Automatisering och Makefiler

F3

Automatisering

Undvik manuella rutiner så långt det är möjligt!

De kommer att behöva utföras igen!

1. Ett *manuellt* test är inte ett test
2. Du kommer att behöva generera testdata
3. Du kommer att förändra poststrukturen, och generera om ”datat”, etc.
4. ...

Den extra tid det tar att skriva ett skript för att utföra en uppgift tjänar man ofta igen ganska snabbt!

Automatiseringsverktyg

1. Skriptspråk för att t.ex.
   generera testdata,
   göra komplexare interaktioner med ett program vid testning,
   konvertera data mellan format,
   generera kod,
   etc.
2. Make eller liknande verktyg för ”build management”

Make

1. Ett program för att hantera build-processer för program, även för att automatisera installationer, testning, med mera.
2. Make eller liknande verktyg för ”build management”

Make

En serie bygginstruktioner i filen `Makefile`.

```
hello:  hello.c
        gcc -Wall -ggdb hello.c -o hello
```

`hello` är ett mål (target), `hello.c` dess beroende (dependency), `gcc -Wall...` är kompileringsdirektivet.

```
foo$ make hello
gcc -Wall -ggdb hello.c -o hello

foo$ ls hello
Makefile hello hello.c

foo$ make hello
'hello' is up to date
```

Notera: Om ett måls beroenden är yngre än målet självt sker byggs målet på nytt. Optimerar kompileringsprocessen vid stora byggen!

Makefile:n i lab 1

```
# compiler settings
C_COMPILER    = gcc
C_OPTIONS     = -Wall

# Clean settings
GEN_EXTENSIONS = *.exe
AUX_EXTENSIONS = *.o

# Version Control settings
VC_PROGRAM = hg

hello:  hello.c
        $(C_COMPILER) $(C_OPTIONS) hello.c -o hello

lintcheck:
        lint hello.c
```

Makefiler

```
bash-3.2$ rm hello

bash-3.2$ make hello
gcc -Wall hello.c -o hello
hello.c: In function 'main':
hello.c:5: warning: control reaches end of non-void function

bash-3.2$ make hello
make: 'hello' is up to date.

bash-3.2$
```

Makefile:n i lab 1

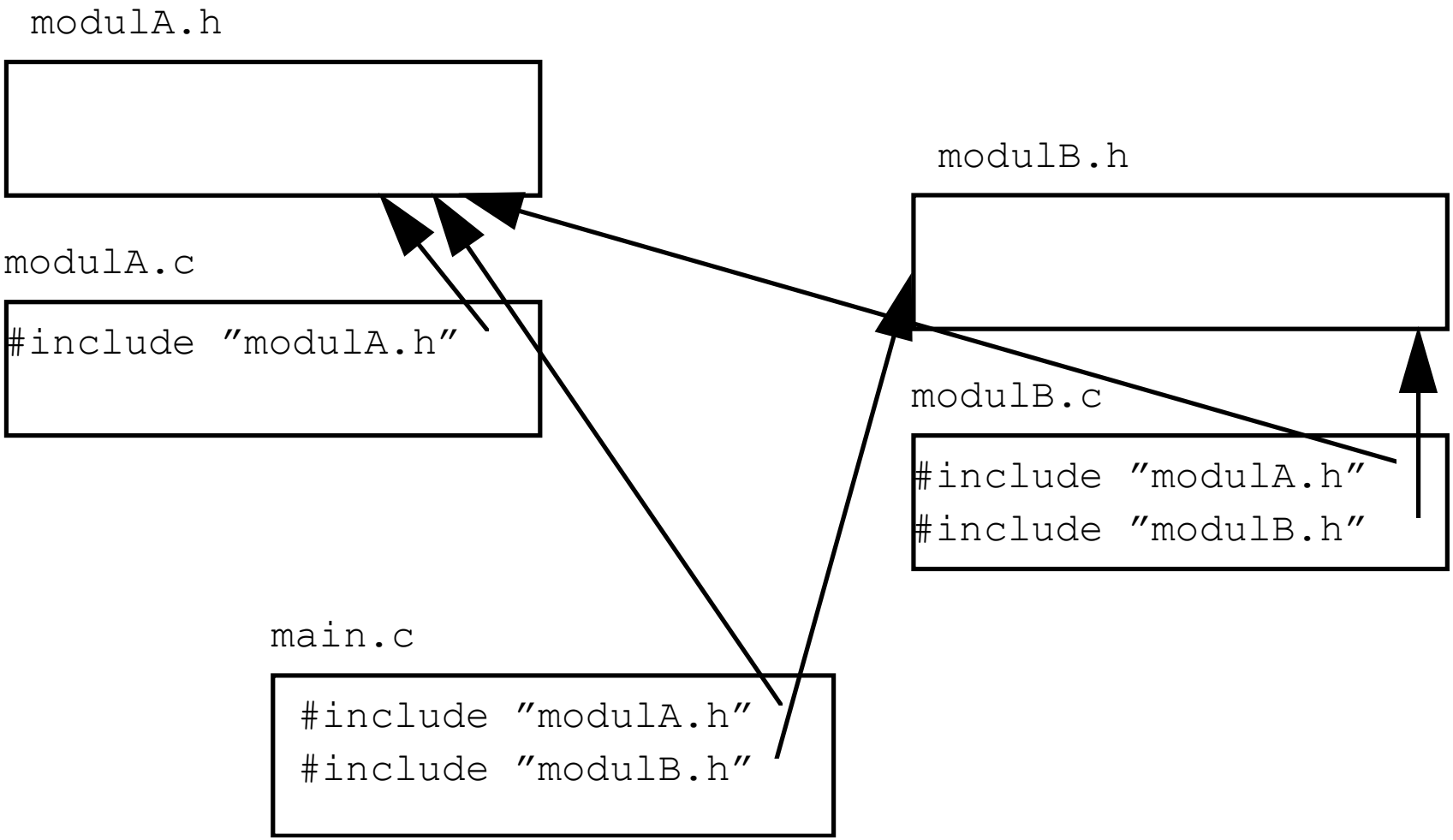
```
# compiler settings
C_COMPILER   = gcc
C_OPTIONS    = -Wall

# Clean settings
GEN_EXTENSIONS = *.exe
AUX_EXTENSIONS = *.o

# Version Control settings
VC_PROGRAM = hg

hello:  hello.c
        $(C_COMPILER) $(C_OPTIONS) hello.c -o hello

lintcheck:
        lint hello.c

test:   hello lintcheck
        ./compare hello nothing Hello
```

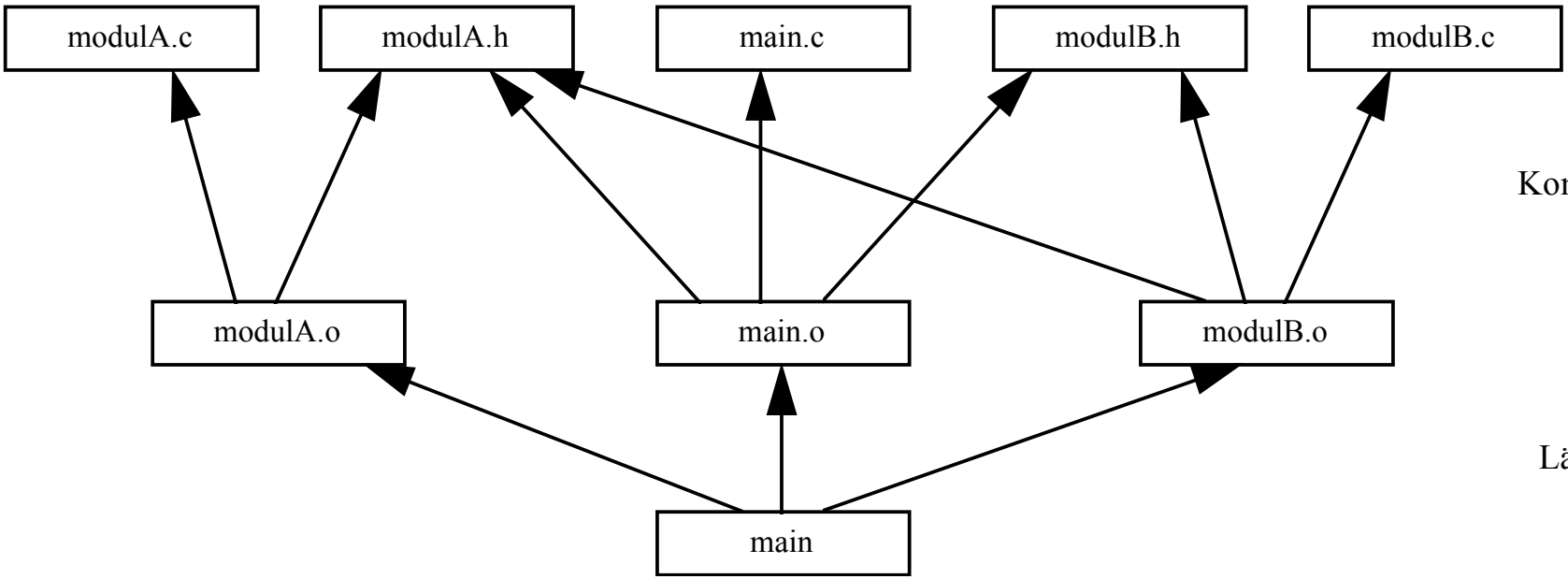
Makefiler

```
bash-3.2$ make test
make: *** No rule to make target ‘test’.  Stop.

bash-3.2$ make test
gcc -Wall hello.c -o hello
hello.c:1:21: error: missing.h: No such file or directory
hello.c: In function ’main’:
hello.c:4: warning: implicit declaration of function ’printf’
hello.c:4: warning: incompatible implicit declaration of built-in function ’printf’
hello.c:5: warning: control reaches end of non-void function
make: *** [hello] Error 1

bash-3.2$
```

Separatkompilerade moduler

1. En C-fil utan `main`-funktion
2. Kan användas för att samla funktioner och strukturer som kan återanvändas i många olika program (t.ex. en länkad lista, I/O-rutiner, etc.)
3. Kod som vill använda en separatkompilerad modul måste definiera de efterfrågade *funktionsprototyperna* (funktionshuvudena) – görs normalt med s.k. ”headerfiler” som vi skall titta mer på senare (**`#include`** `<fil.h>`)
4. Vid länkning måste den separatkompilerade modulen länkas in
5. Ett vanligt program består i regel av ett stort antal separatkompilerade moduler – att kompilera dessa korrekt är tidsödande, krångligt och felbenäget


modulA.h
modulB.h
modulA.c
#include ”modulA.h”
modulB.c
#include ”modulA.h”
#include ”modulB.h”
main.c
#include ”modulA.h”
#include ”modulB.h”


modulA.c
modulA.h
main.c
modulB.h
modulB.c
Kompileringar
modulA.o
main.o
modulB.o
Länkning
main

Make

```
modulA: modulA.h modulA.c
        gcc -Wall -ggdb -c modulA.c

modulB: modulB.h modulB.c
        gcc -Wall -ggdb -c modulB.c

prog:   prog.c
        gcc -Wall -ggdb prog.c modulA.o modulB.o -o prog
```

```
foo$ make modulA
gcc -Wall -ggdb -c modulA.c

foo$ make modulB
gcc -Wall -ggdb -c modulB.c

foo$ make prog
gcc -Wall -ggdb prog.c modulA.o modulB.o -o prog
```

Make (bättre)

```
modulA: modulA.h modulA.c
        gcc -Wall -ggdb -c modulA.c

modulB: modulB.h modulB.c modulA
        gcc -Wall -ggdb -c modulB.c

prog:   prog.c modulA modulB
        gcc -Wall -ggdb prog.c modulA.o modulB.o -o prog
```

```
foo$ make prog
gcc -Wall -ggdb -c modulA.c
gcc -Wall -ggdb -c modulB.c
gcc -Wall -ggdb prog.c modulA.o modulB.o -o prog

foo$ make prog
gcc -Wall -ggdb -c modulA.c
gcc -Wall -ggdb -c modulB.c
gcc -Wall -ggdb prog.c modulA.o modulB.o -o prog
```

Make (ännu bättre)

```
modulA.o: modulA.h modulA.c
          gcc -Wall -ggdb -c modulA.c

modulB.o: modulB.h modulB.c modulA.h modulA.c
          gcc -Wall -ggdb -c modulB.c

prog:     prog.c modulA.o modulB.o
          gcc -Wall -ggdb prog.c modulA.o modulB.o -o prog
```

```
foo$ make modulB.o
gcc -Wall -ggdb -c modulB.c

foo$ make prog
gcc -Wall -ggdb -c modulA.c
gcc -Wall -ggdb prog.c modulA.o modulB.o -o prog

foo$ make prog
'prog' is up to date
```

Make (ännu lite bättre)

```
modulA.o: modulA.h modulA.c
  gcc -Wall -ggdb -c modulA.c

modulB.o: modulB.h modulB.c modulA.o
  gcc -Wall -ggdb -c modulB.c

prog.o:   prog.c
  gcc -Wall -ggdb -c prog.c

prog:   prog.o modulA.o modulB.o
  gcc -ggdb prog.o modulA.o modulB.o -o prog
```

compare i lab 1

```
#!/usr/bin/env python
import sys
from os import popen

if len(sys.argv) < 4:
    print "Usage: compare program-name input expected-output"
else:
    program = sys.argv[1]
    input = sys.argv[2]
    expected = sys.argv[3]

    try:
        program.index("/")
    except:
        program = "./" + program

    output = popen(program + " " + input).read().strip()
    print "Testing",program,input,"against",expected,".......",

    if (output == expected):
        print "OK"
    else:
        print "Error, expected", expected, "got", output
```

Kvalitativa egenskaper hos kod

F3

Vad är bra kod?

. . . att den är funktionellt korrekt, förstås. . .

. . . men lika viktiga är de icke-funktionella aspekterna:

1. Att den är läsbar
2. Att den är enkel att testa (och att det finns test för den!)
3. Att den är enkel att underhålla och utveckla över tid
4. Få beroenden, hög sambandsgrad (low coupling/high cohesion)
5. Lämplig abstraktionsnivå
6. Inga läckande abstraktioner
7. Feltolerant
8. Effektiv
9. Återanvändningsbar
10. Portabel

Läsbar

```
XXX: PROCEDURE OPTIONS(MAIN);
     DECLARE B(1000) FIXED(7,2),
             C FIXED(11,2),
             (I, J) FIXED BINARY;
     C=0;
     DO I = 1 TO 10;
        GET LIST((B(J) DO J TO 1000));
        DO J = 1 TO 1000;
           C = C + B(J);
           END;
        END;
     PUT LIST('SUM IS ', C);
     END XXX;
```

Läsbar

```
XXX: PROCEDURE OPTIONS(MAIN);
     DECLARE A(10000) FIXED(7,2),
             C FIXED(11,2),
             J FIXED BINARY;
     C=0;
     GET LIST((A(J) DO J = 1 TO 10000));
     DO J = 1 TO 10000;
         C = C + A(J);
         END;
     PUT LIST('SUM IS ', C);
     END XXX;
```

Läsbar

```
XXX: PROCEDURE;
     DECLARE A(10000) FIXED(7,2),
             J FIXED BINARY;
     GET LIST((A(J) DO J = 1 TO 10000));
     PUT LIST('SUM IS ', SUM(A));
     END XXX;
```

Läsbar

```
XXX: PROCEDURE;
     DECLARE A(10000) FIXED(7,2),
     GET LIST(A);
     PUT LIST('SUM IS ', SUM(A));
     END XXX;
```

Några tips!

1. Funktioner skall vara korta, helst bara några få rader
2. En funktion skall idealiskt göra endast *en* sak
3. Bryt upp komplexa funktioner i flera mindre
4. Underskatta inte hur viktigt det är med bra namngivning!
5. Använd en funktion för att t.ex. plocka fram data ur en post istället för att göra det direkt – underlättar förändringar
6. Vid lämpliga tillfällen, ta ett steg tillbaka och försök hitta upprepade kodblock som man kan bryta ut i funktioner, etc.
7. Vid fel: ”crash, don’t trash!”
8. Optimera aldrig i förtid
9. `typedef int age;` är nästan alltid en bra idé!
10. YAGNI