TDDC76 – Programmering och datastrukturer

Fortsättning föreläsning 4 och Git

Klas Arvidsson 2020, Oskar Holmström 2019
Institutionen för datavetenskap

LINKÖPINGS UNIVERSITET

Allmän info

Hjälpa varandra

- BRA att prata om hur det fungerar med hjälp av ett exempel helt orelaterad labben
- BRA att diskutera exempel och rita figurer utan någon programkod
- BRA att fråga ”Varför?”

- DÅLIGT att visa din kod för kompisar och att skriva av kod

- Tittar du på färdig kod bör det vara i jämförande syfte – hur gör koden du studerar i jämförelse med hur du redan gjort?
  Bättre? Sämre?

Allmän info

Söka hjälp

- Syfte:
  - *Lära dig kursinnehåll* genom att lösa uppgifterna
  - Dvs att lösa uppgifterna är inte det primära syftet och det är inte för att ”lösa uppgifterna” som du är här

- Fråga dig:
  - Vad nytt lär jag mig av källan, arbetsättet, etc?
  - Är källan tillförlitlig? Leder arbetssättet till nya insikter?

Allmän info

Ch(e)atGPT

- ChatGPT briljerar när det gäller att hitta och sammanfatta känd information.
- ChatGPT lämnar mycket att önska för andra uppgifter. Exempel:
  - Vi provar multiplicera tal.
  - Vi provar lösa problem.
- ChatGPT fabricerar ibland mycket trovärdig information.

Agenda

1. Fortsättning från föreläsning 4 (se slides därifrån)
   - 1.1 Exempel speciella medlemsfunktioner
   - 1.2 Rekursion
2. Git
3. Make
4. Gdb
5. Valgrind

LINKÖPINGS UNIVERSITET

Utan rekursion är vi begränsade

- Utan rekursivt tankesätt:
  1. Jag behöver beräkna n-fakultet
  2. Jag implementerar en funktion som beräknar n-fakultet
  3. Jag anropar funktionen för att beräkna n-fakultet

- Varför begränsar jag mig att inte anropa funktionen före steg 3 trots att jag vet att den kommer finnas?

Med rekursivt tankesätt

1. Jag behöver beräkna n-fakultet och ***förutsätter*** jag har en funktion för det
2. Jag ***anropar*** min funktion för att beräkna en mindre n-fakultet i min implementation (dela upp i ”n” och ”n-1”)
3. Jag identifierar den minsta och trivialt beräknade n-fakulteten för att avbryta anropskedjan

```
int fakultet(int n)
{
  return n*fakultet(n-1);
}

int fakultet(int n)
{
  if ( n == 0 )
    return 1;
  else
    return n*fakultet(n-1);
}
```

Fibonacci

1. Jag behöver beräkna n:te Fibonacci-talet och ***förutsätter*** jag har en funktion
2. Jag ***anropar*** min funktion för att beräkna de två föregående fibonaccitalen (dela upp i ”n”te, ”n-1”te och ”n-2”te)
3. Jag identifierar de två minsta och trivialt beräknade Fibonaccitalet för att avbryta anropskedjan
4. OBS! Tids-komplexitet?

```
unsigned fib(unsigned n)
{
  return fib(n-1)+fib(n-2);
}
```

```
unsigned fib(unsigned n)
{
  if ( n < 2 )
    return n;
  else
    return fib(n-1)+fib(n-2);
}
```

Kopiering med rekursivt tankesätt

1. Jag behöver kopiera en lista och ***förutsätter*** jag har en funktion för det
2. Jag ***anropar*** min funktion för att kopiera en kortare lista (dela upp i ”första elementet” och ”resten av listan”)
3. Jag identifierar att kopiering av tom lista är trivialt och avbryter då anropskedjan
4. OBS! Minnes-komplexitet? Anropsdjup?

Kopiering med rekursivt tankesätt

1. Jag behöver kopiera ett binärt träd och ***förutsätter*** jag har en funktion för det
2. Jag ***anropar*** min funktion för att kopiera två delträd (dela upp i ”den här noden” och ”nodens två delträd”)
3. Jag identifierar att kopiering av tomt träd (ingen nod) är trivialt och avbryter då anropskedjan
4. OBS! Minnes-komplexitet? Anropsdjup?

Destruering med rekursivt tankesätt

1. Jag behöver destruera en lista och ***förutsätter*** jag har en funktion för det
2. Jag ***anropar*** min funktion för att destruera resten av listan (dela upp i ”det här elementet” och ”resten av listan”)
3. Jag identifierar att kopiering av tom lista är trivialt och avbryter då anropskedjan

Rekursiv destruktor

En destruktor fungerar rekursivt!

- Exempel på destruktorer
- Rekursivt (bakifrån)

```
List::Node::~Node()
{
  delete next;
}

List::~List()
{
  delete head;
}
```

Iterativt (framifrån)

```
List::~List()
{
  while ( head != nullptr )
  {
    Node* victim{head};
    head = head->next;
    victim->next = nullptr;
    delete victim;
  }
}
```

Agenda

1. Fortsättning från föreläsning 4 (se slides därifrån)
   - 1.1 Exempel speciella medlemsfunktioner
   - 1.2 Rekursion
2. Git
3. Make
4. Gdb
5. Valgrind

LINKÖPINGS UNIVERSITET

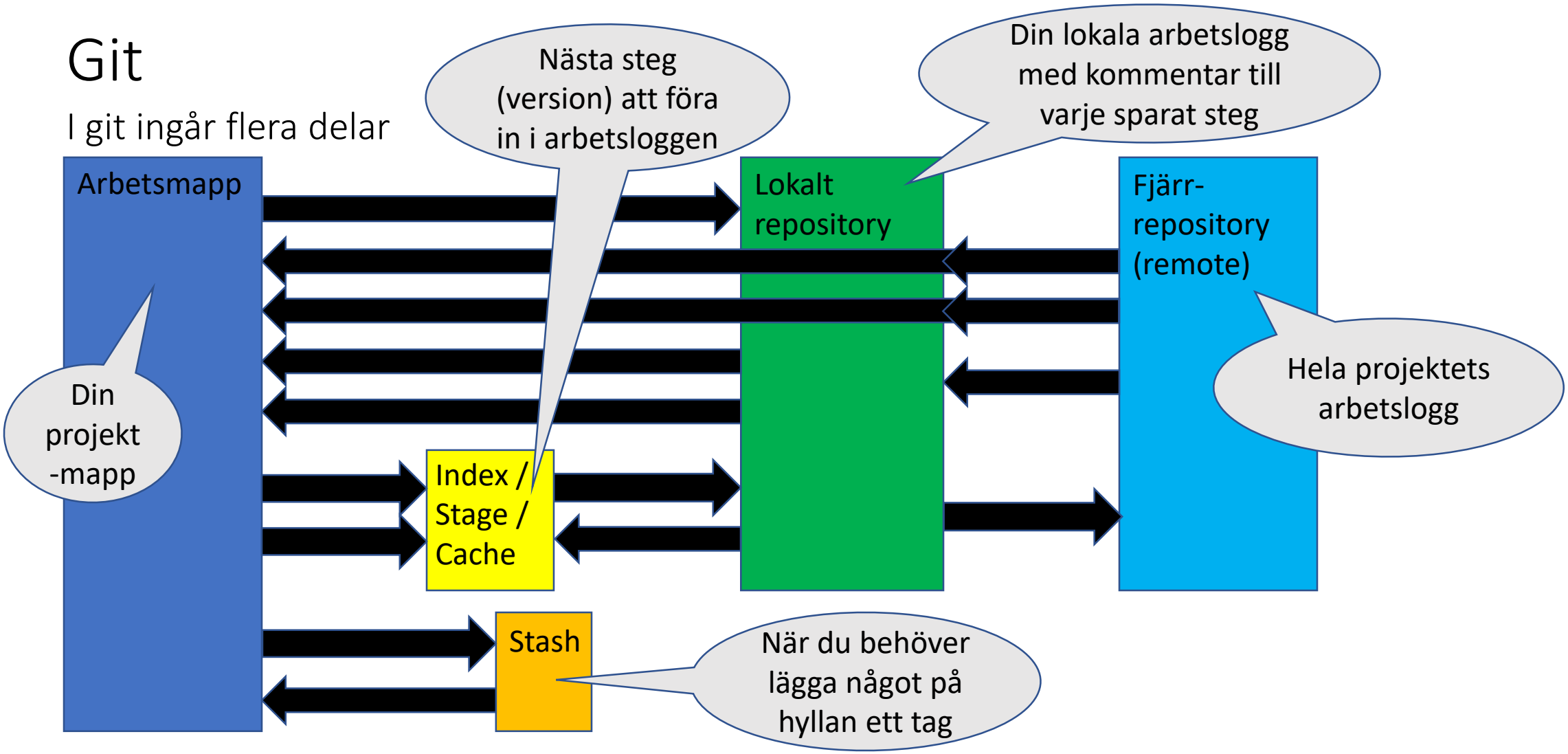
Git

Vad är Git och varför använder vi det?

- Versionshantering, arbetslogg
  - Sparar vår kod (och allt annat vi vill spara) i ögonblicksbilder
  - Vi kan spåra ändringar
  - Vi kan gå tillbaka i tiden
- Samarbete - enkel delning av filer för arbete i grupp
  - Fungerar väl för t.ex. källkod, dokument, presentationer, etc.
- Tillgänglighet
  - Vi kan arbeta på vilken dator som helst
  - Vi kan återskapa arbete om olyckan är framme

Git

I git ingår flera delar



Git

Kommandon du vanligen behöver

- `git status`
  - Sammanfattning hur arbetsmappen skiljer sig från repositoryt
- `git add [filename]`
  - Filen som den ser ut när du kör ”add” kommer att sparas i nästa commit
- `git commit`
  - Sparar alla filer som vi gjort ”add” på sedan föregående commit
- `git push`
  - Skickar alla lokala commits, som ej finns på remote, till remote
- `git pull`
  - Hämta hem alla commits från remote som inte finns i local
- `git clone <remote address>`
  - Skapar en mapp med projektets namn och allt innehåll (samtliga commits) som fanns på remote vid kloningen

Git

Flaggor

- Alla kommandon har flaggor för att modifiera beteendet
- git add
  - -u flaggan lägger till alla filer som ändrats sedan senast men ej nya
  - -i flaggan ger interaktivt skal för att lägga till enbart vissa ändringar
  - --patch flaggan lägger till delar(hunks) av en fil
- git commit
  - -m flaggan låter oss sätta ett bra meddelande på vår commit
  - -a flaggan tar automatiskt med alla ändringar på filer som som finns (som om du skulle gjort git add -u)
  - Kan kombineras: -am
- git reset
  - nollställer staging area så du får börja om med ”git add”

Git

Branch kommandon

- `git branch -a`
  - Listar alla *branches* som finns, både hos *local* och *remote*
- `git checkout –b [branch name]`
  - Skapa en *branch* med namnet [branch name] och flytta till *branch*

Git

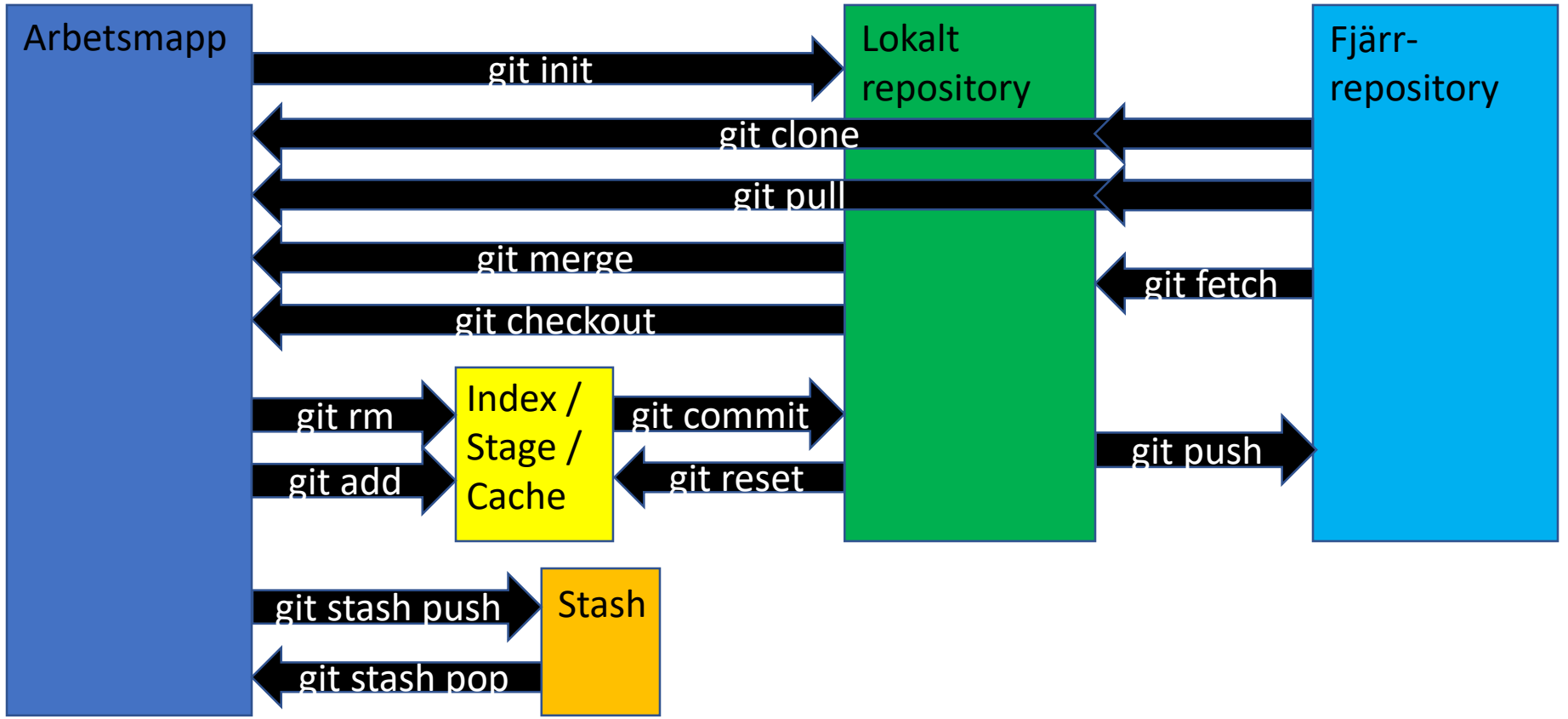
Ytterligare kommandon

- `git diff`
  - Visar skillnaderna mellan filernas nuvarande tillstånd och tillståndet i senaste commit (du kan även jämföra två specifika commits)
- `git log`
  - Visar hela commit-historiken
  - Bra flaggor: `git log --all --graph --oneline --decorate`
- `git checkout`
  - Låter oss hämta ut en specifik commit eller branch
  - Användbart när vi vill gå tillbaka till en gammal commit:

```
git checkout cef0a8e20c19f386655bea3a77e0978f674101f5
```

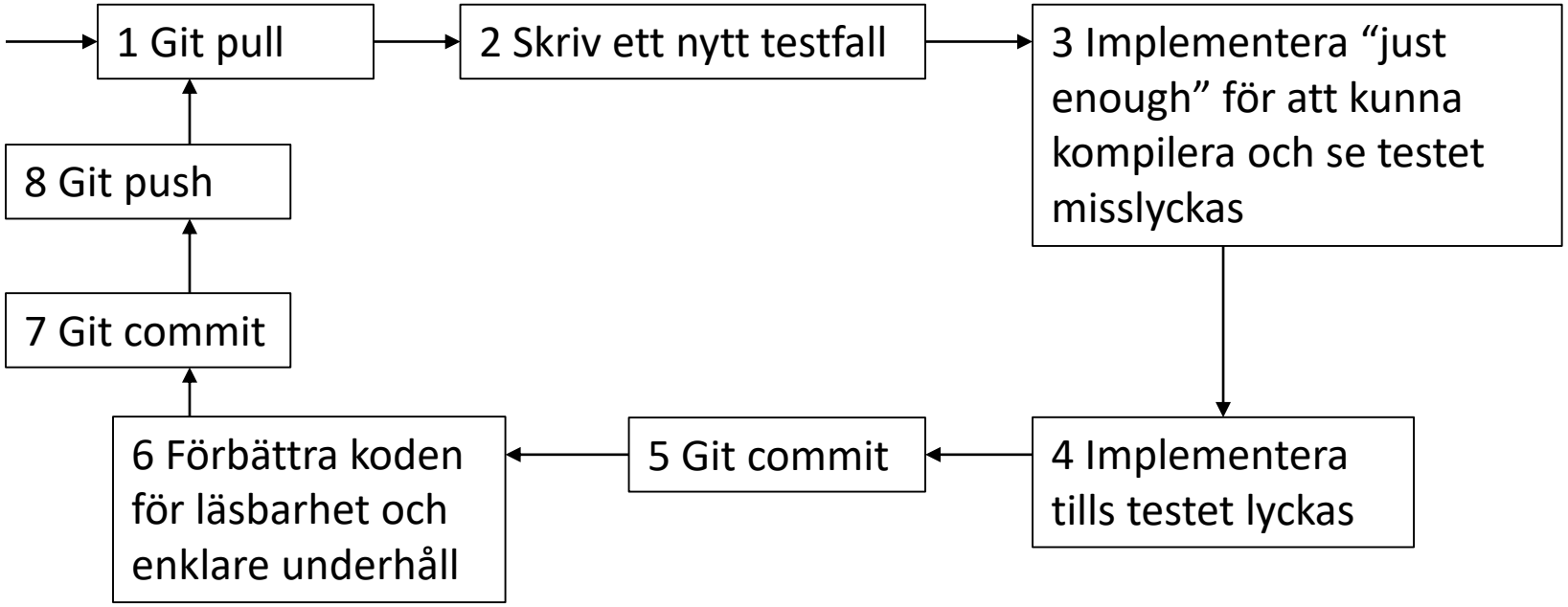
Git

Översikt kommandon



TDD och Git

Exempel hur arbetsflödet kan se ut



Git

Hur fungerar det?

- Du måste stå i en mapp som innehåller ett repository.
- Vi arbetar med Git genom att skriva kommandon i terminalen (Det du ser på `gitlab.liu.se` är bara till för administration.)

Git

Exempel

- *Local* som ligger på din hårddisk
- *Remote* som ligger på internet (t.ex. på Gitlab)
- Flera användare kan ha egna lokala *repositories* (mapp med projektet) kopplade till samma remote

Git

Vad händer när vi gör en commit?

- Vi skapar en ögonblicksbild lokalt av de ändringar som har gjorts

Git

Användning av Git push

- **User 1** har skapat ett program som skriver ut ”Hello World”
- Har utfört git add, git commit och nu git push

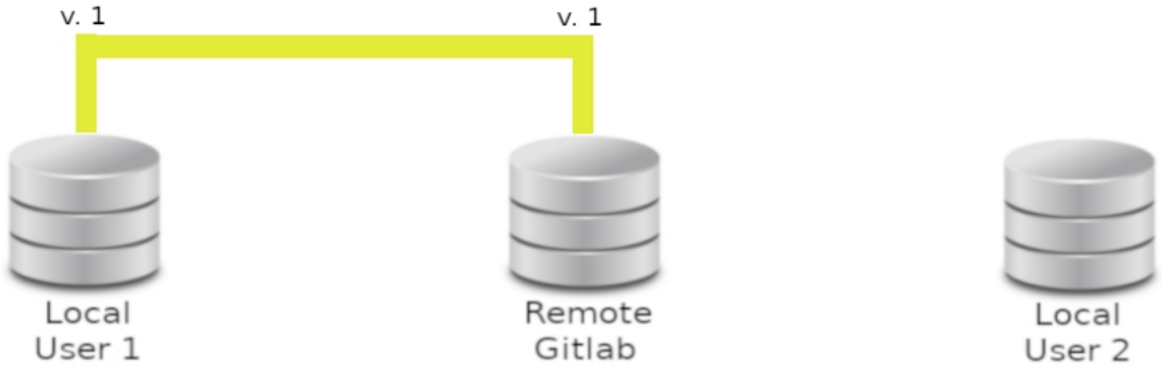
helloworld.cc

```
#include <iostream>
using namespace std;

int main()
{
  cout << ”Hello, World”
              << endl;
}
```

Git

Användning av Git push

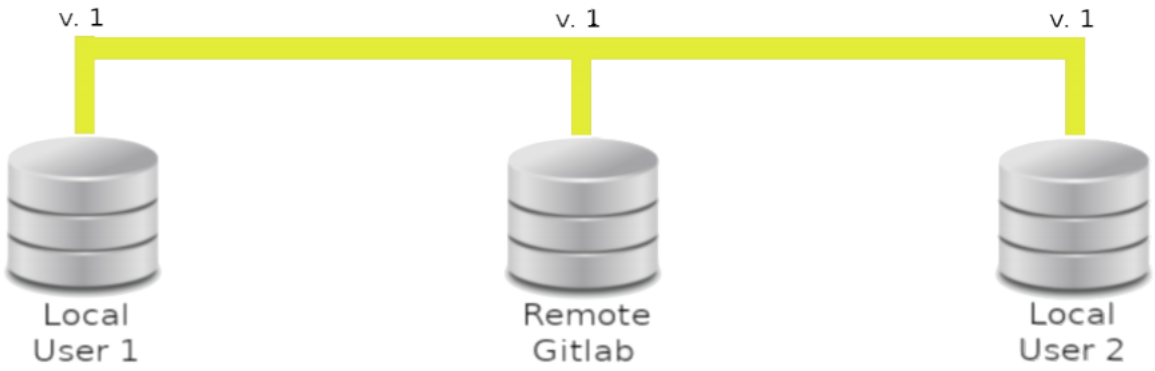
- Dvs. vi har en lokal commit som inte finns på remote och vi utför git push



Git

Användning av Git clone

- En annan användare vill arbeta mot samma remote och utför git clone



Git

Potentiella problem (Merge conflicts)

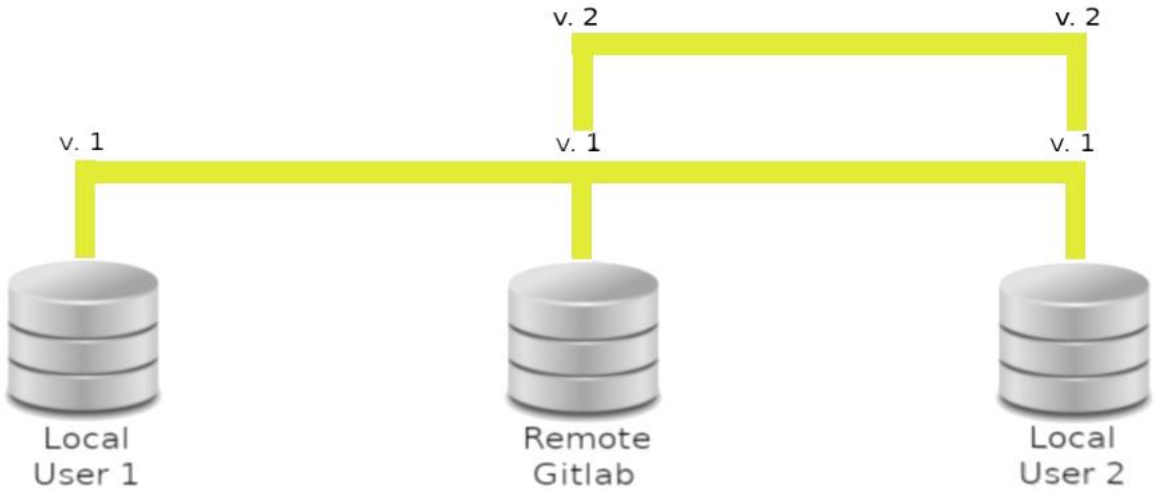
- **User 2** lägger till en utskrift: ”User 2 är bäst!”
- Utför:
  - git add helloworld.cc
  - git commit –m ”Added awesome print”
  - git push

helloworld.cc

```
#include <iostream>
using namespace std;

int main()
{
  cout << ”Hello, World”
       << endl;
  cout << ”User 2 är bäst!”
       << endl;

}
```

Git

Potentiella problem (Merge conflicts)



Git

Potentiella problem (Merge conflicts)

- **User 1** lägger under tiden till en variabel och inläsning och utför:
  - git add helloworld.cc
  - git commit –m ”Input to variable”
  - git push

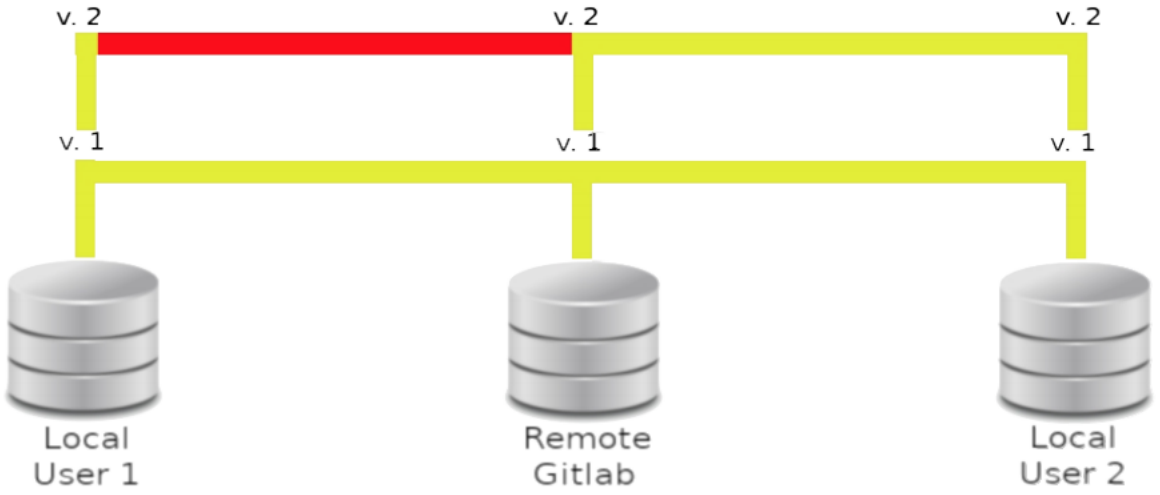
helloworld.cc

```
#include <iostream>
using namespace std;

int main()
{
  cout << ”Hello, World”
       << endl;
  int x{};
  cin >> x;

}
```

Git

Potentiella problem (Merge conflicts)

- Vi har fått en *merge conflict*
  - Samma fil har två olika innehåll på samma version



Git

Hur löser vi merge conflicts?

- Svårare konflikter, som ej kan lösas automatiskt, markeras ut i koden och vi behöver hantera dem själva
  - Detta sker t.ex. när vi gör ändringar till exakt samma rad i koden

- Konflikten markeras i koden på följande vis:

```
<<<<<<< HEAD
//Skillnader tillhörande remote
======= //Skiljerad
//Dina lokala skillnader
>>>>>>>
```

Git

Hur löser vi merge conflicts?

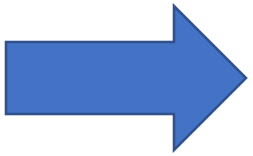
- Vi löser vår merge conflict genom att
  - Ta bort det vi inte vill ha kvar
  - Ta konfliktmarkörerna
  - (Upprepa på alla ställen där det finns merge conflicts)
  - Utför add / commit / push

Git

Hur löser vi merge conflicts?

helloworld.cc

```
#include <iostream>
using namespace std;

int main()
{
  cout << "Hello, World"
       << endl;
<<<<<<< HEAD
  cout << "User 2 är bäst!"
       << endl;
=======
  int x{};
  cin >> x;
>>>>>>>
}
```



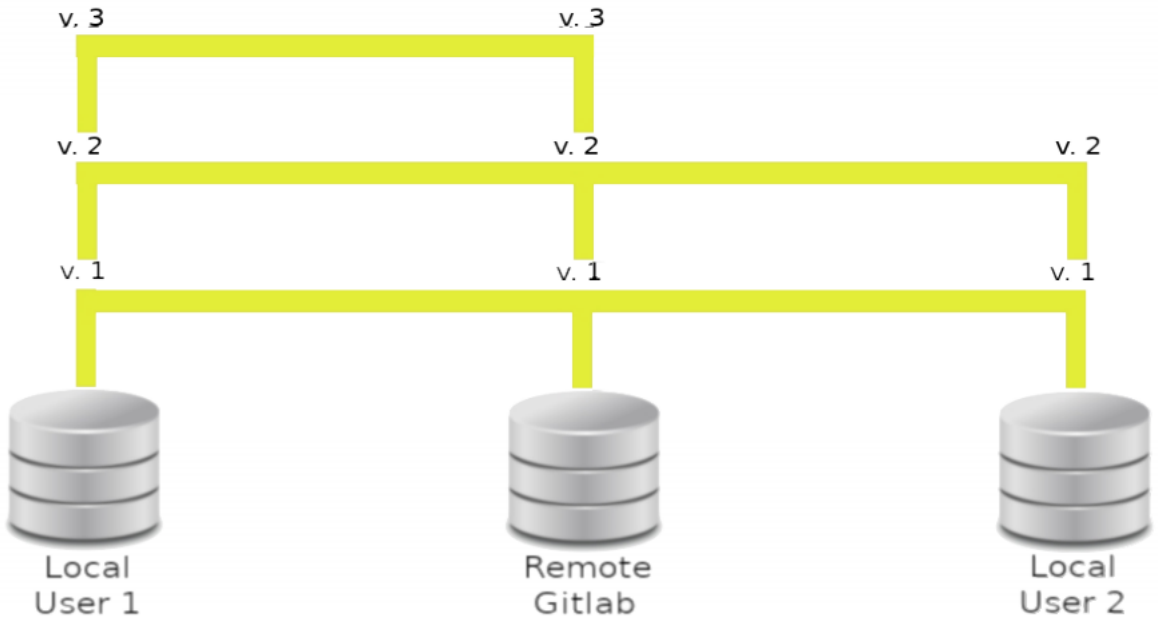
helloworld.cc

```
#include <iostream>
using namespace std;

int main()
{
  cout << "Hello, World"
       << endl;
  cout << "User 2 är !bäst"
       << endl;
  int x{};
  cin >> x;
}
```

Utför add / commit / push
(och säg åt User 2 göra pull)

Git

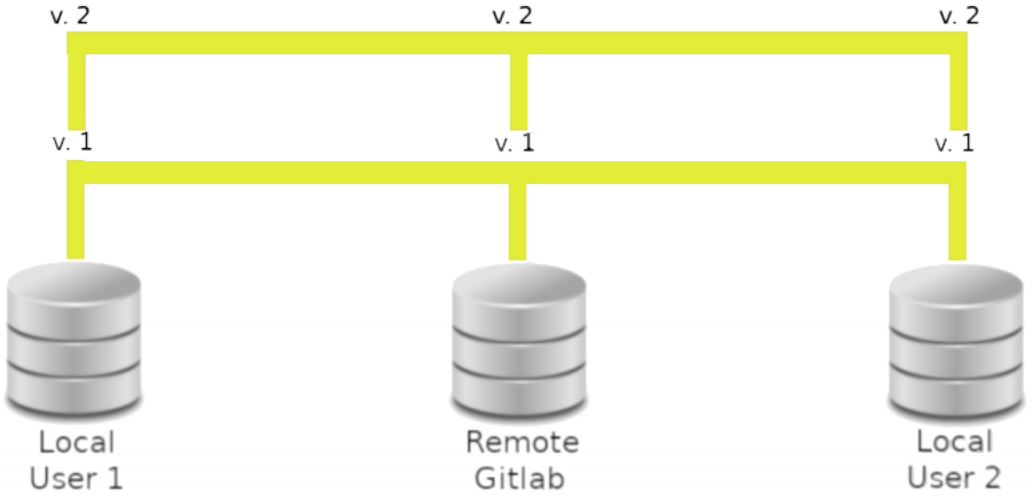
Merge conflict löst!



Git

Hur undviker vi merge conflicts?

- Vi delar upp arbetet
  - User 1 får arbeta i vissa filer och User 2 i andra
  - Vi kan även arbeta i olika branch
- Konflikter kan vi inte alltid undvika! Hur löser vi dem?
  - En lättlöst konflikt löser Git's merge åt oss automatiskt



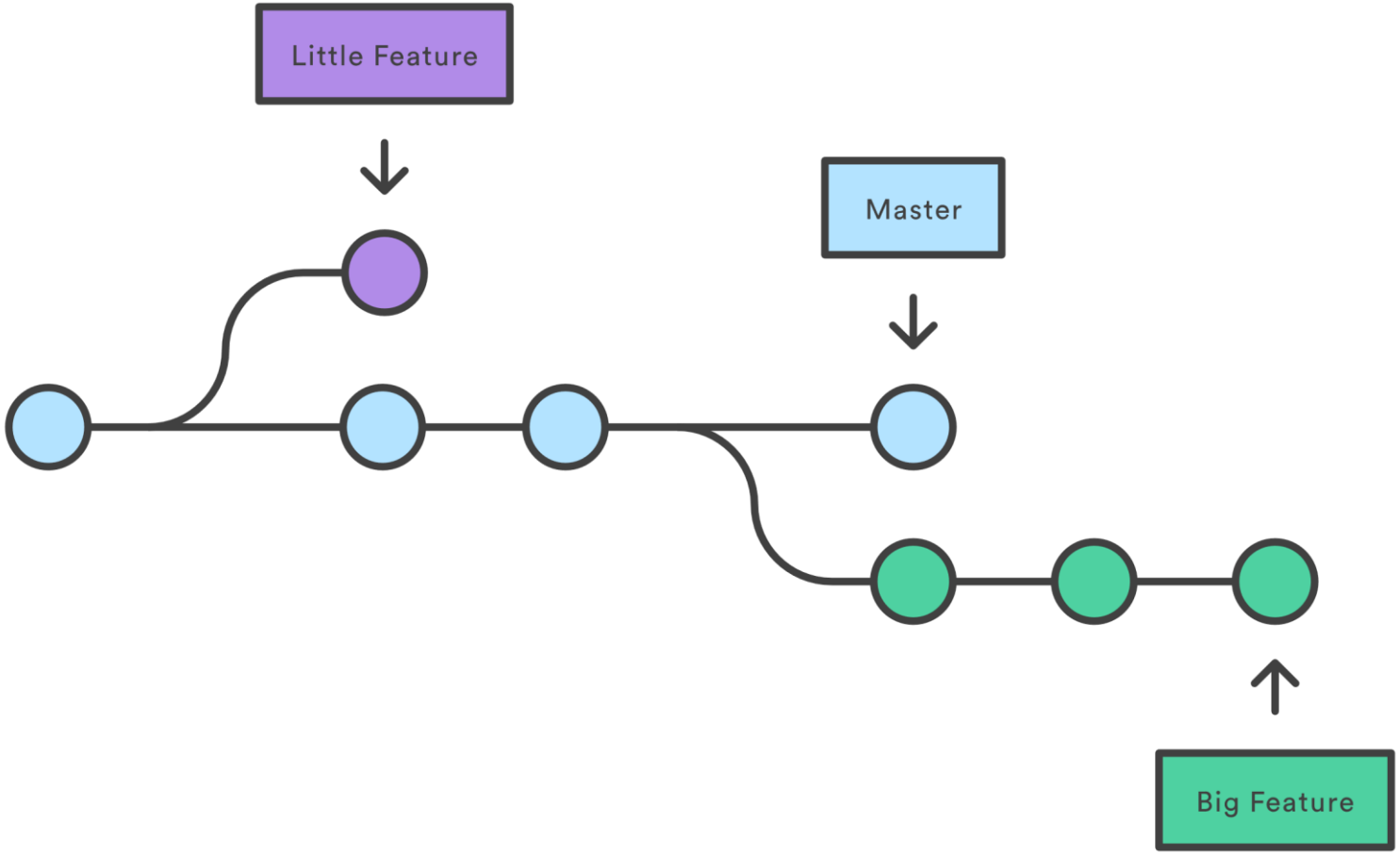
Git

Branches

- Finns en standardbranch, `main`, som alla kan arbeta emot (tidigare kallades `main` för `master`)
- Nackdelar att arbeta mot samma branch:
  - Behöva hantera merge conflicts hela tiden
  - Vill du testa något kommer alla ta del av det
- Istället kan vi göra en avgrening från main där vi kan arbeta fritt i lugn och ro
  - Görs ofta en branch för varje feature man arbetar med
  - När feature fungera mergas allt in i `main`

Git

Branches



Git

Merge branches

- Vi kan föra samman en branch med main genom att utföra:
  - `git checkout main`
  - `git merge [branch name]`

Agenda

1. Fortsättning från föreläsning 4 (se slides därifrån)
   - 1.1 Exempel speciella medlemsfunktioner
   - 1.2 Rekursion
2. Git
3. Make
4. Gdb
5. Valgrind

LINKÖPINGS UNIVERSITET

Space invaders: Många programfiler

Vad är problemet?

```
enemy.cc, enemy.h
asteriod.cc, asteroid.h
ship.cc, ship.h
game_main.cc
test_enemy.cc
test_asteroid.cc
test_ship.cc
test_main.cc
```

Space invaders: Många programfiler

Vad är problemet?

```
enemy.cc, enemy.h
asteriod.cc, asteroid.h
ship.cc, ship.h
game_main.cc
test_enemy.cc
test_asteroid.cc
test_ship.cc
test_main.cc
```

- Vilka kompileringsflaggor ska användas?
- Hur ska varje fil kompileras?
- Hur ska testerna kompileras?
- Hur ska spelet kompileras?
- Måste allt kompileras om när något ändras?

Makefile

Lösningen när det är jobbigt att kompilera!

- `make` är ett verktyg för att hantera kompilering av många filer
- Vi skapar en fil: `Makefile`
  - Innehåller regler för hur filer beror av varandra
  - Innehåller regler för hur allt ska kompileras
  - Kan användas för att automatisera mer än kompilering
- Kommandot `make` kör vår Makefile och de kommandon vi skapat i den för att kompilera filer
- Kommandot `make` genererar automatiskt om varje fil som är äldre än någon av filerna den beror på

Makefile

Grundstruktur för en GNU Makefile

```
VARIABEL=värde
VAR=$(VARIABEL)


fil_att_skapa: filer som behövs
tab--->| kommando 1 hur filen ska skapas
tab--->| kommando 2 hur filen ska skapas


# Kommentar: filen döps “Makefile”
```

Makefile

Exempel

```
enemy.cc, enemy.h, asteriod.cc, asteroid.h
ship.cc, ship.h, game_main.cc
```

Makefile

```
ship.o: ship.cc ship.h
tab--->| g++ -c ship.cc


a.out: ship.o enemy.o asteroids.o game_main.o
tab--->| g++ ship.o enemy.o asteroids.o game_main.o
```

Makefile

Exempel – underhållsvänligare med variabler – här färgkodat!

```
CC=g++
CCFLAGS="-std=c++17 -Wall -Wextra"
LDFLAGS="-lsfml-window -lsfml-graphics -lsfml-system"
OBJS="ship.o enemy.o asteroids.o game_main.o"

game: $(OBJS)
tab--->| $(CC) $(LDFLAGS) $^ -o $@

%.o: %.cc %.h
tab--->| $(CC) $(CCFLAGS) -c $<
```

Makefile

Exempel – specialvariabel och dess källa i fet stil!

```
CC=g++
CCFLAGS="-std=c++17 -Wall -Wextra"
LDFLAGS="-lsfml-window -lsfml-graphics -lsfml-system"
OBJS="ship.o enemy.o asteroids.o game_main.o"

game: $(OBJS)
tab--->| $(CC) $(LDFLAGS) $^ -o $@

%.o: %.cc %.h
tab--->| $(CC) $(CCFLAGS) -c $<
```

Makefile

Exempel – specialvariabel och dess källa i fet stil!

```
CC=g++
CCFLAGS="-std=c++17 -Wall -Wextra"
LDFLAGS="-lsfml-window -lsfml-graphics -lsfml-system"
OBJS="ship.o enemy.o asteroids.o game_main.o"

game: $(OBJS)
tab--->| $(CC) $(LDFLAGS) $^ -o $@

%.o: %.cc %.h
tab--->| $(CC) $(CCFLAGS) -c $<
```

Makefile

Exempel – specialvariabel och dess källa i fet stil!

```
CC=g++
CCFLAGS="-std=c++17 -Wall -Wextra"
LDFLAGS="-lsfml-window -lsfml-graphics -lsfml-system"
OBJS="ship.o enemy.o asteroids.o game_main.o"

game: $(OBJS)
tab--->| $(CC) $(LDFLAGS) $^ -o $@

%.o: %.cc %.h
tab--->| $(CC) $(CCFLAGS) -c $<
```

Makefile

Exempel – specialvariabel och dess källa i fet stil!

```
CC=g++
CCFLAGS="-std=c++17 -Wall -Wextra"
LDFLAGS="-lsfml-window -lsfml-graphics -lsfml-system"
OBJS="ship.o enemy.o asteroids.o game_main.o"

game: $(OBJS)
tab--->| $(CC) $(LDFLAGS) $^ -o $@

%.o: %.cc %.h
tab--->| $(CC) $(CCFLAGS) -c $<
```

Kompilering

Flaggor bra att känna till

- Varna vid alla former av fel

```
g++ -std=c++17 -Wall –Wextra –Weffc++ –Woverloaded-virtual –Wold-style-cast
```

  - studenter missar regelbundet tentor och får kompletteringar på sina laborationer för fel de enkelt kunnat upptäcka genom att använda kompileringsvarningar

- Säg som det är lite i taget (varningar blir fel, men bara tre i taget)

```
–Werror –fmax-errors=3
```

- Kompilera bara filen, länka inte ihop programmet (filen main.o genereras)

```
g++ -c main.cc
```

- Döp om den fil som genereras (programmet heter nu main i stället för a.out)

```
g++ main.cc –o main
```

- Generera extra information för debugger (och andra verktyg som valgrind)

```
g++ -g main.cc
```

Agenda

1. Fortsättning från föreläsning 4 (se slides därifrån)
   - 1.1 Exempel speciella medlemsfunktioner
   - 1.2 Rekursion
2. Git
3. Make
4. Gdb
5. Valgrind

LINKÖPINGS UNIVERSITET

Buggig kod för gdb och valgrind-exempel

Utöver buggar även problem med både inkapsling och abstraktion!

```
Se “list.cc” publicerad bredvid
föreläsningsbilderna.
```

GDB – Gnu DeBugger

Hjälp vid “segmentation fault”

```
$ g++ list.cc
$ gdb ./a.out
Reading symbols from ./a.out...
(gdb)
```

GDB – Gnu DeBugger

Hjälp vid “segmentation fault”

```
(gdb) run
Starting program: /home/klaar36/tddc76-
internal/Slides/a.out


Program received signal SIGSEGV, Segmentation fault.
0x0005555555555251 in print (n=0x0) at list.cc:16
16          for ( ; n->next != nullptr; n = n->next )
(gdb)
```

GDB – Gnu DeBugger

Hjälp vid “segmentation fault”

```
(gdb) backtrace
#0  0x0000555555555251 in print (n=0x0) at list.cc:16
#1  0x0000555555555303 in main () at list.cc:33
(gdb)
```

Agenda

1. Fortsättning från föreläsning 4 (se slides därifrån)
   - 1.1 Exempel speciella medlemsfunktioner
   - 1.2 Rekursion
2. Git
3. Make
4. Gdb
5. Valgrind

LINKÖPINGS UNIVERSITET

Valgrind

Kontrollerar om kod aktiverad av programkörningen har några minnesläckor

```
$ valgrind ./a.out
==697== Memcheck, a memory error detector
==697== Copyright (C) 2002-2017, and GNU GPL'd, by
Julian Seward et al.
==697== Using Valgrind-3.15.0 and LibVEX; rerun with -h
for copyright info
==697== Command: ./a.out
```

Valgrind

Kontrollerar om kod aktiverad av programkörningen har några minnesläckor

```
==697==
==697== HEAP SUMMARY:
==697==     in use at exit: 48 bytes in 3 blocks
==697==   total heap usage: 6 allocs, 3 frees, 73,792 bytes allocated
==697==
```

Valgrind

Kontrollerar om kod aktiverad av programkörningen har några minnesläckor

```
==697== LEAK SUMMARY:
==697==    definitely lost: 48 bytes in 3 blocks
==697==    indirectly lost: 0 bytes in 0 blocks
==697==      possibly lost: 0 bytes in 0 blocks
==697==    still reachable: 0 bytes in 0 blocks
==697==         suppressed: 0 bytes in 0 blocks
==697== Rerun with --leak-check=full to see details of
leaked memory
```

Valgrind

Kontrollerar om kod aktiverad av programkörningen har några minnesläckor

```
==697== For lists of detected and suppressed errors,
rerun with: -s
==697== ERROR SUMMARY: 0 errors from 0 contexts
(suppressed: 0 from 0)
```

Valgrind

Kontrollerar även om kod aktiverad av programkörningen har minnesfel

```
==711== Invalid read of size 8
==711==    at 0x109251: print(Node const*) (list.cc:16)
==711==    by 0x109302: main (list.cc:33)
==711==  Address 0x8 is not stack'd, malloc'd or (recently) free'd
==711==
==711==
==711== Process terminating with default action of signal 11
(SIGSEGV)
==711==  Access not within mapped region at address 0x8
==711==    at 0x109251: print(Node const*) (list.cc:16)
==711==    by 0x109302: main (list.cc:33)
```

Valgrind

Kontrollerar även om kod aktiverad av programkörningen har minnesfel

```
==711== For lists of detected and suppressed errors,
rerun with: -s
==711== ERROR SUMMARY: 1 errors from 1 contexts
(suppressed: 0 from 0)
Segmentation fault
```

www.liu.se

LINKÖPINGS UNIVERSITET