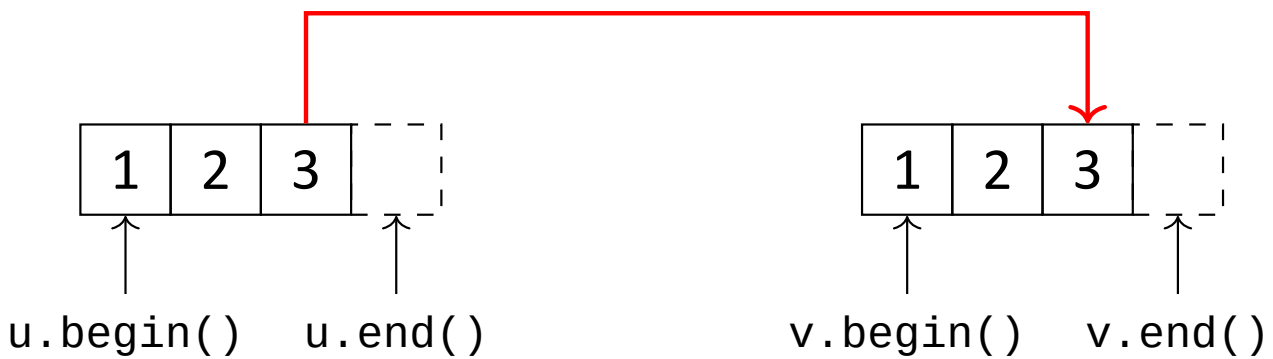
```
1 std::vector<int> u { 1, 2, 3 };
2 // se till att `v` har lika många platser som `u`
3 std::vector<int> v(u.size());
4
5 std::copy(u.begin(), u.end(), v.begin());
```



Modifierande algoritmer

Lösning

```
1 std::vector<int> u { 1, 2, 3 };
2 // se till att `v` har lika många platser som `u`
3 std::vector<int> v(u.size());
4
5 std::copy(u.begin(), u.end(), v.begin());
```



Modifierande algoritmer

Lösning

```
1 std::vector<int> u { 1, 2, 3 };
2 // se till att `v` har lika många platser som `u`
3 std::vector<int> v(u.size());
4
5 std::copy(u.begin(), u.end(), v.begin());
```

Funkar!

Modifierande algoritmer

Varför funkar det nu?

- Anledningen till att det nu funkar är för att vi har garanterat att v innehåller minst lika många platser som u.
- Många modifierande algoritmer jobbar på det här viset. D.v.s. att vi tar värden från ett intervall och skriver det till ett annat.
- Då är det viktigt att ha koll på att i dessa situationer antar algoritmen att det finns tillräckligt med platser i behållaren man skriver till.

Modifierande algoritmer

Intressant problem: Inläsning

Hur gör vi det här med algoritmer?

```
1  std::vector<int> v { };
2
3  int x { };
4  while (std::cin >> x)
5  {
6    v.push_back(x);
7  }
```

Modifierande algoritmer

Inläsning

- Det finns två problem här som gör det svårt att göra inläsning av värden från `std::cin` m.h.a. algoritmer.
- 1: Algoritmer jobbar med iteratorer, så hur kan vi skapa iteratorer som läser från `std::cin`?
- 2: Vi behöver veta hur många platser `v` ska ha för att få plats med allt som användaren matar in, men det vet vi inte förren vi har gjort inläsningen.
- Båda dessa problem går dock att hantera!

Modifierande algoritmer

Inläsning med algoritmer!

```
1 std::vector<int> v { };
2 std::copy(std::istream_iterator<int>{ std::cin },
3           std::istream_iterator<int>{ },
4           std::back_inserter(v));
```

I det här exemplet har vi löst båda problemen som påtalades i föregående slide. Övergripande kan vi se att vi kopierar från `std::cin` till `v` m.h.a. `std::copy`.

Modifierande algoritmer

Inläsning med algoritmer!

```
1 std::vector<int> v { };
2 std::copy(std::istream_iterator<int>{ std::cin },
3           std::istream_iterator<int>{ },
4           std::back_inserter(v));
```

`std::istream_iterator<T>` kallas en *ström-iterator* vilket är en iterator som läser värden av typen T från den specificerade strömmen (`std::cin` i detta fall).

Modifierande algoritmer

Inläsning med algoritmer!

```
1 std::vector<int> v { };
2 std::copy(std::istream_iterator<int>{ std::cin },
3           std::istream_iterator<int>{ },
4           std::back_inserter(v));
```

När strömmen ”tar slut” (d.v.s. när användaren trycker `ctrl+D` i det här fallet) innebär det att strömmen blir ”tom”. Därför säger vi att end-iteratorn för en ström är en `std::istream_iterator<T>` som *inte* innehåller någon ström.

Modifierande algoritmer

Inläsning med algoritmer!

```
1 std::vector<int> v { };
2 std::copy(std::istream_iterator<int>{ std::cin },
3           std::istream_iterator<int>{ },
4           std::back_inserter(v));
```

Istället för att försöka förutspå hur många platser som behövs i `v`, så skapar vi istället en speciell typ av iterator som kallas en `std::back_inserter`. Det är en iterator som inte läser några värden, utan den fungerar istället som så att varje gång man skriver till den så gör den `.push_back()` av värdet till den specificerade behållaren. Den här iteratorn förekommer vanligtvis som ”output” parameter till algoritmer (den tredje iteratorn).

Modifierande algoritmer

Utskrift med algoritmer

Hur gör det här med algoritmer?

```
1 std::vector<int> v { 1, 2, 3 };
2 for (int x : v)
3 {
4   std::cout << x << " ";
5 }
```

Modifierande algoritmer

Utskrift med algoritmer

- Precis som med inläsning blir det inte så uppenbart hur det fungerar när man ska försöka skriva ut saker till terminalen m.h.a. algoritmer.
- Men om man observerar att utskrift egentligen bara är att kopiera data från *behållaren* till *terminalen*, kan man förhoppningsvis inse ganska snabbt att `std::copy()` bör fungera!
- Det är kanske inte helt orimligt att baserat på existensen av `std::istream_iterator` gissa att det finns en motsvarande `std::ostream_iterator` för att skriva ut. Denna iterator fungerar likt `std::back_inserter` men istället för att göra `push_back()` så gör den istället en utskrift.

Modifierande algoritmer

Utskrift med algoritmer

```
1 std::vector<int> v { 1, 2, 3 };
2 std::copy(v.begin(), v.end(),
3           std::ostream_iterator<int>{ std::cout, " " });
```

Vi kopierar *från* v *till* `std::cout` m.h.a. `std::copy` och `std::ostream_iterator`.

Modifierande algoritmer

Utskrift med algoritmer

```
1 std::vector<int> v { 1, 2, 3 };
2 std::copy(v.begin(), v.end(),
3           std::ostream_iterator<int>{ std::cout, " " });
```

Notera att `std::ostream_iterator<T>` är en iterator som skriver ut värden av typen T till den specificerade strömmen.

Modifierande algoritmer

Utskrift med algoritmer

```
1 std::vector<int> v { 1, 2, 3 };
2 std::copy(v.begin(), v.end(),
3           std::ostream_iterator<int>{ std::cout, " " });
```

`std::ostream_iterator` tar inte bara emot en ström när man skapar den, utan den tar även emot en sträng som specificerar vad som ska skrivas ut *efter* varje utskrivet värde.

Modifierande algoritmer

Uppgift #3

Implementera ett program som gör följande:

1. Läs in strängar från `std::cin` till en `std::vector`, användaren trycker `ctrl+D` när denne är klar.
2. Sortera orden
3. Ta bort alla dubletter ur vektorn
4. Skriv ut orden som finns i vektorn till `std::cout`, separerade med mellanslag.

Alla dessa steg kan göras med STL algoritmer och iteratorer.

Modifierande algoritmer

Lösning Uppgift #3

```
1  #include <algorithm>
2  #include <iostream>
3  #include <iterator>
4  #include <string>
5  #include <vector>
6
7  int main()
8  {
9    std::vector<std::string> words {
10     std::istream_iterator<std::string>{ std::cin },
11     std::istream_iterator<std::string>{ }
12   };
13
14   std::sort(words.begin(), words.end());
15   words.erase(std::unique(words.begin(), words.end()), words.end());
16
17   auto osit = std::ostream_iterator<std::string> { std::cout, " " };
18   std::copy(words.begin(), words.end(), osit);
19 }
```

Modifierande algoritmer

Diskussion Uppgift #3

- Här löser vi varje steg m.h.a. algoritmer
- I första steget läser vi in från ström-iteratorer direkt till vektorn. Notera alltså att `std::vector` har en konstruktor som tar emot två *godtyckliga* iteratorer och kopierar sekvensen direkt in i vektorn.
- I andra steget använder vi `std::sort`
- I tredje steget använder vi `std::unique`, som inte tar bort några element utan lägger alla dubletter längst bak och returnerar en iterator till den första dubletten. Detta tillåter oss sedan att använda `.erase()` för att ta bort alla dubletter.
- I sista steget kopierar vi innehållet i `words` till `std::cout` genom att använda `std::copy` och en ström-iterator.

1 Upplägg av storseminarium
2 Behållare
3 Algoritmer
4 Modifierande algoritmer
5 Lambdafunktioner
6 Övningsuppgifter

LINKÖPING UNIVERSITY

lambdafunktioner

`std::transform`

```
1  int add_2(int x)
2  {
3    return x + 2;
4  }
5
6  int main()
7  {
8    std::vector<int> u { 1, 2, 3 };
9    std::vector<int> v(u.size());
10
11   std::transform(u.begin(), u.end(), v.begin(), add_2);
12 }
```

lambdafunktioner

`std::transform`

- `std::transform` är en algoritm som är väldigt snarlik `std::copy`, men innan den kopierar ett värde så skickar den värdet till en funktion (som specificeras vid anropet av `std::transform`) och kopierar istället returvärdet av den funktionen.
- Detta kan vi se någorlunda tydligt i exemplet på föregående slide, där för först specificerar vilket intervall vi vill kopiera (`u.begin()` till `u.end()`), vart vi vill kopiera det till (`v.begin()`) och till sist vilken funktion vi vill anropa för varje värde (`add_2`).
- Det som är viktigt att notera här är att vi skickar *själva funktionen* `add_2` till `std::transform`. D.v.s.: vi anropar aldrig `add_2`, utan det görs inuti `std::transform` (en gång per värde).

lambdafunktioner

`std::transform`

- Det som är tråkigt med hur vi använde `std::transform` i föregående exempel att man måste definiera en funktion som vi sedan kan skicka till `std::transform`.
- Detta innebär att läsaren av koden måste skrolla till definitionen av `add_2` för att förstå exakt vad den funktionen gör och sedan skrolla tillbaka till stället där `std::transform` används.
- Kan vi skapa funktionen direkt i `std::transform` på något vis?
- Svaret är ja! Med något som kallas *lambdafunktioner*!

lambdafunktioner

lambdafunktioner

```
1  std::vector<int> u { 1, 2, 3 };
2  std::vector<int> v(u.size());
3
4  std::transform(u.begin(), u.end(), v.begin(),
5                 [](int x)
6                 {
7                   return x + 2;
8                 });
```

Här ser vi ett exempel på hur vi kan definiera en s.k. lambdafunktion direkt i `std::transform` anropet.

lambdafunktioner

lambdafunktioner

```
1  std::vector<int> u { 1, 2, 3 };
2  std::vector<int> v(u.size());
3
4  std::transform(u.begin(), u.end(), v.begin(),
5                 [](int x)
6  lambdafunktion {
7                   return x + 2;
8                 });
```

Detta uttryck skapar en funktion utan namn (en *lambdafunktion*) som sedan skickas till `std::transform`. Algoritmen kommer att anropa denna namnlösa funktion för varje element. Notera att denna lambdafunktion inte går att använda igen eftersom att den är *namnlös*. Så lambdafunktioner är *temporära*.

lambdafunktioner

lambdafunktioner

```
1  std::vector<int> u { 1, 2, 3 };
2  std::vector<int> v(u.size());
3
4  std::transform(u.begin(), u.end(), v.begin(),
5                 [](int x)
6                 {
7                   return x + 2;
8                 });
```

omfång (capture)

lambdafunktioner är syntaktiskt väldigt lika vanliga funktioner, med två viktiga skillnader: för lambdafunktioner anger vi ingen returtyp, och istället för ett namn skriver vi [ ] (detta kallas lambdafunktionens *omfång* (*capture* på engelska), mer om det strax).

lambdafunktioner

lambdafunktioner

```
1  std::vector<int> u { 1, 2, 3 };
2  std::vector<int> v(u.size());
3
4  std::transform(u.begin(), u.end(), v.begin(),
5                 [](int x) Parametrar
6                 {
7                   return x + 2;
8                 });
```

Utöver de skillnaderna ser vi att inparametrar definieras på samma sätt som för vanliga funktioner...

lambdafunktioner

lambdafunktioner

```
1  std::vector<int> u { 1, 2, 3 };
2  std::vector<int> v(u.size());
3
4  std::transform(u.begin(), u.end(), v.begin(),
5                 [](int x)
6                 {
7                   return x + 2;
8                 });
```

Funktionskropp

... och funktionskroppen definieras på exakt samma sätt som för vanliga funktioner.

lambdafunktioner

Lite begränsningar

- lambdafunktioner fungerar likt vanliga funktioner, men de definieras direkt i ett scope, snarare än i toppnivån av källkodsfilen.
- Detta kan leda till att man luras till att tro att lambdafunktionen har tillgång till alla variabler som finns deklarerade i det scope som lambdafunktionen definieras i. Men så är inte fallet, för lambda funktionen anropas potentiellt i ett annat scope, och dess definition ligger utanför scope:t.
- Så om man vill skicka med information till lambda funktionen från omkringliggande scope kan man göra det på två sätt.
- Man kan lägga till en parameter till lambda funktionen som innehåller den data man vill tillhandahålla. Men detta fungerar inte alltid eftersom att det är den som *anropar* funktionen som anger parametrarna, och i exemplet på föregående slide så är det `std::transform` som *anropar* funktionen. `std::transform` kommer *endast* skicka med en parameter till lambda funktionen, och du kan inte ändra det.
- Så detta lämnar oss bara med ett alternativ: nämligen att använda oss av lambdafunktionens omfång.
- Men innan vi går in på det tittar vi på ett exempel som illustrerar vad vi vill göra.

lambdafunktioner

lambdafunktioners omfång

```
1  std::vector<int> v { 1, 2, 3 };
2
3  int n { };
4  std::cin >> n;
5
6  if (n == 1)
7  {
8    // notera att std::transform kan skriva över de gamla värdena
9    // om vi anger början av v som den tredje parametern.
10   std::transform(v.begin(), v.end(), v.begin(),
11                  [](int x) { return x + 1; });
12 }
13 else if (n == 2)
14 {
15   std::transform(v.begin(), v.end(), v.begin(),
16                  [](int x) { return x + 2; });
17 }
18 // ...
```

lambdafunktioner

lambdafunktioners omfång

- I exemplet ser vi att användaren matar in `n` som vi sedan vill addera till alla tal.
- En naturlig tanke många får i detta läge är att skicka med `n` som en parameter till lambda funktionen, typ såhär:

```
1 std::transform(v.begin(), v.end(), v.begin(),
2                [](int x, int n) { return x + n; });
```

  men det kommer inte fungera, för att vi har inget sätt att säga åt `std::transform` att den behöver skicka med `n` till lambdafunktionen.
- Vi skulle behöva skicka med `n` när lambdat *skapas* istället för när det anropas.
- Detta är *exakt* vad lambdafunktionens omfång är till för!

lambdafunktioner

lambdafunktioners omfång

```
1  std::vector<int> v { 1, 2, 3 };
2
3  int n { };
4  std::cin >> n;
5
6  std::transform(v.begin(), v.end(), v.begin(),
7                 [n](int x) { return x + n; });
```

lambdafunktioner

lambdafunktioners omfång

```
1  std::vector<int> v { 1, 2, 3 };
2
3  int n { };
4  std::cin >> n;
5
6  std::transform(v.begin(), v.end(), v.begin(),
7                 [n](int x) { return x + n; });
```

Innanför hakparanteserna (d.v.s. innanför `[ ]`) kan man ange variabler som man vill föra över från nuvarande scope till lambdafunktionen.

lambdafunktioner

lambdafunktioners omfång

```
1  std::vector<int> v { 1, 2, 3 };
2
3  int n { };
4  std::cin >> n;
5
6  std::transform(v.begin(), v.end(), v.begin(),
7                 [n](int x) { return x + n; });
```

Man kan se omfånget som ett sätt att skicka med samma parameter till *varje* anrop av lambdafunktionen, d.v.s. att `n` kommer finnas tillgänglig varje gång `std::transform` anropar lambdafunktionen utan att `std::transform` behöver göra något annorlunda.

Lambdafunktioner

Mer om omfång

```
1  int n { 2 };
2  std::vector<int> u { 1, 2, 3 };
3  std::vector<int> v(u.size());
4
5  auto add_n = [n](int x) { return x + n; };
6
7  // resultat = { 3, 4, 5 }
8  std::transform(u.begin(), u.end(), v.begin(), add_n);
9  n = 3;
10
11 // resultat = { 3, 4, 5 } ???
12 std::transform(u.begin(), u.end(), v.begin(), add_n);
```

Lambdafunktioner

Me

Vi kan ge lambda funktioner namn i det scope vi definierar den i genom att skapa en variabel m.h.a. `auto` och sedan initiera variabeln med lambdafunktionen. På detta sätt kan vi återanvända samma lambdafunktioner flera gånger i samma scope.

```
1  int n { 2 };
2  std::vector<int> u { 1, 2, 3 };
3  std::vector<int> v(u.size());
4
5  auto add_n = [n](int x) { return x + n; };
6
7  // resultat = { 3, 4, 5 }
8  std::transform(u.begin(), u.end(), v.begin(), add_n);
9  n = 3;
10
11 // resultat = { 3, 4, 5 } ???
12 std::transform(u.begin(), u.end(), v.begin(), add_n);
```

Lambdafunktioner

Me

Nu använder vi vår definierade `add_n` för att ta alla värden från `u` och lägger på värdet av `n` (vilken är definierad som 2) och sparar resultatet i `u`, vilket ger oss det förväntade resultatet `{ 3, 4, 5 }`.

```
1  int n { 2 };
2  std::vector<int> u { 1, 2, 3 };
3  std::vector<int> v(u.size());
4
5  auto add_n = [n](int x) { return x + n; };
6
7  // resultat = { 3, 4, 5 }
8  std::transform(u.begin(), u.end(), v.begin(), add_n);
9  n = 3;
10
11 // resultat = { 3, 4, 5 } ???
12 std::transform(u.begin(), u.end(), v.begin(), add_n);
```

Lambdafunktioner

Me

Sedan försöker vi ändra n för att istället lägga på 3 på varje element i u och spara resultatet i v.

```
1  int n { 2 };
2  std::vector<int> u { 1, 2, 3 };
3  std::vector<int> v(u.size());
4
5  auto add_n = [n](int x) { return x + n; };
6
7  // resultat = { 3, 4, 5 }
8  std::transform(u.begin(), u.end(), v.begin(), add_n);
9  n = 3;
10
11 // resultat = { 3, 4, 5 } ???
12 std::transform(u.begin(), u.end(), v.begin(), add_n);
```

Lambdafunktioner

Mer om omfång

Men detta resulterar i exakt samma resultat som innan?! Varför?

```
1  int n { 2 };
2  std::vector<int> u { 1, 2, 3 };
3  std::vector<int> v(u.size());
4
5  auto add_n = [n](int x) { return x + n; };
6
7  // resultat = { 3, 4, 5 }
8  std::transform(u.begin(), u.end(), v.begin(), add_n);
9  n = 3;
10
11 // resultat = { 3, 4, 5 } ???
12 std::transform(u.begin(), u.end(), v.begin(), add_n);
```

Lambdafunktioner

Mer om omfång

- Kom ihåg att lambdafunktionen har egentligen inte alls koll på vilket scope den definierades i.
- Så när vi fångar en variabel till lambdafunktionen så skapas det en *kopia* av den variabeln in i lambdafunktionen.
- Detta innebär att även fast vi ändrar `n` i vårt scope, så ser inte lambdafunktionen den ändringen, för den har en egen version av variabeln.
- Vi löser detta genom att fånga en *referens* istället!

Lambdafunktioner

Fånga referenser

```
1  int n { 2 };
2  std::vector<int> u { 1, 2, 3 };
3  std::vector<int> v(u.size());
4
5  auto add_n = [&n](int x) { return x + n; };
6
7  // resultat = { 3, 4, 5 }
8  std::transform(u.begin(), u.end(), v.begin(), add_n);
9  n = 3;
10
11 // resultat = { 4, 5, 6 }
12 std::transform(u.begin(), u.end(), v.begin(), add_n);
```

Lambdafunktioner

Fån

Om vi lägger till ett & framför variabeln så fångar vi den som en *referens*, vilket innebär att lambda funktionen har direkt tillgång till variabeln n, och därför fungerar det vi försökte göra i förra exemplet.

```
1  int n { 2 };
2  std::vector<int> u { 1, 2, 3 };
3  std::vector<int> v(u.size());
4
5  auto add_n = [&n](int x) { return x + n; };
6
7  // resultat = { 3, 4, 5 }
8  std::transform(u.begin(), u.end(), v.begin(), add_n);
9  n = 3;
10
11 // resultat = { 4, 5, 6 }
12 std::transform(u.begin(), u.end(), v.begin(), add_n);
```

Lambdafunktioner

Fånga flera variabler

```
1  int x { };
2  int y { };
3
4  auto my_lambda = [x, &y](int a, int b)
5  {
6    int c { a + b };
7    int z { x + y };
8    return a*x + b*y + c*z;
9  };
```

Lambdafunktioner

Fånga flera variabler

- Man kan fånga flera variabler i lambdafunktioners omfång genom att komma-separera alla variabler man är intresserad av.
- Man kan dessutom välja att fånga dem som referens eller kopia. Detta är ett val man gör för varje variabel individuellt.
- Dessutom belyser detta exempel att lambdafunktioner inte är begränsade till en parameter, utan du kan ha så många du behöver (sålänge de stämmer överens med mängden parametrar som algoritmen behöver).
- Notera också att funktionskroppen för lambdafunktioner fungerar precis som för vanliga funktioner, så du kan ha flera satser, du kan skapa variabel o.s.v. precis som i vanliga funktioner.

Lambdafunktioner

Uppgift #4

Skriv en funktion som läser in ett heltal `n` från `std::cin` samt en godtycklig mängd heltal från `std::cin` till en `std::vector`.
Sortera sedan alla heltal i vektorn baserat på deras absoluta avstånd till `n` (använd `std::abs` från `<cmath>`).
Sist ska du skriva ut hela listan till `std::cout`, ett värde per rad.

Lambdafunktioner

Lösning Uppgift #4

```
1  #include <algorithm>
2  #include <iostream>
3  #include <iterator>
4  #include <vector>
5  int main()
6  {
7    int n { };
8    std::cin >> n;
9
10   std::vector<int> values { std::istream_iterator<int>{ std::cin },
11                             std::istream_iterator<int>{ } };
12
13   std::sort(values.begin(), values.end(),
14             [&n](int a, int b)
15             { return std::abs(a - n) < std::abs(b - n); });
16
17   std::copy(values.begin(), values.end(),
18             std::ostream_iterator<int>{ std::cout, "\n" });
19 }
```

1 Upplägg av storseminarium
2 Behållare
3 Algoritmer
4 Modifierande algoritmer
5 Lambdafunktioner
6 Övningsuppgifter

LINKÖPING UNIVERSITY

Övningsuppgifter

Uppgift #5

Skriv ett program som läser in strängar från `std::cin` och räknar hur många gånger varje *unik* sträng förekom.
Skriv sedan ut hur många gånger varje unikt ord förekom till `std::cout`. Orden ska skrivas ut i *sorterad* ordning.
Tips: Välj en lämplig behållare för att hålla koll på antalet förekomster.

Övningsuppgifter

Lösning Uppgift #5

```
1  #include <iostream>
2  #include <map>
3  #include <string>
4  int main()
5  {
6    std::map<std::string, int> frequency { };
7
8    std::string word { };
9    while (std::cin >> word)
10   {
11     ++frequency[word];
12   }
13
14   for (auto&& [word, count] : frequency)
15   {
16     std::cout << word << " förekom " << count
17               << " gånger!" << std::endl;
18   }
19 }
```

Övningsuppgifter

Uppgift #6

Läs in heltal från `std::cin` till en vektor.
Kontrollera om alla tal i vektorn är jämna. Skriv ut strängen `"Alla är jämna!"` om alla tal faktiskt var jämna, annars skriver du ut `"Det finns ojämna..."`.
Notera: Detta är en övning i att hitta algoritmer samt att använda lambdafunktioner.

Övningsuppgifter

Lösning Uppgift #6

```
1  #include <algorithm>
2  #include <iostream>
3  #include <iterator>
4  #include <vector>
5  int main()
6  {
7    std::vector<int> values { std::istream_iterator<int>{ std::cin },
8                              std::istream_iterator<int>{ } };
9
10   bool all_even = std::all_of(values.begin(), values.end(),
11                               [](int x) { return i % 2 == 0; });
12
13   if (all_even) {
14     std::cout << "Alla är jämna!" << std::endl;
15   } else {
16     std::cout << "Det finns ojämna..." << std::endl;
17   }
18 }
```

Övningsuppgifter

Uppgift #7

Läs in heltal från `std::cin` till en vektor.
Sortera sedan alla tal i vektorn som ligger mellan det första negativa talet och slutet (i fallande ordning).
Skriv till sist ut alla tal till `std::cout`, ett tal per rad.

Övningsuppgifter

Lösning Uppgift #7

```
1  #include <algorithm>
2  #include <iostream>
3  #include <iterator>
4  #include <vector>
5  int main()
6  {
7    std::vector<int> values { std::istream_iterator<int>{ std::cin },
8                              std::istream_iterator<int>{ } };
9
10   auto it = std::find_if(values.begin(), values.end(),
11                          [](int x) { return x < 0; });
12
13   std::sort(it, v.end(),  [](int a, int b) { return a > b; });
14
15   std::copy(v.begin(), v.end(),
16             std::ostream_iterator<int>{ std::cout, "\n" });
17 }
```

www.liu.se

LINKÖPING UNIVERSITY