

# Föreläsning 9

---

Tobias Wrigstad

*Bitmanipulering, preprocessorn (och lite om optimering)*



Up: [Mac OS Topics](#)

### 9.4.1 Codesigning the Debugger

The Darwin Kernel requires the debugger to have special permissions before it is allowed to control other processes. These permissions are granted by codesigning the GDB executable. Without these permissions, the debugger will report error messages such as:

```
Starting program: /x/y/foo
Unable to find Mach task port for process-id 28885: (os/kern) failure (0x5).
(please check gdb is codesigned - see taskgated(8))
```

Codesigning requires a certificate. The following procedure explains how to create one:

- Start the Keychain Access application (in /Applications/Utilities/Keychain Access.app)
- Select the Keychain Access -> Certificate Assistant -> Create a Certificate... menu
- Then:
  - Choose a name for the new certificate (this procedure will use "gdb-cert" as an example)
  - Set "Identity Type" to "Self Signed Root"
  - Set "Certificate Type" to "Code Signing"
  - Activate the "Let me override defaults" option
- Click several times on "Continue" until the "Specify a Location For The Certificate" screen appears, then set "Keychain" to "System"
- Click on "Continue" until the certificate is created
- Finally, in the view, double-click on the new certificate, and set "When using this certificate" to "Always Trust"
- Exit the Keychain Access application and restart the computer (this is unfortunately required)

Once a certificate has been created, the debugger can be codesigned as follow. In a Terminal, run the following command:

```
$ codesign -f -s "gdb-cert" <gnat_install_prefix>/bin/gdb
```

where "gdb-cert" should be replaced by the actual certificate name chosen above, and <gnat\_install\_prefix> should be replaced by the location where you installed GNAT. Also, be sure that users are in the Unix group `_developer`.

*sudo killall taskgated*

[https://gcc.gnu.org/onlinedocs/gnat\\_ugn/Codesigning-the-Debugger.html](https://gcc.gnu.org/onlinedocs/gnat_ugn/Codesigning-the-Debugger.html)



```
extern char *strdup(const char *);
```

```
-D_GNU_SOURCE
```



# Preprocessor



# CPP — The C PreProcessor

---

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

<code>#include &lt;stdio.h&gt;</code>		Inkludering av annan fil
<code>#include "myfile.h"</code>		
<code>#ifndef symbol</code>		If-sats som körs vid kompileringen
<code>#define symbol</code>		
<code>...</code>		
<code>#endif</code>		
<code>#define INT_MAX 2147483647</code>		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 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

---

- 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; \    // 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
```

- 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 "skr p"
```

```
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;
}
```

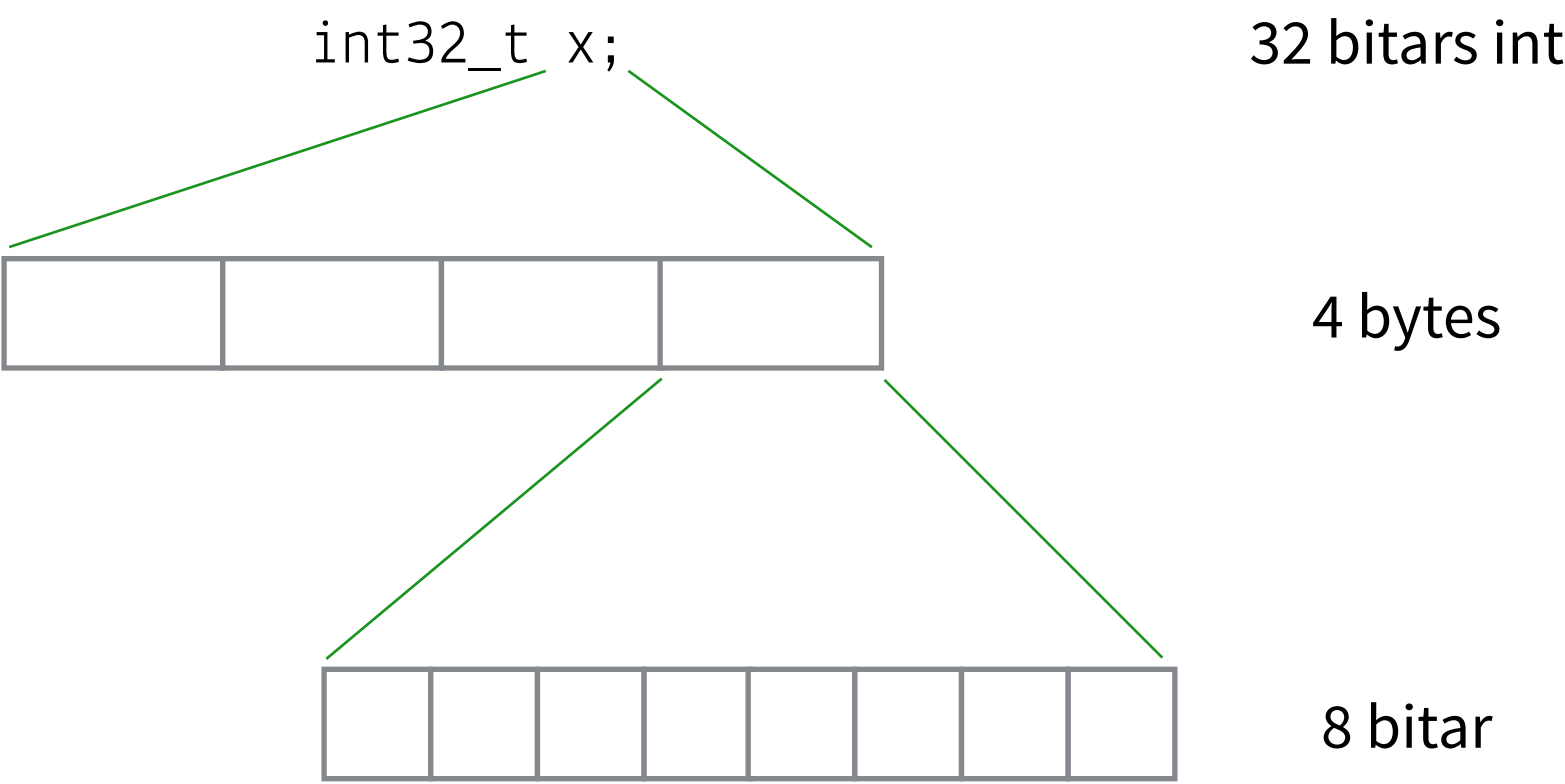
test.c

# Bitmanipulering



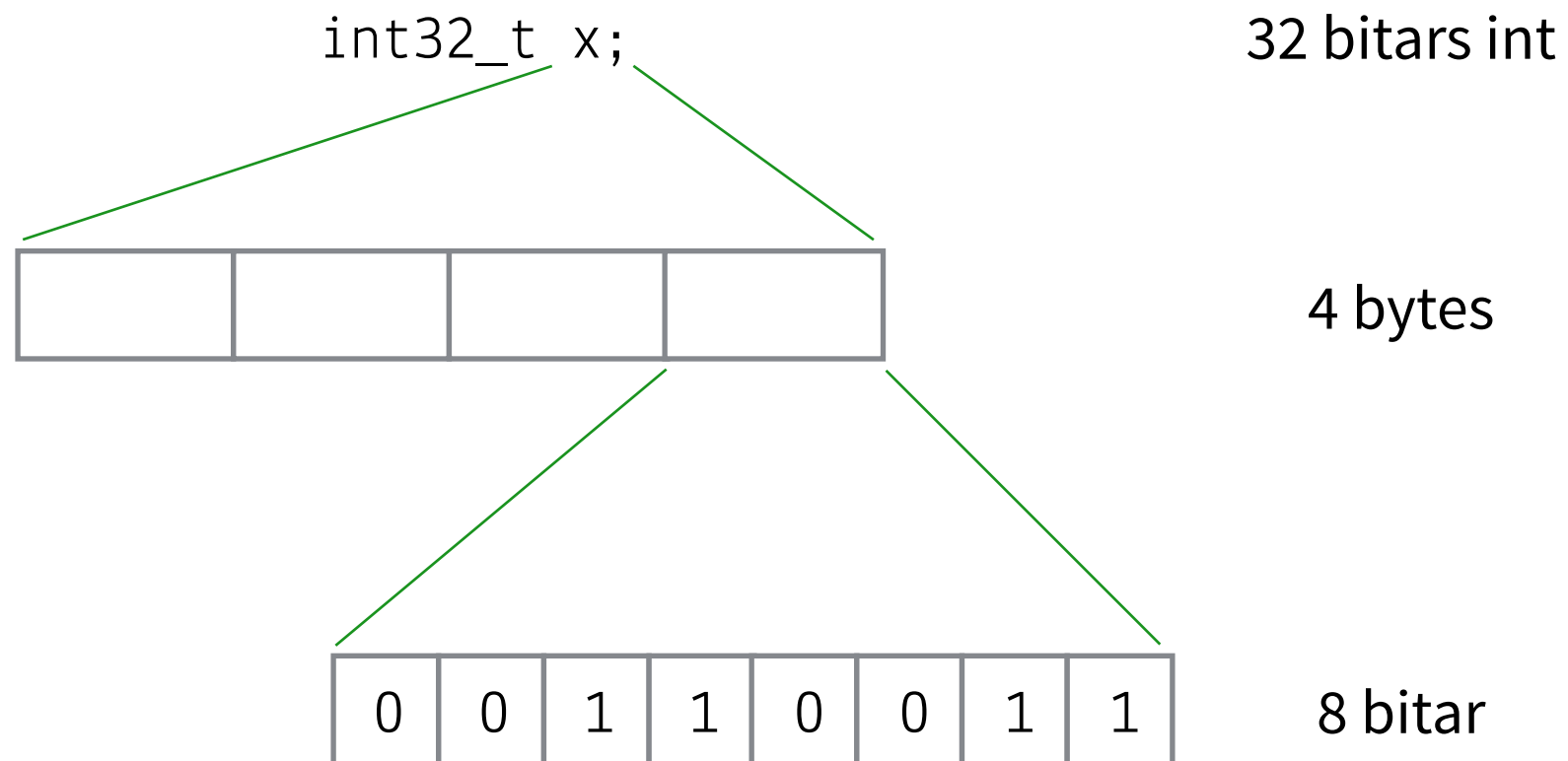


# Bitar



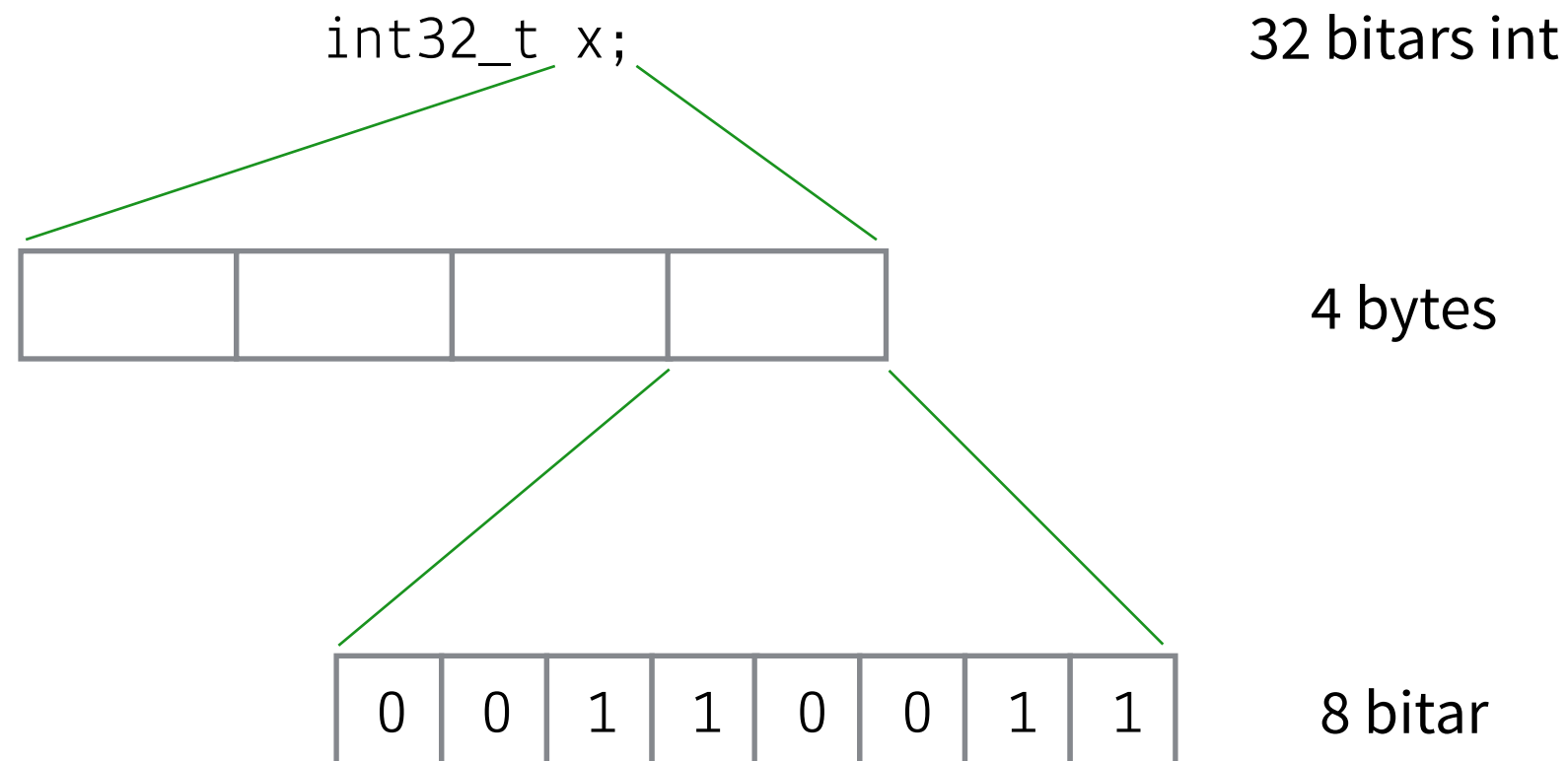
`x = 51;`

# Bitar



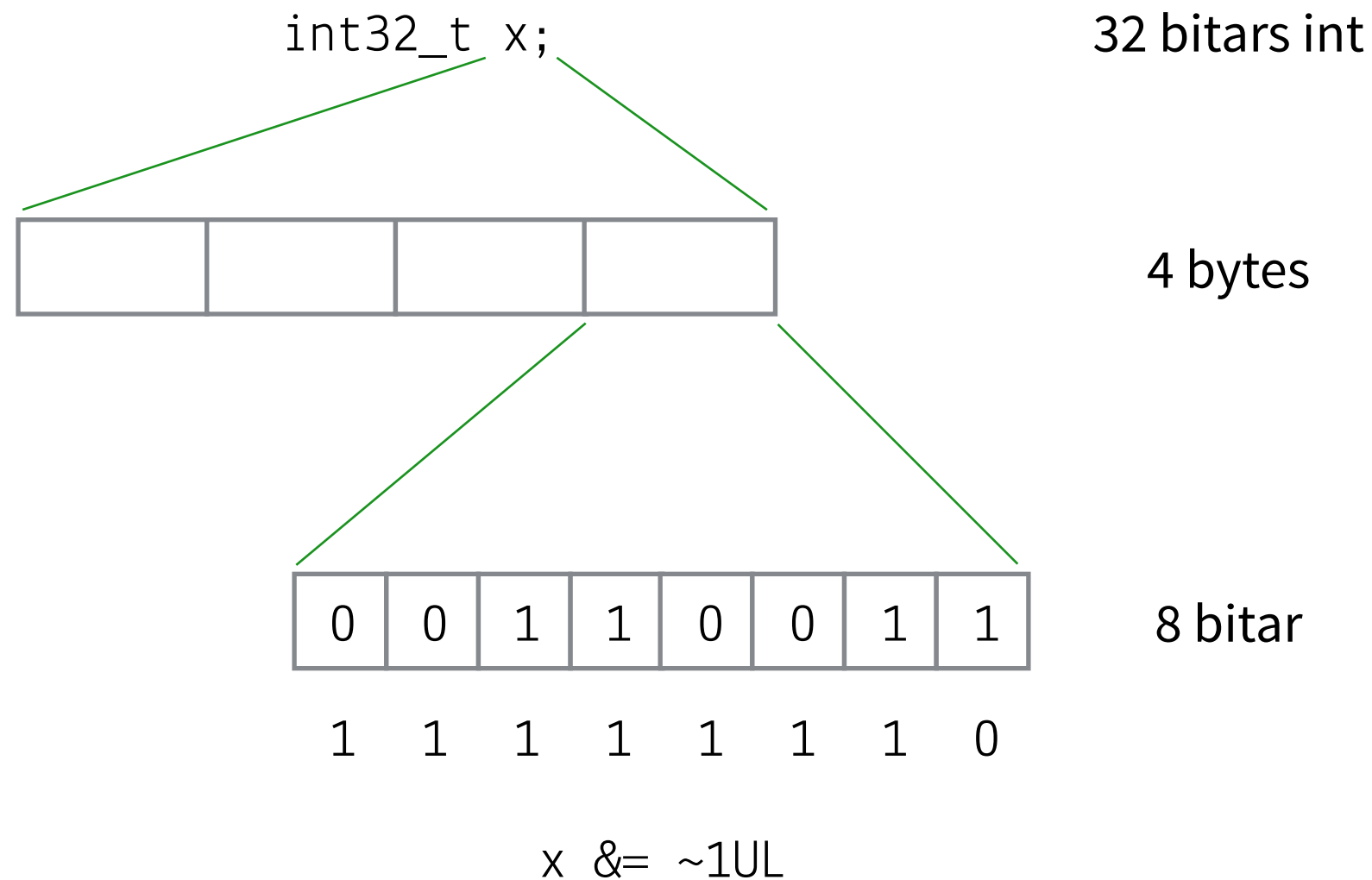
x = 51;

# Sätt bit 0 till 0

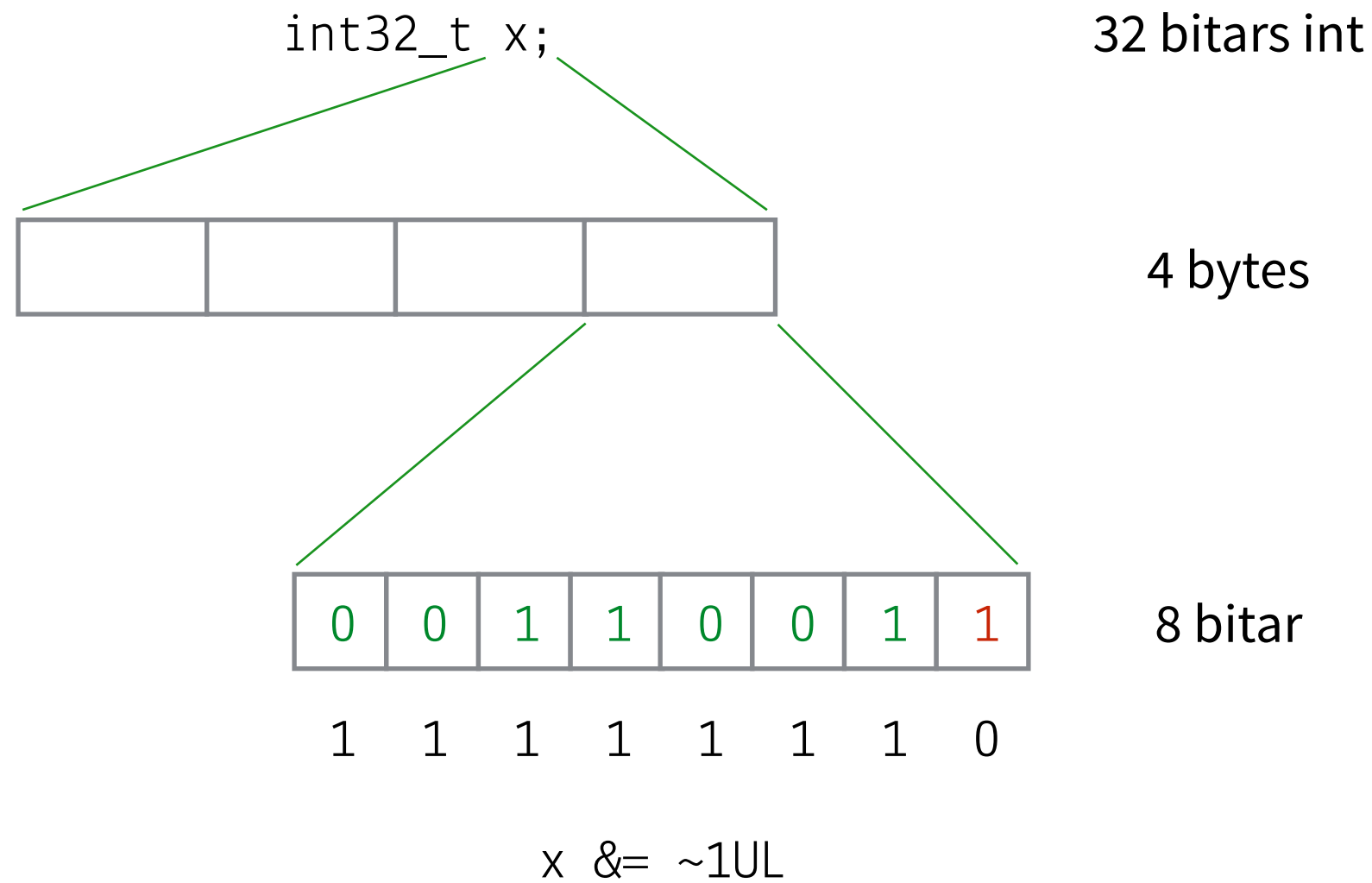


`x &= ~1UL`

# Sätt bit 0 till 0



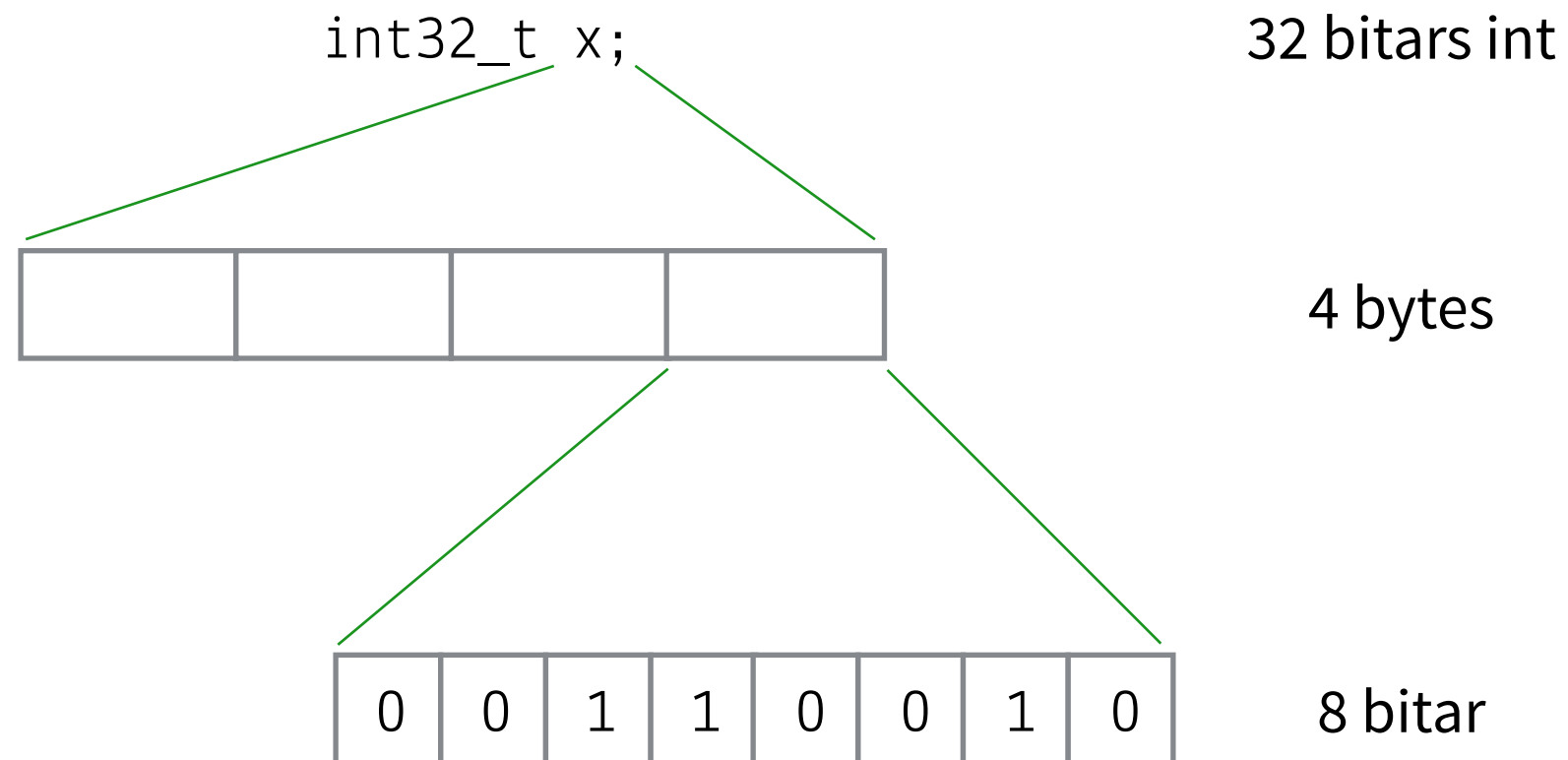
# Sätt bit 0 till 0



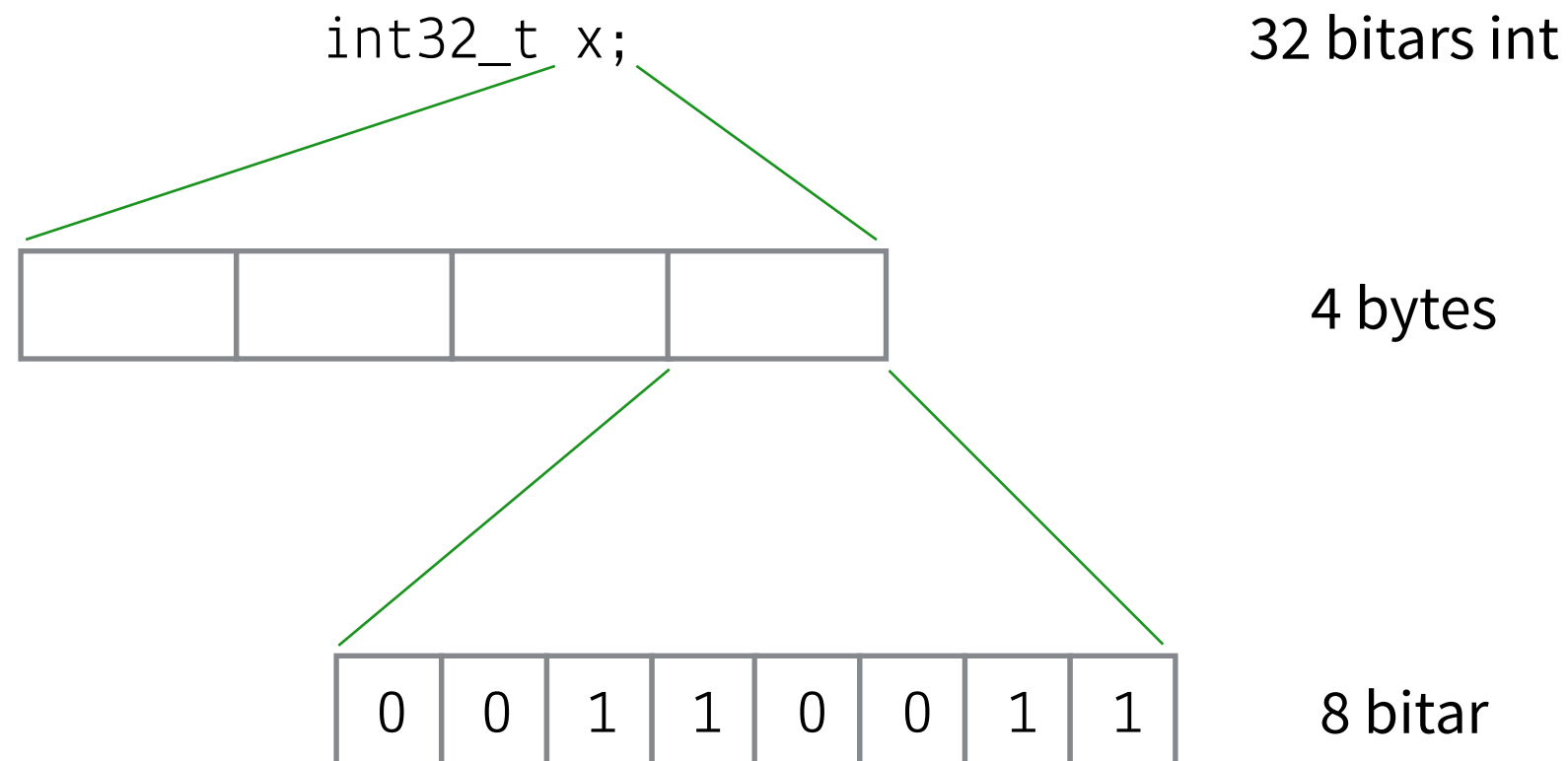


# Sätt bit 0 till 0

---

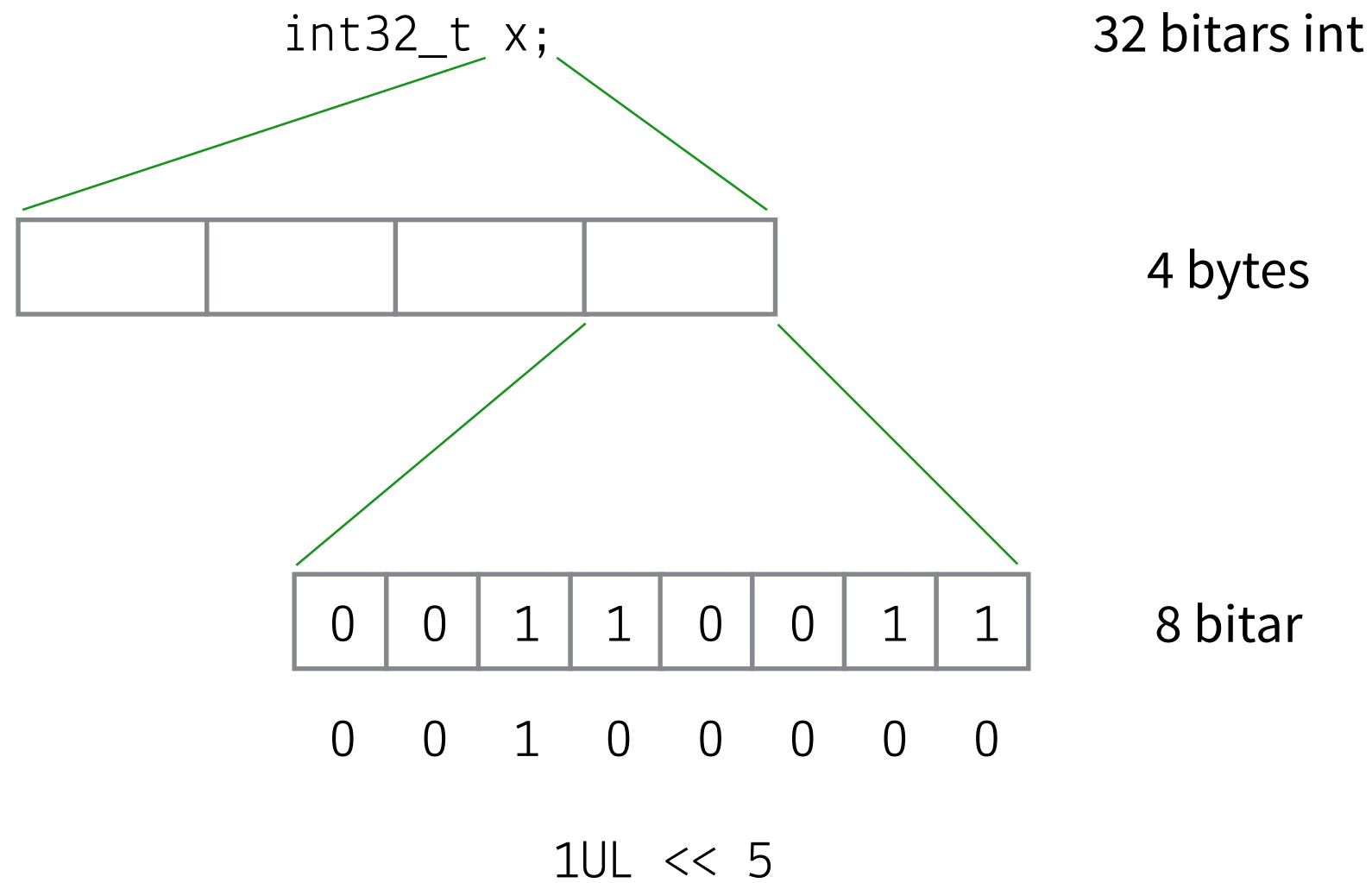


# Sätt bit 5 till 0

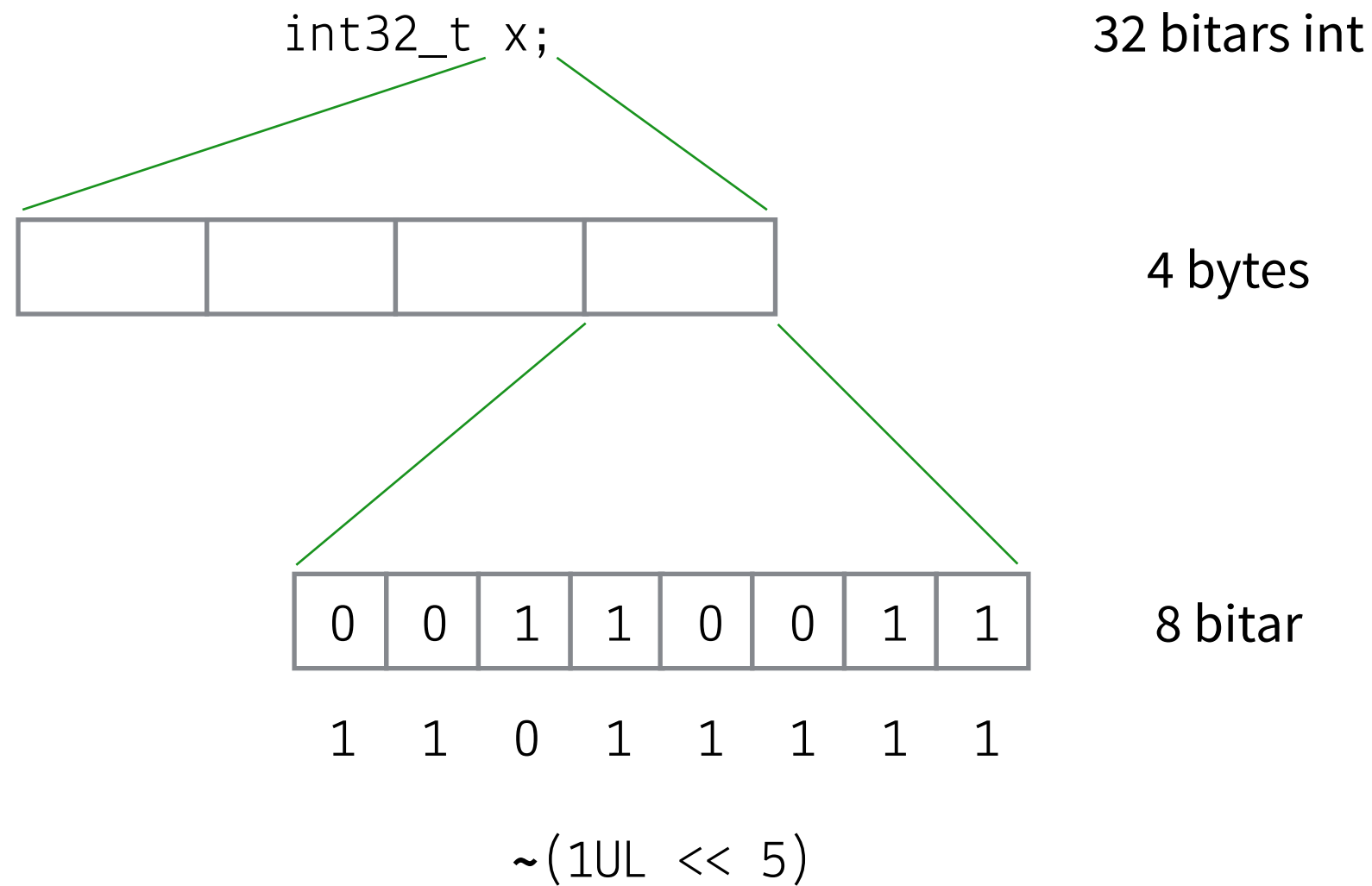


`x &= ~(1UL << 5)`

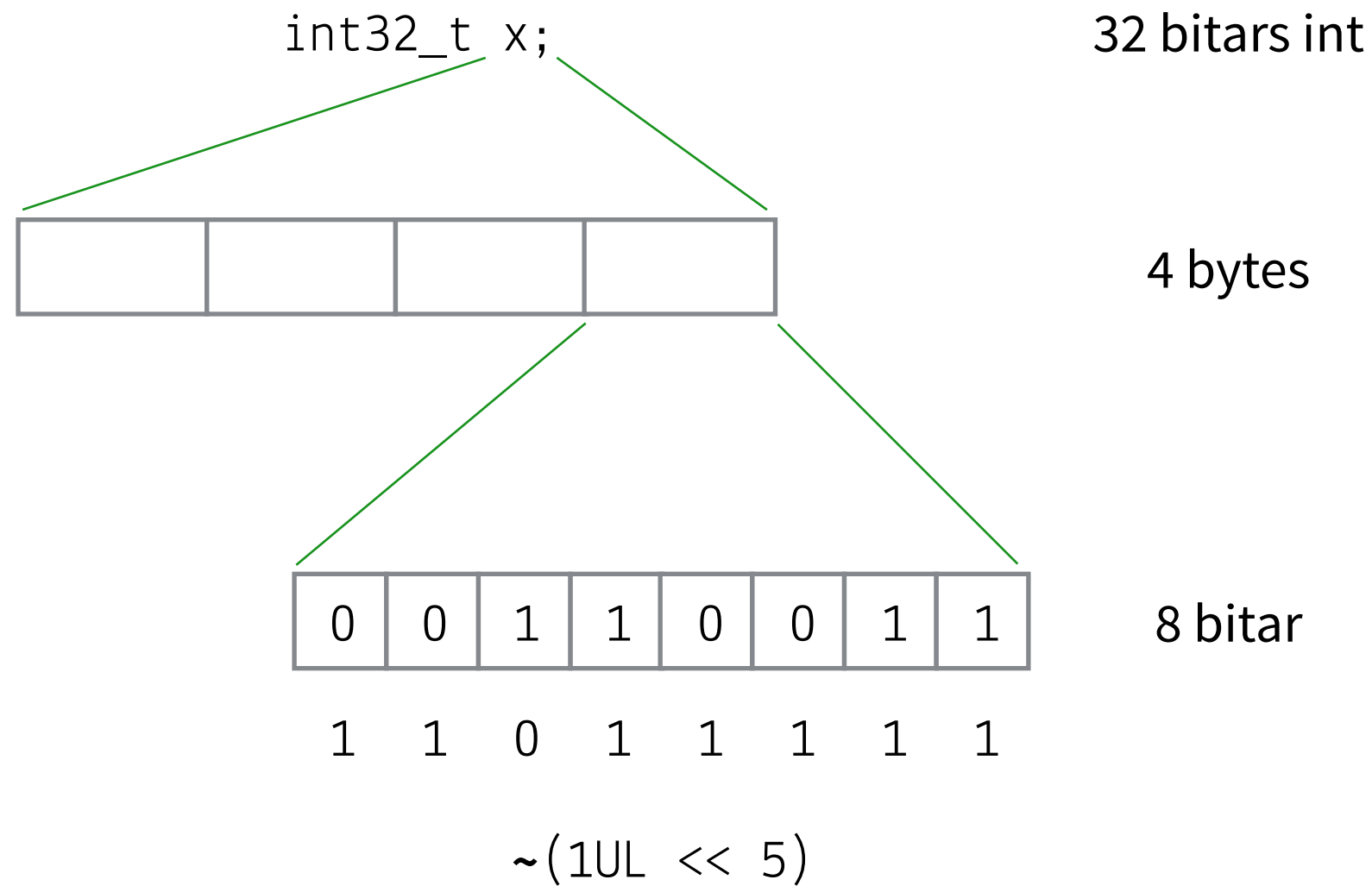
# Sätt bit 0 till 0



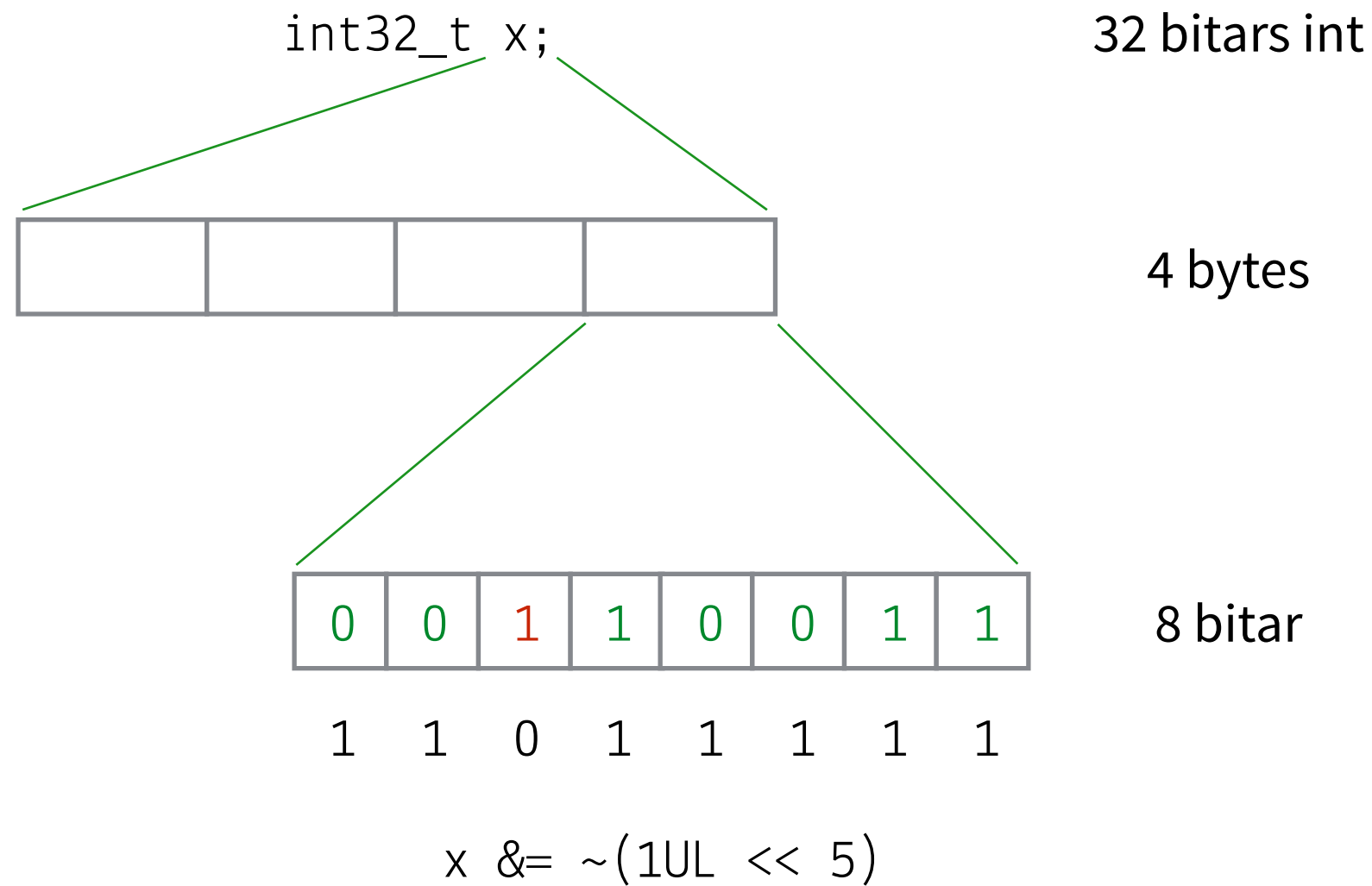
# Sätt bit 0 till 0



# Sätt bit 0 till 0

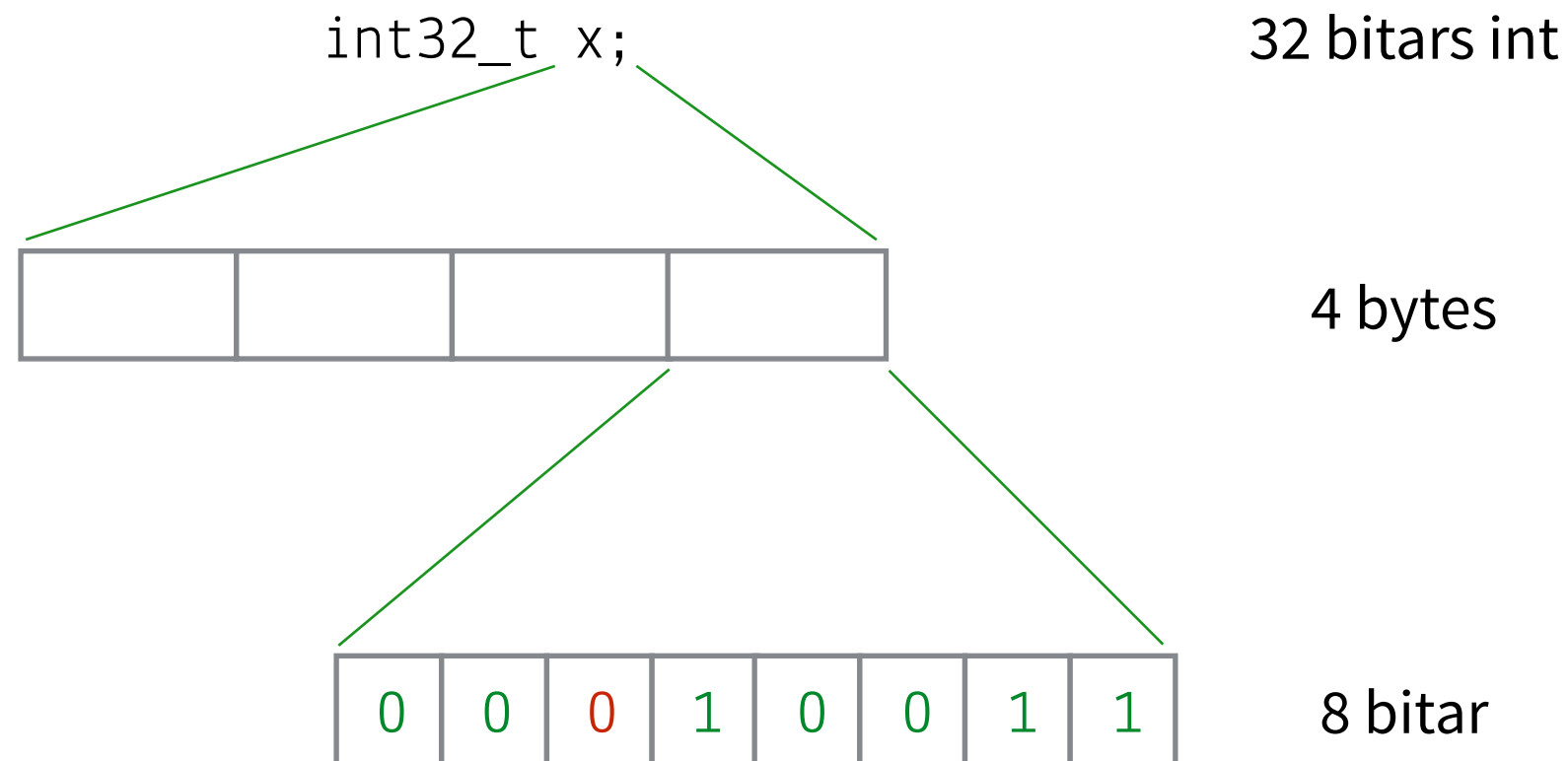


# Sätt bit 0 till 0



# Sätt bit 0 till 0

---



# Bitoperatorer i C

---

Operator	Betydelse
~	Unär, negation
&	och
	eller
^	xor
<<	vänsterskift
>>	högerskift



# Bitoperatorer i C

Operator	Bitström
<code>a = 1UL</code>	00000000 00000000 00000000 00000001
<code>b = ~1UL</code>	11111111 11111111 11111111 11111110
<code>a &amp; b</code>	00000000 00000000 00000000 00000000
<code>a   b</code>	11111111 11111111 11111111 11111111
<code>7 ^ 3</code>	$0111 \wedge 0011 = 0100 = 4$
<code>a &lt;&lt; 3</code>	00000000 00000000 00000000 00001000
<code>b &gt;&gt; 5</code>	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...)

```
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[]; // Cool C99 trick
};

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);
}
```



# Optimering



# Tre regler för optimering

---

- Den första regeln för optimering är

*Gör det inte*

- Den andra regeln för optimering är

*Gör det inte förrän du har klara bevis på att optimering är nödvändigt*

- Den tredje regeln för optimering är

*Du kan inte räkna ut vad som bör optimeras — du behöver profileringsdata*

# Följdsatser

---

- Följdsats §1

*Optimering tenderar att göra bra kod oläslig*

- Följdsats §2

*Kostnaden för programmets prestandaökning betalas igen i underhåll*

# Optimering

---

- De viktigaste optimeringarna är i regel algoritmiska

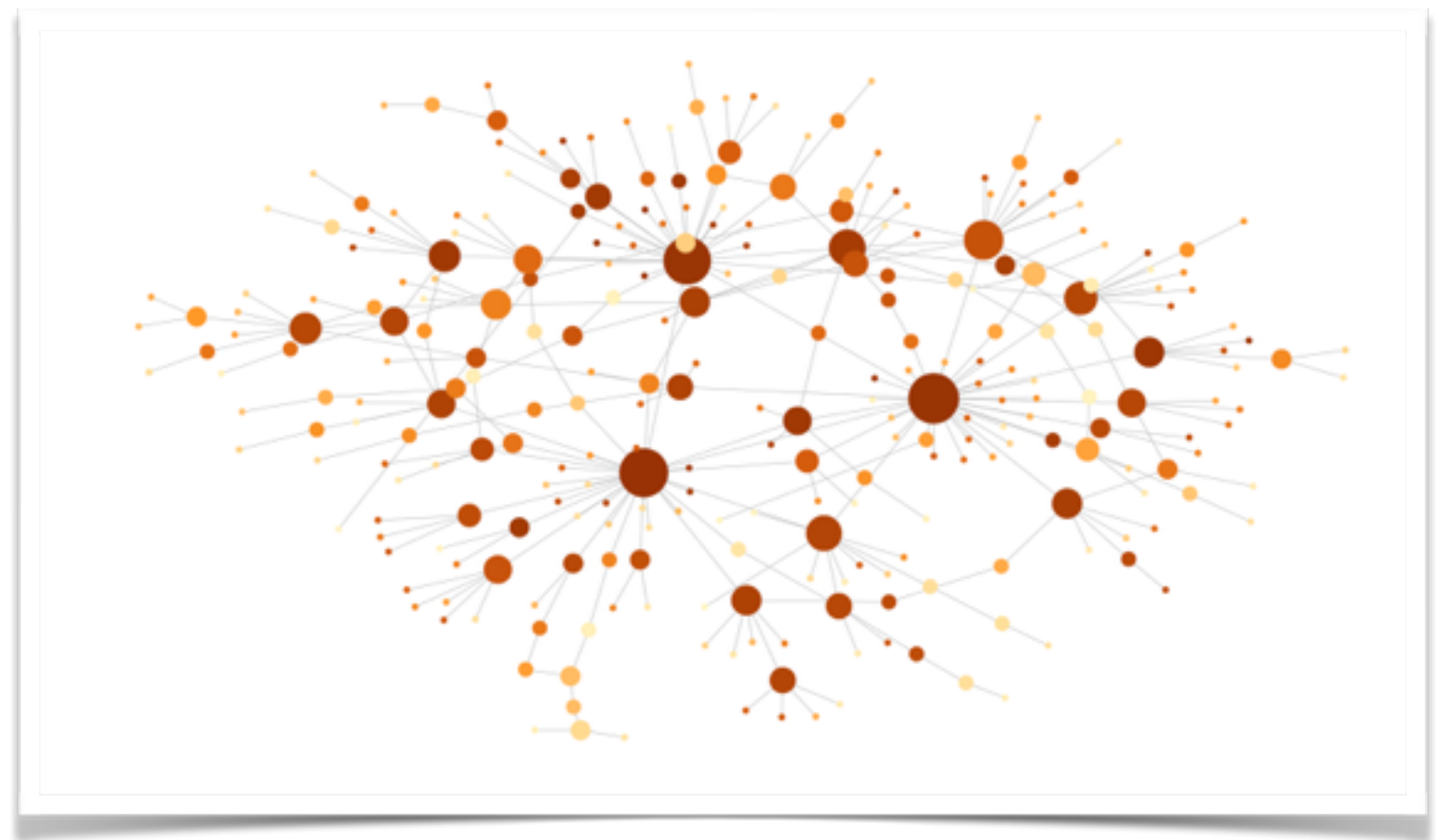
Använd ett bättre sätt att räkna fram det värde du vill ha

# Exempel: Preferential Attachment

- Problem: simulera uppbyggnad av nätverk, t.ex. sociala nätverk

Sannolikheten för att X skall vara kopplad till varje ny person ökar ju fler kopplingar X redan har

*A **preferential attachment process** is any of a class of processes in which some quantity, typically some form of wealth or credit, is distributed among a number of individuals or objects according to how much they already have, so that those who are already wealthy receive more than those who are not.*





# Hur attackerar vi problemet?

---

- Varje gång en ny person läggs till i grafen skall hen kopplas till  $K$  personer i grafen
- Sannolikheten är proportionell mot antalet kopplingar en redan har
- Om man tillåts ha flera kopplingar till samma person blir sannolikheten skev
- Problem:

*Hur håller vi reda på antalet kopplingar för varje person?*

*Hur tar vi hänsyn till det när vi slumpar fram nya kopplingar?*

# Lösning: datastrukturen för grafen

5	1	5	3	5	4	6	1	6	5	6	3	7	1	7	5	7	6	8	7	...
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	-----



länk från 5 till 1

- Varje gång vi lägger till något i grafen på plats P kan vi bara välja ett slumpvist element till vänster om P

*Sannolikheten att bli vald är proportionell mot kopplingsgraden*

*Vi optimerar nu för att slumpvalet skall vara effektivt — hur vet vi att det ens är viktigt?*

# Profilering

---

- Använda gprof för att ta fram en profil för ett program

```
$ gcc -pg myprog.c -o myprog
```

```
$ ./myprog
```

```
$ gprof gmon.out myprog > profile.txt
```

```
$ less profile.txt
```

```
c=0;
/*
```

Nu pekar c på minnesadressen 00000 (?), som är ett read-only minne och bara säger noll. ?  
det är bra att göra en sådan initialisering direkt så att man inte råkar skriva över random data.  
Varför görs inte det automatiskt då?

```
----- */
```

Flat profile:

Each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	s/call	s/call	name
16.01	1.30	1.30	32001519	0.00	0.00	treeinsert
15.83	2.59	1.29	15999889	0.00	0.00	worker_commit
12.19	3.58	0.99	1599989	0.00	0.00	treeinsert
10.10	4.40	0.82				array_get
9.91	5.20	0.81	15999890	0.00	0.00	worker_generate_connection
8.99	5.93	0.73				run
6.65	6.47	0.54	1599989	0.00	0.00	worker_add_full_node
6.04	6.96	0.49	16001630	0.00	0.00	random_rand
4.31	7.31	0.35	16001630	0.00	0.00	random_coin_flip
2.40	7.51	0.20	15999889	0.00	0.00	dependencies_fire
2.34	7.70	0.19	8000865	0.00	0.00	worker_resolved_backward_dependency
2.09	7.87	0.17	8000765	0.00	0.00	worker_resolved_node
0.86	7.94	0.07				run_thread
0.62	7.99	0.05	1	0.05	6.37	worker_explore
0.49	8.03	0.04				cpu_core_pause
0.37	8.06	0.03				array_set
0.12	8.07	0.01	1	0.01	0.01	random__init
0.12	8.08	0.01	1	0.01	0.01	treeinsert__init
0.12	8.09	0.01				array_size
0.12	8.10	0.01				messageq_push
0.12	8.11	0.01				pony_thread_create
0.12	8.12	0.01				quiescent
0.06	8.12	0.01				dependencies_insert



# Hur attackerar vi problemet?

---

- Varje gång en ny person läggs till i grafen skall hen kopplas till  $K$  personer i grafen
- Sannolikheten är proportionell mot antalet kopplingar en redan har
- Om man tillåts ha flera kopplingar till samma person blir sannolikheten skev
- Problem:

>30% av tiden

*Hur håller vi reda på antalet kopplingar för varje person?*

*Hur tar vi hänsyn till det när vi slumpar fram nya kopplingar?*

# Profilering och optimering under kursen

---

- Profiler inte ett interaktivt program! (t.ex. lagerhanteraren)

99,999% av dess tid går åt till att vänta på att användaren trycker på en tangent

- Välj någon liten del

T.ex. det binära sökträd som håller i alla varor

Skriv ett program som lägger in 1 M varor och tar bort 100 k varor

Profiler detta program!