

Imperativ och objektorienterad programmeringsmetodik

Föreläsning 7 av många

Tobias Wrigstad

Testning



Vad är mjukvarutestning?

Ett sätt att förstå och dokumentera ”krav” och förväntat beteende hos ett system

Programkod beskriver saker entydigt

Programkod kan exekveras för att testa om beteendet är det förväntade

Ett sätt att hitta defekter (kollokvialt: buggar) i kod under utveckling

Defekter är i det generella fallet oundvikliga vid utveckling

Många kockar redar samma source

Krav eller omständigheter förändras över tid

Tester minskar riskerna med refaktorering och explorativ programmering

*Testning kan enbart
påvisa förekomsten av fel
– inte avsaknaden*

*Ett test som inte kan
automatiseras är ett
dåligt (farligt) test!*



Vad är mjukvarutestning? – olika typer och dimensioner av testning

- Whitebox — när koden är känd
- Blackbox — när endast gränssnittet är känt

- Testning av **funktionella krav**

Enhetstester, integrationstest, smoke test, acceptanstest, ...

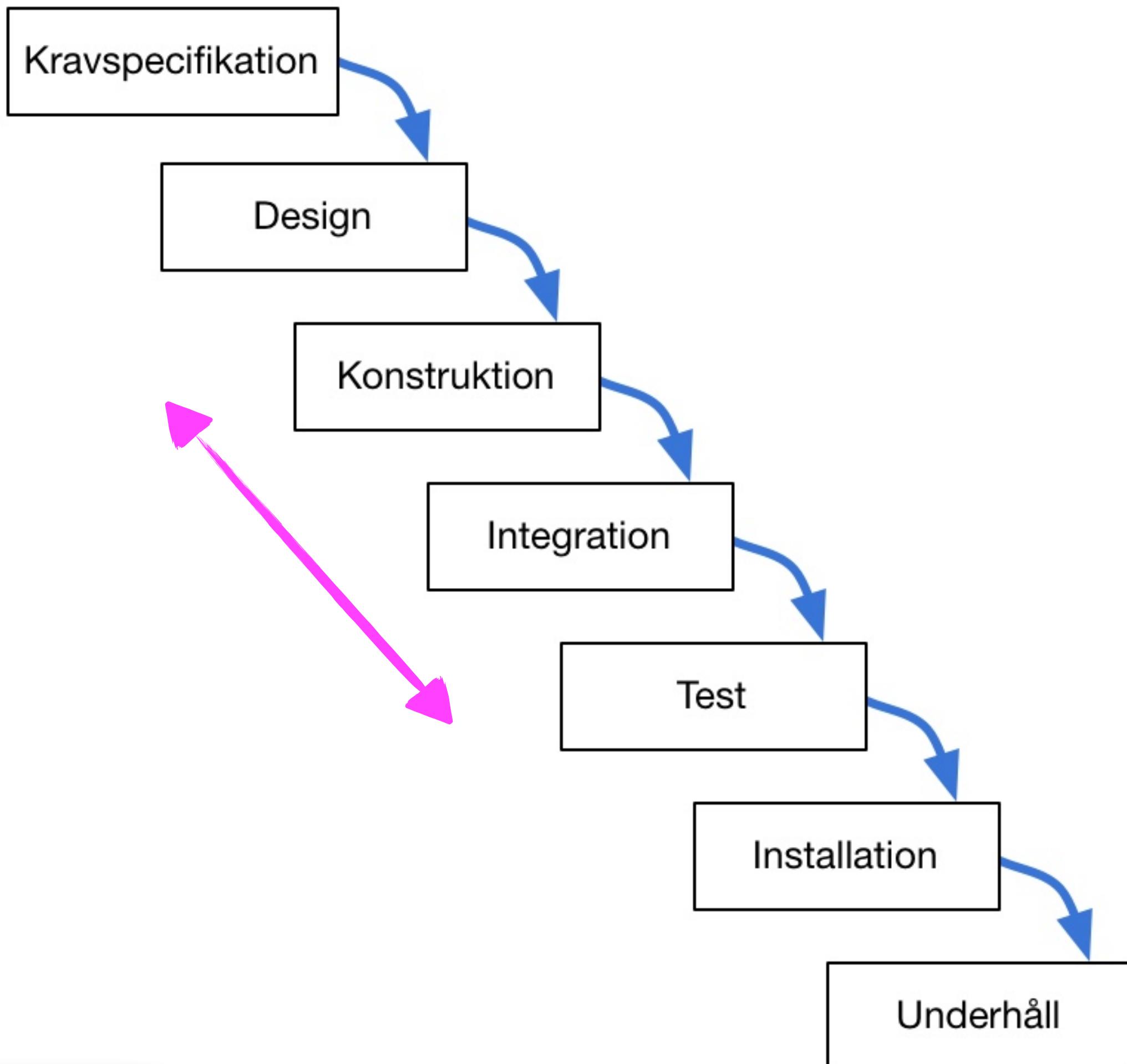
- Testning av **icke-funktionella krav**

Prestanda, stresstest, skalbarhet, usability, ...

- Underhållstestning

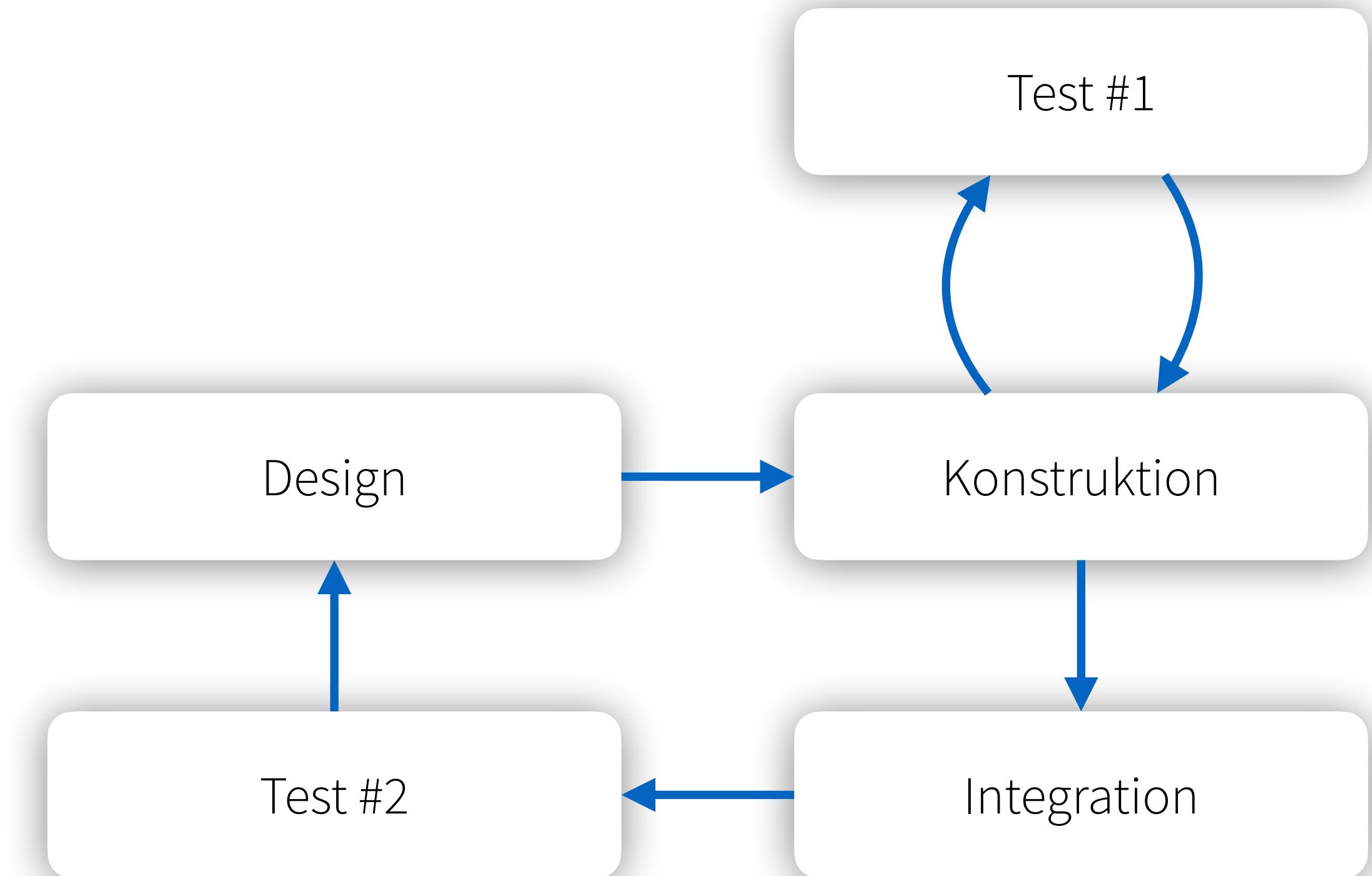
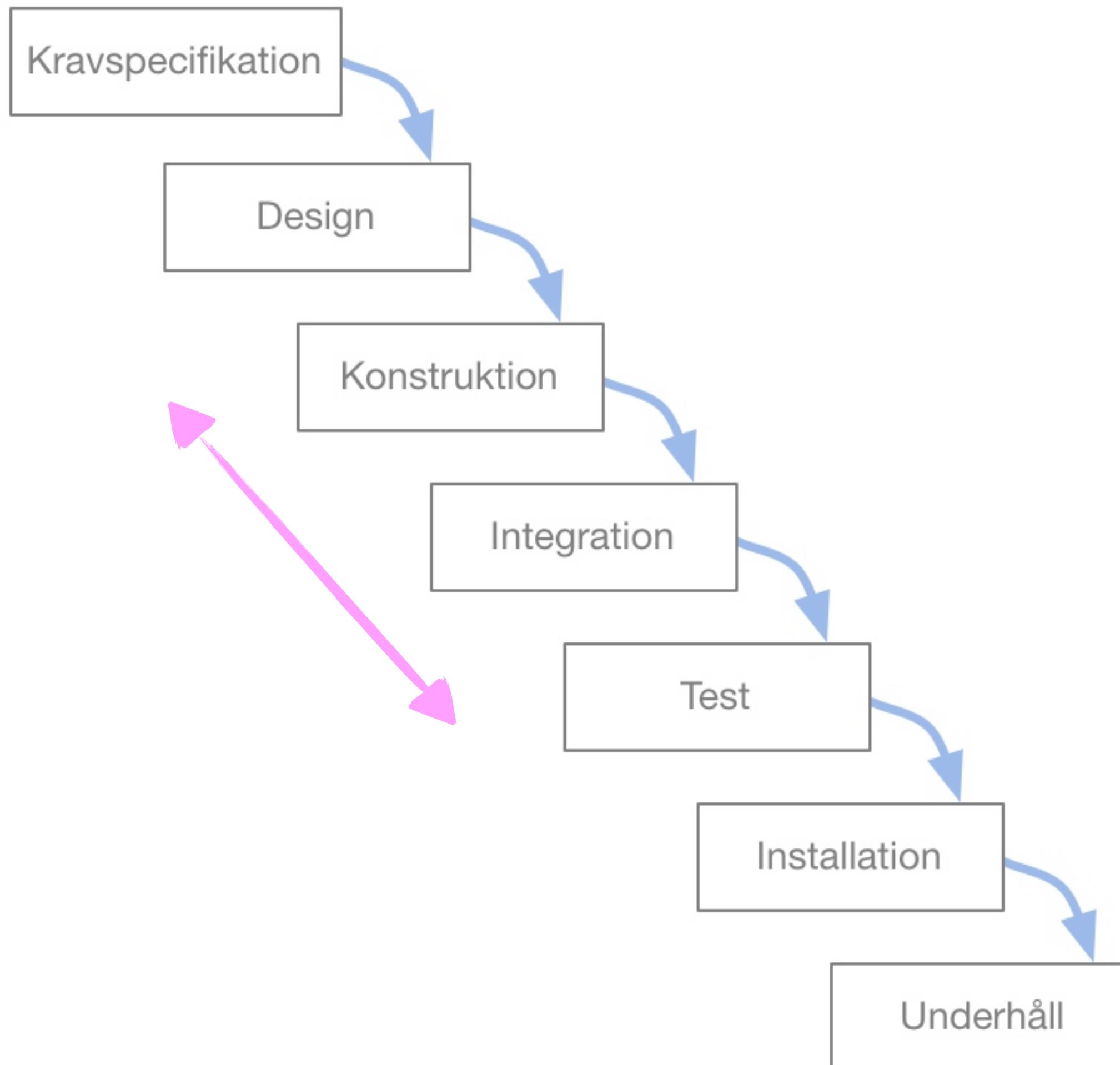
Regressionstest

En ålderdomlig och förlegad syn på testning



Vi kommer att prata mer om processer, livcykelmodeller och utvecklingsmetoder senare under kursen

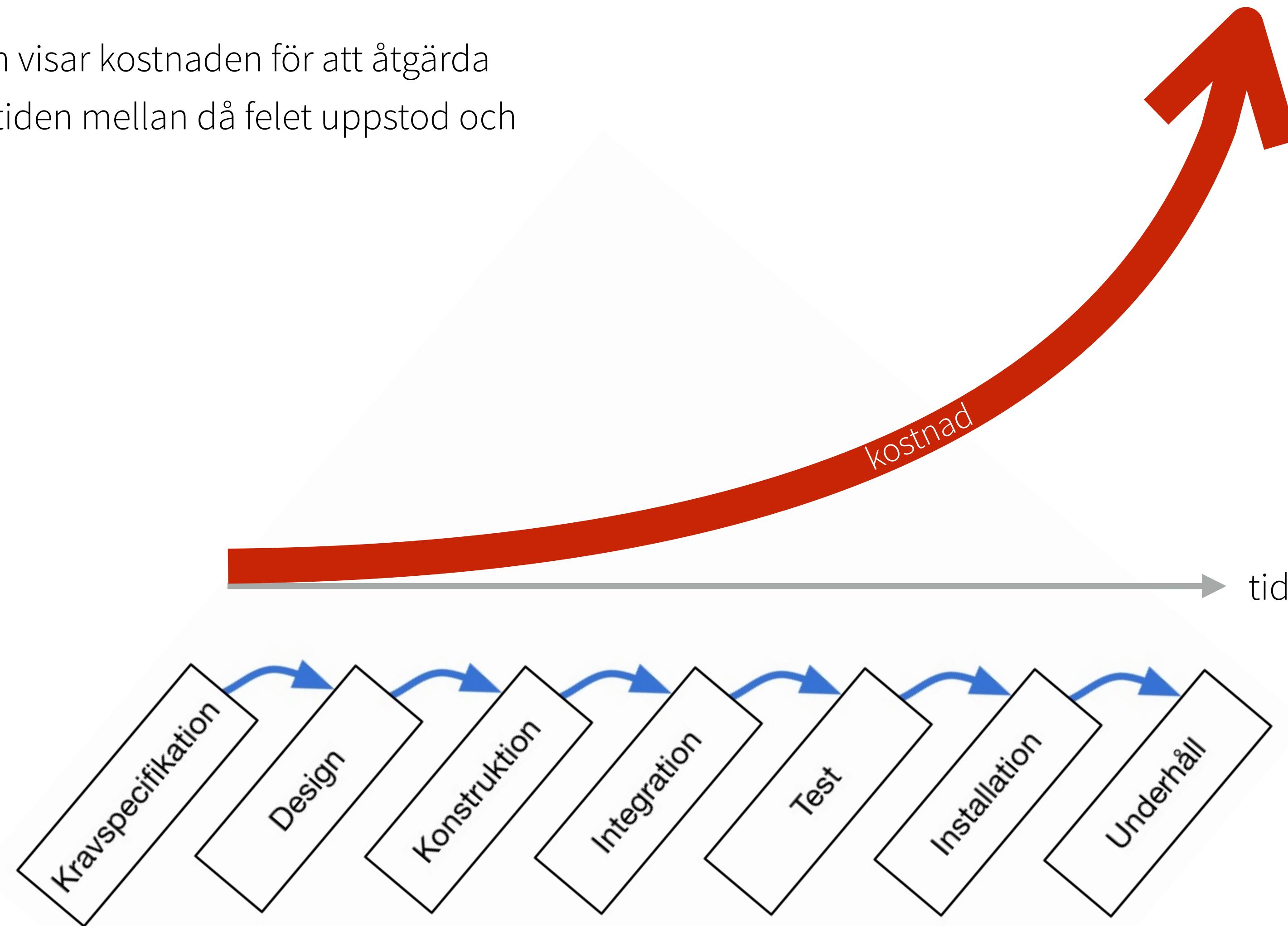
En ålderdomlig och förlegad syn på testning



Modern utveckling närmar sig problemet gradvist
(med vissa undantag förstås)

Kostnaden för att fixa fel

En schematisk bild som visar kostnaden för att åtgärda fel som en funktion av tiden mellan då felet uppstod och då det upptäcktes





Den här kurser nosar bara på testning

- **Enhetstestning** (dagens fokus)

Pröva en liten del av programmet i isolation

- **Integrationstestning** (från och med steg 10 i inlämningsuppgift 1)

Hur olika moduler/programdelar samverkar

- **Regressionstestning** (från och med inlämningsuppgift 2)

Undvik regression vid underhåll/bugfixning

(Regression = återgång till tidigare felbenäget tillstånd)

Enhetstestning

Test av förväntat beteende (utdata) givet visst indata

Testar även förväntat felbeteende

Testa små enheter så att när ett test går sönder skall det vara otvetydigt vart problemet uppstått

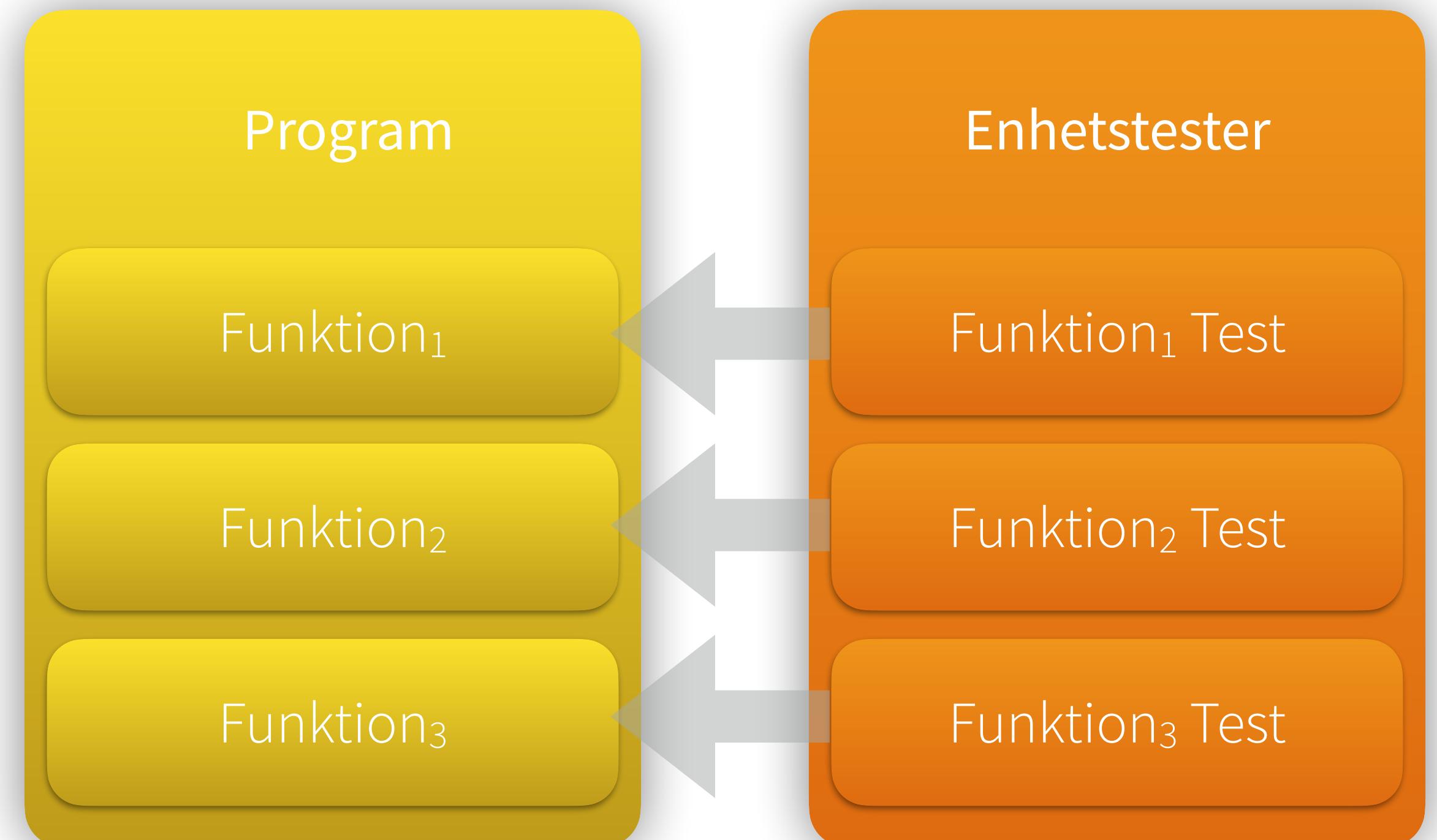
Enkelt uttryck är ett test ett par

Indata

Förväntat utdata

Vi vill testa $f(\text{indata}) == \text{förväntat utdata}$?

Ofta kan det krävas lite extra arbete att skapa rätt förutsättningar för att köra testen



Kort recap: representation av helta (i C)

- Helta i C lagras binärt som en sekvens av 1:or och 0:or

Kort recap: representation av helta (i C)

- Heltal i C lagras binärt som en sekvens av 1:or och 0:or
- Av pragmatiska och praktiska skäl har ett helta en fix längd, t.ex. en int representeras som 32 bitar på en viss maskin

Vi har många olika intar av olika längd för olika tillfällen

Kort recap: representation av helta (i C)

- Heltal i C lagras binärt som en sekvens av 1:or och 0:or
- Av pragmatiska och praktiska skäl har ett helta en fix längd, t.ex. en int representeras som 32 bitar på en viss maskin

Vi har många olika intar av olika längd för olika tillfällen

- Om en datatyps storlek är fix finns det också en övre gräns för hur många olika tal (i detta fall) som kan lagras i typen

`sizeof(int) = 4 bytes = 32 bitar = 2^{32} möjliga värden`

Kort recap: representation av heltalet (i C)

- Heltalet i C lagras binärt som en sekvens av 1:or och 0:or
- Av pragmatiska och praktiska skäl har ett heltalet en fix längd, t.ex. en int representeras som 32 bitar på en viss maskin

Vi har många olika intar av olika längd för olika tillfällen

- Om en datatyps storlek är fix finns det också en övre gräns för hur många olika tal (i detta fall) som kan lagras i typen
 $\text{sizeof(int)} = 4 \text{ bytes} = 32 \text{ bitar} = 2^{32}$ möjliga värden
- Samma sak gäller för flyttal där man också måste ta med precision i beräkningen

Kort recap: representation av heltalet (i C)

- Heltalet i C lagras binärt som en sekvens av 1:or och 0:or
- Av pragmatiska och praktiska skäl har ett heltalet en fix längd, t.ex. en int representeras som 32 bitar på en viss maskin

Vi har många olika intar av olika längd för olika tillfällen

- Om en datatyps storlek är fix finns det också en övre gräns för hur många olika tal (i detta fall) som kan lagras i typen
`sizeof(int) = 4 bytes = 32 bitar = 2^{32} möjliga värden`
- Samma sak gäller för flyttal där man också måste ta med precision i beräkningen

```
#include <stdio.h>

int main(void) {
    printf ("%-.20f\n", 3.6);
    return 0;
}
```

Kort recap: representation av heltalet (i C)

- Heltalet i C lagras binärt som en sekvens av 1:or och 0:or
- Av pragmatiska och praktiska skäl har ett heltalet en fix längd, t.ex. en int representeras som 32 bitar på en viss maskin

Vi har många olika intar av olika längd för olika tillfällen

- Om en datatyps storlek är fix finns det också en övre gräns för hur många olika tal (i detta fall) som kan lagras i typen
`sizeof(int) = 4 bytes = 32 bitar = 2^{32} möjliga värden`
- Samma sak gäller för flyttal där man också måste ta med precision i beräkningen

```
#include <stdio.h>

int main(void) {
    printf ("%.20f\n", 3.6);
    return 0;
}
```



3.6000000000000008882

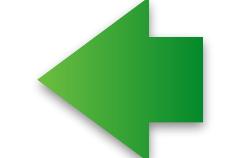
Demo: Test av operationer på heltal

- 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)

$$++42 = 43$$

($x=42, y=43$)

$$++0 = 1$$

$$++-1 = 0$$

$$++\text{INT_MAX} = \text{INT_MIN}$$



Vi kan testa alla fall!



Skall vi verkligen göra 2^{32} tester?

Hur funkar en funktion?

```
int prefix_increment(int input)
{
    if (input == 1) return 2;
    if (input == 2) return 3;
    if (input == 3) return 4;
    if (input == 4) return 5;
    if (input == 5) return 6;
    if (input == 6) return 7;
    ...
    if (input == 1024) return 1024;
    if (input == 1025) return 1026;
    if (input == 1026) return 1027;
    ...
}
```

Design (urval) av testfall

- 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)

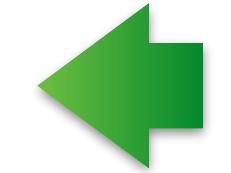
$$++42 = 43$$

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

$$++0 = 1$$

$$++-1 = 0$$

$$++\text{INT_MAX} = \text{INT_MIN}$$



| x | y | Anmärkning |
|---------|---------|----------------------------------|
| 42 | 43 | Vanlig aritmetik |
| 0 | 1 | Gräns, från ± till positiv |
| -1 | 0 | Gräns, från negativ till ± |
| INT_MAX | INT_MIN | Gräns, från positiv till negativ |

Testdesign

- Att skriva tester kan vara lika komplext som ”produktionskod”
- Effektiv testning kräver kunskap om domänen
 - Vilka är gränsfallen?
 - Vilka är de vanliga felen?
- Två olika syner på test
 - *Hur kan jag ha sönder programmet?*
 - *Hur kan jag förbättra programmet?*

Photo by Christina @ wocintechchat.com on Unsplash



Demo: låt oss bygga testfall för funktionen `trim`

- Funktionen `trim` finns i många standardbibliotek och används för att ta bort inledande och avslutande blanksteg i en sträng.

Blanksteg: ' ' (mellanslag), '\t' (tab), '\n' (newline), '\r' (carriage return), '\f' (form feed) (typiskt)

- Funktionsdesign (á la SIMPLE)

Hitta första tecknet att kopiera från, hitta sista tecknet att kopiera från — kopiera dessa till starten av strängen och lägg till '\0'

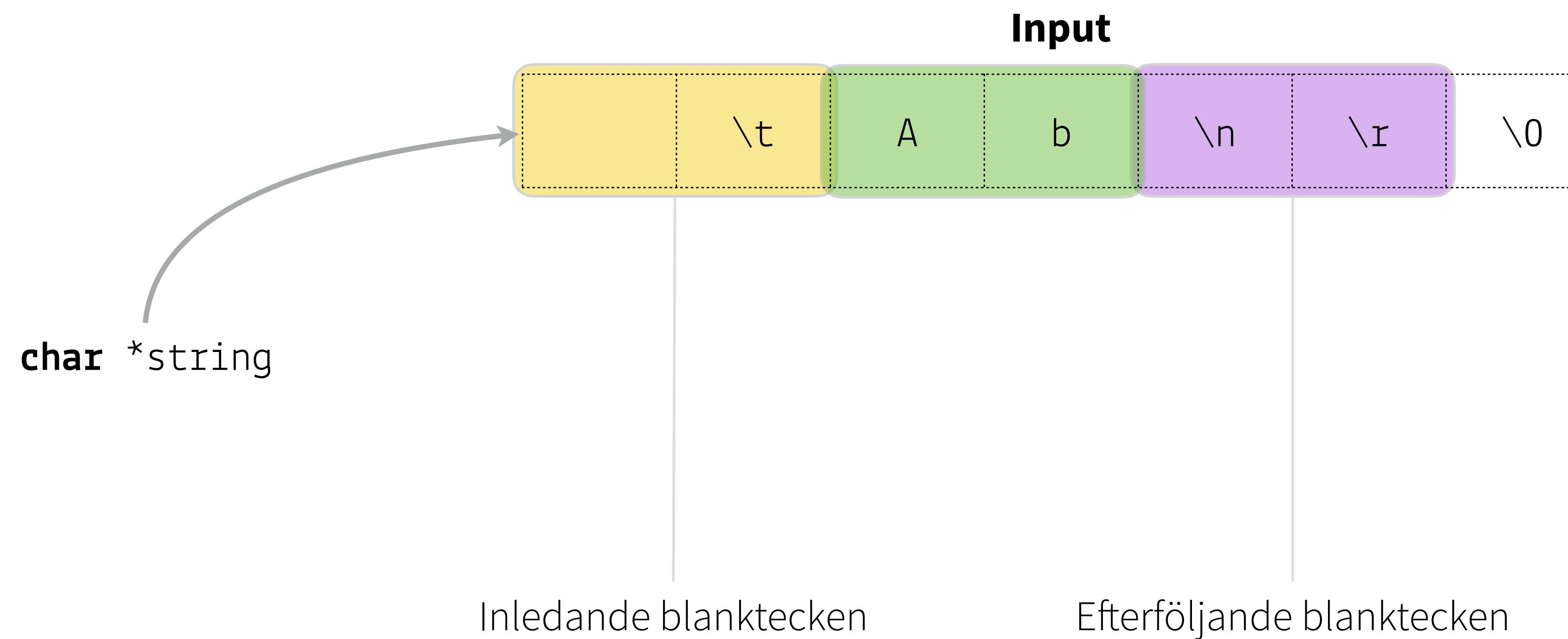
Hjälpfunktioner för att hitta första och sista tecknen som inte är blanksteg (`char *find_first_non_blank(char *s)`, etc.)

Hjälpfunktion för att avgöra om ett tecken är ett blanksteg (`bool is_blank(char c)`, etc.)

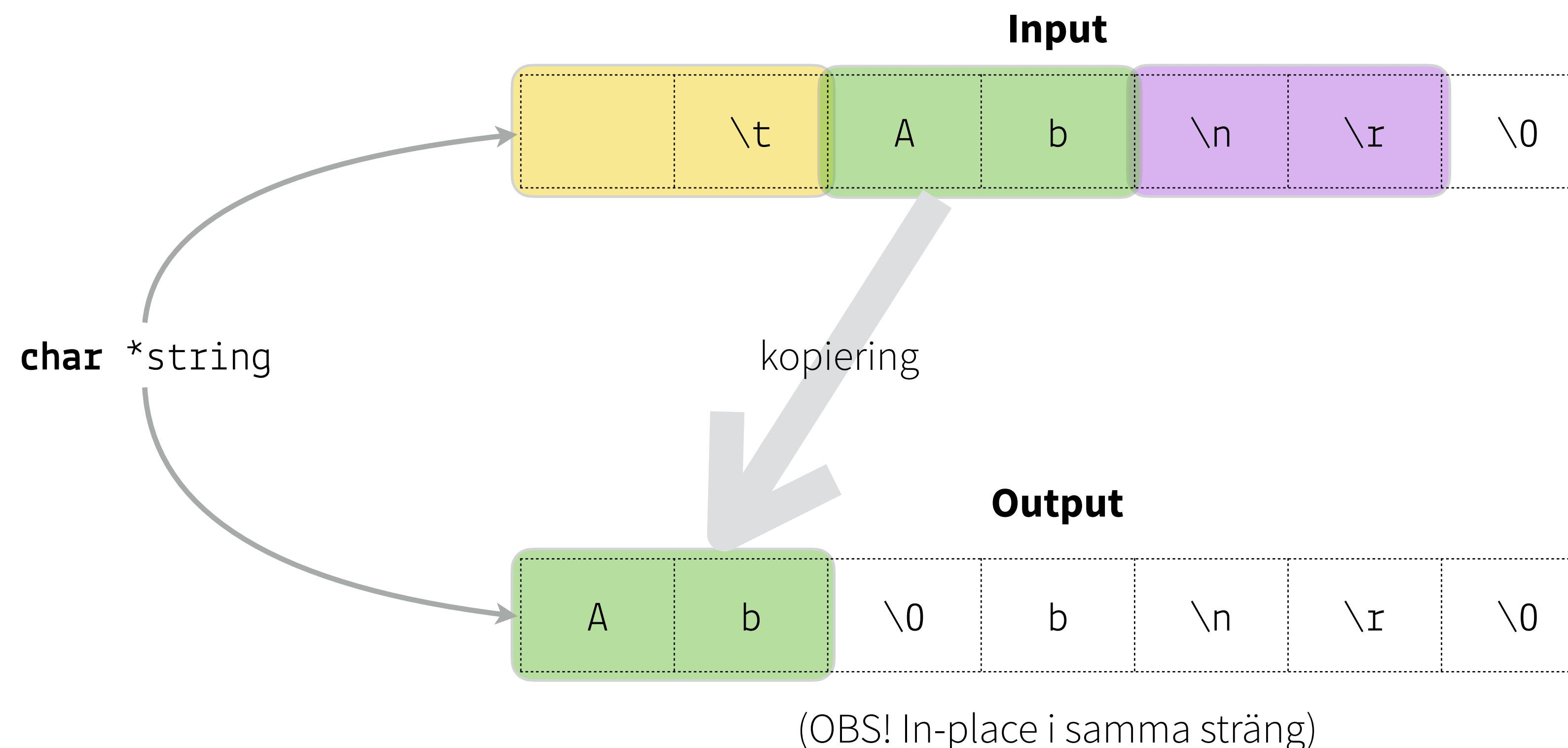
Vad är lämpligt att testa?

Vad är lämpliga testfall?

En rimlig implementation av trim



En rimlig implementation av trim



Testa bottom-up

- Vi börjar med att etablera att `is_blank(char c)` är korrekt eftersom andra funktioner bygger på denna

Vad är lämpliga testfall?

Positiva test: vi kan räkna upp de få fall för vilka funktionen bör svara `true`, så vi kan pröva samtliga

Negativa test: kan pröva samtliga andra tecken (åtminstone för vissa teckentabeller) men räcker med några olika fall

```
void test_is_blank()
{
    assert(is_blank(' '));
    assert(is_blank('\t'));
    assert(is_blank('\n'));
    assert(is_blank('\r'));
    assert(is_blank('\f'));

    char *no_blanks = "abcdefABCDEF!#%&/()_,.";
    while (*no_blanks)
    {
        assert(is_blank(*no_blanks) == false);
        ++no_blanks;
    }
    assert(is_blank('\0') == false);
}
```

Testa bottom-up

- Vi börjar med att etablera att `is_blank(char c)` är korrekt eftersom andra funktioner bygger på denna

Vad är lämpliga testfall?

Positiva test: vi kan räkna upp de få fall för vilka funktionen bör svara `true`, så vi kan pröva samtliga

Negativa test: kan pröva samtliga andra tecken (åtminstone för vissa teckentabeller) men räcker med några olika fall

Gemener

```
void test_is_blank()
{
    assert(is_blank(' '));
    assert(is_blank('\t'));
    assert(is_blank('\n'));
    assert(is_blank('\r'));
    assert(is_blank('\f'));

    char *no_blanks = "abcdefABCDEF!#%&/()_,.";
    while (*no_blanks)
    {
        assert(is_blank(*no_blanks) == false);
        ++no_blanks;
    }
    assert(is_blank('\0') == false);
}
```

Testa bottom-up

- Vi börjar med att etablera att `is_blank(char c)` är korrekt eftersom andra funktioner bygger på denna

Vad är lämpliga testfall?

Positiva test: vi kan räkna upp de få fall för vilka funktionen bör svara `true`, så vi kan pröva samtliga

Negativa test: kan pröva samtliga andra tecken (åtminstone för vissa teckentabeller) men räcker med några olika fall

Gemener

Versaler

```
void test_is_blank()
{
    assert(is_blank(' '));
    assert(is_blank('\t'));
    assert(is_blank('\n'));
    assert(is_blank('\r'));
    assert(is_blank('\f'));

    char *no_blanks = "abcdefABCDEF!#%&/()_,.";
    while (*no_blanks)
    {
        assert(is_blank(*no_blanks) == false);
        ++no_blanks;
    }
    assert(is_blank('\0') == false);
}
```

Testa bottom-up

- Vi börjar med att etablera att `is_blank(char c)` är korrekt eftersom andra funktioner bygger på denna

Vad är lämpliga testfall?

Positiva test: vi kan räkna upp de få fall för vilka funktionen bör svara `true`, så vi kan pröva samtliga

Negativa test: kan pröva samtliga andra tecken (åtminstone för vissa teckentabeller) men räcker med några olika fall

Gemener

Versaler

Specialtecken

```
void test_is_blank()
{
    assert(is_blank(' '));
    assert(is_blank('\t'));
    assert(is_blank('\n'));
    assert(is_blank('\r'));
    assert(is_blank('\f'));

    char *no_blanks = "abcdefABCDEF!#%&/()_,.";
    while (*no_blanks)
    {
        assert(is_blank(*no_blanks) == false);
        ++no_blanks;
    }
    assert(is_blank('\0') == false);
}
```

Testa bottom-up

- Vi börjar med att etablera att `is_blank(char c)` är korrekt eftersom andra funktioner bygger på denna

Vad är lämpliga testfall?

Positiva test: vi kan räkna upp de få fall för vilka funktionen bör svara `true`, så vi kan pröva samtliga

Negativa test: kan pröva samtliga andra tecken (åtminstone för vissa teckentabeller) men räcker med några olika fall

Gemener

Versaler

Specialtecken

Nulltecknet!

```
void test_is_blank()
{
    assert(is_blank(' '));
    assert(is_blank('\t'));
    assert(is_blank('\n'));
    assert(is_blank('\r'));
    assert(is_blank('\f'));

    char *no_blanks = "abcdefABCDEF!#%&/()_,.";
    while (*no_blanks)
    {
        assert(is_blank(*no_blanks) == false);
        ++no_blanks;
    }
    assert(is_blank('\0') == false);
}
```

Assertions (bra att ha för sanity, vi skall se bättre för tester snart)

- En försäkran om att ett viss villkor är uppfyllt i programmet

```
assert(P);  
assert(++x == y);
```

Om den slår ut: Assertion failed: (++x == y), function main, file my_file.c, line 5.

- Finns i C:s standartbibliotek: #include <assert.h> men även flådigare i t.ex. CUnit som vi snart skall se
- Vanligt att ha med i utvecklingskod, men inte i produktionskod (tas bort iom kompilering)

gcc -DNDEBUG ... slår av assertions i allt som kompileras

Eller #define NDEBUG på rätt ställe(n) i koden (mindre säkert men mer flexibelt)

Test av `find_first_non_blank(char *)`

- Kan definitivt inte testa alla strängar!
- Gränsfall

Test av `find_first_non_blank(char *)`

- Kan definitivt inte testa alla strängar!
- Gränsfall

Tomma strängen

Test av `find_first_non_blank(char *)`

- Kan definitivt inte testa alla strängar!
- Gränsfall

Tomma strängen

Inga blanka alls

Test av `find_first_non_blank(char *)`

- Kan definitivt inte testa alla strängar!
- Gränsfall

Tomma strängen

Inga blanka alls

Enbart blanka

Test av `find_first_non_blank(char *)`

- Kan definitivt inte testa alla strängar!
- Gränsfall

Tomma strängen

Inga blanka alls

Enbart blanka

Ett inledande blanktecken, sedan icke-blanka

Test av `find_first_non_blank(char *)`

- Kan definitivt inte testa alla strängar!
- Gränsfall

Tomma strängen

Inga blanka alls

Enbart blanka

Ett inledande blanktecken, sedan icke-blanka

Flera inledande blanka tecken, sedan icke-blanka

Test av `find_first_non_blank(char *)`

- Kan definitivt inte testa alla strängar!
- Gränsfall

Tomma strängen

Inga blanka alls

Enbart blanka

Ett inledande blanktecken, sedan icke-blanka

Flera inledande blanka tecken, sedan icke-blanka

Omväxlande

Test av `find_first_non_blank(char *)`

- Kan definitivt inte testa alla strängar!
- Gränsfall

Tomma strängen

Inga blanka alls

Enbart blanka

Ett inledande blanktecken, sedan icke-blanka

Flera inledande blanka tecken, sedan icke-blanka

Omväxlande

NULL — och vad är rätt beteende då???

Test av `find_first_non_blank(char *)`

- Kan definitivt inte testa alla strängar!
- Gränsfall

Tomma strängen

Inga blanka alls

Enbart blanka

Ett inledande blanktecken, sedan icke-blanka

Flera inledande blanka tecken, sedan icke-blanka

Omväxlande

NULL — och vad är rätt beteende då???

- Och motsvarande (spegelvänt!) för `find_last_non_blank(char *)`

Test av `find_first_non_blank(char *)`

```
void test_find_first_non_blank()
{
    char *empty = "";
    char *no_blanks = "12345";
    char *only_blanks = "      ";
    char *one_blank_first = " 1234";
    char *three_blanks_first = "    12";
    char *two_blanks_then_mix = "  1  ";

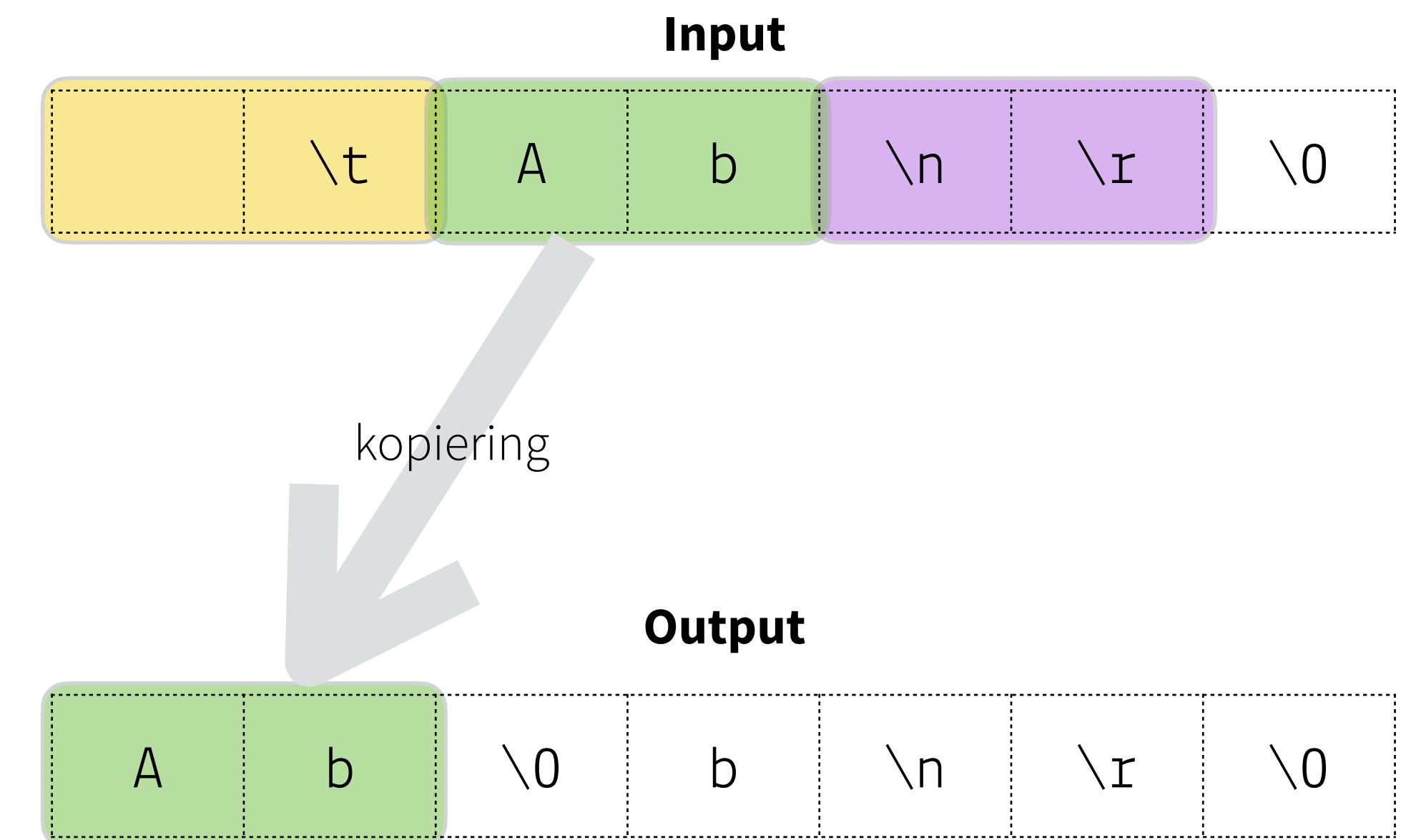
    assert(find_first_non_blank(empty)           - empty          == 0);
    assert(find_first_non_blank(no_blanks)        - no_blanks       == 0);
    assert(find_first_non_blank(only_blanks)       - only_blanks    == (long) strlen(only_blanks));
    assert(find_first_non_blank(one_blank_first)   - one_blank_first == 1);
    assert(find_first_non_blank(three_blanks_first) - three_blanks_first == 3);
    assert(find_first_non_blank(two_blanks_then_mix) - two_blanks_then_mix == 2);
}
```

Tid för eftertanke

Vad har vi?

En förståelse (black-box-style) för hur `trim` fungerar

- + En påhittad funktion `is_blank` som är ordentligt testad
- + En påhittad funktion `find_first_non_blank` som är ordentligt testad
- + En påhittad funktion `find_last_non_blank` som är ordentligt testad
- = nog för att göra testfall för `trim`



Insikt: testfall för `trim` kan härledas från `find_first_non_blank` och `find_last_non_blank`

Test av trim

- Testfallen är ”unionen” av testfallen för våra ”find-funktioner”
- För varje testfall, kontrollera

Att returadressen alltid är samma som argumentet

Att resultatsträngen har förväntat innehåll

- Viktig förbättring

Bryt upp varje assert i två

(Här motiverat av begränsat utrymme)

```
void test_trim()
{
    char empty[] = "";
    char no_blanks[] = "12345";
    char only_blanks[] = "      ";
    char one_blank_first[] = " 1234";
    char three_blanks_first[] = "    12";
    char two_blanks_then_mix[] = "   1  ";
    char one_blank_last[] = "4321  ";
    char three_blanks_last[] = "21    ";

    char *test = trim(empty);
    assert(test == empty && strcmp(test, "") == 0);

    test = trim(no_blanks);
    assert(test == no_blanks && strcmp(test, "12345") == 0);

    test = trim(only_blanks);
    assert(test == only_blanks && strcmp(test, "") == 0);

    test = trim(one_blank_first);
    assert(test == one_blank_first && strcmp(test, "1234") == 0);

    test = trim(three_blanks_first);
    assert(test == three_blanks_first && strcmp(test, "12") == 0);
```

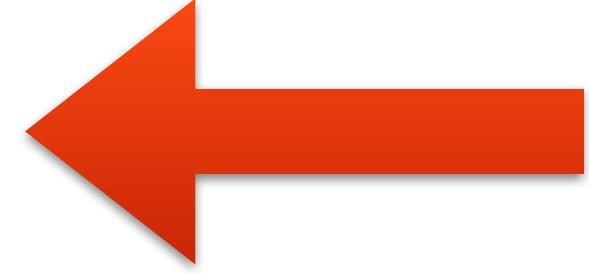
CUnit

- På denna kurs använder vi CUnit som enhetstestramverk för C-kod (och JUnit för Java-kod senare)
- CUnit har en mängd hjälpfunktioner för att skriva tester som ofta underlättar testning

Skapa grupper av tester, slå av och på olika tester, etc.

- Framförallt har CUnit en hel uppsättning av olika asserts som vi skall använda

C:s vanliga assert **kraschar programmet** så fort den slår ut — det hindrar alla tester från att köras vilket kan dölja fel



- Inkludera i testfilerna `#include <CUnit/Basic.h>`
- Tillägg sist till kompileringsflaggorna för länkning

`gcc ... -lcunit`

test_trim med CUnit

- Testfallen är ”unionen” av testfallen för våra ”find-funktioner”

- För varje testfall, kontrollera

Att returadressen alltid är samma som argumentet

Att resultatsträngen har förväntat innehåll

- Viktig förbättring

Bryt upp varje assert i två

(Här motiverad av begränsat utrymme)

```
void test_trim()
{
    char empty[] = "";
    char no_blanks[] = "12345";
    char only_blanks[] = "      ";
    char one_blank_first[] = " 1234";
    char three_blanks_first[] = "    12";
    char two_blanks_then_mix[] = " 1  ";
    char one_blank_last[] = "4321 ";
    char three_blanks_last[] = "21  ";

    char *test = trim(empty);
    CU_ASSERT_PTR_EQUAL(test, empty);
    CU_ASSERT_STRING_EQUAL(test, "");

    test = trim(no_blanks);
    CU_ASSERT_PTR_EQUAL(test, no_blanks);
    CU_ASSERT_STRING_EQUAL(test, "12345");

    test = trim(only_blanks);
    CU_ASSERT_PTR_EQUAL(test, only_blanks);
    CU_ASSERT_STRING_EQUAL(test, "");

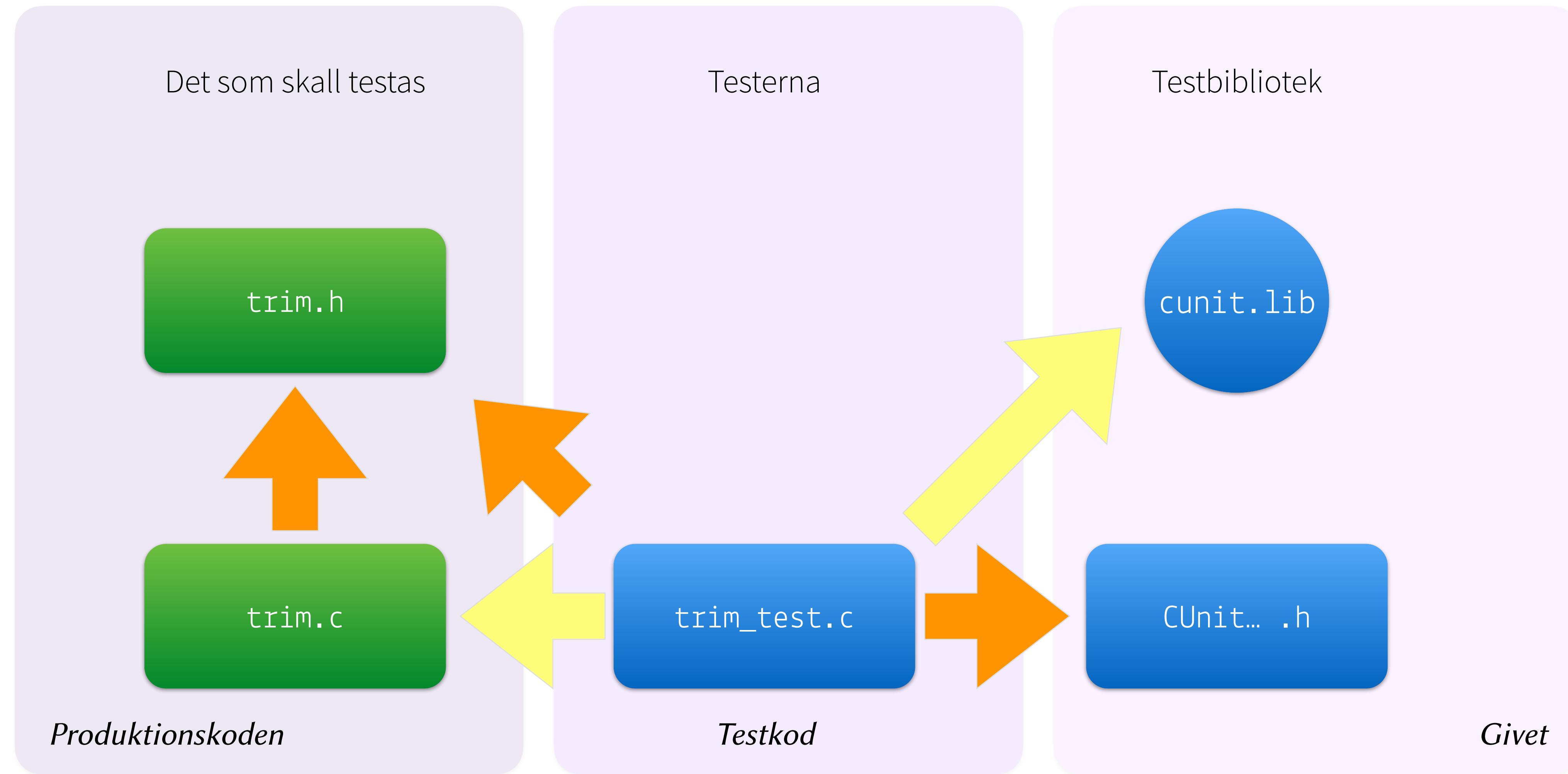
    test = trim(one_blank_first);
    CU_ASSERT_PTR_EQUAL(test, one_blank_first);
    CU_ASSERT_STRING_EQUAL(test, "1234");
```

Test av `find_first_non_blank(char *)`

```
void test_find_first_non_blank()
{
    char *empty = "";
    char *no_blanks = "12345";
    char *only_blanks = "      ";
    char *one_blank_first = " 1234";
    char *three_blanks_first = "    12";
    char *two_blanks_then_mix = "  1  ";

    CU_ASSERT_EQUAL(find_first_non_blank(empty)           - empty            , 0);
    CU_ASSERT_EQUAL(find_first_non_blank(no_blanks)        - no_blanks         , 0);
    CU_ASSERT_EQUAL(find_first_non_blank(only_blanks)      - only_blanks       , (long) strlen(only_blanks));
    CU_ASSERT_EQUAL(find_first_non_blank(one_blank_first) - one_blank_first , 1);
    CU_ASSERT_EQUAL(find_first_non_blank(three_blanks_first)- three_blanks_first, 3);
    CU_ASSERT_EQUAL(find_first_non_blank(two_blanks_then_mix)- two_blanks_then_mix, 2);
}
```

Enhetstestning under fas 1 (procedurell programmering i C)



Ett tests anatomি

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

Operationer varvat med ASSERTs

- Riv ned testet
- 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

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

    CU_ASSERT_PTR_NOT_NULL(s == NULL);
    CU_ASSERT_EQUAL(stack_height(s), 0);

    stack_free(s);
}
```

CUnits asserts (av 35)

| Assert | Finns Fatal? |
|--|--------------|
| CU_ASSERT_TRUE(value) | Ja |
| CU_ASSERT_FALSE(value) | Ja |
| CU_ASSERT_EQUAL(actual, expected) | Ja |
| CU_ASSERT_NOT_EQUAL(actual, expected) | Ja |
| CU_ASSERT_PTR_EQUAL(actual, expected) | Ja |
| CU_ASSERT_PTR_NULL(value) | Ja |
| CU_ASSERT_PTR_NOT_NULL(value) | Ja |
| CU_ASSERT_STRING_EQUAL(actual, expected) | Ja |
| CU_FAIL(message) | Ja |

Testdriven utveckling

En alternativ ansats till utveckling där vi startar med att skriva testkod innan vi skriver produktionskod

När man skriver tester måste man fundera på vad specifikationen säger och som en sidoeffekt får man en beskrivning i form av programkod som dessutom kan exekveras

Du har tydliga kriterier för när du är klar

Metoden i ett nötskal:

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

Skriv ned detta i termer av tester, innan du börjar implementera funktionen (kan också vara ett antal funktioner som arbetar tillsammans)

Börja implementera — målet är att få testerna att passera

Kan göras parallellt i ett programmeringspar: person A skriver tester, person B skriver produktionskoden

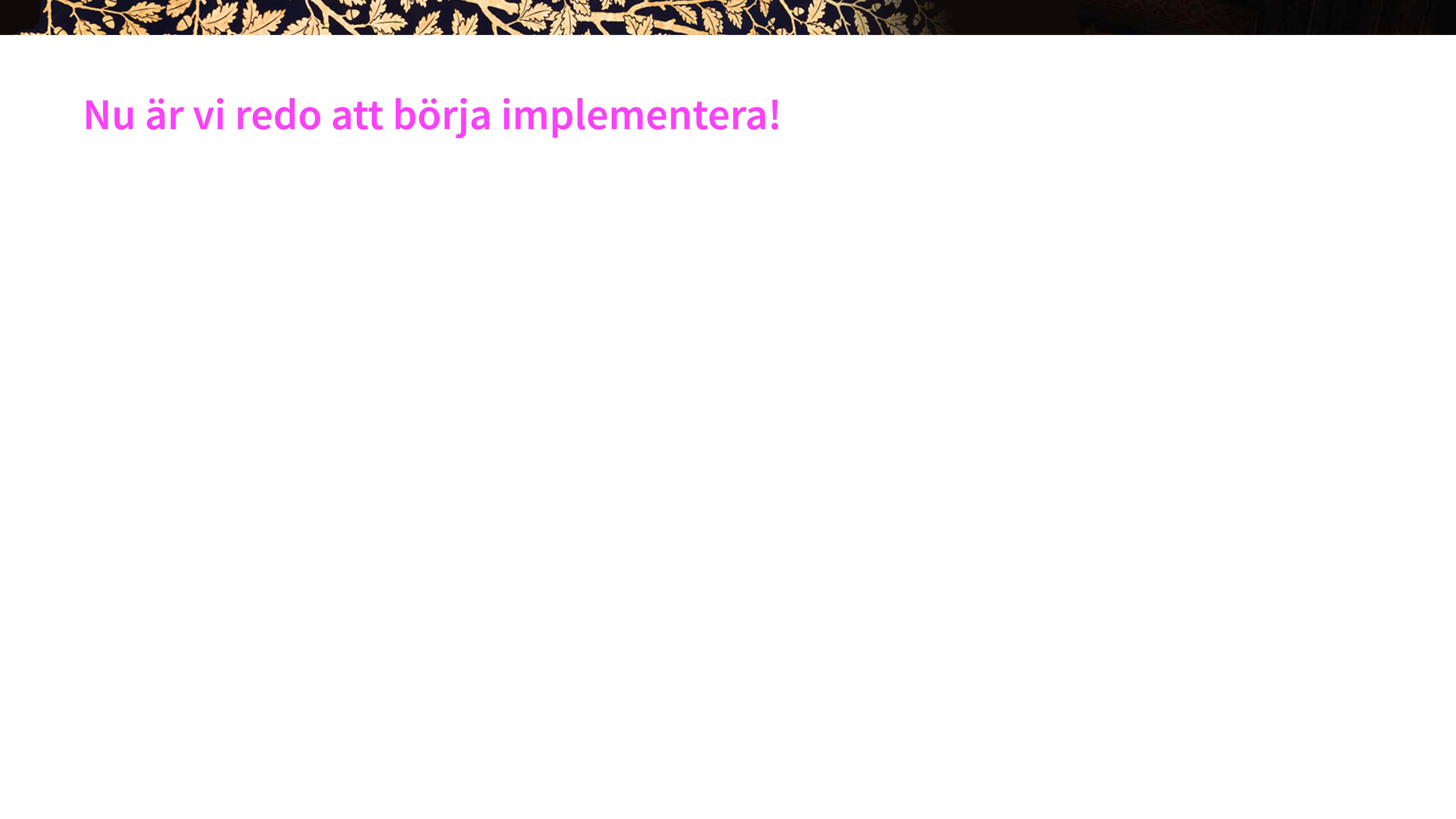
Ha alltid ett körande program – även om det ”bara” är tester

```
char *find_first_non_blank(char *s)
{
    // TODO
    return s;
}
```

```
bool is_blank(char c)
{
    // TODO
    return false;
}
```

```
char *find_last_non_blank(char *s)
{
    // TODO
    return s;
}
```

```
char *trim(char *string)
{
    // TODO
    return string;
}
```



Nu är vi redo att börja implementera!

Nu är vi redo att börja implementera!

```
bool is_blank(char c)
{
    return strchr(" \t\n\r\f", c) != NULL;
}
```

Nu är vi redo att börja implementera!

```
char *find_first_non_blank(char *s)
{
    while (*s && is_blank(*s))
    {
        ++s;
    }
    return s;
}
```

```
bool is_blank(char c)
{
    return strchr(" \t\n\r\f", c) != NULL;
}
```

Nu är vi redo att börja implementera!

```
char *find_first_non_blank(char *s)
{
    while (*s && is_blank(*s))
    {
        ++s;
    }
    return s;
}
```

```
bool is_blank(char c)
{
    return strchr(" \t\n\r\f", c) != NULL;
}
```

```
char *find_last_non_blank(char *s)
{
    s += strlen(s) - 1;

    while (*s && is_blank(*s))
    {
        --s;
    }
    return s;
}
```

Nu är vi redo att börja implementera!

```
char *find_first_non_blank(char *s)
{
    while (*s && is_blank(*s))
    {
        ++s;
    }
    return s;
}
```

```
char *find_last_non_blank(char *s)
{
    s += strlen(s) - 1;

    while (*s && is_blank(*s))
    {
        --s;
    }
    return s;
}
```

```
bool is_blank(char c)
{
    return strchr(" \t\n\r\f", c) != NULL;
}
```

```
char *trim(char *string)
{
    char *begin = find_first_non_blank(string);
    char *end = find_last_non_blank(string);

    char *to = string;

    while (begin <= end)
    {
        *to++ = *begin++;
    }
    *to = '\0';

    return string;
}
```

(Så här ser det ut när) Alla tester passerar

```
stw:/tmp/ $ ./a.out

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

Suite: Trim tests
Test: test trim ...passed
Test: find first non-blank ...passed
Test: find last non-blank ...passed
Test: char is blank ...passed

Run Summary:    Type  Total    Ran  Passed  Failed  Inactive
               suites      1       1     n/a       0       0
                  tests      4       4       4       0       0
                 asserts     56      56      56       0     n/a

Elapsed time =    0.000 seconds
stw:/tmp/ $ █
```



(Så här ser det ut när) Alla tester inte passerar

```
stw:/tmp/ $ ./a.out
```

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

```
Suite: Trim tests
```

```
Test: test trim ...FAILED
```

1. test.c:152 - CU_ASSERT_STRING_EQUAL(test,"1")
2. test.c:156 - CU_ASSERT_STRING_EQUAL(test,"4321")
3. test.c:160 - CU_ASSERT_STRING_EQUAL(test,"21")

```
Test: find first non-blank ...passed
```

```
Test: find last non-blank ...FAILED
```

1. test.c:110 - CU_ASSERT_EQUAL(find_last_non_blank(no_blanks) - no_blanks,(long) strlen(no_blanks) - 1)
2. test.c:112 - CU_ASSERT_EQUAL(find_last_non_blank(one_blank_last) - one_blank_last,3)
3. test.c:113 - CU_ASSERT_EQUAL(find_last_non_blank(three_blanks_last) - three_blanks_last,1)
4. test.c:114 - CU_ASSERT_EQUAL(find_last_non_blank(two_blanks_then_mix) - two_blanks_then_mix,2)

```
Test: char is blank ...passed
```

| Run Summary: | Type | Total | Ran | Passed | Failed | Inactive |
|--------------|---------|-------|-----|--------|--------|----------|
| | suites | 1 | 1 | n/a | 0 | 0 |
| | tests | 4 | 4 | 2 | 2 | 0 |
| | asserts | 56 | 56 | 49 | 7 | n/a |

```
char *find_last_non_blank(char *s)
{
    s += strlen(s) - 1;

    while (*s && is_blank(*s))
    {
        --s;
    }
    return s + 1;
}
```



MVP Dashboard Interface

```
CUnit - A Unit testing framework for C - Version 2.1-3
http://cunit.sourceforge.net/

Progress [██████████] 100% (4 tests run)

Tests Run :      4   Success :      2   Failed :      2

===== Suite Run Summary =====
TOTAL SUITES:    1
    Run:        1
    Skipped:    0
    Inactive:   0

===== Test Run Summary =====
TOTAL TESTS:    4
    Run:        4
    Skipped:    0
    Successful: 2
    Failed:     2
    Inactive:   0

===== Assertion Summary =====
TOTAL ASSERTS:  56
    Passed:    49
    Failed:     7

===== Failure Summary =====
TOTAL FAILURES: 7

(R)un (S)elected (L)ist (A)ctivate (F)ailures (O)ptions (H)elp (Q)uit
```

CU_curses_run_tests();

Vad är mjukvarutestning?

Ett sätt att förstå och dokumentera ”krav” och förväntat beteende hos ett system

Programkod beskriver saker entydigt

Programkod kan exekveras för att testa om beteendet är det förväntade

Ett sätt att hitta defekter (kollokvialt: buggar) i kod under utveckling

Defekter är i det generella fallet oundvikliga vid utveckling

Många kockar redar samma source

Krav eller omständigheter förändras över tid

Tester minskar riskerna med refaktorering och explorativ programmering

*Testning kan enbart
påvisa förekomsten av fel
– inte avsaknaden*

*Ett test som inte kan
automatiseras är ett
dåligt (farligt) test!*

Explorativt försök till förbättring

```
char *trim(char *string)
{
    char *begin = find_first_non_blank(string);
    char *end = find_last_non_blank(string);
    char *to = string;

    while (begin <= end)
    {
        *to++ = *begin++;
    }
    *to = '\0';

    return string;
}
```



```
char *trim(char *string)
{
    char *begin = find_first_non_blank(string);
    char *end = find_last_non_blank(string);
    char *to = string;

    memmove(to, begin, end - begin);
    to[end - begin] = '\0';

    return string;
}
```

Explorativt försök till förbättring

”Onödigt” dyrt — går generellt förbi målet och ”backar” sedan

```
char *find_last_non_blank(char *s)
{
    s += strlen(s) - 1;

    while (*s && is_blank(*s))
    {
        --s;
    }

    return s;
}
```



```
char *find_last_non_blank(char *s)
{
    char *last_non_blank = s;

    while (*s)
    {
        if (is_blank(*s) == false) last_non_blank = s;
        ++s;
    }

    return last_non_blank;
}
```

Ytterligare förbättringar är möjliga...

Regressionstestning

- En (stor) uppsättning enhetstester för olika delar av programmet
- Möjlighet att köra dem automatiskt [tas upp på föreläsning om automation]

- Vid varje förändrings:

Kör alla enhetstester automatiskt

Notera vilka som passerar och vilka som inte passerar

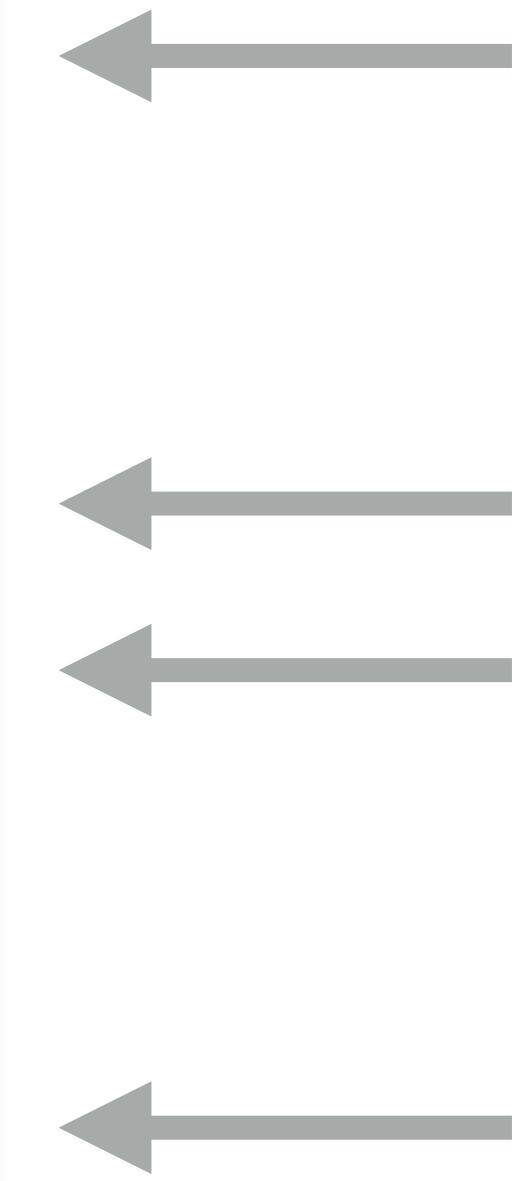
Implementera förändringen

Kör alla enhetstester automatiskt igen

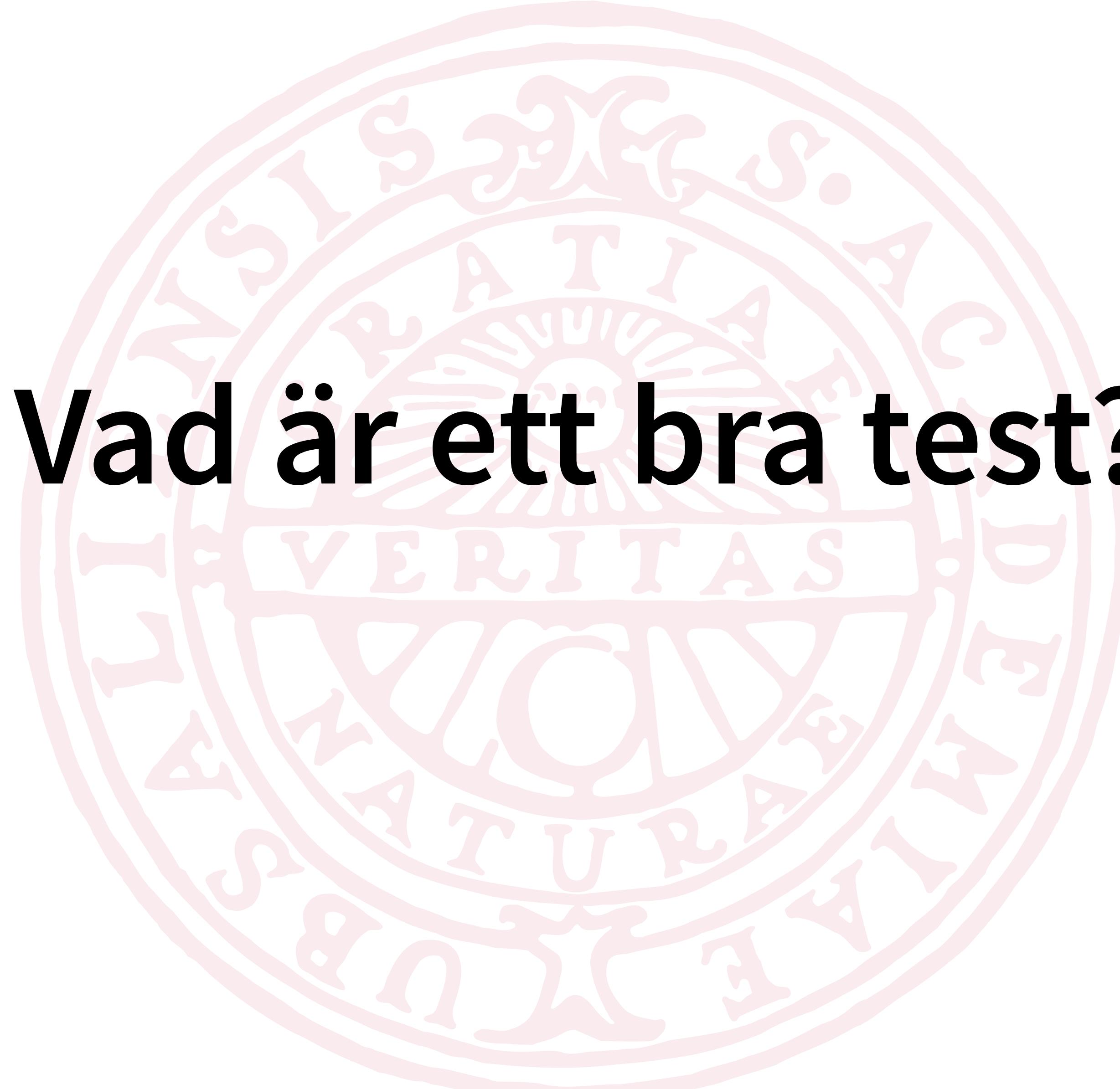
Stäm av mot föregående lista — verkar det rimligt?

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% |



Adapted from Programming Productivity (Jones 1986), Software Defect-Removal Efficiency (Jones 19969, What We Have Learned About Fighting Defects (Shull et al. 2002)



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

Inte så värdefullt att jämföra X vs Y% testtäckning – se det som en **grov** skattning av testkvalitet och främst ett verktyg för att hitta vilken kod som ännu inte testas!

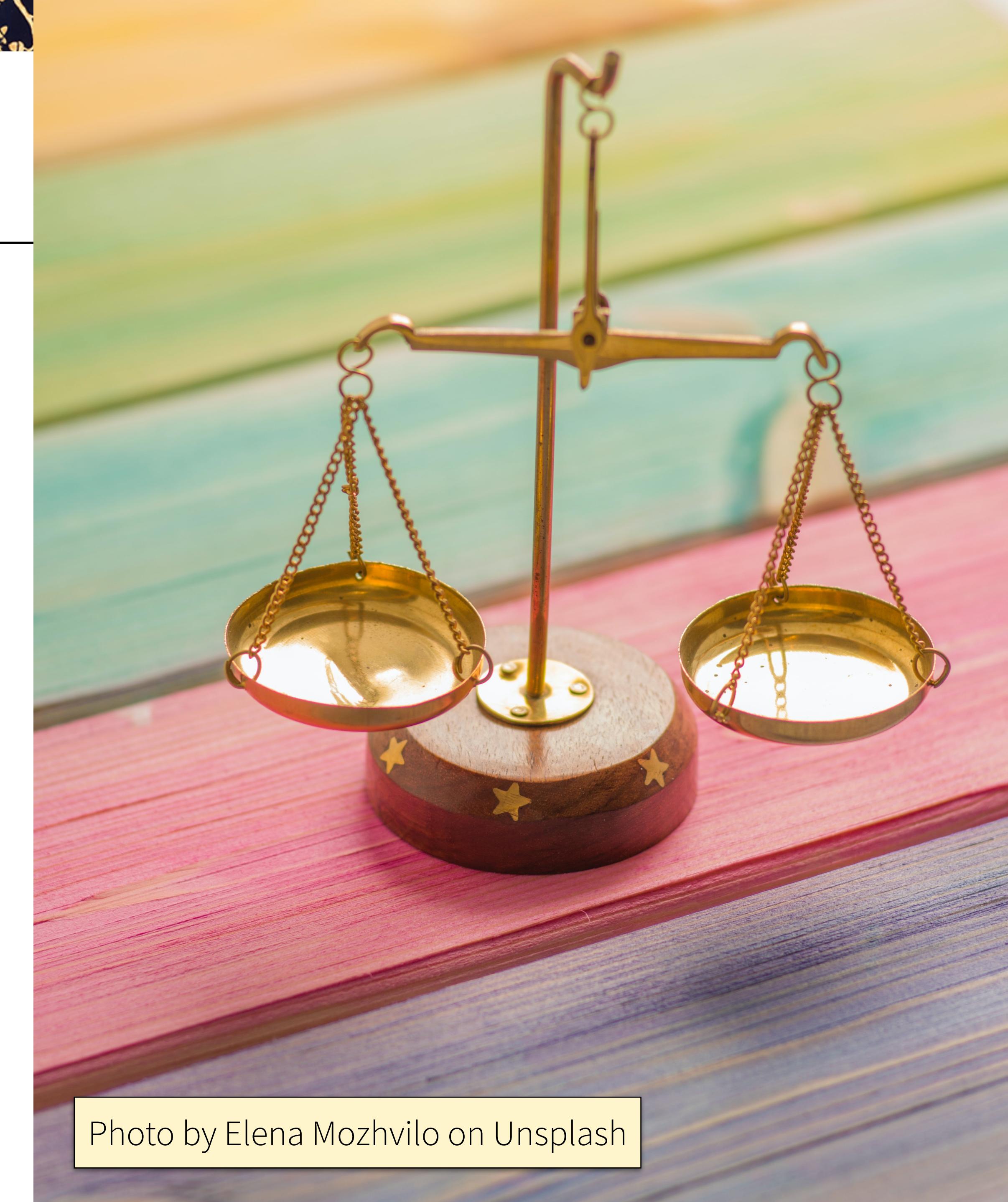


Photo by Elena Mozhvilo on Unsplash



Några exempel på hur man kan tänka på testtäckning

Strukturell testtäckning – utgångspunkten är alla möjliga vägar genom koden / kodens struktur

Exekveras alla satsar i alla möjliga ordningar?

Exekveras alla villkor i alla möjliga ordningar?

Exekveras alla loopar i alla möjliga repetitioner?

Testas alla möjliga kombinationer av booleska villkor?

Funktionsorienterad testtäckning – utgångspunkten är alla funktioner (i betydelsen funktionalitet) i det som skall testas

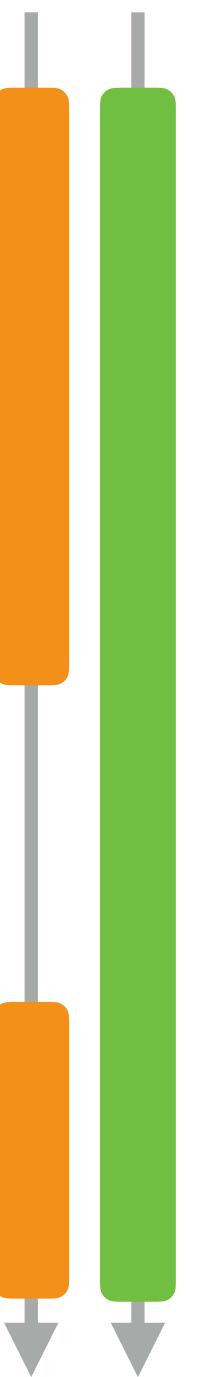
Testar vi alla funktioner med alla möjliga ”indata”?

Beteende-orienterad testtäckning – utgångspunkten är tillståndsövergångarna i systemet under test

Testar vi alla möjliga tillstånd och alla möjliga övergångar mellan tillstånd?

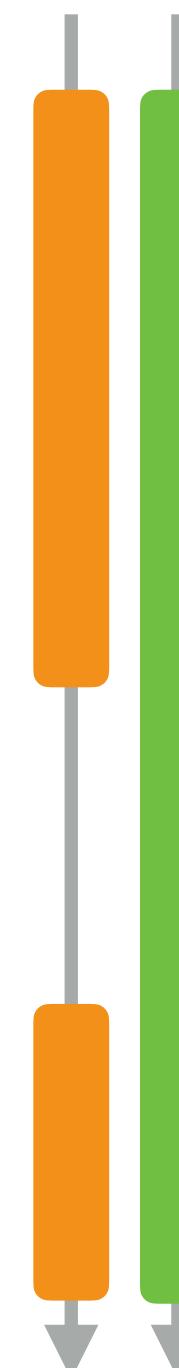
Strukturell testtäckning

- Testfallet "" testar alla  rader
- Testfallet "12345" testar alla  rader
- Men hur är det med koden inuti `find...blank`?



Strukturell testtäckning

- Testfallet "" testar alla  rader
- Testfallet "12345" testar alla  rader
- Men hur är det med koden inuti `find_..._blank`?



```
char *trim(char *string)
{
    char *begin = find_first_non_blank(string);
    char *end = find_last_non_blank(string);

    char *to = string;

    while (begin <= end)
    {
        *to++ = *begin++;
    }
    *to = '\0';

    return string;
}
```

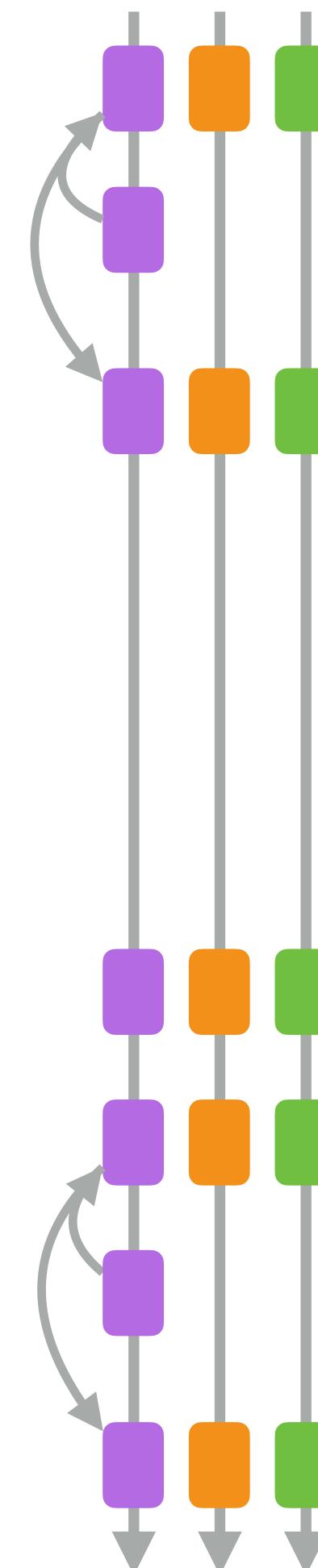
Strukturell testläckning

- Testfallet "" testar alla  rader
- Testfallet "12345" testar alla  rader
- Men hur är det med koden inuti `find...blank`?

Testar aldrig loop-kropparna och vi går aldrig tillbaka och
testar loop-villkoren mer än en gång (innan vi går in i
loopen)

Testfallet " 1 2 " testar alla  rader samt alla  rader
på föregående bild

Vi behöver normalt verktyg för att undersöka detta
eftersom alla vägar växer så fort



Strukturell testläckning

- Testfallet "" testar alla  rader
- Testfallet "12345" testar alla  rader
- Men hur är det med koden inuti `find_..._blank`?

Testar aldrig loop-kropparna och vi går aldrig tillbaka och testar loop-villkoren mer än en gång (innan vi går in i loopen)

Testfallet " 1 2 " testar alla  rader samt alla  rader på föregående bild

Vi behöver normalt verktyg för att undersöka detta eftersom alla vägar växer så fort



```
char *find_first_non_blank(char *s)
{
    while (*s && is_blank(*s))
    {
        ++s;
    }
    return s;
}
```

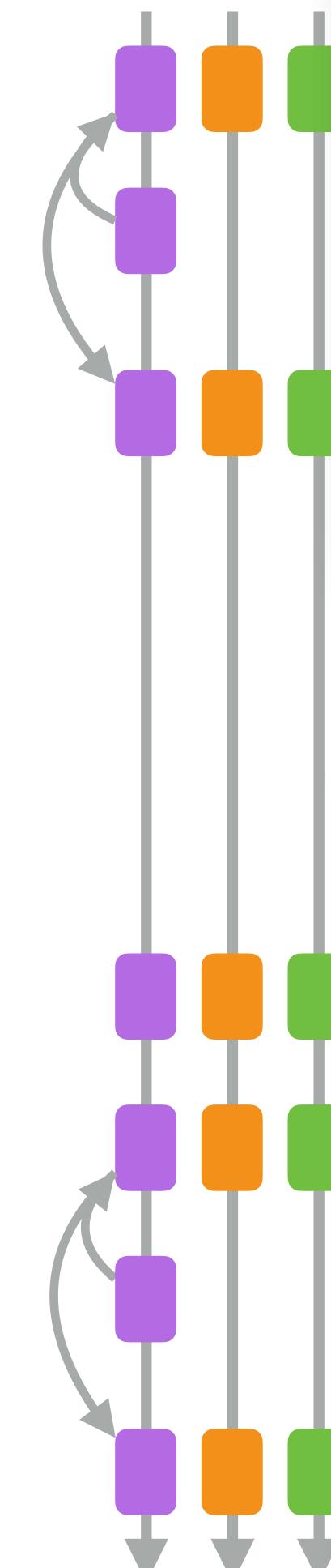
Strukturell testläckning

- Testfallet "" testar alla  rader
- Testfallet "12345" testar alla  rader
- Men hur är det med koden inuti `find_..._blank`?

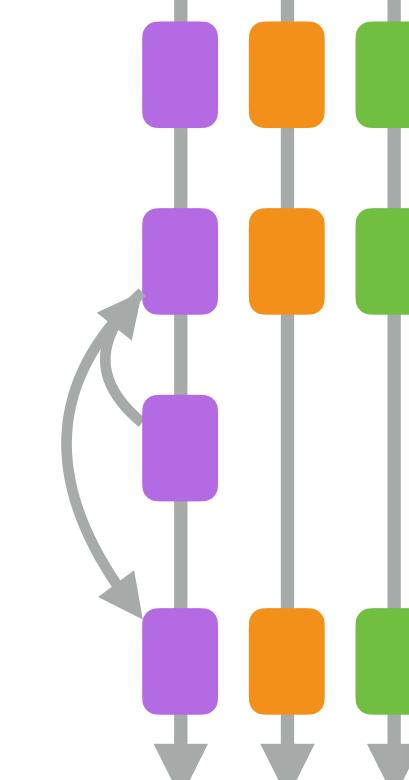
Testar aldrig loop-kropparna och vi går aldrig tillbaka och testar loop-villkoren mer än en gång (innan vi går in i loopen)

Testfallet " 1 2 " testar alla  rader samt alla  rader på föregående bild

Vi behöver normalt verktyg för att undersöka detta eftersom alla vägar växer så fort



```
char *find_first_non_blank(char *s)
{
    while (*s && is_blank(*s))
    {
        ++s;
    }
    return s;
}
```

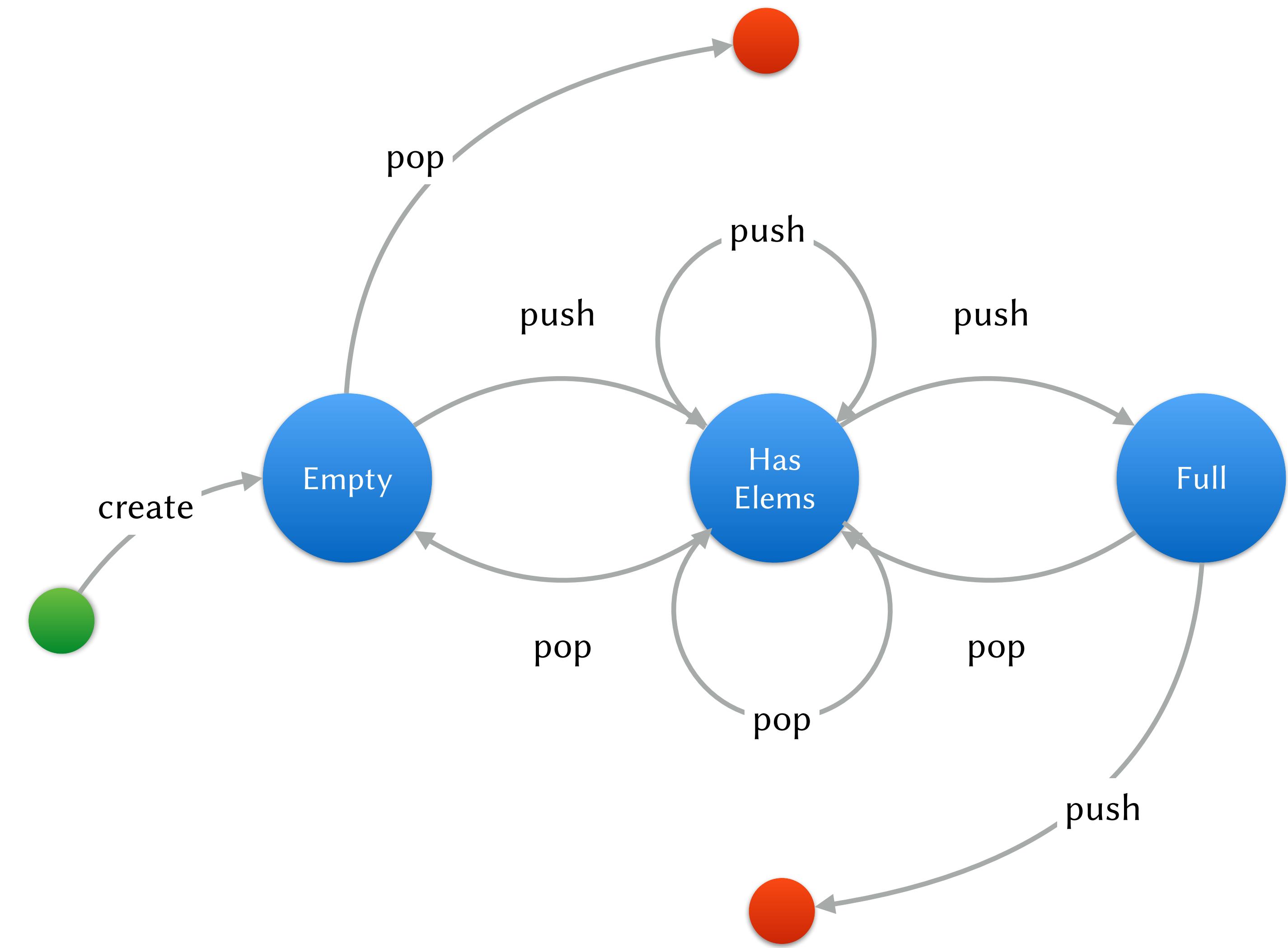


```
char *find_last_non_blank(char *s)
{
    s += strlen(s) - 1;

    while (*s && is_blank(*s))
    {
        --s;
    }
    return s;
}
```

Beteende-orienterad teststäckning

- Vår utgångspunkt är en modell för systemets under test tillstånd
- Exempel: en stack-datastruktur
 - Möjliga valida tillstånd: tom, full, eller mellan tom och full ("har element")
 - Möjliga invalida tillstånd:
"underflow" (dvs. ta bort i tom stack) och
"overflow" (dvs. lägg till en full stack)
 - Möjliga tillståndsövergångar: push (lägg in element), pop (ta bort element)
- Skriv testkod som prövar alla dessa



MVP testtäckning med gcc, gcov och lcov

```
$ gcov -abcfu trim_test.c

Function 'main'
Lines executed:80.00% of 25

Function 'clean_suite'
Lines executed:100.00% of 2

Function 'init_suite'
Lines executed:100.00% of 2

Function 'test_trim'
Lines executed:100.00% of 34

Function 'test_find_last_non_blank'
Lines executed:100.00% of 14

Function 'test_find_first_non_blank'
Lines executed:100.00% of 14

Function 'test_is_blank'
Lines executed:100.00% of 12

Function 'trim'
Lines executed:100.00% of 8

Function 'find_last_non_blank2'
Lines executed:0.00% of 5

Function 'find_last_non_blank'
Lines executed:100.00% of 6

Function 'find_first_non_blank'
Lines executed:100.00% of 4

Function 'is_blank'
Lines executed:100.00% of 2

File 'trim_test.c'
Lines executed:92.19% of 128
Branches executed:87.50% of 32
Taken at least once:68.75% of 32
Calls executed:93.26% of 89

Creating 'trim_test.c.gcov'
```

a = all blocks

b = branch probabilities

c = branch counts

f = function summaries

u = unconditional branches

MVP testtäckning med gcc, gcov och lcov

Kompilera programmet

```
gcc --coverage -l/usr/include/CUnit/ trim_tests.c -lcunit
```

Kör programmet som vanligt

```
./a.out
```

Kör lcov för att processera datat

```
lcov --capture --directory . --output-file coverage.info
```

Generera en HTML-vy

```
genhtml coverage.info --output-directory out
```

Öppna HTML-filen

```
firefox out/index.html
```

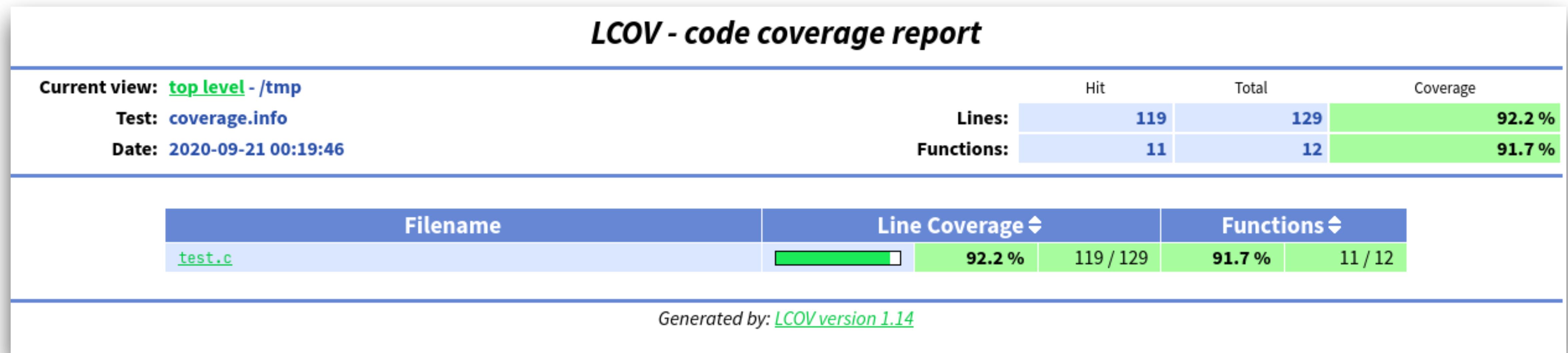
```
18 :  
19 :     98 : char *find_first_non_blank(char *s)  
20 :     : {  
21 :     217 :     while (*s && is_blank(*s))  
22 :     :     {  
23 :     119 :         ++s;  
24 :     :     }  
25 :     98 :     return s;  
26 :     : }  
27 :     :  
28 :     98 : char *find_last_non_blank(char *s)  
29 :     : {  
30 :     98 :     char *last_non_blank = s;  
31 :     :  
32 :     483 :     while (*s)  
33 :     :     {  
34 :     385 :         if (is_blank(*s) == false) last_non_blank = s + 1;  
35 :     385 :         ++s;  
36 :     :     }  
37 :     :  
38 :     98 :     return last_non_blank;  
39 :     : }  
40 :     :  
41 :     0 : char *find_last_non_blank2(char *s)  
42 :     : {  
43 :     0 :     s += strlen(s) - 1;  
44 :     :  
45 :     0 :     while (*s && is_blank(*s))  
46 :     :     {  
47 :     0 :         --s;  
48 :     :     }  
49 :     0 :     return s;  
50 :     : }  
51 :     :  
52 :     56 : char *trim(char *string)  
53 :     : {  
54 :     56 :     char *begin = find_first_non_blank(string);  
55 :     56 :     char *end = find_last_non_blank(string);  
56 :     56 :     char *to = string;  
57 :     :  
58 :     238 :     while (begin <= end)  
59 :     :     {  
60 :     182 :         *to++ = *begin++;  
61 :     :     }  
62 :     :  
63 :     56 :     *to = '\0';
```

Exekverades

Död kod!

Exekverades

MVP testtäckning med gcc, gcov och lcov



Viktiga tankar om testning

- Testning kan enbart påvisa förekomsten av fel — inte avsaknaden

Om du är intresserad av bevisbart korrekt programvara finns kurser för dig senare i utbildningen

- Ett test som inte kan automatiseras är ett dåligt (farligt) 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

Oerhört viktigt att ha den inställningen — annars drabbas man lätt av "confirmation bias"

Vad händer t.ex. i ett programmeringspar när en skriver tester och en annan skriver koden som skall testas?



Dijkstra / Bild: Wikipedia



Väl mött på labben!

Backup: Stack-exemplet

```

#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;
}

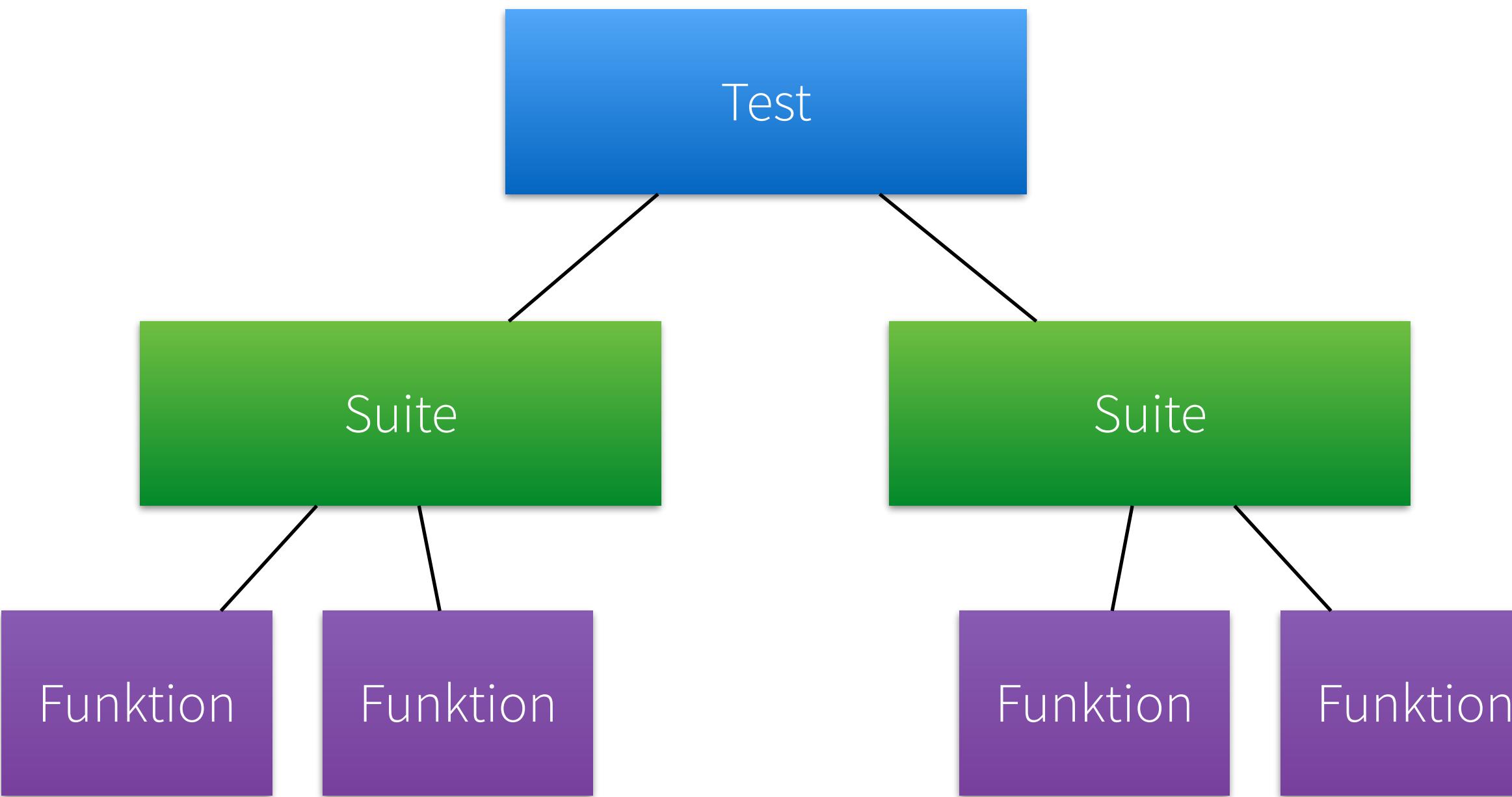
void ioopm_stack_destroy(stack_t *s)
{
    while (s->top)
    {
        ioopm_stack_pop(s);
    }
    free(s);
}

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

```

stack.c

Ett testprogram i CUnit



```
#include <string.h>
#include <stdbool.h>
#include <CUUnit/Basic.h>

int init_suite(void)
{
    return 0;
}

int clean_suite(void)
{
    return 0;
}

void test1(void)
{
    CU_ASSERT(true);
}

void test2(void)
{
    CU_ASSERT(true);
}

int main()
{
    CU_pSuite test_suite1 = NULL;

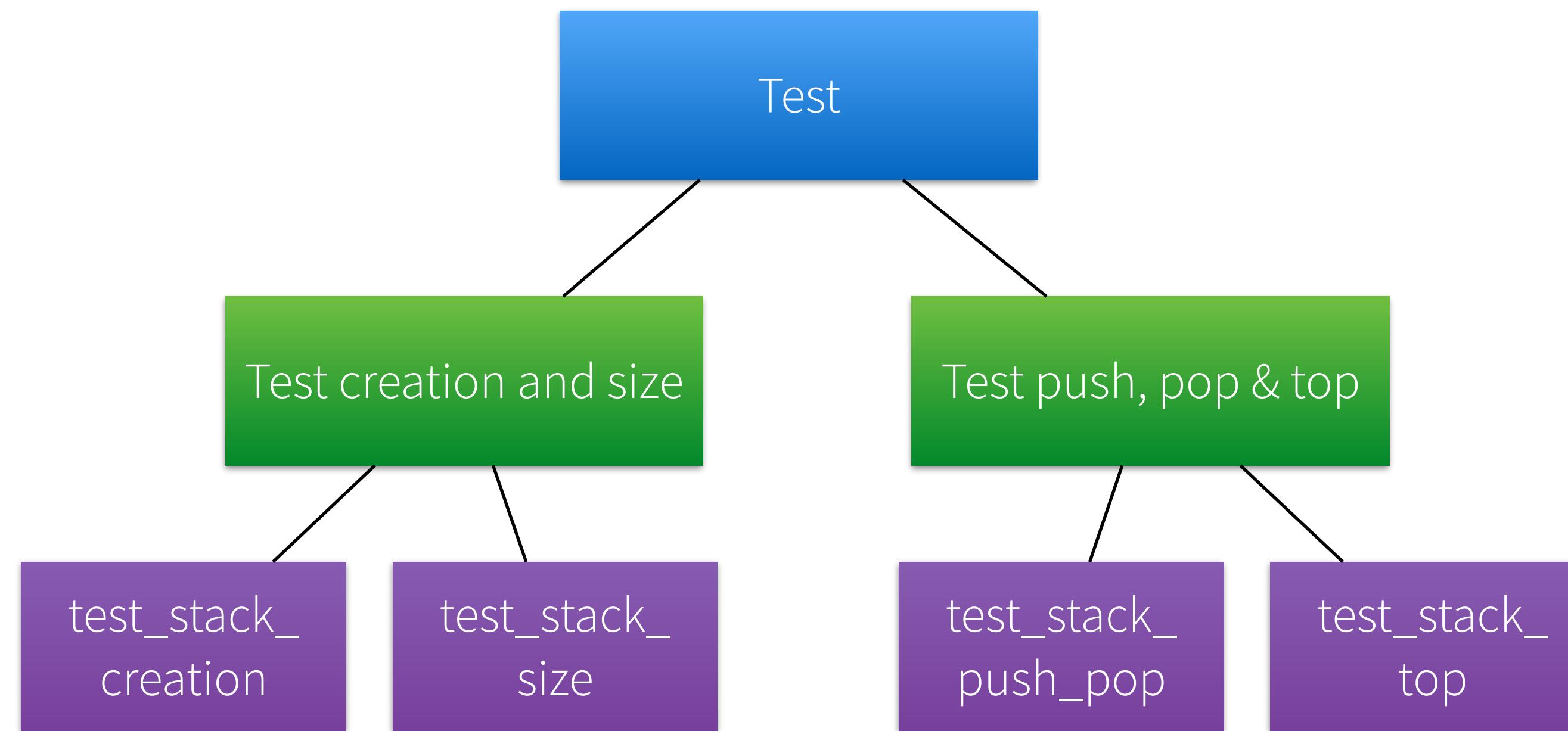
    if (CUE_SUCCESS != CU_initialize_registry())
        return CU_get_error();

    test_suite1 = CU_add_suite("Test Suite 1", init_suite, clean_suite);
    if (NULL == test_suite1)
    {
        CU_cleanup_registry();
        return CU_get_error();
    }

    if (
        (NULL == CU_add_test(test_suite1, "test 1", test1)) ||
        (NULL == CU_add_test(test_suite1, "test 2", test2))
    )
    {
        CU_cleanup_registry();
        return CU_get_error();
    }

    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    CU_cleanup_registry();
    return CU_get_error();
}
```

stack_test.c



main()

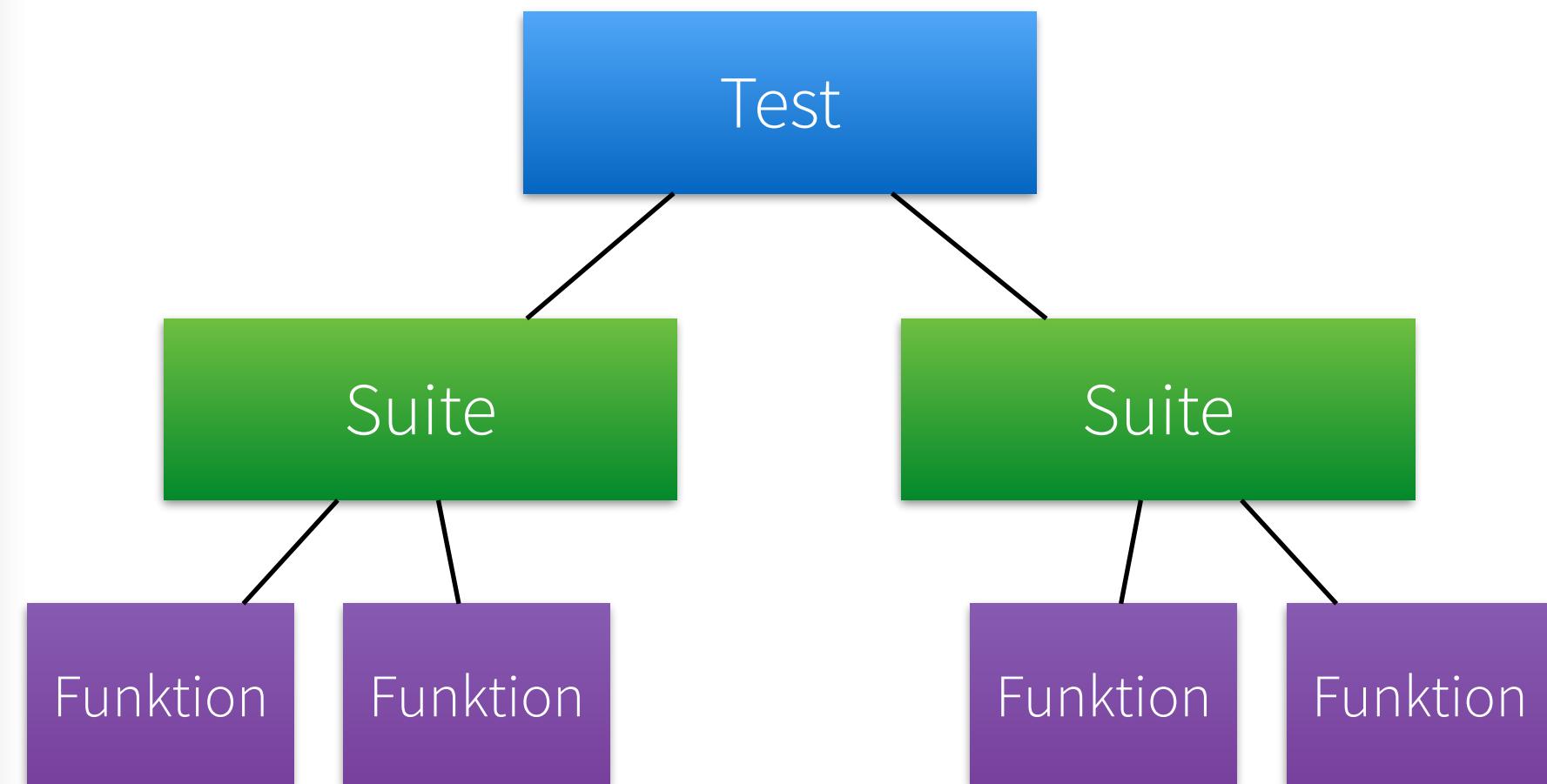
```
int main(int argc, char *argv[])
{
    // Initialise
    CU_initialize_registry();

    // 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 pushpoptop = CU_add_suite("Test push, pop and top", NULL, NULL);
    CU_add_test(pushpoptop, "Push and pop", test_stack_push_pop);
    CU_add_test(pushpoptop, "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();
    int values = 10;

    for (int i = values; i > 0; --i)
    {
        ioopm_stack_push(s, i);
    }

    for (int i = 0; i < values; ++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);
}
```