TDDC76

Operatorer, upprepning, funktioner, mm

Eric Ekström & Klas Arvidsson

Institutionen för datavetenskap

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Allmän info

- Kom ihåg att anmäla er i WebReg
- Det som benämns “LAB2” i WebReg och Ladok motsvarar hela labserien

Namespace

Måste vi skriva `std::` varje gång?

```
#include <iostream>
#include <iomanip>



int main()
{
  std::cout << std::setw(6)
       << std::setfill('.')
       << 10 << std::endl;
  return 0;
}
```

Namespace

Måste vi skriva `std::` varje gång?

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
  cout << setw(6)
       << setfill('.')
       << 10 << endl;
  return 0;
}
```

Namespace

- Samla funktioner och variabler under ett namn
- Kan särskilja mellan grupper som har olika namn på variabler och funktioner
- `using namespace [namn];` använder vi inte då det finns en krock mellan namn

Flera strömmar

- En ström kopplas till en datakälla vi kan hämta tecken från eller ett datamål vi kan skicka tecken till
- Tecken kommer i en viss ordning som inte kan ändras
- Vi mellanlagrar tecken i strömmen för att kunna hantera dessa effektivare
- Fungerar lika oberoende av datakälla/datamål

Flera strömmar

Exempel på andra typer av strömmar vi kan använda.

```
#include <iostream>
istream cin;                        // Tangentbord som källa
iostream cout;                      // Tangentbord som mål

#include <fstream>
ifstream infile  {"config.txt"};// En fil som källa
ofstream outfile {"save.txt"};  // En fil som mål

#include <sstream>
istringstream iss {teststring}; // kopia av en sträng som källa
ostringstream oss {};           // samlar all data i en sträng
oss.str();                      // Ger den sparade strängen
```

Fler strömmar

```
#include <iostream>
#include <fstream>

int main()
{
  std::string line {};
  std::ifstream ifs {"fil.txt"};

  while ( std::getline(ifs, line) )
  {
    std::cout << line << std::endl;
  }

  ifs.close();
}
```

Fler strömmar

Inläsning av flera data per rad.

```
std::string line {};

while ( std::getline(ifs, line) )
{
  istringstream iss {line};

  string name {};
  int age {};

  if ( iss >> name >> age )
  {
    std::cout << name << " är " age << std::endl;
  }
}

ifs.close();
```

Operatorer

Vilka operatorer finns i c++?

Aritmetiska operatorer	`+ - * / % ++ --`
Jämförelseoperatorer	`< > <= >= == !=`
Logiska operatorer	`|| && !`
Strömoperatorer	`<< >>`
Tilldelning	`= += -= *= /= %=`
Minnesoperatorer	`* & ->`
Bitoperatorer	`<< >> ~ ^ & |`

Operatorer

- Aritmetiska operatorer fungerar som vanligt. Beräkningsordning som i matematiken. Kan ändras med parenteser.
- `operator%` ger resten vid en division.
- Datatypen styr vad som görs! Heltal delat på heltal ger heltalsdivision.

```
a = 4 + 2;

x = 7 / 3; // x = 2
y = 7 % 3; // y = 1
```

Operatorer

- Många operatorer har en kombinerad operator med tilldelning.
- Öka eller minska med ett går att göra med `operator++` och `operator--`. Dessa finns i två varianter: prefix och postfix.

```
x = x + 7;
x += 7;

int y {};
y++;
++y;
y++;
--y;
```

Operatorer

Vad är skillnaden mellan prefix och postfix?

```
int x { 1 };
std::cout << x    << std::endl;
std::cout << ++x << std::endl;
std::cout << x    << std::endl;

std::cout << x    << std::endl;
std::cout << x++ << std::endl;
std::cout << x    << std::endl
```

```

1
2
2

2
2
3
```

Postfix motsvarar alltså uttrycket

```
int ret {x};
x = x + 1;
```

där `ret` är returvärdet av operatorn.

Operatorer

Jämförelseoperatorerna ger ett sanningsvärde som svar.

- Hur många av de 6 operatorerna kan du skapa som ett uttryck av de andra?
- Ex. `x < y == y > x`

De logiska operatorerna ger också sanningsvärde som svar.

- || - logisk OR
- && - logisk AND
- ! - logisk NOT

Operatorer

- Operatorer i sammansatta uttryck utvärderas i bestämd ordning.
- Om du inte intuitivt vet ordningen, använd parenteser!
- Kolla operator precedence på cppreference!

```
int x { 1 +  2 * 3  - 4 };
int y { 1 + (2 * 3) - 4 };

bool b { true ||  false && true  };
bool b { true || (false && true) };
```

Operatorer

- Short-circuit evaluation används för AND och OR.
- Ide: Utvärdera första uttrycket och avgör om andra uttrycket behöver utvärderas.

```
int x {1};
if (x < 5 || x++)
{
  //Instruktioner
}
```

```
int x {1};
if (x >= 5 && x >= 3)
{
  //Instruktioner
}
```

Scope

- Ett program består av satser (instruktioner)
- Satser som omsluts av {} ligger i ett block

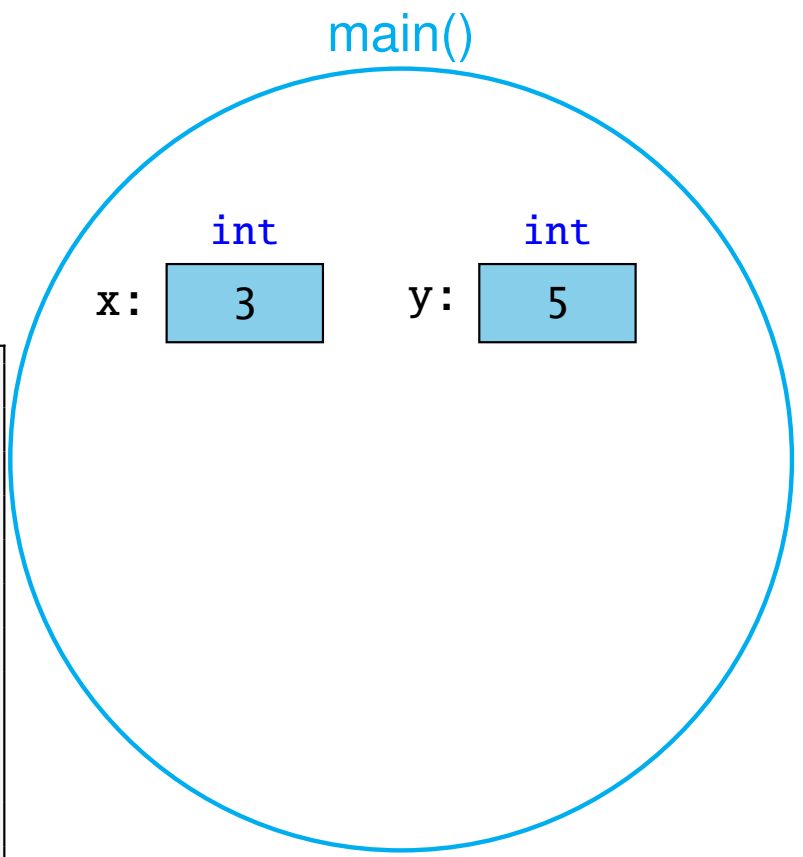
```
//Utanför blocket

if ( villkor )
{
  // Inne i blocket
}

// Också utanför blocket
```

- Alla deklarationer (t.ex. variabler) har ett scope
- Variabelns scope styrs av var den är deklarerad.

Scope

- Vi kan skapa block var som i koden
- De deklarationer som görs inom ett block gäller endast där
- Vi säger att variabelns scope är det blocket

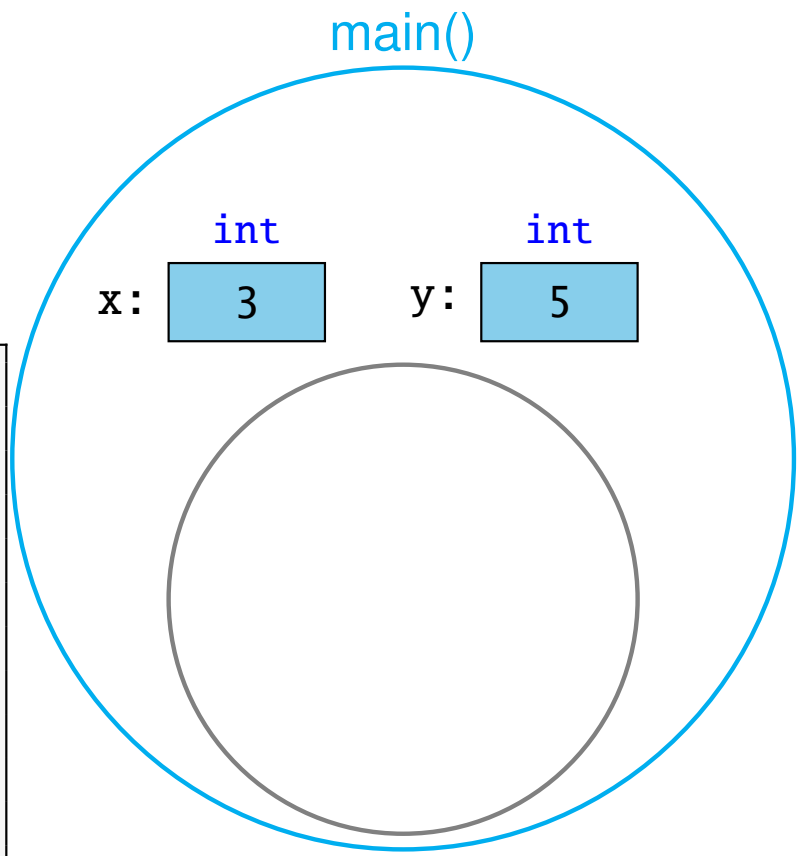
```
#include <iostream>

int main()
{
  int x {3}, y {5};
  {
    int x {4};
    int z {};
    cout << x << ' ' << y << endl;
  }
    z = 6; // Fel, z finns inte här
}
```



Scope

- Vi kan skapa block var som i koden
- De deklarationer som görs inom ett block gäller endast där
- Vi säger att variabelns scope är det blocket

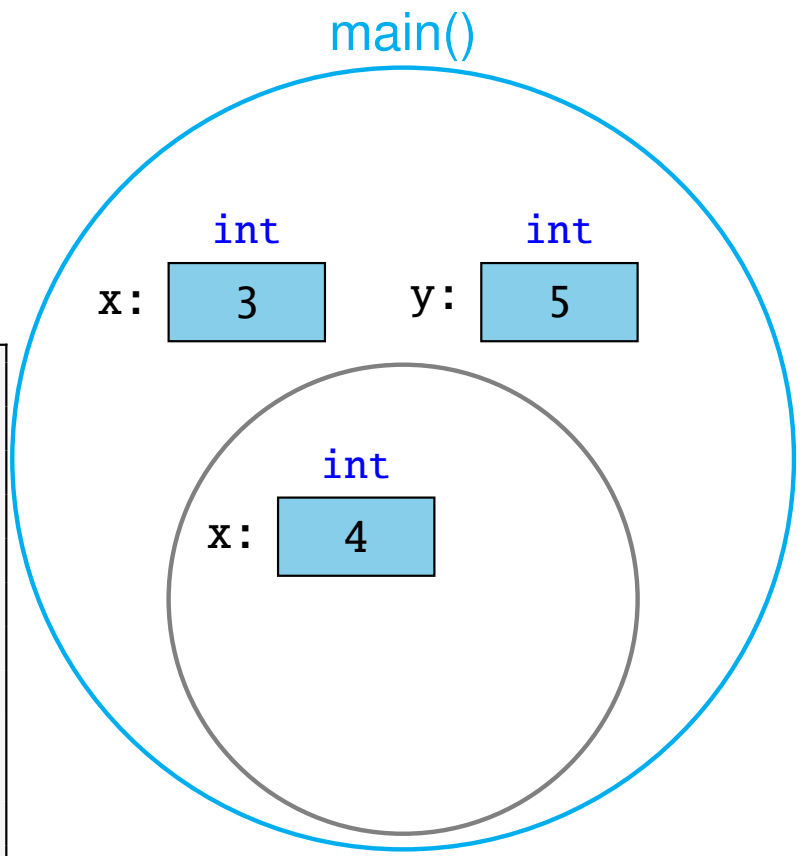
```
#include <iostream>

int main()
{
  int x {3}, y {5};
  {
    int x {4};
    int z {};
    cout << x << ' ' << y << endl;
  }
    z = 6; // Fel, z finns inte här
}
```



Scope

- Vi kan skapa block var som i koden
- De deklarationer som görs inom ett block gäller endast där
- Vi säger att variabelns scope är det blocket

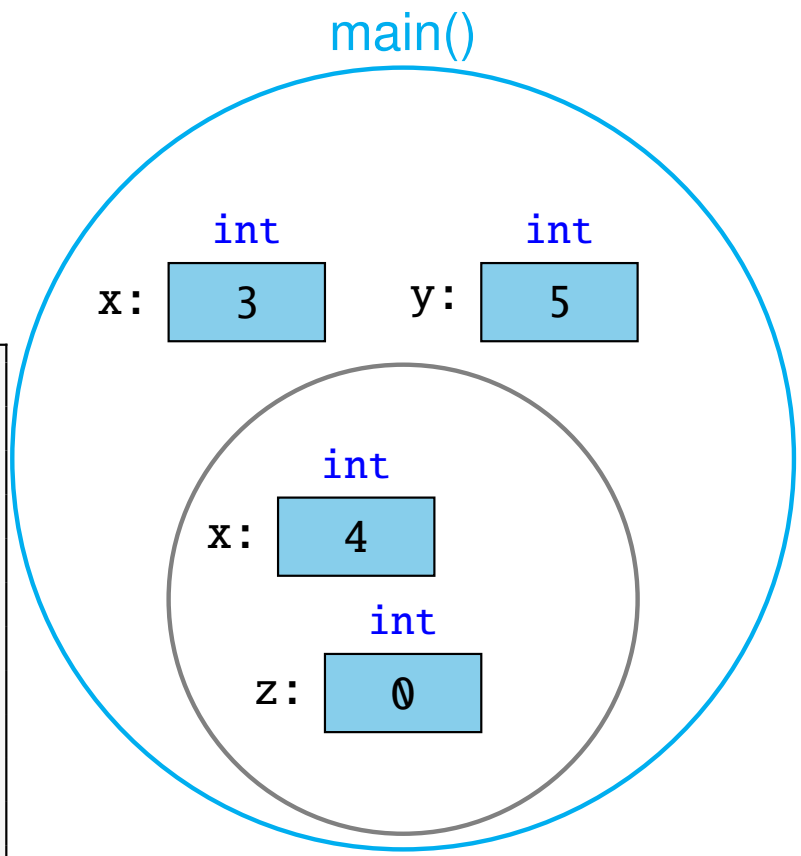
```
#include <iostream>

int main()
{
  int x {3}, y {5};
  {
    int x {4};
    int z {};
    cout << x << ' ' << y << endl;
  }
    z = 6; // Fel, z finns inte här
}
```



Scope

- Vi kan skapa block var som i koden
- De deklarationer som görs inom ett block gäller endast där
- Vi säger att variabelns scope är det blocket

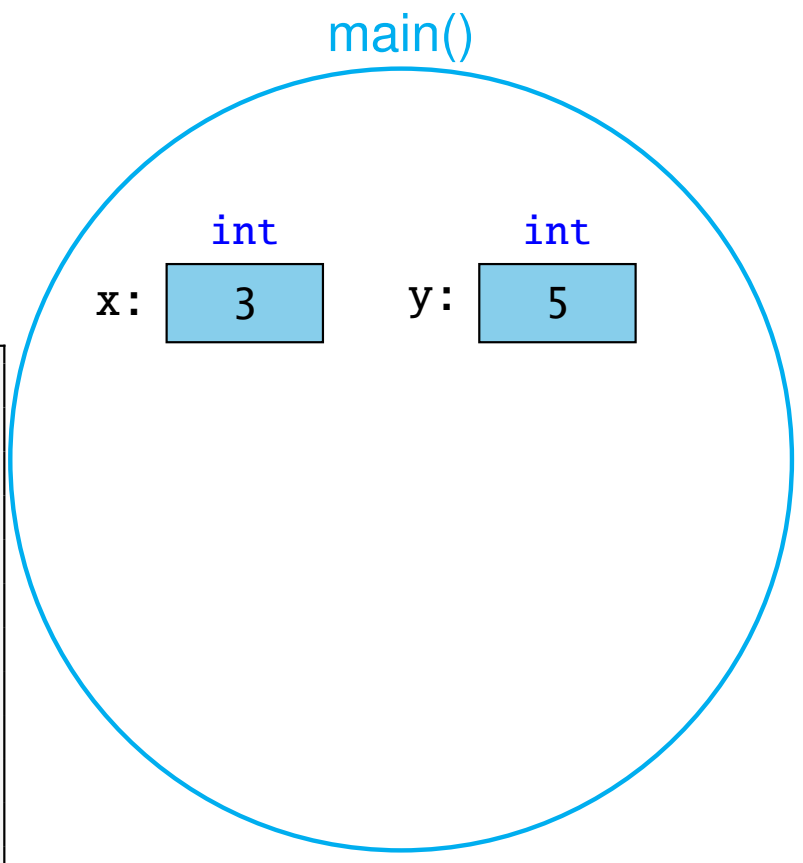
```
#include <iostream>

int main()
{
  int x {3}, y {5};
  {
    int x {4};
    int z {};
    cout << x << ' ' << y << endl;
  }
    z = 6; // Fel, z finns inte här
}
```



Scope

- Vi kan skapa block var som i koden
- De deklarationer som görs inom ett block gäller endast där
- Vi säger att variabelns scope är det blocket

```
#include <iostream>

int main()
{
  int x {3}, y {5};
  {
    int x {4};
    int z {};
    cout << x << ' ' << y << endl;
  }
    z = 6; // Fel, z finns inte här
}
```



Funktioner

- En funktion är en samling instruktioner. Dvs ett block med ett namn
- Kan anropas för att utföra funktionens instruktioner
- Vi behöver inte veta exakt vilka instruktioner funktionen består av för att använda den. Det räcker att veta funktionens deklaration.
- Ska ha ett exakt syfte och inga sidoeffekter

```
[returtyp] funktion_namn([parametrar])
{
  // Instruktioner
  [möjlig retursats]
}
```

Funktioner

```cpp
#include <iostream>

using namespace std;

void hello_world(); //Deklaration

int main()
{
  hello_wolrd();   // Anrop
  cout << "-------" << endl;
  hello_world();   // Anrop
}

void hello_world() // Definition
{
  cout << "Hello world!" << endl;
}
```

```
Hello world!
-------
Hello World!
```

Funktioner

Vad händer om man glömmer returnera?

```
int slarver()
{
  // return 9;
}
int main()
{
  int x { slarver() };
}
```

Funktioner

Vad händer om man glömmer returnera?

```
int slarver()
{
  // return 9;
}
int main()
{
  int x { slarver() };
}
```

Komplettering på labben! Men kompilatorn kan rädda oss!

```
  g++ -std=c++17 -Wall -Wextra -Weffc++ slarver.cc

  slarver.cc: In function 'int slarver()':
  slarver.cc:4:1 warning: no return statement in function
        returning non-void [-Wreturn-type]
```

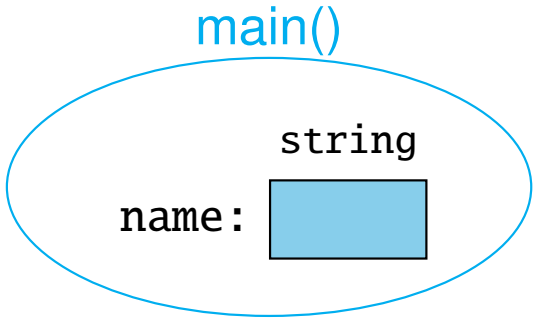
Funktioner

- Funktioner kan ha flera retursatser. Vid `return` avslutas funktion.
- En funktion som inte behöver ett returvärde har returtypen void.

Funktioner

```
int main()
{
  string name {};
  name = get_username();
  return 0;
}
```

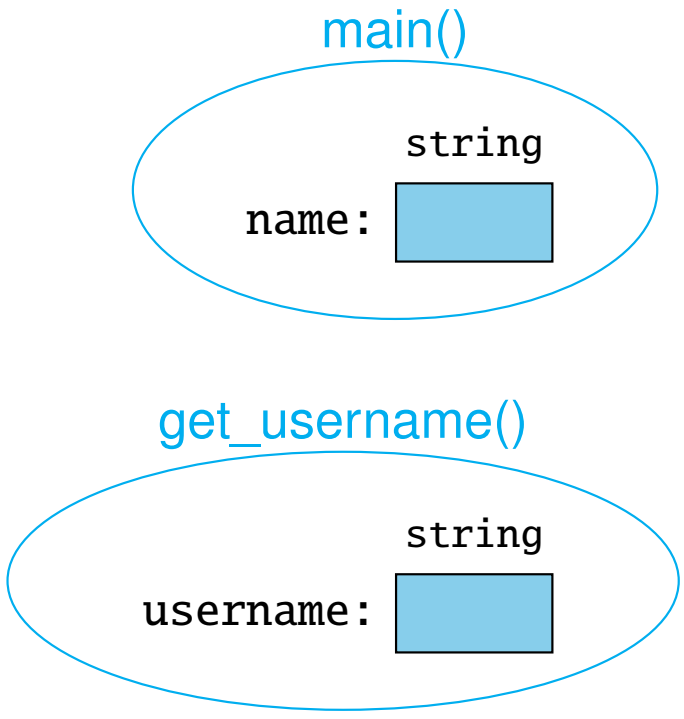
```
string get_username()
{
  string username {};
  cout << "Skriv namn: ";
  cin >> username;
  return username;
}
```



Funktioner

```
int main()
{
  string name {};
  name = get_username();
  return 0;
}
```

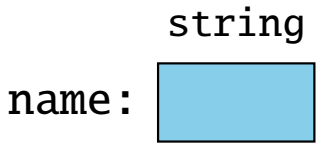
```
string get_username()
{
  string username {};
  cout << "Skriv namn: ";
  cin >> username;
  return username;
}
```

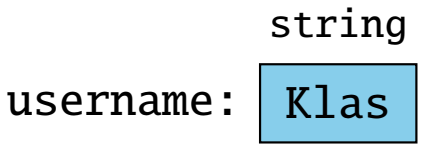


Funktioner

```
int main()
{
  string name {};
  name = get_username();
  return 0;
}
```

```
string get_username()
{
  string username {};
  cout << "Skriv namn: ";
  cin >> username;
  return username;
}
```

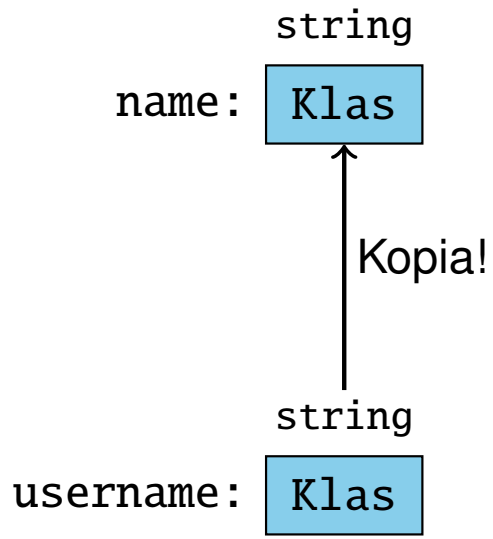




Funktioner

```
int main()
{
  string name {};
  name = get_username();
  return 0;
}
```

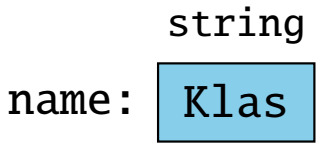
```
string get_username()
{
  string username {};
  cout << "Skriv namn: ";
  cin >> username;
  return username;
}
```



Funktioner

```
int main()
{
  string name {};
  name = get_username();
  return 0;
}
```

```
string get_username()
{
  string username {};
  cout << "Skriv namn: ";
  cin >> username;
  return username;
}
```



Funktioner - parameteröverföring

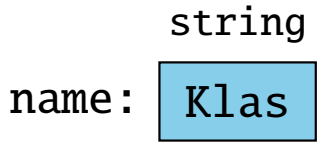
Hur skickar vi data till en funktion?

- Anroparen skickar in ett argument (en variabel eller värde)
- Funktionen tar emot argumentet i en parameter (en variabel)
- flera argument kan skickas till funktionen. Ordningen avgör vilken parameter de sparas i.
- Argument kan överföras på flera sätt
  - kopia (som får ändras, eller inte ändras)
  - referens
  - referens som inte får ändras

Funktioner - parameteröverföring

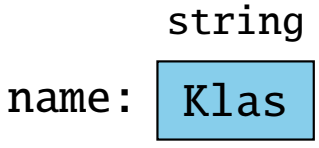
```
int main()
{
  string name { "Klas" };
  print(name);
  return 0;
}
```

```
string print(string firstname)
{
  cout << "Jag heter "
       << firstname
       << end;
}
```

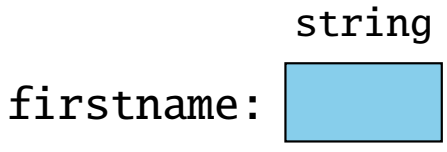


Funktioner - parameteröverföring

```
int main()
{
  string name { "Klas" };
  print(name);
  return 0;
}
```



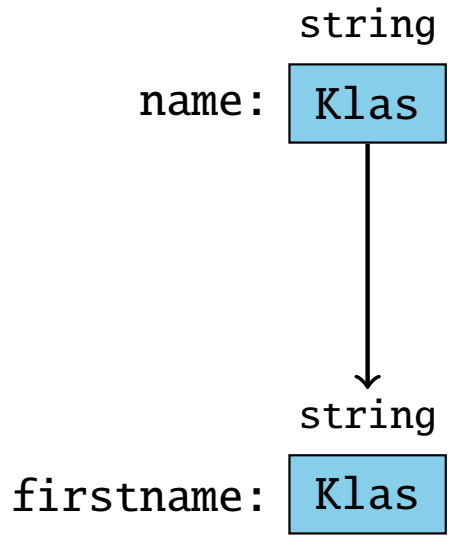

```
string print(string firstname)
{
  cout << "Jag heter "
       << firstname
       << end;
}
```



Funktioner - parameteröverföring

```
int main()
{
  string name { "Klas" };
  print(name);
  return 0;
}
```

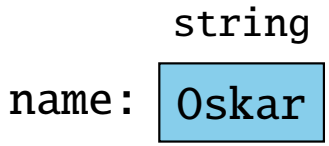
```
string print(string firstname)
{
  cout << "Jag heter "
       << firstname
       << end;
}
```



Funktioner - parameteröverföring

```
int main()
{
  string name { "Oskar" };
  print(name);
  return 0;
}
```

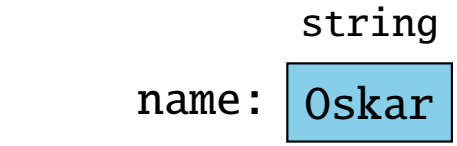
```
string print(string const firstname)
{
  cout << "Jag heter "
       << firstname
       << end;

}
```

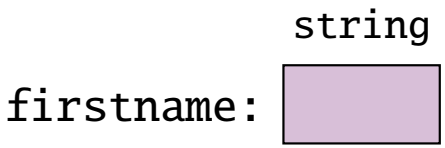


Funktioner - parameteröverföring

```
int main()
{
  string name { "Oskar" };
  print(name);
  return 0;
}
```

```
string print(string const firstname)
{
  cout << "Jag heter "
       << firstname
       << end;

}
```

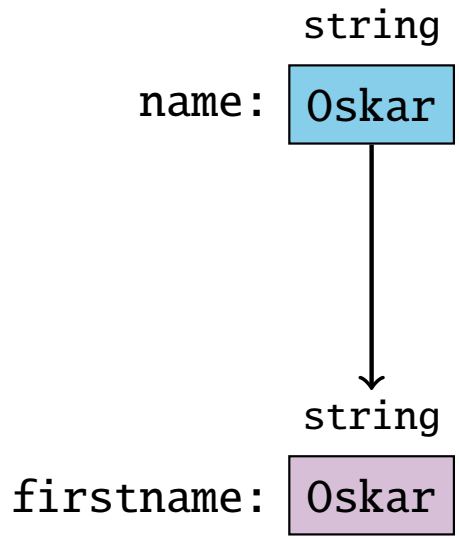




Funktioner - parameteröverföring

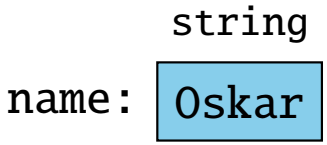
```
int main()
{
  string name { "Oskar" };
  print(name);
  return 0;
}
```

```
string print(string const firstname)
{
  cout << "Jag heter "
       << firstname
       << end;

}
```



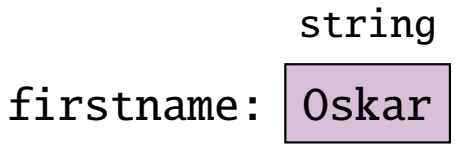
Funktioner - parameteröverföring

```
int main()
{
  string name { "Oskar" };
  print(name);
  return 0;
}
```




```
string print(string const firstname)
{
  cout << "Jag heter "
       << firstname
       << end;
  firstname = "Eric";
}
```

KOMPILERAR EJ!



Referenser

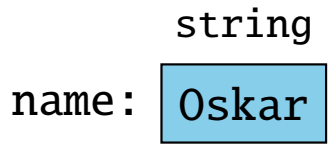
- Alla variabler är sparade någonstans
- Istället för att kopiera värdet ger vi direkt tillgång till värdet
- Minnesplatsen får två namn
- Kan referera till variabler i ett annat scope
  - Var försiktig, variabeln vi refererar till kan gå ur sitt scope!

```
int main()
{
  int x {5};
  int& y {x};
  x = 7;
  cout << y; // ???
}
```

Funktioner - parameteröverföring

```
int main()
{
  string name { "Klas" };
  print(name);
  return 0;
}
```

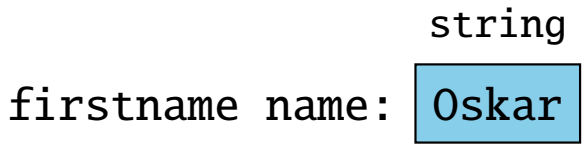
```
string print(string const & firstname)
{
  cout << "Jag heter "
       << firstname
       << end;
}
```



Funktioner - parameteröverföring

```
int main()
{
  string name { "Klas" };
  print(name);
  return 0;
}
```

```
string print(string const & firstname)
{
  cout << "Jag heter "
       << firstname
       << end;
}
```



Funktioner

Vilken typ av parameteröverföring ska jag välja?

1. Vill jag ändra på argumentet?
   -> Referens

VIKTIGT!!

2. Är det en inbyggd datatyp? (int, float, bool, etc.)
   -> Kopia
3. Är det en sammansatt datatyp? (string, vector, etc.)
   -> Konstant referens

Vektorer

- En vektor lagrar en sekvens av värden
- Vi måste ange datatypen för värden som ska lagras i sekvensen
- Alla värden har samma datatyp.

```
#include <vector>

int main()
{
  std::vector<int> v { 1, 5, 2 };
  std::cout << v.at(0) << v.at(1) << v.at(2) << std::endl;
  return 0;
}
```

Vektorer

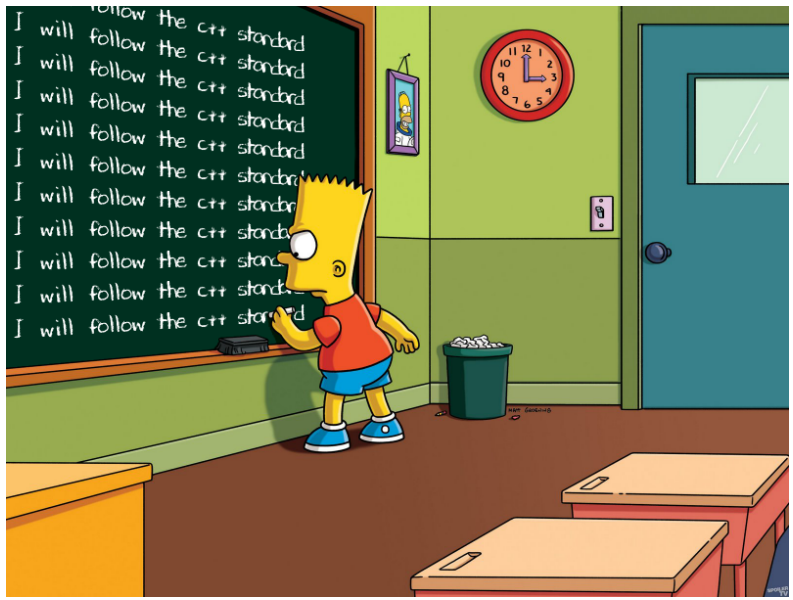
- Vi kan lägga till värden sist i en vektor
- Vad mer kan vi göra med en vektor? Kolla c++reference!

```
#include <vector>

int main()
{
  std::vector<int> v {};
  v.push_back(1);
  v.push_back(5);
  v.push_back(2);

  std::cout << v.at(0) << v.at(1) << v.at(2) << std::endl;
  return 0;
}
```

Upprepning

- På vilka olika sätt kan vi repetera?
  - Ett specifikt antal gånger
  - Tills något händer

- Tre olika språkkonstruktioner
  - `while`
  - `do-while`
  - `for`



Upprepning - while

Upprepa instruktionerna i blocket så länge villkoret är sant.

```
while ( villkor )
{
  // Block med instruktioner
}
```

- Använd när vi inte vet hur länge vi ska iterera
- Kör noll eller flera gånger

Upprepning - while

```
std::string name {};
std::cout << "Vad heter du?" << std::endl;
std::cin >> name;

while (name != "Klas")
{
  std::out << "Bättre namn kan du! Vad heter du?" << std::endl;
  std::cin >> name;
}
std::cout << "Nämen! Det var ett fint namn." << std::endl;
```

Upprepning - do while

Upprepa instruktionerna i blocket så länge villkoret är sant.

```
do
{
  // Block med instruktioner
}
while ( villkor );
```

- Använd när vi inte vet hur länge vi ska iterera
- Kör en eller flera gånger

Upprepning - do while

```
std::string name {};

do
{
  std::out << "Vad heter du?" << std::endl;
  std::cin >> name;
}
while (name != "Klas");

std::cout << "Nämen! Det var ett fint namn." << std::endl;
```

Upprepning - for

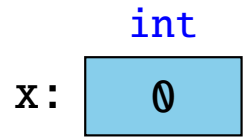
- Upprepar instruktionerna i blocket så länge villkoret är sant.
- I slutet av varje varv kör vi intruktionen i ’end’
- Innan första varvet, kör satsen i init

```
for (init; villkor; end)
{
  // Block med instruktioner
}
```

- Använd när vi kan räkna ut exakt hur länge vi ska iterera.
- Skiljer sig markant från python!

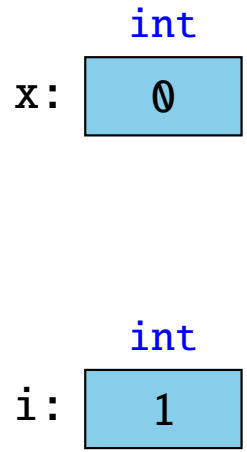
Upprepning - for

```
#include <iostream>

int main()
{
  int x {};

  for (int i {1}; i < 4; ++i)
  {
    x += i;
  }
  std::cout << x << std::end;
}
```



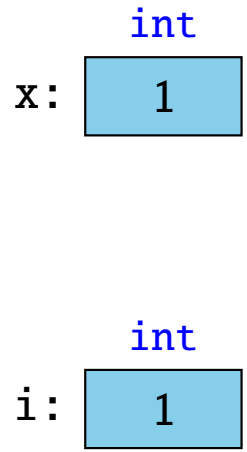
Upprepning - for

```
#include <iostream>

int main()
{
  int x {};

  for (int i {1}; i < 4; ++i)
  {
    x += i;
  }
  std::cout << x << std::end;
}
```



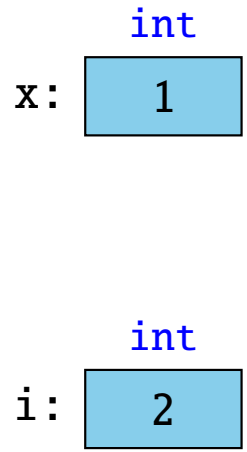
Upprepning - for

```
#include <iostream>

int main()
{
  int x {};

  for (int i {1}; i < 4; ++i)
  {
    x += i;
  }
  std::cout << x << std::end;
}
```



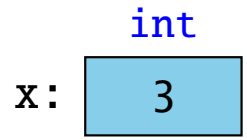
Upprepning - for

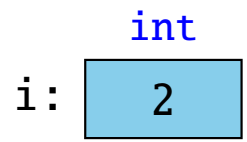
```
#include <iostream>

int main()
{
  int x {};

  for (int i {1}; i < 4; ++i)
  {
    x += i;
  }
  std::cout << x << std::end;
}
```



Upprepning - for

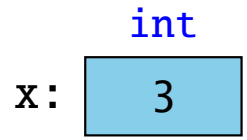
```
#include <iostream>

int main()
{
  int x {};

  for (int i {1}; i < 4; ++i)
  {
    x += i;
  }
  std::cout << x << std::end;
}
```

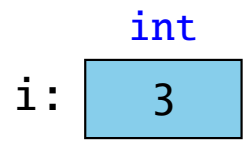




Upprepning - for

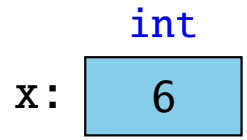
```
#include <iostream>

int main()
{
  int x {};

  for (int i {1}; i < 4; ++i)
  {
    x += i;
  }
  std::cout << x << std::end;
}
```

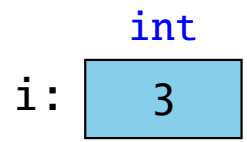




Upprepning - for

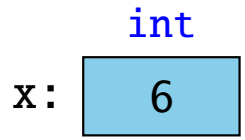
```
#include <iostream>

int main()
{
  int x {};

  for (int i {1}; i < 4; ++i)
  {
    x += i;
  }
  std::cout << x << std::end;
}
```

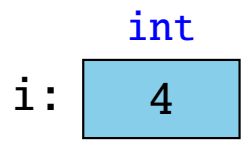




Upprepning - for

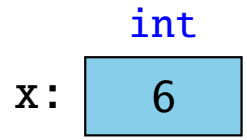
```
#include <iostream>

int main()
{
  int x {};

  for (int i {1}; i < 4; ++i)
  {
    x += i;
  }
  std::cout << x << std::end;
}
```





Upprepning - for

```
#include <iostream>

int main()
{
  int x {};

  for (int i {1}; i < 4; ++i)
  {
    x += i;
  }
  std::cout << x << std::end;
}
```



Upprepning

Skriver vi snarlik kod i följd som följer ett mönster så är det en signal att upprepning kan vara bra att använda.

```
x += 2;
x += 3;
x += 4;
x += 5;
x += 6;
x += 7;
```

```
for (int i {}; i < 6; ++i)
{
  x += 2 + i;
}
```

Upprepning

Ibland måste vi förstärka mönster för att kunna använda upprepning.

```
cout << "1: Måndag"  << endl;
cout << "2: Tisdag"  << endl;
cout << "3: Onsdag"  << endl;
cout << "4: Torsdag" << endl;
cout << "5: Fredag"  << endl;
```

```
vector<string> days {
  "Mån", "Tis", "Ons", "Tors", "Fre" };

for (int i {}; i < days.size(); ++i)
{
  cout << i << ": "
       << days.at(i) << "dag" << endl;
}
```

Aggregat

- Vi kan samla data som tillsammans skapar en större betydelse
- Skapas som en ny datatyp med ett namn och de data som ingår

```
struct Person
{
  string name;
  int shoe_size;
  double length;
};
```

- En struct beskriver den nya datatypen. Ingen variabel skapas!

Aggregat

Nu kan vi skapa variabler av den nya datatypen i våra program.

```
void print(Person const& p)
{
  std::cout << "Namn:        " << p.name      << std::endl;
  std::cout << "skostorlek: " << p.shoe_size << std::endl;
  std::cout << "Längd:       " << p.length    << std::endl;
}

int main()
{
  Person examinator { "Klas", 42, 1.89 };
  print(examinator);
  return 0;
}
```

Inkludering

- Varför inkluderar vi?
  - Vi vill inte uppfinna hjulet igen eller kopiera kod varje gång vi vill använda något.
- Vad händer när vi inkluderar?
  - Koden i den andra filen koperias automatiskt in.
- Varför inkluderar vi inte bara allt?
  - Det tar längre och längre tid att kompilera!
  - Inkludera endast det vi använder

Slut för idag

Registrera er i kursen och anmäl er i WebReg!

www.liu.se

LINKÖPING UNIVERSITY