

Defensiv programmering

Tobias Wrigstad
tobias.wrigstad@it.uu.se



Observationer kring ”vanlig programmering”

- Programmerare tenderar att fokusera på de problem som måste lösas för att programmet skall ”fungera”
- Programmerare gör vanligen antaganden kring t.ex.

Hur en funktion kommer att anropas (t.ex. med korrekt indata)

Hur miljön som programmet kör i beteeri sig (t.ex. ingen tar bort katalogen jag står i under körning)

Användaren är vänligt inställd (och matar in korrekta data)

- Nya programmerare kan ofta glömma t.ex.

Att program förändras och muterar över tid

Att en rad kod läses oftare än den modifieras



Defensiv programmering

- En "princip" för att skapa feltolerant kod, introducerades av Kernighan och Ritchie
 - 1.) Gör aldrig några antaganden
 - 2.) Klä skott för misstag både inifrån och utifrån (även din kod förändras)
 - 3.) Tillämpa standarder
 - 4.) "Keep it simple"
- Termen kommer av defensive driving – man vet inte vad andra kommer att göra, så försök att köra så att du är trygg oavsett vad de gör

Handlar i grund och botten om att också ta ansvar även för "andras" fel

Mjukvaran skall fungera korrekt även med trasig indata

- Balansakt: felkontroller gör kod komplicerad (och kostar klockcykler)



”Skit in–skit ut!”

- En dålig princip!

- Istället:

Skit in – inget ut

Skit in – felmeddelande ut

Skit in är inte möjligt



Förhållandet till indata

- Indata till en funktion är en stor felkälla

Okontrollerad och oförutsägbar – kan t.o.m. ha ont uppsåt eller vara av ett slag som programmeraren inte tänkt på

- Defensiv programmering menar att vi skall "anta det värsta om all indata"

Fångar fel innan de leder till problem

Förenklar debuggning



Validering av indata

- För indata till en funktion

Definiera vad som är giltiga värden för alla parametrar

Validera allt indata mot denna definition

Bestäm ett beteende för funktionen om valideringen misslyckas



Exempel på validering

- Vanliga

- Är pekare NULL?

- Är index eller storlekar positiva?

- Division med noll

- Indexering inom storleksgränserna?

- Omöjliga värden

- Negativ skostorlek?

- Pre/postvillkor – använd som valideringsvillkor

- Antaganden bör dokumenteras med **assertions** (t.ex. bufferten är aldrig NULL)



Assertions

- En assertion är en konstruktion i ett program som tillåter programmet att kontrollera sig självt under körning

Assertions innehåller villkor som evalueras till sant eller falskt

Falskt: vi har upptäkt något som inte borde ha hänt i programmet

- Bra i små program, ovärdeliga i stora program eller program med höga krav på tillförlitlighet
- En assertion har normalt 1–2 komponenter

Ett villkor som förväntas hålla under körning

Ett (frivilligt) felmeddelande



Hur en assert kan byggas enkelt i C

- En enkel assert

```
#define assert(exp,msg) if (!exp) \
    fprintf(stderr, "Assert %s failed: %s\n", #exp, msg);
```

assert(1 < 0, "Dum assert");

ger

"Assert 1 < 0 failed: Dum assert."

(läs mer om makron i C för att förstå `#define`, etc.)



Hur en assert kan byggas enkelt i C

- En något mer raffinerad assert

```
#define assert(exp,msg) if (!exp) \  
    fprintf(stderr,"Assert failed (%s): %s:%d: %s: %s\n", \  
            #exp, \  
            __FILE__, \  
            __LINE__, \  
            __func__, \  
            msg);
```

assert(1 < 0, "Dum assert");

ger

"Assert failed (1 < 0): myprog.c:27: main: Dum assert."

(läs mer om makron i C för att förstå #define, etc.)



Olika program kräver olika felhantering

- Robusthet = undertryck fel

Program som strömmar realtidsvideo (bör hellre tappa frames än ackumulera "lagg")

Datorspel (ingen märker om ett event "försvinner")

- Korrekthet = undertryck aldrig fel

Datorspel (vars virtuella föremål är värdta faktiska pengar)

En magnetröntgen (bör inte skapa påhittade cancerdiagnoser)

- Ett viktigt beslut i högnivådesign — hur skall vi hantera fel?



Defensiv programmering

- Vad kan gå fel i detta program?

```
int average(int length, int values[]) {  
    int sum = 0;  
    for (int i=0; i < length; ++i) {  
        sum += values[i];  
    }  
    return sum / length;  
}
```



Defensiv programmering

- Vad kan gå fel i detta program?

- $\text{length} >$ den faktiska längden på values-arrayen
- $\text{length} \leq 0$
- $\text{values} == \text{NULL}$

```
int average(int length, int values[]) {  
    int sum = 0;  
    for (int i=0; i < length; ++i) {  
        sum += values[i];  
    }  
    return sum / length;  
}
```



Defensiv programmering

- Vad kan gå fel i detta program?

- $\text{length} >$ den faktiska längden på `values`-arrayen
- $\text{length} \leq 0$
- **values == NULL**

```
int average(int length, int values[]) {  
    assert(values);  
  
    int sum = 0;  
    for (int i=0; i < length; ++i) {  
        sum += values[i];  
    }  
    return sum / length;  
}
```

Kräver `#include <assert.h>`



Defensiv programmering

- Vad kan gå fel i detta program?

- **length >** den faktiska längden på values-arrayen
- **length <= 0**

- **values == NULL**

```
int average(int length, int values[]) {  
    assert(values);  
    assert(length > 0);  
  
    int sum = 0;  
    for (int i=0; i < length; ++i) {  
        sum += values[i];  
    }  
    return sum / length;  
}
```



Bra användning av assert!

Dokumenterar ett
viktigt villkor i average.

Defensiv programmering

- Vad kan gå fel i detta program?

- **length >** den faktiska längden på values-arrayen
- **length <= 0**

- **values == NULL**

```
int average(int length, int values[]) {  
    assert(values);  
    assert(length > 0);  
  
    int sum = 0;  
    for (int i=0; i < length; ++i) {  
        sum += values[i];  
    }  
    return sum / length;  
}
```

Borde verifieras vid
anropsplatsen!



Defensiv programmering

- Vad kan gå fel i detta program?

```
void myfunc(char *input) {  
    char *buffer = malloc(2048);  
    ...  
    strcpy(buffer, input);  
    ...  
}
```



Defensiv programmering

- Vad kan gå fel i detta program?

```
void myfunc(char *input) {  
    assert(input);  
  
    char *buffer = malloc(2048);  
  
    ...  
    strcpy(buffer, input);  
  
    ...  
}
```

Kräver `#include <assert.h>`



Defensiv programmering

- Vad kan gå fel i detta program?

```
void myfunc(char *input) {  
    assert(input);  
  
    char *buffer = malloc(2048);  
  
    ...  
    strncpy(buffer, input, 2048);  
    ...  
}
```

Använd **alltid** funktioner
med gränsvärden!



Defensiv programmering

- Vad kan gå fel i detta program?

```
void myfunc(char *input) {  
    assert(input);  
  
    char *buffer = malloc(2048);  
    // memset or calloc better  
    for (int i=0; i<2048; ++i)  
        buffer[i] = '\0';  
  
    . . .  
    strncpy(buffer, input, 2048);  
    . . .  
}
```



NAME

abort -- cause abnormal program termination

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdlib.h>
```

```
void  
abort(void);
```

DESCRIPTION

The **abort()** function causes abnormal program termination to occur, unless the signal SIGABRT is being caught and the signal handler does not return.

Any open streams are flushed and closed.

IMPLEMENTATION NOTES

The **abort()** function is thread-safe. It is unknown if it is async-cancel-safe.

The **abort()** function causes a report to be generated by Crash Reporter. If you wish to terminate without generating a crash report, use **exit(3)** instead.

RETURN VALUES

The **abort()** function never returns.

SEE ALSO

sigaction(2), **exit(3)**

STANDARDS

The **abort()** function conforms to ISO/IEC 9945-1:1990 ("POSIX.1"). The **abort()** function also conforms to ISO/IEC 9899:1999 ("ISO C99") with the implementation specific details as noted above.



Defensiv programmering

- Vad kan gå fel i detta program?

```
#define BUFSIZ 2048

void myfunc(char *input) {
    assert(input);

    char *buffer = malloc(BUFSIZ);
    // memset or calloc better
    for (int i=0; i< BUFSIZ; ++i)
        buffer[i] = '\0';

    ...
    strncpy(buffer, input, BUFSIZ);
    ...
}
```



Defensiv programmering

- Vad kan gå fel i detta program?

```
#define BUFSIZ 2048

void myfunc(char *input) {
    assert(input);

    char *buffer = malloc(BUFSIZ);
    // memset or calloc better
    for (int i=0; i< BUFSIZ; ++i) {
        buffer[i] = '\0';
    }

    ...
    strncpy(buffer, input, BUFSIZ);
    ...
}
```



Vad gör man när ett indata inte är valitt?

- Returnera ett "**neutralt** värde"

T.ex. 0, tomma strängen, NULL

För en punkt i planet utan x-värde, använd y-värdet (ta inte detta som en regel)

- Ta nästa data

Om man läser stock ticks för IBM kan man vänta till **nästa** IBM stock tick

Om man läser av tryck 10 gånger/sek, returnera **nästa** läsning

- Återanvänd ett gammalt data

Föregående tryckavläsning

Rita ut det som fanns där på skärmen **förra** framen



Vad gör man när ett indata inte är valitt?

- Ta ett angränsande valitt värde

Ersätt en negativ stränglängd med 0, en negativ kostnad med 0

- Logga varningar

Skriv en felrapport i en logg för att underlätta felsökning senare

Går att kombinera med alla föregående tekniker eller "bara kör på"

Om loggar behålls i produktionskod: fundera över om de exponerar data och kanske borde krypteras eller liknande.

- Returnera en felkod

OBS: Hanterar inte felet utan tvingar någon annan att ta hand om det!

T.ex. i form av funktionens returvärde eller en felflagga (errno i C)



Vad gör man när ett indata inte är valitt?

- Terminera programmet

"Crash don't trash"

Standardlösning i kritiska system

Problem: hur kan man backa ur på ett säkert sätt?



Tumregler för defensiv programmering

- Använd assertions för fel som **aldrig** borde uppkomma och annan felhantering för fel som kan tänkas uppkomma
- Stoppa aldrig kod med sidoeffekter i en assertion

Vad händer när assertions plockas bort i produktionskoden?

- Använd assertions för att dokumentera och verifiera pre- och postvillkor
- För verkligt robust kod, använd felhantering utöver assertions
- Och tillämpa offensiv programmering för att se programmets "defensiva beteende"



”Offensiv programmering”

- Se till att fel inte omedvetet undertrycks

- Exempel:

Ha ett default-fall i en switch-sats som säger "Oops! Vi har glömt ett fall här!" (och ev. avbryter exekveringen)

Gör så att asserts avbryter programmets exekvering

Använd allt minne för att testa programmets beteende när minnet är fullt

Förstör format på filer och strömmar för att se hur filhanteringen klarar det

Fyll ett objekt med skräpdata precis innan det frigörs

Inkludera mekanismer för att överföra kraschdata eller loggfiler automatiskt ifrån levererade system



Kapsla in/isolera fel

- För mycket felhantering är också en felkälla

Försök att isolera felkontroller, felhantering och konsekvenser (jmf. information hiding)

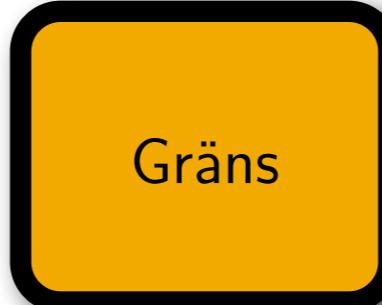
- Exempel

Felkontroller i alla publika funktioner, alla interna funktioner förutsätter att data är korrekt

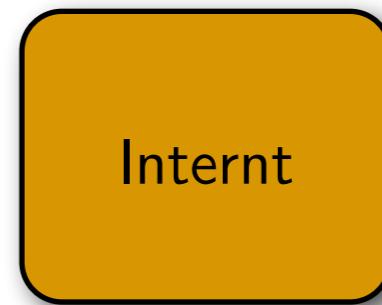
Ha flera kritiska ringar som kontrollerar olika typer av fel



Utgå från att
detta data är korrupt
eller opålitligt



Ansvarar för att
"säkra upp"
passerande data



Kan nu utgå från
att data är korrekt
och pålitligt



Produktionskod och utvecklingskod

- Utvecklingskoden kan ofta ta sig friheter som inte produktionkoden kan
 - Behöver inte gå lika fort
 - Behöver inte vara lika snål med resurser
 - etc.
- Detta ger ökad frihet att skriva utvecklingkod som underlättar debuggning och felsökning
 - T.ex. kod som kontrollerar datas integritet
 - Debug-läget in MS Word har en loop som kollar att dokumentet inte har blivit korrupt som kör flera gånger i sekunden



Vad som når produktionskoden

- Lämna kvar kontroller för viktiga fel
- Ta bort kontroller för triviala fel
- Ta bort kontroller som terminerar med hårdta kraschar vid invalitt data
- Lämna kvar kod som hjälper programmet terminera på ett förtjänstfullt sätt
- Logga fel för att underlätta felsökning
- Se till att alla felmeddelanden är trevliga

"You shoudn't have come here. The system has fucked up..."



Checklista för defensiv programmering

- Skyddar sig funktionen mot **dåliga indata**?
- Används **assertions** för att **dokumentera** omständigheter som aldrig borde uppstå, inklusive **pre- och postvillkor**?
- Används **assertions enbart** för att dokumentera omständigheter som **aldrig borde uppstå**?
- Används tekniker för att **minskar skadan från fel** och för att **minskar mängden kod** som måste "bry sig om" felhantering?
- Används **informationsgömningsprincipen** för att kapsla in interna förändringar?
- Har **hjälpfunktioner** implementerats i **utvecklingskoden** som hjälper till vid felsökning och debuggning?
- Är **mängden** defensiv programmering adekvat – varken för mycket eller för lite?
- Används **offensiva programmeringstekniker** för att minska risken att fel inte uppmärksamas under utveckling?

Adapterad från Steve McConnell's utmärkta "Code Complete"



Skydda dig mot dig själv!

Rädda en framtida du — redan idag



Skydda dig mot dig själv

```
if (foo())  
    bar;
```

```
while (bork())  
    f = f->next;
```



Skydda dig mot dig själv

```
#define square(n) n*n;  
square(3+4); // 19
```



Skydda dig mot dig själv

```
#define square(n) n*n;
```

```
square(3+4); // 19
```

```
#define square(n) (n*n);
```

```
square(3+4); // 49
```

```
int x = 4;
```

```
square(x++); // 20 and x == 6
```



Skydda dig mot dig själv

```
#define square(n) n*n;
```

```
square(3+4); // 19
```

```
#define square(n) (n*n);
```

```
square(3+4); // 49
```

```
int x = 4;  
square(x++); // 20 and x == 6
```

```
#define SQUARE(n) (n*n);
```

```
SQUARE(...);
```

Synligt att det
är ett makro!



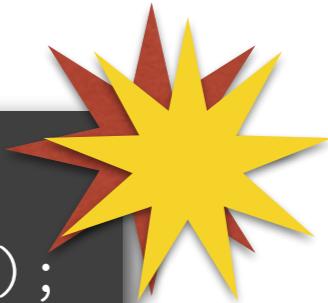
Skydda dig mot dig själv

```
#define DECLARE(varname,T,f1,v1,f2,v2) \  
    T varname = malloc(sizeof(T));           \  
    varname.f1 = v1;                         \  
    varname.f2 = v2;                         \  

```

```
DECLARE(new, Link, element, 42, next, NULL);  
list->last->next = new;
```

```
if (condition)  
    DECLARE(new, Link, element, 42, next, NULL);
```



Skydda dig mot dig själv

```
#define DECLARE(varname,T,f1,v1,f2,v2) \  
    T varname = malloc(sizeof(T));           \  
    varname.f1 = v1;                         \  
    varname.f2 = v2;                         \  

```

```
DECLARE(new, Link, element, 42, next, NULL);  
list->last->next = new;
```

```
if (condition)  
    DECLARE(new, Link, element, 42, next, NULL);
```

```
if (condition)  
    T varname ...  
    varname ...
```



Vettigt
användande av
block!

Skydda dig mot dig själv

```
#define DECLARE(varname,T,f1,v1,f2,v2) { \
    T varname = malloc(sizeof(T)); \
    varname.f1 = v1; \
    varname.f2 = v2; \
}
```

```
if (condition)
    DECLARE(new, Link, element, 42, next, NULL);
else
    foo;
```

```
if (condition)
{ ... };
else
    foo;
```



Skydda dig mot dig själv

```
#define DECLARE(varname,T,f1,v1,f2,v2) do { \
    T varname = malloc(sizeof(T)); \
    varname.f1 = v1; \
    varname.f2 = v2; \
} while (0);
```

Makrot ser ut som skam men
det är "gold standard"...



Namngiven initiering av struktar

```
struct person {  
    char *name;  
    uint8_t age;  
    struct person *spouse;  
    struct person *children;  
    uint8_t no_children;  
};
```

```
struct person p = { .name = "Peter", .age = 32 };
```



A color photograph of a man with a shaved head and a slight smile, wearing a dark olive-green jacket over a light-colored shirt. He is seated at a white table outdoors, leaning forward with his hands resting on the surface. In front of him are four blue cans of Pilsener Urquell beer. The background shows a green lawn and some trees under a clear sky.

ÖL? Plötsligt!

Namngiven initiering av struktar

```
struct person {  
    char *name;  
    uint8_t age;  
    struct person *spouse;  
    struct person *children;  
    uint8_t no_children;  
};
```

```
struct person p = { .name = "Peter", .age = 32 };
```

C bjuder på initiering av spouse,
children och no_children!



Namngiven initiering av struktar

```
struct person {  
    char *name;  
    uint8_t age;  
    struct person *spouse;  
    struct person *children;  
    uint8_t no_children;  
};
```

```
struct person p = { .name = "Peter", .age = 32 };
```

```
struct person p = { .age = 32, .name = "Peter" };
```

```
struct person p = { .name = "P", no_children = 0, .age = 32, };
```

```
struct person p = { };
```



”Defaultparametrar” och namngivna argument

```
bool register_person(struct person p) { ... }
```

```
register_person((struct person){.name = "Bob"});
```

```
#define PERSON(__VARGS__) ((struct person){__VARGS__})
```

```
bool register_person(struct person p) { ... }
```

```
register_person(PERSON(.name = "Bob"));
```



”Defaultparametrar” och namngivna argument

Med hjälp av ett makro kan vi skapa defaultvärdet
för en strukt! (Dock ej per funktion.)

```
#define PERSON(__VARGS__) \
    ((struct person){.name="Fred", __VARGS__})\

bool register_person(struct person p) { ... }

register_person(PERSON(.name = "Bob"));
```

”Skrivs över” av .name="Bob"
nedan, annars blir name "Fred".



Några ytterligare tumregler

Som inte har med defensiv programmering att göra
men som ändå passar in i sammanhanget



Några tumregler för att skriva bra kod

- Tydliggör beroenden mellan satser
- Ge namn för att tydliggöra beroenden och kopplingar
- Sista utväg: använd kommentarer för att lyfta fram beroenden som på inget annat sätt blir synliga i koden
- Koden bör vara läsbar utifrån och in
- Gruppera relaterade satser
- Faktorera ut orelaterade grupper till egna funktioner



Tumregler för namngivning

- Använd namn som tydligt beskriver vad en variabel representerar
- Använd namn från domänen i första hand, inte programrepresentationen
- Använd namn som är tillräckligt långa för att slippa "avkodning" (strpbrk)
- Använd loopindex med meningsfulla namn (alltså ej i, j, k) för loopar med fler än 1–2 rader
- Ersätt löpande namn på temp-variabler med meningsfulla namn
- Innebörden av booleska variabler (vid true/false) skall vara tydlig
- Döp konstanter för att fånga deras innebörd, inte deras värde



Tumregler för namngivning

- Var konsekvent
- Utveckla konventioner (och dokumentera dem)
- Skilj ut lokala / privata/ globala data
- Skilj ut konstanter / uppräkningsbara typer / variabler
- Välj en namnformattering efter läsbarhet
- Tag hänsyn till språkstandardarden i utformandet av namnkonventionen



Tumregler för variabelbenämning

- Undvik:

- Missledande eller tvetydiga namn

- Namn med liknande innebörd

- Namn som är identiska upp till 1–2 tecken

- Namn som innehåller siffror

- Medvetna felstavningar i syfte att förkorta namn

- Namn på ord som ofta stavas fel eller läses fel

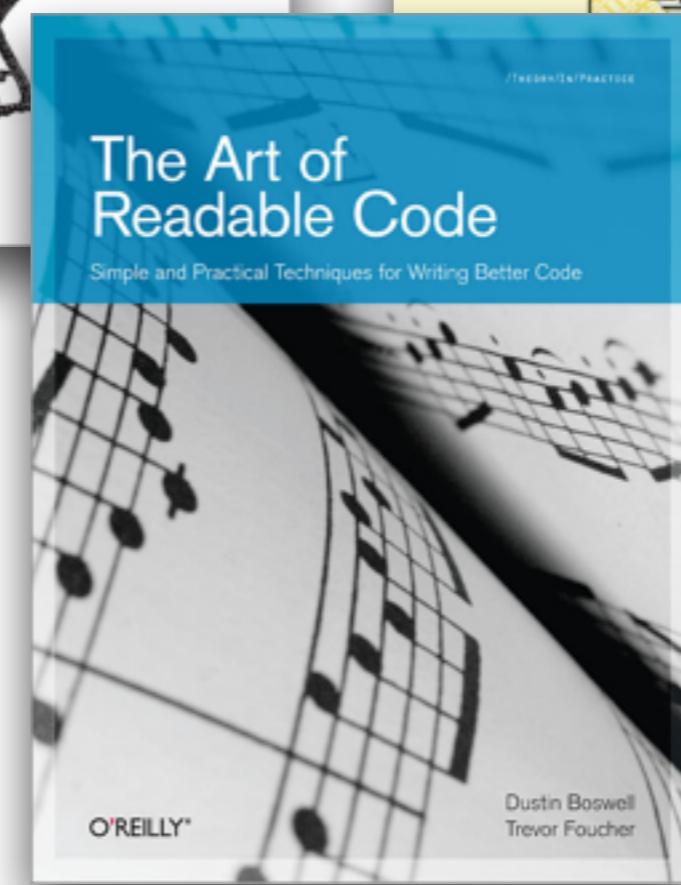
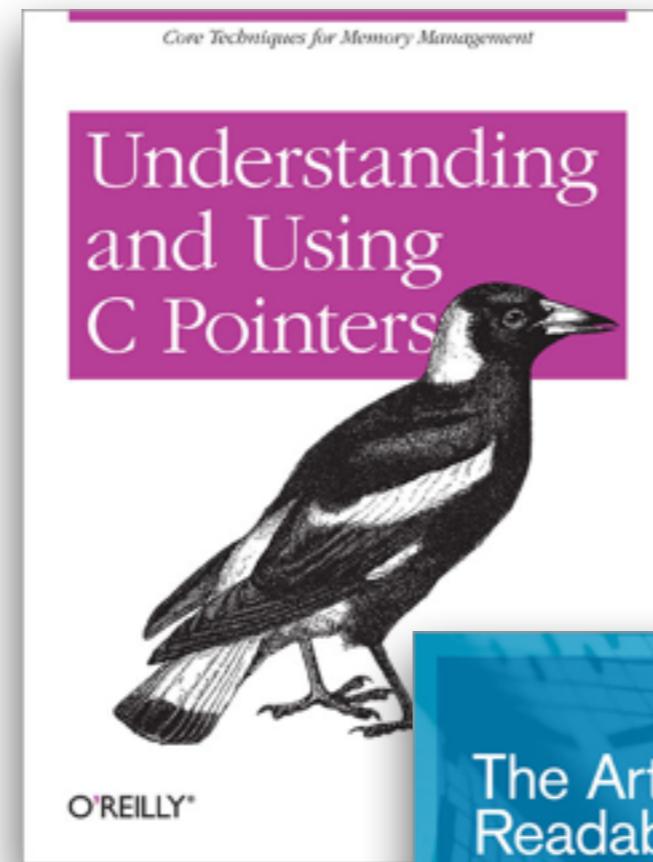
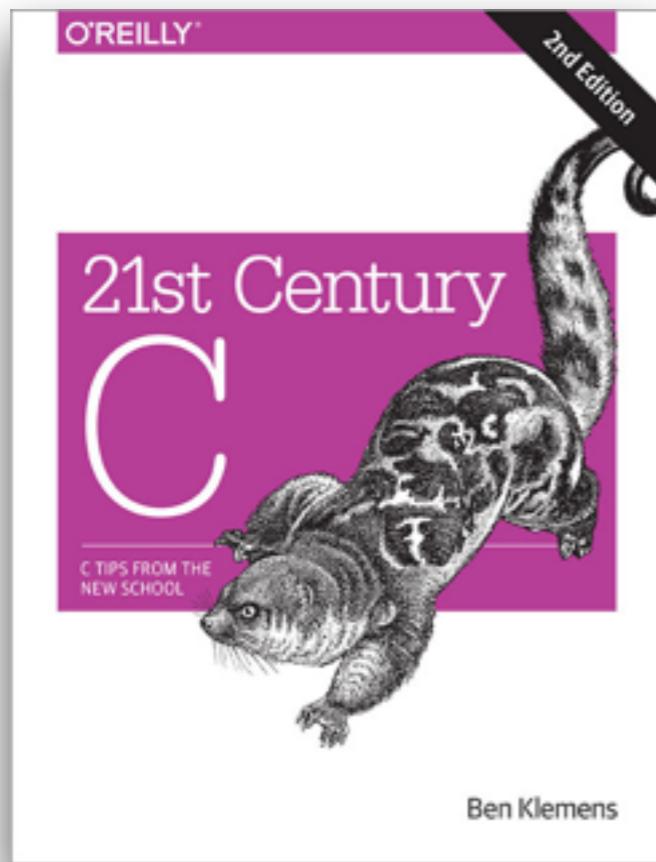
- Namn som överlappar med namn på standardfunktioner och -variabler

- Namn som är helt orelaterade

- Namn som innehåller tecken som är svåra att läsa



Några boktips



UU