Föreläsning 11

Andreina Francisco
Based on the slides by Tobias Wrigstad

Bra och läsbar kod
Defensiv programmering

Korrekthet och prestanda (performance) är inte allt

- Underhållsbarhet (maintainability)
- Portabilitet
- Läsbarhet
- Komplexitet
- testbarhet
- …

Alla dessa (7) är av lika vikt!

Det handlar alltså inte bara om att ”få det att funka”…

It’s not just about “making it work”

Tips för att skriva bra (och läsbar) kod

Några tumregler för att skriva bra kod

- Tydliggör beroenden mellan satser (Clarify dependencies between clauses)
- Ge namn för att tydliggöra beroenden och kopplingar
  (Give names to clarify dependencies and connections)
- Sista utväg: använd kommentarer för att lyfta fram beroenden som på inget annat sätt blir synliga i koden (use comments to highlight dependencies)

- Koden bör vara läsbar utifrån och in (Code should be readable from the outside in)
- Gruppera relaterade satser
- Faktorera ut orelaterade grupper till egna funktioner

Gruppera relaterade satser

```
node_t **node_find(node_t **n, int value)
{
  if (*n == NULL || Value(n) == value) return n;

  if (Value(n) > value) return node_find(Left(n), value);
  if (Value(n) < value) return node_find(Right(n), value);

  assert(false);
  return NULL;
}
```

Stoppvillkor

Villkor för att gå vidare i trädet

Defensiv del

Faktorera ut orelaterade grupper till egna funktioner

```
vara_t add_vara_to_db(Lager l)
{
  char input[256], *tmp, shelf[10], location[10];
  int price, number;
  vara_t prod = createProduct();
  printf("Skriv in varans namn: ");
  readline(input, 256, stdin);
  prod->name = malloc(strlen(input)+1);
  strncpy(prod->name, input, strlen(input));
  prod->name[strlen(input)] = '\0';
  puts(prod->name);

  printf("Beskrivning: ");
  readline(input, 256, stdin);
  prod->description = malloc(strlen(input)+1);
  strncpy(prod->description, input, strlen(input));
  prod->description[strlen(input)] = '\0';
  puts(prod->description);

  while(true)
    {
      while(true)
        {
          puts("Lagerhylla: ");
          readline(input, 256, stdin);
          tmp = shelfName(input);
          if (tmp == NULL)
            {
              printf("Fel försök igen.\n");
              continue;
            }
          strncpy (shelf, tmp, strlen(tmp)+1);
          puts(input);
          break;
        }

      while(true)
        {
          puts("Lagerhylla: ");
          readline(input, 256, stdin);
          tmp = shelfPlacement(input);
          if (tmp == NULL)
            {
              printf("Fel försök igen.\n");
              continue;
            }

          strncpy(location, tmp, strlen(tmp)+1);
          puts(input);

          break;
        }

      strcat(shelf, location);
      prod->location = malloc(strlen(shelf)+1);
      strncpy(prod->location, shelf, strlen(shelf));
      puts(prod->location);
      if (checkShelf(l, prod->location) == 1)
        {
          printf("Upptaget, väl en annan plats.\n");
          continue;
        }
      else
        {
          break;
        }
    }

  price = ask_int_question("Priset: ");
  prod->price = price;
  printf("%d\n", prod->price);

  number = ask_int_question("Antal: ");
  prod->number = number;
  printf("%d\n", prod->number);

  puts("============================");
  printf("Varans namn :        %s\n", prod->name);
  printf("       beskrivning : %s\n", prod->description);
  printf("       plats :       %s\n", prod->location);
  printf("       pris :        %d\n", prod->price);
  printf("       lagerplats :  %d\n", prod->number);
  puts("============================");

  return prod;
}
```

Många bra egenskaper (characteristics)

– Tydliga (Clear) grupper

– Läcker inte minne (memory)

– Bra namngivning

– Korrekt

1 funktion — 86 rader

Undvik (Avoid!) magiska konstanter

```
vara_t add_vara_to_db(Lager l)
{
  char input[256], *tmp, shelf[10], location[10];
  int price, number;
  vara_t prod = vara_new();
  printf("Skriv in varans namn: ");
  readline(input, 256, stdin);
  prod->name = malloc(strlen(input)+1);
  strncpy(prod->name, input, strlen(input));
  prod->name[strlen(input)] = '\0';
  puts(prod->name);

  printf("Beskrivning: ");
  readline(input, 256, stdin);
  prod->description = malloc(strlen(input)+1);
  strncpy(prod->description, input, strlen(input));
  prod->description[strlen(input)] = '\0';
  puts(prod->description);

  for(;;)
    {
      while(true)
        {
          puts("Lagerhylla: ");
          readline(input, 512, stdin);
          tmp = shelfName(input);
          if (tmp == NULL)
            {
              printf("Fel försök igen.\n");
              continue;
            }
          strncpy (shelf, tmp, strlen(tmp)+1);
```

Jobbigt att ta sig igenom (through)

Gruppera saker efter samhörighet (affinity)

```
vara_t add_vara_to_db(Lager l)
{
  char input[256], *tmp, shelf[10], location[10];
  int price, number;
  vara_t prod = vara_new();
  printf("Skriv in varans namn: ");
  readline(input, 256, stdin);
  prod->name = malloc(strlen(input)+1);
  strncpy(prod->name, input, strlen(input));
  prod->name[strlen(input)] = '\0';
  puts(prod->name);

  printf("Beskrivning: ");
  readline(input, 256, stdin);
  prod->description = malloc(strlen(input)+1);
  strncpy(prod->description, input, strlen(input));
  prod->description[strlen(input)] = '\0';
  puts(prod->description);

  for(;;)
    {
      while(true)
        {
          puts("Lagerhylla: ");
          readline(input, 512, stdin);
          tmp = shelfName(input);
          if (tmp == NULL)
            {
              printf("Fel försök igen.\n");
              continue;
            }
          strncpy (shelf, tmp, strlen(tmp)+1);
```

Används 70 rader senare

Gruppera saker efter samhörighet

```
vara_t add_vara_to_db(Lager l)
{
  char input[256], *tmp, shelf[10], location[10];
  int price, number;

  vara_t prod = vara_new();

  printf("Skriv in varans namn: ");
  readline(input, 256, stdin);
  prod->name = malloc(strlen(input)+1);
  strncpy(prod->name, input, strlen(input));
  prod->name[strlen(input)] = '\0';
  puts(prod->name);

  printf("Beskrivning: ");
  readline(input, 256, stdin);
  prod->description = malloc(strlen(input)+1);
  strncpy(prod->description, input, strlen(input));
  prod->description[strlen(input)] = '\0';
  puts(prod->description);

  for(;;)
    {
      while(true)
        {
          puts("Lagerhylla: ");
          readline(input, 512, stdin);
          tmp = shelfName(input);
          if (tmp == NULL)
            {
              printf("Fel försök igen.\n");
              continue;
```

Gruppera saker efter samhörighet

```
vara_t add_vara_to_db(Lager l)
{
  char input[256], *tmp, shelf[10], location[10];
  int price, number;

  vara_t prod = vara_new();

  printf("Skriv in varans namn: ");
  readline(input, 256, stdin);
  prod->name = malloc(strlen(input)+1);
  strncpy(prod->name, input, strlen(input));
  prod->name[strlen(input)] = '\0';
  puts(prod->name);

  printf("Beskrivning: ");
  readline(input, 256, stdin);
  prod->description = malloc(strlen(input)+1);
  strncpy(prod->description, input, strlen(input));
  prod->description[strlen(input)] = '\0';
  puts(prod->description);

  for(;;)
    {
      while(true)
        {
          puts("Lagerhylla: ");
          readline(input, 512, stdin);
          tmp = shelfName(input);
          if (tmp == NULL)
            {
              printf("Fel försök igen.\n");
              continue;
```

Slacksvariabler

Object of interest

En inläsning av namn

En inläsning av beskrivning

Under Jura-perioden hade vi inte editorer som kunde ”go to definition” — det skall inte påverka hur vi skriver kod 2019

Undvik (Avoid!) nästlade loopar

```
for(;;)
   {
     while(true)
       {
         puts("Lagerhylla: ");
         readline(input, 256, stdin);
         tmp = shelfName(input);
         if (tmp == NULL)
           {
             printf("Fel försök igen.\n");
             continue;
           }
       strncpy (shelf, tmp, strlen(tmp)+1);
       puts(input);
       break;
     }

  while(true)
    {
      puts("Lagerhylla: ");
      readline(input, 256, stdin);
      tmp = shelfPlacement(input);
      if (tmp == NULL)
        {
          printf("Fel försök igen.\n");
          continue;
        }

      strncpy(location, tmp, strlen(tmp)+1);
      puts(input);

      break;
```

När avslutas denna kod?

Undvik upprepningar

```
vara_t add_vara_to_db(Lager l)
{
  char input[256], *tmp, shelf[10], location[10];
  int price, number;
  vara_t prod = createProduct();
  printf("Skriv in varans namn: ");
  readline(input, 256, stdin);
  prod->name = malloc(strlen(input)+1);
  strncpy(prod->name, input, strlen(input));
  prod->name[strlen(input)] = '\0';
  puts(prod->name);

  printf("Beskrivning: ");
  readline(input, 256, stdin);
  prod->description = malloc(strlen(input)+1);
  strncpy(prod->description, input, strlen(input));
  prod->description[strlen(input)] = '\0';
  puts(prod->description);

  while(true)
    {
      while(true)
        {
          puts("Lagerhylla: ");
          readline(input, 256, stdin);
          tmp = shelfName(input);
          if (tmp == NULL)
            {
              printf("Fel försök igen.\n");
              continue;
            }
          strncpy (shelf, tmp, strlen(tmp)+1);
          puts(input);
          break;
        }

      while(true)
        {
          puts("Lagerhylla: ");
          readline(input, 256, stdin);
          tmp = shelfPlacement(input);
          if (tmp == NULL)
            {
              printf("Fel försök igen.\n");
              continue;
            }

          strncpy(location, tmp, strlen(tmp)+1);
          puts(input);

          break;
        }

      strcat(shelf, location);
      prod->location = malloc(strlen(shelf)+1);
      strncpy(prod->location, shelf, strlen(shelf));
      puts(prod->location);
      if (checkShelf(l, prod->location) == 1)
        {
          printf("Upptaget, väl en annan plats.\n");
          continue;
        }
      else
        {
          break;
        }
    }

  price = ask_int_question("Priset: ");
  prod->price = price;
  printf("%d\n", prod->price);

  number = ask_int_question("Antal: ");
  prod->number = number;
  printf("%d\n", prod->number);

  puts("============================");
  printf("Varans namn :        %s\n", prod->name);
  printf("       beskrivning : %s\n", prod->description);
  printf("       plats :       %s\n", prod->location);
  printf("       pris :        %d\n", prod->price);
  printf("       lagerplats :  %d\n", prod->number);
  puts("============================");

  return prod;
}
```

```
vara_t add_vara_to_db(Lager l)
{
  char input[256], *tmp, shelf[10], location[10];
  int price, number;
  vara_t prod = createProduct();
  printf("Skriv in varans namn: ");
  readline(input, 256, stdin);
  prod->name = malloc(strlen(input)+1);
  strncpy(prod->name, input, strlen(input));
  prod->name[strlen(input)] = '\0';
  puts(prod->name);

  printf("Beskrivning: ");
  readline(input, 256, stdin);
  prod->description = cmalloc(strlen(input)+1);
  strncpy(prod->description, input, strlen(input));
  prod->description[strlen(input)] = '\0';
  puts(prod->description);

  while(true)
    {
      while(true)
        {
          puts("Lagerhylla: ");
          readline(input, 256, stdin);
          tmp = shelfName(input);
          if (tmp == NULL)
            {
              printf("Fel försök igen.\n");
              continue;
            }
          strncpy (shelf, tmp, strlen(tmp)+1);
          puts(input);
          break;
        }

      while(true)
        {
          puts("Lagerhylla: ");
          readline(input, 256, stdin);
          tmp = shelfPlacement(input);
          if (tmp == NULL)
            {
              printf("Fel försök igen.\n");
              continue;
            }

          strncpy(location, tmp, strlen(tmp)+1);
          puts(input);
```

readline
strncpy
256

Faktorera ut orelaterade grupper till egna funktioner

```
vara_t add_vara_to_db(Lager l)
{
  char input[256], *tmp, shelf[10], location[10];
  int price, number;
  vara_t prod = createProduct();
  printf("Skriv in varans namn: ");
  readline(input, 256, stdin);
  prod->name = malloc(strlen(input)+1);
  strncpy(prod->name, input, strlen(input));
  prod->name[strlen(input)] = '\0';
  puts(prod->name);

  printf("Beskrivning: ");
  readline(input, 256, stdin);
  prod->description = malloc(strlen(input)+1);
  strncpy(prod->description, input, strlen(input));
  prod->description[strlen(input)] = '\0';
  puts(prod->description);

  while(true)
    {
      while(true)
        {
          puts("Lagerhylla: ");
          readline(input, 256, stdin);
          tmp = shelfName(input);
          if (tmp == NULL)
            {
              printf("Fel försök igen.\n");
              continue;
            }
          strncpy (shelf, tmp, strlen(tmp)+1);
          puts(input);
          break;
        }

      while(true)
        {
          puts("Lagerhylla: ");
          readline(input, 256, stdin);
          tmp = shelfPlacement(input);
          if (tmp == NULL)
            {
              printf("Fel försök igen.\n");
              continue;
            }

          strncpy(location, tmp, strlen(tmp)+1);
          puts(input);

          break;
        }

      strcat(shelf, location);
      prod->location = malloc(strlen(shelf)+1);
      strncpy(prod->location, shelf, strlen(shelf));
      puts(prod->location);
      if (checkShelf(l, prod->location) == 1)
        {
          printf("Upptaget, väl en annan plats.\n");
          continue;
        }
      else
        {
          break;
        }
    }

  price = ask_int_question("Priset: ");
  prod->price = price;
  printf("%d\n", prod->price);

  number = ask_int_question("Antal: ");
  prod->number = number;
  printf("%d\n", prod->number);

  puts("============================");
  printf("Varans namn :        %s\n", prod->name);
  printf("       beskrivning : %s\n", prod->description);
  printf("       plats :       %s\n", prod->location);
  printf("       pris :        %d\n", prod->price);
  printf("       lagerplats :  %d\n", prod->number);
  puts("============================");

  return prod;
}
```

ask_string_question

ask_string_question

ask_shelf_question

ask_shelf_question

ask_int_question

ask_int_question

display_vara

Resultat av att bryta ut funktionerna

```
vara_t add_vara(db_t *db)
{
  vara_t prod = vara_new();

  prod->name = ask_question_string("Skriv in varans namn:");
  prod->desc = ask_question_string("Beskrivning:");

  do
    {
      char *location = ask_question_shelf("Lagerhylla: ");

      if (already_in_use(db, location))
        {
          printf("Upptaget, välj en annan plats.\n");
          free(location);
        }
      else
        {
          prod->location = location;
        }
    }
  while (prod->location == NULL);

  prod->price = ask_question_int("Priset: ");
  prod->number = ask_question_int("Antal: ");

  display_vara(prod);

  return prod;
}
```

- Färre temporära värden
- Temporära värden allokeras nära användning
- Funktionsanropen lyften abstraktionsnivån på koden
- Läsbar **utifrån och in**: *Att verifiera överens-stämmelse (compliance) med specen enklare*
- Får plats på skärmen™

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 (be consistent!!)
- Utveckla (develop/use) konventioner (och dokumentera dem)
- Skilj ut (separate) lokala / privata/ globala data
- Skilj ut konstanter / uppräkningsbara (enumeration or enum) typer / variabler
- Välj en namnformattering efter läsbarhet
- Tag hänsyn till (take into account) språkstandardarden i utformandet av namnkonventionen

    T.ex. skillnad i namngivning mellan Makron och funk_tioner

Undvik (AVOID!) detta när du döper variabler

- Missledande (array vs list) eller tvetydiga (ambiguous) namn (state / temp)
- Namn med liknande innebörd (meaning) -> e.g. get/fetch/retrieve
- Also avoid using the same name for two different concepts
- Namn som är identiska upp till 1–2 tecken -> my_new_node_l / my_new_node_r
- Namn som innehåller siffror
- Medvetna felstavningar i syfte att förkorta namn (Deliberate misspellings in order to shorten names) -> Seriously! NO!
- Namn på ord som ofta stavas fel eller läses fel (misspelled, misread, hard to say)
- Namn som överlappar med namn på standardfunktioner och -variabler
- Namn som är helt orelaterade (my_cow)
- Namn som innehåller tecken som är svåra att läsa (`I` vs `l` vs `1` / `O` vs `0` vs `Q` / etc)

Vilken är bäst?

```
bool node_contains(node_t **n, int value)
{
  return *node_find(n, value) != NULL;
}
```

```
bool node_contains(node_t **node, int value)
{
  return *node_find(node, value) != NULL;
}
```

```
bool node_contains(node_t **root_of_tree, int value)
{
  return *node_find(root_of_tree, value) != NULL;
}
```

Vilken är bäst?

```
bool node_contains(node_t **n, int value)
{
  return *node_find(n, value) != NULL;
}
```

```
bool node_contains(node_t **node, int value)
{
  return *node_find(node, value) != NULL;
}
```

```
bool node_contains(node_t **root_of_tree, int value)
{
  return *node_find(root_of_tree, value) != NULL;
}
```

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

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);
```

- No need to know how the database is implemented -> Avoid unnecessary work/bugs
- *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 (Important things first)
- Saker som hör ihop tillsammans (t.ex. deklarera variabler nära där de används)
- Avstånd mellan orelaterade saker (styckebryt!) (separate unrelated statements)
- Lyft fram struktur och semantik genom kodstruktur (semantics / meaning / comments)
- Var konsekvent (t.ex. `Makro`) (be consistent)
- **Tydligt är bättre än kortfattat! (Clear is MUCH better than concise!)**
- Var korrekt (`my_set` är ett dåligt namn på en lista)

Refaktorera i diskreta steg: 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;
}
```

Kodkommentarer

- Kodkommentarer kostar!

  De måste hållas i synk med vad koden faktiskt gör (sync comments with changes!)

- Kommentarer i kod skall berätta varför — inte vad (why oh why!)

  Förklara sådant som läsaren har svårt att veta (the reader already knows the what)

- Kommentarer för kommentarers skull är alltid fel (get_name() -> returns the name of…)
- Ifrågasätt kommentarerna: bidrar de till något? (is this useful? REALLY?)

  This seems to work?

- Ha inte utkommenderad kod i programmet (should it be?)

Kommentarer ≠ Koddokumentation

- Verktyg som Doxygen låter oss lägga koddokumentation i kommentarer som extraheras och skapar HTML-filer
- Med kommentarer avses inte koddokumentation, som vi med fördel gör i header-filerna

Defensiv programmering

Observationer kring ”vanlig programmering”

- Programmerare tenderar att fokusera på de problem som måste lösas för att programmet skall ”fungera” (first version?)
- Programmerare gör vanligen antaganden kring t.ex. (common assumptions)

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

  Hur miljön som programmet kör i beteer sig (behaving) (t.ex. ingen tar bort katalogen jag står i under körning) (I’m the only one here)

  Användaren är vänligt inställd (och matar in korrekta data) (the user is friendly and doesn’t make mistakes)

- Nya programmerare kan ofta glömma (forget) t.ex.

  Att program förändras och muterar över tid

  Att en rad kod läses oftare än den modifieras (read > modified)

Defensiv programmering

- En ”princip” för att skapa feltolerant kod, introducerades av Kernighan och Ritchie

  1.) Gör aldrig några antaganden (avoid making assumptions)

  2.) Klä skott för misstag både inifrån och utifrån (även din kod förändras)

  3.) Tillämpa standarder (apply standards)

  4.) ”Keep it simple” (KISS)

- 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 (drive safe!)

  Handlar i grund och botten om att också ta ansvar även för ”andras” fel (it’s your problem too)

  Mjukvaran skall fungera korrekt även med trasig indata (error handling)

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

”Skit in–skit ut!” (GIGO)

- En dålig princip! (how do you know it’s garbage??)
- Istället: (let’s find other options!)

  Skit in – inget ut

  Skit in – felmeddelande ut

  Skit in är inte möjligt

Förhållandet (Relationship) 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 (the programmer didn't think of it)

- Defensiv programmering menar att vi skall ”anta det värsta om all indata” (assume the worst)

  Fångar fel innan de leder till problem (early bug detection)

  Förenklar (simplifies) 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 (size limits)?

- Omöjliga värden (in the given context)

  Negativ skostorlek? (shoe size / height / weight)

- Pre/postvillkor – använd som valideringsvillkor (use conditions for validation)

- Antaganden (assumptions) 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

  Om 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 (make your code reliable)

- En assertion har normalt 1–2 komponenter

  Ett villkor som förväntas hålla under körning (condition)

  Ett (frivilligt) felmeddelande (error message / description)

Olika program kräver (requires) olika felhantering

- Robusthet = undertryck fel (suppress errors)

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

  Datorspel (ingen märker om ett event ”försvinner”)

- Korrekthet = undertryck aldrig fel (NEVER suppress/ignore errors)

  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`
- `Other ideas / solutions?`

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 Evil Corp kan man vänta till nästa Evil Corp stock tick

Om man läser av tryck (pressure) 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? (back out safely)

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

(Make sure that errors are not unknowingly / unconsciously suppressed)

Exempel: (Check GIGO!)

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 (interrupt) programmets exekvering

Check full memory: 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

Input is too long / big / out of range

Kapsla in/isolera fel

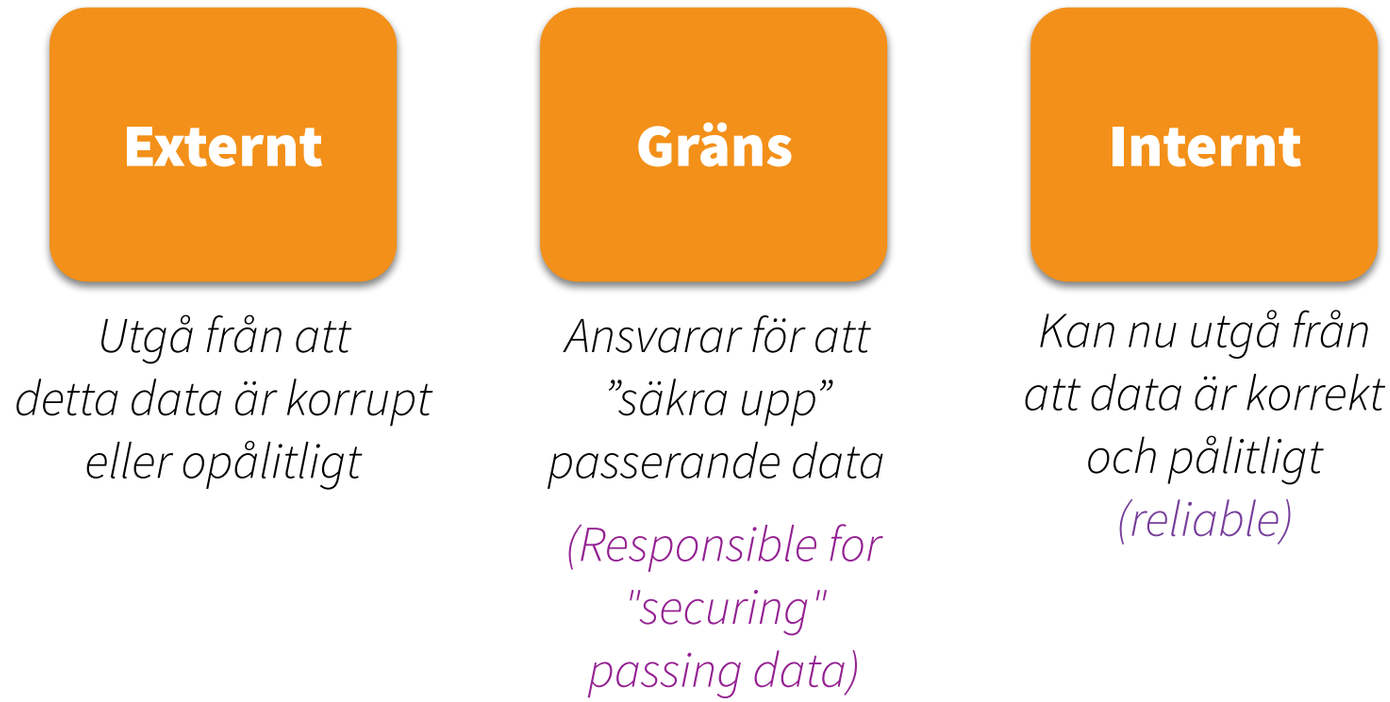
För mycket felhantering är också en felkälla

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

Exempel

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

Ha flera kritiska ringar som kontrollerar olika typer av fel



Produktionskod och utvecklingskod (development)

- Utvecklingskoden kan ofta ta sig friheter som inte produktionkoden kan

  Time efficiency: Behöver inte gå lika fort

  Memory/etc efficiency: Behöver inte vara lika snål med resurser

  etc.

- Detta ger ökad frihet att skriva utvecklingkod som underlättar debuggning och felsökning (you're free to obsess and tripple check that everything is as expected)

  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 (reaches) produktionskoden

- Lämna kvar kontroller för viktiga fel (error handling)
- Ta bort kontroller för triviala fel (bye bye tripple checks)
- Ta bort kontroller som terminerar med hårda **kraschar** vid invalitt data (no drama!)
- Lämna kvar kod som hjälper programmet terminera på ett förtjänstfullt sätt (again, avoid severe crashes here!)
- Logga fel för att underlätta felsökning (troubleshooting)
- Se till att alla felmeddelanden är trevliga (nice and informative!)

  ”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 McConnell’s utmärkta ”Code Complete”

Checklist: Defensive Programming

- Does the routine protect itself from bad input data?
- Have you used assertions to document assumptions, including preconditions and postconditions?
- Have assertions been used only to document conditions that should never occur?
- Have barricades been created to contain the damaging effect of errors and reduce the amount of code that has to be concerned about error processing?
- Has information hiding been used to contain the effects of changes so that they won't affect code outside the routine or class that is changed?
- Have debugging aids been installed in such a way that they can be activated or deactivated without a great deal of fuss?
- Is the amount of defensive programming code appropriate—neither too much nor too little?
- Have you used offensive programming techniques to make errors difficult to overlook during development?

From Steve McConnell's excellent "Code Complete"

Skydda dig mot dig själv!

(Bonusmaterial för den hugade)

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
```

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

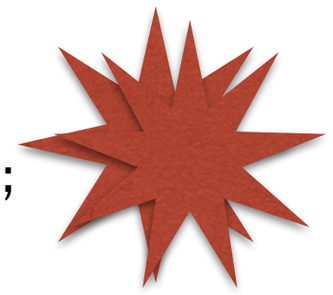
Skydda dig mot dig själv

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);
```



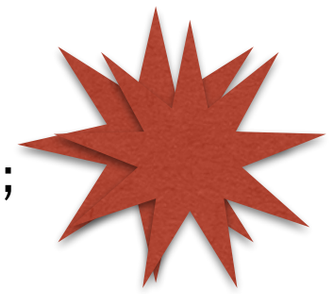
Skydda dig mot dig själv

Vad kan gå fel i detta program?

```
if (condition)
   T varname …
```

expanderas till

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



Skydda dig mot dig själv

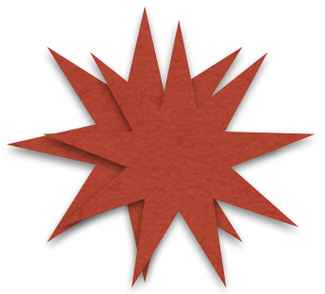
Vettigt användande av block!

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.)

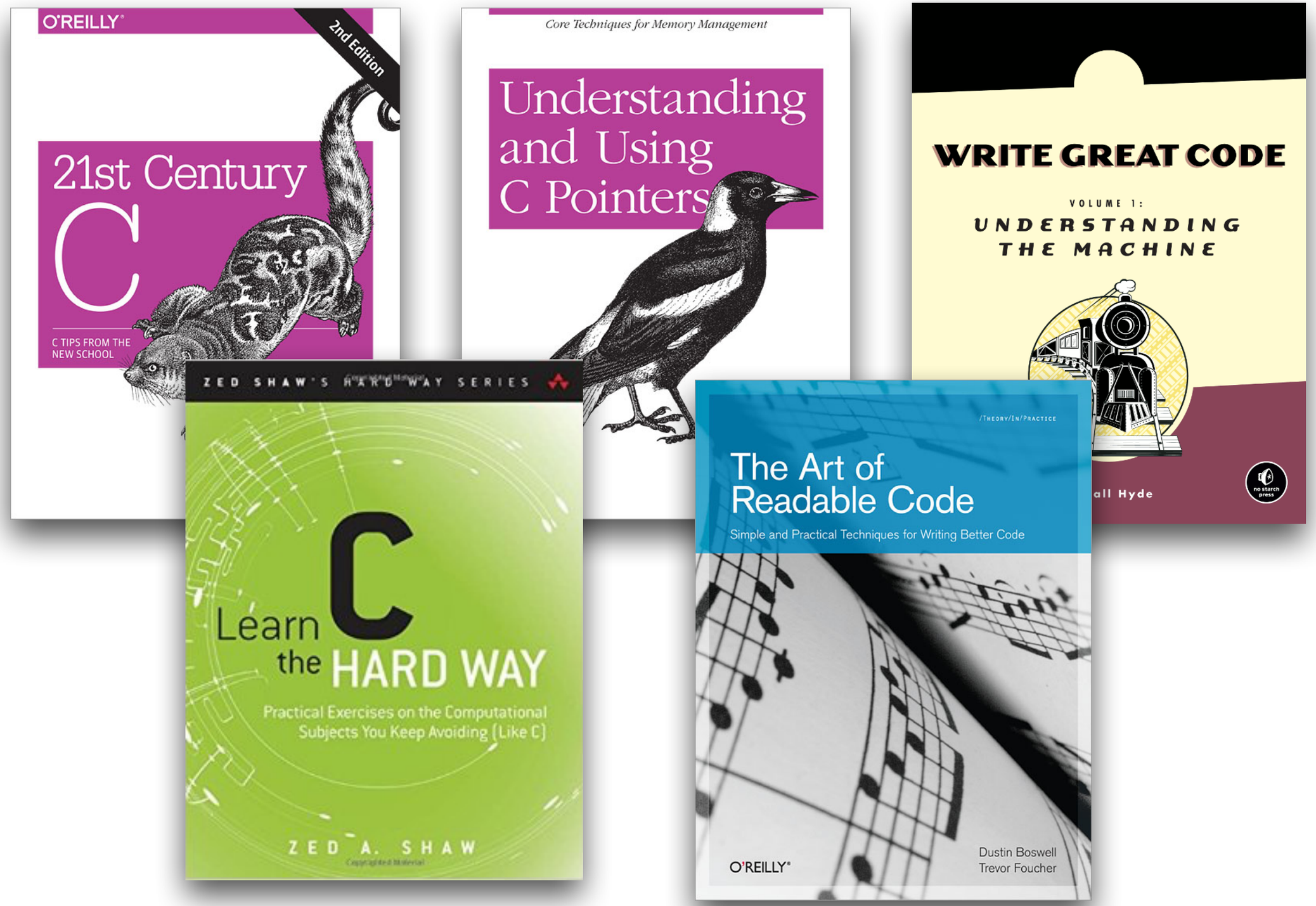
```
#define Person(__VARGS__) \
    ((struct person) { .name=”Fred”, __VARGS__ })

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

register_person(Person(.name = ”Bob"));
```

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

Några boktips



Föreläsning 12

Andreina Francisco
Based on the slides by Tobias Wrigstad

Bitmanipulering
Preprocessorn

Preprocessorn

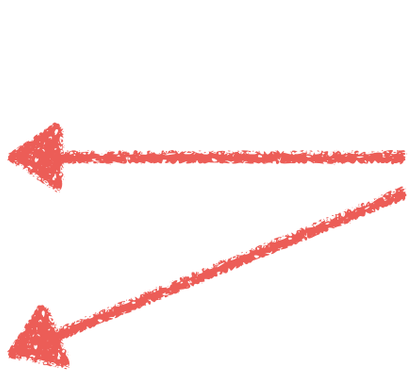
CPP — The C PreProcessor

- Text Replacement happens before the compiler
- Vi har sett några exempel tidigare:

```
#include <stdio.h>
#include "myfile.h"
```

Includes other files

```
#ifndef symbol
#define symbol
...
#endif
```



If-statement that is run during compilation

```
#define INT_MAX   2147483647
```

Definition av konstant (eg. makro)

- CPP — runs before the compiler automatically (rarely or never runs individually)

Preprocessor “operations”

- Continued lines (lines ending in \ ) are merged into one long line
- Göra om kod till strängar #

  aka Token stringification: The # operator (known as the "Stringification Operator") converts a token into a C string literal

- Konkatenera uttryck ##

  aka Token concatenation: The ## operator (known as the "Token Pasting Operator") concatenates two tokens into one token.

- You can also turn flags on or off during compilation
- All comments are replaced with single spaces

Preprocessor “operations”

```
// CPP program to illustrate (#) operator
#include <stdio.h>
#define mkstr(s) #s
int main(void)
{
    printf(mkstr(p=“42\n”));
    return 0;
}
```

```
  Output: "p = \"42\\n\";"
```

```
// Program to illustrate (##) operator
#include <stdio.h>
#define concat(a, b) a##b
int main(void)
{
    int xy = 30;
    printf("%d", concat(x, y));
    return 0;
}
```

```
    Output: 30
```

```c
#define DECLARE_LIST(name, elem) \
  typedef struct name##_t name##_t; \
  typedef bool (*name##_cmp_fn)(elem* a, elem* b); \
  typedef elem* (*name##_map_fn)(elem* a, void* arg); \
  typedef void (*name##_free_fn)(elem* a); \
  name##_t* name##_pop(name##_t* list, elem** data); \
  name##_t* name##_push(name##_t* list, elem* data); \
  name##_t* name##_append(name##_t* list, elem* data); \
  name##_t* name##_next(name##_t* list); \
  name##_t* name##_index(name##_t* list, ssize_t index); \
  elem* name##_data(name##_t* list); \
  elem* name##_find(name##_t* list, elem* data); \
  ssize_t name##_findindex(name##_t* list, elem* data); \
  bool name##_subset(name##_t* a, name##_t* b); \
  bool name##_equals(name##_t* a, name##_t* b); \
  name##_t* name##_map(name##_t* list, name##_map_fn f, void* arg); \
  name##_t* name##_reverse(name##_t* list); \
  size_t name##_length(name##_t* list); \
  void name##_free(name##_t* list); \
```

Preprocessing Macros

- För robusthet — sätt parenteser runt varje arguments användande, och hela makrot

```
#define Max(a,b) ( (a) < (b) ? (b) : (a) )
```

- For longer macros, use the following form:

```
do {  ...  } while (0);
```

Robusta preprocessormakron?

- Hitta felen!

```
#define sqr_a(x) (x)*(x)

#define sqr_b(x) (x)*(x)

#define sqr_c(x) x*x

int a = 9;

int b = sqr_a(++a);

int c = sqr_b(a)+10;

int b = sqr_c(a-10);
```

Preprocessormakron

- Textual replacement does not work “like C”:

```
#define Max(a,b) a < b ? b : a


Max(x + y, z);          // x + y < z ? z : x + y
Max(++x, z);            // ++x < z ? z : ++x
```

Robusta preprocessormakron?

- Exchange / swap two values without a temporal variable by using XOR

```
#define Swap(a,b) { \
          a ^= b;   \
          b ^= a;   \  // Illustrerar 5-radersmakro
          a ^= b;   \
        }           \
```

- Funkar bra

```
if (x < y)
  Swap(x, y);
```

Robusta preprocessormakron?

- Byt plats på två värden utan en tredje ”slaskvariabel”, mha xor

```
#define Swap(a,b) a ^= b; b ^= a; a ^= b;
```

- Funkar inte så bra (varför?!)

```
if (x < y)
  Swap(x, y);
```

- Kompilerar inte ens

```
if (x < y)
  Swap(x, y);
else
  Swap(x, z);
```

Printline debugging

- I lämplig headerfil

```
#define Debug
```

- Överallt i resten av koden

```
#ifdef Debug
printf("…"); // Spårutskrift
#endif
```

- This is often used for other conditions in the code, for example, platform, OS, etc.

```c
#ifndef __min_assert_h__
#define __min_assert_h__

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

#define __minute__assert(kind, msg, b)                                  \
  if (b) { printf("Assertion failed: %s(%s), file %s, line %d\n",       \
                  kind, msg, __FILE__, __LINE__); exit(EXIT_FAILURE); } \

#define assertTrue(arg)                                                 \
  do { __minute__assert("assertTrue",  #arg, !(arg)); } while (0); \
#define assertFalse(arg)                                                \
  do { __minute__assert("assertFalse", #arg,  (arg)); } while (0); \

#endif
```

massert.h

```c
#ifndef __min_assert_h__
#define __min_assert_h__

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

#define __minute__assert(kind, msg, b)                                  \
  if (b) { printf("Assertion failed: %s(%s), file %s, line %d\n",       \
                  kind, msg, __FILE__, __LINE__); exit(EXIT_FAILURE); } \

#define assertTrue(arg)                                                 \
  do { __minute__assert("assertTrue", #arg, !(arg)); } while (0); \
#define assertFalse(arg)                                                \
  do { __minute__assert("assertFalse", #arg,  (arg)); } while (0); \

#endif
```

massert.h

```c
#include "massert.h"

int main(int argc, char *argv[])
{
  assertTrue(argc > 1);
  return 0;
}
```

test.c

```
$ gcc -Wall -g -o test test.c
$ ./test
Assertion failed: assertTrue(argc > 1), file test.c, line 5
```

```
$ cpp -E test.c

... // bortklippt ”skräp”

int main(int argc, char *argv[])
{
  do { if (!(argc > 1)) { printf("Assertion failed:
    %s(%s), file %s, line %d\n", "assertTrue",
      #argc > 1, "test.c", 5); exit(1); }; } while (0);;
  return 0;
}
```

```c
#include "massert.h"

int main(int argc, char *argv[])
{
  assertTrue(argc > 1);
  return 0;
}
```

test.c

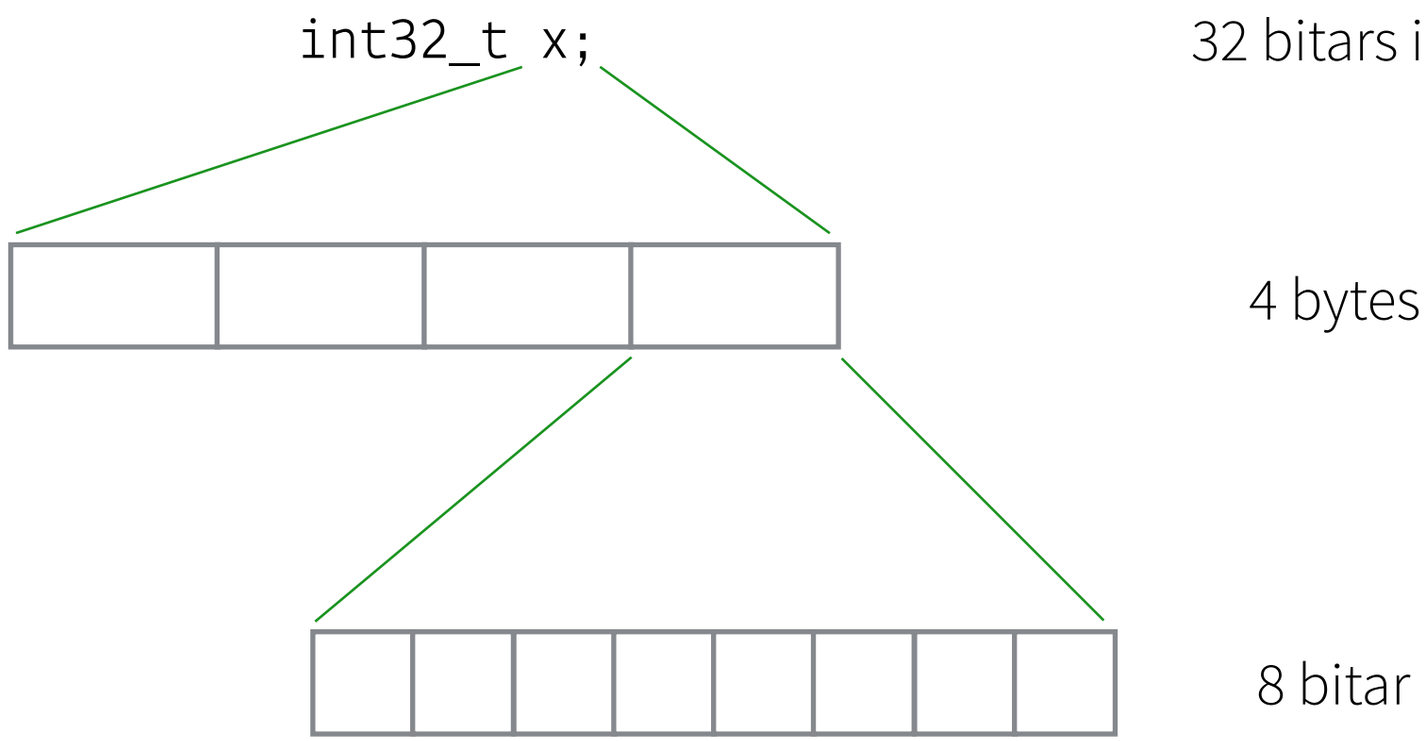
Bit Manipulation

Bitmanipulering

- Commands for manipulating individual bits
- We have seen that C looks at the memory as an array of chars (usually bytes)
- Now we shall see that it is possible to look at them as arrays of individual bits

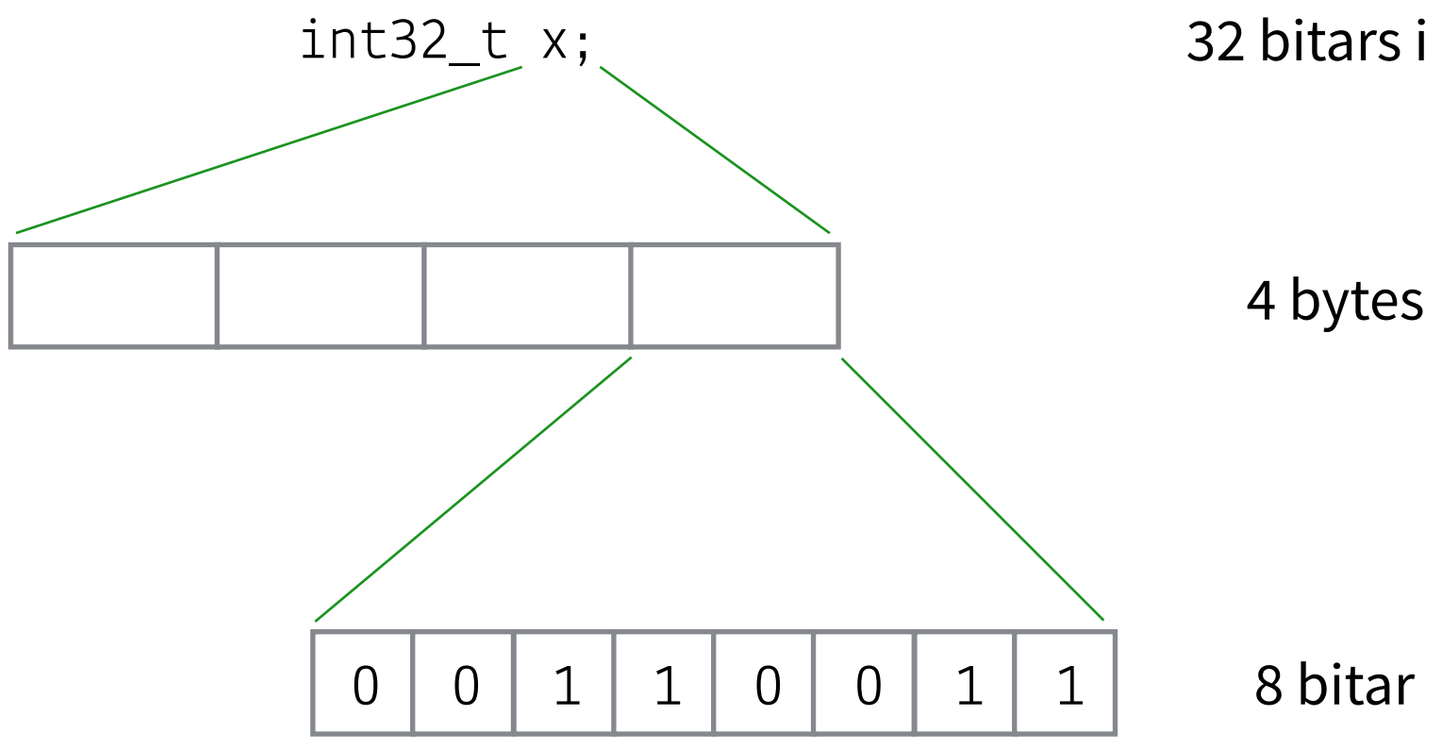
- *Note that all the code that manipulates individual bits is difficult to port!*

Bitar



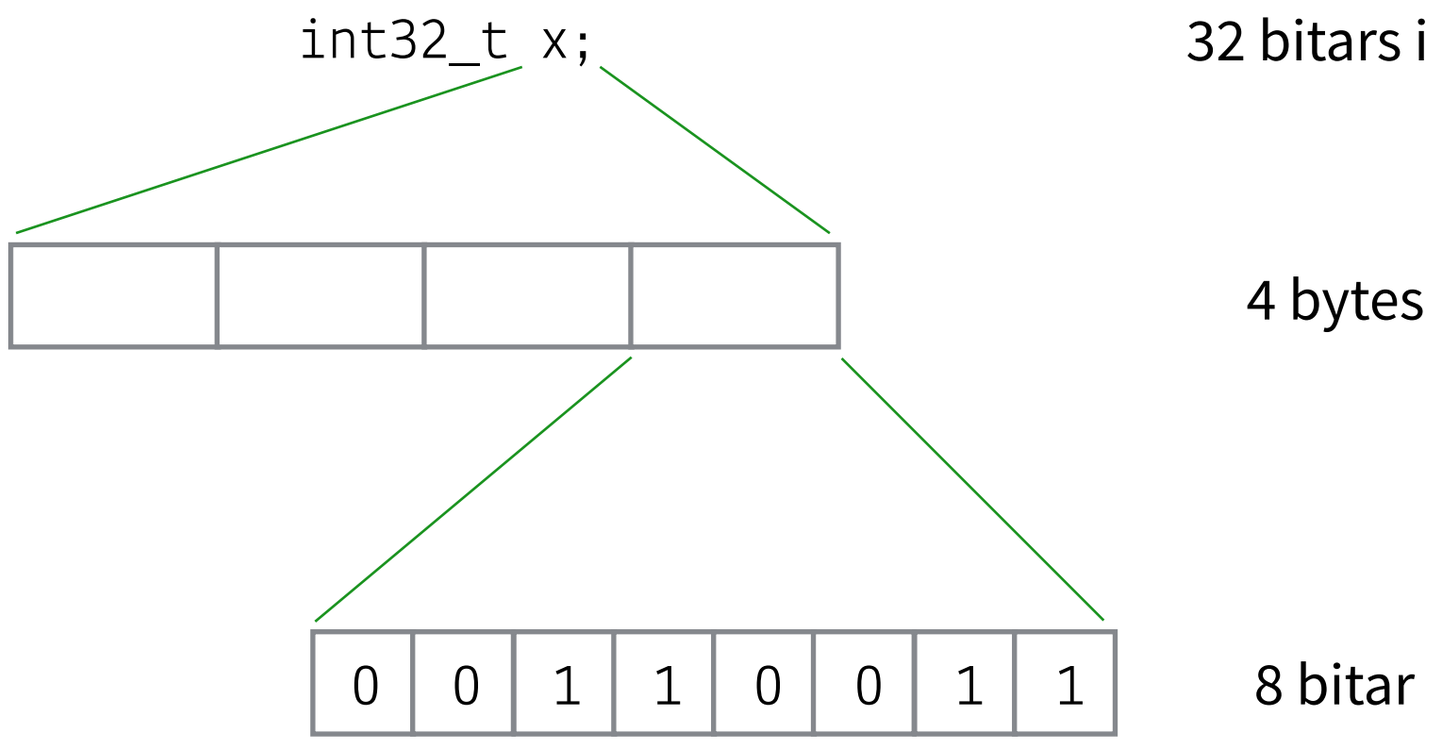

```
x = 51;
```

Bitar




```
x = 51;
```

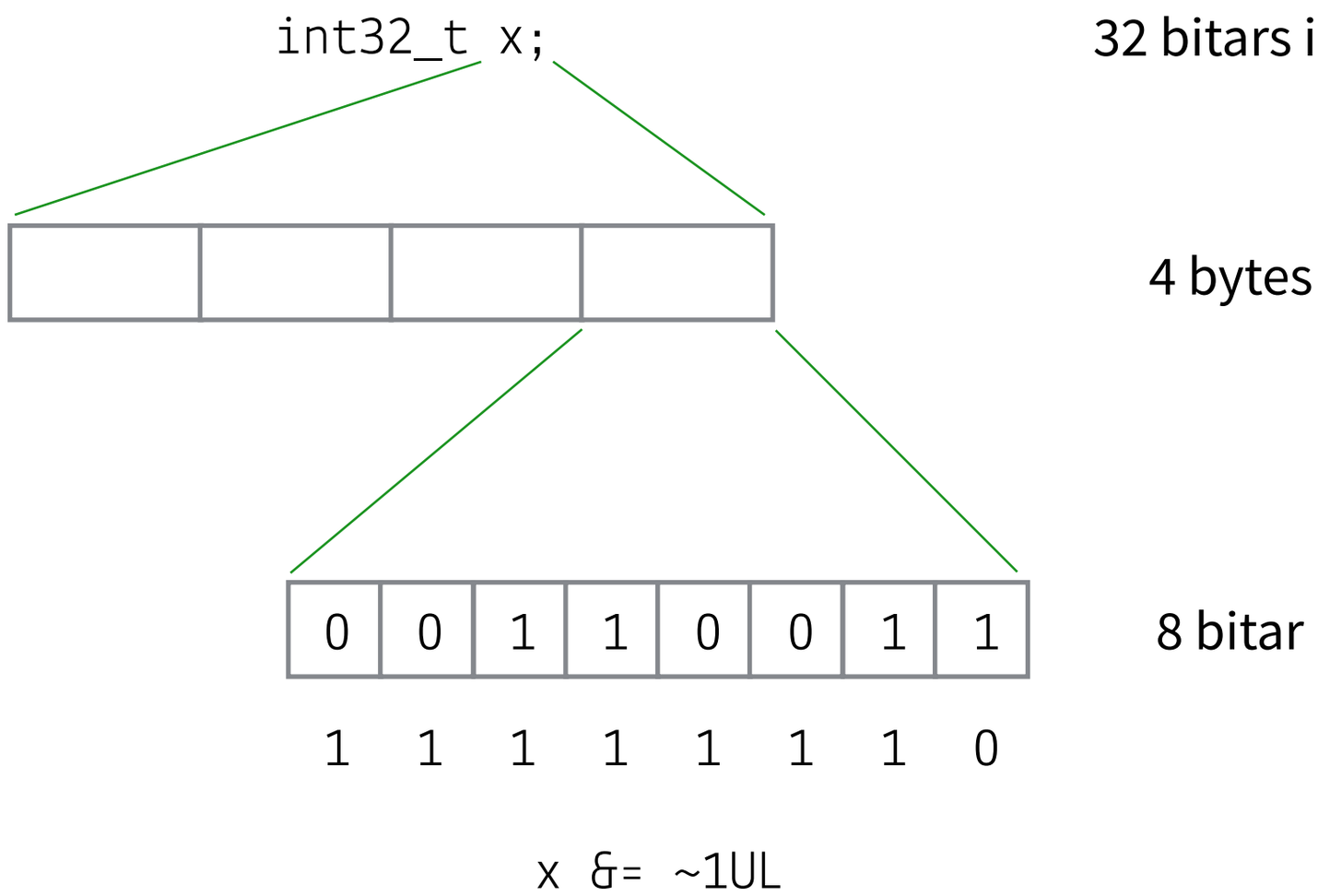
Sätt bit 0 till 0



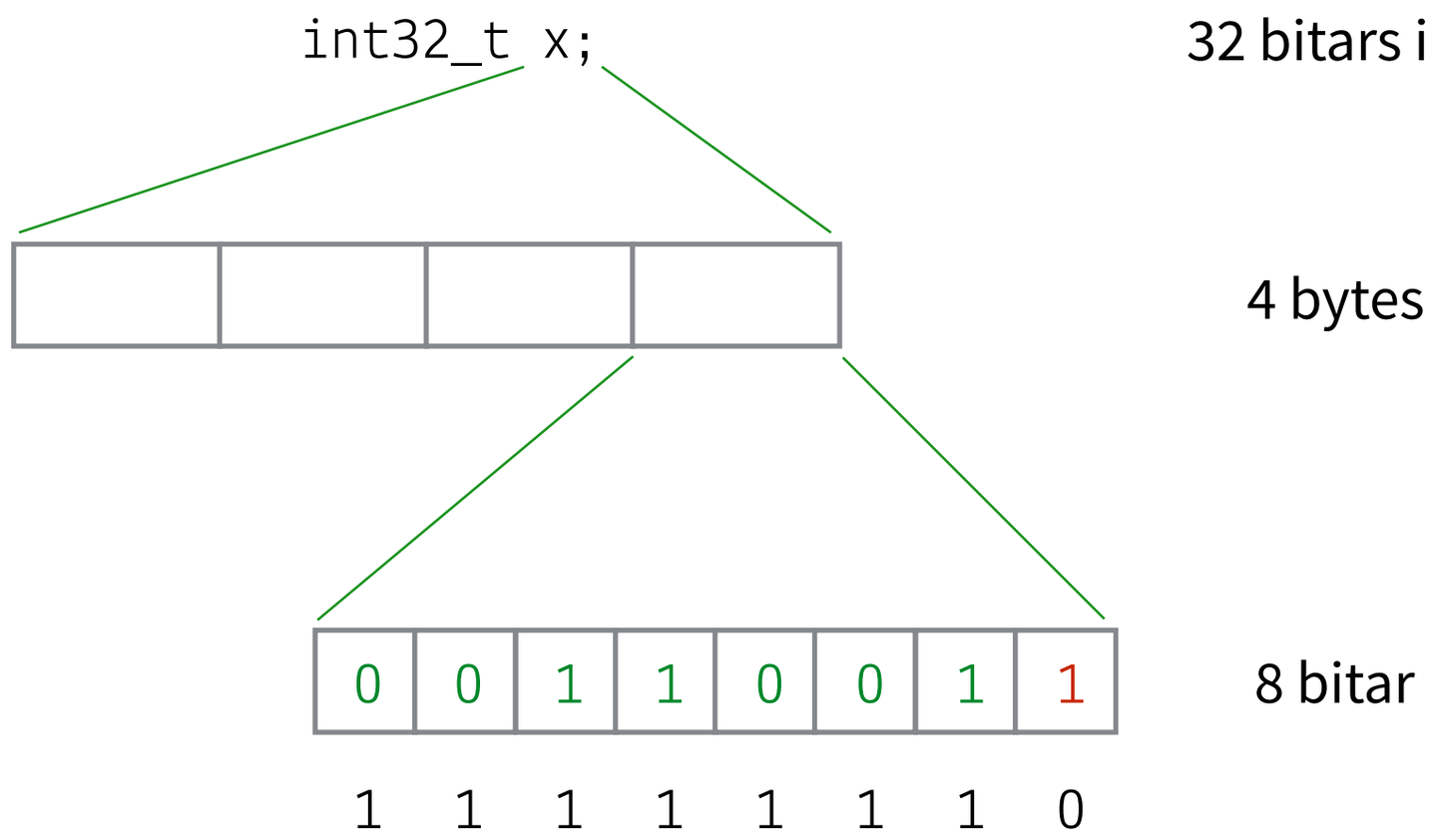

```
x &= ~1UL
```

Note: 1UL is an unsigned long int

Sätt bit 0 till 0

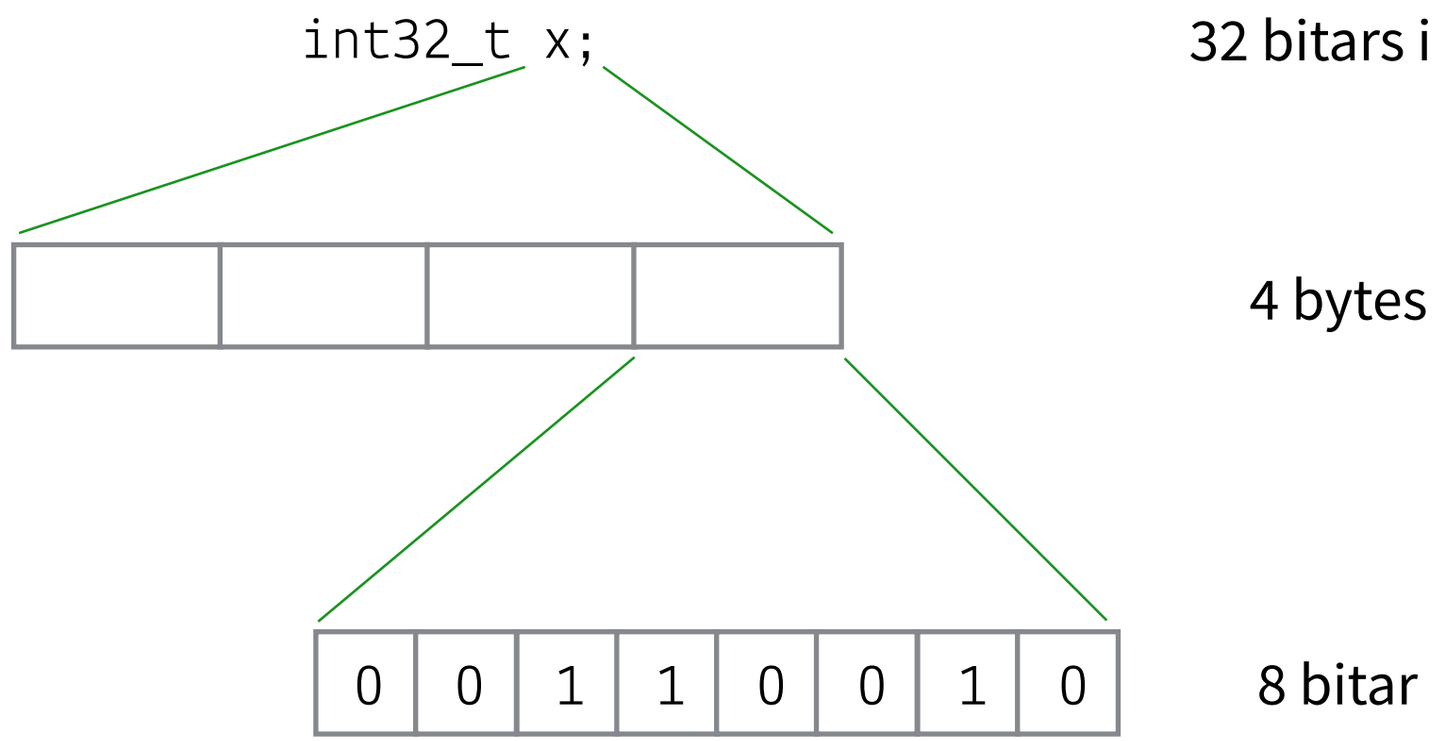


Sätt bit 0 till 0

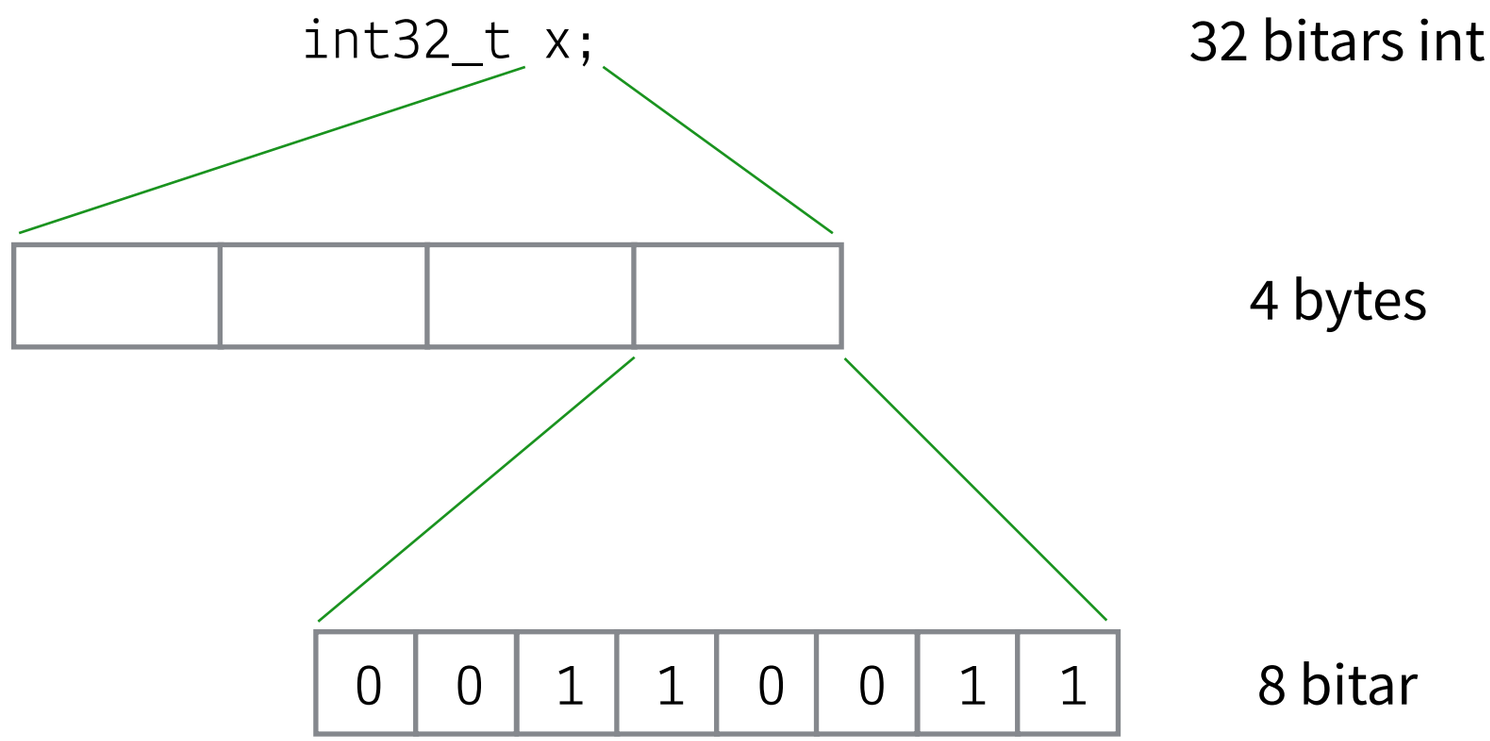


x &= ~1UL

Sätt bit 0 till 0

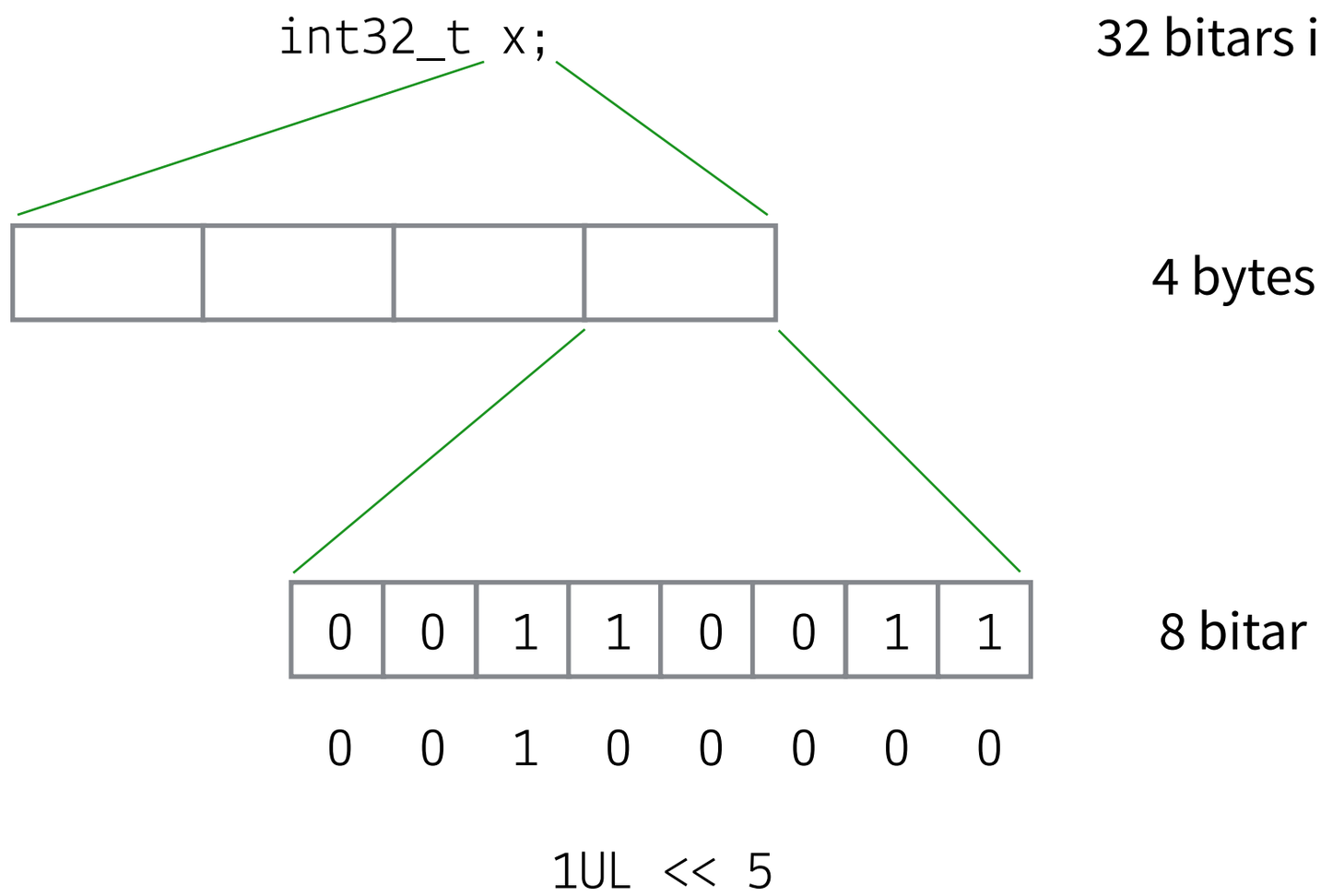


Sätt bit 5 till 0

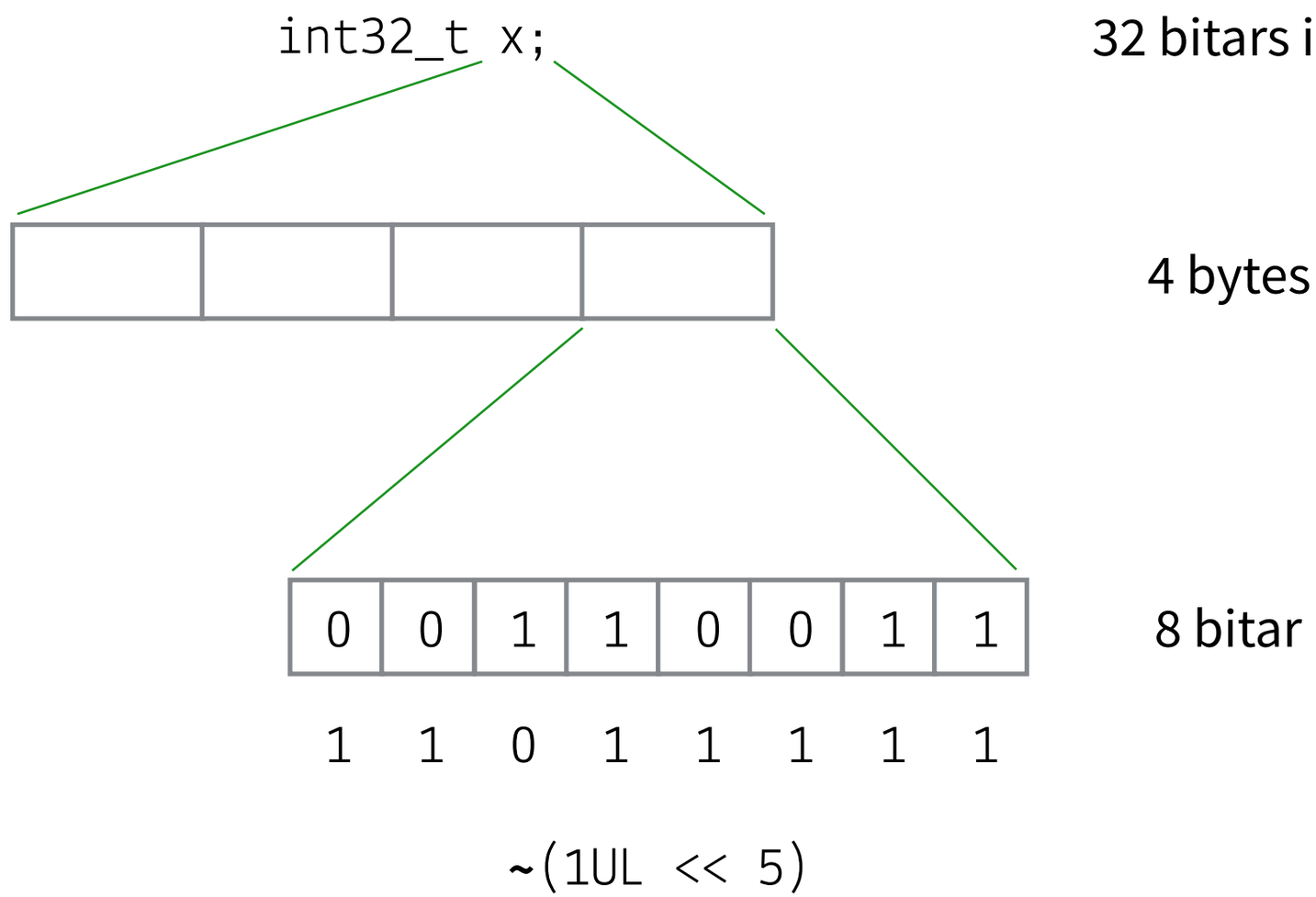



```
x &= ~(1UL << 5)
```

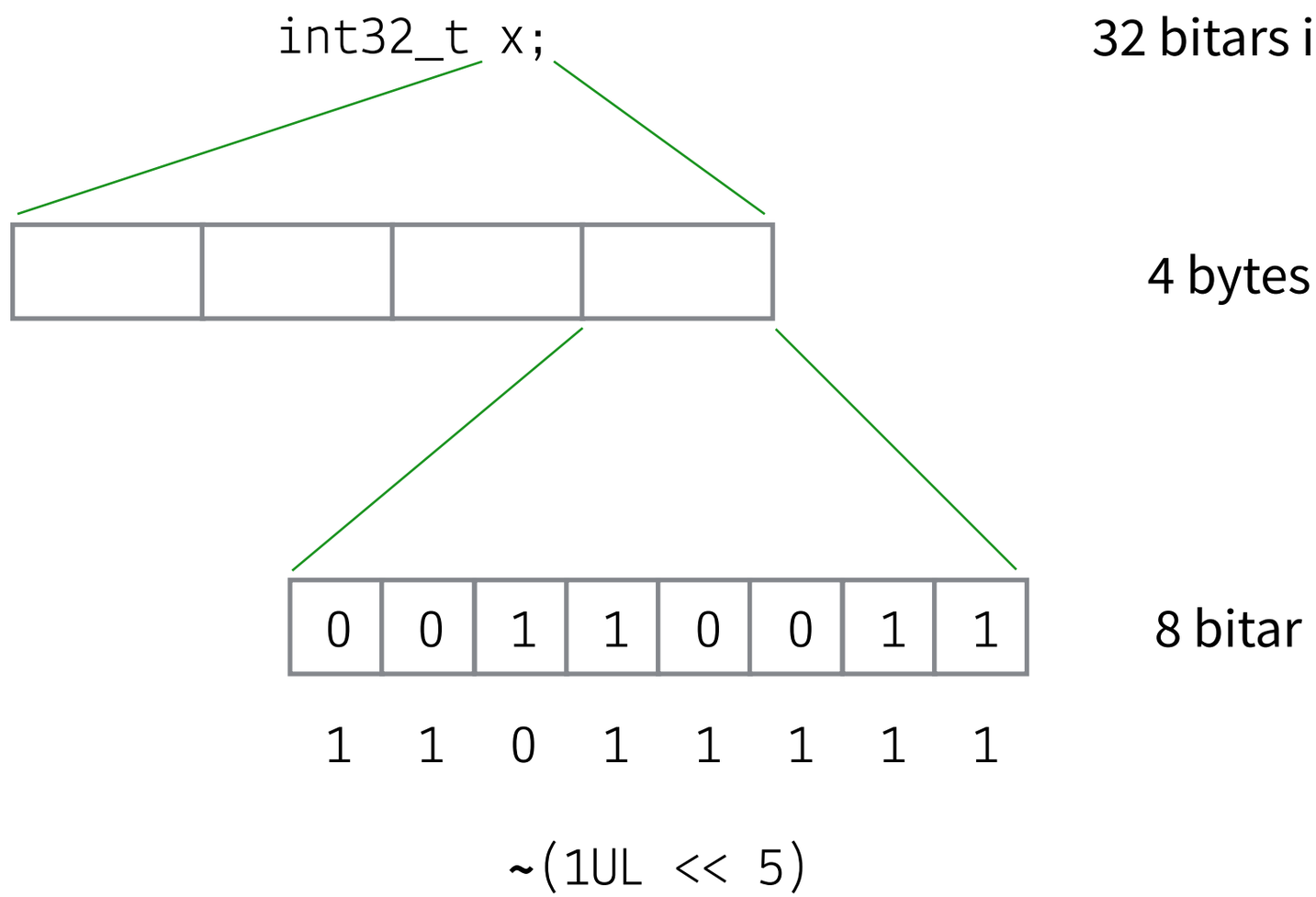
Sätt bit 0 till 0



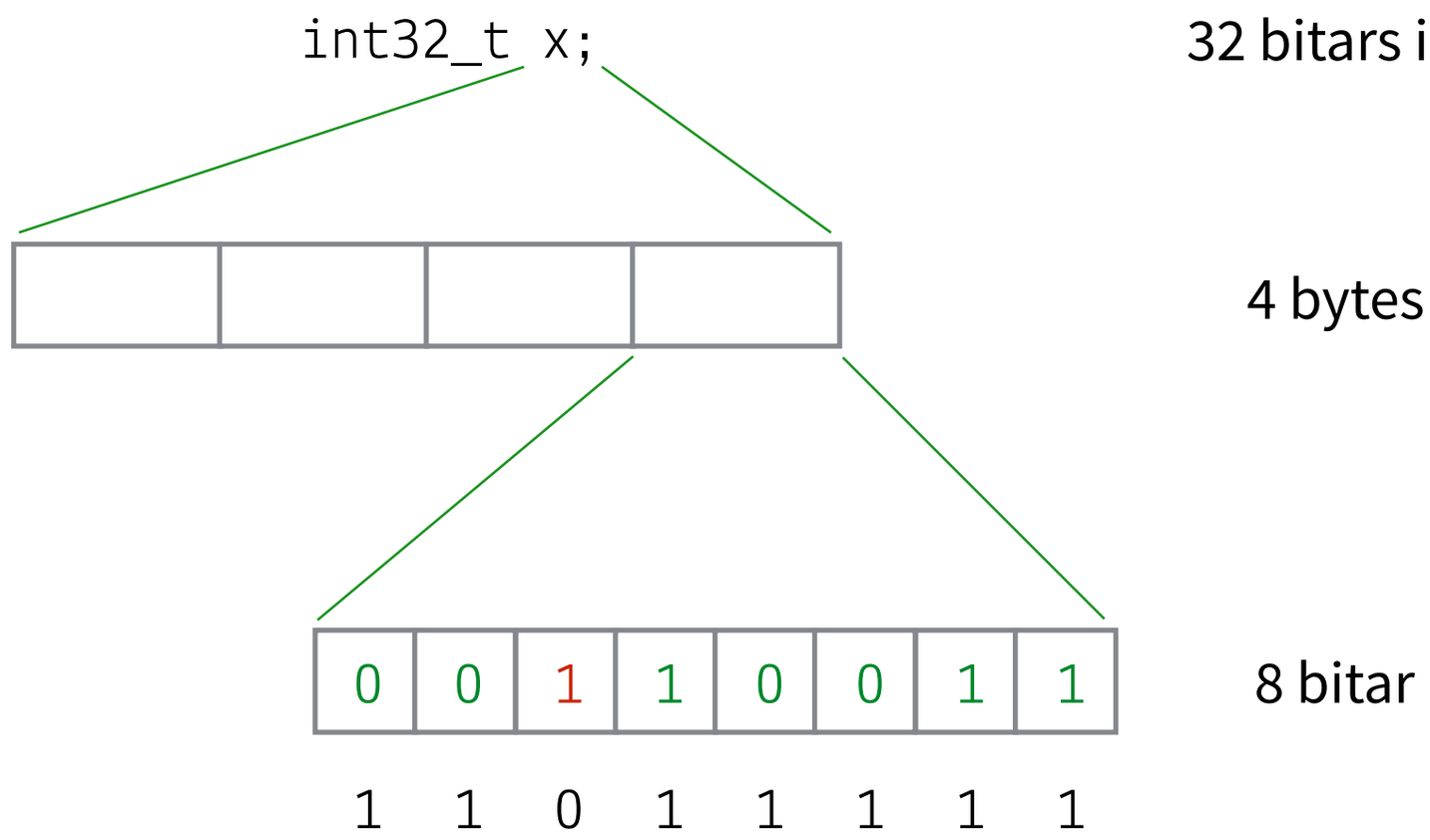
Sätt bit 0 till 0



Sätt bit 0 till 0

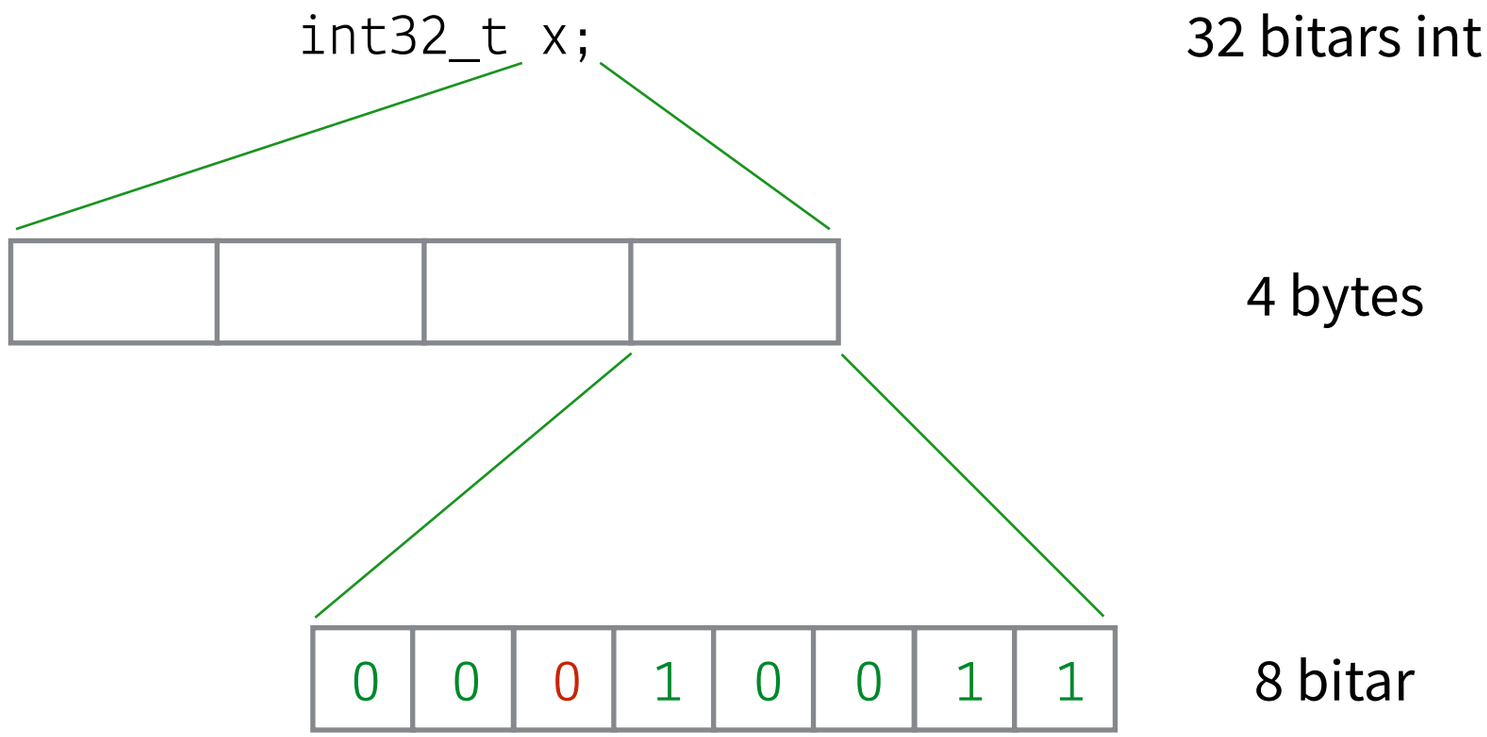


Sätt bit 0 till 0




```
x &= ~(1UL << 5)
```

Set bit 0 to 0



Bitoperatorer i C

Operator	Betydelse
~	Unär, negation
&	och / and
|	eller / or
^	xor
<<	vänsterskift / leftshift
>>	högerskift / rightshift

Bitoperatorer i C

Operator	Bitström
a = 1UL	00000000 00000000 00000000 00000001
b = ~1UL	11111111 11111111 11111111 11111110
a & b	00000000 00000000 00000000 00000000
a | b	11111111 11111111 11111111 11111111
7 ^ 3	0111 ^ 0011 = 0100 = 4
a << 3	00000000 00000000 00000000 00001000
b >> 5	00000111 11111111 11111111 11111111

Applications

- Bitflaggor

```
unsigned long SWITCHED_ON  = 1UL << 15;
unsigned long USES_PADDING = 1UL << 3;
SWITCHED_ON | USES_PADDING
```

- Hardware programming
- Memory-critical applications

  Ex. ”bitset” (komprimeringsfaktor 8) -> A bitset emulates an array of Booleans

  Projektarbetet tidigare år (info om varje objekt…)

bitset.c

```c
typedef struct bitset bitset_t;

#define Byte_index(siz,idx) (siz / idx)
#define Bit_index(siz,idx)  (siz % idx)
#define On(a)    1UL << (a)
#define Off(a) ~(1UL << (a))
#define Index_check(a, b) assert(0 <= (a) && (a) < (b))

struct bitset
{
  size_t size;
  uint8_t bits[]; // flexible member from C99
};

bitset_t *bs_new(size_t siz)
{
  bitset_t *b = calloc(1, siz + sizeof(size_t));
  return b;
}

void bs_free(bitset_t *b)
{
  free(b);
}
```

```c
bool bs_contains(bitset_t *b, size_t v)
{
  Index_check(v, b->size / 8);

  return b->bits[v / 8] & On(v % 8);
}

void bs_set(bitset_t *b, size_t v)
{
  Index_check(v, b->size / 8);

  b->bits[v / 8] |= On(v % 8);
}

void bs_unset(bitset_t *b, size_t v)
{
  Index_check(v, b->size / 8);

  b->bits[v / 8] &= Off(v % 8);
}
```

Interesting Facts About Bitwise Operators

- The left shift and right shift operators should not be used for negative numbers:

  If any of the operands is a negative number, it results in undefined behaviour!!

  For example results of both -1 << 1 and 1 << -1 is undefined.

- If the number is shifted more than the size of integer, the behaviour is undefined.

  For example, 1 << 33 is undefined if integers are stored using 32 bits.

- The bitwise XOR operator is the most useful operator from technical interview perspective!

Fun (Interview!) Problem

For a given array of repeated elements, exactly one element is not repeated. You need to return the non-repeated element.

Input: [1, 2, 5, 4, 6, 8, 9, 2, 1, 4, 5, 8, 9]
Output: 6

Resultat från enkäten om inlupp 1

Får jag redovisa >4 mål per labb?

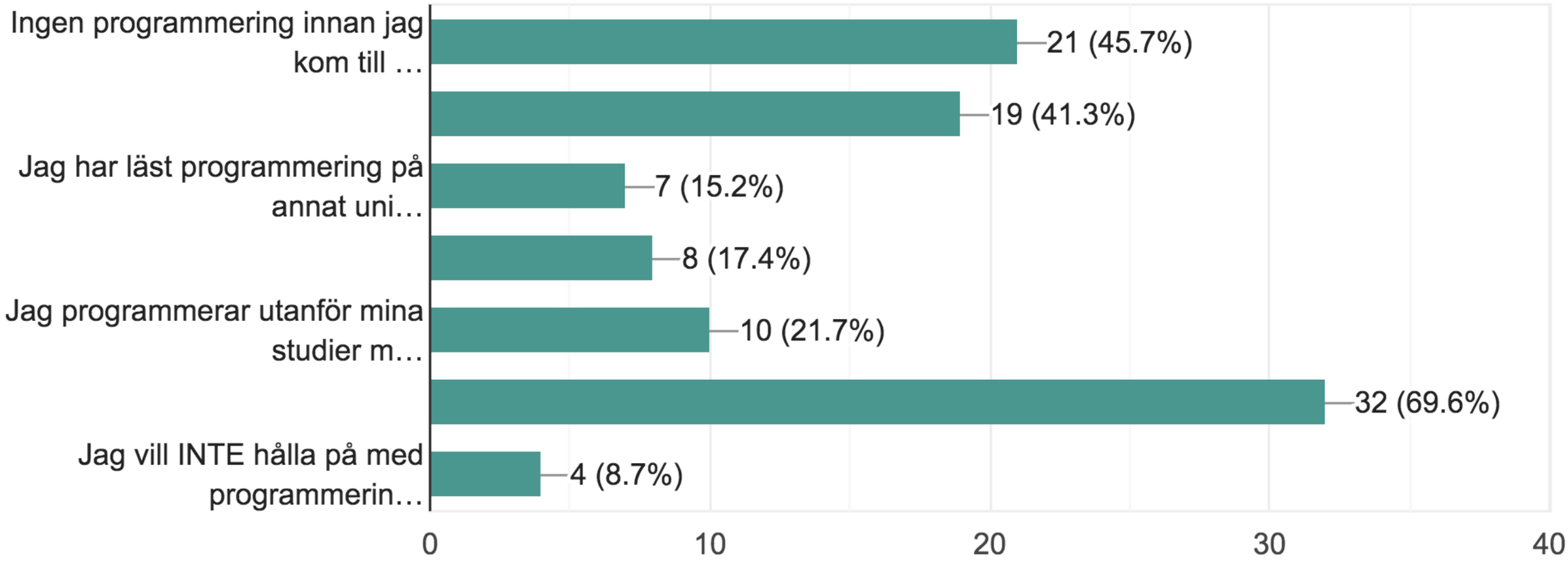
NEJ

What Do You Think?

Last Year's Cloud

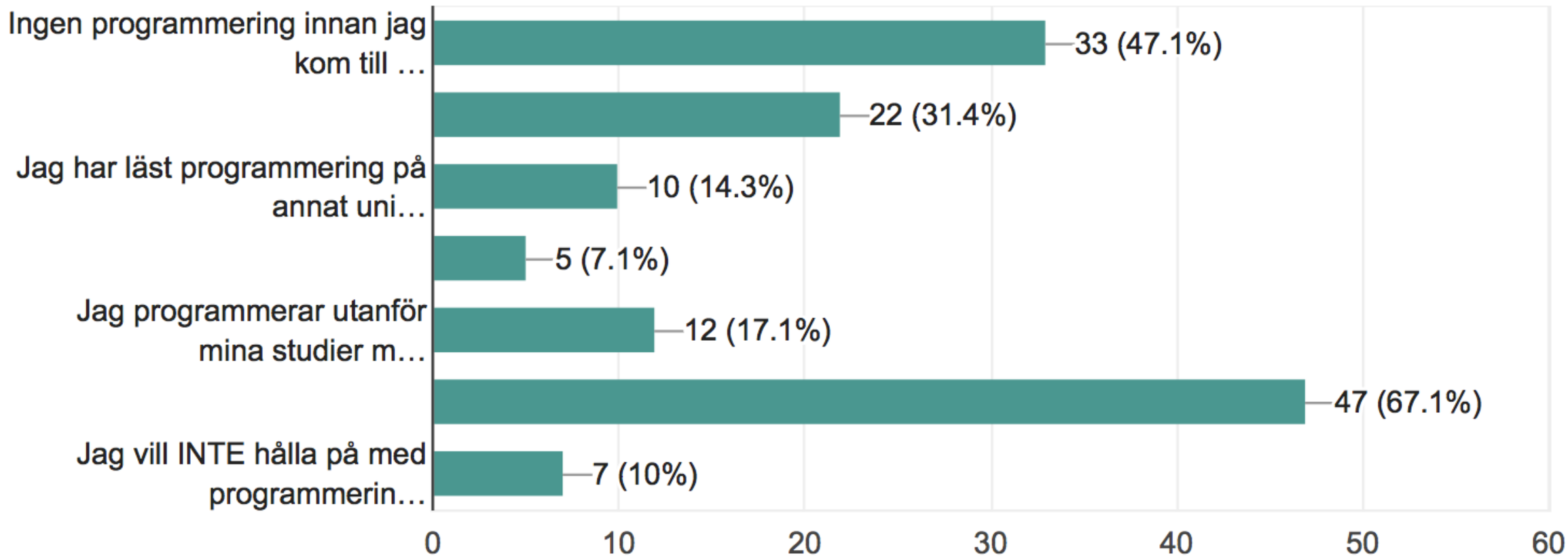
Klicka i allt som stämmer på dig

46 responses



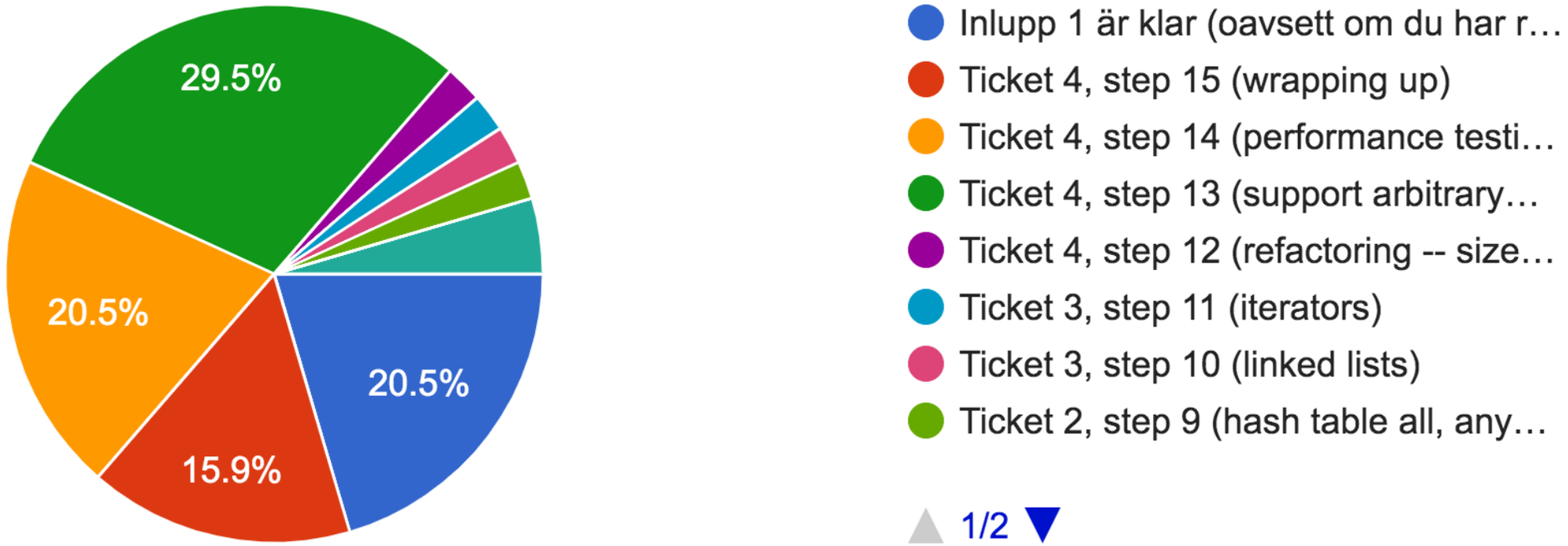
Klicka i allt som stämmer på dig

70 responses



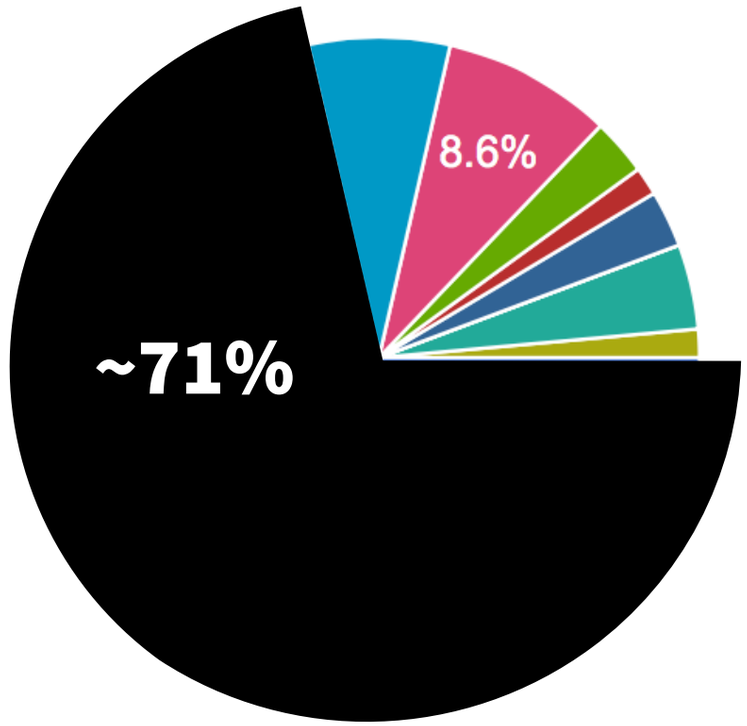
Vilket steg ligger du på i inlupp 1?

44 responses



Vilket steg ligger du på i inlupp 1?

70 responses

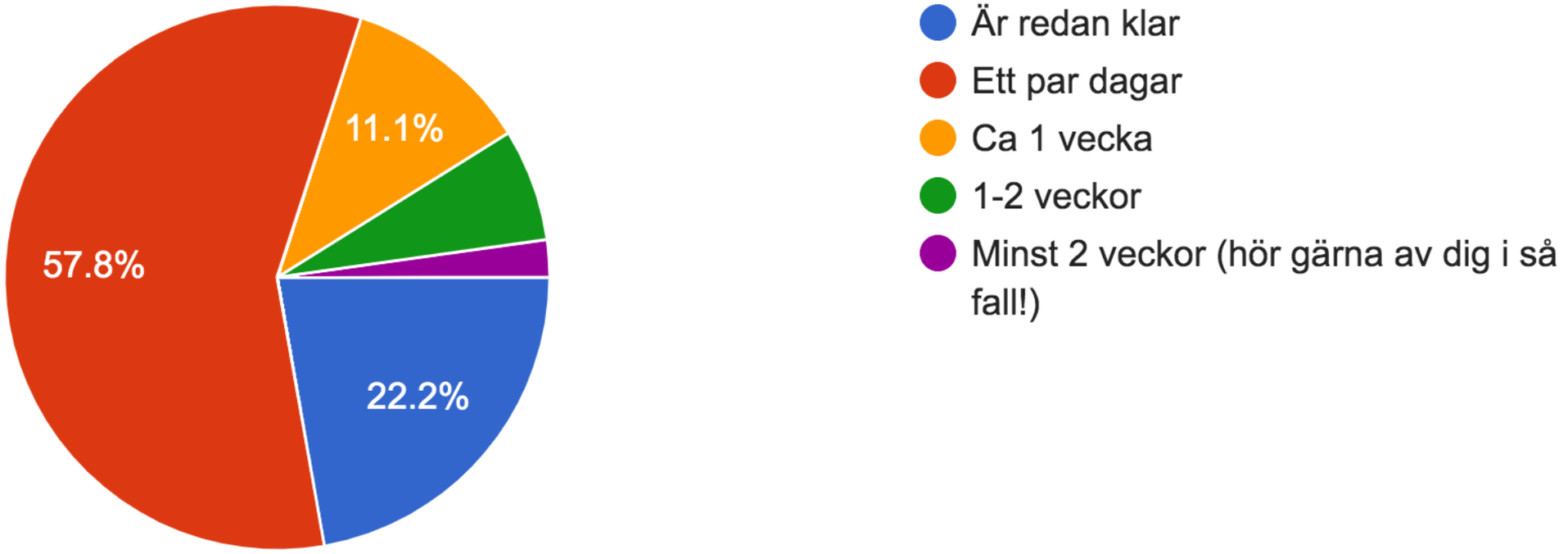


- Inlupp 1 är klar (oavsett om du har…
- Ticket 4, step 15 (wrapping up)
- Ticket 4, step 14 (performance testi…
- Ticket 4, step 13 (support arbitrary…
- Ticket 4, step 12 (refactoring -- size…
- Ticket 3, step 11 (iterators)
- Ticket 3, step 10 (linked lists)
- Ticket 2, step 9 (hash table all, any…

▲ 1/2 ▼

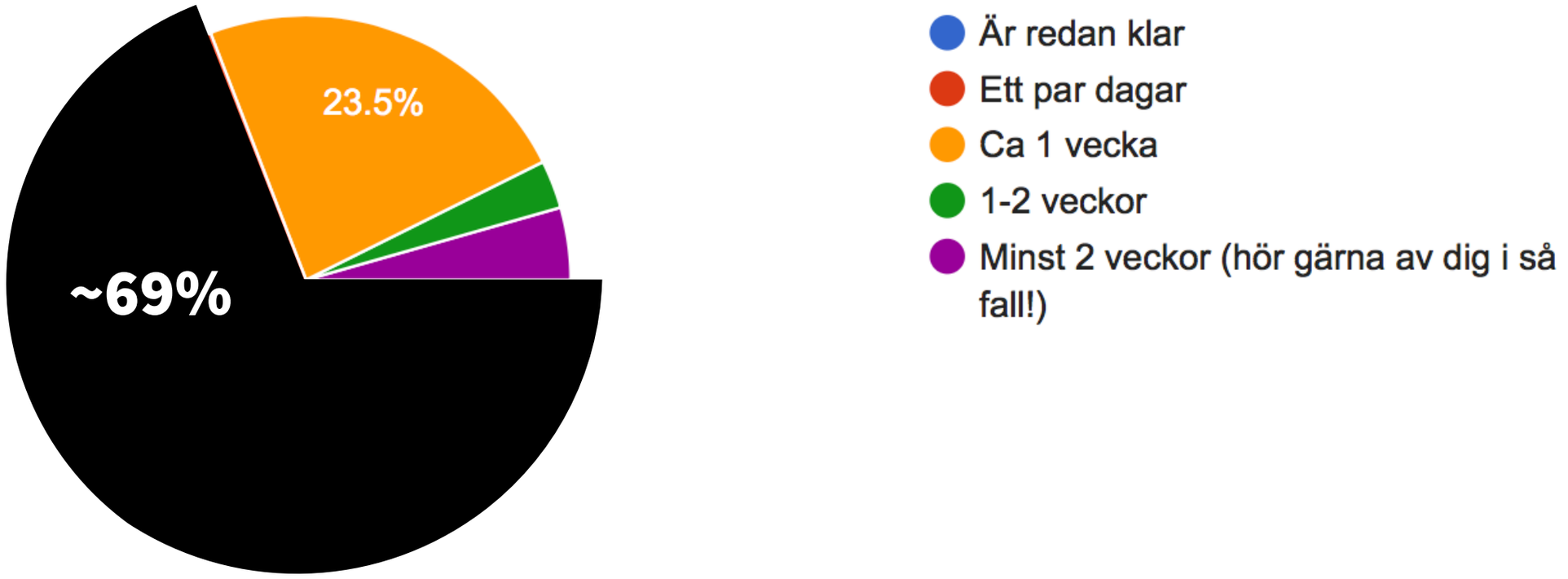
När bedömer du att du skulle kunna vara klar med inlupp 1?

45 responses

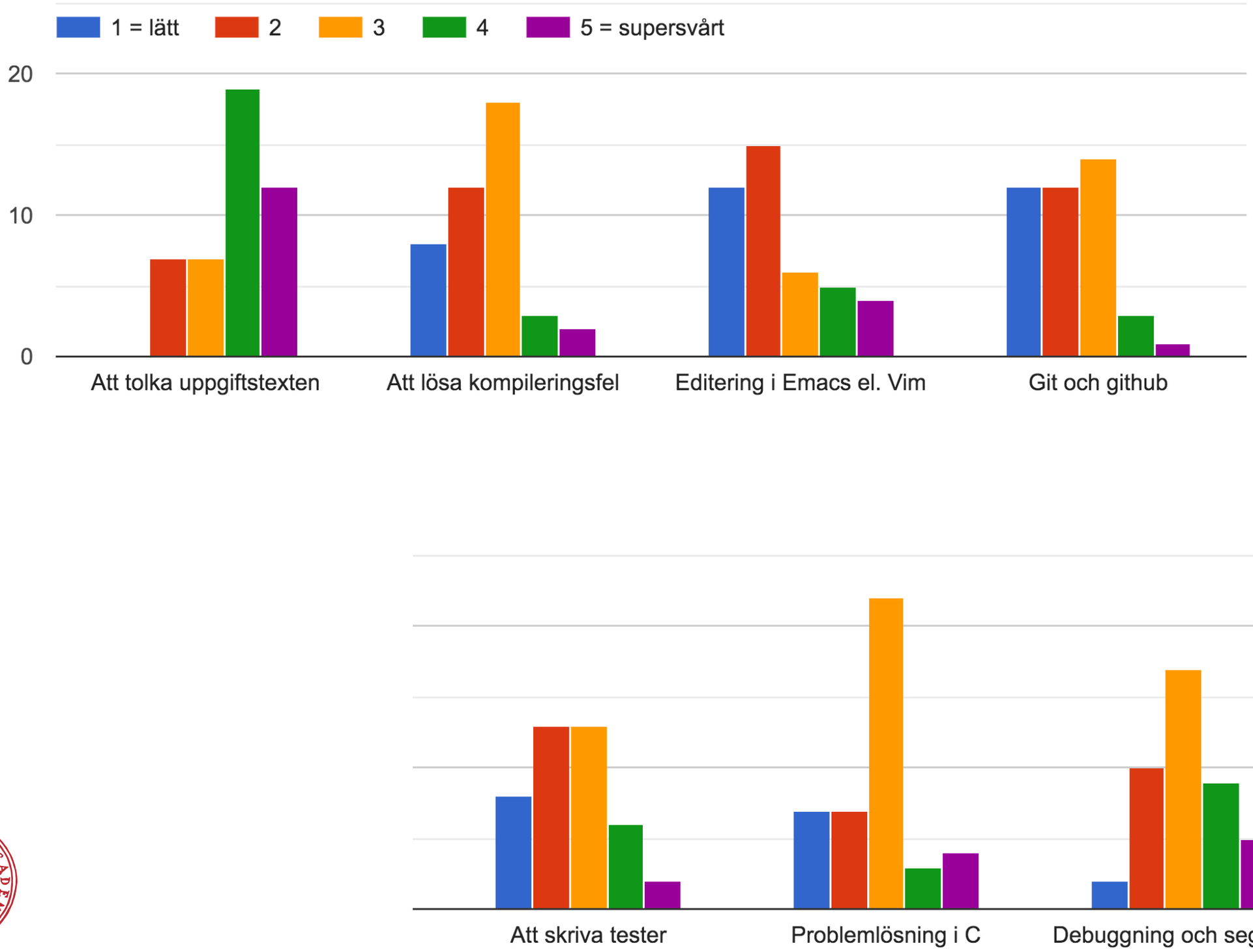


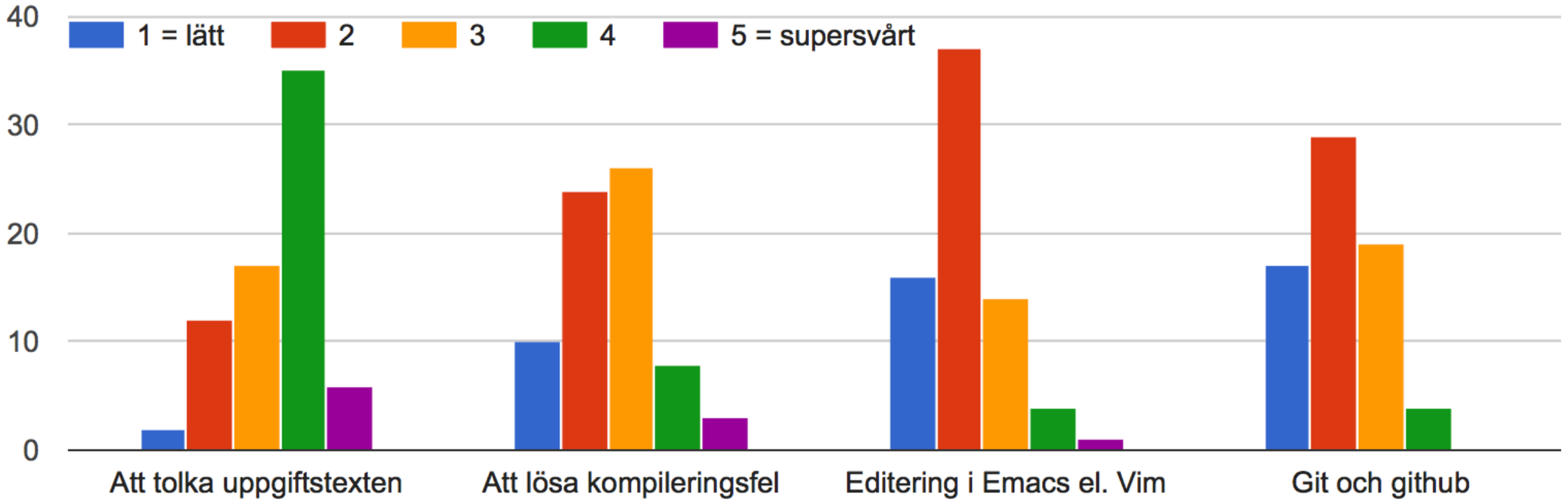
När bedömer du att du skulle kunna vara klar med inlupp 1?

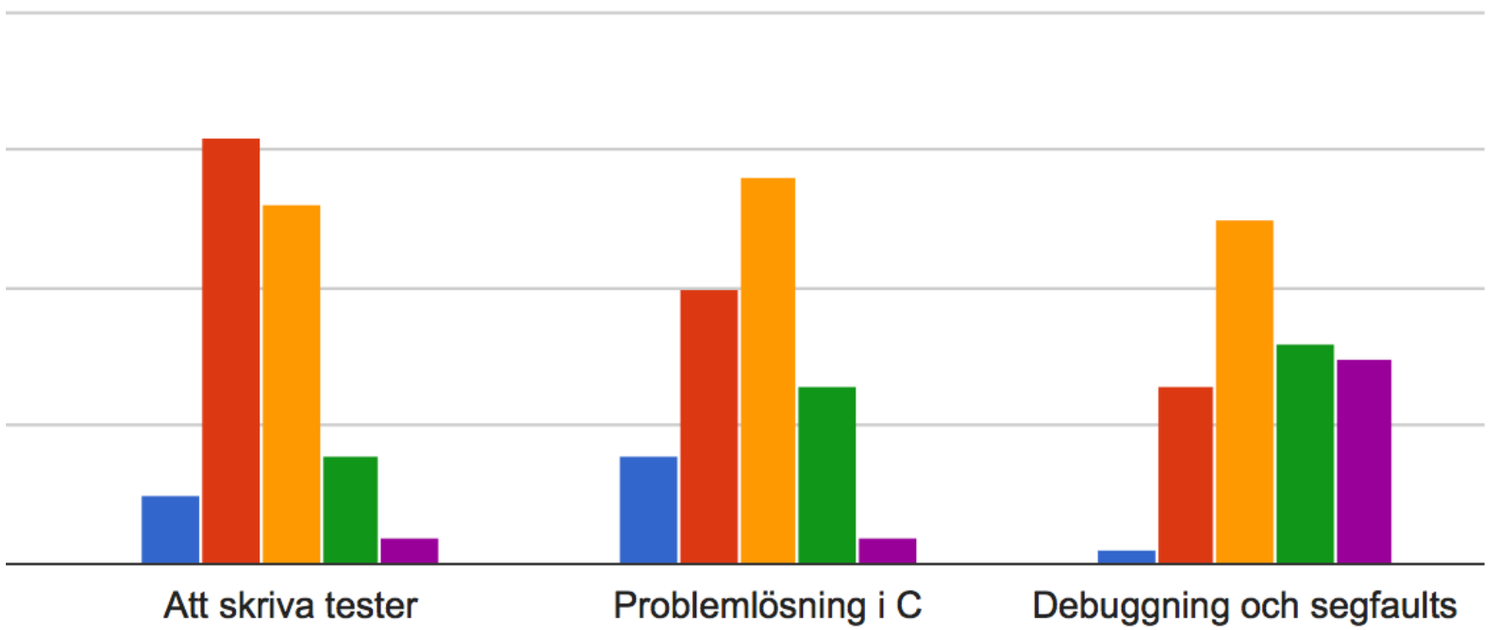
68 responses



Ranka olika aktiviteter under arbetets gång




40
30
20
10
0
1 = lätt
2
3
4
5 = supersvårt
Att tolka uppgiftstexten
Att lösa kompileringsfel
Editering i Emacs el. Vim
Git och github


Att skriva tester
Problemlösning i C
Debuggning och segfaults