Föreläsning 13

Tobias Wrigstad

Defensiv programmering
Läsbar kod (2/2)

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 beteer 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`)

```
#include <assert.h>
```

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

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 värda 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?

```
int average(int length, int values[])
{
 int sum = 0;

  for (int i=0; i < length; ++i)
  {
    sum += values[i];
  }

  return sum / length;
}
```

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

Defensiv programmering

- Vad kan gå fel i detta program?

```
int average(int length, int values[])
{
  assert(values);
  int sum = 0;

  for (int i=0; i < length; ++i)
  {
    sum += values[i];
  }

  return sum / length;
}
```

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

Defensiv programmering

- Vad kan gå fel i detta program?

```
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;
}
```

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

Defensiv programmering

- Vad kan gå fel i detta program?

```
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.

Borde verifieras vid anropsplatsen!

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

Defensiv programmering

- Vad kan gå fel i detta program?

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

```
#include <assert.h>
```

Defensiv programmering

- Vad kan gå fel i detta program?

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

Defensiv programmering

- Vad kan gå fel i detta program?

```
#define Buf_size 2048
```

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

  char *buffer = calloc(Buf_size, sizeof(char));
  ...
  strncpy(buffer, input, Buf_size);
  ...
}
```

Vad gör man när indatat inte är validt?

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)

Använd 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 bildrutan

Vad gör man när indatat inte är validt?

Ta ett angränsande validt 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 indatat inte är validt?

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

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 hårda 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 minska skadan från fel och för att minska 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 uppmärksammas under utveckling?

Från Steve McDonnell’s utmärkta ”Code Complete”

Skydda dig mot dig själv!

Rädda en framtida du — redan idag

Skydda dig mot dig själv

Vad kan gå fel i detta program?

```
if (foo())
  bar;

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

Inte robust vid utökning!

Skydda dig mot dig själv

Vad kan gå fel i detta program?

Makron kopierar text!

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

square(3+4);
```

Skydda dig mot dig själv

Vad kan gå fel i detta program?

```
#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
```

Illa!

Skydda dig mot dig själv

Vad kan gå fel i detta program?

```
#define Square(n) ((n)*(n))
```

Nu är det ”tydligt” att `Square` är ett Makro

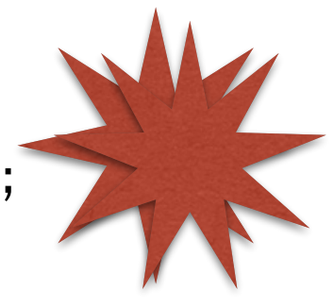
Skydda dig mot dig själv

Vad kan gå fel i detta program?

```
#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

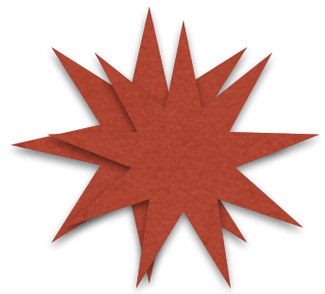
Vad kan gå fel i detta program?

```
#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 …
```



Skydda dig mot dig själv

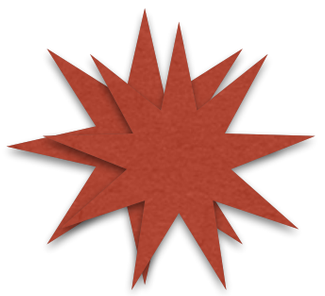
Vettigt användande av block!

Vad kan gå fel i detta program?

```
#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

Vad kan gå fel i detta program?

```
#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 = (struct person) { .name = "Peter", .age = 32 };
```

Namngiven initiering av struktar

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

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

C bjuder på initiering av `spouse`, `children` och `no_children`!

Namngiven initiering av struktar

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


person_t p = (person_t) { .name = "Peter", .age = 32 };

person_t p = (person_t) { .age = 32, .name = "Peter" };

person_t p = (person_t) { .name = "P", no_children = 0, .age = 32, };

person_t p = (person_t) { };
```

”Defaultparametrar” och namngivna argument

Med hjälp av ett makro kan vi skapa defaultvärden 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"));
```

”Defaultparametrar” och namngivna argument

Med hjälp av ett makro kan vi skapa defaultvärden 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”`.

Tips för att skriva bra (och läsbar) kod

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

Några tumregler för att skriva bra kod

- Tyddliggör 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 många rader
- Ersätt löpande namn på temporära 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

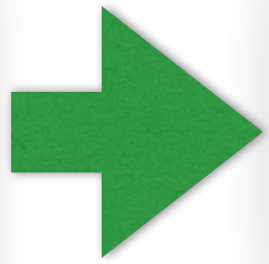
Undvik detta när du döper variabler

- 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

Upprepningar

- Om du ser upprepningar, ta bort dem!

```
puts("Name:");
fscanf("%s", name, stdin);
puts("Description:");
fscanf("%s", desc, stdin);
puts("Shelf:");
fscanf("%s", buf, stdin);
validate_shelf(buf);
```



```
ask_q_str("Name:", name);
ask_q_str("Desc:", desc);
ask_q_str("Shelf:", buf);
validate_shelf(buf);
```

Plus definition av `ask_q_str`

Indirektion är (nästan alltid) av godo

- Peta inte direkt i databasen — använd ett mellansteg

```
DB.goods[DB.total++] = g;
```

```
db->goods[db->total] = g;
++db->total;
```

```
add_good_to_db(db, g);
```

- *Om vi vill byta hur databasen är representerad (t.ex. från array till träd) behöver vi inte ändra utanför* `add_good_to_db` *i sista fallet!*

Struktur

- Att skriva kod är som att skriva prosatext
- Det viktigaste först så man inte missar det
- Saker som hör ihop tillsammans (t.ex. deklarera variabler nära där de används)
- Avstånd mellan orelaterade saker (styckebryt!)
- Lyft fram struktur och semantik genom kodstruktur
- Var konsekvent (t.ex. `Makro`)
- **Tydligt är bättre än kortfattat!**
- Var korrekt (`my_set` är ett dåligt namn på en lista)

https://github.com/IOOPM-UU/ioopm15/tree/master/extramaterial/utvikningar

Refaktorera: Exempel

```
void foo()
{
  g = 24;
}

int g; // global

int main(void)
{
  g = 42;

  foo();

  printf("%d\n", g);

  return 0;
}
```

Refaktorering:

Gör globala variabler lokala

Exempel [steg 1 — Flytta in variabeln i main]

```c
void foo()
{
  g = 24;
}

int main(void)
{
  int g; // local

  g = 42;

  foo();

  printf("%d\n", g);

  return 0;
}
```

```
globals.c: In function 'foo':
globals.c:5:3: error: 'g' undeclared (first use in this function)
  g = 24;
  ^
```

Exempel [steg 2 — lägg till parametrar där den används]

```c
void foo(int *g)
{
  *g = 24;
}

int main(void)
{
  int g; // local

  g = 42;

  foo();

  printf("%d\n", g);

  return 0;
}
```

```
globals.c: In function 'main':
globals.c:14:3: error: too few arguments to function 'foo'
  foo();
  ^
globals.c:3:6: note: declared here
 void foo(int *g)
      ^
```

Exempel [steg 3 — se till att skicka in den som argument]

```c
void foo(int *g)
{
  *g = 24;
}

int main(void)
{
  int g; // local

  g = 42;

  foo(&g);

  printf("%d\n", g);

  return 0;
}
```

Inga kompileringsfel

Programmen sida vid sida

```c
void foo()
{
  g = 24;
}

int g; // global

int main(void)
{
  g = 42;

  foo();

  printf("%d\n", g);

  return 0;
}
```

```c
void foo(int *g)
{
  *g = 24;
}

int main(void)
{
  int g; // local

  g = 42;

  foo(&g);

  printf("%d\n", g);

  return 0;
}
```

Vilken är bäst?

```
bool should_we_pick_bananas()
{
  if (gorilla_is_hungry())
    {
      if (bananas_are_ripe())
        {
          return true;
        }
      else
        {
          return false;
        }
    }
  else
    {
      return false;
    }
}
```

```
return gorilla_is_hungry() && bananas_are_ripe();
```

Från ”Become a Better Programmer”, Goodliffe, O’Reilly 2015

Vilken är bäst?

```
if (something)
  {
    return true;
  }
else
  {
    return false;
  }
```

```
return something;
```

Från ”Become a Better Programmer”, Goodliffe, O’Reilly 2015

Kodkommentarer

- Kodkommentarer kostar!

  De måste hållas i synk med vad koden faktiskt gör

- Kommentarer i kod skall berätta varför — inte vad

  Förklara sådant som läsaren har svårt att veta

- Kommentarer för kommentarers skull är alltid fel

- Ifrågasätt kommentarerna: bidrar de till något?

  This seems to work?

- Ha inte utkommenderad kod i programmet

Några boktips