

# Föreläsning 11

---

Tobias Wrigstad

*Bra och läsbar kod  
Defensiv programmering*



# Korrekthet och prestanda är inte allt

---

- Underhållsbarhet
- Portabilitet
- Läsbarhet
- Komplexitet
- testbarhet
- ...

Alla dessa (7) är av lika vikt!

*Det handlar alltså inte  
bara om att ”få det att  
funka”...*



# Tips för att skriva bra (och läsbar) kod

---



# 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

# Gruppera relaterade satser

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

    if (Value(n) > value) return node_find(Left(n), value);
    if (Value(n) < value) return node_find(Right(n), value); ← Villkor för att gå vidare i trädet

    assert(false);
    return NULL; ← 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);
    strcpy(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);
    strcpy(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;
            }
            strcpy(shelf, tmp, strlen(tmp)+1);
            puts(input);
            break;
        }

        while(true)
        {
            puts("Lagertylla: ");
            readline(input, 256, stdin);
            tmp = shelfPlacement(input);
            if (tmp == NULL)
            {
                printf("Fel försök igen.\n");
                continue;
            }
            strcpy(location, tmp, strlen(tmp)+1);
            puts(input);
            break;
        }

        strcat(shelf, location);
        prod->location = malloc(strlen(shelf)+1);
        strcpy(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

- Tydliga grupper
- Läcker inte minne
- Bra namngivning
- Korrekt

1 funktion – 86 rader

# Undvik 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);
    strcpy(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);
    strcpy(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;
            }
            strcpy(shelf, tmp, strlen(tmp)+1);
            puts(input);
            break;
        }
    }
}
```

*Jobbigt att ta sig igenom*

# 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);
    strcpy(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);
    strcpy(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;
            }
            strcpy(shelf, tmp, strlen(tmp)+1);
            puts(input);
            break;
        }
    }
}
```

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;
            }
            strcpy(shelf, tmp, strlen(tmp)+1);
        }
    }
}
```

# 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);
    strcpy(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);
    strcpy(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;
            }
            strcpy(shelf, tmp, strlen(tmp)+1);
        }
    }
}
```

Slaskvariabler

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 2016*

# Undvik 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);
    strcpy(prod->name, input);
    prod->name[strlen(input)] = '\0';
    puts(prod->name);

    printf("Beskrivning: ");
    readline(input, 256, stdin);
    prod->description = malloc(strlen(input)+1);
    strcpy(prod->description, 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;
            }
            strcpy(shelf, tmp, strlen(tmp)+1);
            puts(input);
            break;
        }

        while(true)
        {
            while(true)
            {
                puts("Lagerhylla: ");
                readline(input, 256, stdin);
                tmp = shelfPlacement(input);
                if (tmp == NULL)
                {
                    printf("Fel försök igen.\n");
                    continue;
                }
                strcpy(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;
                }
                strcpy(shelf, tmp, strlen(tmp)+1);
                puts(input);
                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);
    strcpy(prod->name, input);
    prod->name[strlen(input)] = '\0';
    puts(prod->name);

    printf("Beskrivning: ");
    readline(input, 256, stdin);
    prod->description = malloc(strlen(input)+1);
    strcpy(prod->description, 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;
            }
            strcpy(shelf, tmp, strlen(tmp)+1);
            puts(input);
            break;
        }

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

    return prod;
}
```

readline  
strcpy  
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);
    strcpy(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);
    strcpy(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;
            }
            strcpy(shelf, tmp, strlen(tmp)+1);
            puts(input);
            break;
        }

        while(true)
        {
            puts("Lagertylla: ");
            readline(input, 256, stdin);
            tmp = shelfPlacement(input);
            if (tmp == NULL)
            {
                printf("Fel försök igen.\n");
                continue;
            }
            strcpy(location, tmp, strlen(tmp)+1);
            puts(input);
            break;
        }

        strcat(shelf, location);
        prod->location = malloc(strlen(shelf)+1);
        strcpy(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 överensstämmelse 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” (st̄rbk)
- 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

T.ex. skillnad i namngivning mellan Makron och funk\_tioner

# Undvik detta när du döper variabler

---

- Missledande eller tvetydiga namn
- Namn med liknande innehörd
- 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

# 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();
```

# Vilken är bäst?

---

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

```
return something;
```

# 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 (styckebröt!)
- 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)

# 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]

---

```
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]

---

```
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]

---

```
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

---

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

int g; // global

int main(void)
{
    g = 42;

    foo();

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

    return 0;
}
```

```
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

- 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

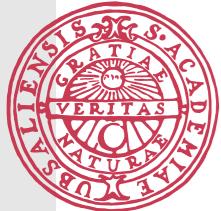
# Kommentarer ≠ Koddokumentation

---

- 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”
- 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 väntigt 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

**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
- 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 ”försinner”)

- 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?

```
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
- $length \leq 0$
- $values == \text{NULL}$

# 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**
- $values == \text{NULL}$

Bra användning av assert!

Dokumenterar ett viktigt villkor i average.

# 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`

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);
    ...
}
```

# 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 indata inte är giltig?

---

## 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 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 indata inte är giltig?

---

## Ta ett angränsande giltigt värde

Ersätt en negativ sträng längd 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 indata inte är giltig?

---

## 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

**Extern**

*Utgå från att  
detta data är korrupt  
eller opålitligt*

**Gräns**

*Ansvarar för att  
"säkra upp"  
passerande data*

**Internt**

*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

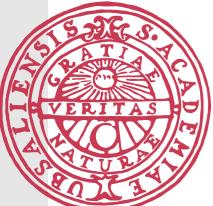
---

- 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 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 uppmärksamas under utveckling?

*Från Steve McConnell's utmärkta ”Code Complete”*

# Skydda dig mot dig själv!

(Bonusmaterial för den hugade)

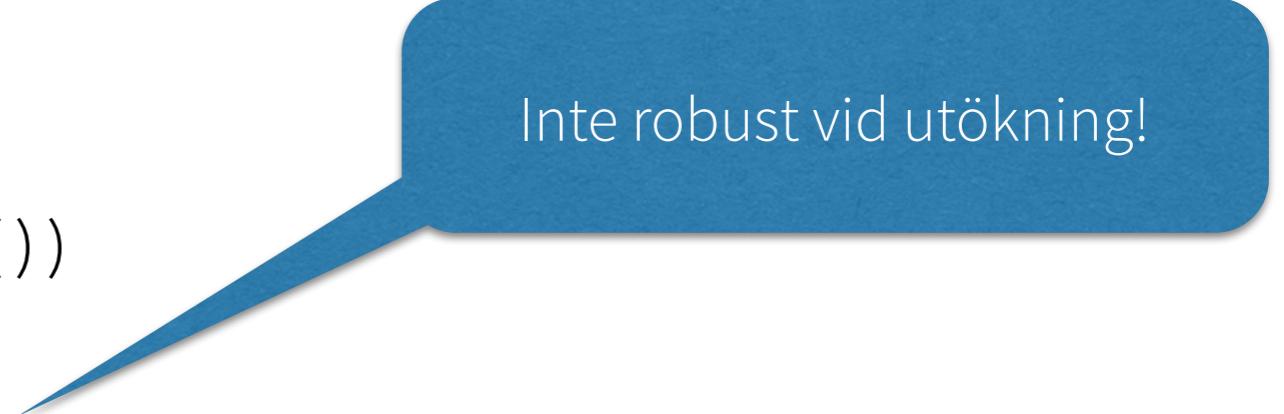


# Skydda dig mot dig själv

---

Vad kan gå fel i detta program?

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



Inte robust vid utökning!

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

# 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

---

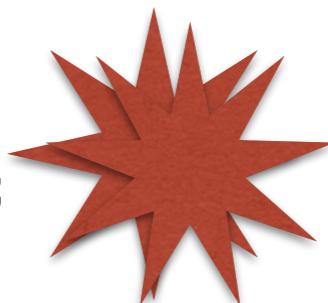
Vad kan gå fel i detta program?

```
#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);
```

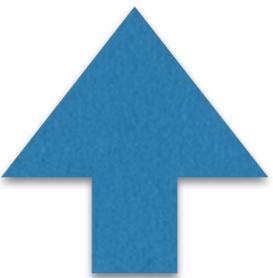


# Skydda dig mot dig själv

---

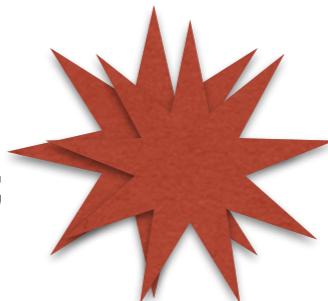
*Vad kan gå fel i detta program?*

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



*expanderas till*

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



# Skydda dig mot dig själv

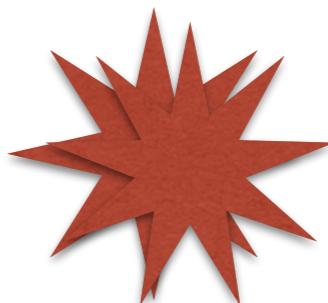
Vettigt  
användande av  
block!

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;
};

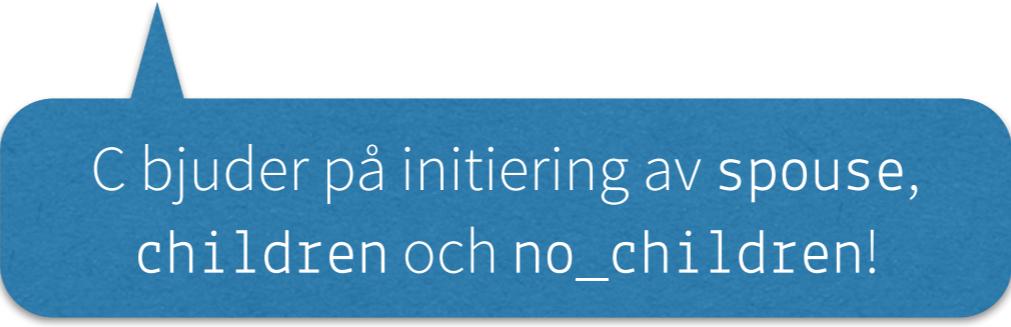
struct person p = (struct person) { .name = "Peter", .age = 32 };
```

# 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.)*

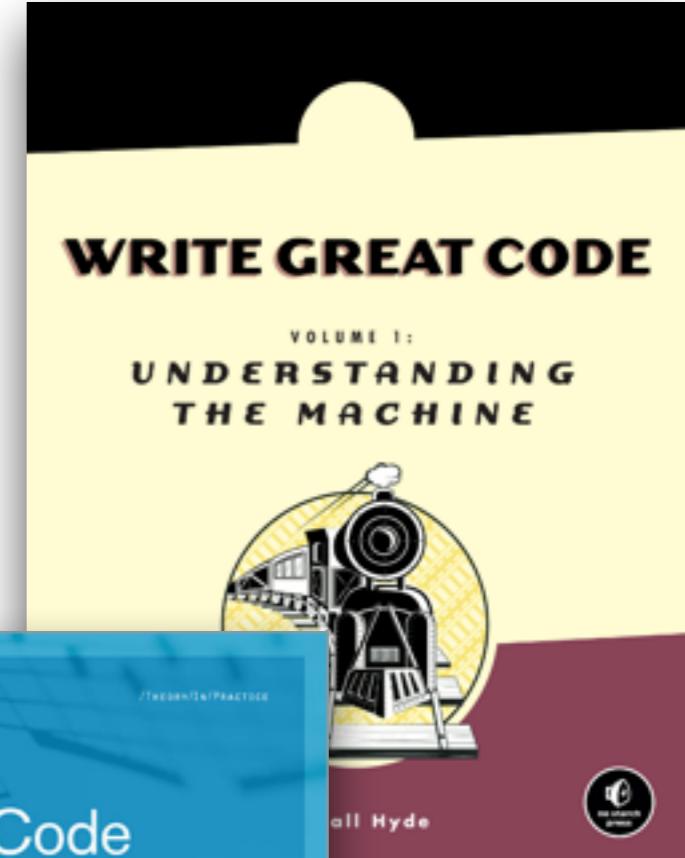
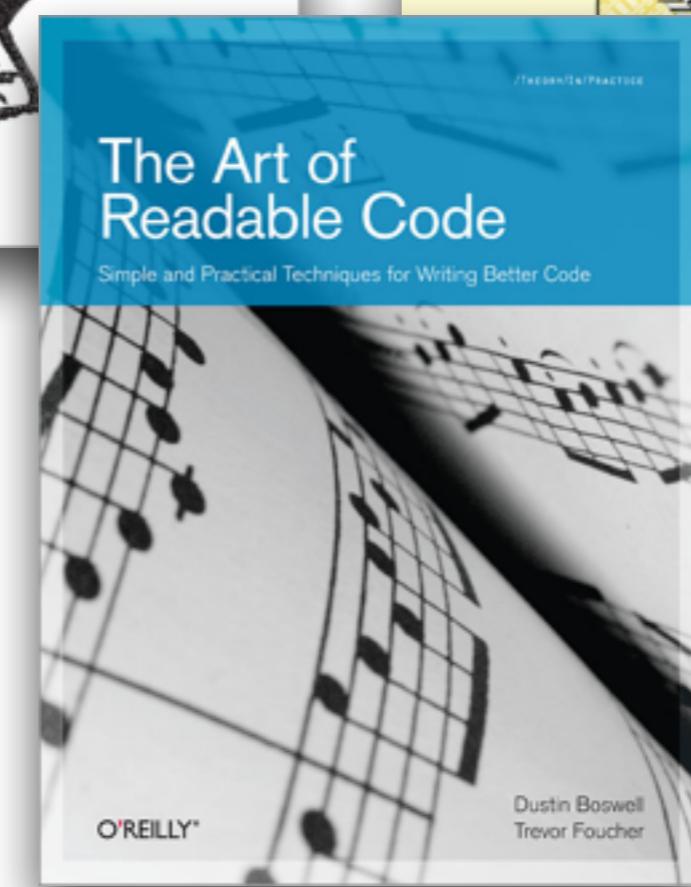
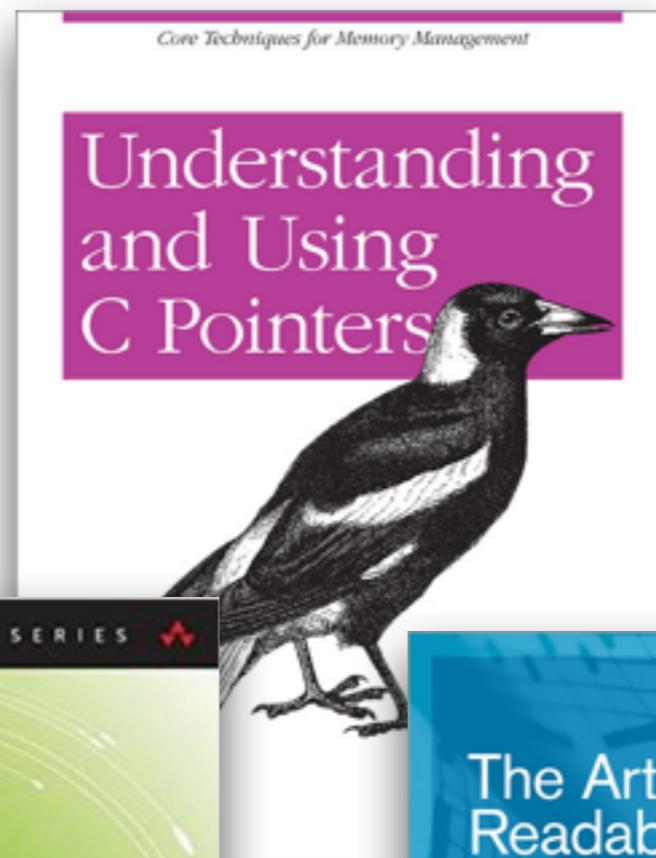
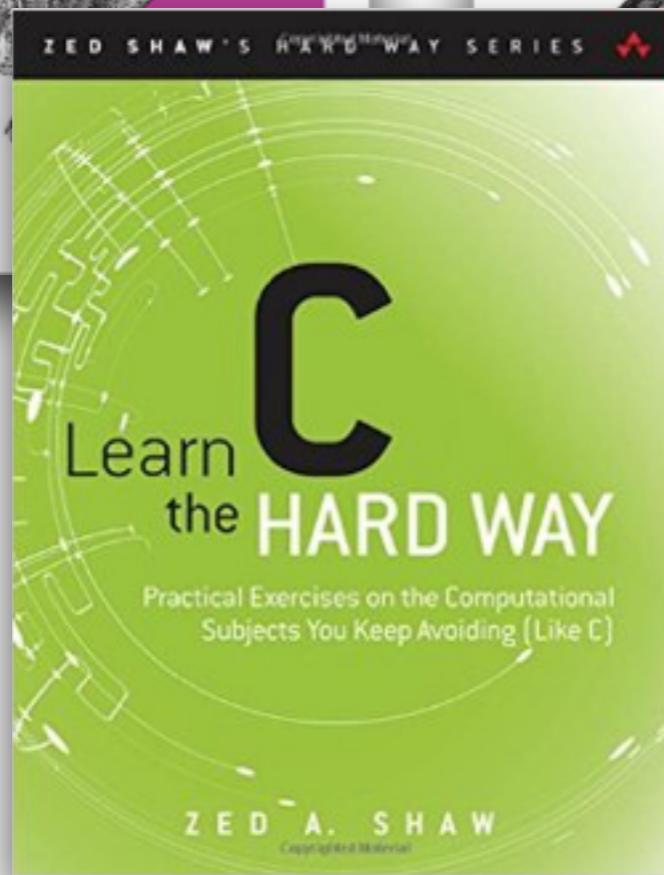
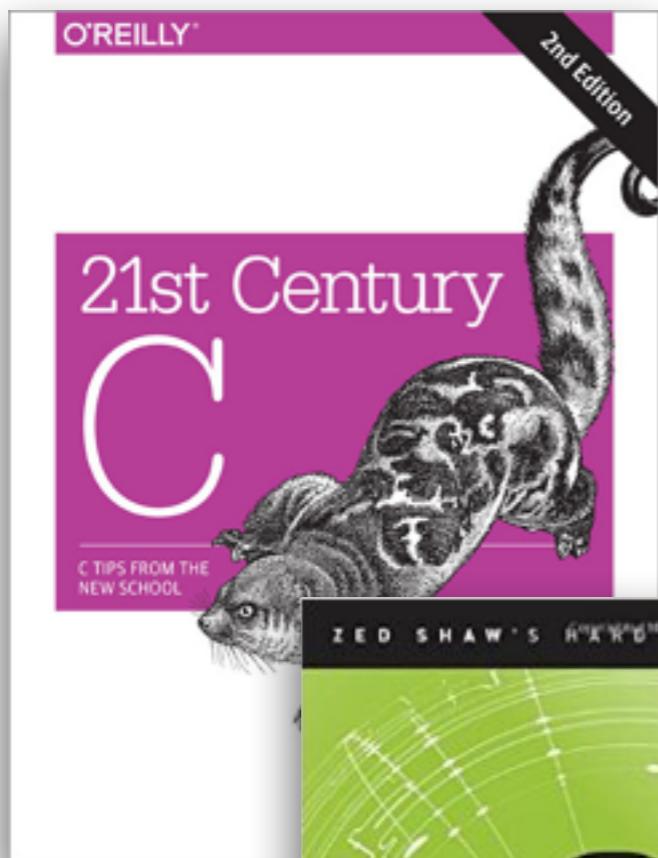
```
#define Person(__VARGS__) \
    ((struct person) { .name="Fred", __VARGS__ })\

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

---

Tobias Wrigstad

*Bitmanipulering  
Preprocessorn*



# Preprocessor



# CPP – The C PreProcessor

- Textersättning som sker innan kompilatorn
- Vi har sett några exempel tidigare:

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



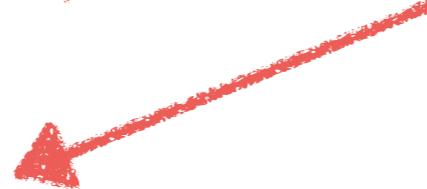
Inkludering av annan fil

```
#ifndef symbol
#define symbol
```



If-sats som körs vid kompileringen

```
...
#endif
```



```
#define INT_MAX 2147483647
```



Definition av konstant (eg. makro)

- CPP – körs innan kompilatorn automatiskt (körs sällan eller aldrig enskilt)

# Preprocessor”operatorer”

---

- Makron som spänner över flera rader \
  - Göra om kod till strängar #
  - Konkatenera uttryck ##
  - Man kan även stänga av eller sätta flaggor vid kompilering
- 
- Otroligt primitivt

Man kan göra **fantastiska** saker med dem

T.ex. definiera hela datastrukturer, e.g. Define\_List(int)

```
#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);
```



# Preprocessormakron

---

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

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

- För längre makron, använd följande form

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

# Robusta preprocessormakron?

---

- Hitta felet!

```
#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

---

- Textuell ersättning, fungerar inte ”som 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?

---

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

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

*// Illustrerar 5-radersmakro*

- 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
```

- Detta används ofta också för andra villkor i kod, t.ex. plattform, OS, etc.

```
#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



```
#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



```
#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;
}
```

```
#include "massert.h"

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

test.c

# Bitmanipulering



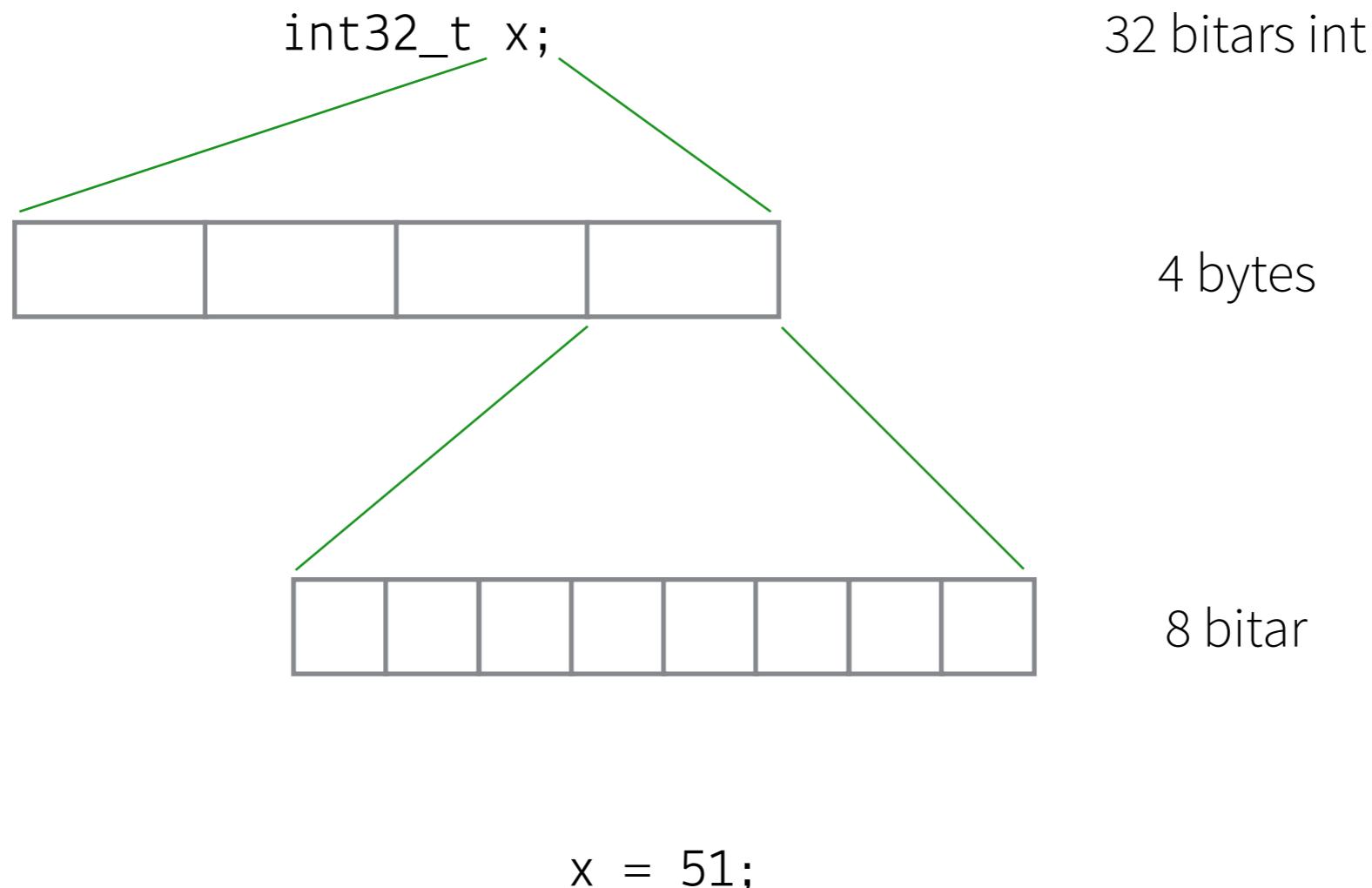
# Bitmanipulering

---

- Kommandon för att manipulera enskilda bitar
  - Vi har sett att C ser på minnet som en array av chars (i regel bytes)
  - Nu skall vi se att det går att ser på dem som arrayer av enskilda bitar
- 
- *Notera att all kod som manipulerar enskilda bitar är svår att portera!*

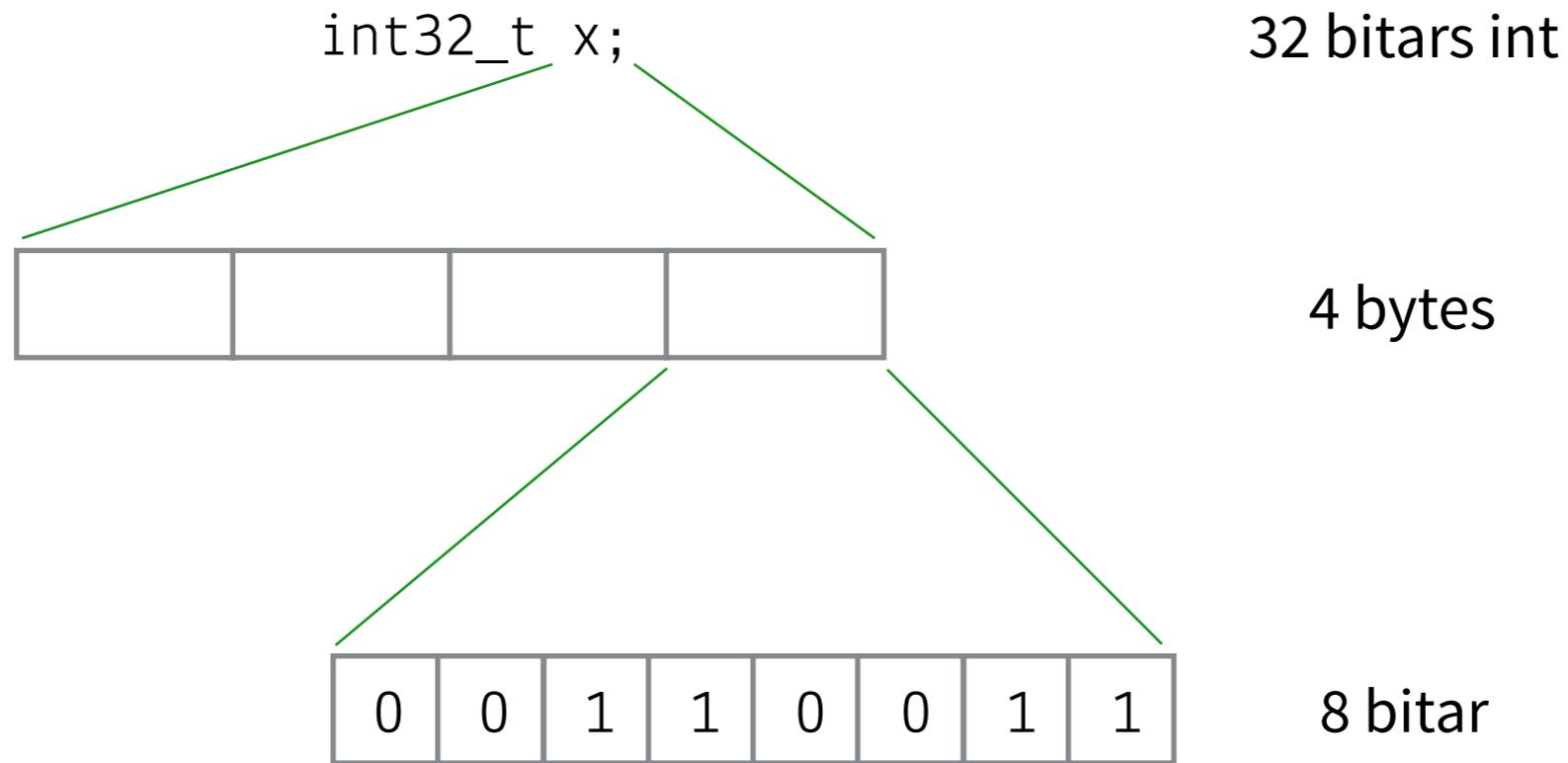
# Bitar

---



# Bitar

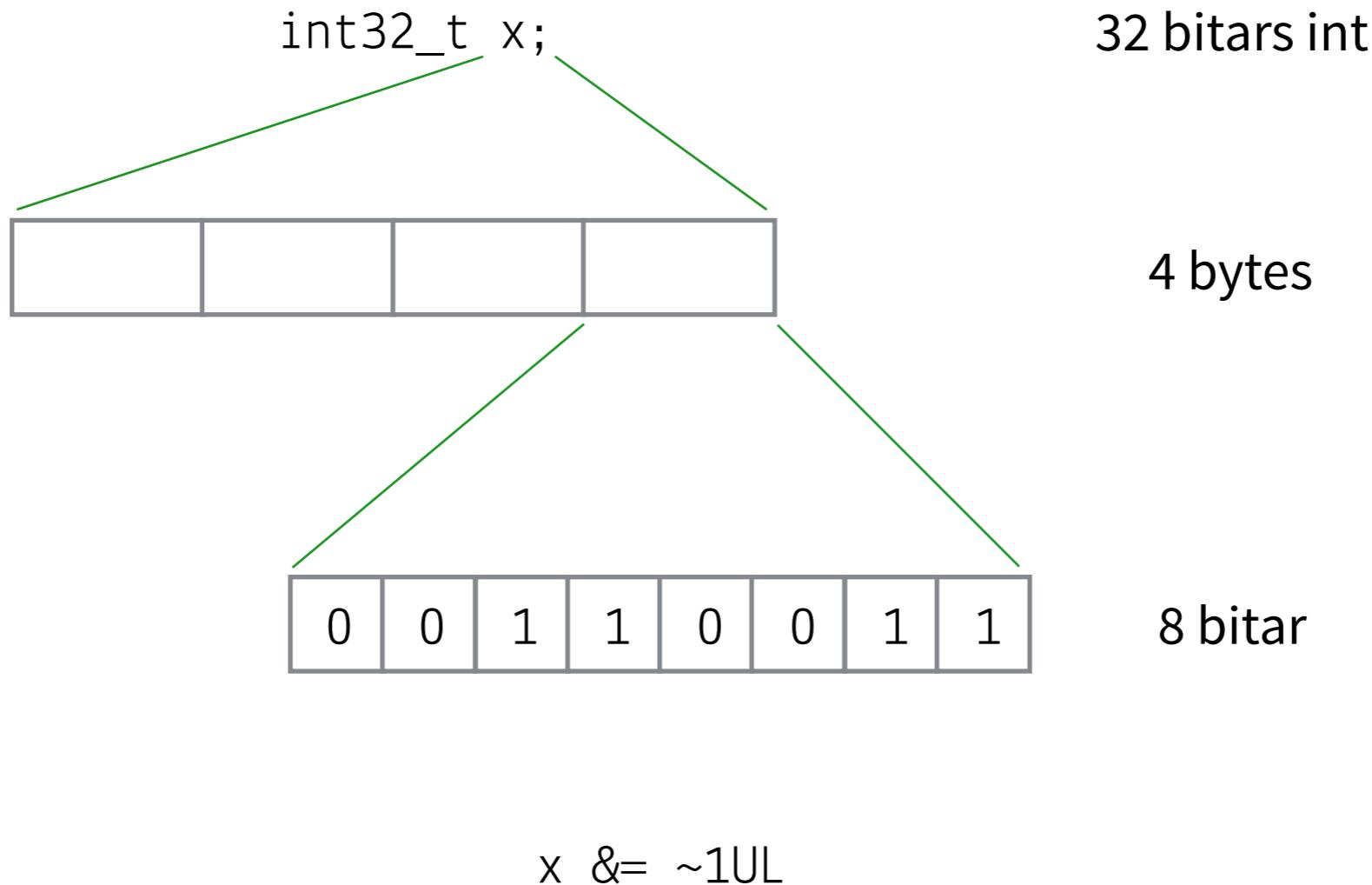
---



`x = 51;`

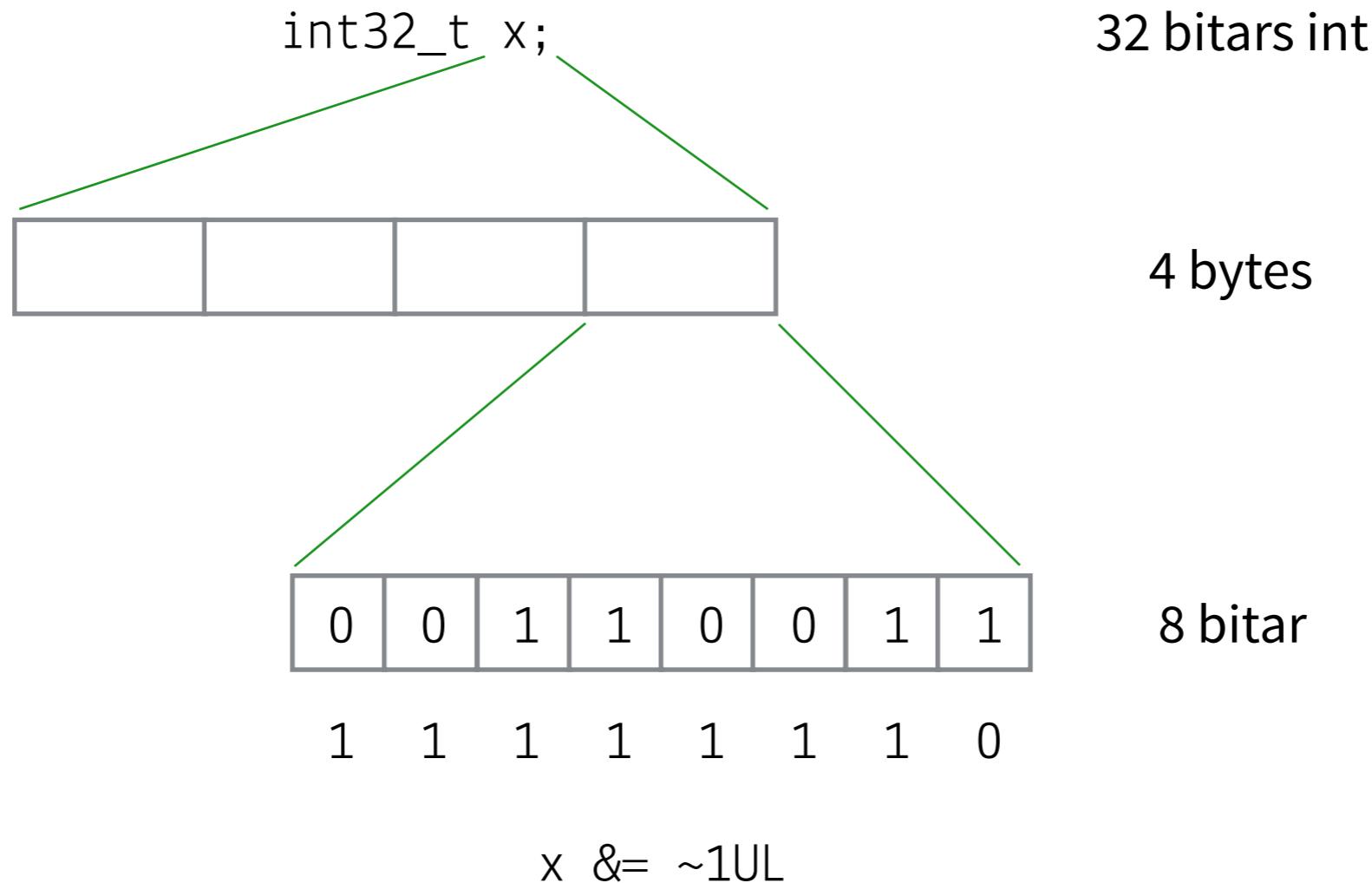
# Sätt bit 0 till 0

---



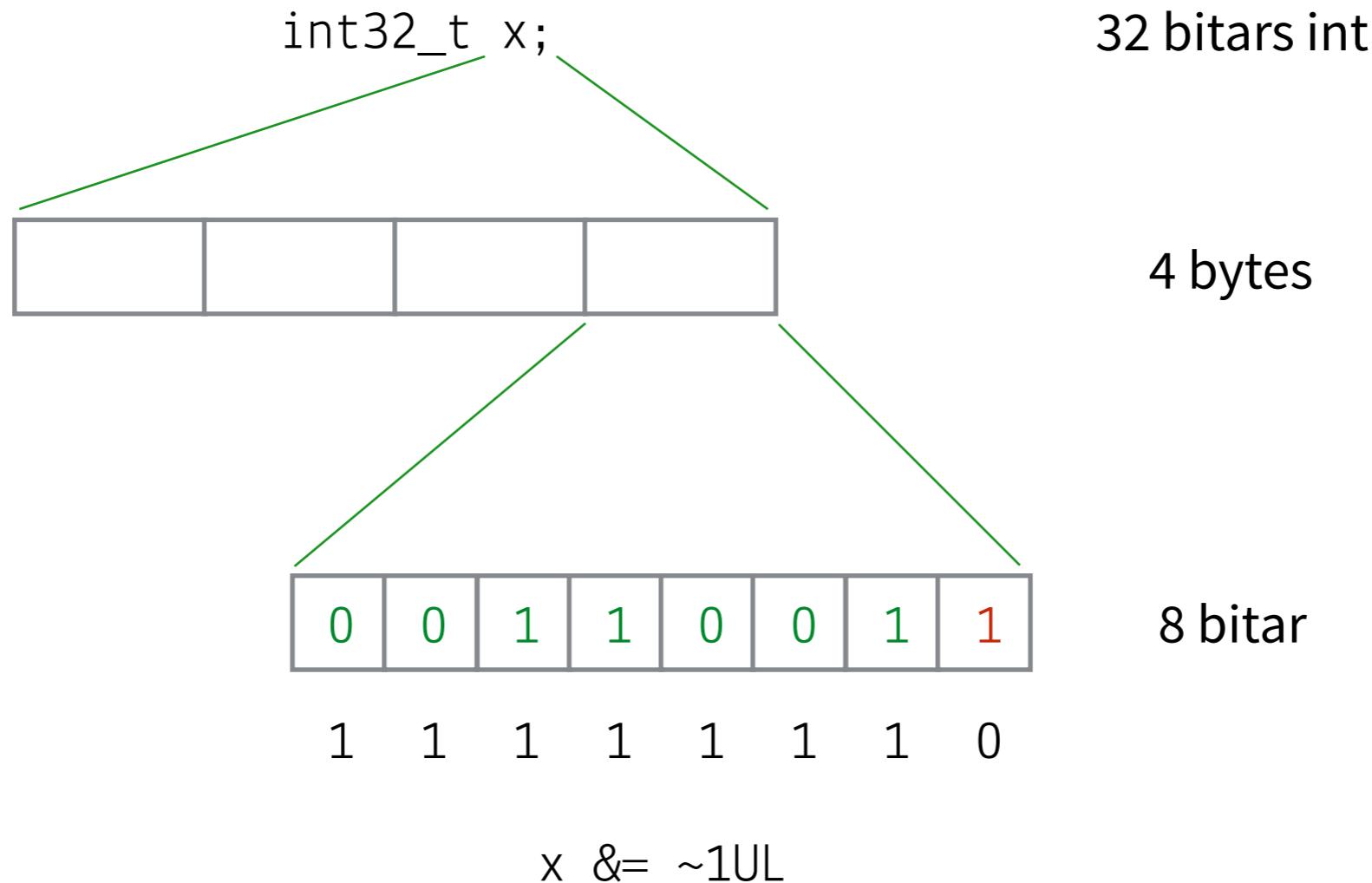
# Sätt bit 0 till 0

---



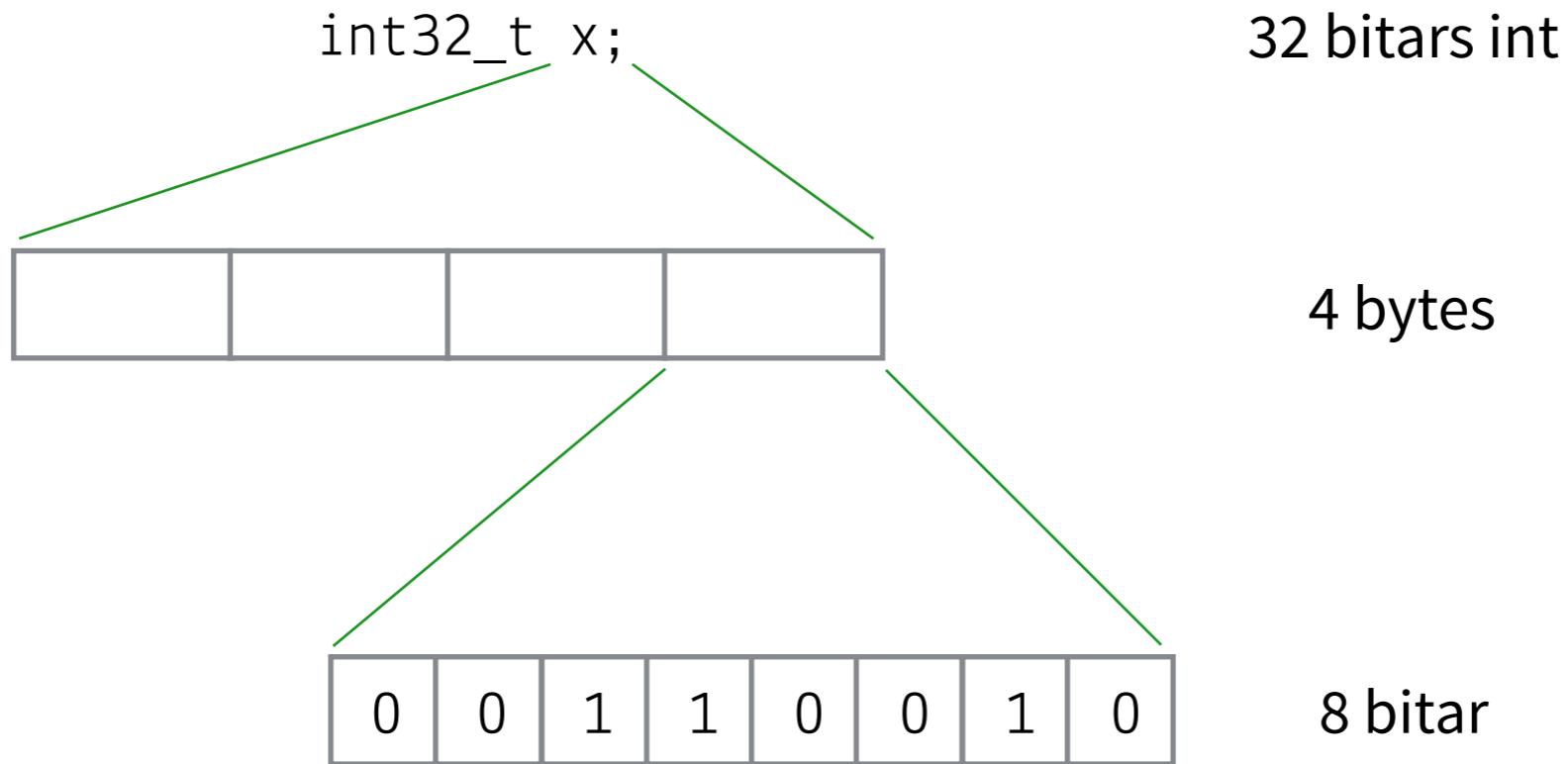
# Sätt bit 0 till 0

---



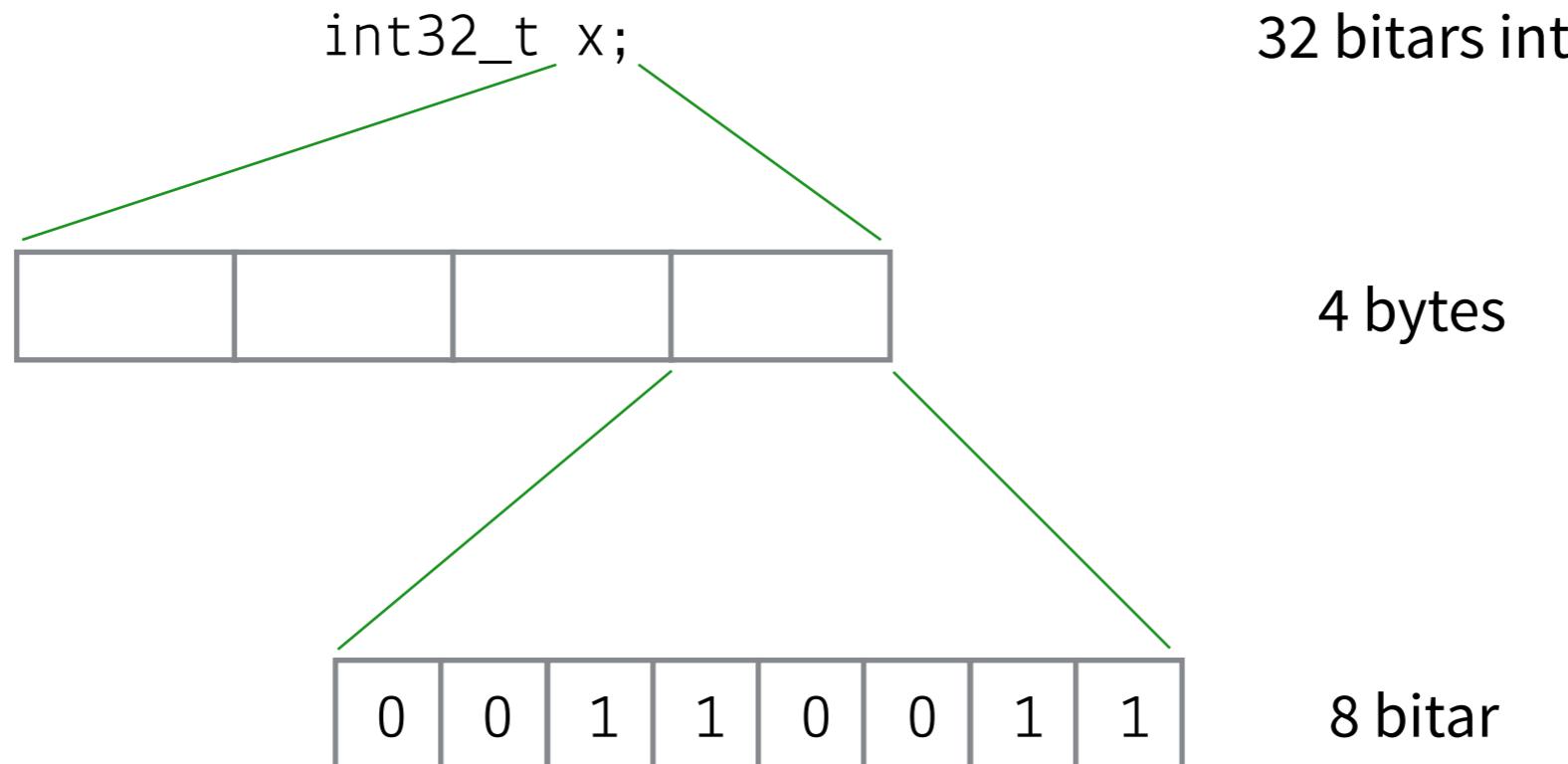
# Sätt bit 0 till 0

---



# Sätt bit 5 till 0

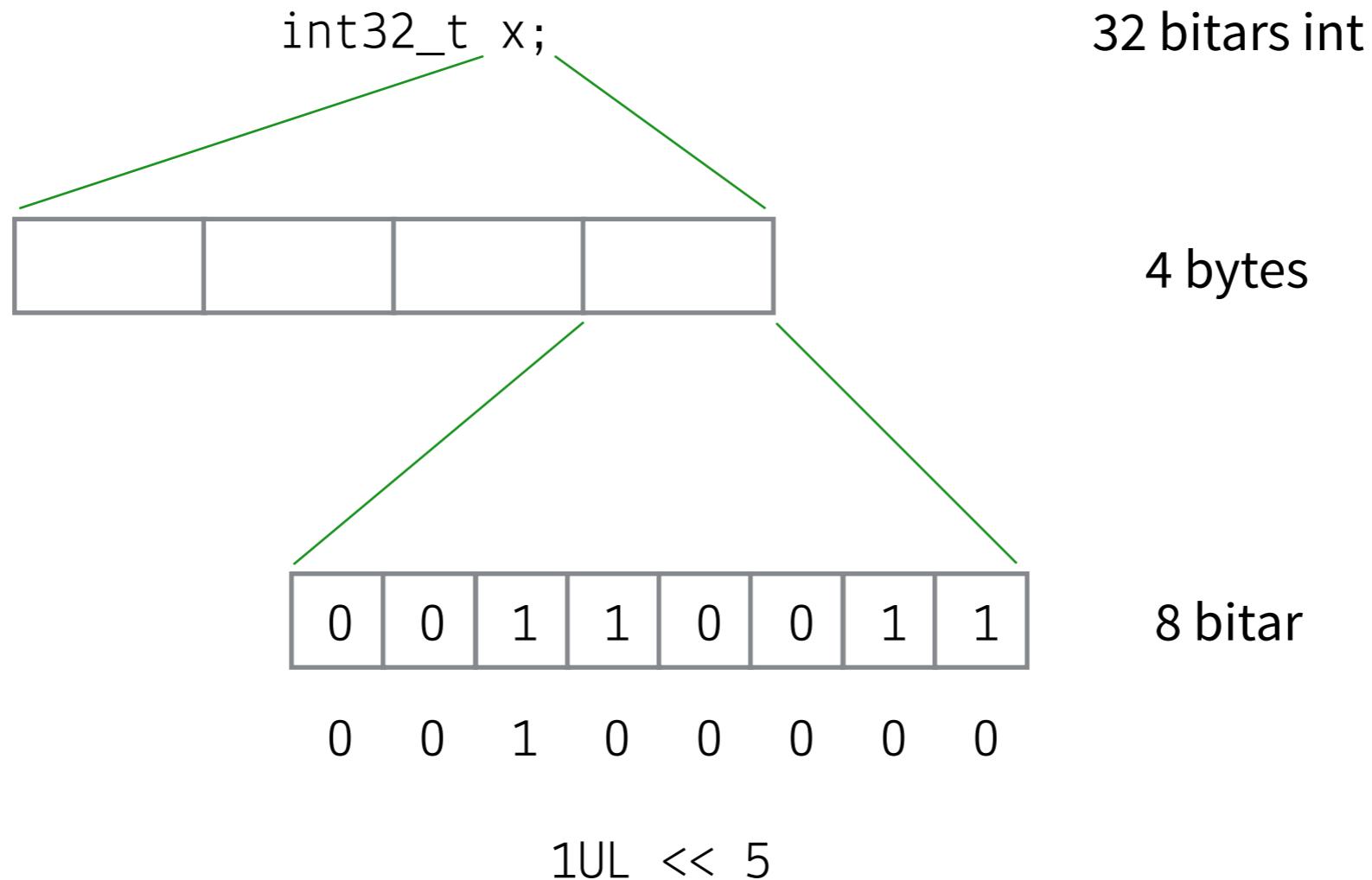
---



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

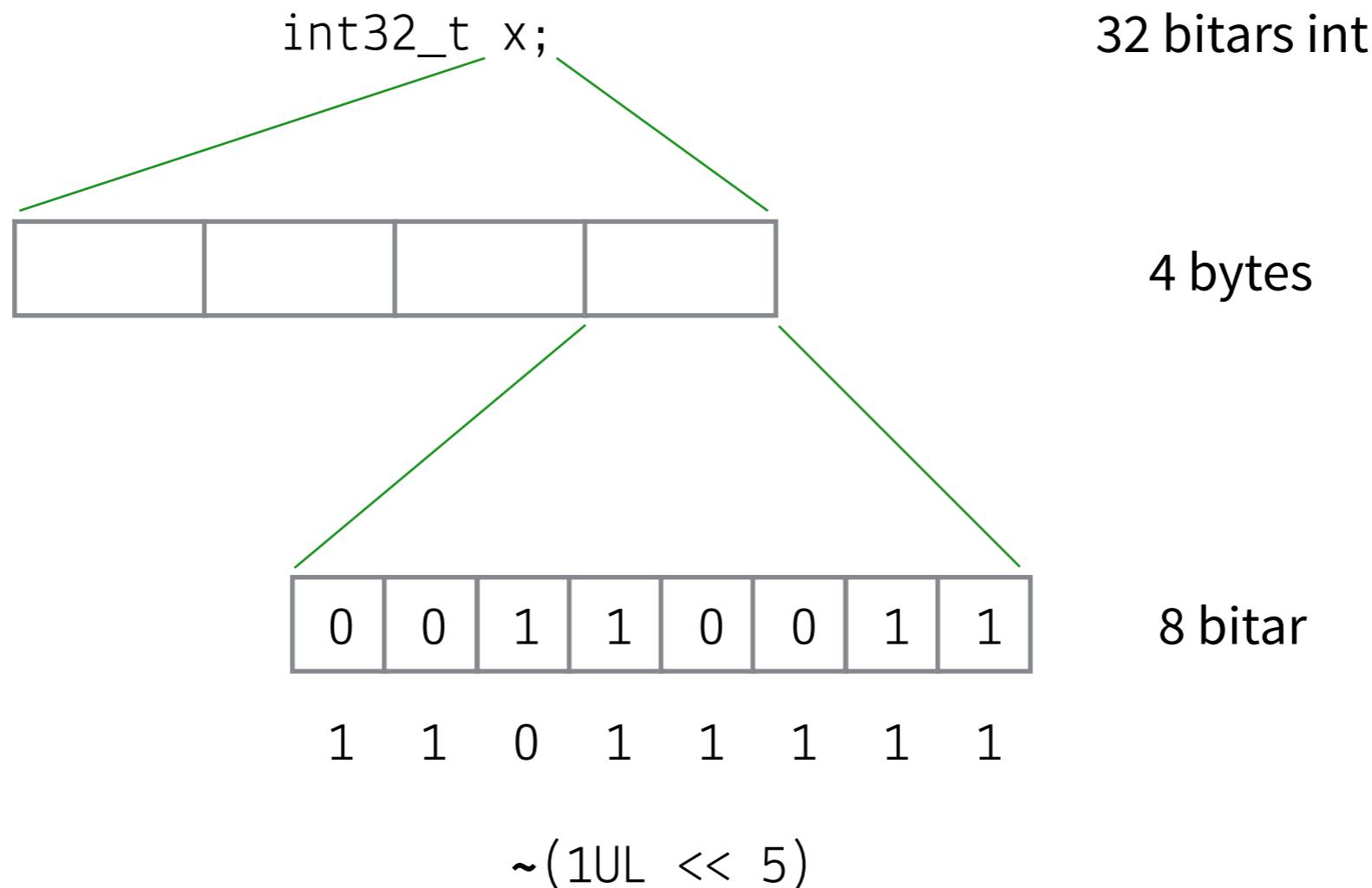
# Sätt bit 0 till 0

---



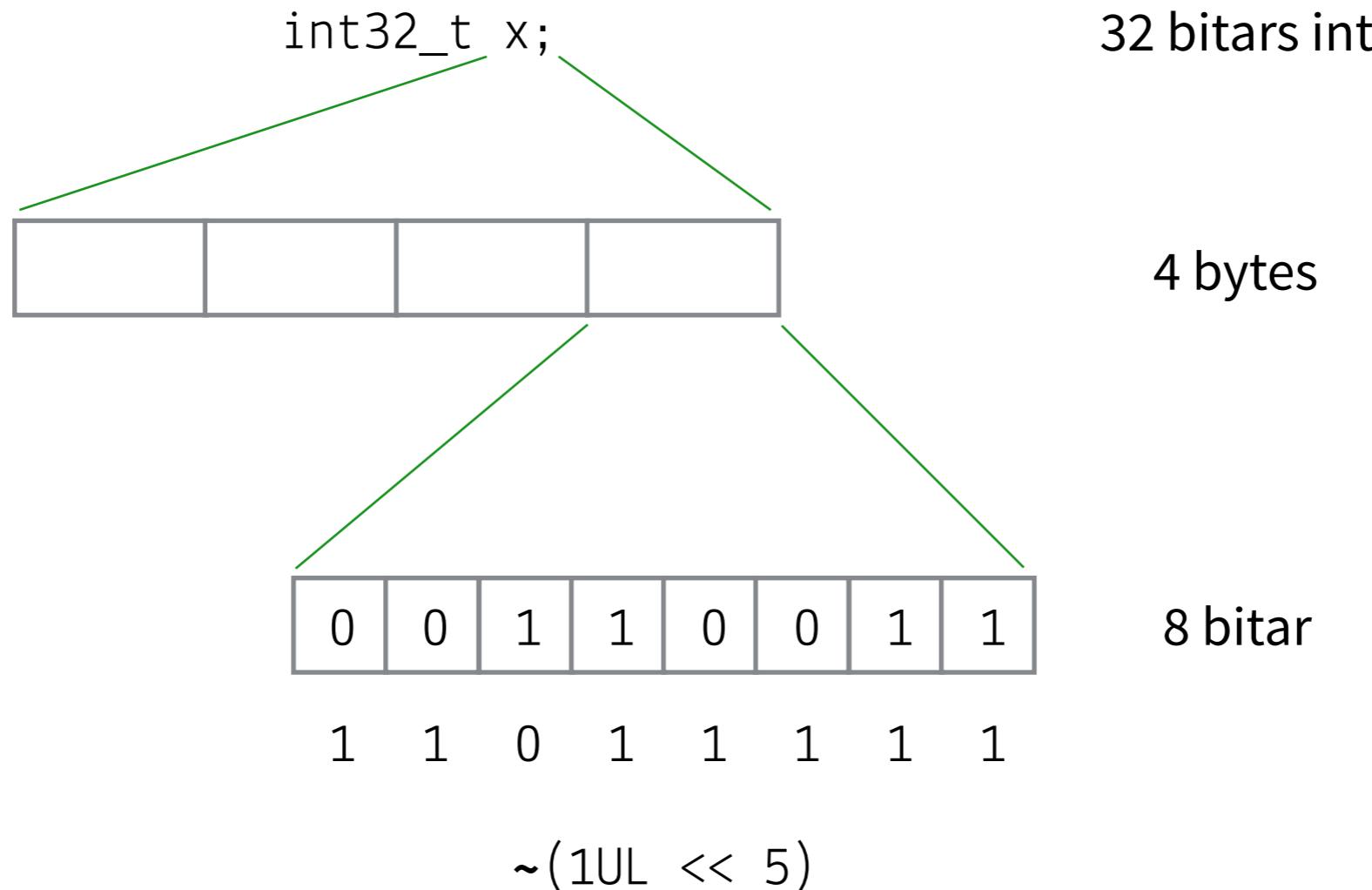
# Sätt bit 0 till 0

---



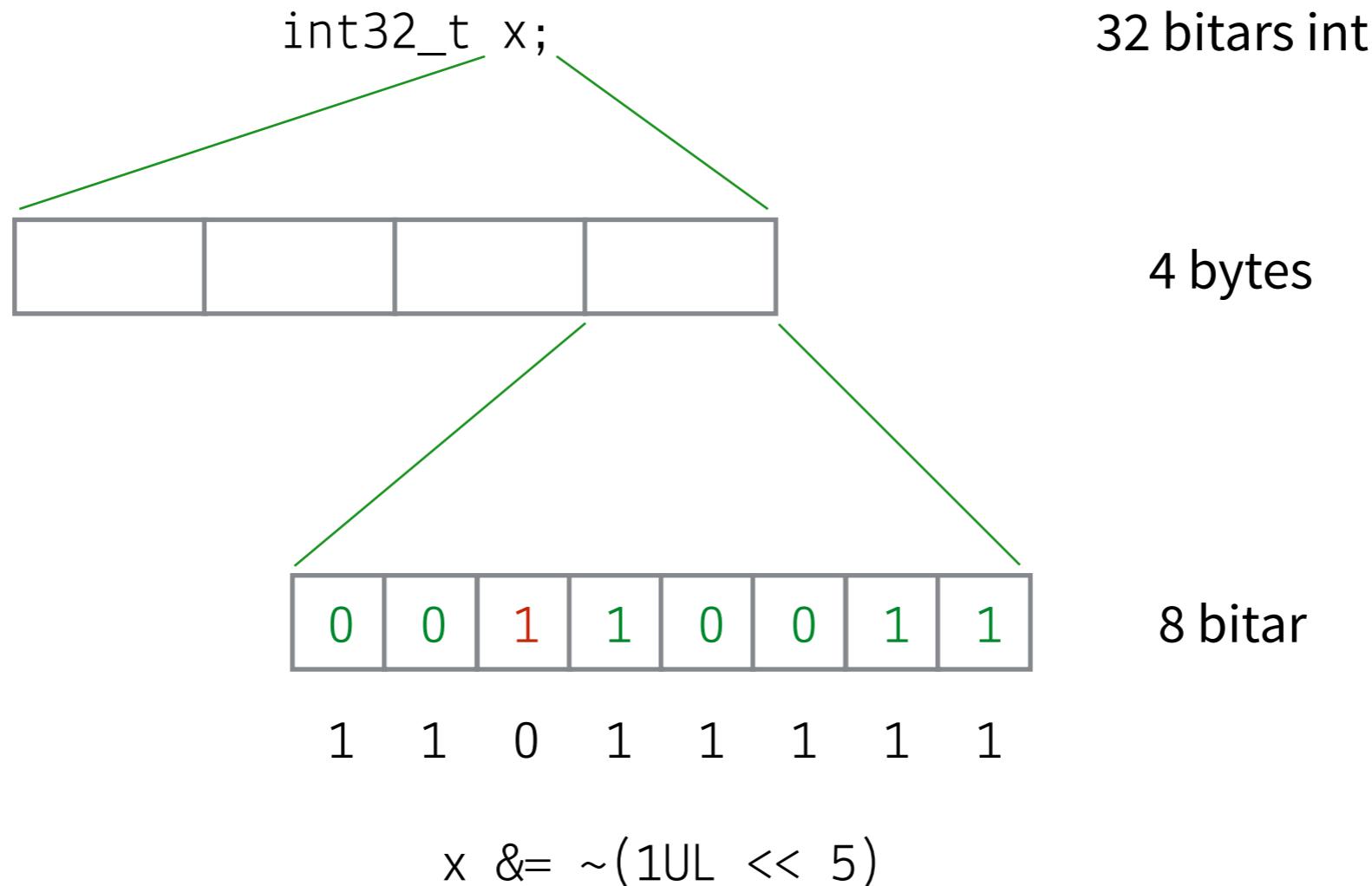
# Sätt bit 0 till 0

---



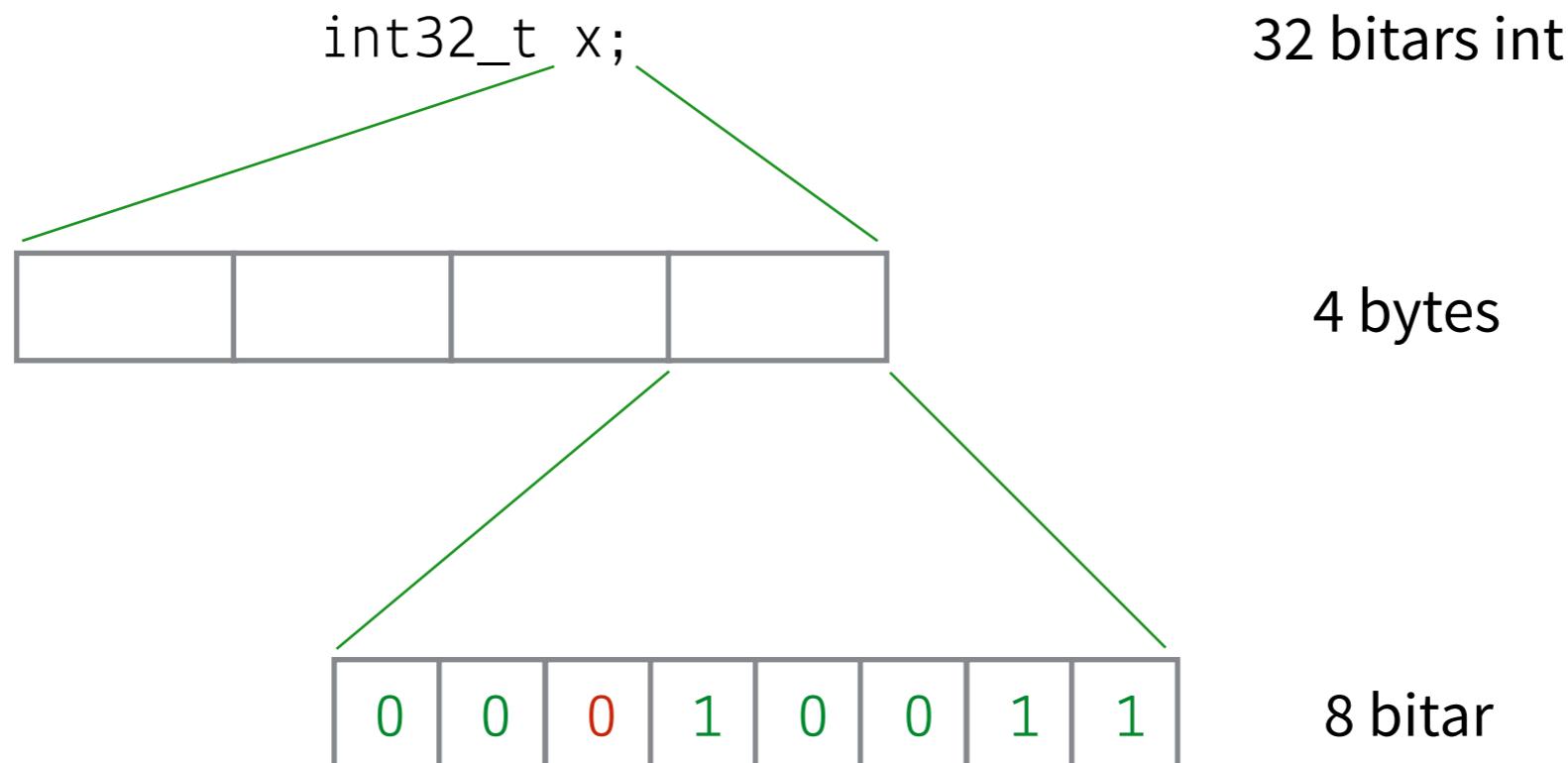
# Sätt bit 0 till 0

---



# Sätt bit 0 till 0

---



# Bitoperatorer i C

---

Operator	Betydelse
<code>~</code>	Unär, negation
<code>&amp;</code>	och
<code> </code>	eller
<code>^</code>	xor
<code>&lt;&lt;</code>	vänsterskift
<code>&gt;&gt;</code>	högerskift

# 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

# Användningsområden

---

- Bitflaggor

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

- Hårdvarunära programmering
- Minneskritiska applikationer

Ex. "bitset" (komprimeringsfaktor 8)

Projektarbetet tidigare år (info om varje objekt...)

## bitset.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);
}
```

```
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);
}
```

