

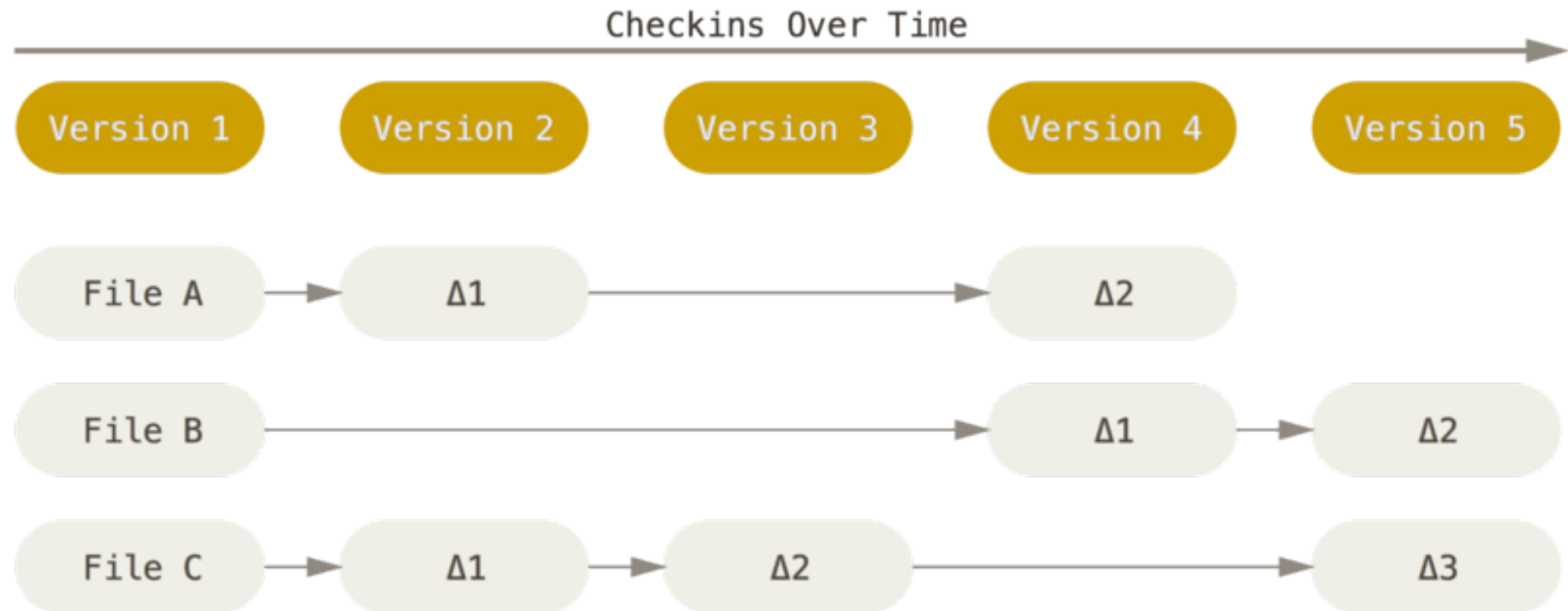
# Föreläsning 7

---

Tobias Wrigstad

*Versionshantering med git*

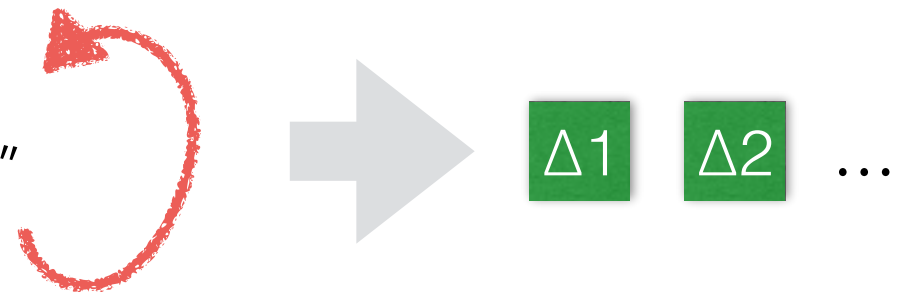




Arbetsflöde:

```
$ git add file.c file.h
```

```
$ git commit -m "Explanatory message"
```



M  $\Delta 1$   $\Delta 2$   $\Delta 3$   $\Delta 4$

You

$\Delta 2$

git add  
git commit

$\Delta 2$   
git push

Rejected  
(Out of sync)

$\Delta 3$   $\Delta 4$   
git pull

M  $\Delta 1$   $\Delta 2$   $\Delta 3$   $\Delta 4$

GitHub

$\Delta 1$   
git pull

$\Delta 3$   
git push

$\Delta 2$   
git pull

$\Delta 3$   $\Delta 4$   
git push

M  $\Delta 1$   $\Delta 3$   $\Delta 2$   $\Delta 4$

Partner

git add  
git commit

$\Delta 3$

( Merge )

git add  
git commit

$\Delta 4$



# Föreläsning 7

---

Tobias Wrigstad

*Testning*  
*CUnit*



"Software bugs cost the U.S. economy  
an estimated \$59.5 billion per year.

An estimated \$22.2 billion  
could be eliminated by  
**improved** testing that enables  
earlier and more effective  
identification and removal of defects."

- US Department of Commerce (NIST)



# Finn felet!

---

```
int count_spaces(char *str)
{
    int length = strlen(str);
    int count = 0;
    for(int i = 0; i <= length; ++i)
    {
        if(str[i] == ' ')
            count++;
    }
    return(count);
}
```

# Finn felet!

---

```
int count_spaces(char *str)
{
    int length = strlen(str);
    int count = 0;
    for(int i = 0; i <= length; ++i)
    {
        if(str[i] == ' ')
            count++;
    }
    return(count);
}
```

**Bug: <= istf < i for-loopen**

Inte helt enkel att upptäcka, t.ex.  
str = "A B C" ser inte felet, men  
str = " L" ser det

Testning kan bara  
påvisa förekomsten av  
fel, inte avsaknaden

— *Dijkstra*





**Ett test som inte kan  
utföras automatiskt är  
ett dåligt test!**



När du testar  $P$  är ditt  
mål inte att bekräfta  
att  $P$  är felfritt...



...utan att hitta  
så många fel som  
möjligt i P



# Vad är ett test?

---

- Låt säga att vi vill testa  $f(x) = y$  för någon given "funktion"  $f$

$x$  är indata

$y$  är utdata

$x$  och  $y$  **tillsammans** är ett **test**, eller **testfall**

- Exempelvis, för  $f = ++$  (prefix)

$++42 = 43$   $(x = 42, y = 43)$

$++0 = 1$

$++-1 = 0$

$++INT\_MAX = INT\_MIN$  *// kompilerar dock ej, men tanken*

Skall vi verkligen göra  $2^{32}$  tester?

# Varför har vi valt just dessa testfall?

---

x	y	Anmärkning
42	43	Vanlig aritmetik
0	1	Gräns, från $\pm$ till positiv
-1	0	Gräns, från negativ till $\pm$
INT_MAX	INT_MIN	Gräns, från positiv till negativ

# Testdesign

---

- Det är svårt att skriva tester
- Kräver kunskap om domänen

Vilka är gränfallen?

Vilka är de vanliga felen?

- Två syner på test
  - *Hur kan jag ha sönder programmet?*
  - *Hur kan jag förbättra programmet?*

# Den här kursen nosar bara på testning

---

- **Enhetstestning** (dagens fokus)

Pröva en liten del av programmet i isolation

- **Integrationstestning** (under projektet)

Hur olika moduler/programdelar samverkar

- **Regressionstestning** (under projektet)

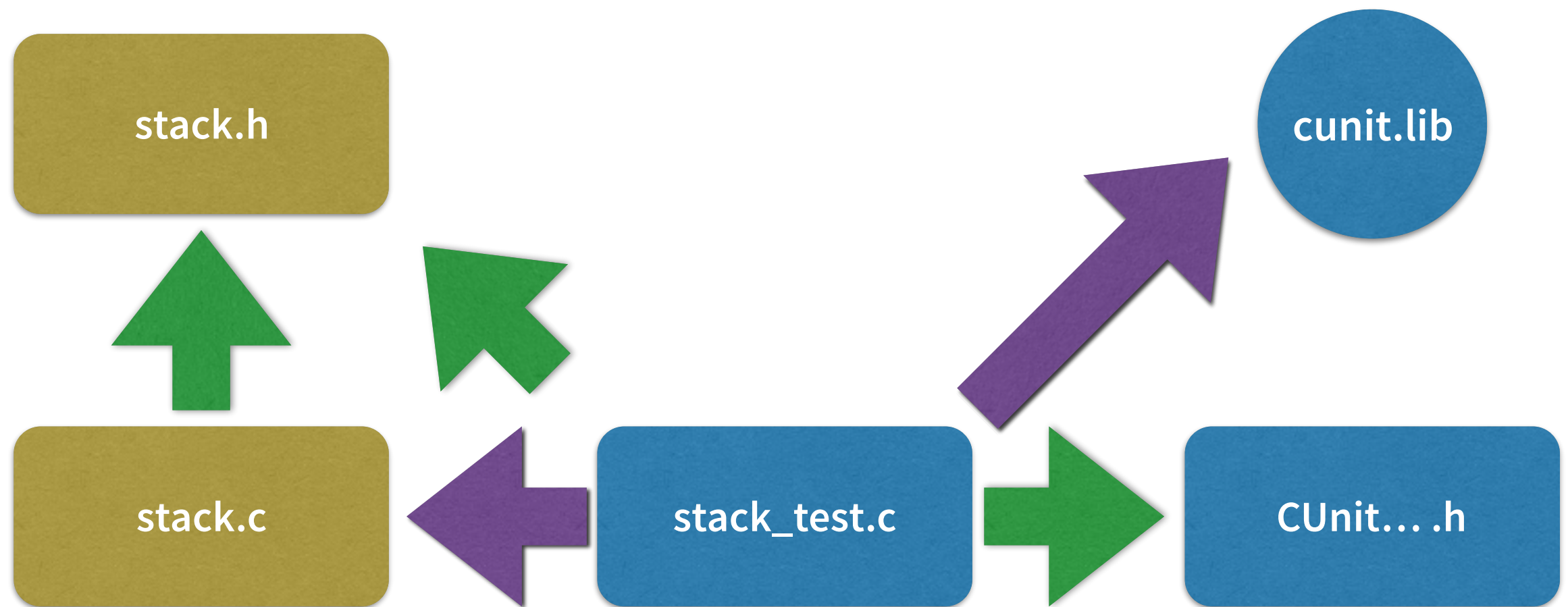
Undvik regression vid underhåll/buggfixning

(Regression = återgång till tidigare felbenäget tillstånd)

*Det som skall testas*

*Testerna*

*Testbibliotek*





# Ett tests anatomi

- En test = en funktion
- Skapa data som behövs för testet
- Utför testet

Operationer varvat med ASSERTs

- Riv ned testet

```
void test_stack_creation()
{
    int_stack_t *s = stack_new();

    CU_ASSERT_FALSE(s == NULL);
    CU_ASSERT_TRUE(stack_height(s) == 0);

    stack_free(s);
}
```

- Viktigt att testa så litet som möjligt
- Viktigt att riva skapa och riva ned vid varje test

Många verktyg (inkl. CUnit) har stöd för att göra det lättare

# Asserts (35 stycken!)

Assert	Finns Fatal?
<code>CU_ASSERT_TRUE(value)</code>	Ja
<code>CU_ASSERT_FALSE(value)</code>	Ja
<code>CU_ASSERT_EQUAL(actual, expected)</code>	Ja
<code>CU_ASSERT_NOT_EQUAL(actual, expected)</code>	Ja
<code>CU_ASSERT_PTR_EQUAL(actual, expected)</code>	Ja
<code>CU_ASSERT_PTR_NULL(value)</code>	Ja
<code>CU_ASSERT_PTR_NOT_NULL(value)</code>	Ja
<code>CU_ASSERT_STRING_EQUAL(actual, expected)</code>	Ja
<code>CU_FAIL(message)</code>	Ja

# Exempel

---

```
void *data = malloc(2048);  
CU_ASSERT_PTR_NOT_NULL(data);
```

```
char buf[256];  
scanf("%s", buf);  
char *word = my_copy_to_heap_function(buf);  
CU_ASSERT_STRING_EQUAL(buf, word);
```

```
treenode_t *t = treenode_new_leaf(key, data);  
CU_ASSERT_TRUE(t->key == key);  
CU_ASSERT_TRUE(t->data == data);  
CU_ASSERT_PTR_NULL(t->left);  
CU_ASSERT_PTR_NULL(t->right);
```

inkapsling?

# Exempel

---

```
void *data = malloc(2048);  
CU_ASSERT_PTR_NOT_NULL(data);
```

```
char buf[256];  
scanf("%s", buf);  
char *word = my_copy_to_heap_function(buf);  
CU_ASSERT_STRING_EQUAL(buf, word);
```

```
treenode_t *t = treenode_new_leaf(key, data);  
CU_ASSERT_TRUE(tn_get_key(t) == key);  
CU_ASSERT_TRUE(tn_get_data(t) == data);  
CU_ASSERT_PTR_NULL(tn_get_left(t));  
CU_ASSERT_PTR_NULL(tn_get_right(t));
```

vad testas?

# Regressionstestning

---

- En (stor) uppsättning enhetstester för olika delar av programmet
- Möjlighet att köra dem automatiskt

- Vid varje förändrings:

Kör alla enhetstester automatiskt

Notera vilka som passerar och vilka som inte passerar

Implementera förändringen

Kör alla enhetstester automatiskt igen

Stäm av mot föregående lista — verkar det rimligt?

*Table 20-2. Defect-Detection Rates*

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%
High-volume beta test (>1,000 sites)	60%	75%	85%



*Från McConnell's  
Code Complete*



**Vad skall man  
testa för något?**



```

#include <stdio.h>
#include <stdlib.h>

typedef struct stack stack_t;
typedef struct node node_t;

struct stack
{
    node_t *top;
};

struct node
{
    int value;
    node_t *next;
};

stack_t *ioopm_stack_create()
{
    return calloc(1, sizeof(stack_t));
}

void ioopm_stack_push(stack_t *s, int value)
{
    s->top = node_create(value, s->top);
}

int ioopm_stack_top(stack_t *s)
{
    return s->top->value;
}

```

```

int ioopm_stack_pop(stack_t *s)
{
    int value = ioopm_stack_top(s);
    node_t *old_top = s->top;
    s->top = s->top->next;
    free(old_top);
    return value;
}

void ioopm_stack_destroy(stack_t *s)
{
    while (s->top)
    {
        ioopm_stack_pop(s);
    }
    free(s);
}

int ioopm_stack_size(stack_t *s)
{
    int height = 0;
    for (node_t *cursor = s->top;
         cursor = cursor->next)
    {
        ++height;
    }
    return height;
}

```



# Vad är ett bra test?



# Coverage — testtäckning

---


- För att testkvaliteten skall vara hög vill vi testa så många delar som möjligt av  $f$
- För att minska kostnader vill vi undvika fler än ett test som testar samma sak

Handlar också om omöjligheten att testa alla fall

- Vi skall se gcov senare under denna föreläsning

# Strukturell testtäckning

Ut-parameter mha pekare



```
int ioopm_stack_pop(stack_t *s, int v, bool *underflow)
{
    if (s->top == NULL)
    {
        *underflow = true;
    }
    else
    {
        *underflow = false;
    }
    ...
}
```

- **Naiv strategi:** plocka slumpmässiga indata
- **Problem:** är inte säkra på att vi prövar båda vägar genom if-satsen
- **Slutsats:** vi behöver nog med testfall för att pröva båda "branches"

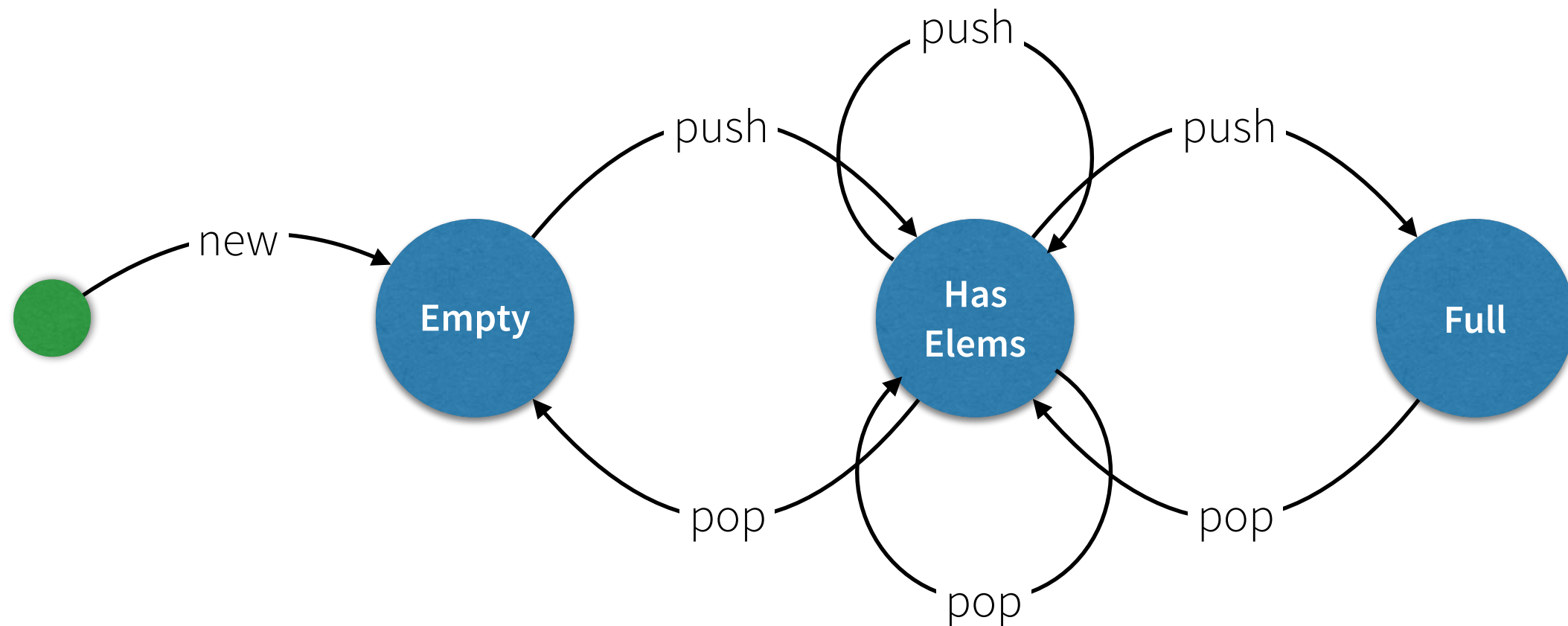
# Funktionsorienterad testtäckning

---

```
int ioopm_stack_size(int_stack_t *s)
```

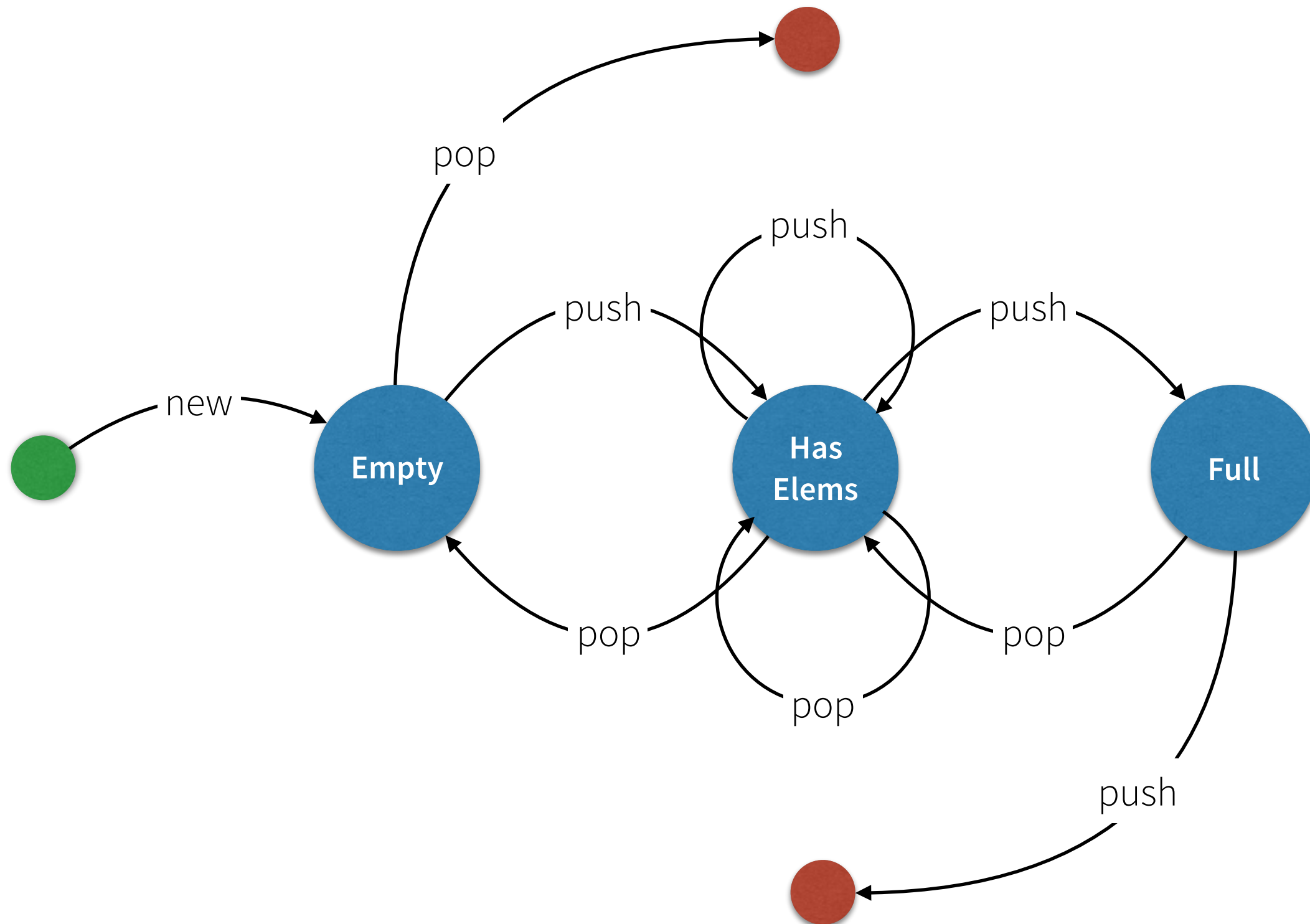
- **Naiv strategi:** skapa många stackar och pröva deras storlek
- **Problem:** inte säkert att vi prövar alla fall (tom stack, full stack, etc.)
- **Slutsats:** behöver fall som täcker alla möjliga *klasser av* input

# Beteende-orienterad testtäckning



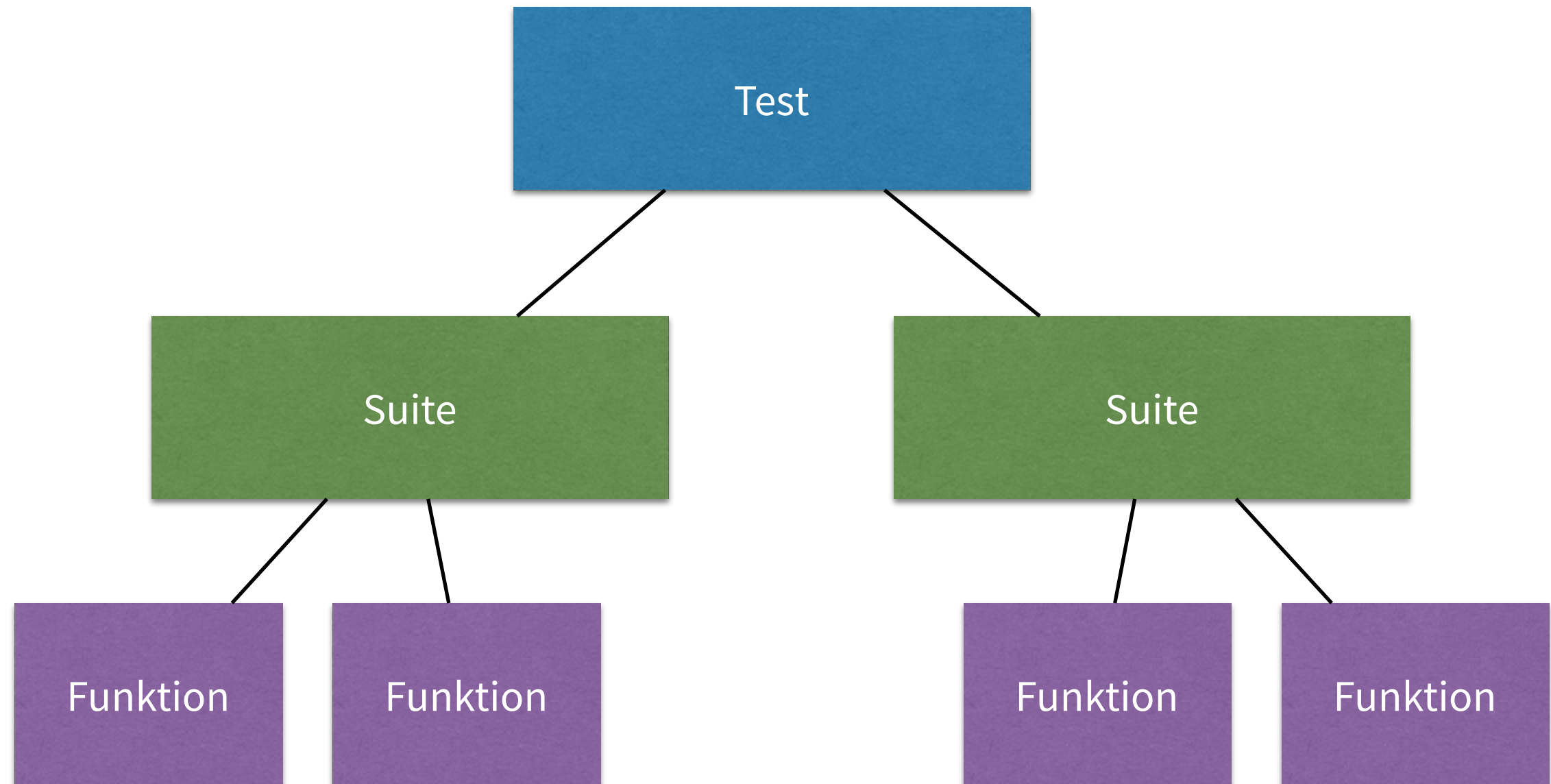
- **Naiv strategi:** plocka slumpmässiga värden på  $x$  och  $y$  och pröva  $x == y$
- **Problem:** är inte säkra på att vi prövar alla viktiga fall
- **Slutsats:** vi behöver testfall som prövar alla tillstånd och tillståndsövergångar

# Beteende-orienterad testtäckning



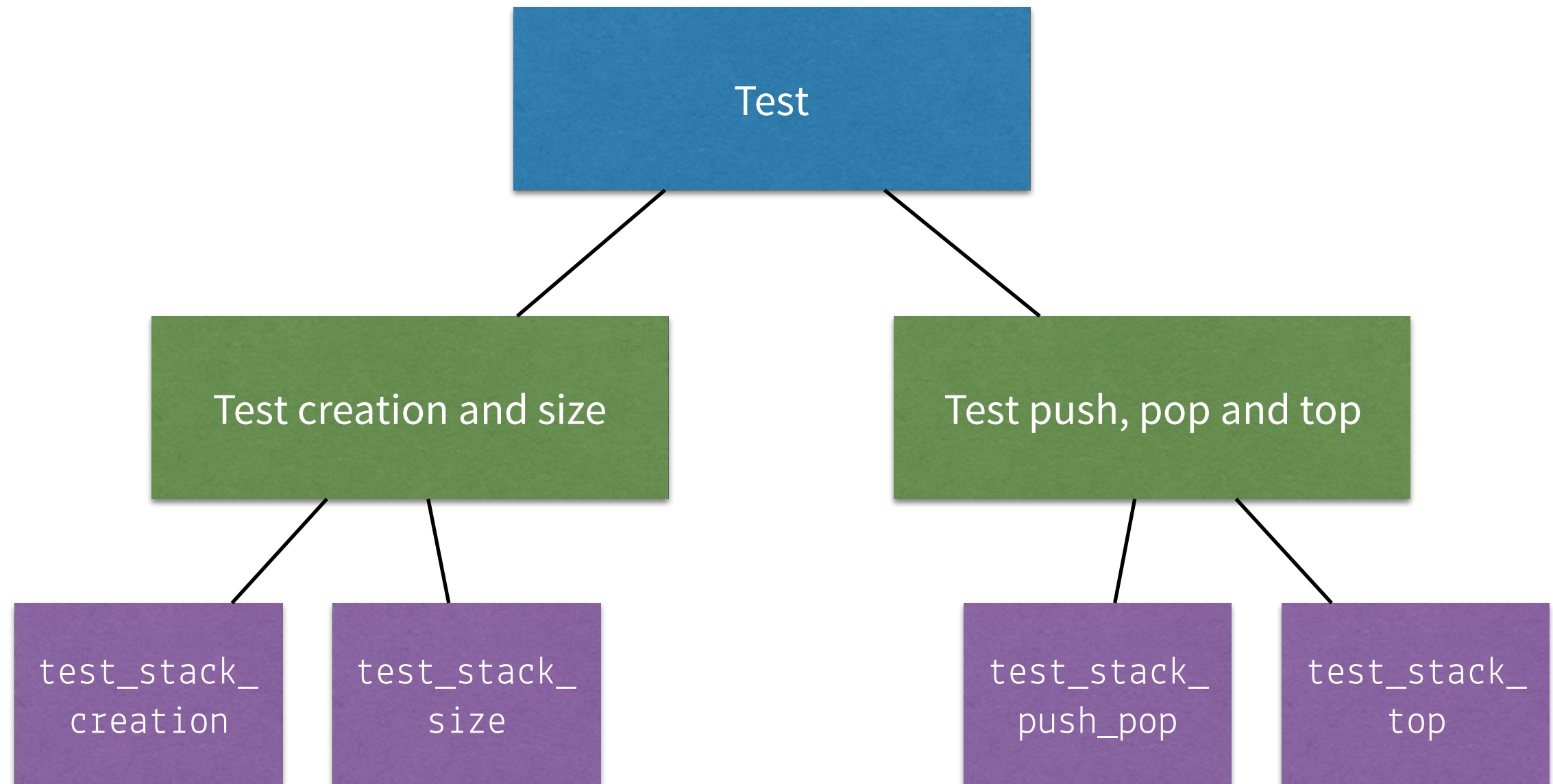
# Ett testprogram i CUnit

---



# stack\_test.c

---





# main()

```
int main(int argc, char *argv[])
{
    // Initialise
    CU_initialize_registry();

    // Set up suites and tests
    CU_pSuite creation = CU_add_suite("Test creation and height", NULL, NULL);
    CU_add_test(creation, "Creation", test_stack_creation);
    CU_add_test(creation, "Size", test_stack_size);

    CU_pSuite pushpop = CU_add_suite("Test push, pop and top", NULL, NULL);
    CU_add_test(pushpop, "Push and pop", test_stack_push_pop);
    CU_add_test(pushpop, "Top", test_stack_top);

    // Actually run tests
    CU_basic_run_tests();

    // Tear down
    CU_cleanup_registry();
    return 0;
}
```

# Skapa en stack

---

```
void test_stack_creation()
{
    stack_t *s = ioopm_stack_create();

    CU_ASSERT_FALSE(s == NULL);
    CU_ASSERT_TRUE(ioopm_stack_size(s) == 0);

    ioopm_stack_destroy(s);
}
```

# Stackhöjden

---

```
void test_stack_size()
{
    stack_t *s = ioopm_stack_create();

    CU_ASSERT_TRUE(ioopm_stack_size(s) == 0);
    ioopm_stack_push(s, 0);
    CU_ASSERT_TRUE(ioopm_stack_size(s) == 1);
    ioopm_stack_push(s, 0);
    CU_ASSERT_TRUE(ioopm_stack_size(s) == 2);
    ioopm_stack_pop(s);
    CU_ASSERT_TRUE(ioopm_stack_size(s) == 1);
    ioopm_stack_pop(s);
    CU_ASSERT_TRUE(ioopm_stack_size(s) == 0);

    ioopm_stack_destroy(s);
}
```

# Lägga till och ta bort

---

```
void test_stack_push_pop()
{
    stack_t *s = ioopm_stack_create();
    const int values[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    for (int i = sizeof(values) / sizeof(values[0]); i > 0; --i)
    {
        ioopm_stack_push(s, values[i - 1]);
    }

    for (int i = 0; i < sizeof(values) / sizeof(int); ++i)
    {
        CU_ASSERT_TRUE(ioopm_stack_pop(s) == values[i]);
    }

    ioopm_stack_destroy(s);
}
```

# Titta på översta elementet

---

```
void test_stack_top()
{
    stack_t *s = ioopm_stack_create();

    ioopm_stack_push(s, 1);
    CU_ASSERT_TRUE(ioopm_stack_top(s) == 1);
    ioopm_stack_push(s, 20);
    CU_ASSERT_TRUE(ioopm_stack_top(s) == 20);
    ioopm_stack_pop(s);
    CU_ASSERT_TRUE(ioopm_stack_top(s) == 1);

    ioopm_stack_destroy(s);
}
```

```
$ gcc -o stack_test stack_test.c stack.c -lcunit
```

```
$ ./stack_test
```

CUnit - A unit testing framework for C - Version 2.1-3  
<http://cunit.sourceforge.net/>

Run Summary:	Type	Total	Ran	Passed	Failed	Inactive
	suites	2	2	n/a	0	0
	tests	4	4	4	0	0
	asserts	20	20	20	0	n/a

Elapsed time = 0.000 seconds

```
$ gcc -o stack_test stack_test.c stack.c -lcunit
```

```
$ ./stack_test
```

CUnit - A unit testing framework for C - Version 2.1-3  
<http://cunit.sourceforge.net/>

Run Summary:	Type	Total	Ran	Passed	Failed	Inactive
	suites	2	2	n/a	0	0
	tests	4	4	4	0	0
	asserts	20	20	20	0	n/a

Elapsed time = 0.000 seconds

```
$ gcc -o stack_test stack_test.c stack.c -lcunit
```

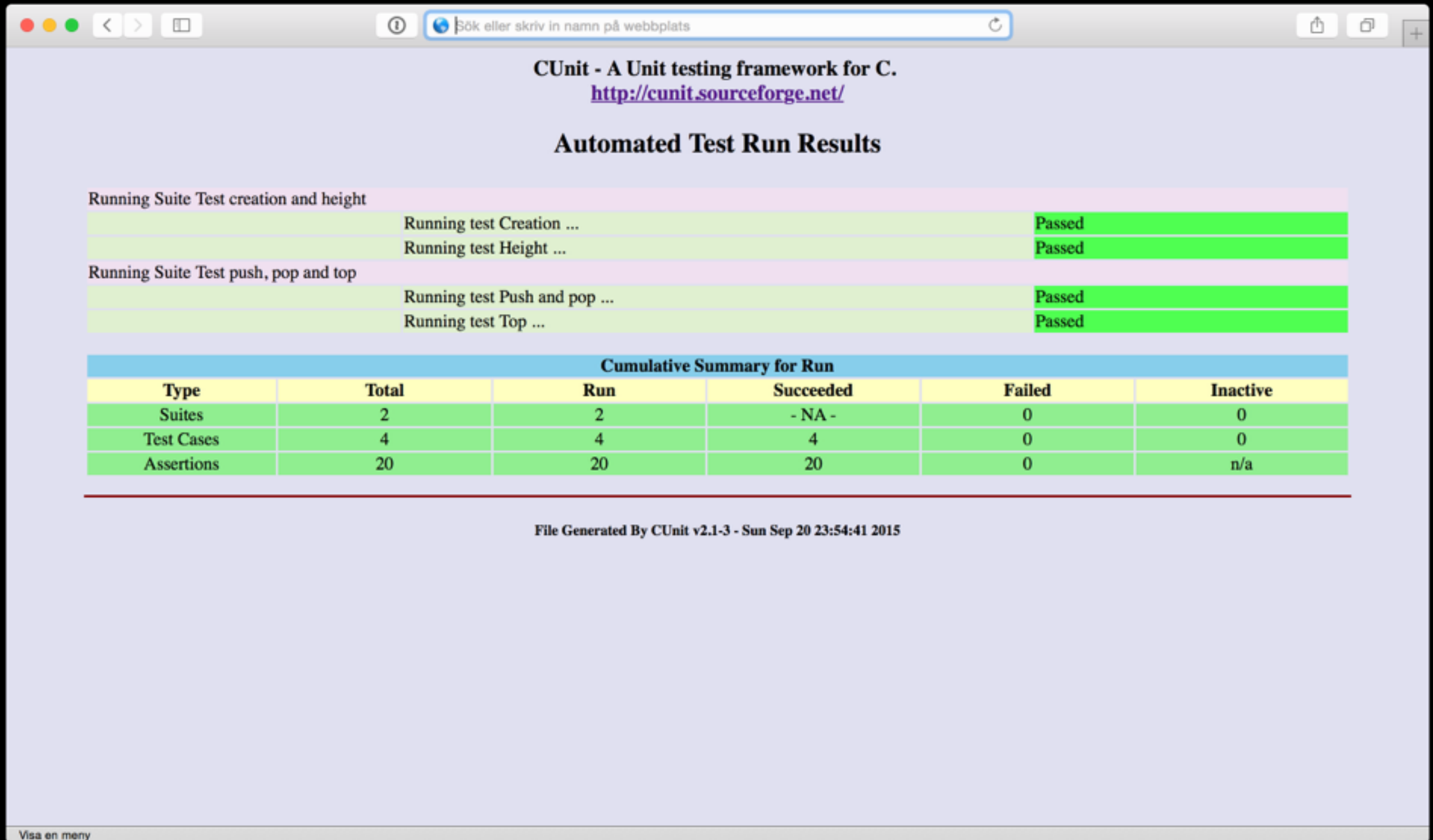
```
$ ./stack_test
```

CUnit - A unit testing framework for C - Version 2.1-3  
<http://cunit.sourceforge.net/>

Run Summary:	Type	Total	Ran	Passed	Failed	Inactive
	suites	2	2	n/a	0	0
	tests	4	4	4	0	0
	asserts	20	20	20	0	n/a

Elapsed time = 0.000 seconds





```
#include <CUnit/Automated.h>
CU_automated_run_tests(); // run and generate XML file
```

```
$ gcc --coverage -o stack_test stack_test.c stack.c -lcunit
```

```
$ gcc --coverage -o stack_test stack_test.c stack.c -lcunit
```

```
$ ./stack_test
```

```
...
```

```
$ ./gcov stack_test.c stack.c
```

```
File 'stack_test.c'
```

```
Lines executed:100.00% of 45
```

```
stack_test.c:creating 'stack_test.c.gcov'
```

```
File 'stack.c'
```

```
Lines executed:72.41% of 29
```

```
stack.c:creating 'stack.c.gcov'
```

```
$ gcov -abcfu stack.c
Function 'stack_new'
Lines executed:100.00% of 5
Branches executed:100.00% of 2
Taken at least once:50.00% of 2
No calls
```

```
Function 'stack_free'
Lines executed:100.00% of 2
No branches
No calls
```

```
Function 'stack_push'
Lines executed:100.00% of 6
Branches executed:100.00% of 2
Taken at least once:100.00% of 2
No calls
```

```
Function 'stack_pop'
Lines executed:100.00% of 6
Branches executed:100.00% of 2
Taken at least once:100.00% of 2
No calls
```

*a = all blocks*

*b = branch probabilities*

*c = branch counts*

*f = function summaries*

*u = unconditional branches*


```
$ gcov -abcfu stack.c
Function 'stack_new'
Lines executed:100.00% of 5
Branches executed:100.00% of 2
Taken at least once:50.00% of 2
No calls
```

```
Function 'stack_free'
Lines executed:100.00% of 2
No branches
No calls
```

```
Function 'stack_push'
Lines executed:100.00% of 6
Branches executed:100.00% of 2
Taken at least once:100.00% of 2
No calls
```

```
Function 'stack_pop'
Lines executed:100.00% of 6
Branches executed:100.00% of 2
Taken at least once:100.00% of 2
No calls
```

```
stack_t *ioopm_stack_new()
{
    stack_t *result = malloc(...);
    if (result)
    {
        *result = (stack_t) { .height = 0 };
    }
    return result;
}
```



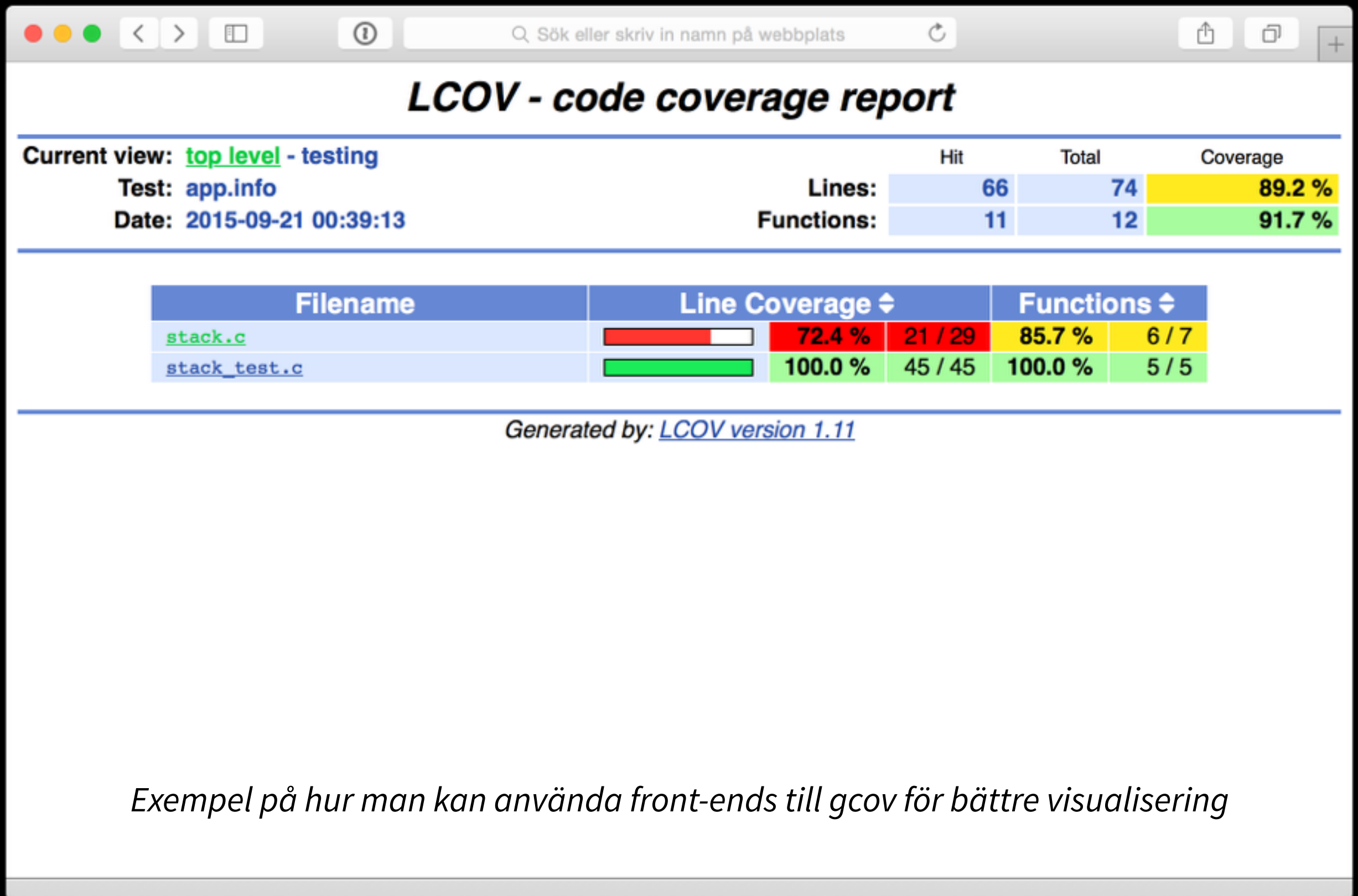
*a = all blocks*

*b = branch probabilities*

*c = branch counts*

*f = function summaries*

*u = unconditional branches*



# Öppna problem ännu så länge

---

- Hur kontrollerar vi sido-effekter?

T.ex. `ioopm_stack_destroy()`

Utskrifter i terminalfönstret eller filer skrivna på hårddisken

# Skapa era egna enhetstester

---

- Utgå gärna från `stack_test.c` som en mall för era egna tester
- Låt er inspireras av de ASSERTs som finns i CUnit-dokumentationen
- **Gränsvärden!**
- Tänk på att mäta code coverage för att få en uppfattning om testens kvalitet
- Automatisera allt! (Se exempel från denna föreläsnings utdelade material)

```
make test
```



# Några tips vid test av länkad lista

---

- Tomma listan
- $f$  i början, i mitten och i slutet

$f$  = borttagning, insättning, etc.

- Inkapsling

Växer storleken som den skall?

Testa sortering genom att göra insättning och kolla index

Testa dublettshantering genom att kolla om storleken växer vid dublettinsättning

...

# Några tips vid test av binärt sökträd

---

- Tomma trädet
- $f$  vid ingen förälder men barn, vid både förälder och barn, och bara förälder

$f$  = borttagning, insättning, etc.

- Inkapsling

Växer djup och storlek som de skall?

Testa balansering med hjälp av djupet

Testa dubletter med hjälp av storlek

Kan tillhandahålla funktioner för att kontrollera "layout" etc. utan direkt åtkomst

# Exempel: testa lagerdatabas

---

- Börja med kraven!

Dubletter?  
Kortaste avstånd?  
...etc.

- Den hypotetiska funktionen "Packa en pall"

Lägg i en vara, kolla att pris stämmer  
Lägg i en till, ändrades priset korrekt  
Vad händer vid borttagning?  
...etc.

- Notera alla specialfall i din kod och testa extra för dem!

# Testdriven utveckling

---

- Vid implementation av funktionen  $f$ , ta reda på vad  $f$  skall göra

Skriv ned detta i termer av tester

Gör detta innan du börjar implementera funktionen

- Börja implementera, målet är att få testerna att passera

Varje testfall kan vara en enhet att implementera

Du har tydliga kriterier för när du är klar

- Svårt att göra i praktiken — öva och ge inte upp!

# Föreläsning 7–8

---

Tobias Wrigstad

*Byggverktyg*



```
$ gcc stack_test.c stack.c -lcunit
$ ./a.out
SEGMENTATION FAULT...
$ gdb ./a.out...
$ gcc -g stack_test.c stack.c -lcunit
$ ./a.out
SEGMENTATION FAULT...
$ gdb ./a.out...
$ emacs stack.c
$ gcc -g stack_test.c stack.c -lcunit
$ ./a.out
SEGMENTATION FAULT...
$ gdb ./a.out...
$ emacs stack.c
$ gcc stack_test.c stack.c -lcunit
$ gcc -g stack_test.c stack.c -lcunit
$ ./a.out
SEGMENTATION FAULT...
$ gdb ./a.out...
$ emacs stack.c
$ gcc -g stack_test.c stack.c -lcunit
$ ./a.out
SEGMENTATION FAULT...
$ gdb ./a.out...
```

# Varför byggverktyg?

---

- Ett program består ofta av många filer
- Vi vill kunna bygga programmet på olika sätt  
e.x: för produktion (-O2), för test (-g --coverage), etc.
- Vi vill bara bygga om de filer som behöver byggas om efter en ändring  
för att minimera byggtiden
- Vi vill enkelt kunna ändra på vilka flaggor som används för att bygga programmet  
e.x.: lägga till en länkflagga

# Makefiler

mål

beroenden (ofta målfiler i andra mål)

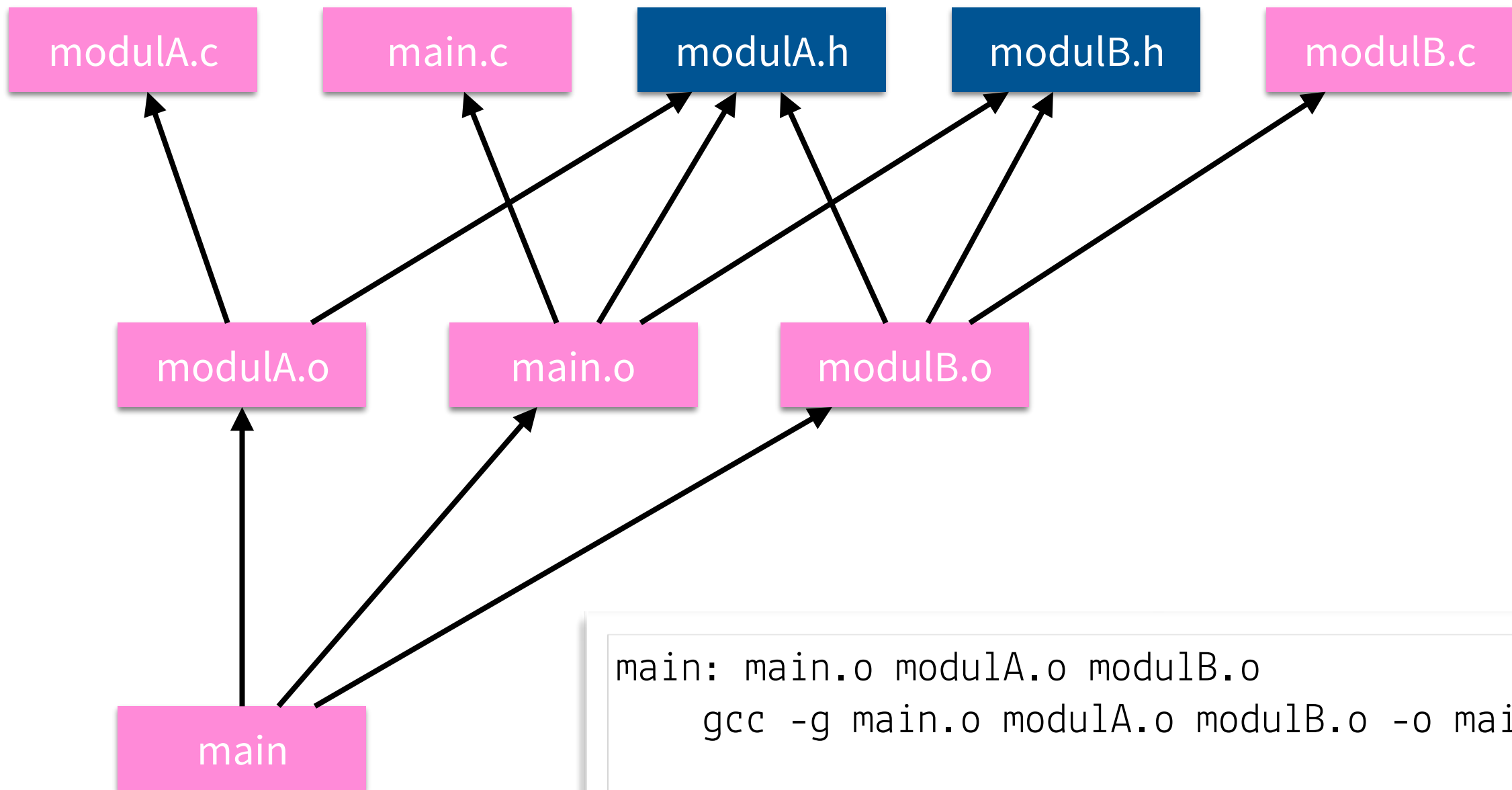
```
main: main.o modulA.o modulB.o  
      gcc -g main.o modulA.o modulB.o -o main
```

kommando som utförs om målfilen inte  
finns eller är äldre än någon av de filer den  
är beroende av

Döps till Makefile, kör med make







Exempel:

```
make main.o
make main
```

```
main: main.o modulA.o modulB.o
    gcc -g main.o modulA.o modulB.o -o main

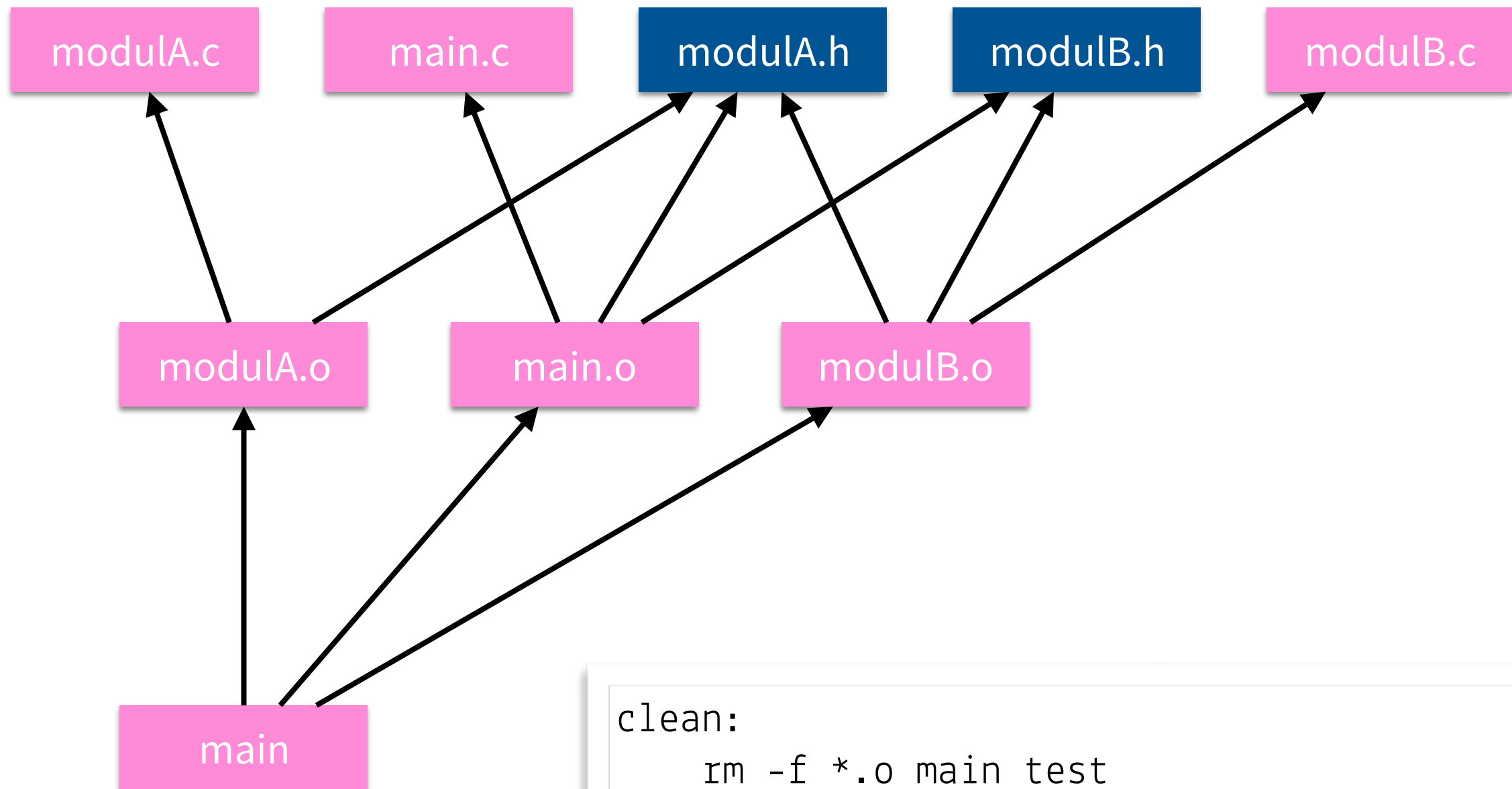
main.o: main.c modulA.h modulB.h
    gcc -c -g -Wall main.c

modulA.o: modulA.c modulA.h
    gcc -c -g -Wall modulA.c

modulB.o: modulB.c modulA.h modulB.h
    gcc -c -g -Wall modulB.c
```

Makefile





Exempel:

make test  
make clean

```
clean:  
    rm -f *.o main test  
  
memtest: test  
    valgrind --leak-check=full ./test < input  
  
test: modulA.o modulB.o tests.c  
    gcc modulA.o modulB.o tests.c -o test  
    ./test
```

Makefile



# Variabler och wildcards

```
main: myprog
```

```
C_COMPILER      = gcc
```

```
C_OPTIONS       = -Wall -pedantic -g
```

```
C_LINK_OPTIONS  = -lm
```

```
CUNIT_LINK      = -lcunit
```

```
%.o: %.c
```

```
$(C_COMPILER) $(C_OPTIONS) $? -c
```

```
myprog: file1.o file2.o file3.o
```

```
$(C_COMPILER) $(C_LINK_OPTIONS) $? -o $@
```

```
test1: mytests1.o file1.o
```

```
$(C_COMPILER) $(C_LINK_OPTIONS) $(CUNIT_LINK) $? -o $@
```

```
clean:
```

```
rm -f *.o myprog
```



# Make är inte världens bästa byggverktyg

---

- Funkar jättebra för t.ex. C
- Funkar inte så bra för Java, t.ex. — för att namnstandarderna inte funkar
- Finns många alternativ: cmake, premake, ant, etc.

Många DSL:er kompilerar *till* make

- Korsar den bron när du kommer till den — nu räcker make fint!

# Föreläsning 8

---

Tobias Wrigstad

*Profilering och optimering*



# Nya bilder 2018

