

Binary Exploit - III





About

- ▶ u1f383 🎃
- ▶ Pwner
- ▶ Zoolab
- ▶ NCtfU / Xx TSJ xX /
Goburin'



Outline

- ▶ Prior knowledge
 - ⌚ Memory layout
 - ⌚ Protection
 - ⌚ Bypass ASLR
 - ⌚ One gadget



Outline

- ▶ Heap introduction
 - ⌚ Memory allocation
 - ⌚ Data structure
- ▶ Code tracing
- ▶ Vulnerability
 - ⌚ UAF
 - ⌚ Heap overflow
 - ⌚ Double free



Outline

- ▶ Exploitation goal
 - ⦿ Write hook
- ▶ Exploitation tech
 - ⦿ Tcache poisoning
 - ⦿ Fastbin attack
 - ⦿ Overlapping chunks



Outline

- ▶ Safe linking
- ▶ How2heap tech
- ▶ Appendix



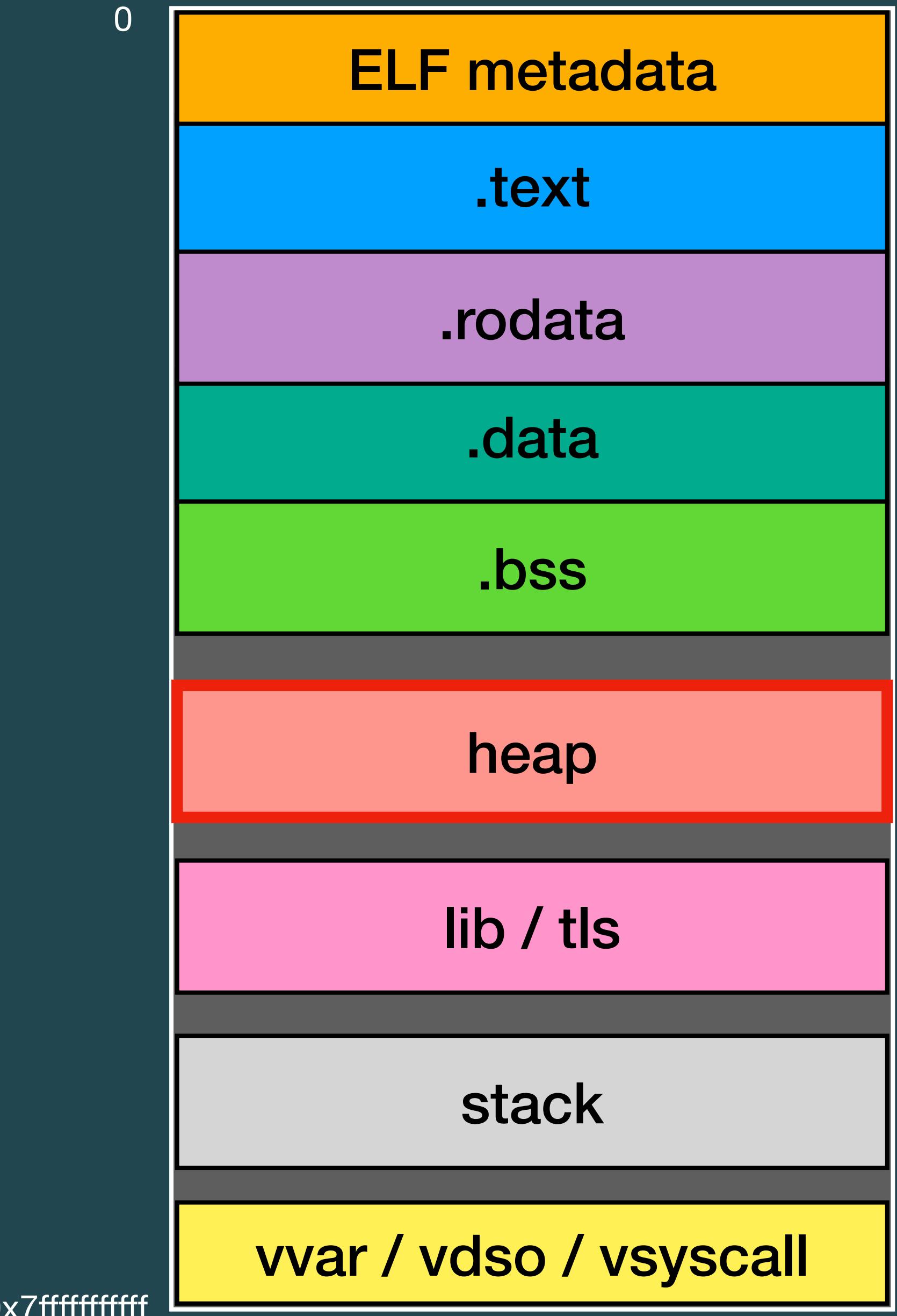


Prior knowledge

\$ Prior knowledge

Memory Layout

- ▶ **.text** - 程式碼
- ▶ **.rodata** - read-only data，如字串
- ▶ **.data** - 初始化後的變數
- ▶ **.bss** - 尚未初始化的變數
- ▶ **heap** - 動態分配的記憶體空間
- ▶ **lib** - shared library
- ▶ **tls** - thread local storage
- ▶ **stack** - 用來儲存當前 function 執行狀態的空間



\$ Prior knowledge Protection

- ▶ **PIE** - 程式碼會以相對位址的方式表示，而非絕對位址
- ▶ **NX** - .text 之外的 section 不會有執行權限
- ▶ **Canary** - 在 stack 的結尾塞入一個隨機數，return 前透過檢查是否有被修改來判斷執行是否出現問題
- ▶ **RELRO** - 分成 Full / Partial / No 三種型態，分別代表在 runtime 解析外部 function 時使用的不同機制
- ▶ **Seccomp** - 制定規則來禁止/允許呼叫特定的 syscall
- ▶ **ASLR** - 程式載入時，stack、heap 等記憶體區塊會使用隨機的位址作為 base address

\$ Prior knowledge

Bypass ASLR

- ▶ 當我們取得某個在 library 當中的位址，由於 offset 皆是固定的，因此只需要減去他在 library 當中的 offset，能得到 library **base address**，繞過 **ASLR**
- ▶ 有了 library base address，也能加上其他 function 的 offset 來取得該 function 在 library 中的位址
- ▶ 在 heap exploitation 當中較常用到的 library 變數或 function 有：
 - ⌚ __malloc_hook
 - ⌚ __free_hook
 - ⌚ system()

\$ Prior knowledge

One gadget

- ▶ library 內會有長得像是 `execve("/bin/sh", NULL, NULL)` 的程式碼，而在滿足一定的條件下可以直接 get shell，這樣的程式碼片段就稱作 one gadget
- ▶ 可以用工具 `one_gadget` 來搜尋
- ▶ 不過並不是直接執行就會有 shell，而是必須符合一些條件，如 register 的值，或是 stack layout 等等

```
$ one_gadget /lib/x86_64-linux-gnu/libc.so.6
0xe6c7e execve("/bin/sh", r15, r12)
constraints:
[r15] == NULL || r15 == NULL
[r12] == NULL || r12 == NULL

0xe6c81 execve("/bin/sh", r15, rdx)
constraints:
[r15] == NULL || r15 == NULL
[rdx] == NULL || rdx == NULL

0xe6c84 execve("/bin/sh", rsi, rdx)
constraints:
[rsi] == NULL || rsi == NULL
[rdx] == NULL || rdx == NULL
```



Heap introduction

\$ Heap introduction

Basic

- ▶ 在 compile time 程式無法確定需要多少空間，因此需要有方法能夠在 runtime 分配以及釋放記憶體，才能有效的利用記憶體
- ▶ 服務根據種類以及開發單位，會有不同的記憶體管理機制
 - ⦿ Glibc - ptmalloc
 - ⦿ Google - tcmalloc
 - ⦿ Facebook - jemalloc
 - ⦿ ...

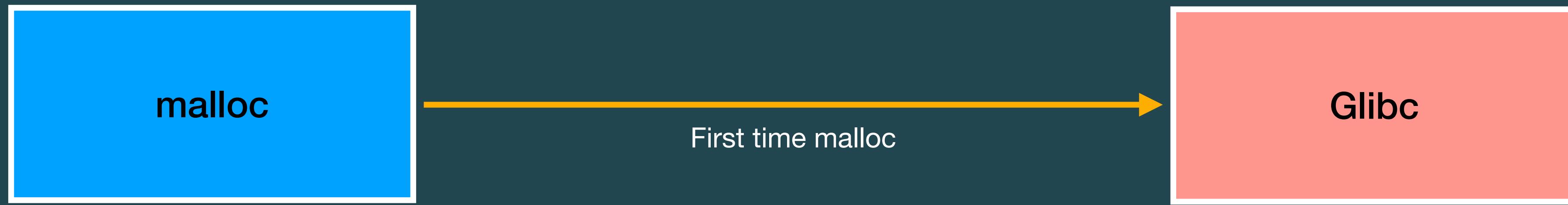
\$ Heap introduction

Basic

- ▶ 在 C 當中，最常用來分配記憶體的 function 為 `malloc`，而 `free` 則是用來釋放記憶體
- ▶ 其他與 memory allocation 相關的 function
 - ⦿ `realloc` - 更新已分配的記憶體大小
 - ⦿ `calloc` - 在回傳前清空記憶體的內容

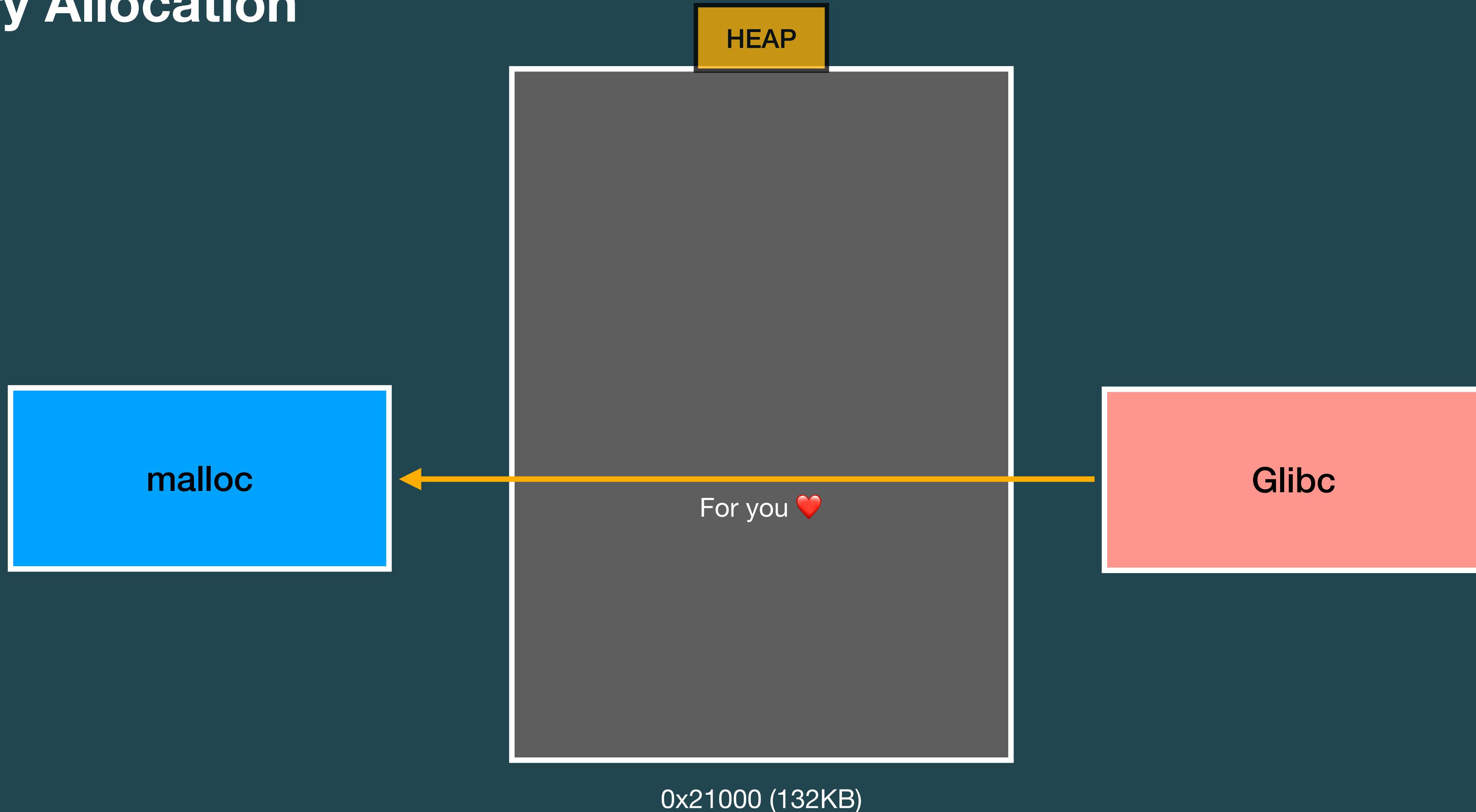
\$ Heap introduction

Memory Allocation



\$ Heap introduction

Memory Allocation



\$ Heap introduction

Memory Allocation

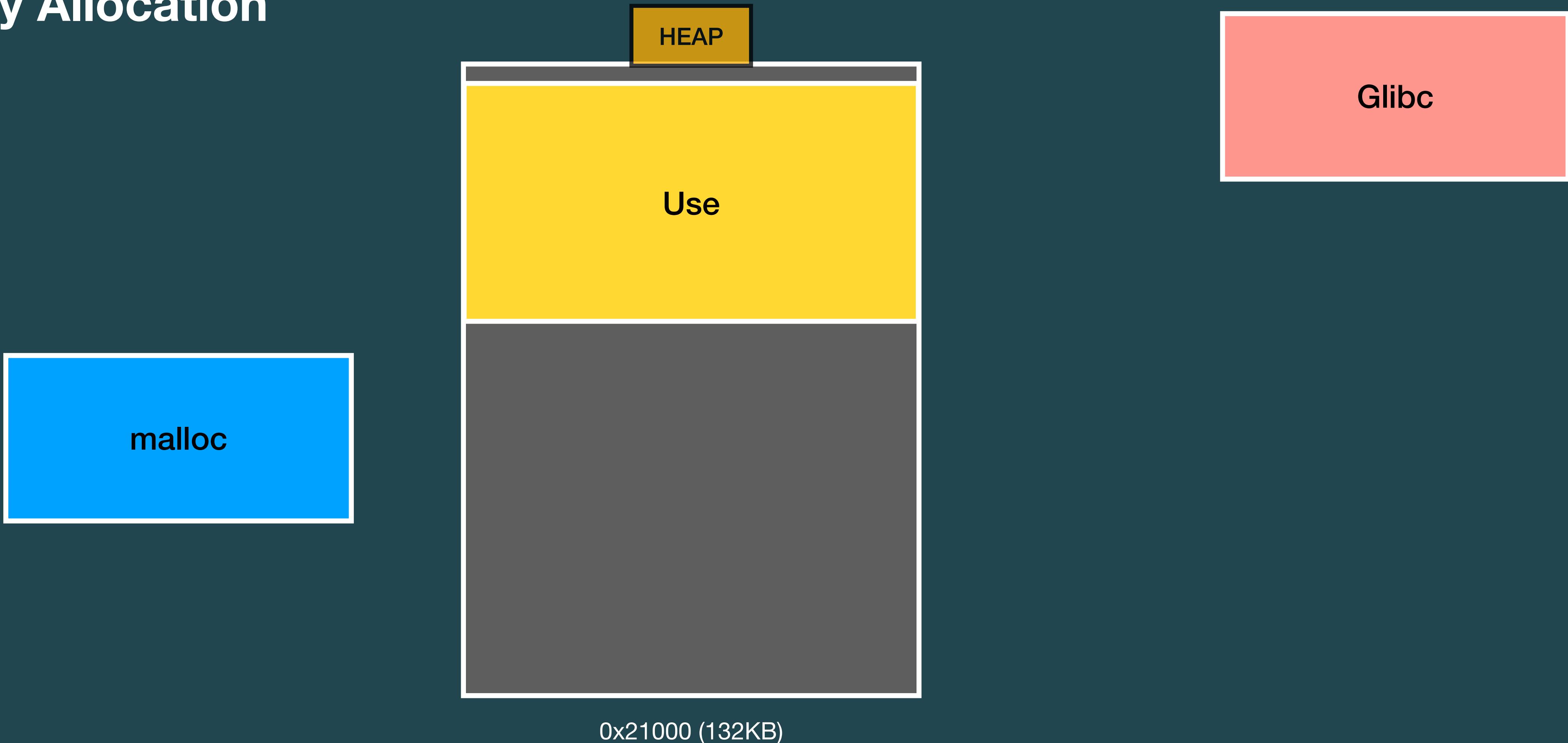
malloc



0x21000 (132KB)

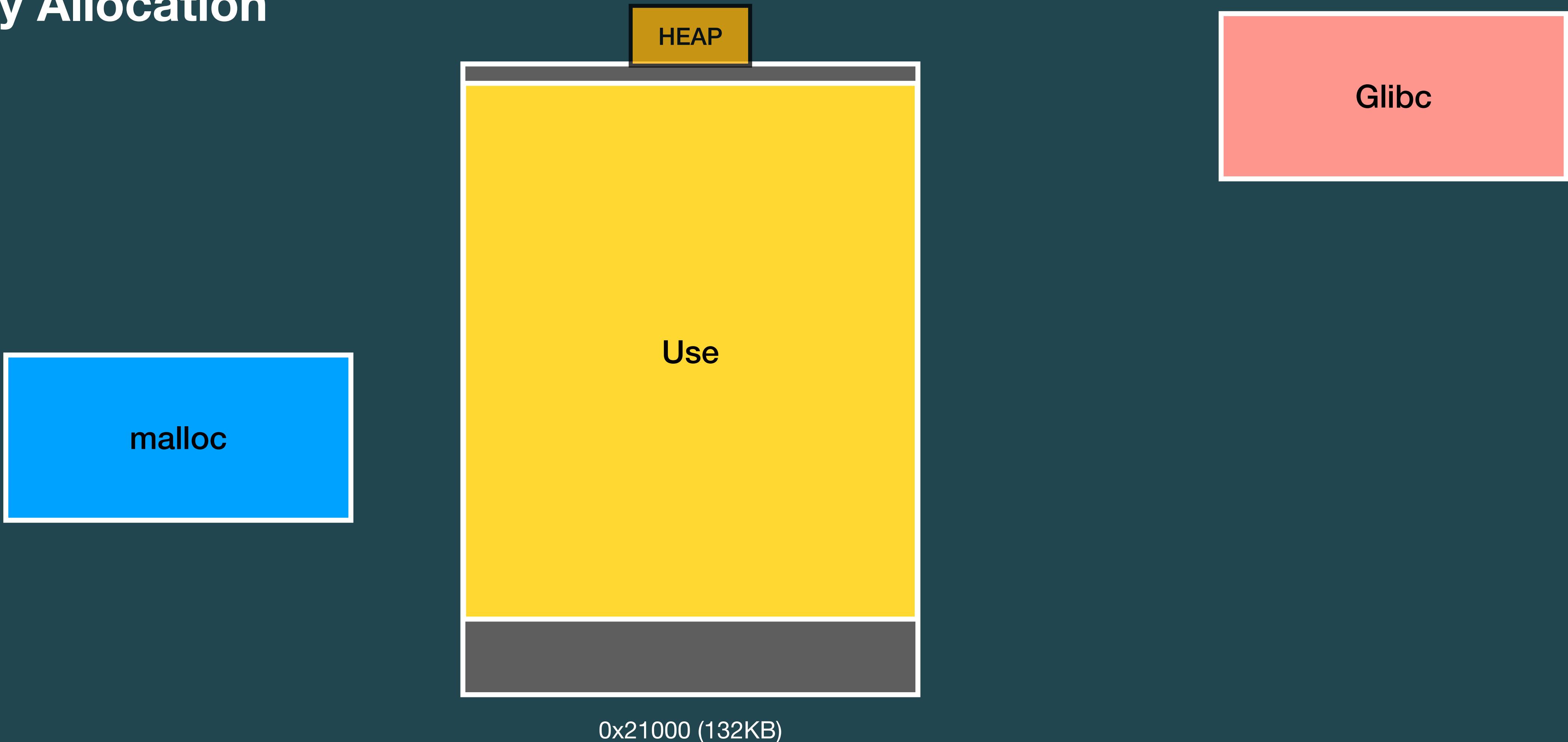
\$ Heap introduction

Memory Allocation



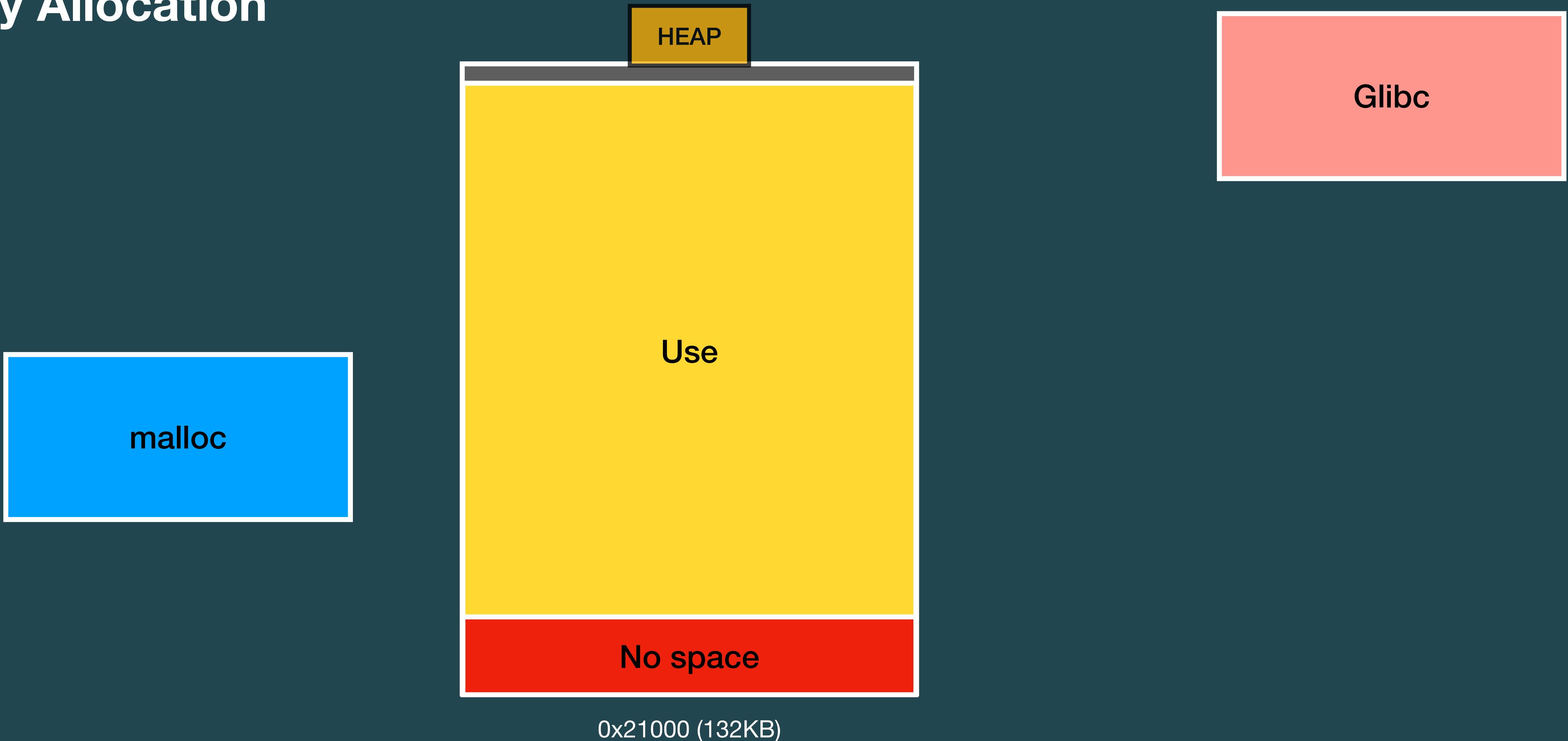
\$ Heap introduction

Memory Allocation



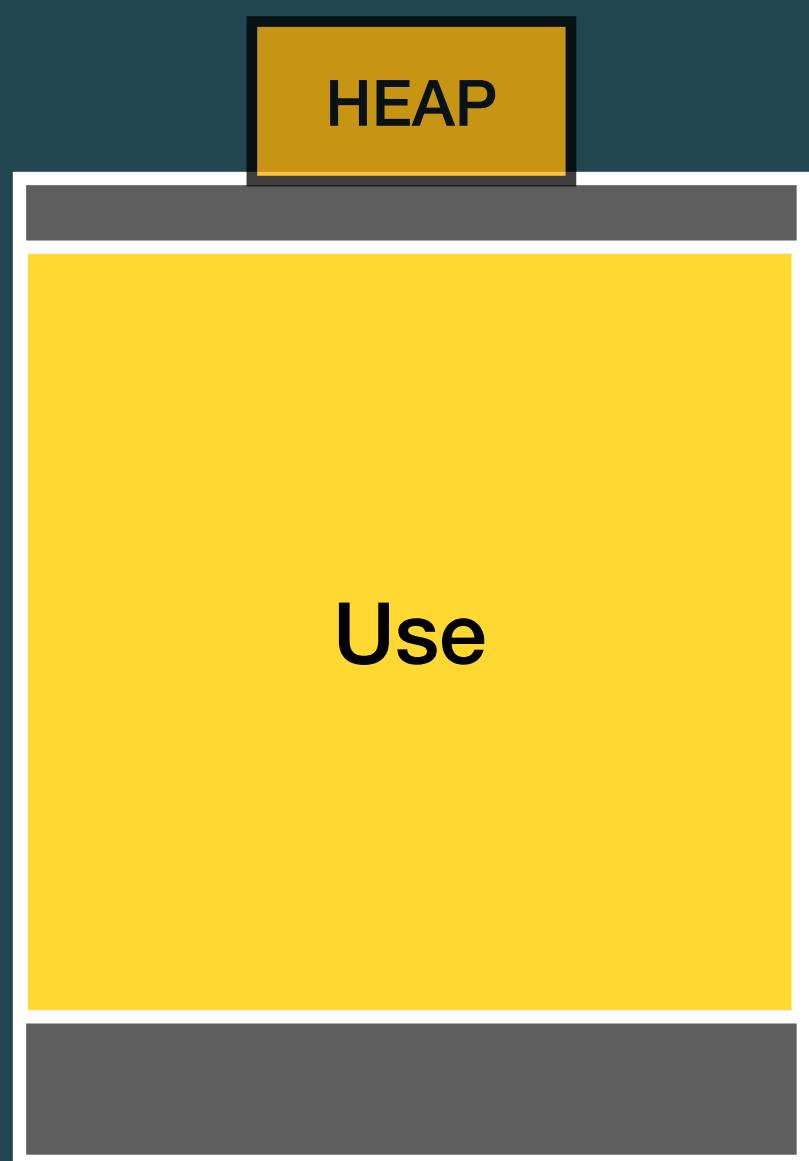
\$ Heap introduction

Memory Allocation



\$ Heap introduction

Memory Allocation



0x21000 (132KB)

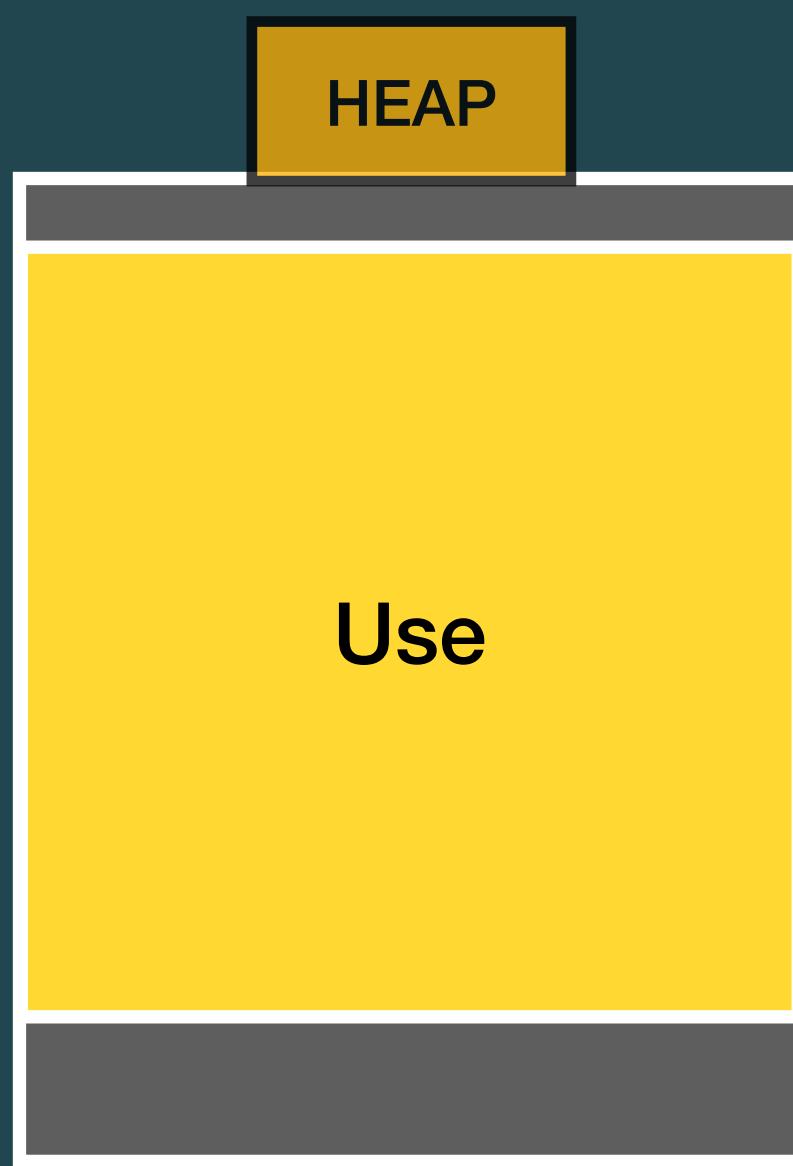


more



\$ Heap introduction

Memory Allocation



malloc

mmap

Glibc

sys_mmap

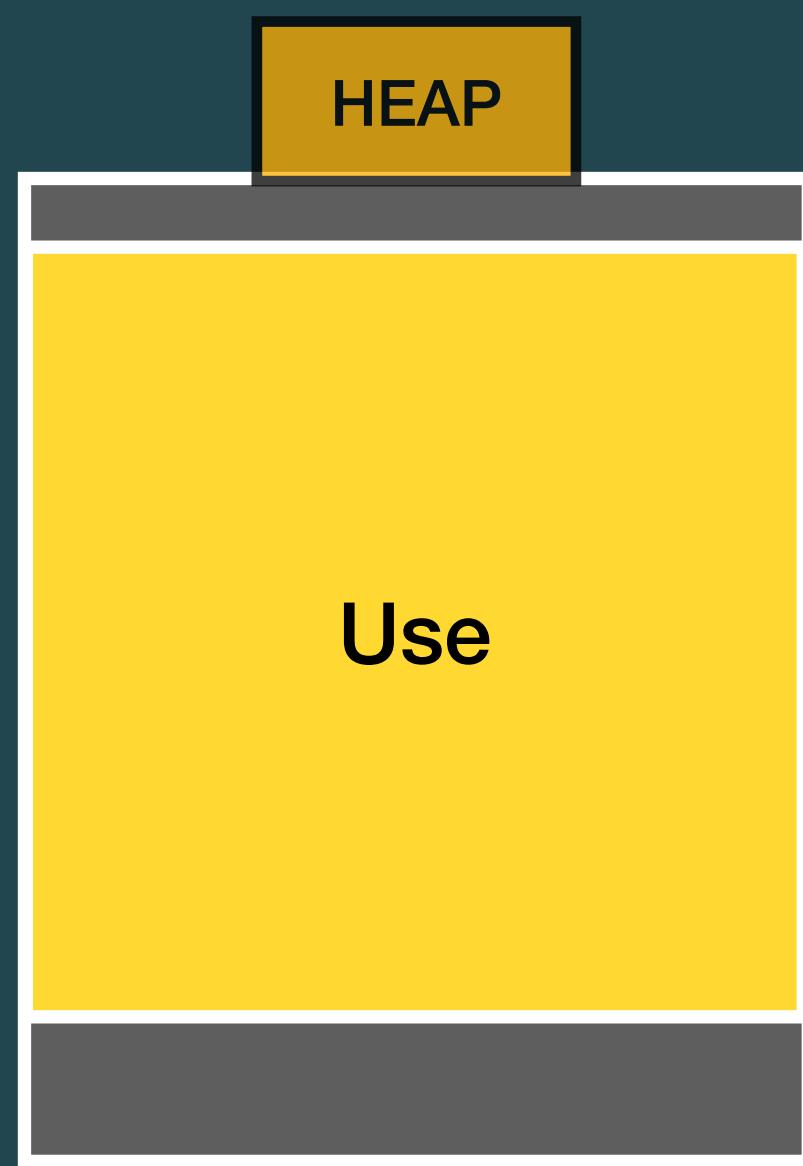
more

$\geq 0x20000$
(128 KB)

當 request size $\geq 0x20000$ ，
glibc 會分配一塊新的記憶體給你

\$ Heap introduction

Memory Allocation



malloc

brk

Glibc

sys_brk

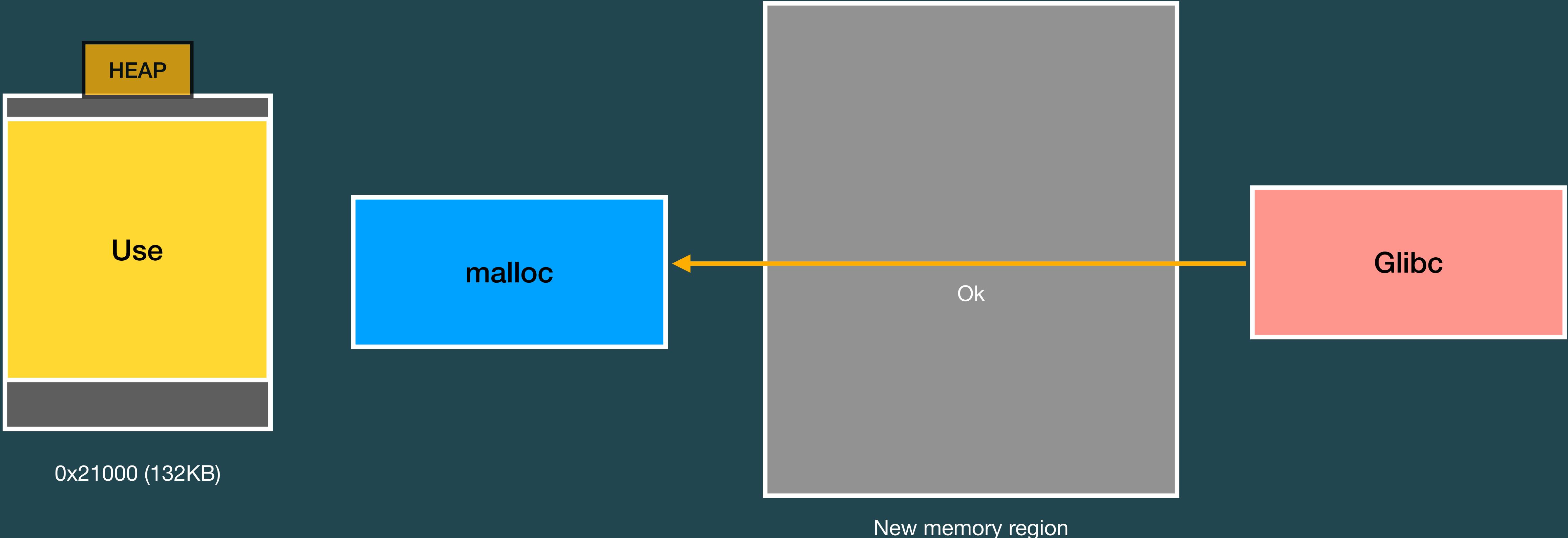
more

< 0x20000
(128 KB)

當 request size < 0x20000 ,
glibc 會擴展當前的 heap

\$ Heap introduction

Memory Allocation



\$ Heap introduction

Memory Allocation



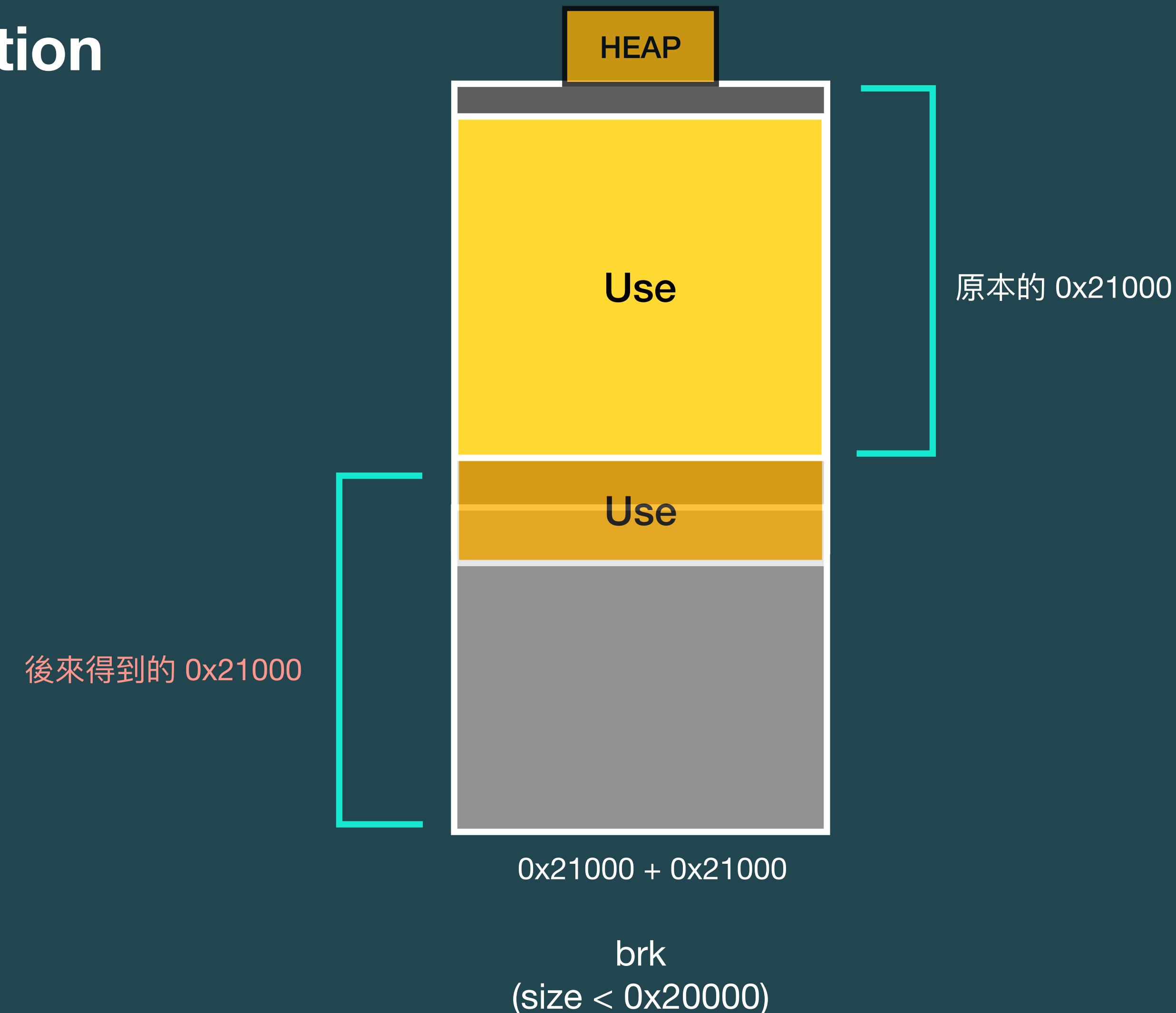
0x21000



mmap
(size \geq 0x20000)

\$ Heap introduction

Memory Allocation

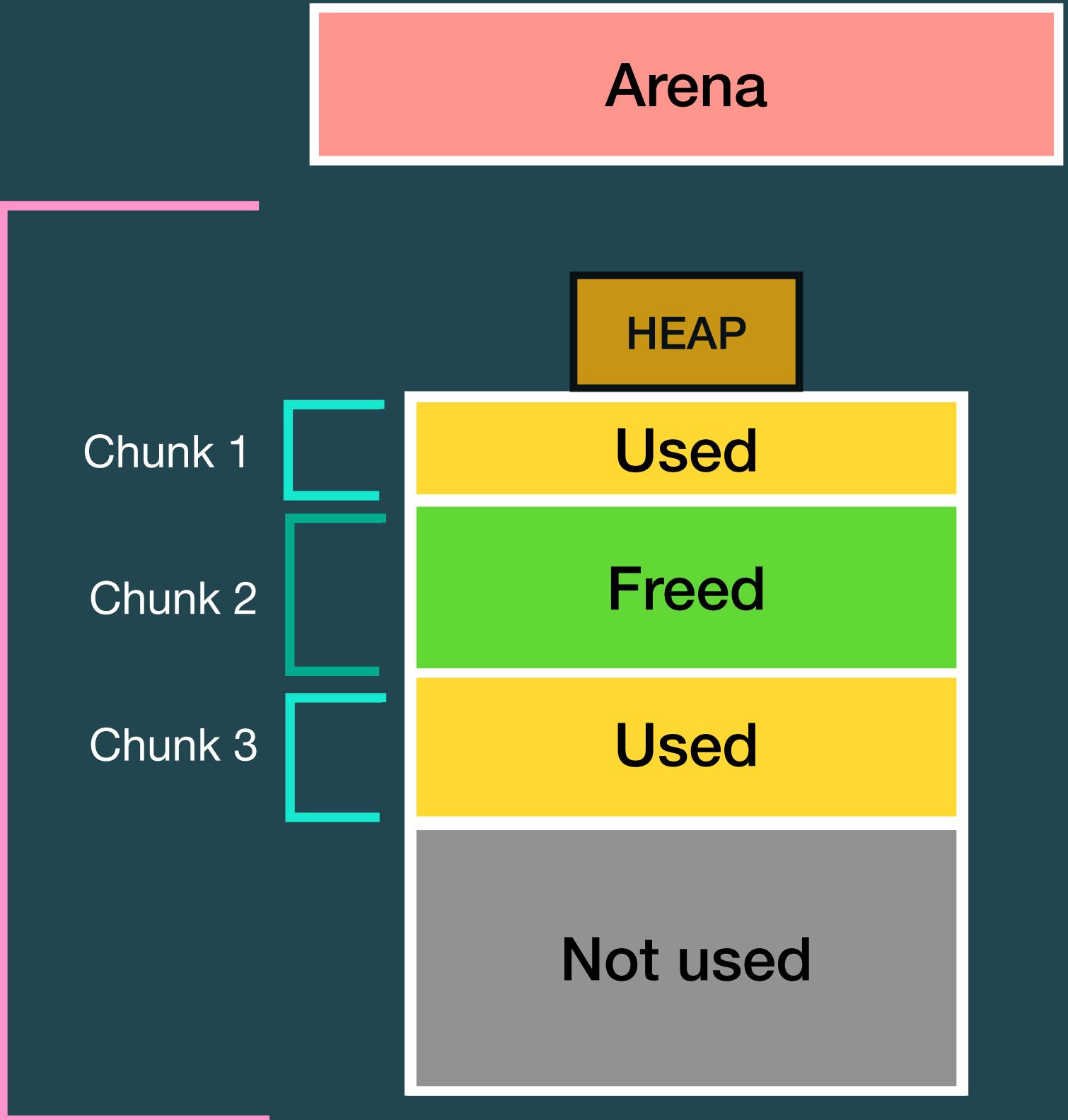


\$ Heap introduction

Data Structure

- 整個記憶體的分配機制由三個部分構成：
 - ◎ 使用者透過 malloc 取得的小塊記憶體 - chunk
 - ◎ 用來分發這些小塊記憶體的大塊記憶體 - heap
 - ◎ 用來儲存大塊記憶體的相關資訊的資料結構 - arena

哪些還沒用、
釋放掉的 chunk 位址、
...



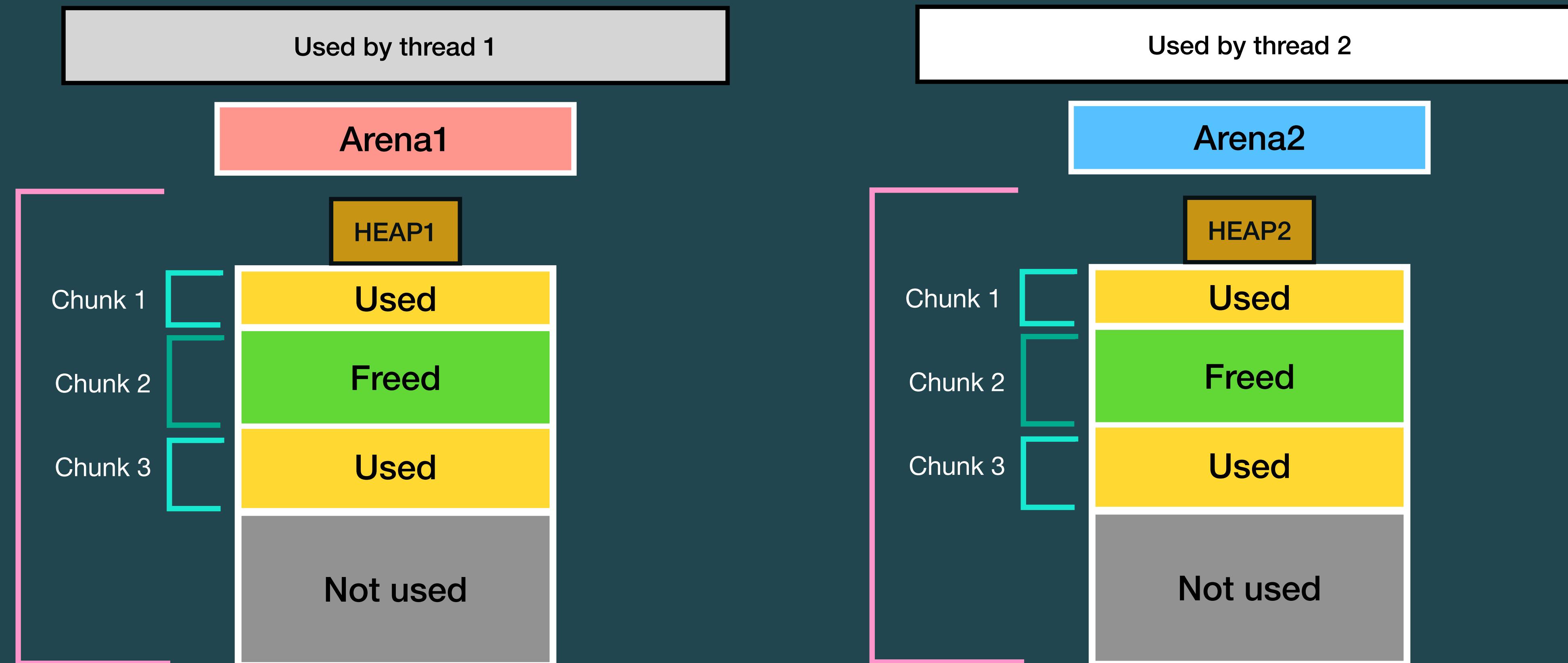
\$ Heap introduction

Data Structure

- ▶ Chunk - **struct malloc_chunk**
- ▶ Heap - **struct _heap_info**
 - ⦿ 每個 **thread** 會有一個 heap
- ▶ Arena - **struct malloc_state**
 - ⦿ 正常情況下，每個 **thread** 會有一個 arena
 - ⦿ 但是在 **thread** 的**數量太多**的情況下，多個 **thread** 會共同使用一個 arena
 - ⦿ main thread 使用的 arena 稱作 **main_arena**

\$ Heap introduction

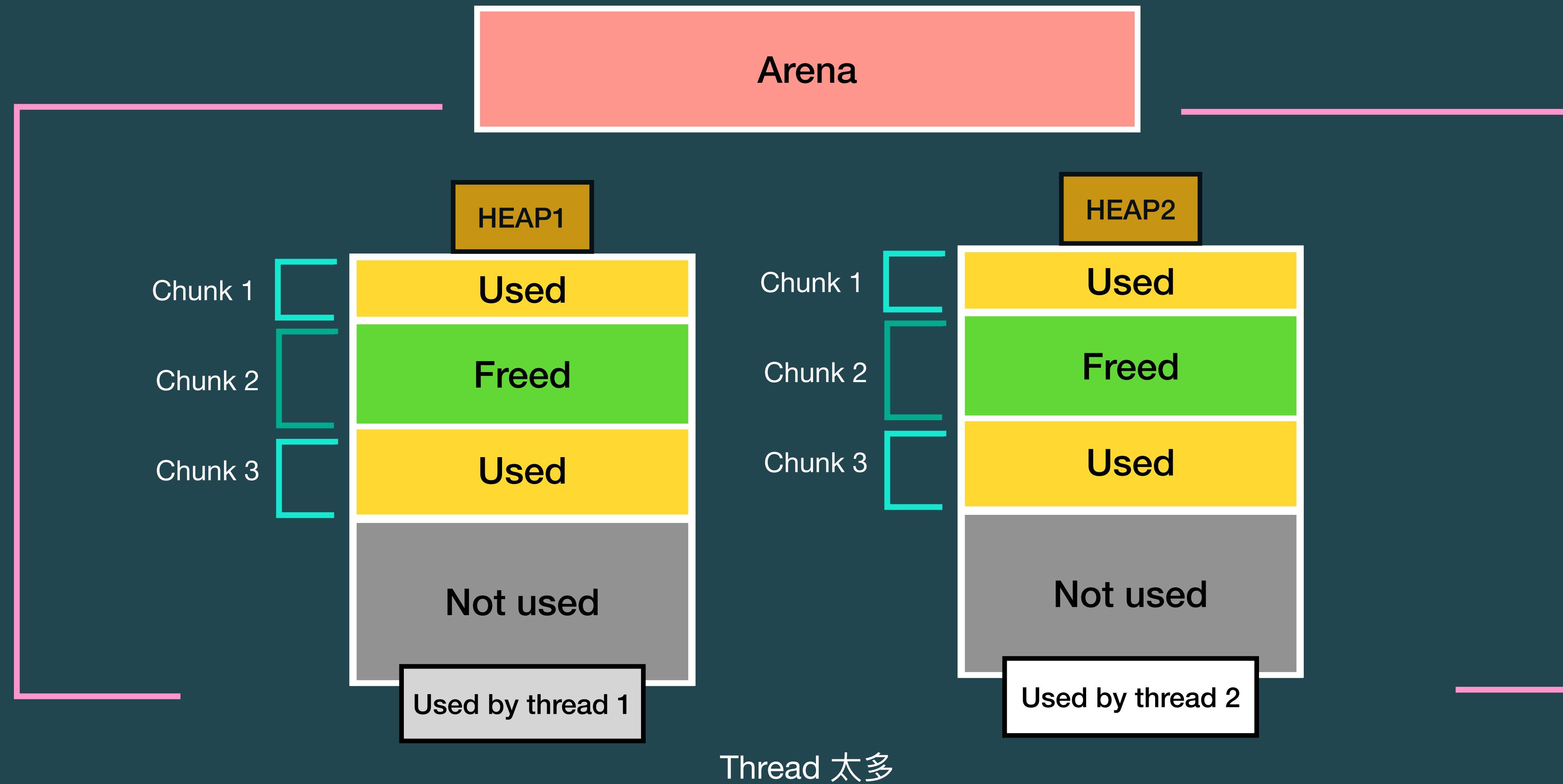
Data Structure



正常情況

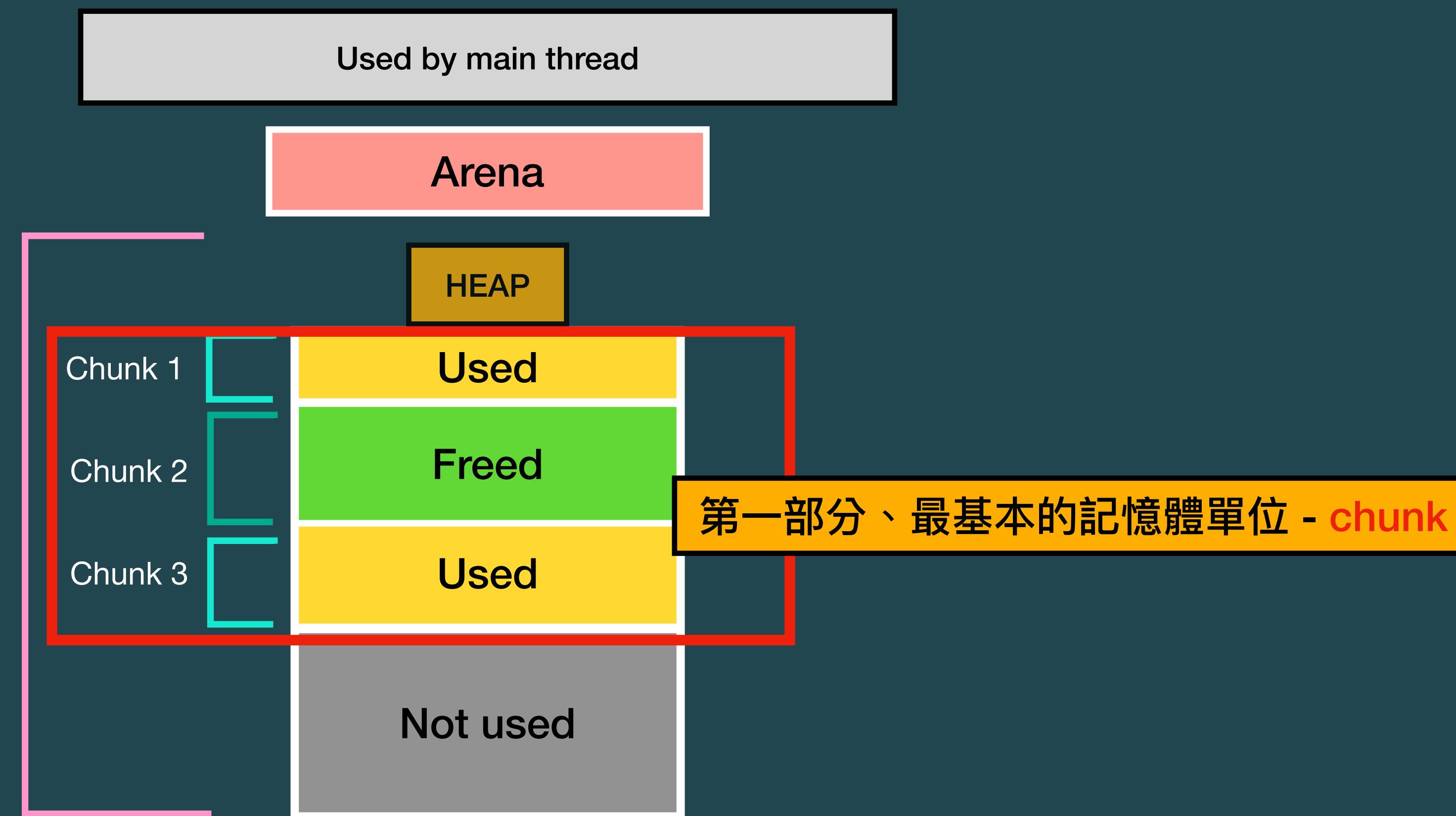
\$ Heap introduction

Data Structure



\$ Heap introduction

Data Structure



\$ Heap introduction

Data Structure - Chunk

- ▶ **Chunk** 為動態記憶體分配的基本單位，最小為 **0x20 bytes**
- ▶ 會對齊 **0x10 bytes**，也就是大小會是 **0x20, 0x30, 0x40 ...**
- ▶ 正在使用中的 chunk 稱作 **allocated** chunk
- ▶ 已被釋放的 chunk 稱作 **freed** chunk
 - ⦿ 在釋放時，chunk 會根據情況放在不一樣的回收區，稱作 **bin**
 - ⦿ 後續再請求記憶體時，就會從這些 **bin** 取出 chunk 紿使用者用
 - ⦿ Chunk 的結構與在 bin 當中的 freed chunk 的連接方式有很大的關係

\$ Heap introduction

Data Structure - Chunk



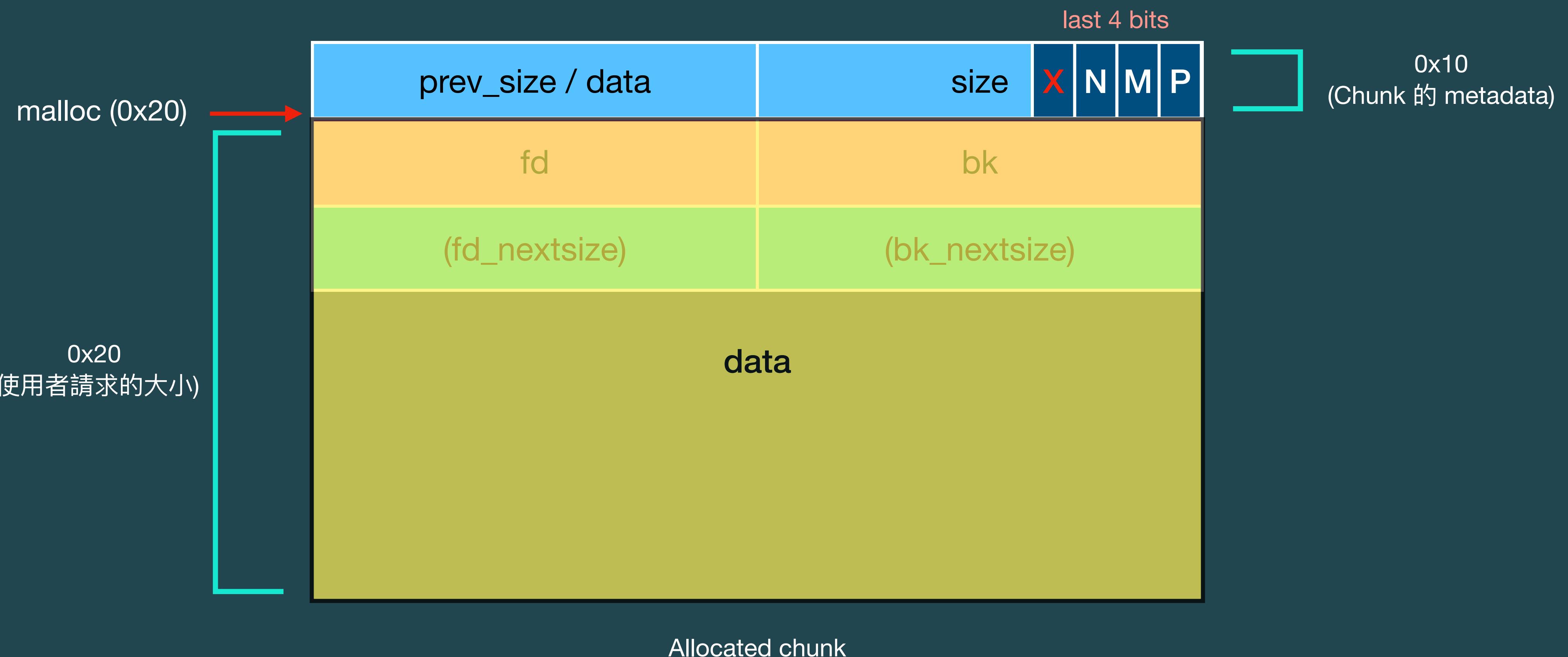
The screenshot shows a terminal window with the title "u1f383@u1f383:/". The code displayed is the definition of the `malloc_chunk` structure from the glibc source code. The structure includes fields for previous and current sizes, forward and back pointers, and next size fields for large bin chunks.

```
struct malloc_chunk {  
  
    INTERNAL_SIZE_T      mchunk_prev_size; /* 如果前一個為 freed chunk，儲存其大小 */  
    INTERNAL_SIZE_T      mchunk_size;        /* 儲存當前 chunk 的大小 */  
  
    /* —————— 以下只有 freed chunk 才會使用 —————— */  
    struct malloc_chunk* fd;                 /* forward */  
    struct malloc_chunk* bk;                 /* back */  
  
    /* — 以下只有 large bin chunk 才會使用 — */  
    struct malloc_chunk* fd_nextsize;  
    struct malloc_chunk* bk_nextsize;  
};
```

<https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c#L1048>

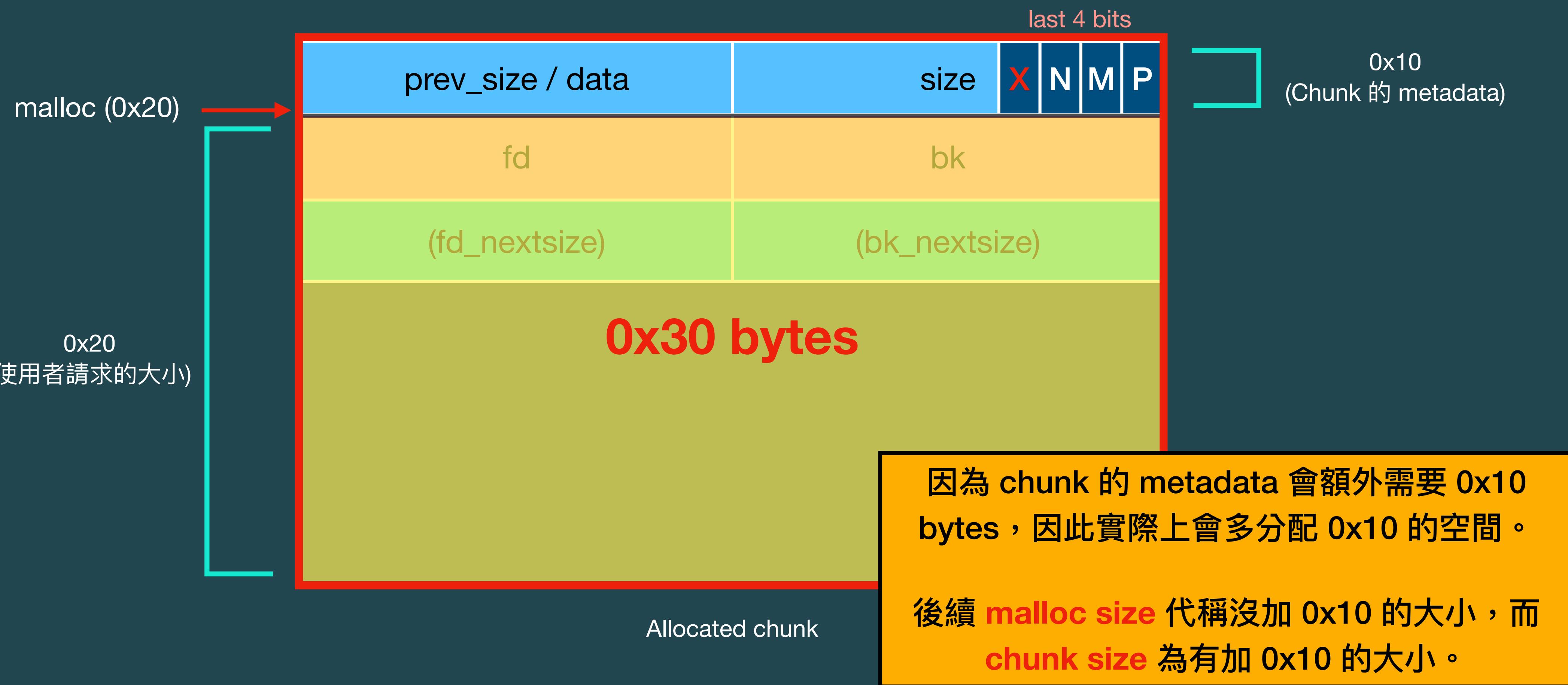
\$ Heap introduction

Data Structure - Chunk



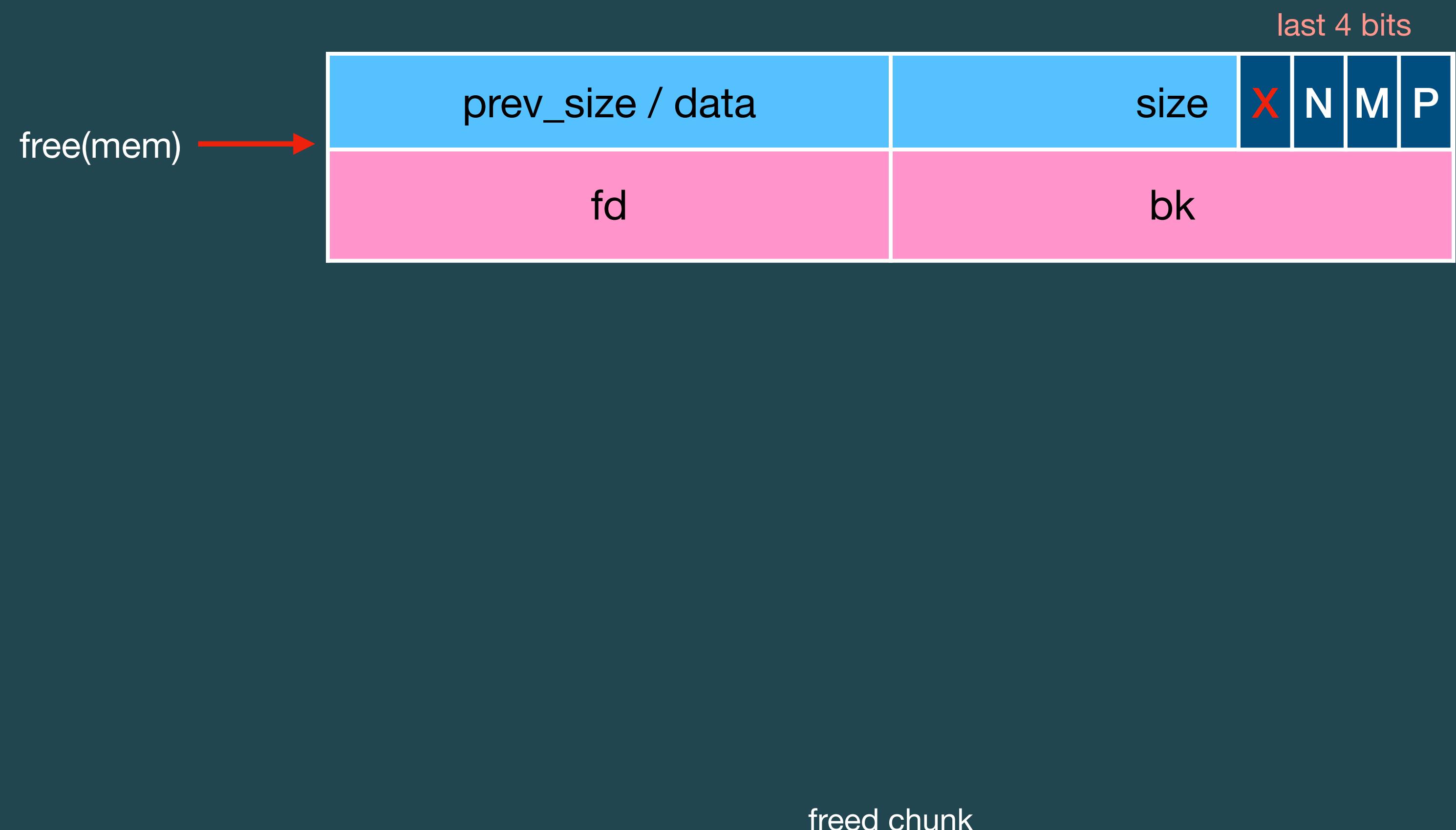
\$ Heap introduction

Data Structure - Chunk



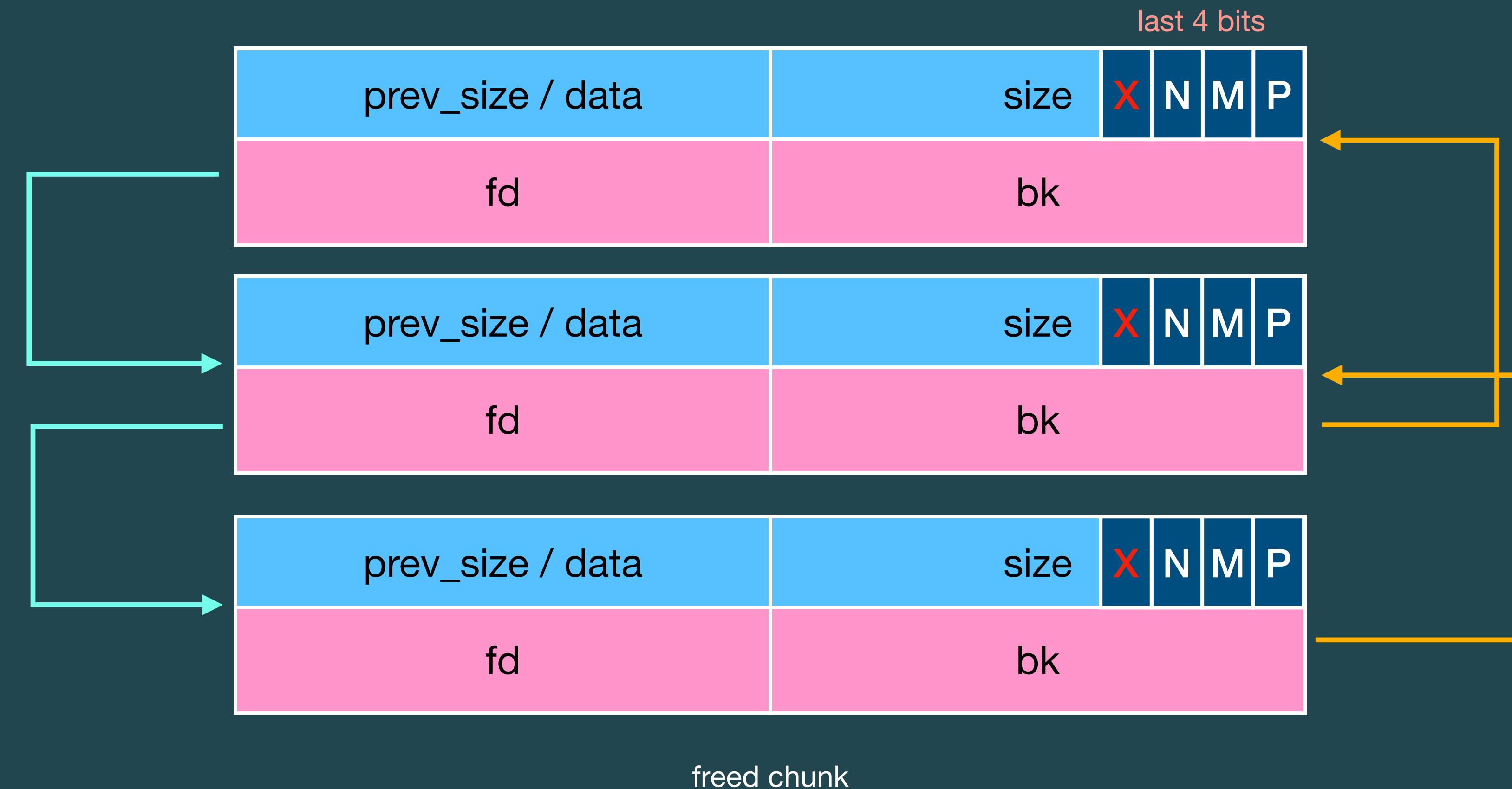
\$ Heap introduction

Data Structure - Chunk



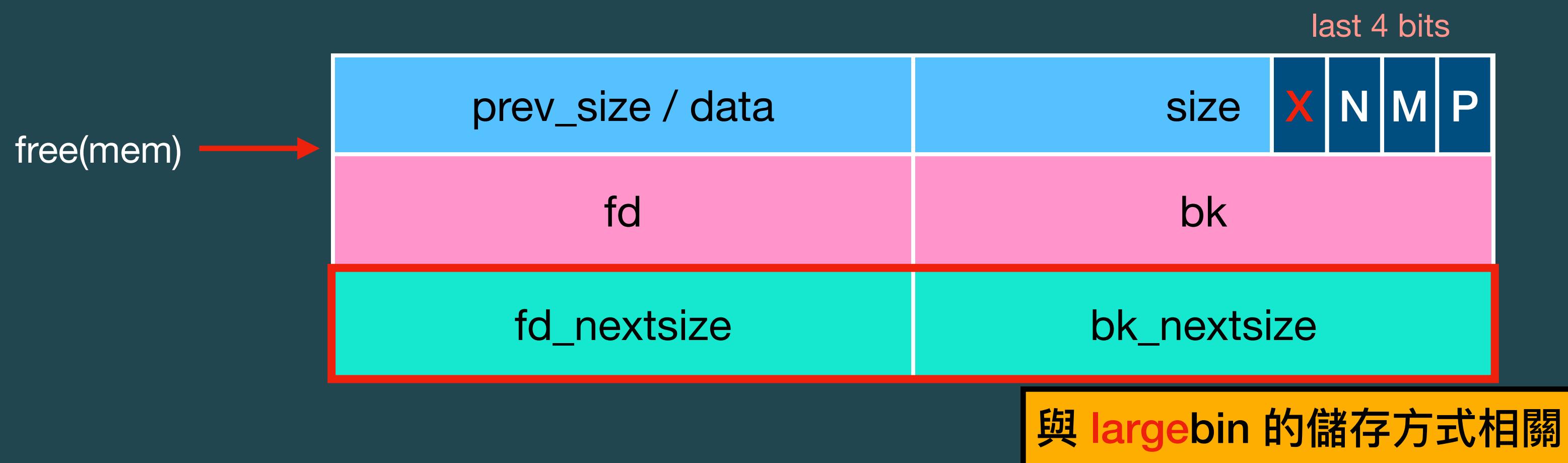
\$ Heap introduction

Data Structure - Chunk



\$ Heap introduction

Data Structure - Chunk

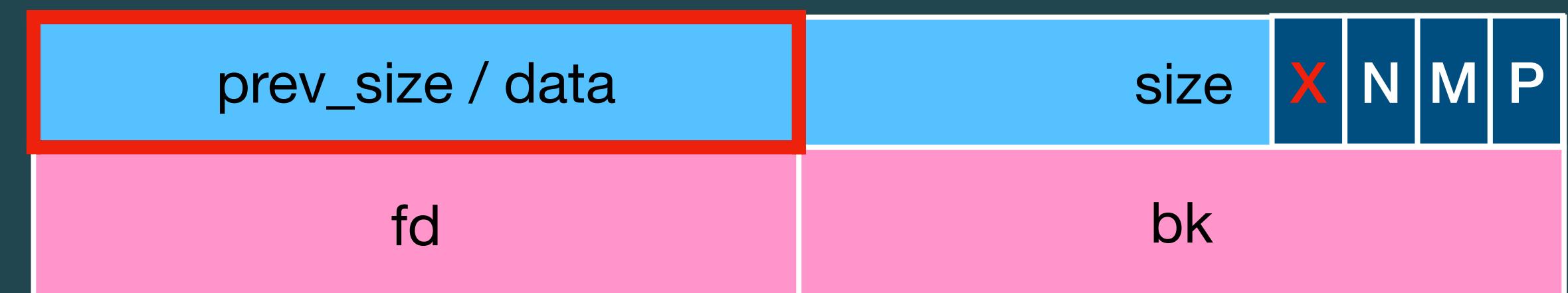


large bin chunk

\$ Heap introduction

Data Structure - Chunk

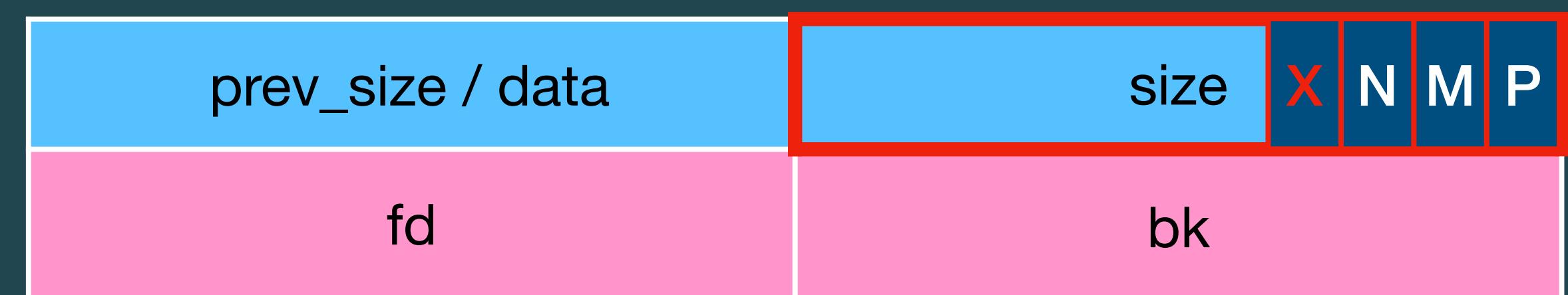
- ▶ **prev_size / data** - 看前一塊 chunk 是否正在使用
 - ⦿ 如果前一塊為 **freed chunk** - 存前一塊 chunk 的 chunk size (**prev_size**)
 - ⦿ 如果前一塊為 **allocated chunk** - 用來給前一塊 chunk 存資料 (**data**)
- ▶ 因為 chunk 可以多使用下個 chunk 的 **data** 欄位，因此在請求記憶體時能夠多分配 8 bytes
 - ⦿ E.g. malloc size 為 0x28 時會拿到 chunk size 為 0x30 的 chunk → 0x28 (request) - 0x8 (data) + 0x10 (header) — 對齊 0x10 → 0x30
 - ⦿ 0x29 則會拿到 chunk size 0x40 的 chunk
0x31 — 對齊 0x10 → 0x40



\$ Heap introduction

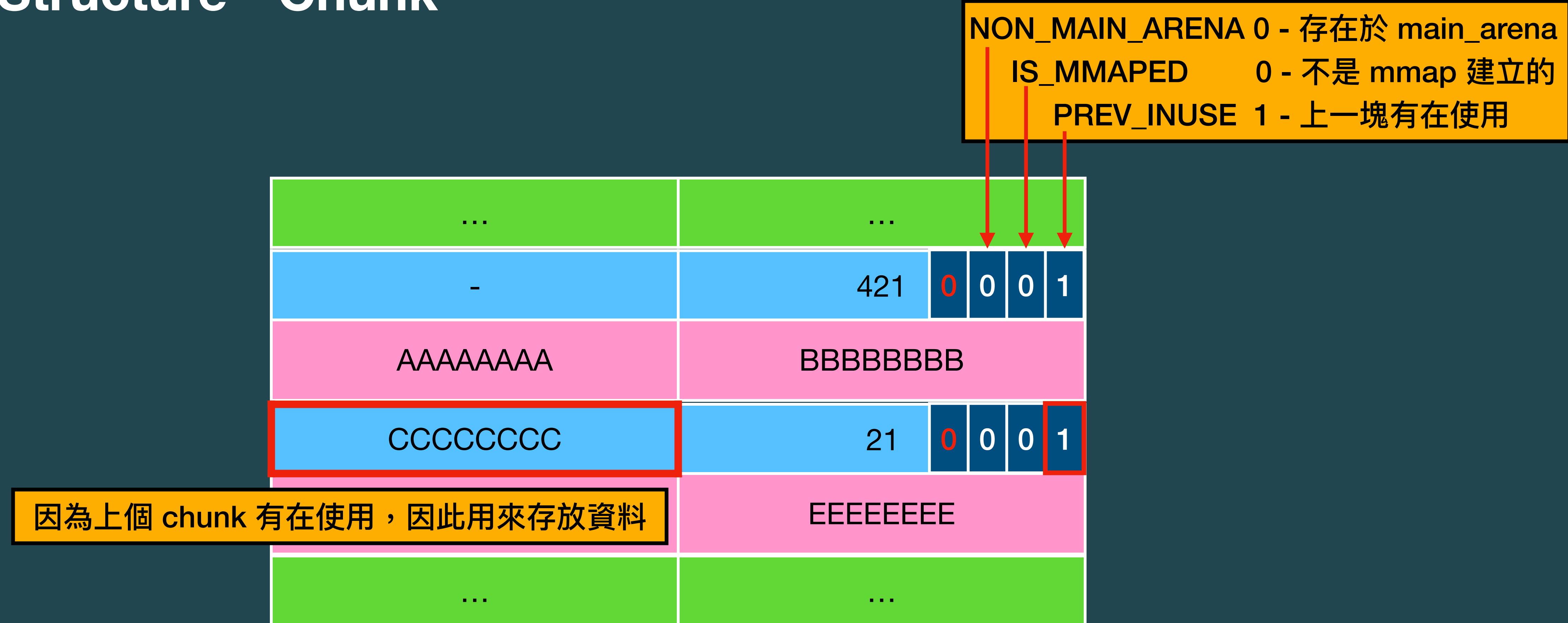
Data Structure - Chunk

- ▶ **size** - 存 chunk 的 chunk size，並且 size 會對齊 0x10 bytes (0b10000)
- ▶ 由於對齊，末四個 bits 會沒用到 (0b10000)，因此做為 chunk 的 metadata
 - ⦿ **X** - 沒用
 - ⦿ **N (NON_MAIN_arena)** - chunk 是否存在於 main_arena
 - ⦿ **M (IS_MAPPED)** - chunk 是否透過 `mmap` 建立的
 - ⦿ **P (PREV_INUSE)** - 前一塊是否正在使用



\$ Heap introduction

Data Structure - Chunk



\$ Heap introduction

Data Structure - Chunk

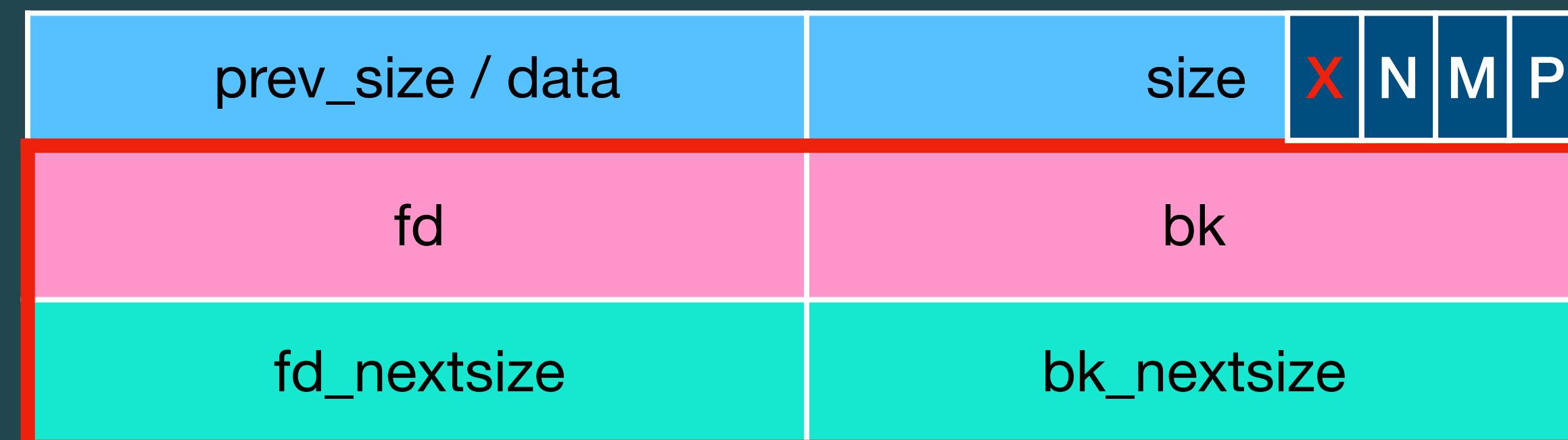
NON_MAIN_arena 0 - 存在於 main_arena
IS_MMAPED 0 - 不是 mmap 建立的
PREV_INUSE 1 - 上一塊有在使用



\$ Heap introduction

Data Structure - Chunk

- ▶ **fd** - 指向同個 bin 中的下一塊 freed chunk (forward)
- ▶ **bk** - 指向同個 bin 中的上一塊 freed chunk (back)
- ▶ **fd_nextsize** - 在 large bin 的 freed chunk 指向下一個大小的 freed chunk
- ▶ **bk_nextsize** - 在 large bin 的 freed chunk 指向上一個大小的 freed chunk



\$ Heap introduction

Data Structure - Chunk

- ▶ bin 用來紀錄 freed chunk，而根據 chunk 的大小與 glibc 的優化機制，一共分成 5 種不同的 bin

名稱	Chunk size	使用方式	優先度	補充
Tcache bin	0x20 ~ 0x410	FILO (stack)	最高 (1)	被 free 後，並不會 unset 下個 chunk 的 PREV_INUSE bit
Fastbin	0x20 ~ 0x80	FILO (stack)	2	被 free 後，並不會 unset 下個 chunk 的 PREV_INUSE bit
Small bin	0x20 ~ 0x3f0	FIFO (queue)	3	-
Large bin	>= 0x400	FIFO (queue)	最低 (5)	-
Unsorted bin	>= 0x90	-	4	當對應大小的 tcache 滿時才會進來

\$ Heap introduction

Data Structure - Chunk

- ▶ **Tcache (bin)** - 剛被釋放的 chunk 的快取空間
- ▶ **Fastbin** - 儲存大小較小的 chunk
- ▶ **Small bin** - 儲存大小中等的 chunk
- ▶ **Large bin** - 儲存大小較大的 chunk
- ▶ **Unsorted bin** - 剛被釋放的 chunk 的暫存區
- ▶ 在 tcache / fastbin / smallbin 當中，每隔 0x10 大小會有一個 subbin
 - ⦿ E.g. fastbin 的 chunk size 為 0x20~0x80，因此會有 0x20、0x30、...、0x80 subbin

\$ Heap introduction

Data Structure - Tcache

- ▶ Tcache 存放 chunk size 為 $0x20 \sim 0x410$ 的 chunk，使用方式為 FILO
- ▶ 每個 subbin 最多只能存放 7 個 chunk，彼此用單向的 linked list 相連
- ▶ 優先度最高，在對應大小的 tcache 滿後 (存放 7 個) 才會改用其他 bin

\$ Heap introduction

Data Structure - Tcache

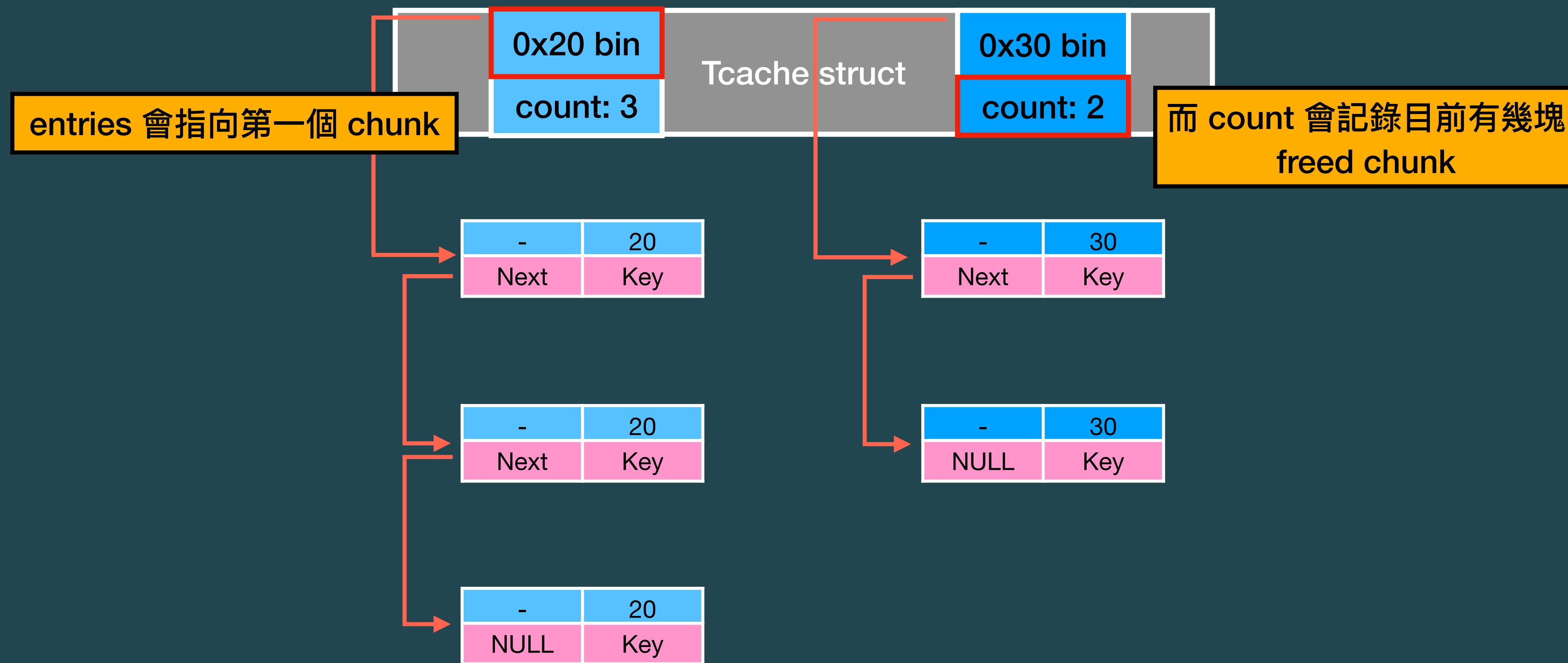
- ▶ `struct tcache_entry` 結構特別用來描述在 tcache 當中的 chunk
- ▶ `struct tcache_perthread_struct` 結構用來記錄 subbin 的第一塊 tcache chunk 位址，以及目前 subbin 共有多少個 tcache chunk
- ⌚ heap 初始化時就會被分配，大小為 0x290

```
typedef struct tcache_entry
{
    struct tcache_entry *next;
    struct tcache_perthread_struct *key;
} tcache_entry;
```

```
typedef struct tcache_perthread_struct
{
    /* TCACHE_MAX_BINS 大小為 64 */
    uint16_t counts[TCACHE_MAX_BINS];
    tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;
```

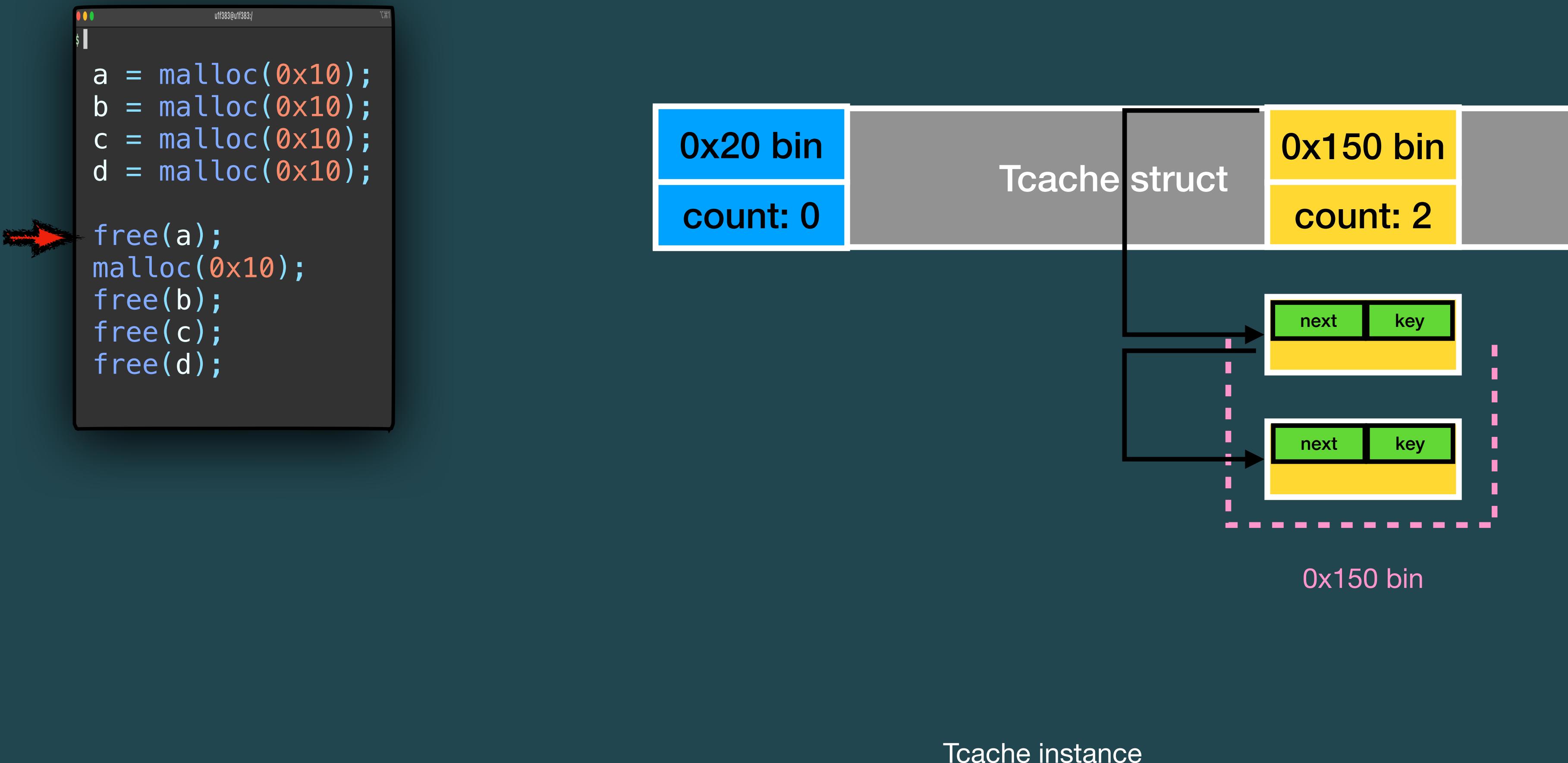
\$ Heap introduction

Data Structure - Tcache



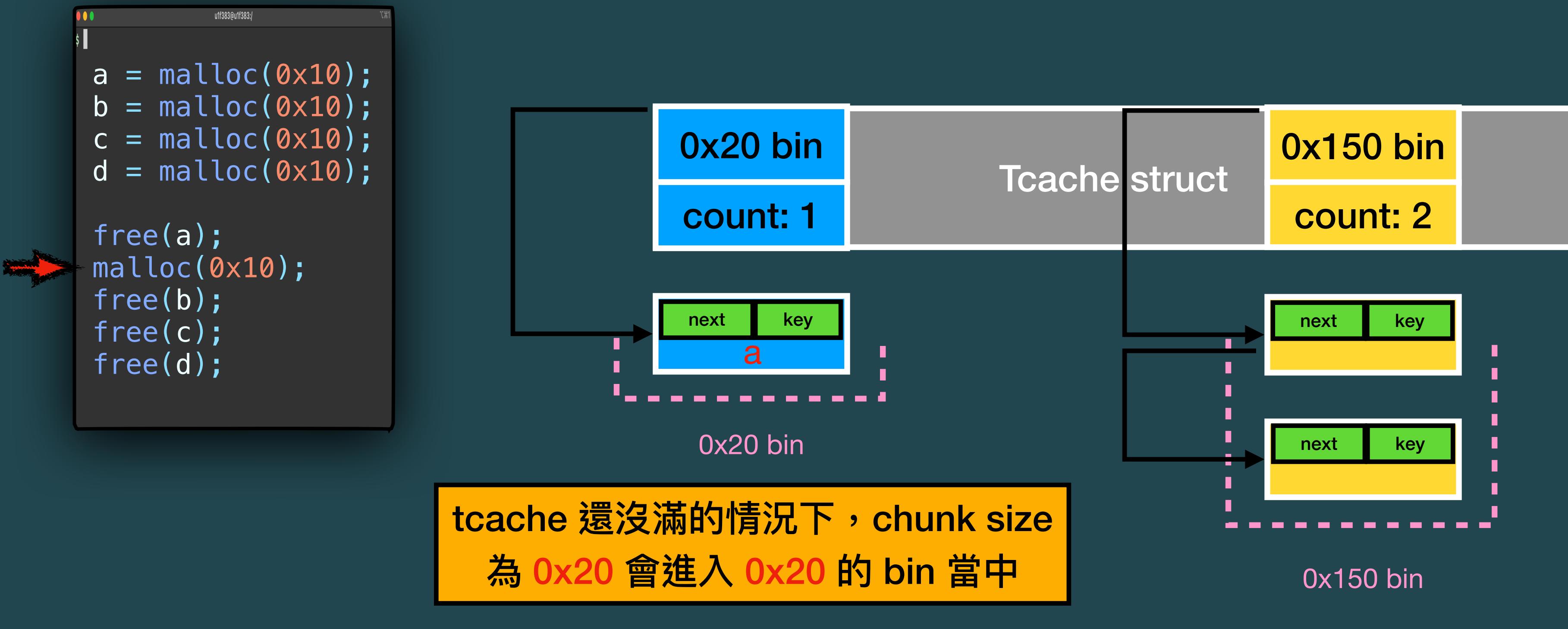
\$ Heap introduction

Data Structure - Tcache



\$ Heap introduction

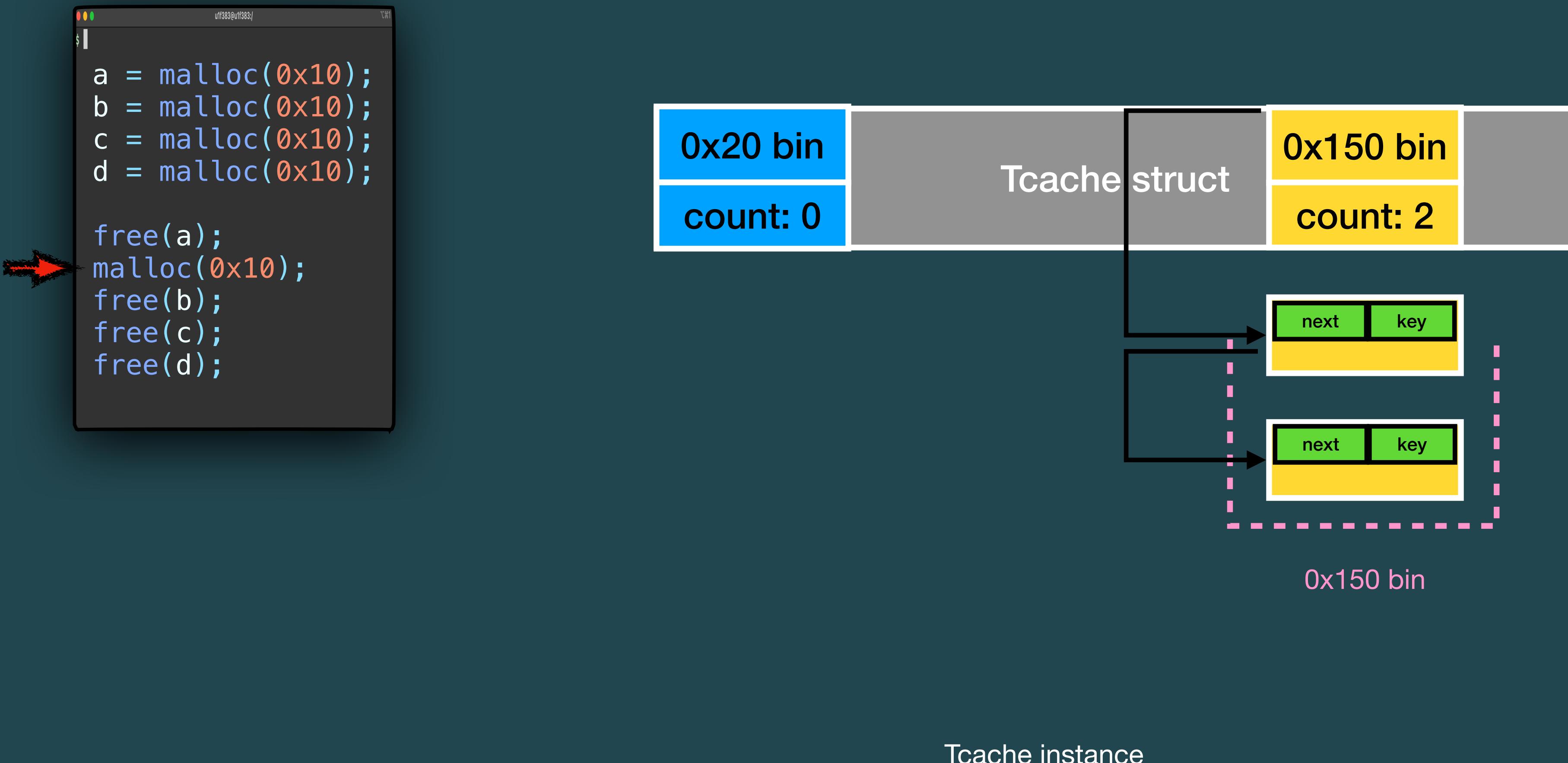
Data Structure - Tcache



Tcache instance

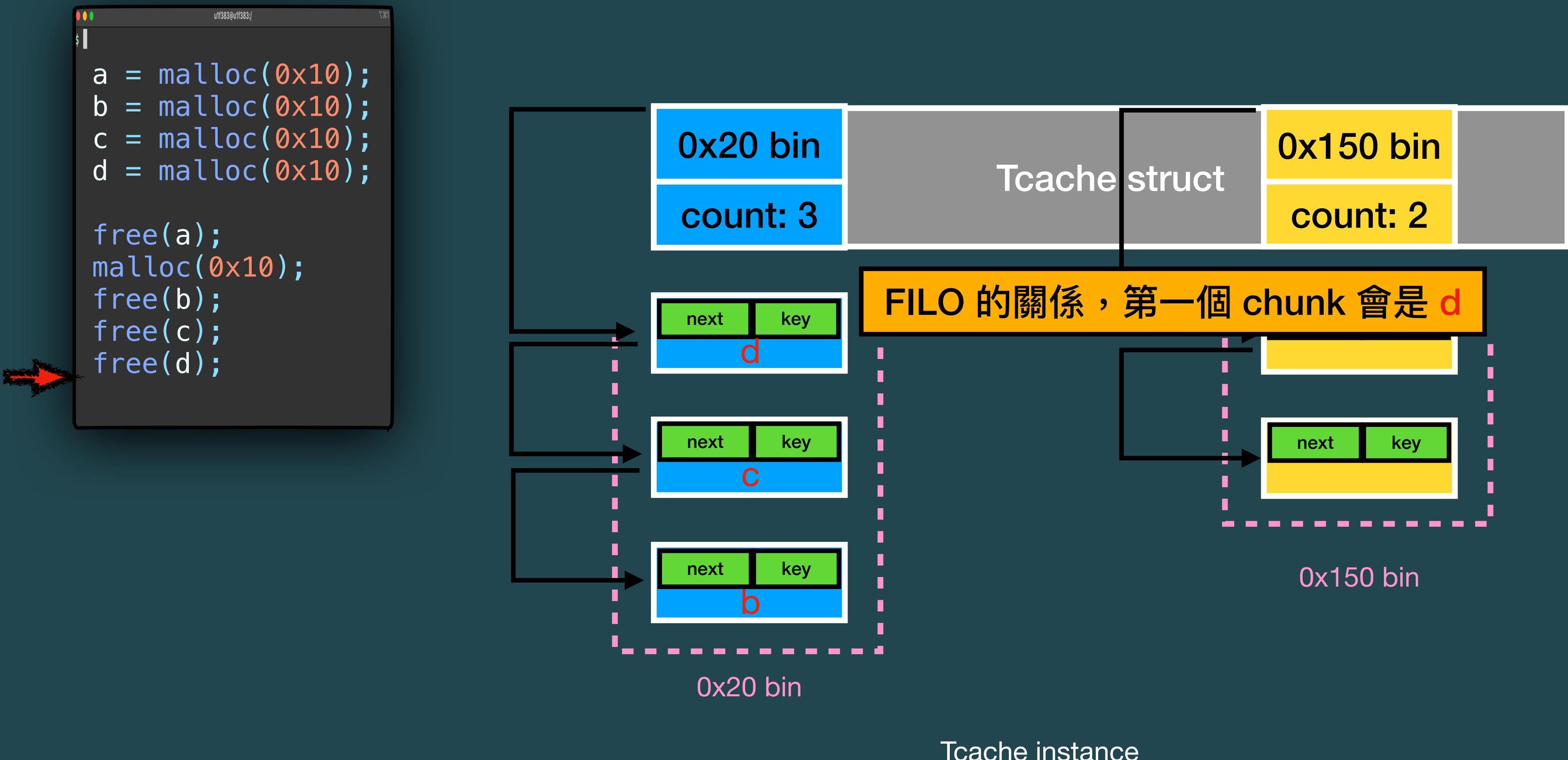
\$ Heap introduction

Data Structure - Tcache



\$ Heap introduction

Data Structure - Tcache

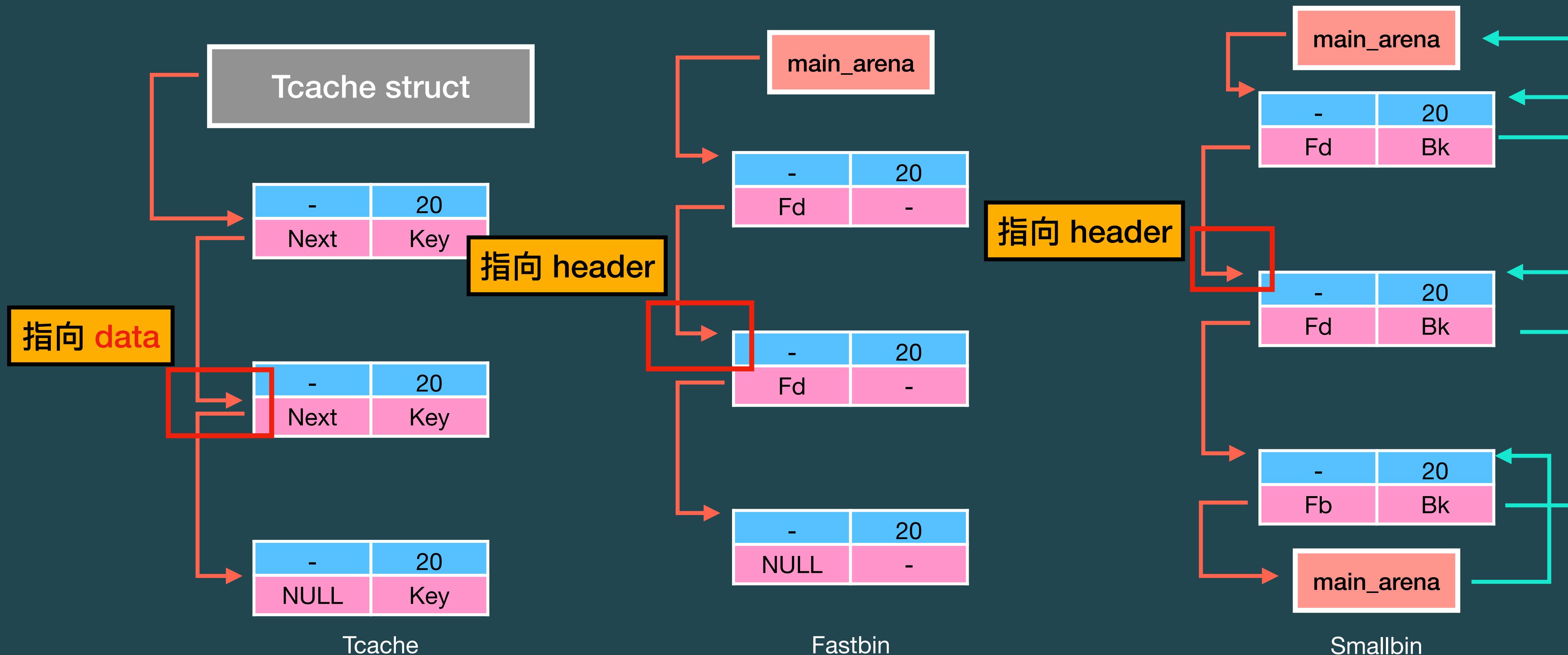


Tcache instance

\$ Heap introduction

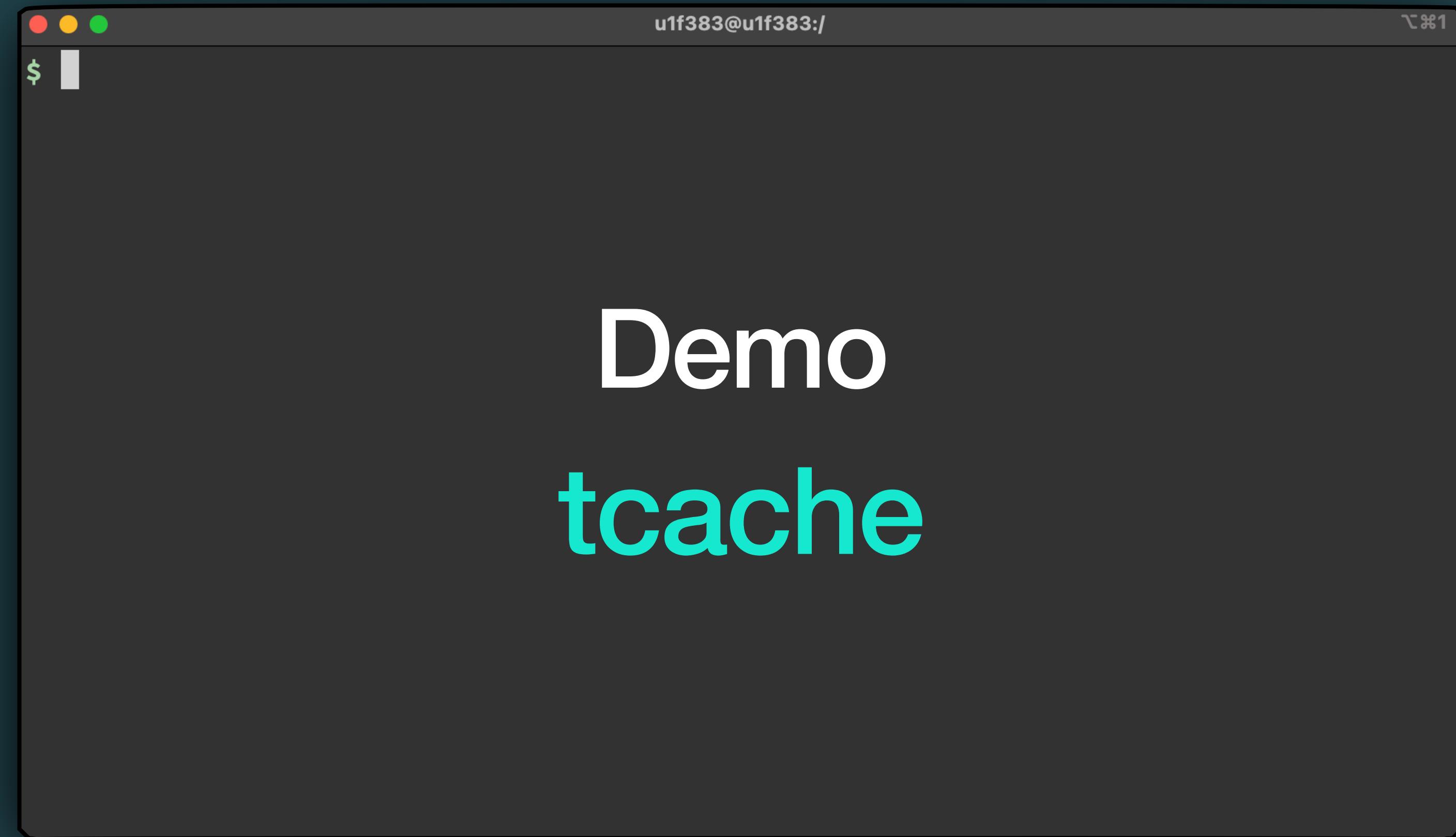
Data Structure - Tcache

- ▶ Tcache 的 fd 不像其他種類的 chunk 指向 header，而是指向 data



\$ Heap introduction

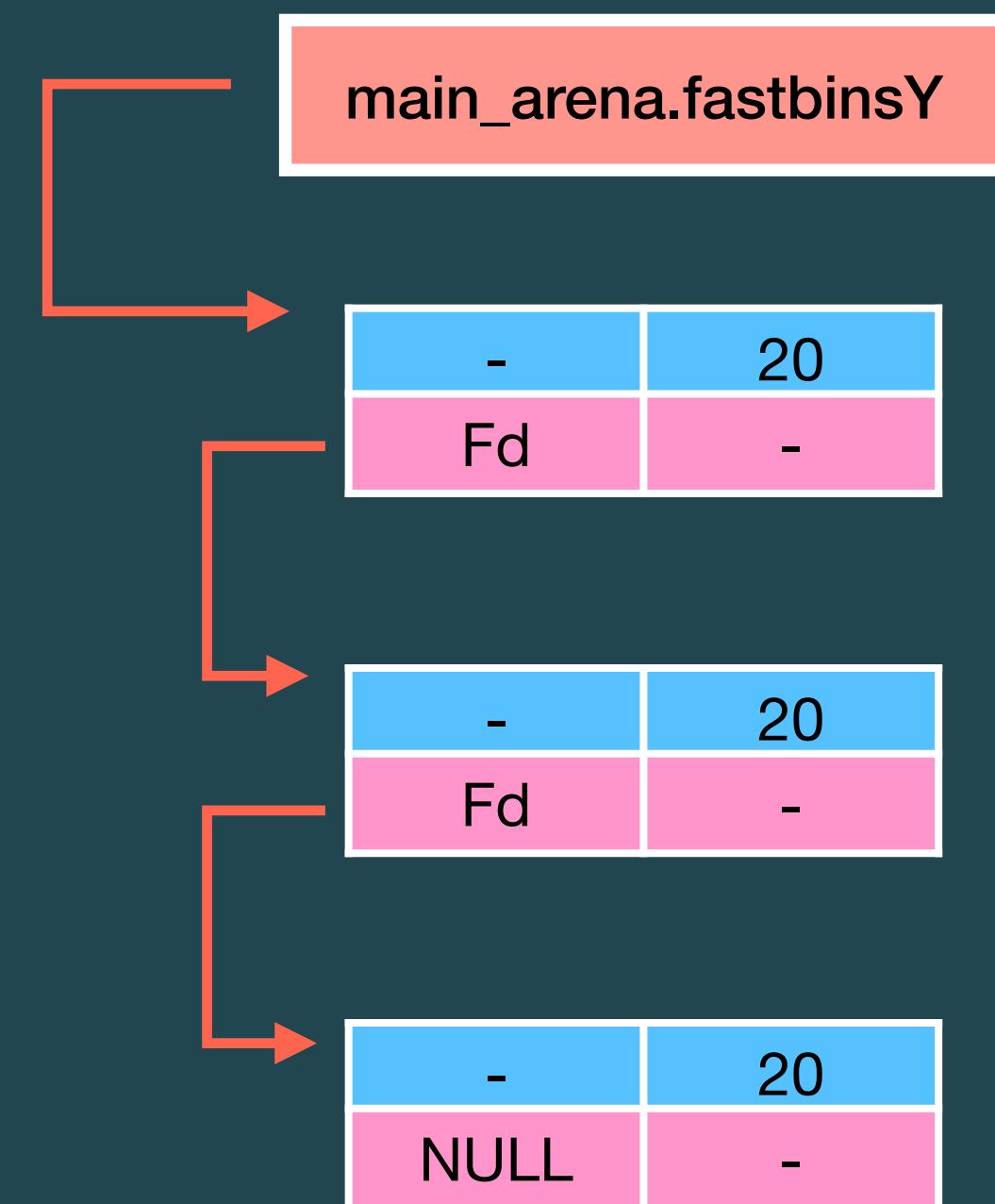
Data Structure - Tcache



\$ Heap introduction

Data Structure - Fastbin

- ▶ Fastbin 存放 chunk size 為 $0x20 \sim 0x80$ 的 chunk，並且數量沒有限制
- ▶ 使用方式與 tcache 相同，為 FILO
- ▶ 當 tcache 滿時，如果 free 大小為 $0x20 \sim 0x80$ 的 chunk，則會直接進入 fastbin

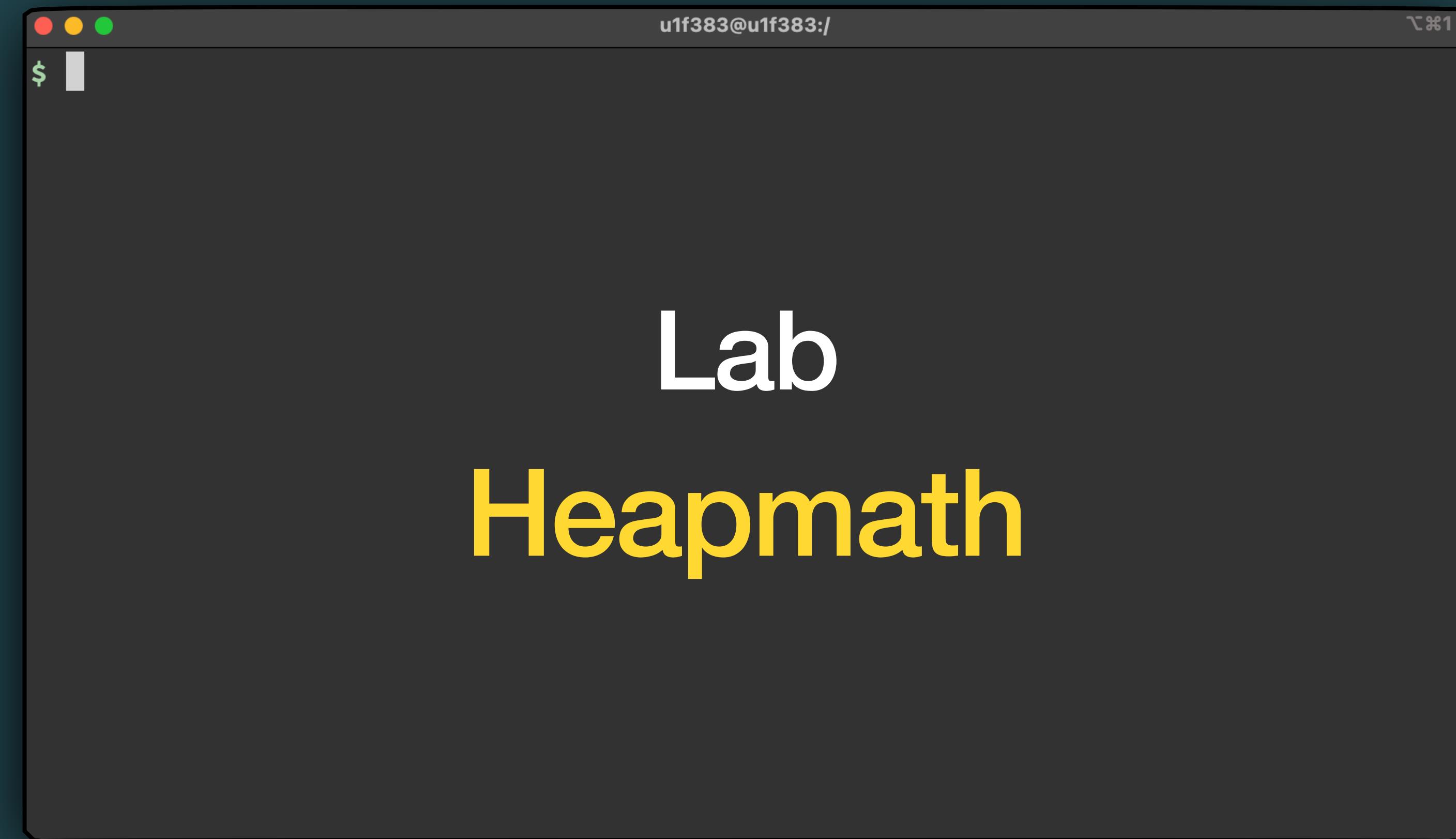


\$ Heap introduction

Data Structure - Fastbin



\$ Heap introduction



\$ Heap introduction

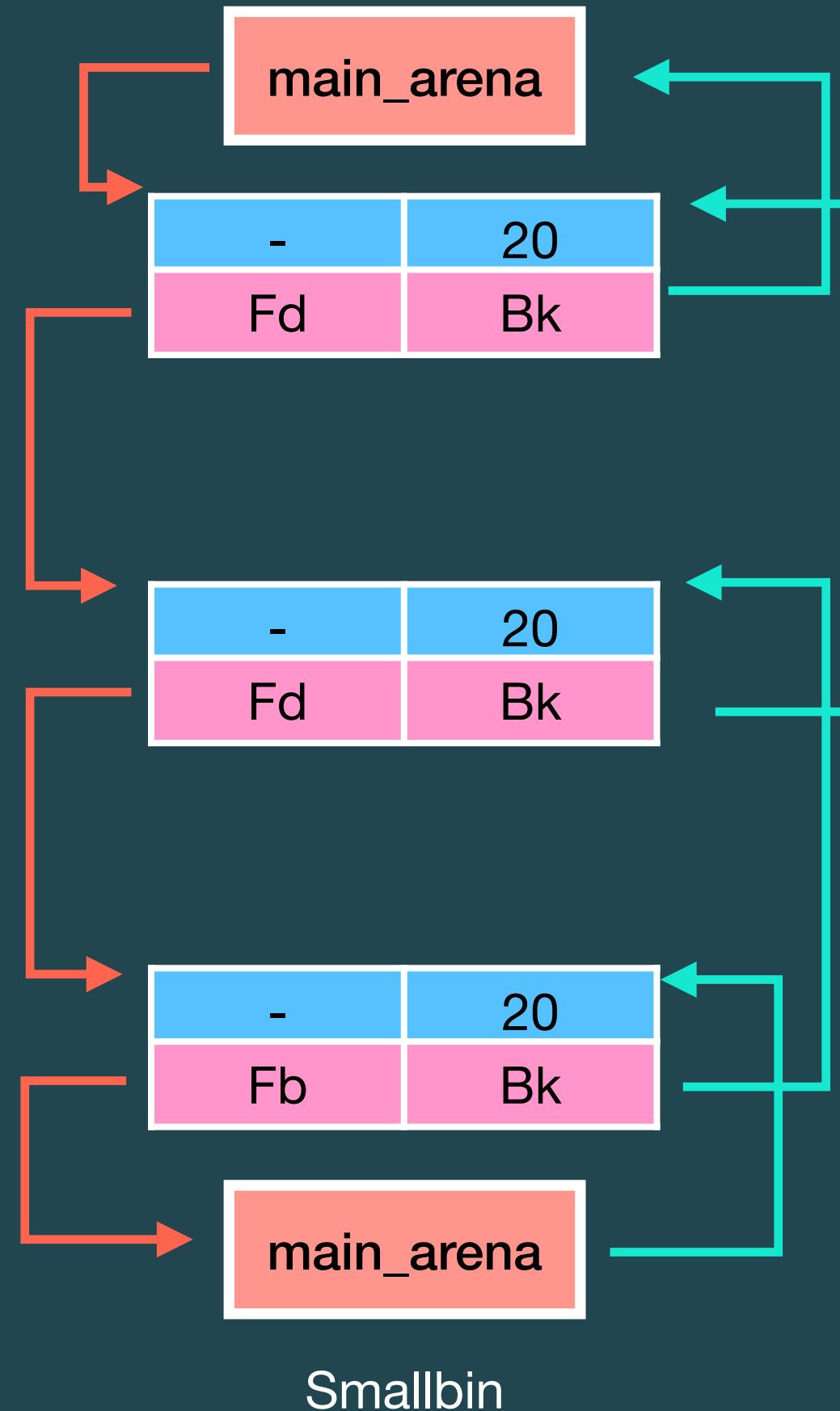
Lab - Heapmath

- ▶ 了解 tcache、fastbin bin 的連接方式是非常基本的
- ▶ 此 lab 會需要回答：
 - ⌚ Freed chunk 在 tcache 內的連接順序
 - ⌚ 一組 chunk 的距離運算
 - ⌚ Tcache fd 的計算
 - ⌚ Fastbin fd 的計算

\$ Heap introduction

Data Structure - Small bin

- ▶ **Smallbin** 存放 chunk size 為 $0x20 \sim 0x3f0$ 的 chunk
- ▶ 使用方式為 FIFO
- ▶ 其中 $0x20 \sim 0x80$ 的大小與 fastbin 重疊
 - ⦿ 直接 free 的 chunk 會進 fastbin
 - ⦿ 從 unsorted bin 放回去時會進 smallbin



\$ Heap introduction

Data Structure - Small bin



\$ Heap introduction

Data Structure - Large bin

- ▶ **Largebin** 存放 chunk size $\geq 0x400$ 的 chunk，使用方式為 FIFO
- ▶ 每個 subbin 都存放多種不同 size 的 chunk
 - ⌚ 32 bins — +0x40
 - > 0x400, 0x410, 0x420, 0x430 → 0-bin
 - > 0x440, 0x450, 0x460, 0x470 → 1-bin
 - > ...

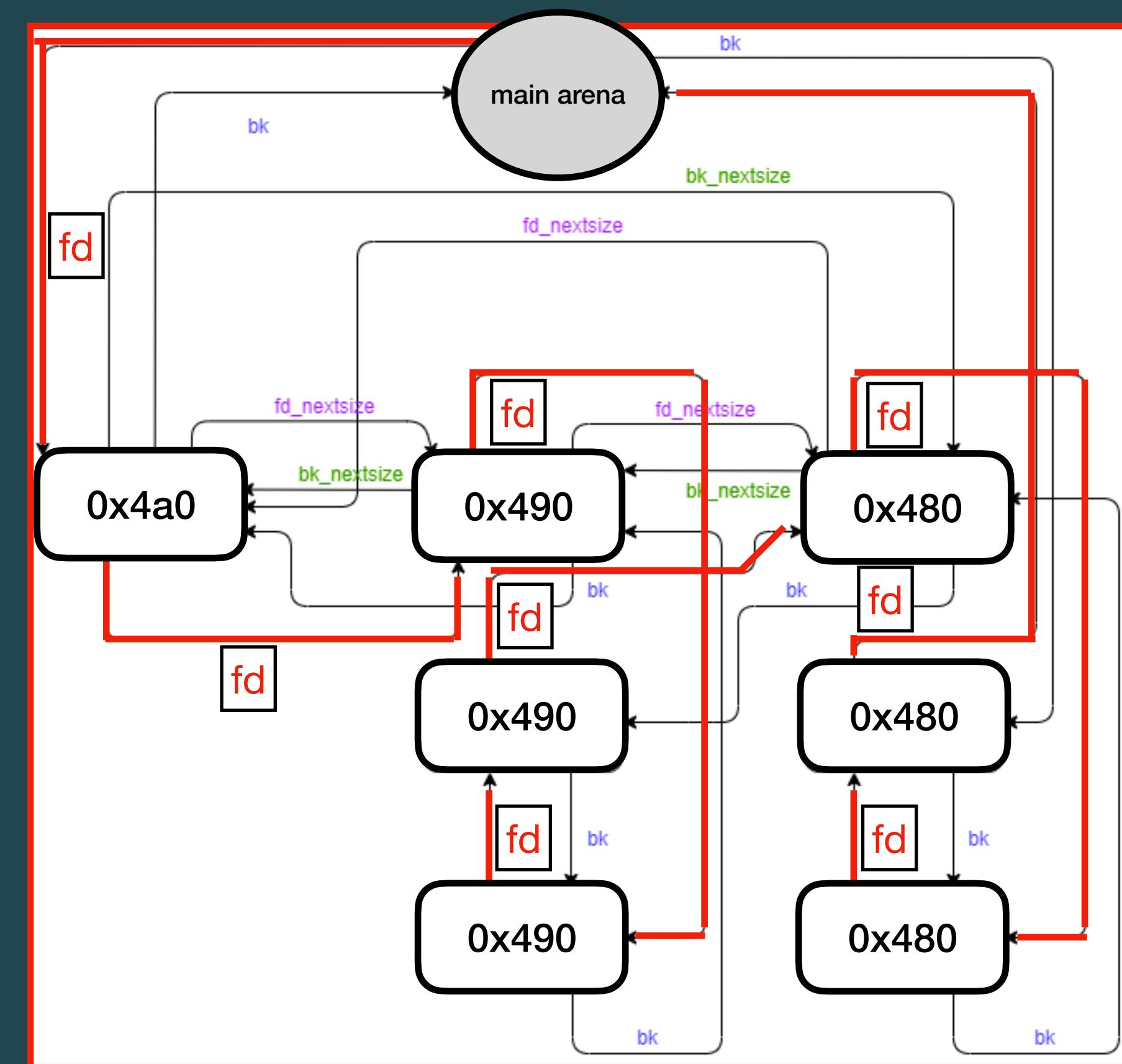
\$ Heap introduction

Data Structure - Large bin

- ▶ 每個 subbin 都存放多種不同 size 的 chunk (cont.)
 - ⦿ 16 bins — +0x200
 - ⦿ 8 bins — +0x1000
 - ⦿ 4 bins — +0x8000
 - ⦿ 2 bins — +0x40000
 - ⦿ 1 bins — the rest
- ▶ 使用 `fd_nextsize` 與 `bk_nextsize` 將不同 size 的 chunk 串起來

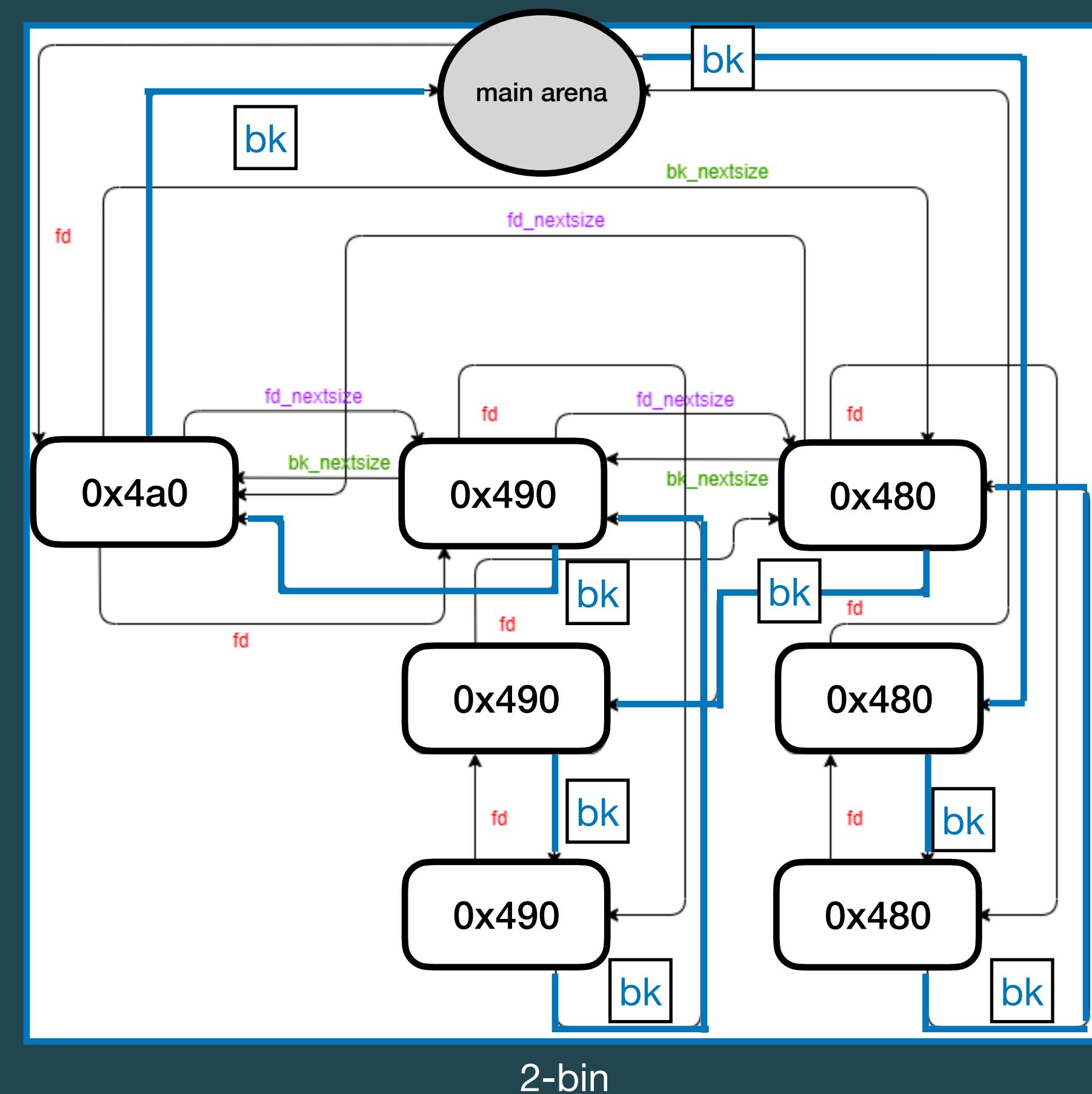
\$ Heap introduction

Data Structure - Large bin



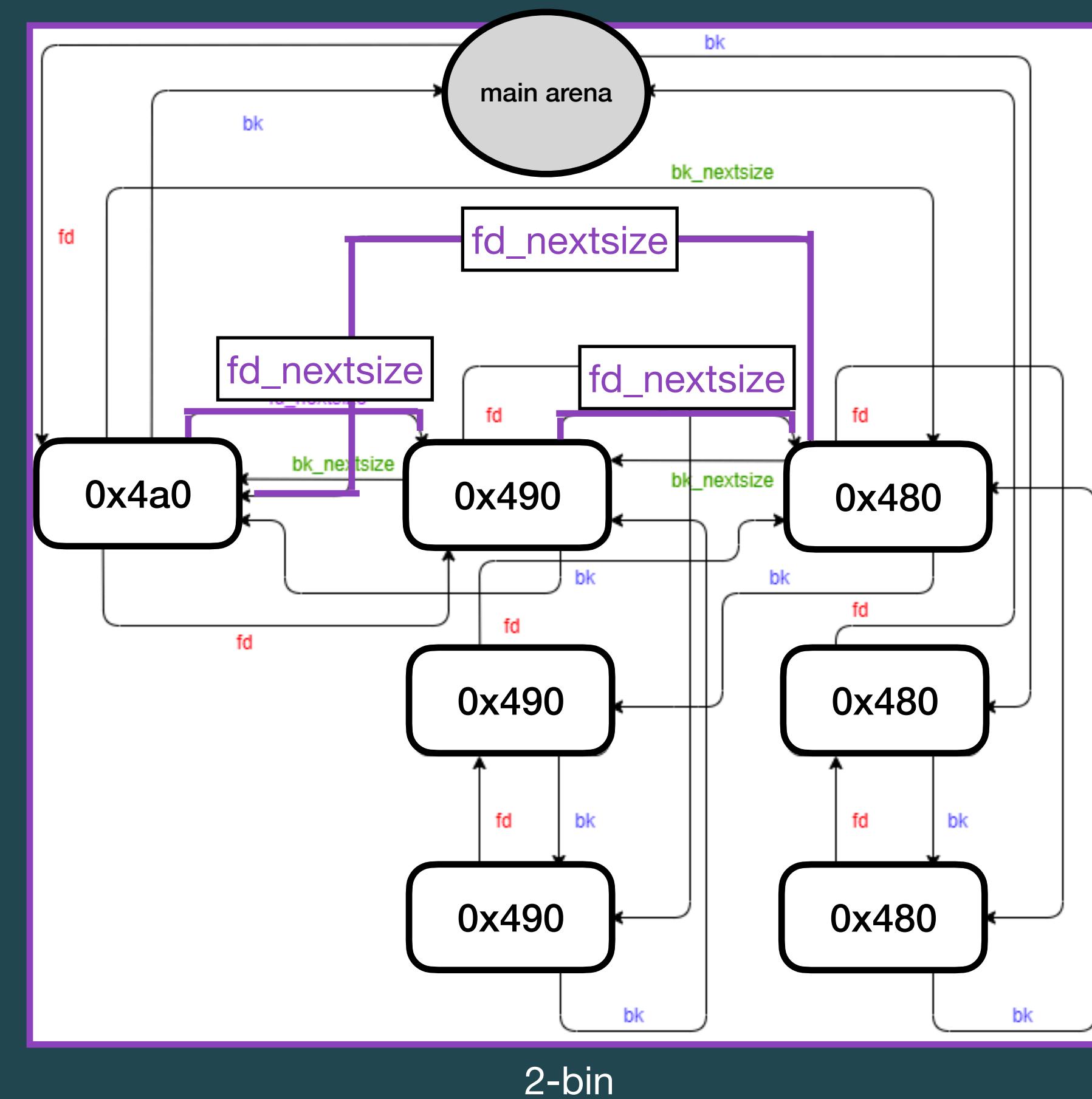
\$ Heap introduction

Data Structure - Large bin



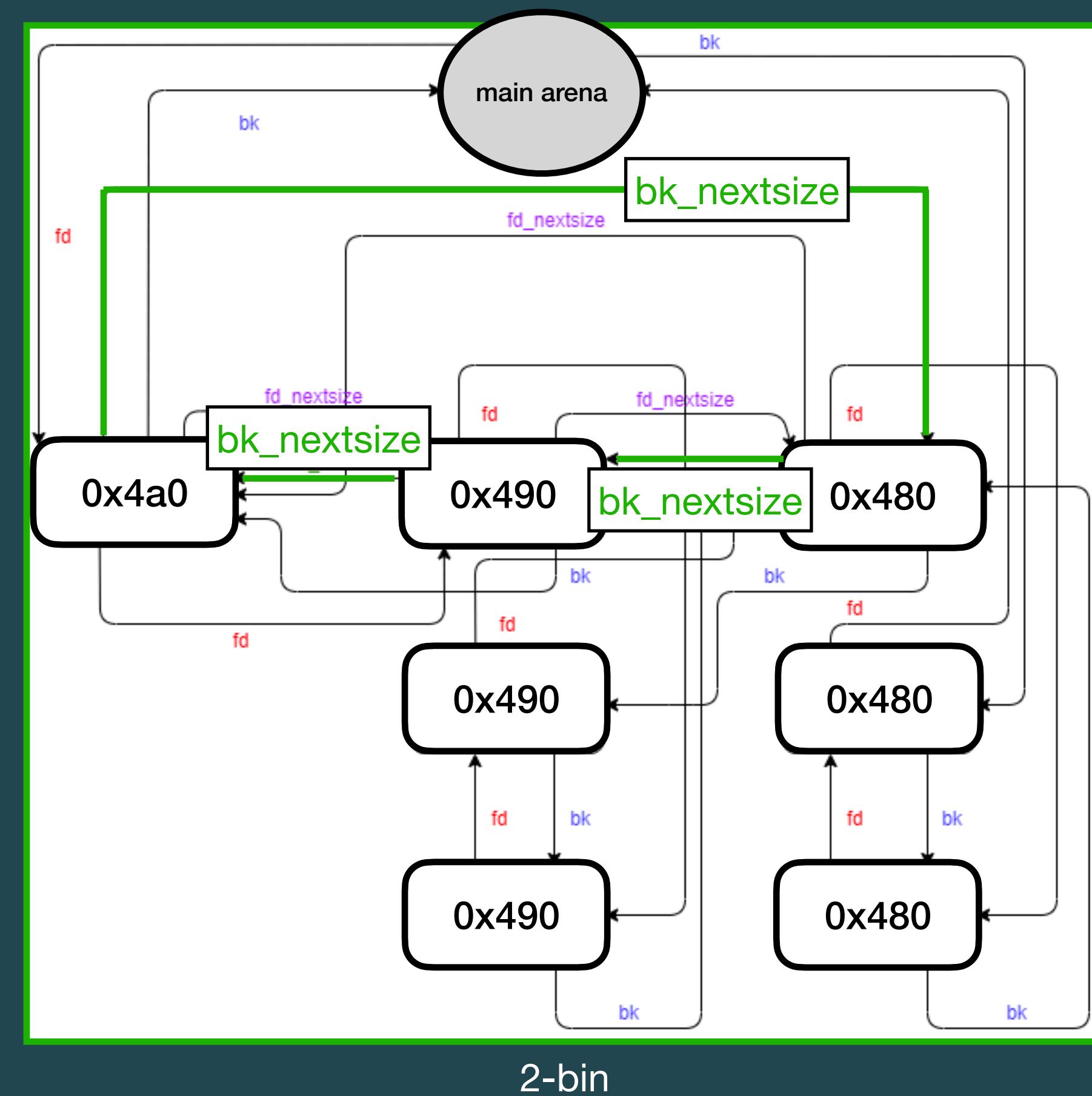
\$ Heap introduction

Data Structure - Large bin



\$ Heap introduction

Data Structure - Large bin



\$ Heap introduction

Data Structure - Large bin



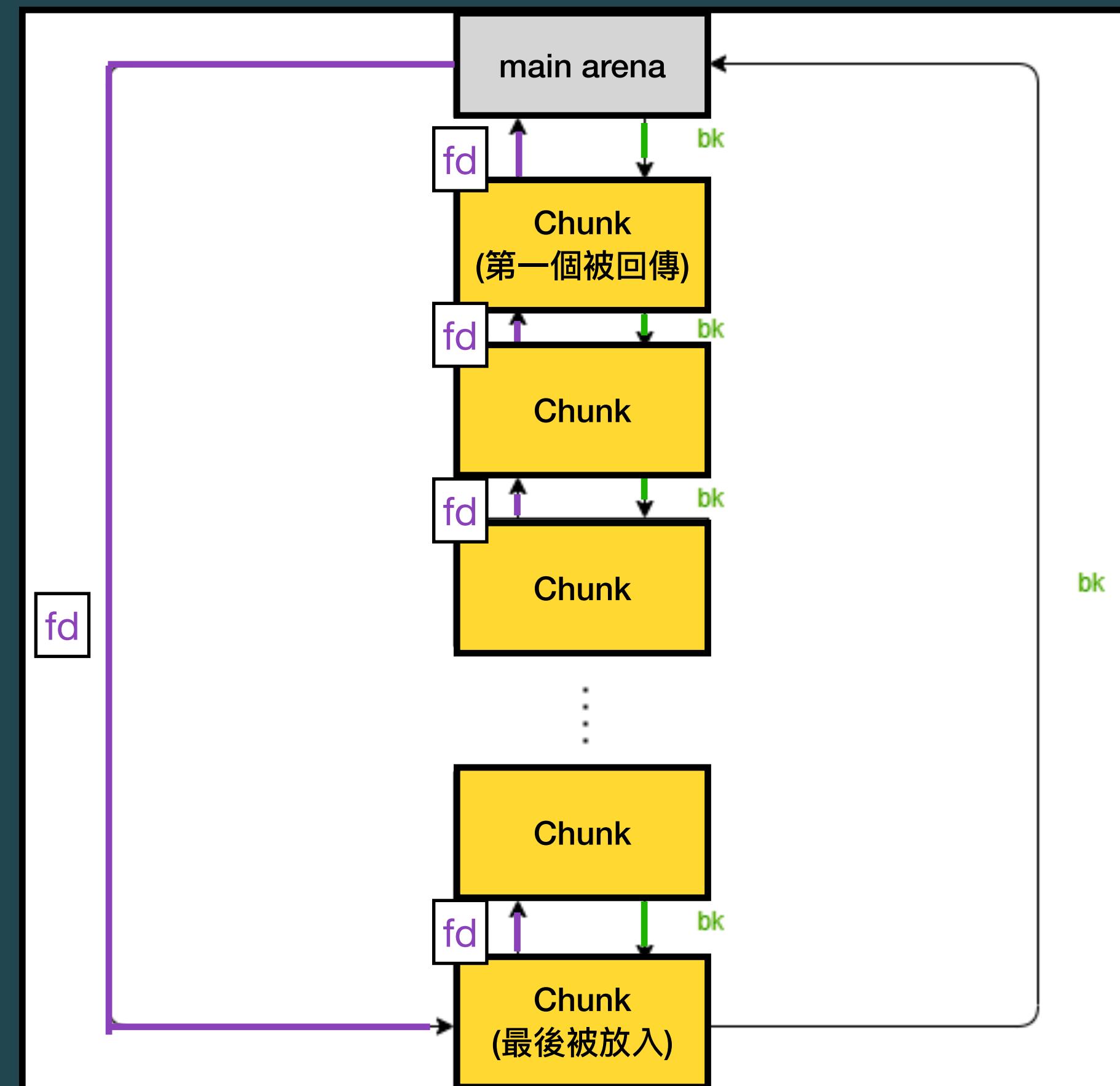
\$ Heap introduction

Data Structure - Unsorted bin

- ▶ **Unsorted bin** - 對應大小的 tcache 已經滿的情況下，並且 chunk size 非 fastbin chunk 的大小，則此 chunk 被 free 後會暫存在 unsorted bin
 - ⦿ 當收到請求後，如果對應大小的 tcache、fastbin、smallbin 為空，則會遍歷整個 unsorted bin
 - ⦿ 若 unsorted bin 中有 chunk 的大小 \geq (請求的大小 + 0x20)，會直接從該塊 chunk 切下來回傳
 - > 0x20 為最小的 chunk size
 - ⦿ 若沒有，則會將這些暫存的 chunk 放至對應的 bin (small bin, large bin, ...)

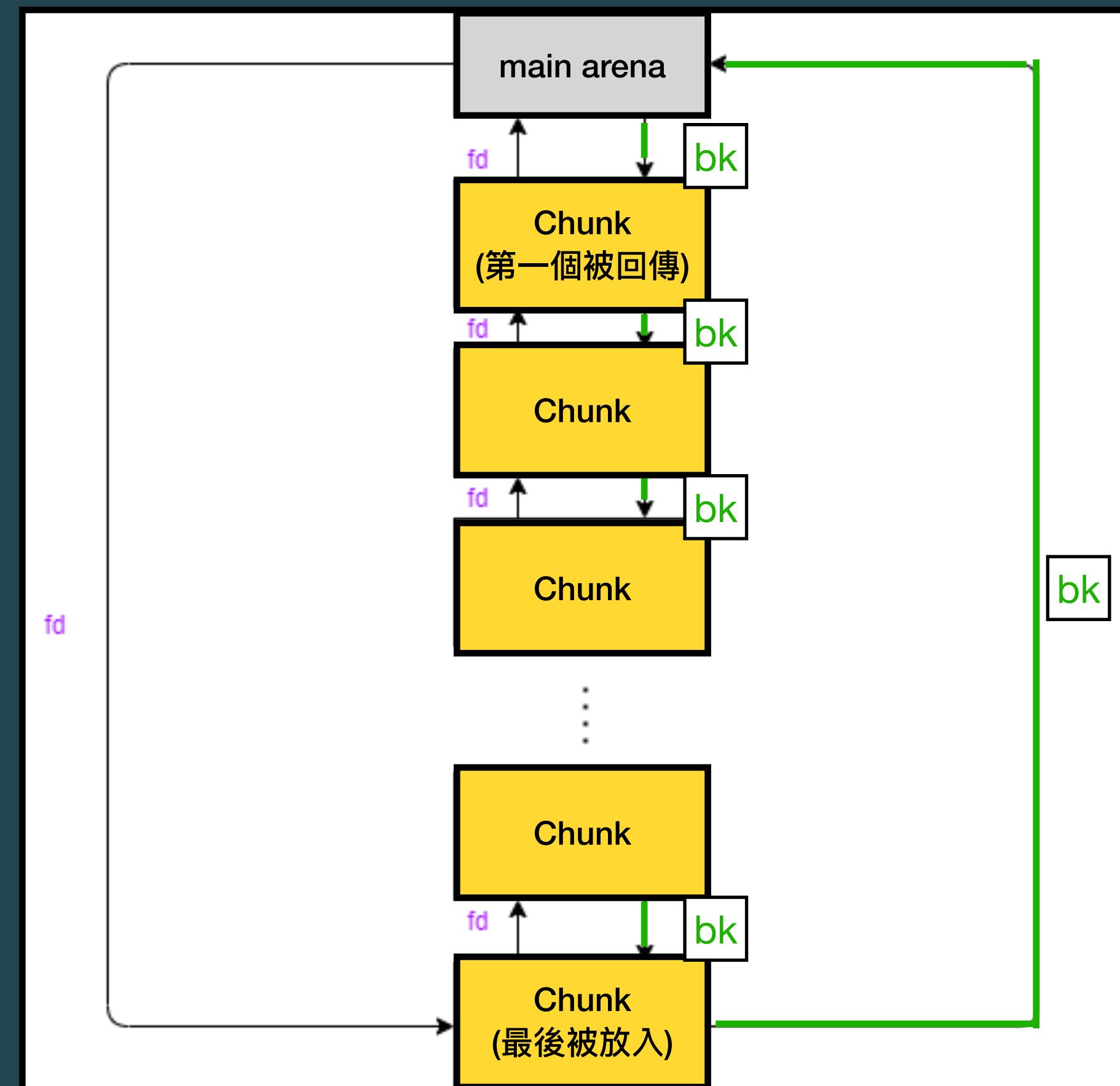
\$ Heap introduction

Data Structure - Unsorted bin



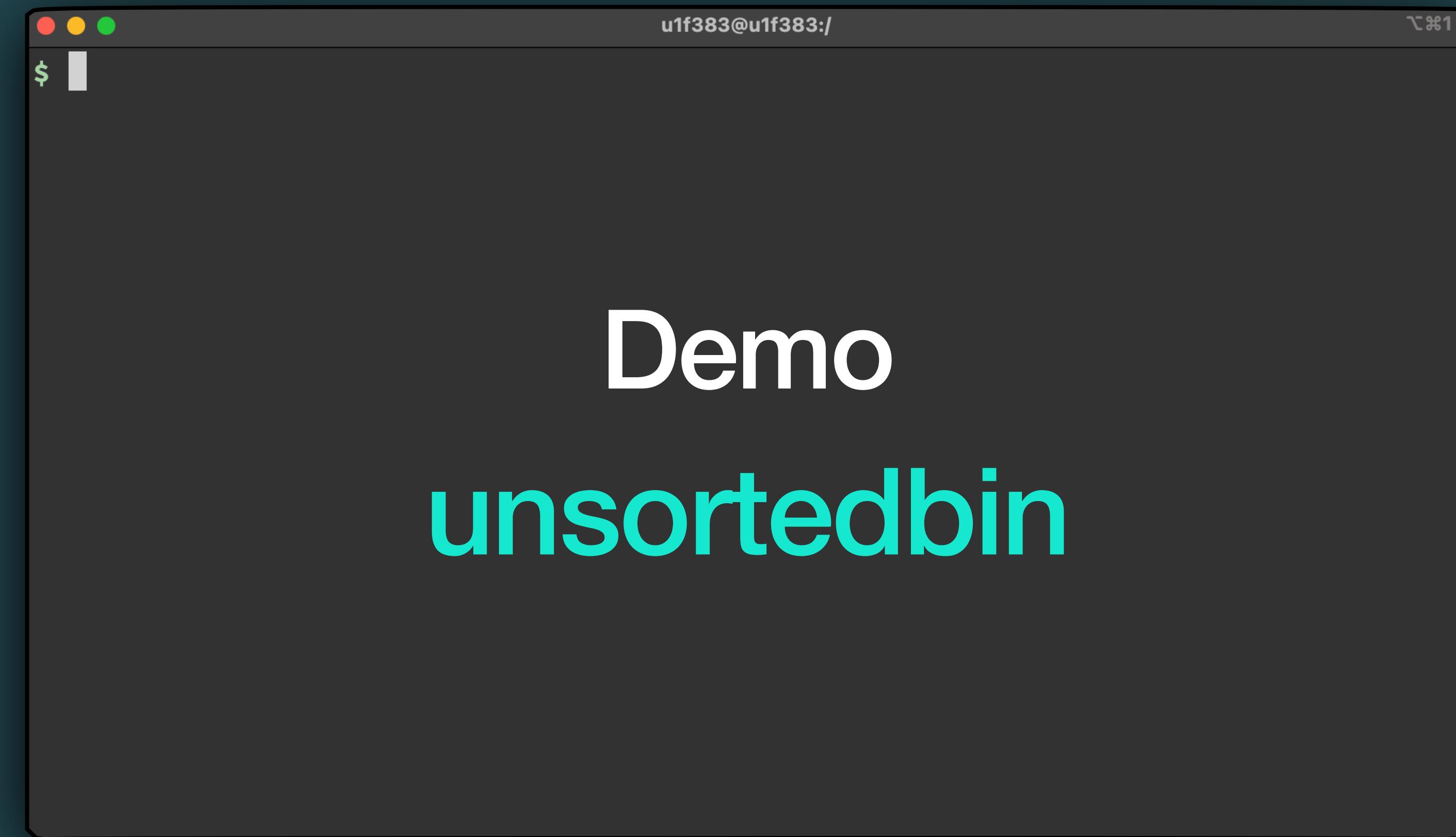
\$ Heap introduction

Data Structure - Unsorted bin



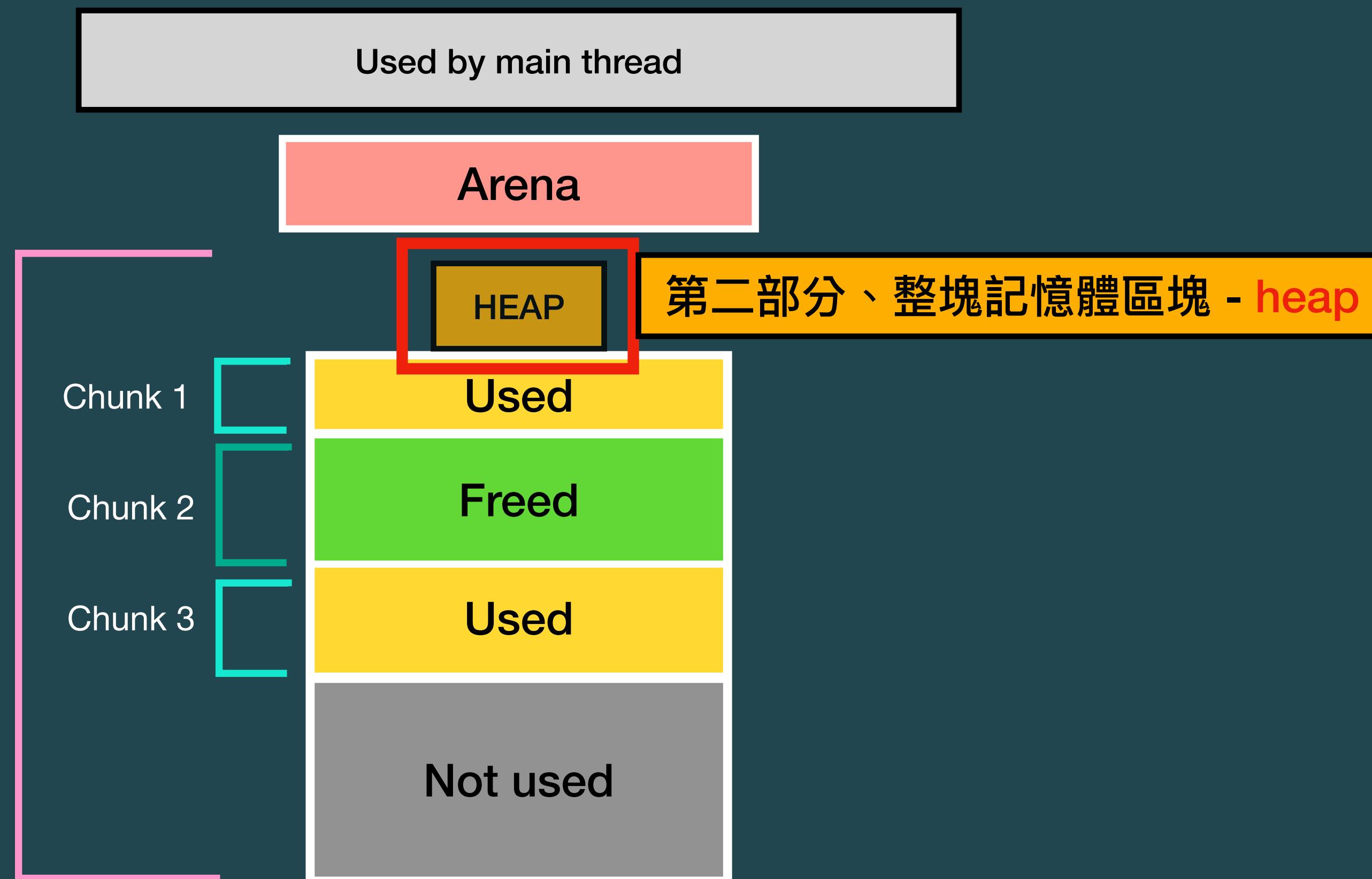
\$ Heap introduction

Data Structure - Unsorted bin



\$ Heap introduction

Data Structure - Heap



\$ Heap introduction

Data Structure - Heap



A screenshot of a terminal window titled "u1f383@u1f383:/". The terminal shows a block of C code defining a structure named `_heap_info`. The code includes comments explaining the fields: `mstate ar_ptr` (指向當前 heap 使用的 arena), `*prev` (指向上一塊 heap), `size_t size` (當前 heap 的大小), `size_t mprotect_size` (prot 為 `RROT_READ|PROT_WRITE` 的空間大小). It also includes a padding array `char pad[-6 * SIZE_SZ & MALLOC_ALIGN_MASK]` for alignment.

```
typedef struct _heap_info
{
    mstate ar_ptr;          /* 指向當前 heap 使用的 arena */
    struct _heap_info *prev; /* 指向上一塊 heap */
    size_t size;            /* 當前 heap 的大小 */
    size_t mprotect_size;   /* prot 為 RROT_READ|PROT_WRITE 的空間大小 */

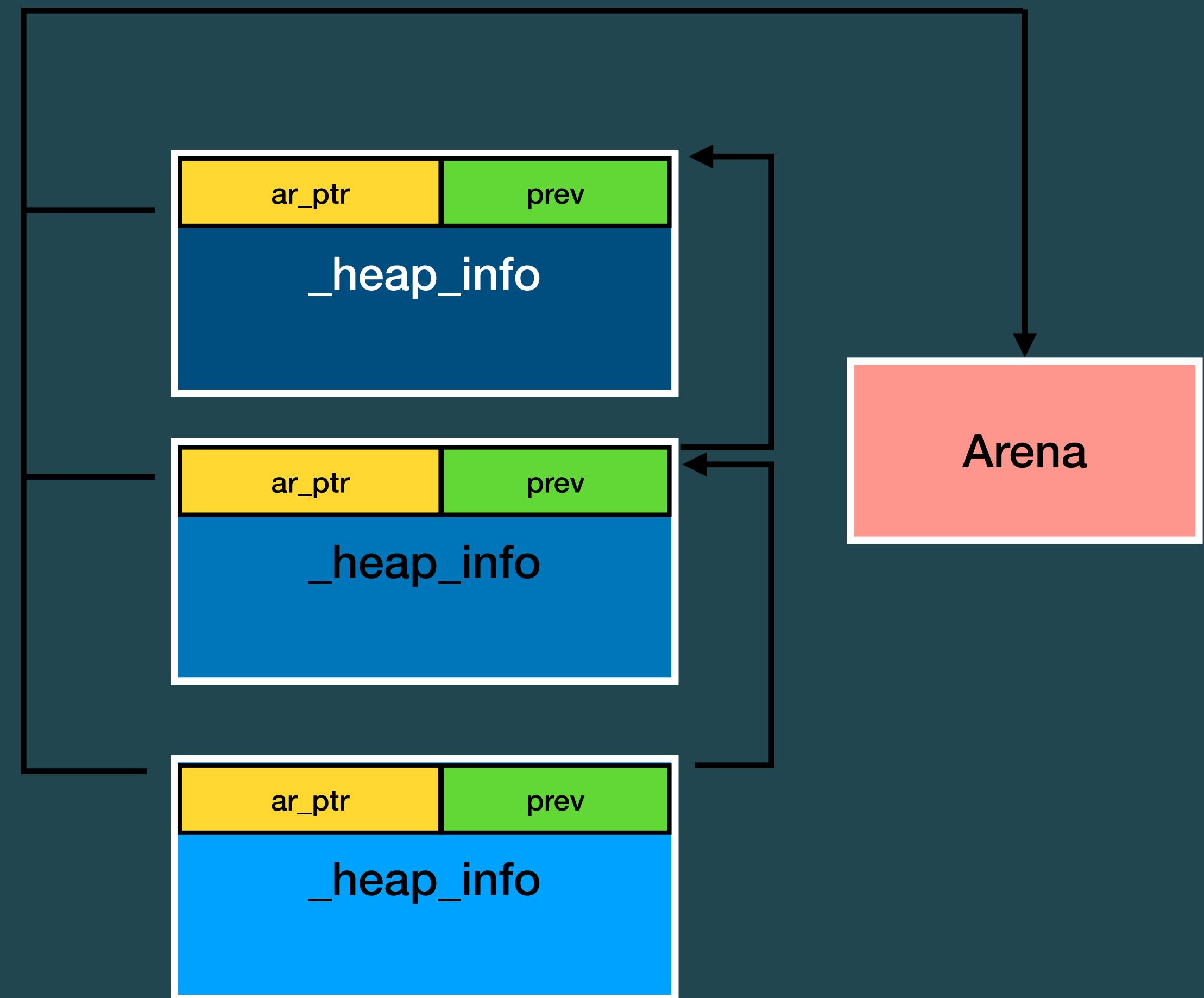
    /* 用來做 alignment */
    char pad[-6 * SIZE_SZ & MALLOC_ALIGN_MASK];
} heap_info;
```

<https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/arena.c#L53>

\$ Heap introduction

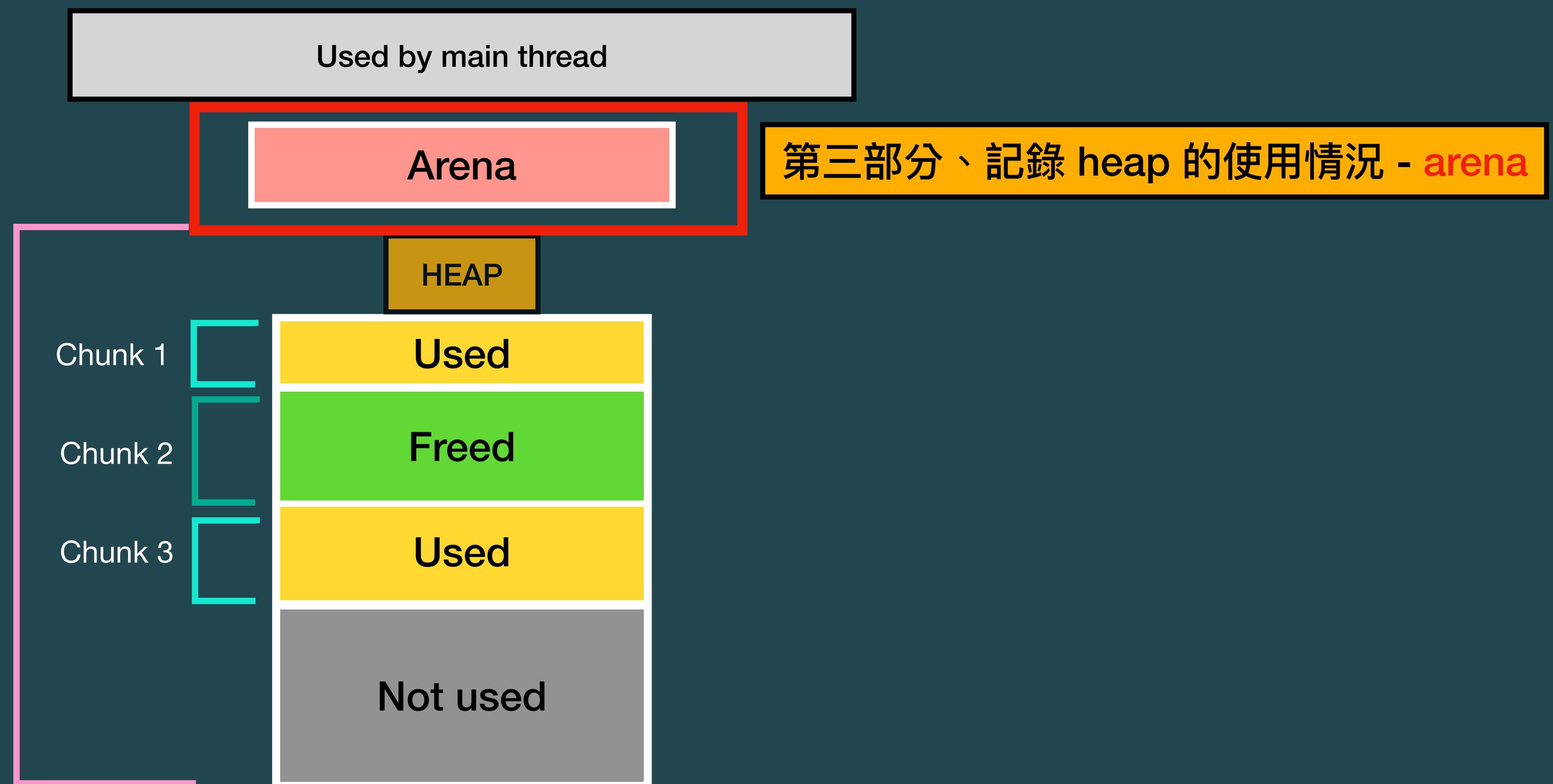
Data Structure - Heap

- ▶ Size 與 mprotect_size 預設為 0x21000
- ▶ Main thread 本身是沒有 _heap_info，因此 heap exploit 並不常用到 _heap_info



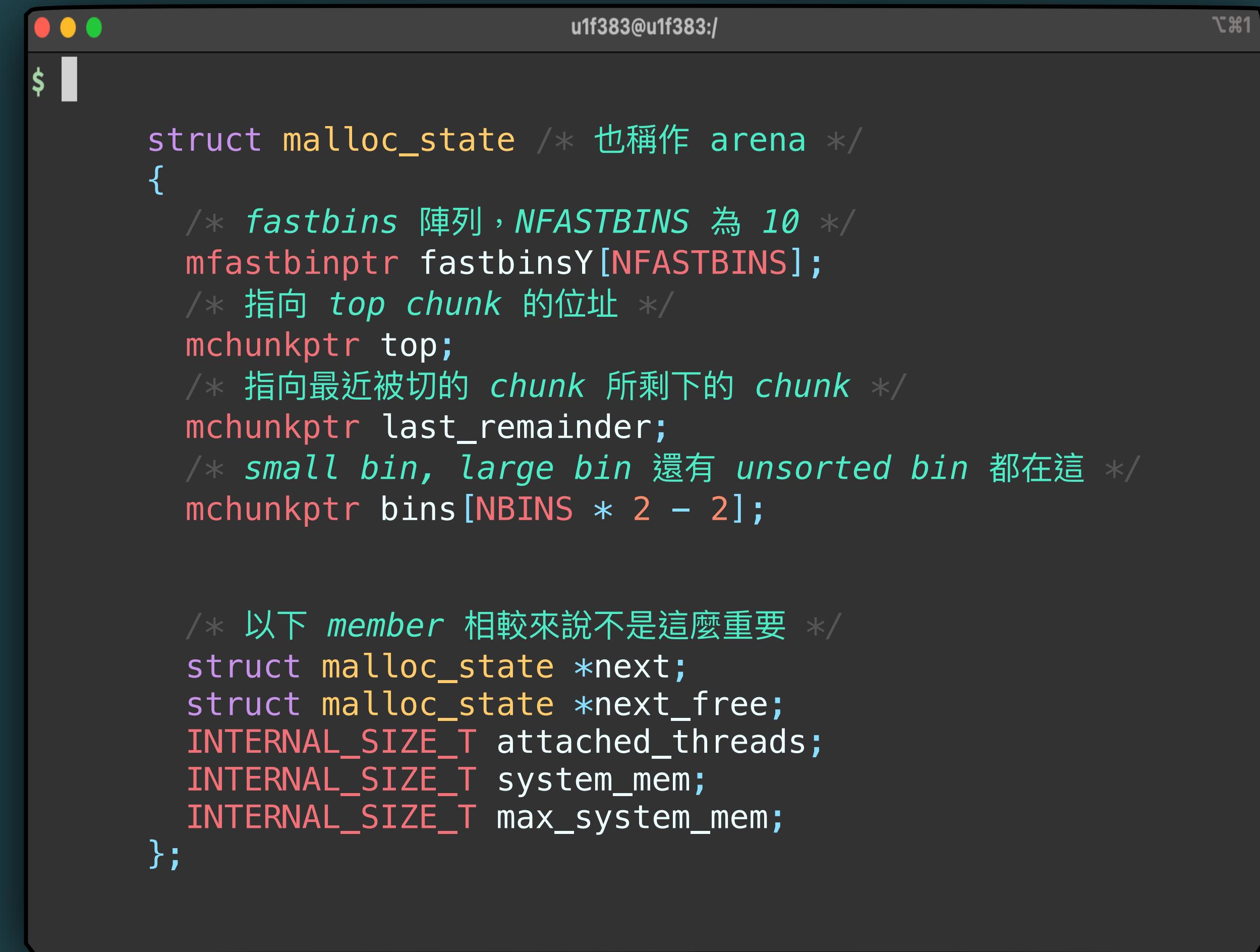
\$ Heap introduction

Data Structure - Arena



\$ Heap introduction

Data Structure - Arena



A screenshot of a terminal window titled "u1f383@u1f383:/". The window contains the following C code:

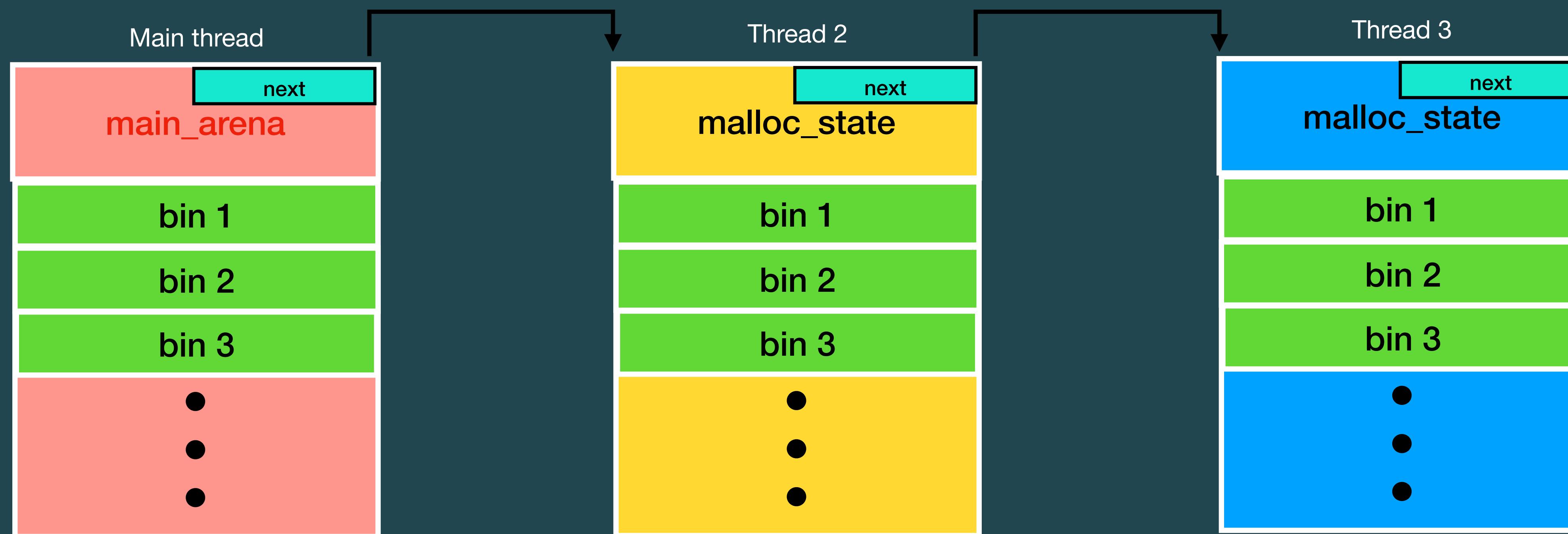
```
struct malloc_state /* 也稱作 arena */
{
    /* fastbins 陣列，NFASTBINS 為 10 */
    mfastbinptr fastbins[NFASTBINS];
    /* 指向 top chunk 的位址 */
    mchunkptr top;
    /* 指向最近被切的 chunk 所剩下的 chunk */
    mchunkptr last_remainder;
    /* small bin, large bin 還有 unsorted bin 都在這 */
    mchunkptr bins[NBINS * 2 - 2];

    /* 以下 member 相較來說不是這麼重要 */
    struct malloc_state *next;
    struct malloc_state *next_free;
    INTERNAL_SIZE_T attached_threads;
    INTERNAL_SIZE_T system_mem;
    INTERNAL_SIZE_T max_system_mem;
};
```

\$ Heap introduction

Data Structure - Arena

- ▶ Main thread 的 malloc_state 也稱作 **main_arena**
- ▶ 由於後續只考慮一個 thread 的情況，基本上只會使用到 **main_arena**



\$ Heap introduction

Data Structure - Arena

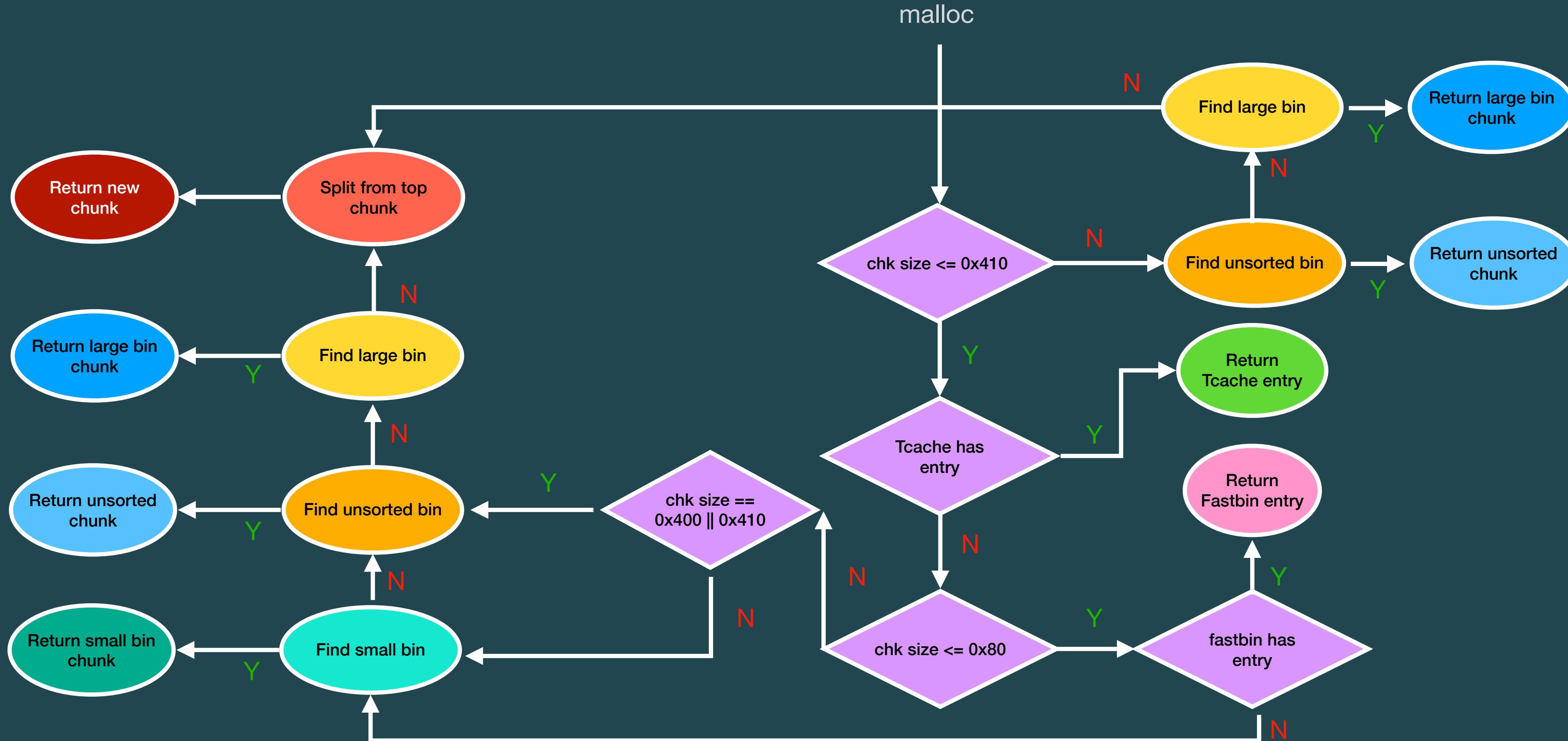




Code tracing

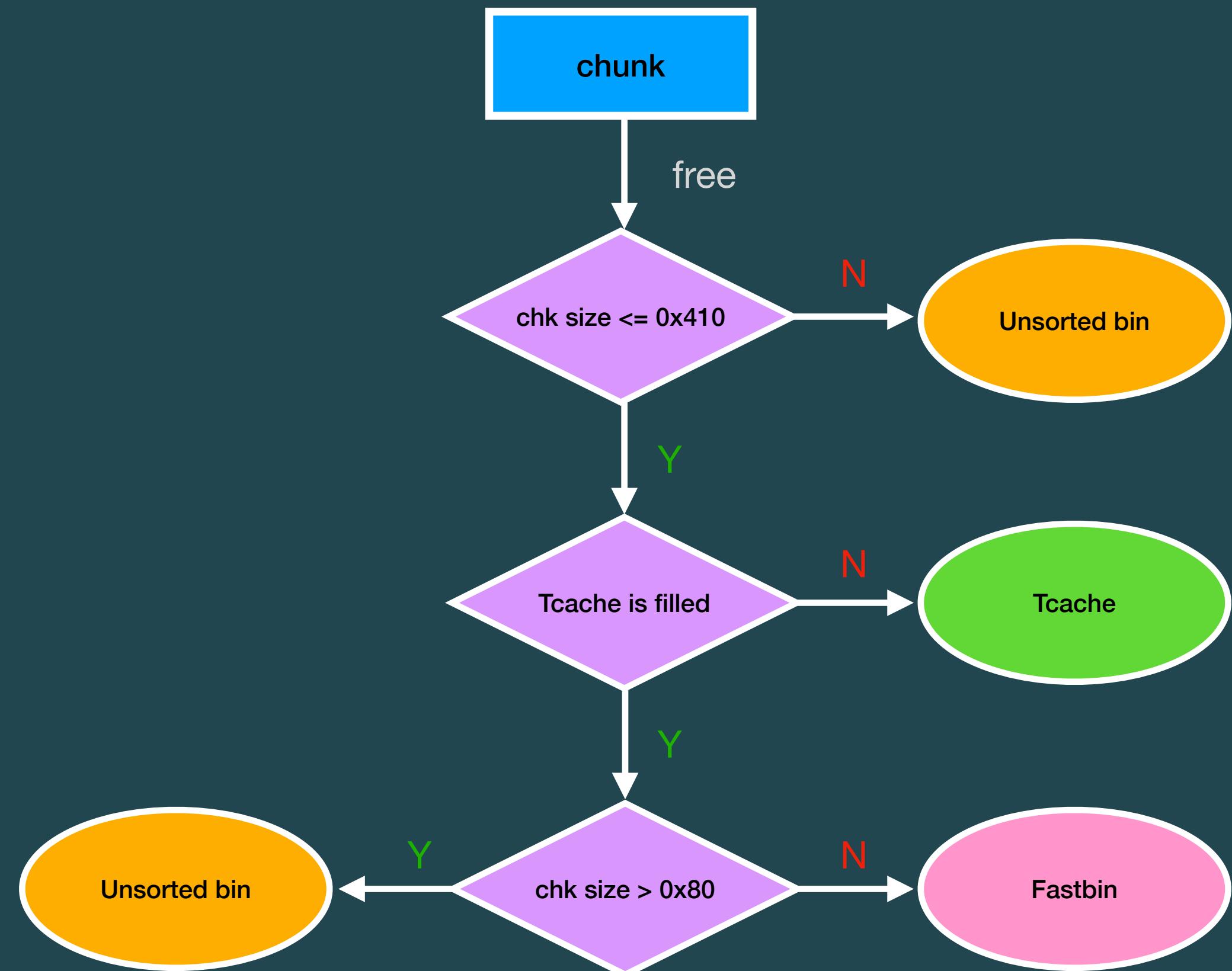
\$ Code tracing

Malloc



\$ Code tracing

Free





Vulnerability

\$ Vulnerability

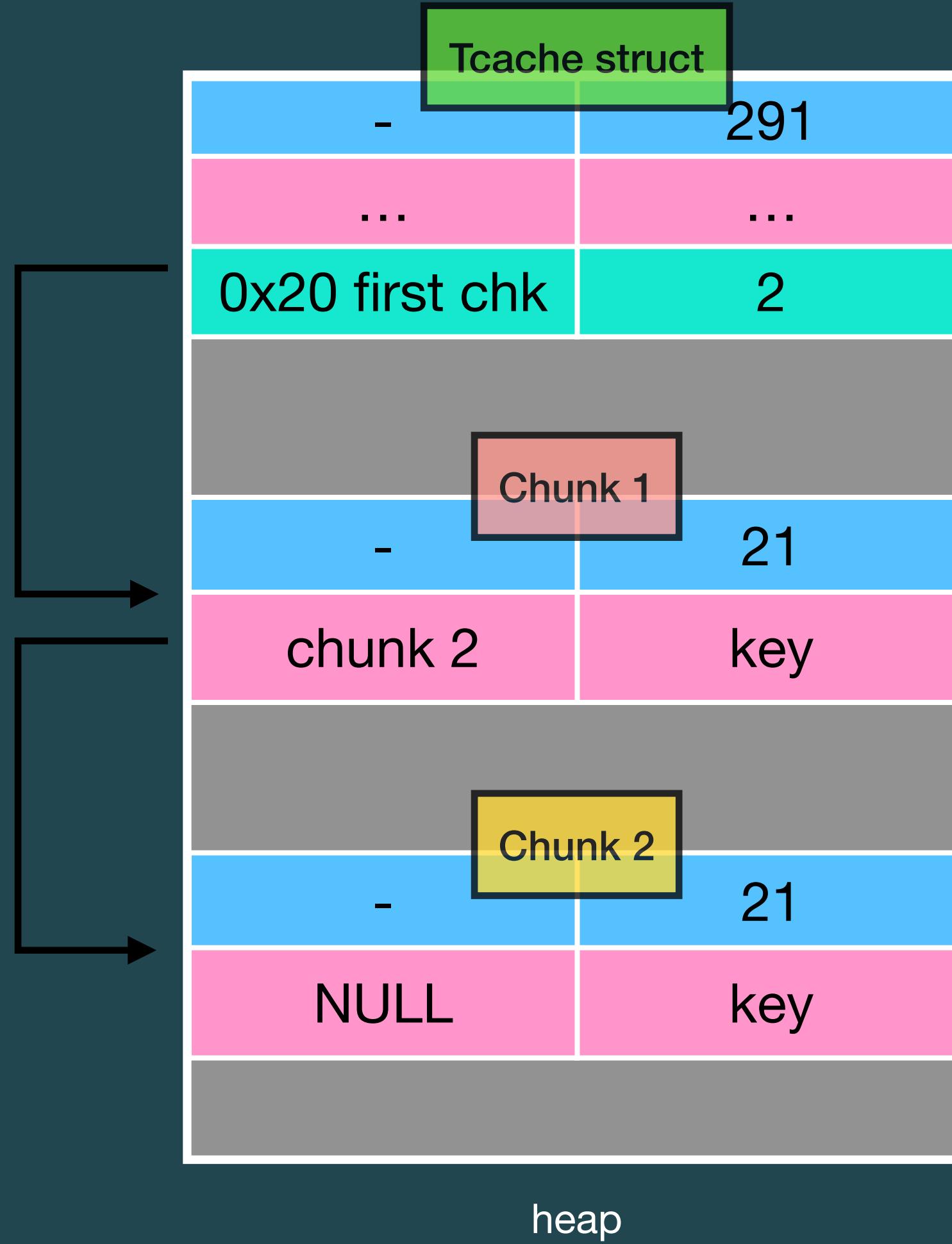
UAF

- ▶ Use-**A**fter-**F**ree
- ▶ Chunk 在被 free 後並沒有把 pointer 設為 **N****U****L****L**，導致使用到已經被釋放的記憶體
- ▶ 這種指向已釋放資源的 pointer 又稱 **d****a****n****g****l****i****ng** pointer

\$ Vulnerability

UAF - Example

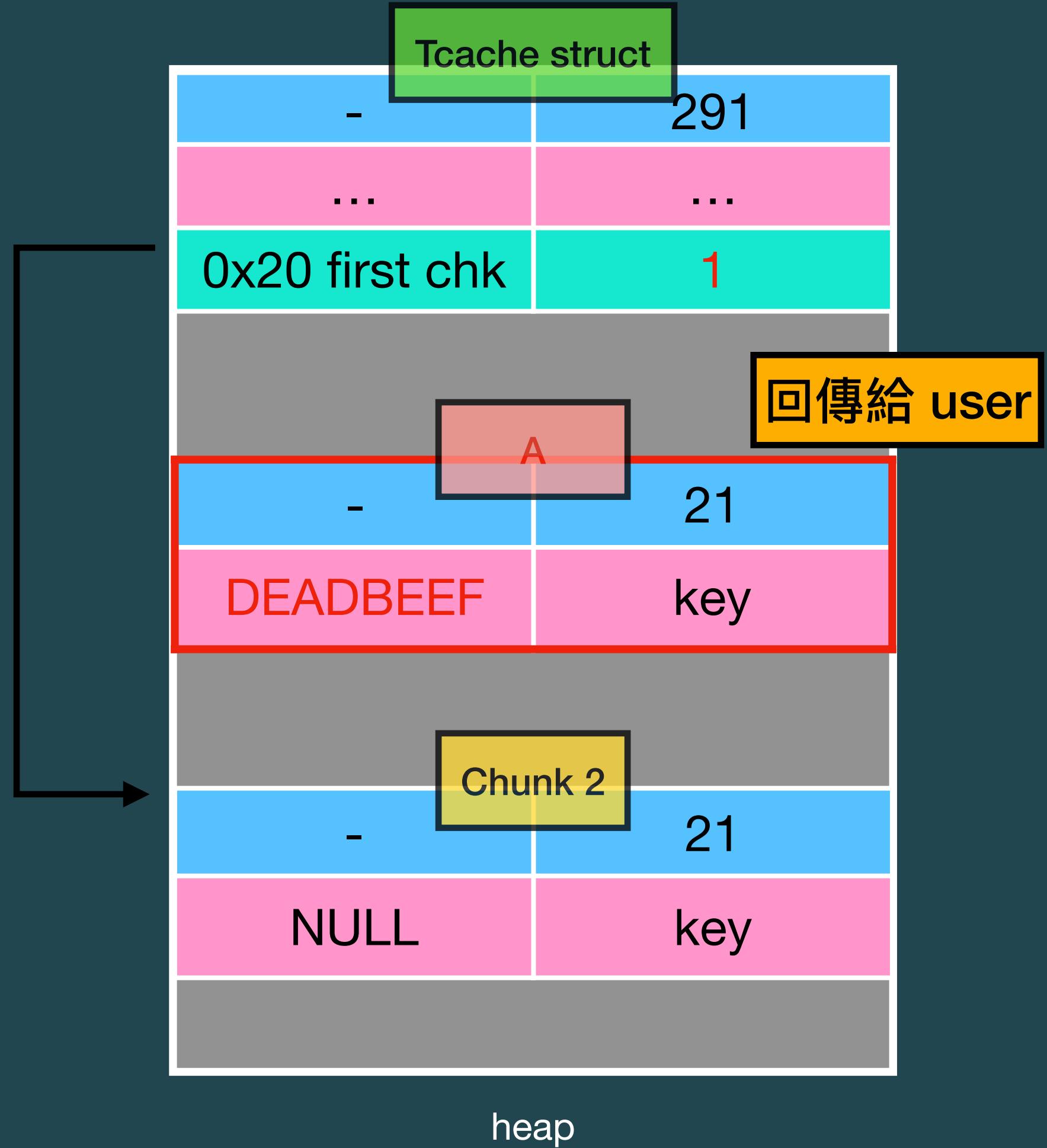
```
u1f383@u1f383:~$ 
unsigned long *a = malloc(0x10);
*a = 0xdeadbeef;
free(a);
*a = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
```



\$ Vulnerability

UAF - Example

```
u1f383@u1f383:/ $  
unsigned long *a = malloc(0x10);  
*a = 0xdeadbeef;  
free(a);  
*a = 0xdeadbeef;  
malloc(0x10);  
malloc(0x10);  
$
```

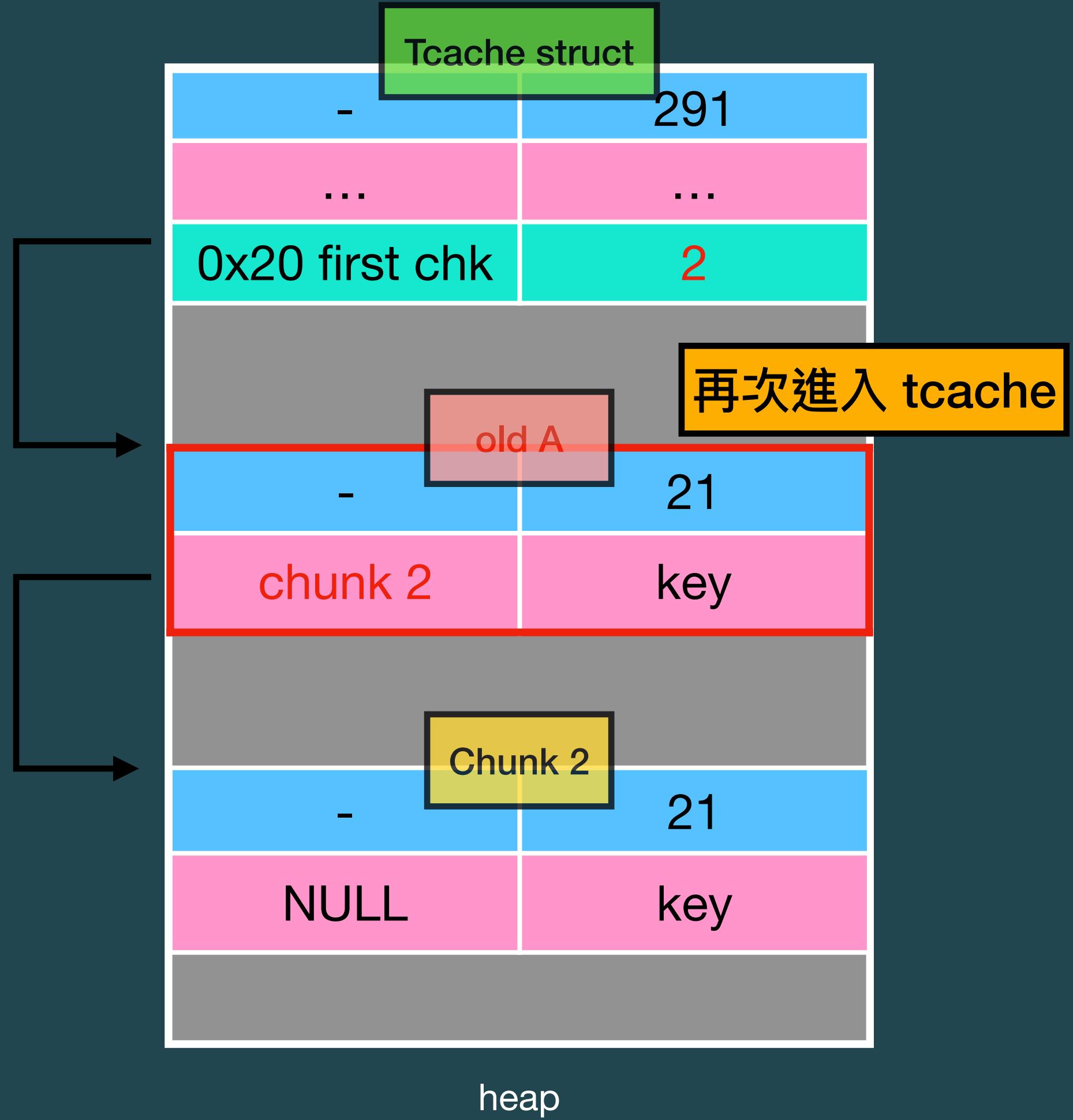


\$ Vulnerability

UAF - Example

```
u1f383@u1f383:/
```

```
unsigned long *a = malloc(0x10);
*a = 0xdeadbeef;
free(a);
*a = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
```

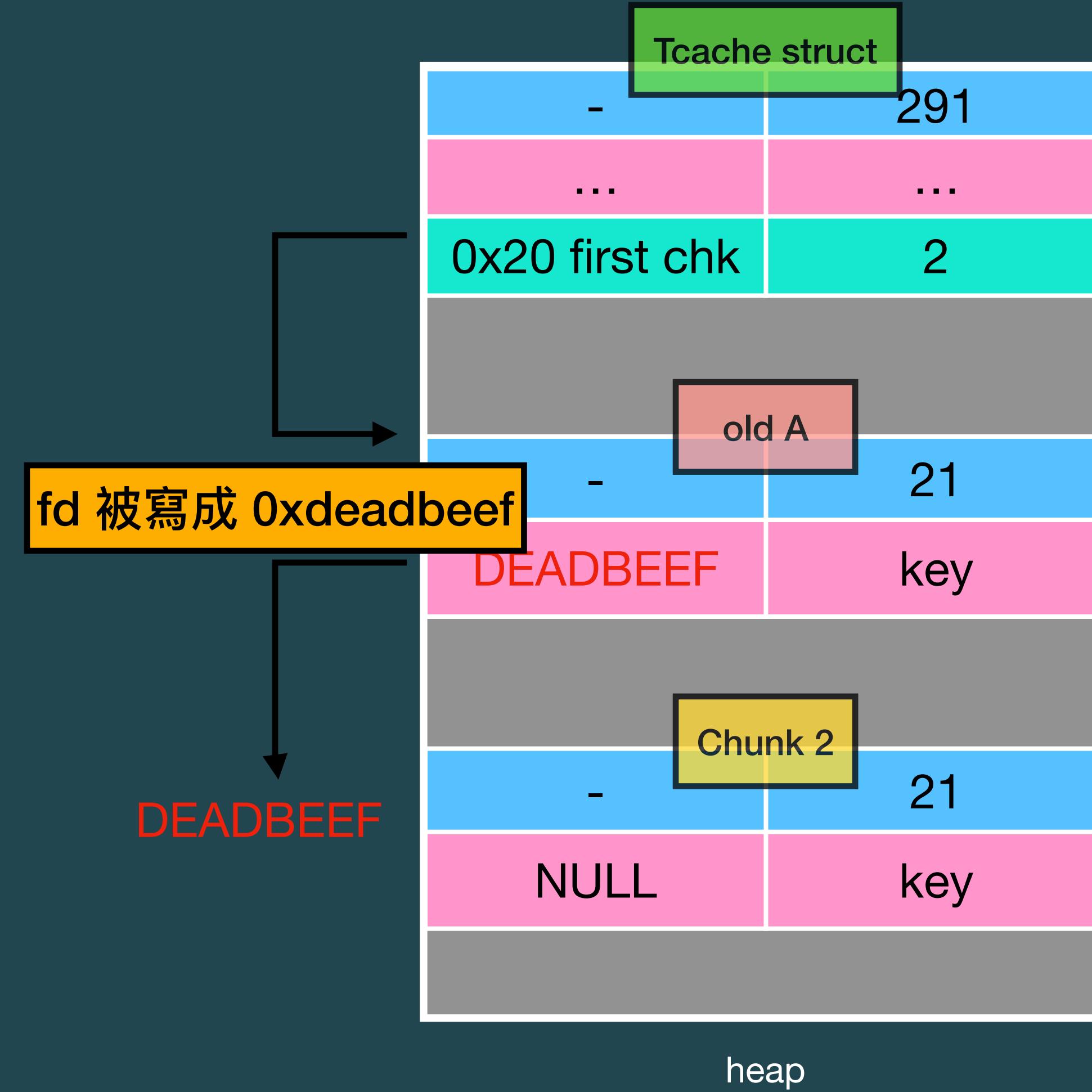


\$ Vulnerability

UAF - Example

```
u1f383@u1f383:/ $  
unsigned long *a = malloc(0x10);  
*a = 0xdeadbeef;  
free(a);  
*a = 0xdeadbeef;  
malloc(0x10);  
malloc(0x10);
```

a 被釋放後仍被使用

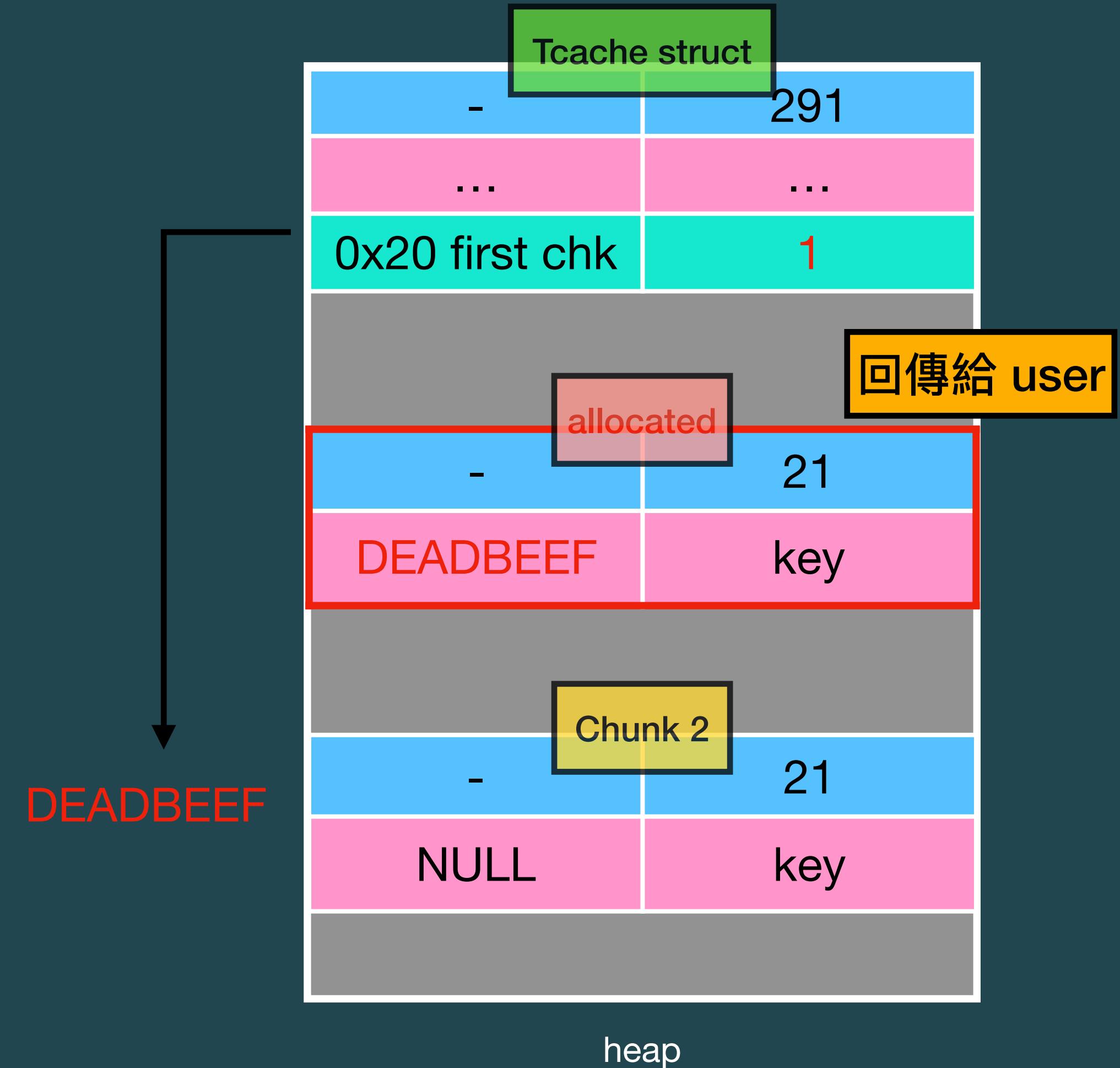


\$ Vulnerability

UAF - Example

```
u1f383@u1f383:/
```

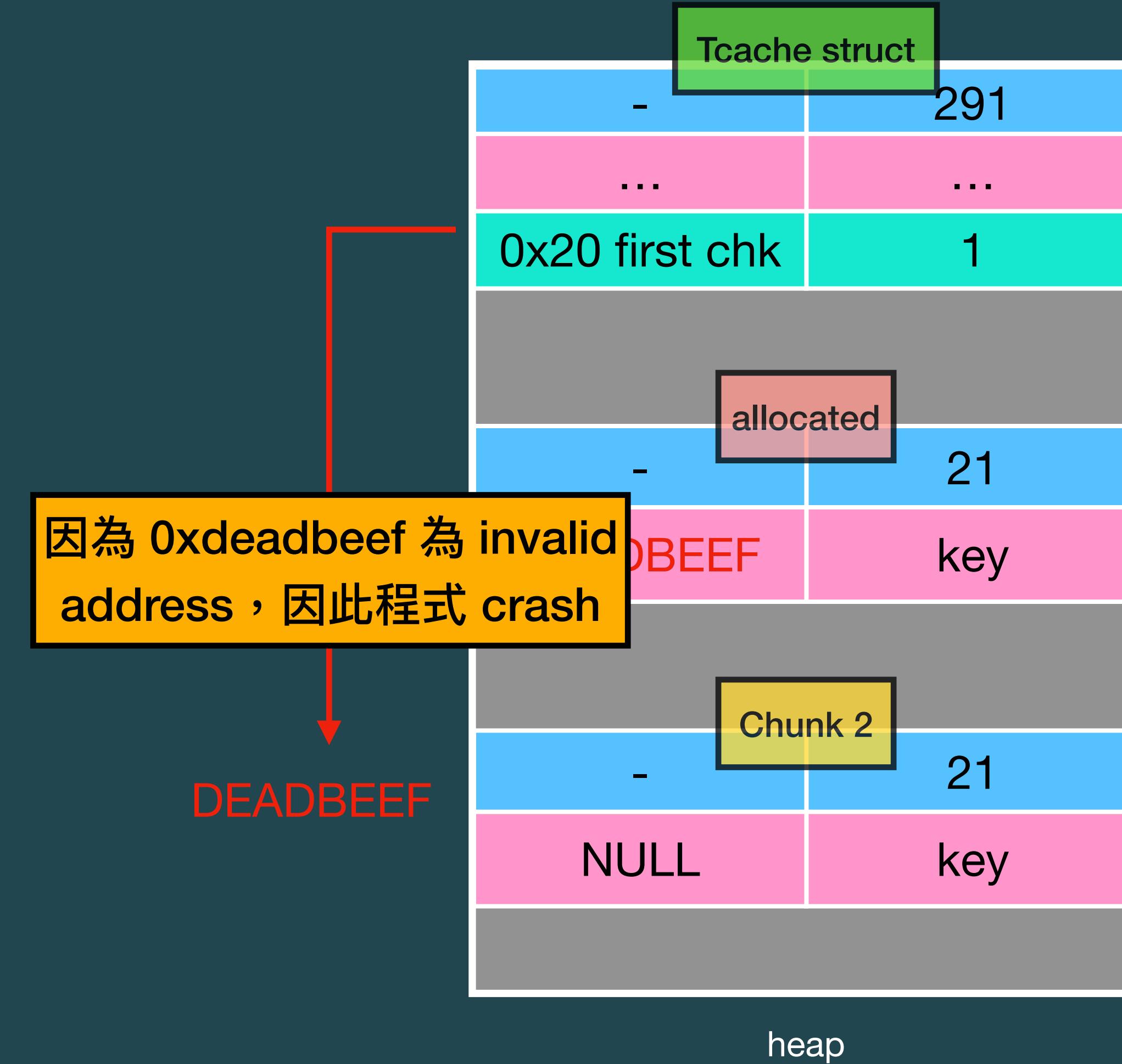
```
unsigned long *a = malloc(0x10);
*a = 0xdeadbeef;
free(a);
*a = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
```



\$ Vulnerability

UAF - Example

```
u1f383@u1f383:/ $  
unsigned long *a = malloc(0x10);  
*a = 0xdeadbeef;  
free(a);  
*a = 0xdeadbeef;  
malloc(0x10);  
malloc(0x10);  
$
```



\$ Vulnerability UAF



\$ Vulnerability

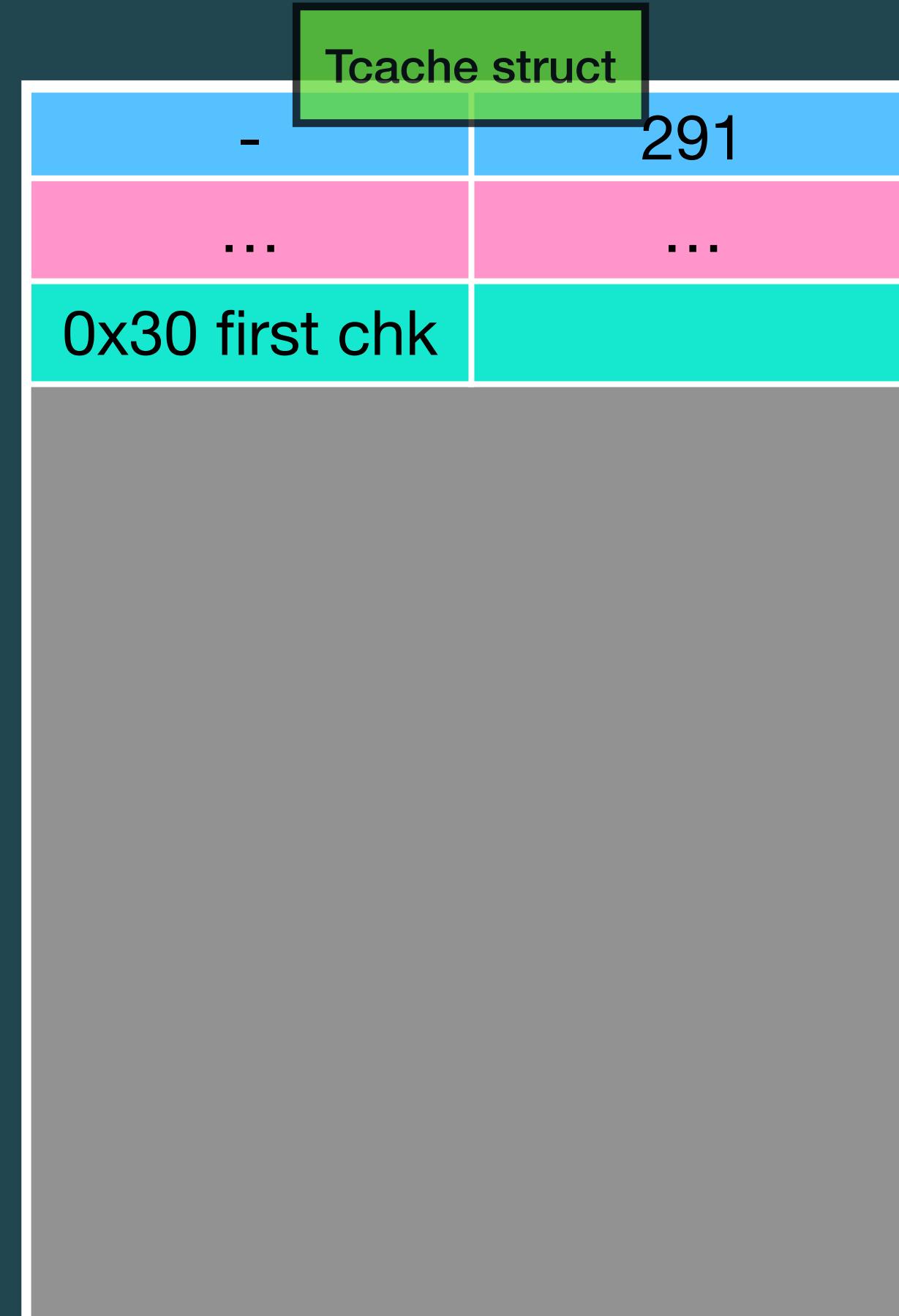
Heap overflow

- ▶ 在讀取資料時並沒有檢查好長度，導致可以越界寫到其他 chunk
 - ⦿ 如果越界寫到的是 allocated chunk，可以寫 chunk 內的
 - > 敏感資料，例如 function pointer
 - > Size，就可以釋放任意 size 的 chunk
 - ⦿ 如果越界寫到的是 freed chunk，可以寫 chunk 的
 - > Fd、bk，改變 linked list 的連接
 - > PREV_INUSE bit，讓 glibc 誤以為上一塊 chunk 已經被釋放

\$ Vulnerability

Heap overflow - Example

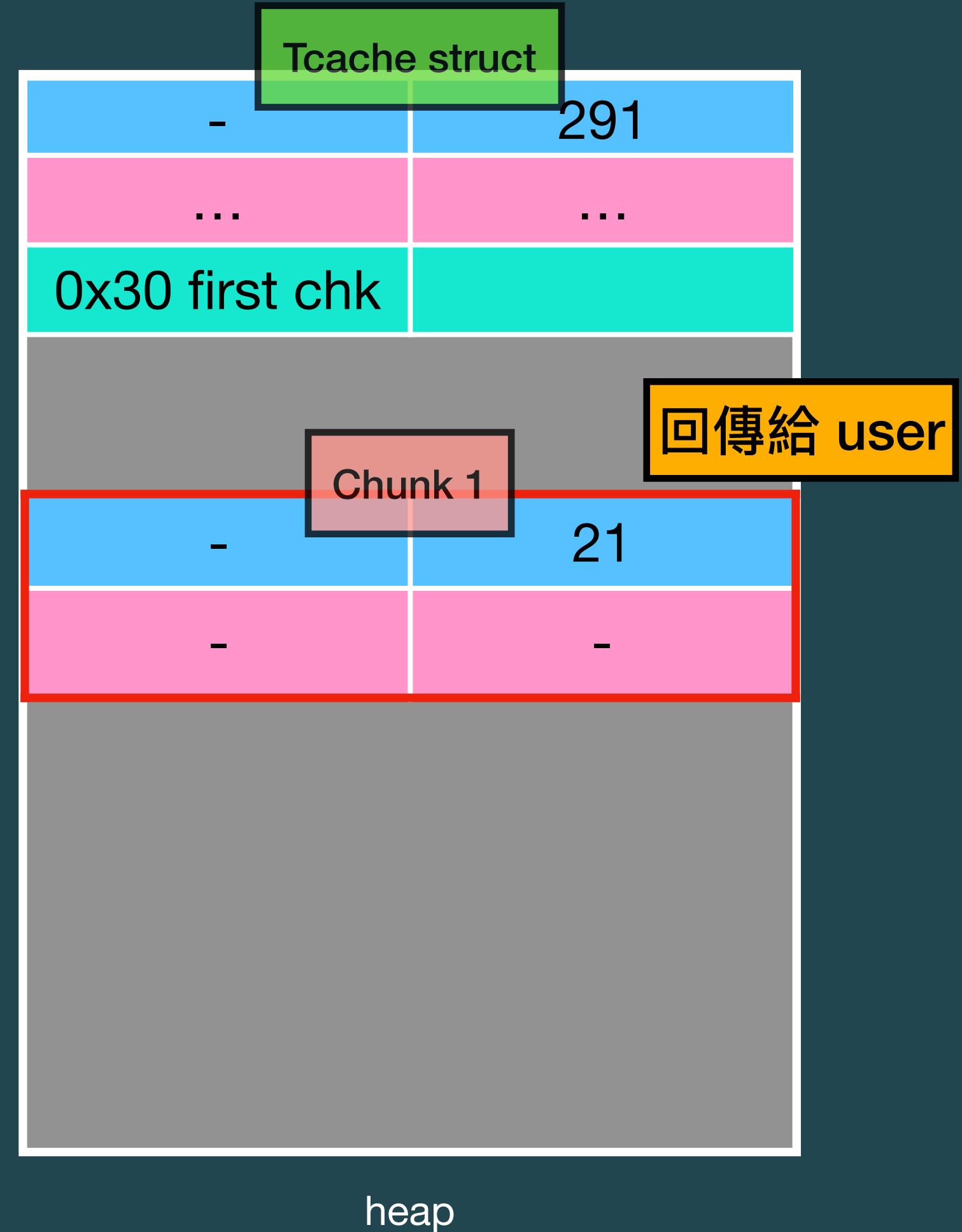
```
u1f383@u1f383:~$  
unsigned long *chk1 = malloc(0x10);  
void *chk2 = malloc(0x10);  
chk1[3] = 0x31;  
free(chk2);
```



\$ Vulnerability

Heap overflow - Example

```
u1f383@u1f383:~$  
unsigned long *chk1 = malloc(0x10);  
void *chk2 = malloc(0x10);  
chk1[3] = 0x31;  
free(chk2);
```

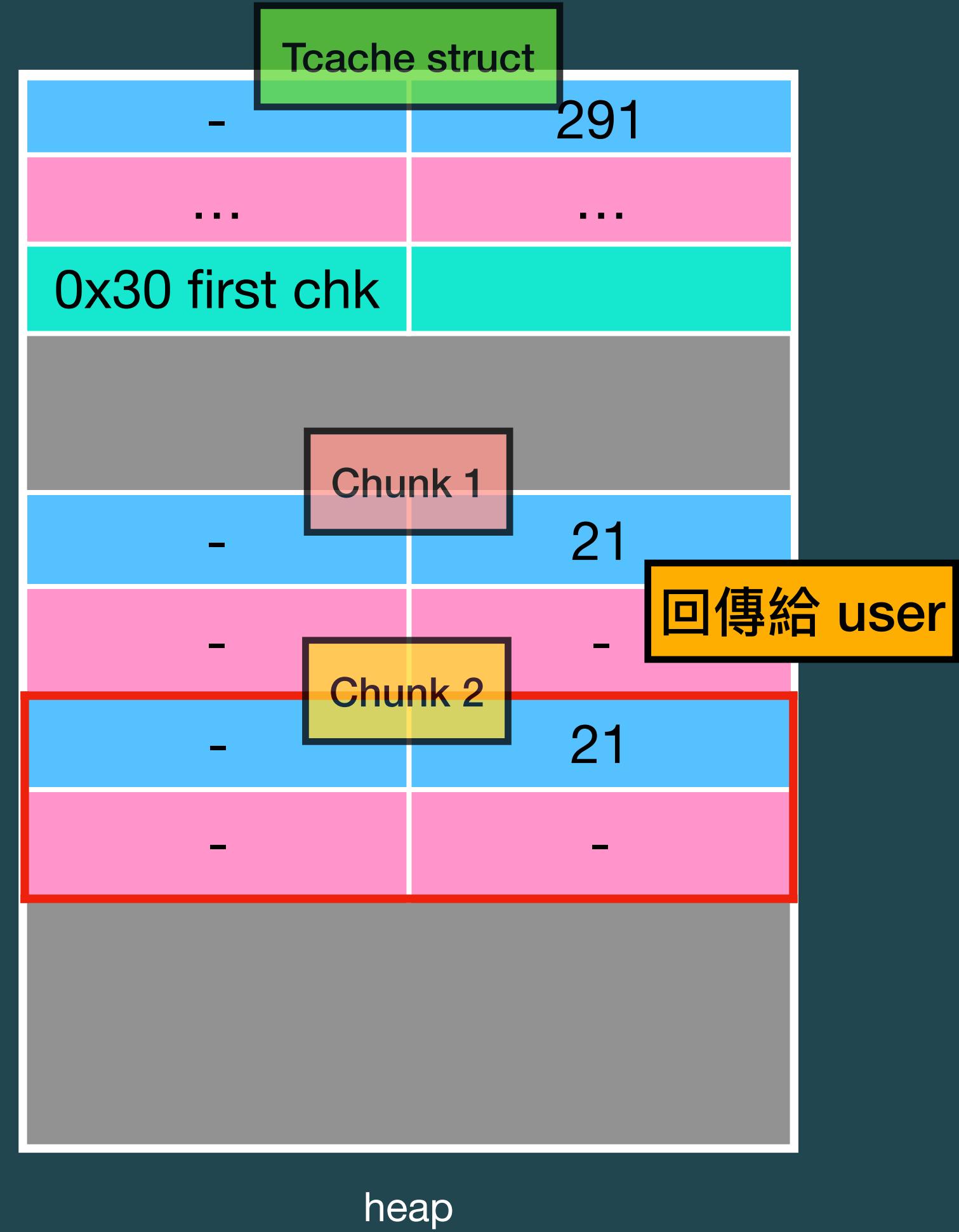


\$ Vulnerability

Heap overflow - Example

```
u1f383@u1f383:/
```

```
unsigned long *chk1 = malloc(0x10);
void *chk2 = malloc(0x10);
chk1[3] = 0x31;
free(chk2);
```

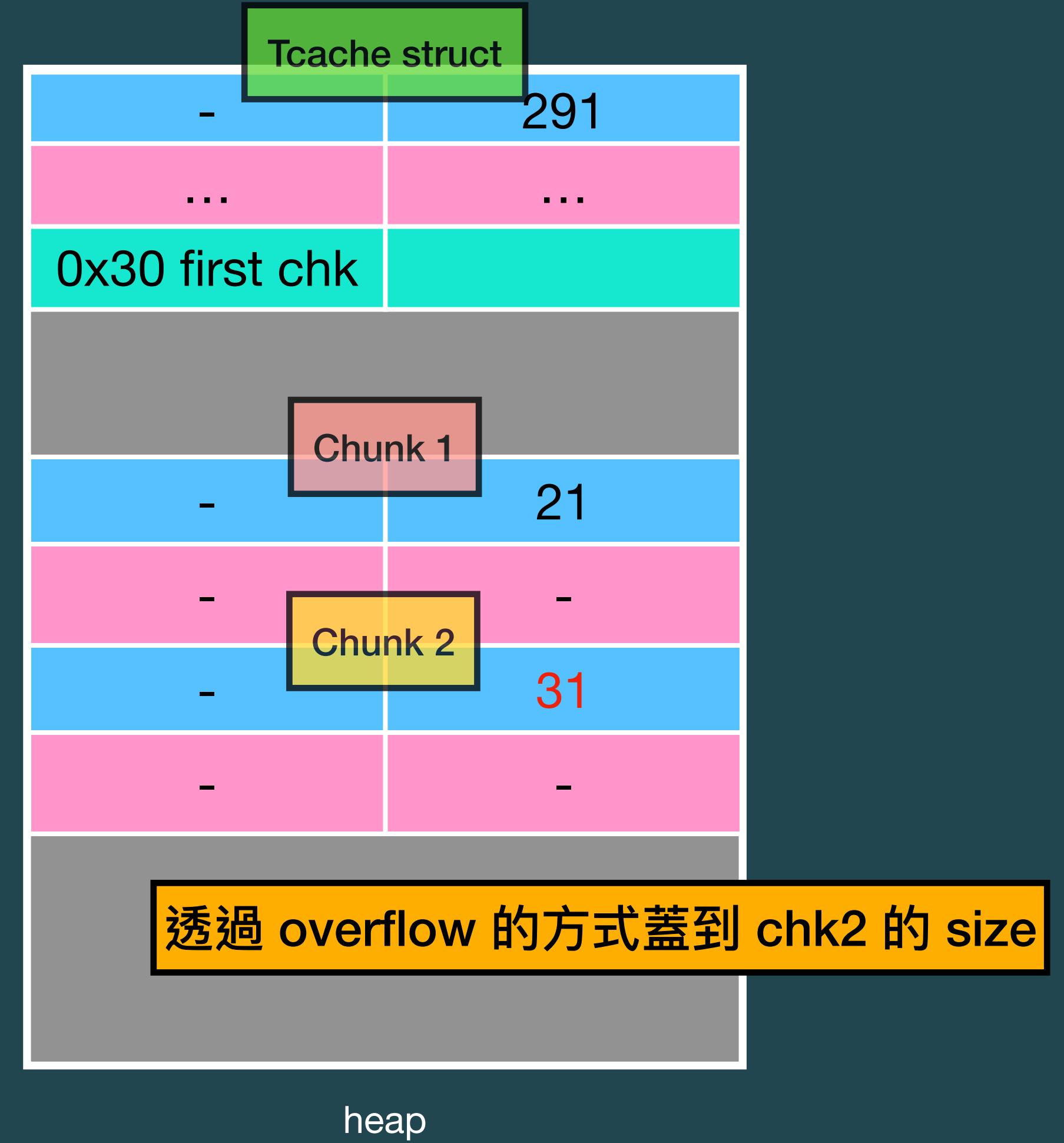


\$ Vulnerability

Heap overflow - Example

```
u1f383@u1f383:/
```

```
unsigned long *chk1 = malloc(0x10);
void *chk2 = malloc(0x10);
chk1[3] = 0x31;
free(chk2);
```

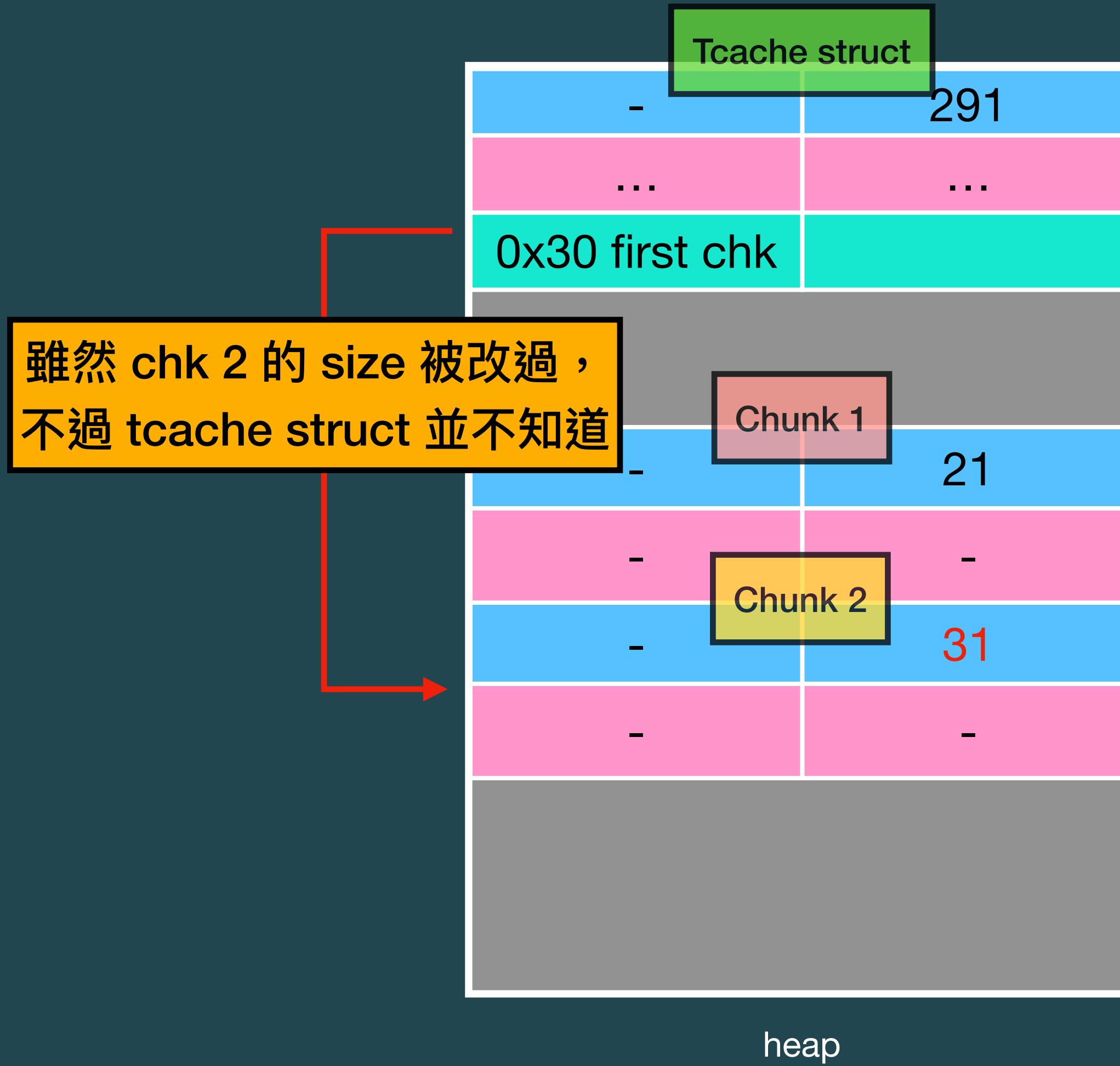


\$ Vulnerability

Heap overflow - Example

```
u1f383@u1f383:/
```

```
unsigned long *chk1 = malloc(0x10);
void *chk2 = malloc(0x10);
chk1[3] = 0x31;
free(chk2);
```



\$ Vulnerability

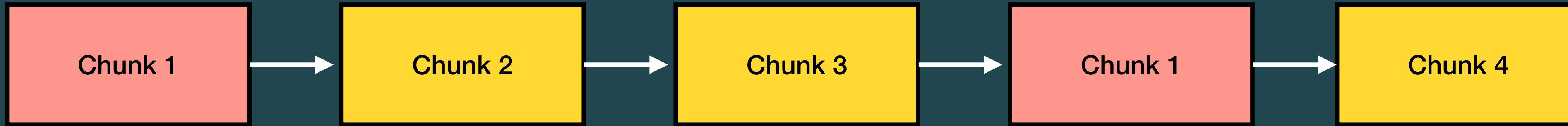
Heap overflow - Example



\$ Vulnerability

Double Free

- ▶ 當 chunk 被 free 兩遍，導致 bin 當中記錄了兩個相同的 chunk，就被稱作 double free
- ▶ 當透過 malloc 取出 chunk 時，因為 chunk 仍被 bin 所記錄，因此對 chunk 做修改就等於直接改動了 freed chunk
- ▶ 由於 double free 常發生，因此 glibc 當中有許多檢查機制來偵測這些異常的行為，如果要利用的話，就需要繞過那些檢查機制，讓 glibc 認為目前的行為是合法的

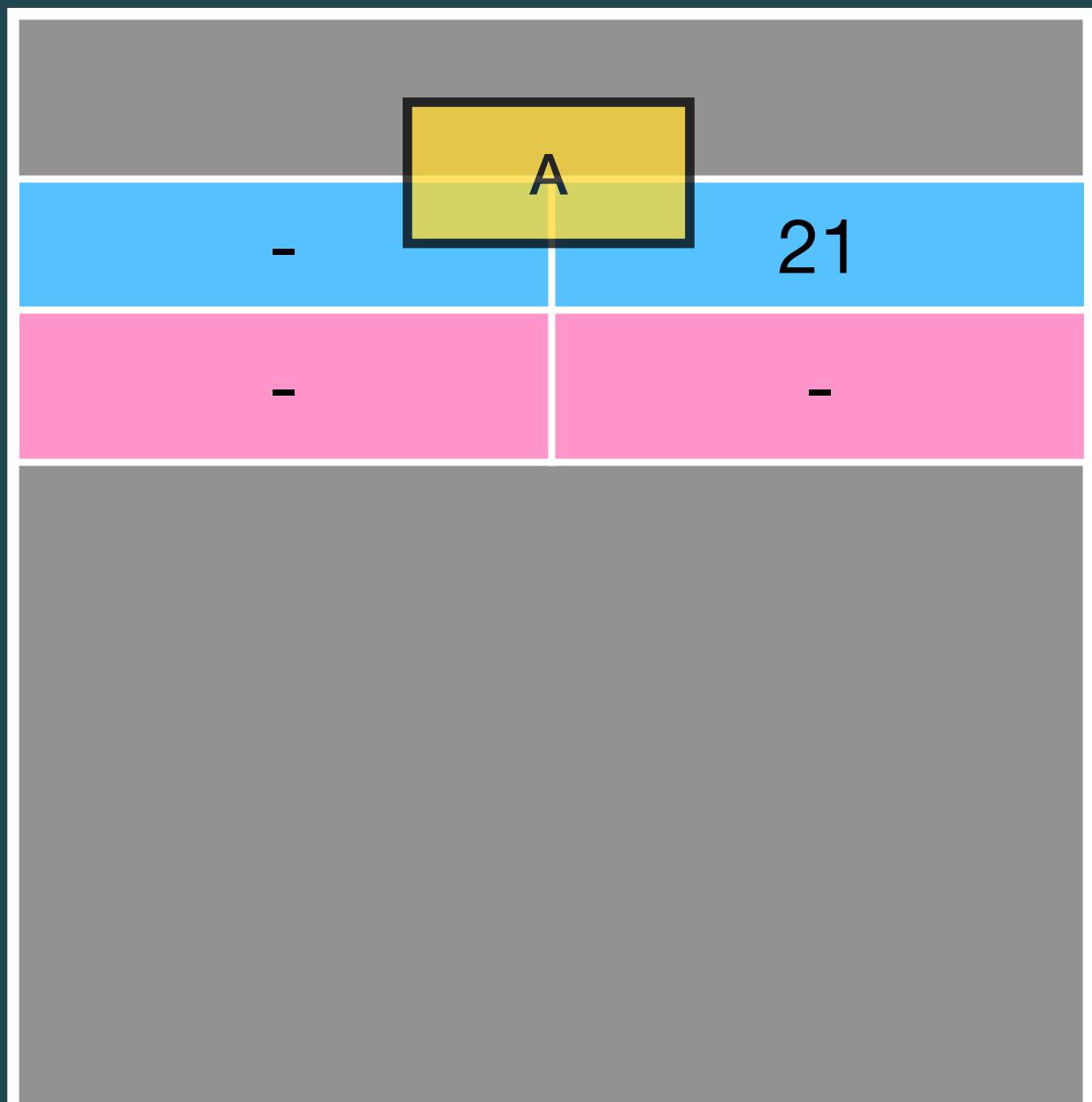
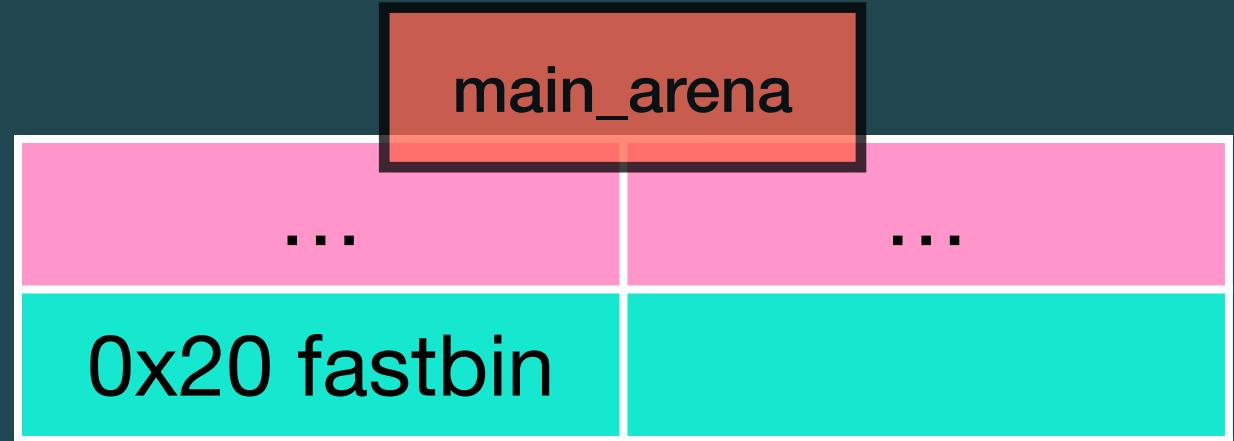


\$ Vulnerability

Double Free - Example

假設 tcache 已經被填滿

```
free(A);  
free(A);
```

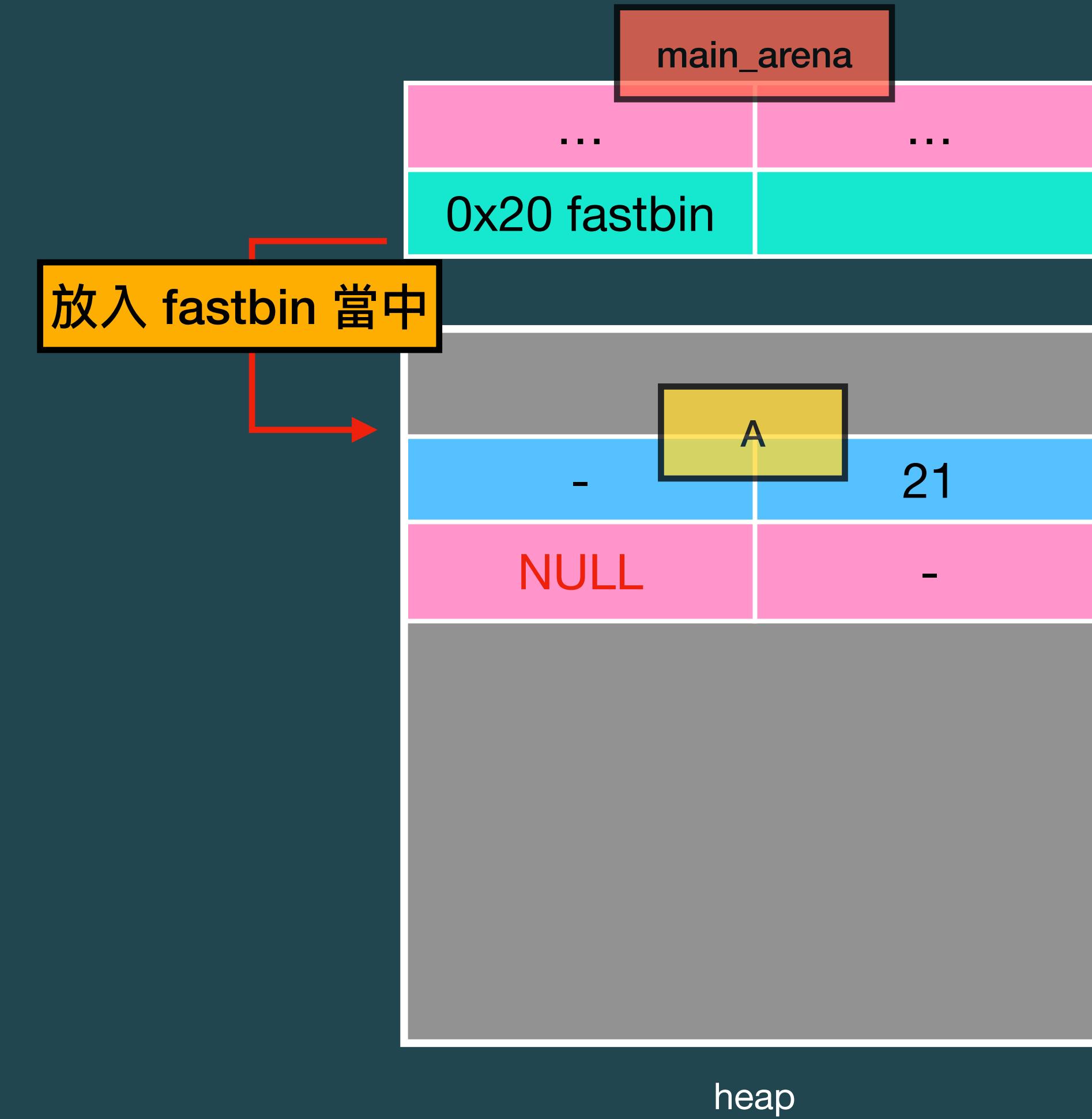


heap

\$ Vulnerability

Double Free - Example

```
free(A);  
free(A);
```

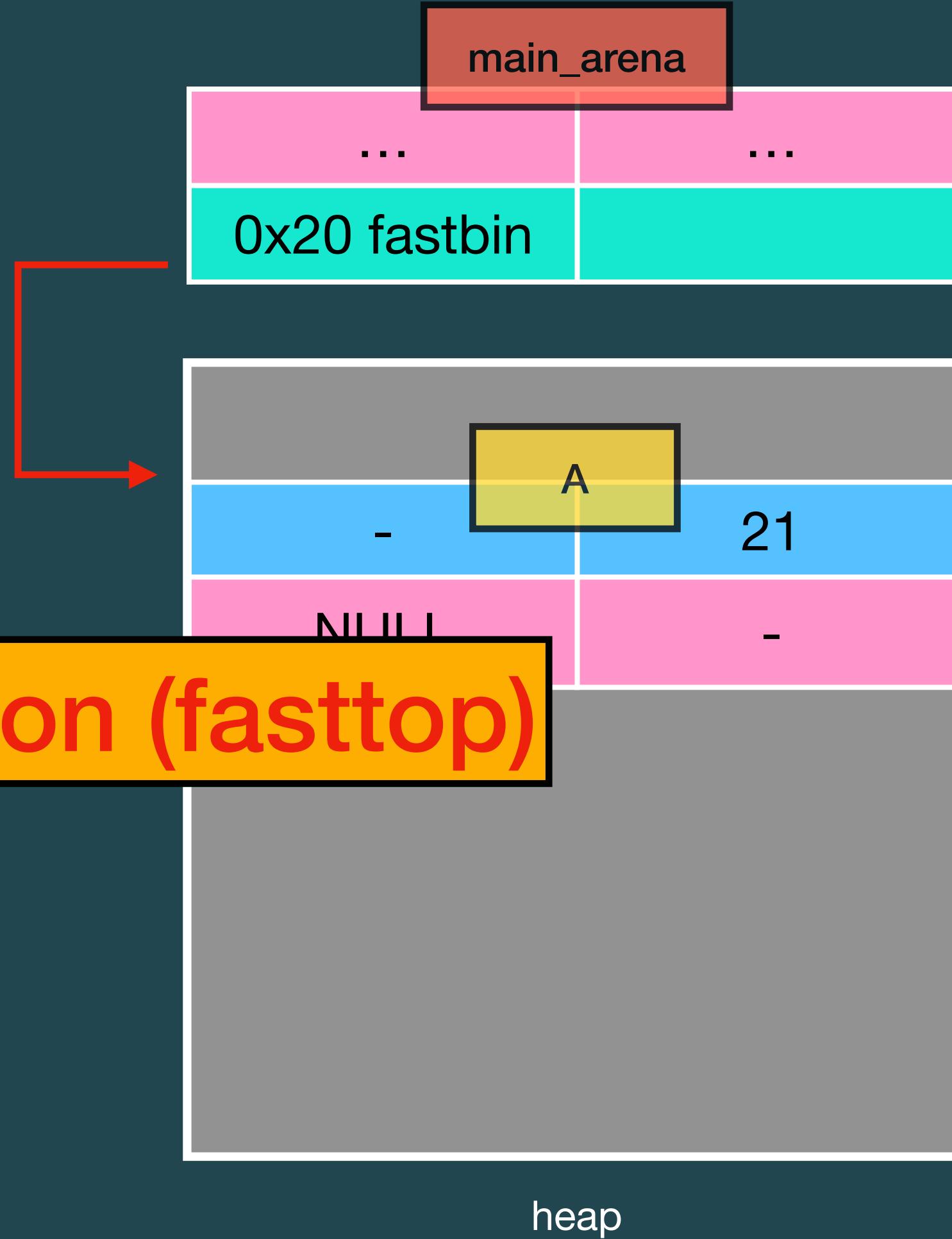


\$ Vulnerability

Double Free - Example

```
free(A);  
free(A);
```

double free or corruption (fasttop)



\$ Vulnerability

Double Free - Example

glibc 檢查到有非法行為

1. 取出對應大小的第一個 fastbin chunk

2. 比較是否與即將要釋放的 chunk 相同

```
free(A);
free(A);
```

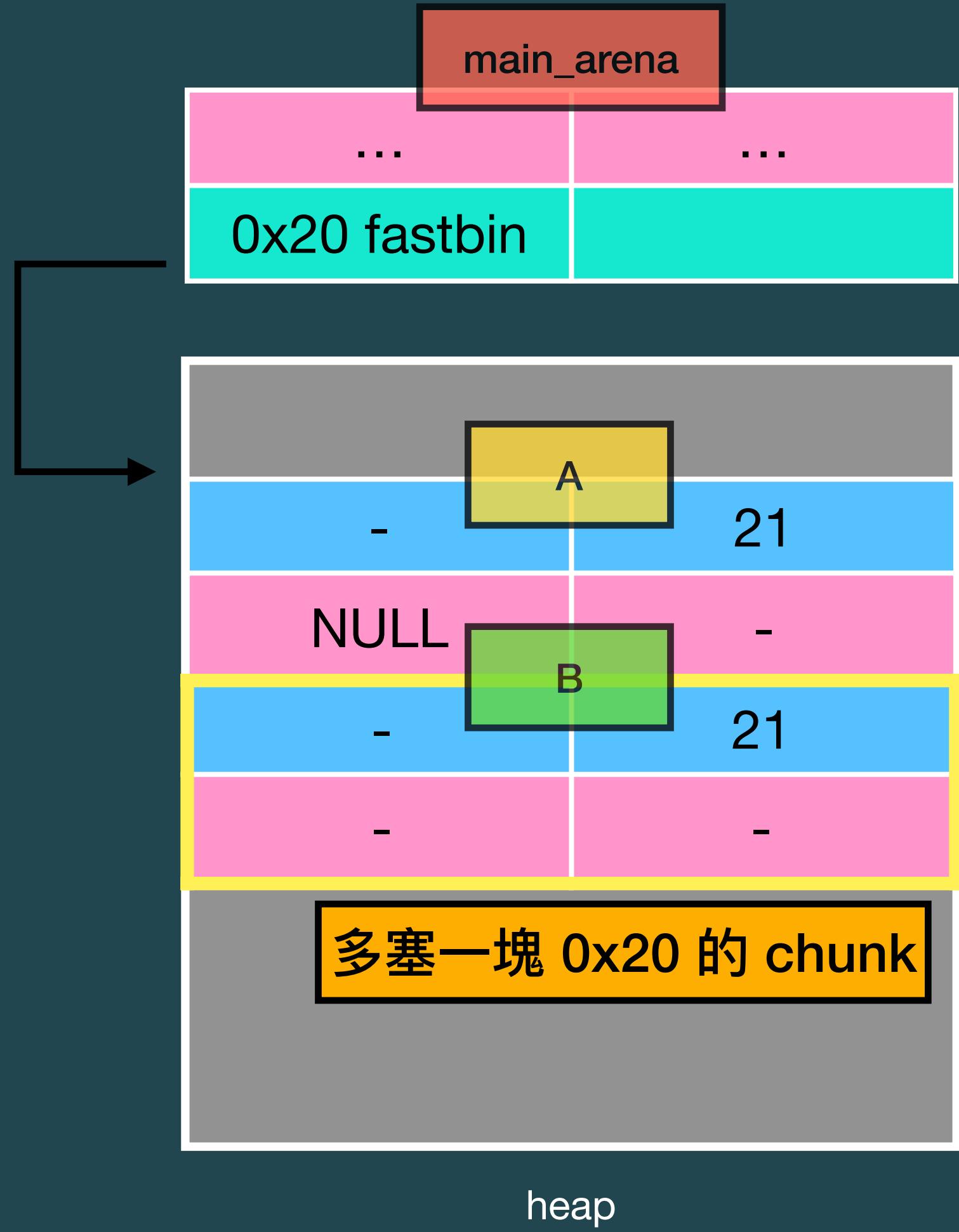
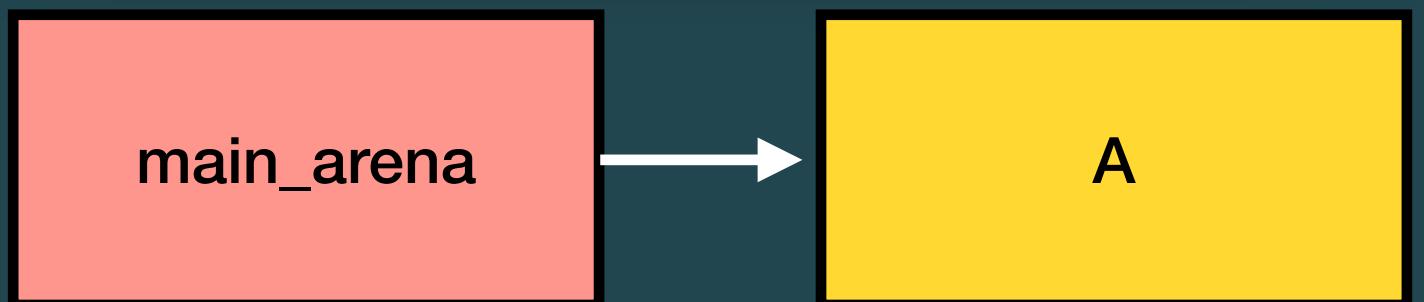
```
static void _int_free (mchunkptr p)
{
    ...
    size = chunksize (p);
    ...
    unsigned int idx = fastbin_index(size);
    fb = &fastbin (av, idx);
    mchunkptr old = *fb, old2;

    if (SINGLE_THREAD_P)
    {
        if (__builtin_expect (old == p, 0))
            malloc_printerr ("double free or corruption (fasttop)");
        ...
    }
}
```

\$ Vulnerability

Double Free - Example

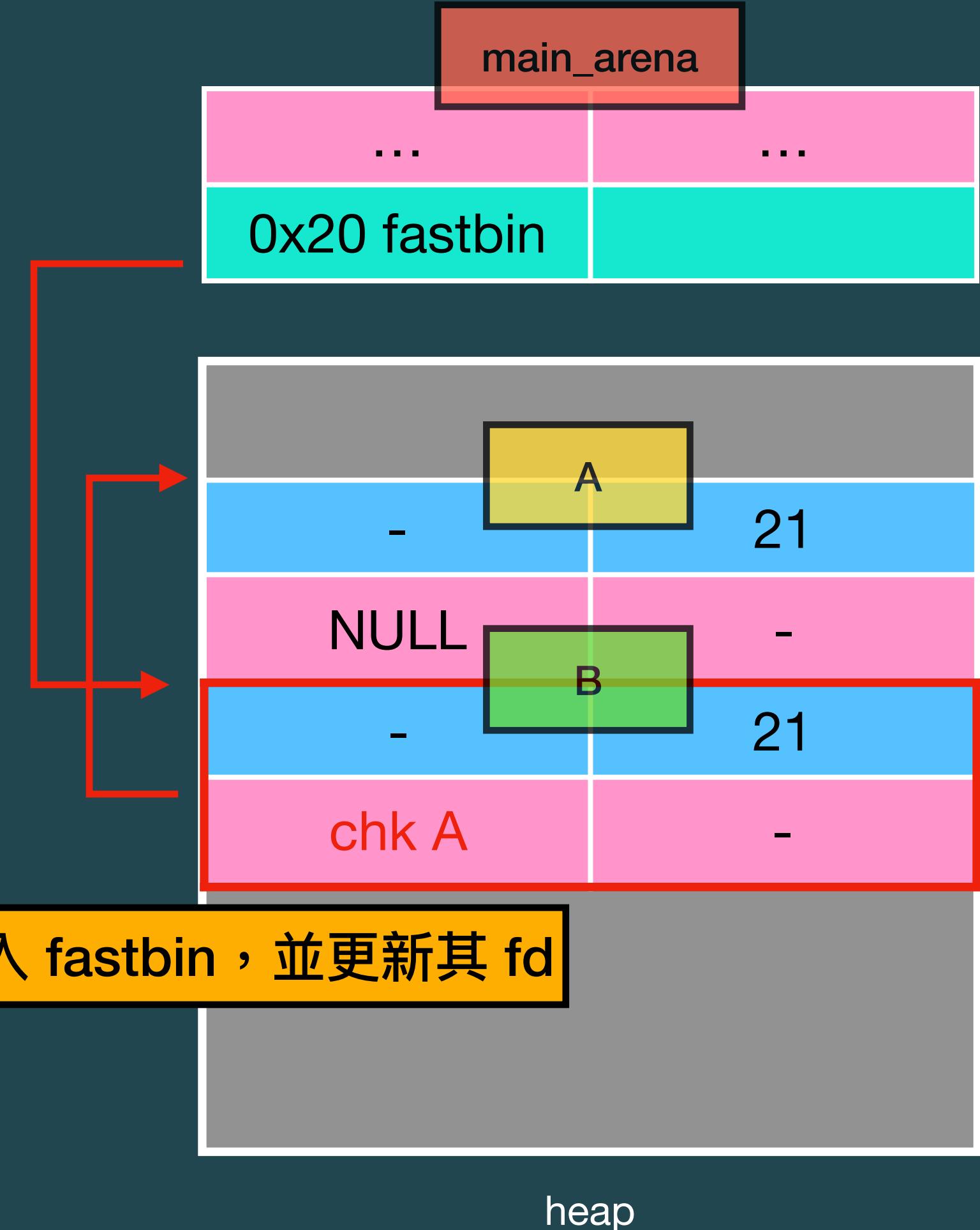
```
u1f383@u1f383:/ $ free(A);
free(B);
free(A);
// clean tcache
A = malloc(0x10);
*A = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xdeadbeef
```



\$ Vulnerability

Double Free - Example

```
u1f383@u1f383:/ $ free(A);
free(B);
free(A);
// clean tcache
A = malloc(0x10);
*A = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xdeadbeef
```



\$ Vulnerability

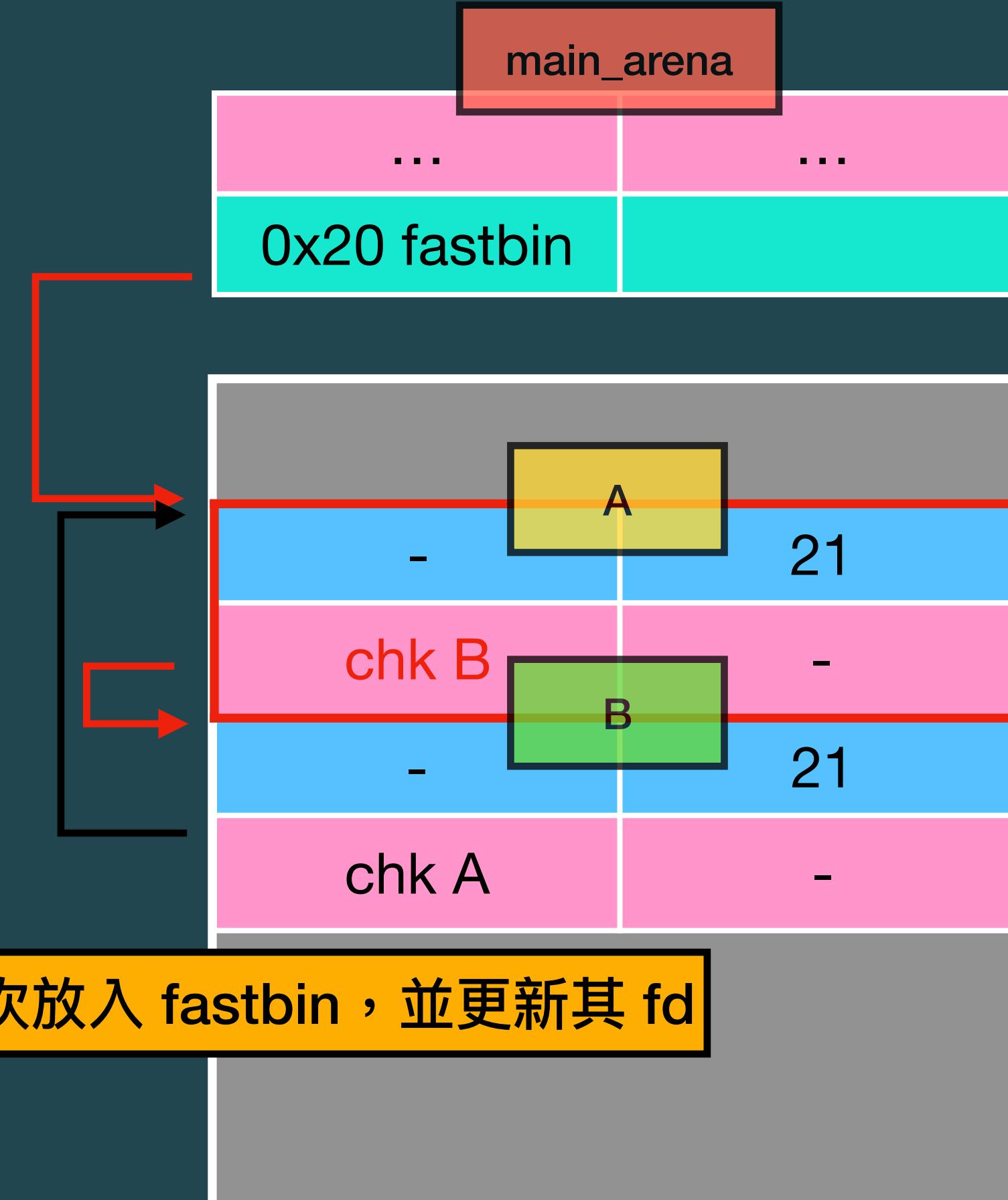
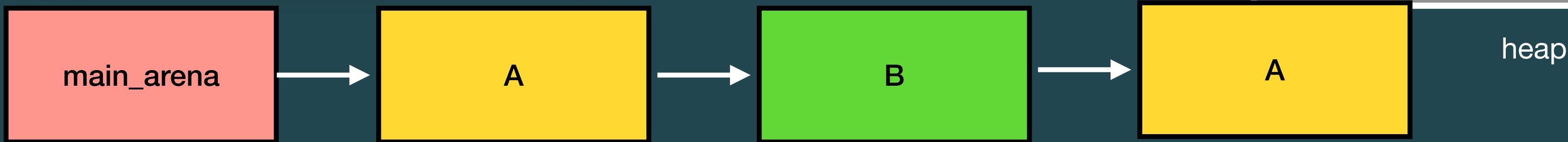
Double Free - Example



\$ Vulnerability

Double Free - Example

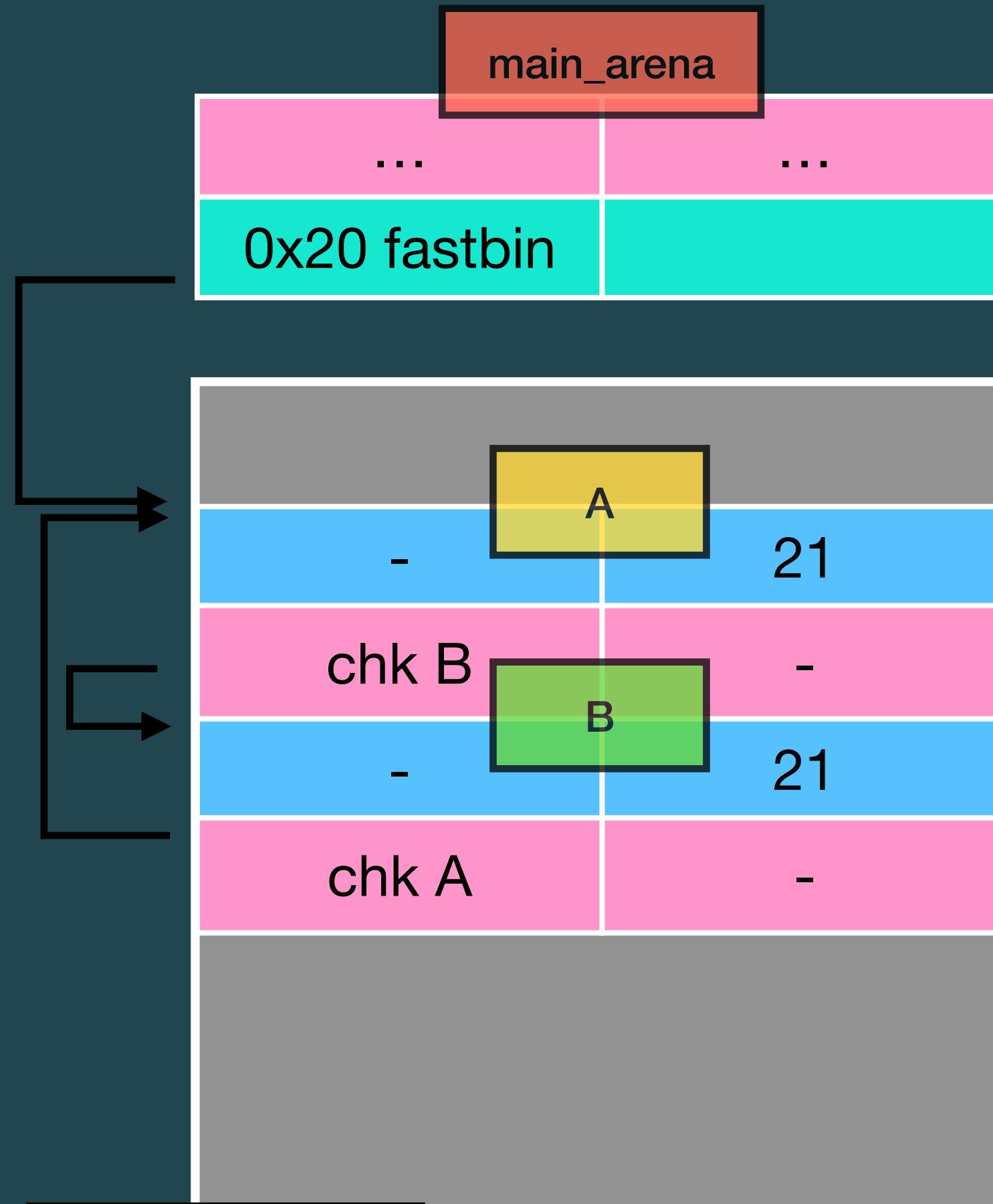
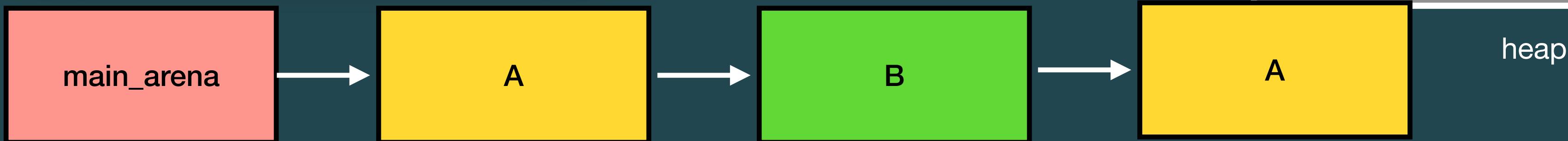
```
u1f383@u1f383:/ $ free(A);
free(B);
free(A);
// clean tcache
A = malloc(0x10);
*A = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xdeadbeef
```



\$ Vulnerability

Double Free - Example

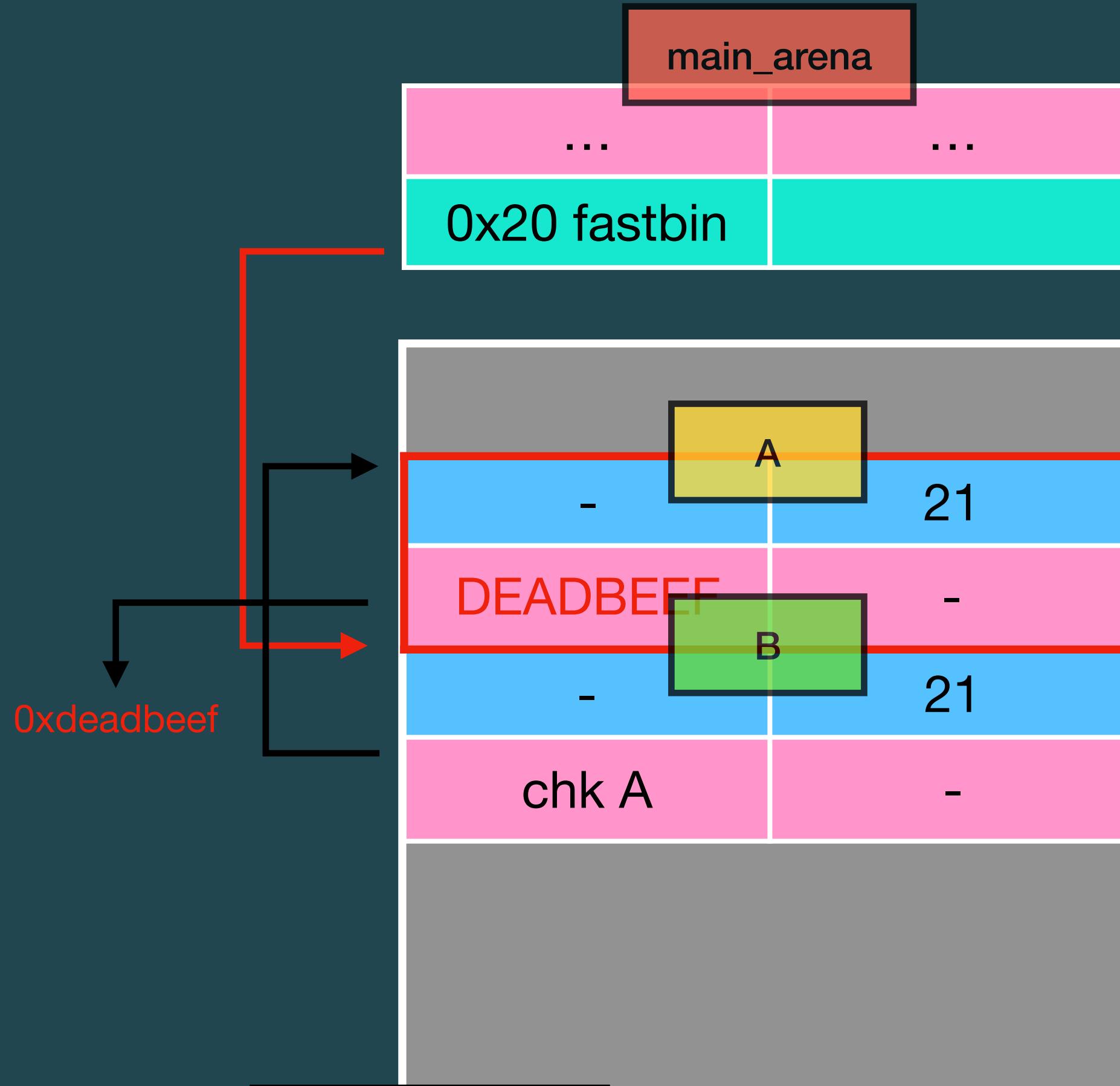
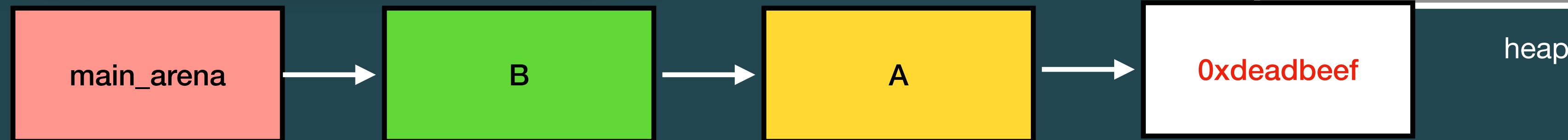
```
free(A);  
清空 tcache，確保 malloc 拿到 fastbin chunk  
// clean tcache  
A = malloc(0x10);  
*A = 0xdeadbeef;  
malloc(0x10);  
malloc(0x10);  
malloc(0x10); // 0xdeadbeef
```



\$ Vulnerability

Double Free - Example

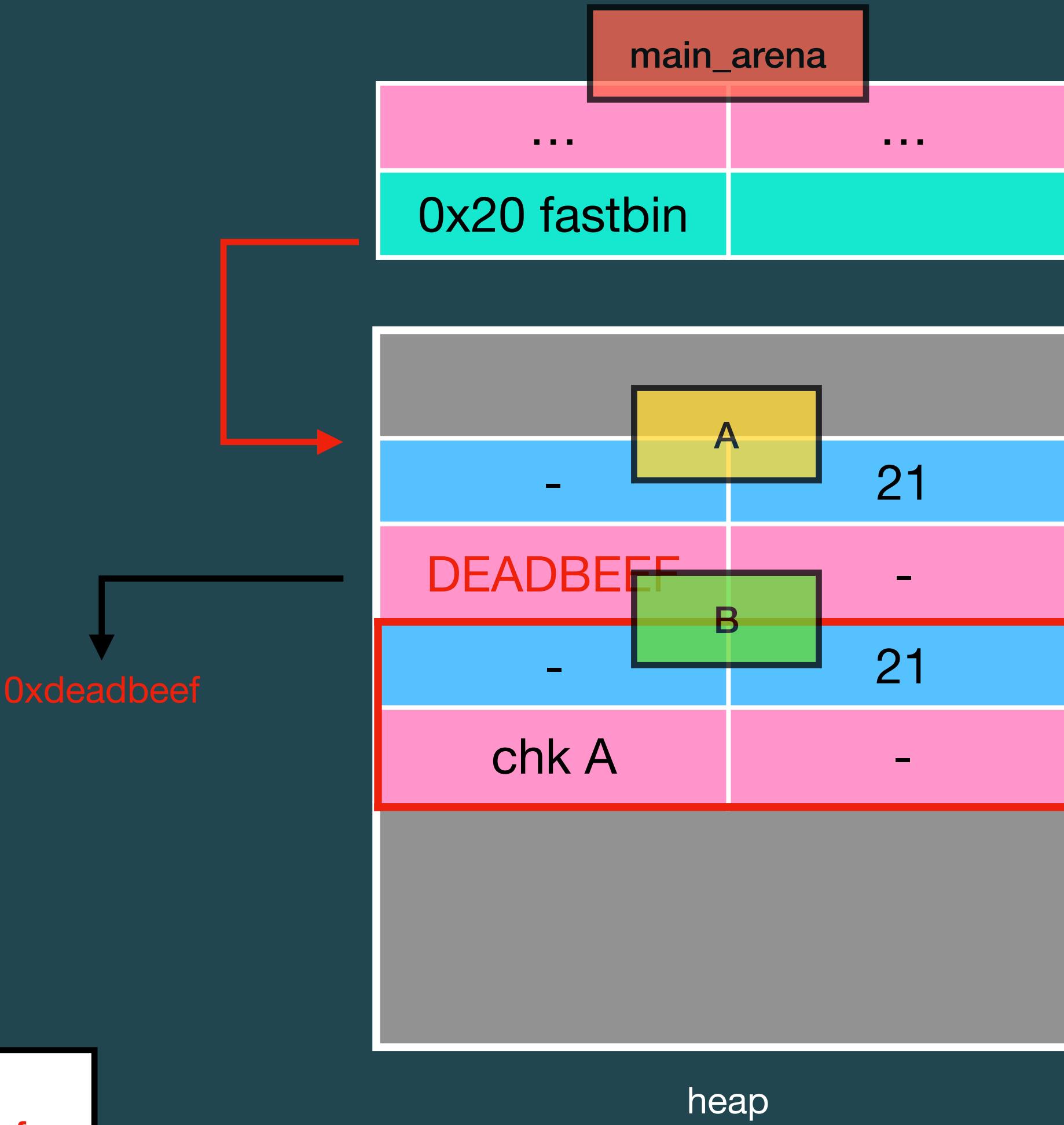
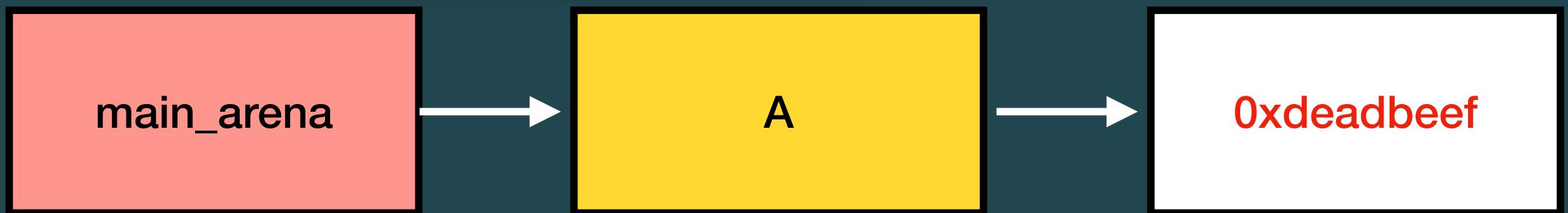
```
u1f383@u1f383:/ $ free(A);
free(B):
取出 chk A，並且將 fd 改成 0xdeadbeef
A = malloc(0x10);
*A = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xdeadbeef
```



\$ Vulnerability

Double Free - Example

```
u1f383@u1f383:/ $ free(A);  
free(B);  
free(A);  
// clean tcache  
A = malloc(0x10);  
*A = 0xdeadbeef;  
malloc(0x10); 取出 chk B  
malloc(0x10);  
malloc(0x10); // 0xdeadbeef
```



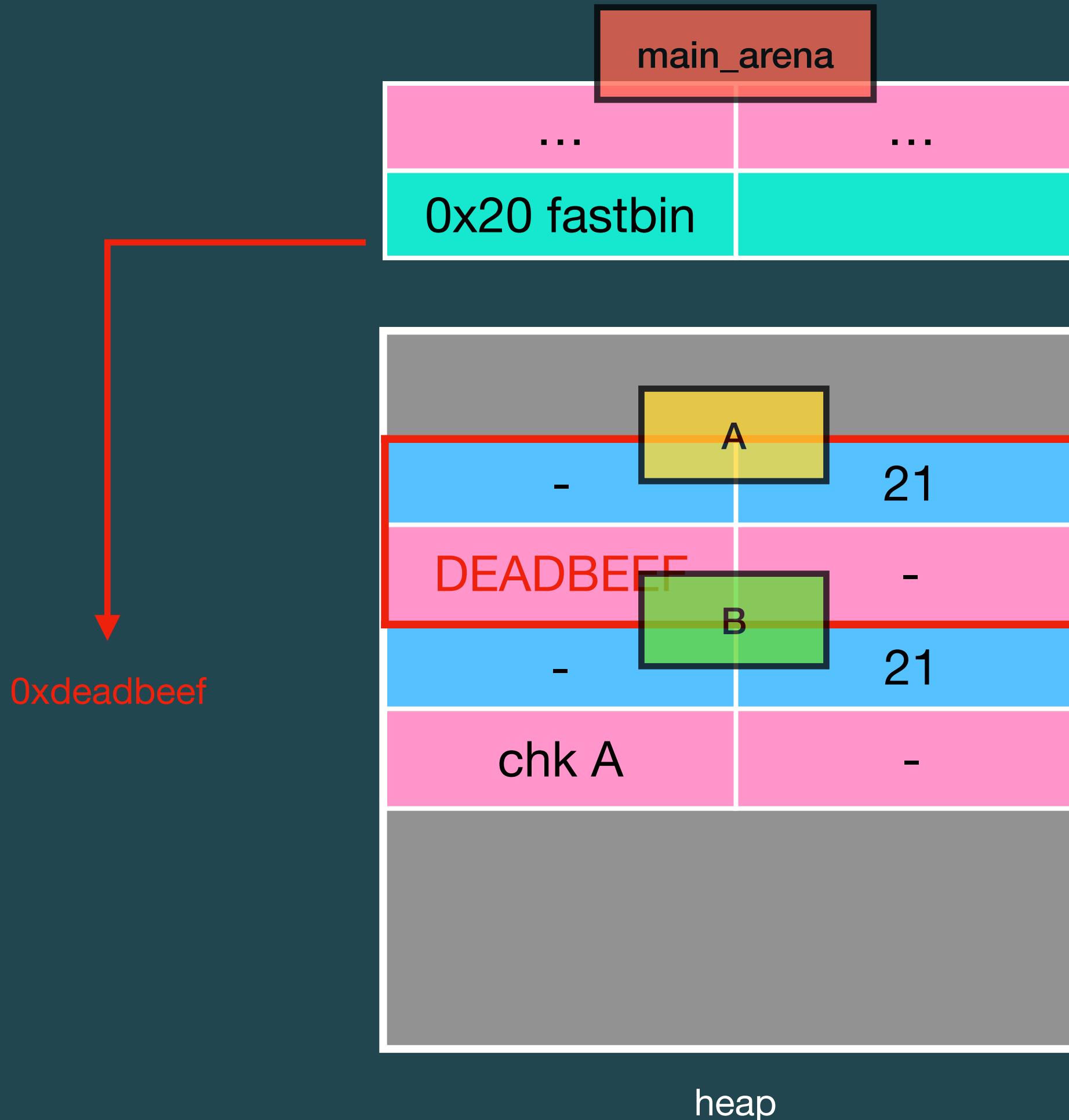
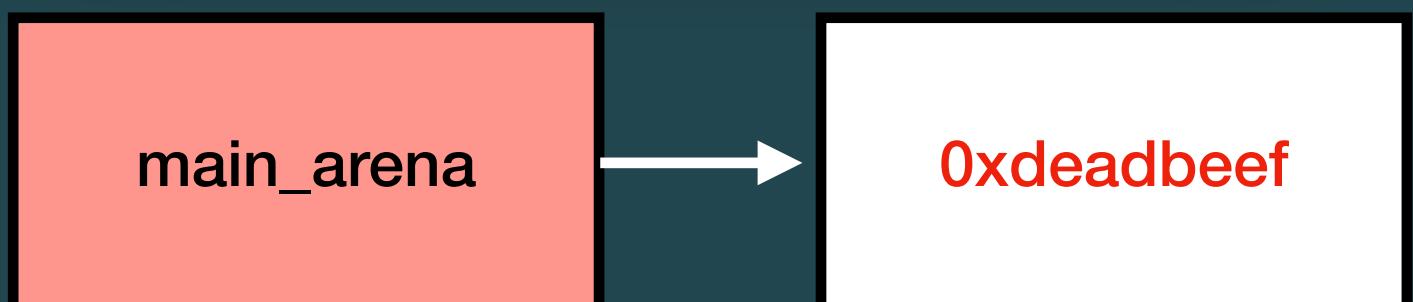
\$ Vulnerability

Double Free - Example

```
u1f383@u1f383:/ $ free(A);  
free(B);  
free(A);  
// clean tcache  
A = malloc(0x10);  
*A = 0xdeadbeef;
```

再次取出 chk A，此時 0x20 fastbin 指向 0xdeadbeef

→ malloc(0x10); // 0xdeadbeef



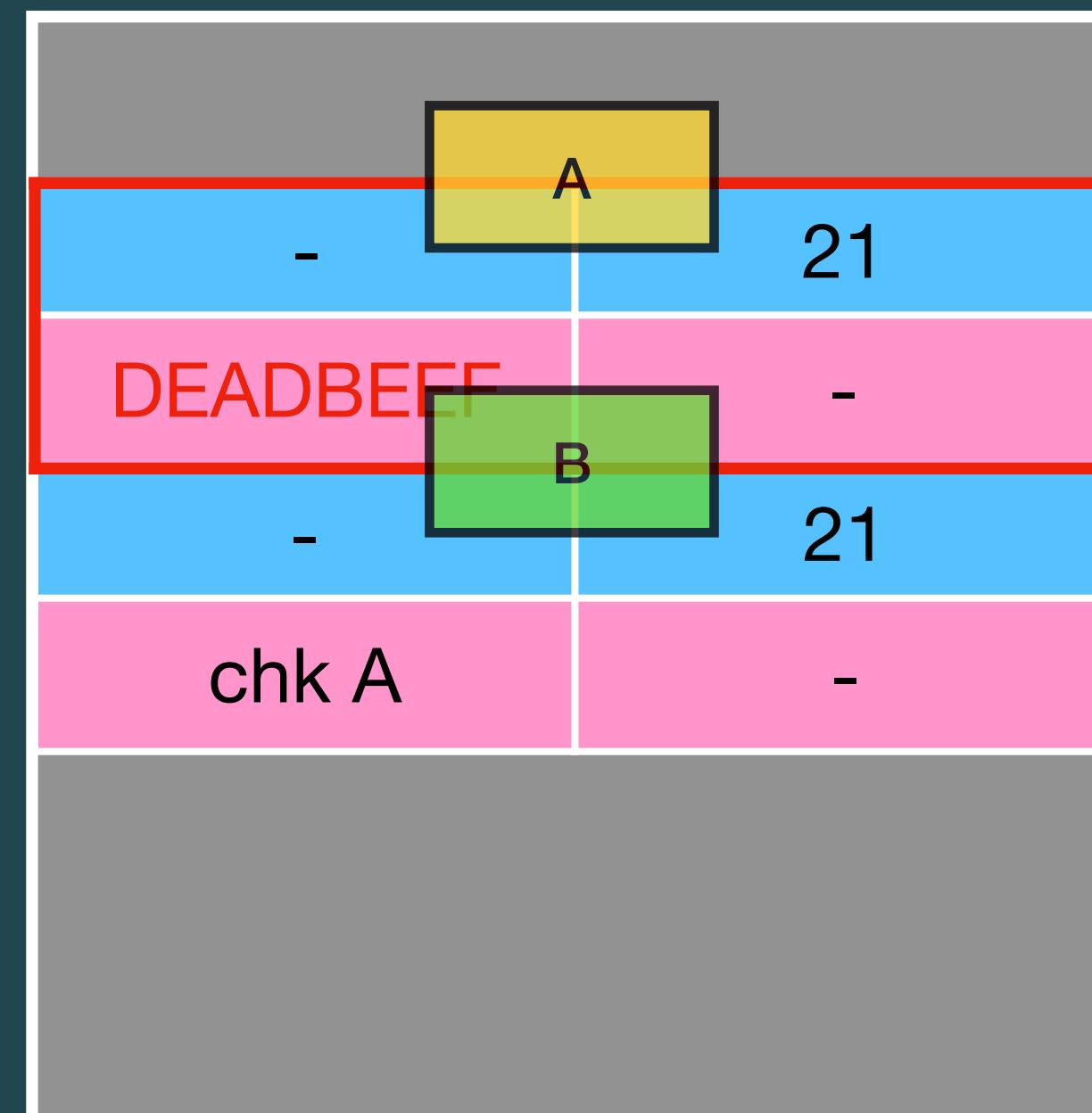
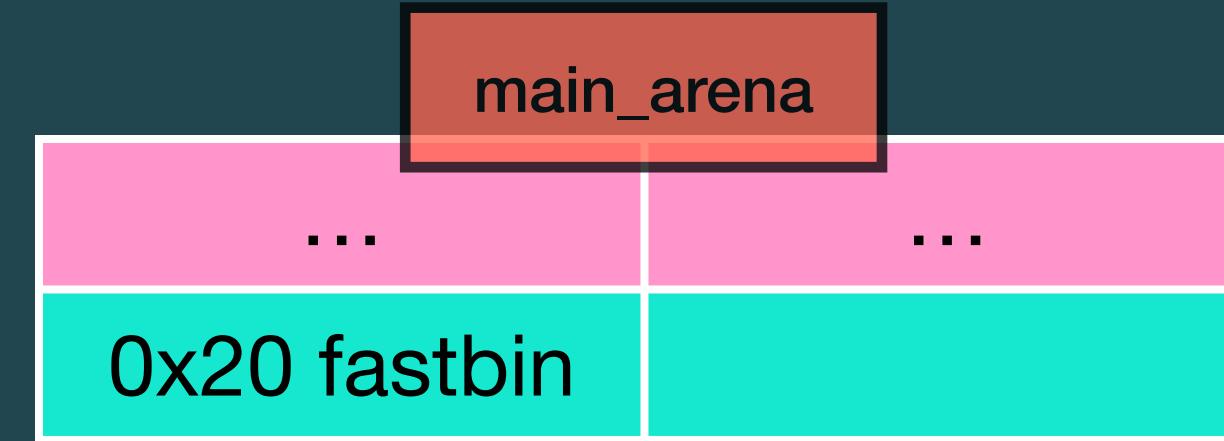
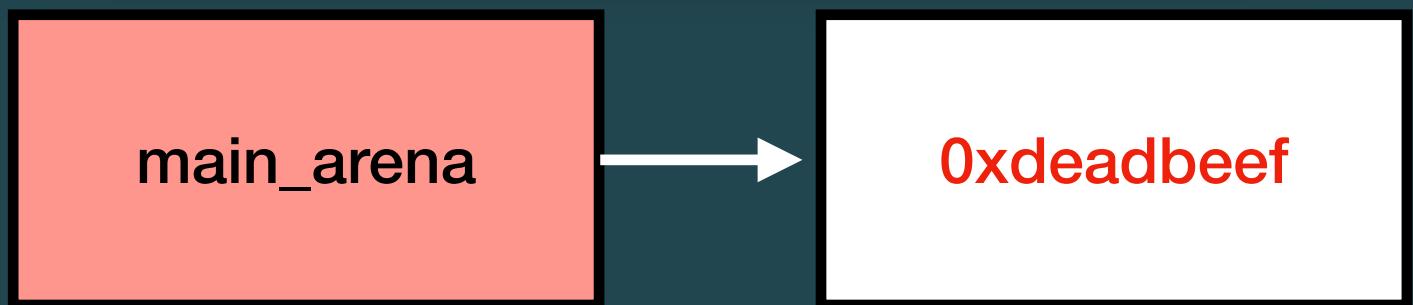
\$ Vulnerability

Double Free - Example

```
u1f383@u1f383:/
```

```
$ free(A);
free(B);
free(A);
// clean tcache
A = malloc(0x10);
*A = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xdeadbeef
```

程式 crash，因為 0xdeadbeef 為 invalid address



heap

\$ Vulnerability



\$ Vulnerability

Lab - Market

- ▶ 程式設計中需要好好的**初始化**，並且在釋放記憶體時清除相關敏感資料
- ▶ 若初始化的流程是使用者可以控制的，則很有可能透過**殘留在記憶體內的資料**做利用



Exploitation goal

\$ Exploitation goal

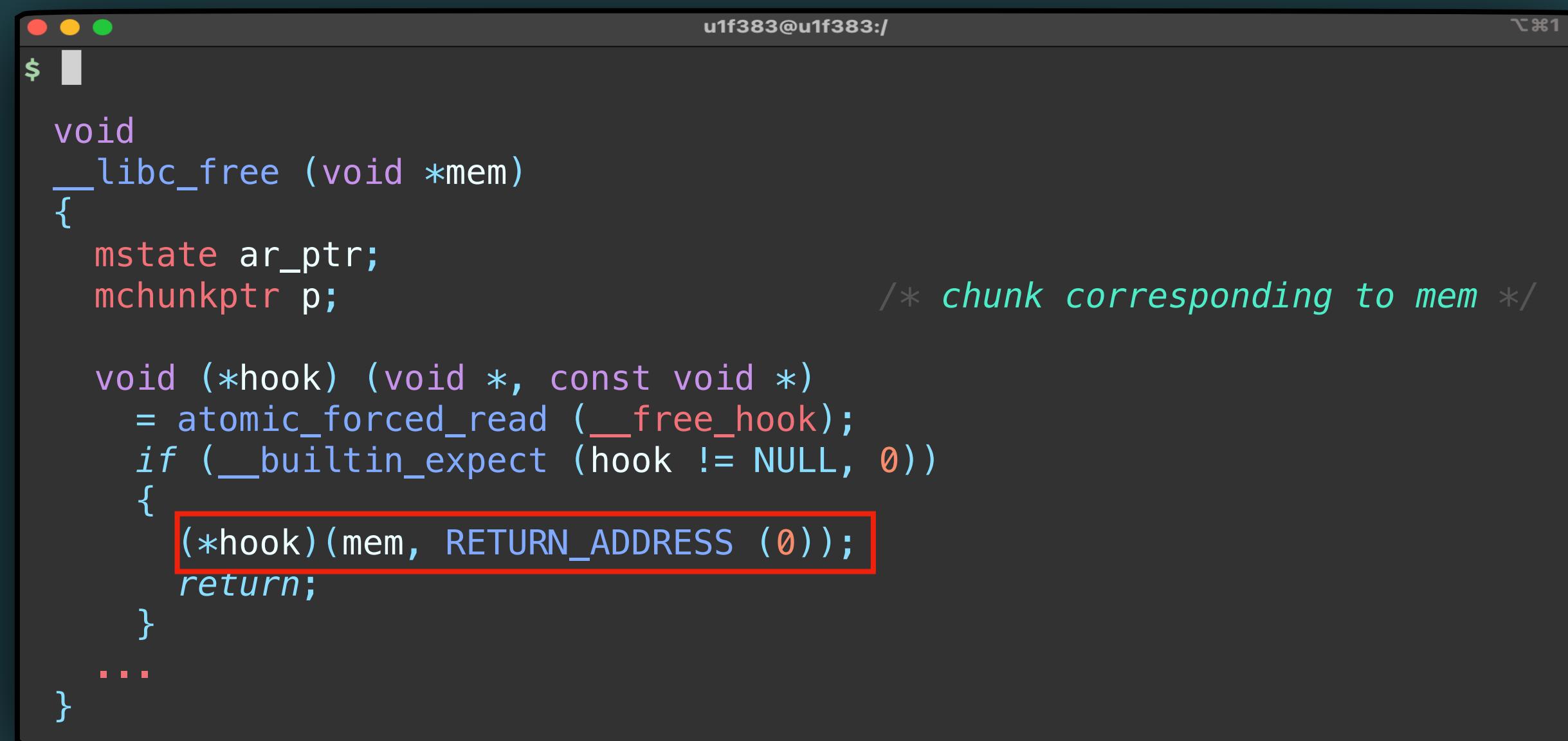
hook

- ▶ 跟記憶體分配相關的 function 都可以加上 hook 來自定義行為
- ▶ 常見的有 `_free_hook`、`_malloc_hook` 以及 `_realloc_hook`
- ▶ 透過定義這些 hook，可以讓程式在呼叫 hooked function 時，跳轉至我們在 hook 中所寫入的位址

\$ Exploitation goal

__free_hook

- ▶ __free_hook 不為 NULL 時，內容可以作為 function pointer 取代 free 的行為
- ▶ 例如寫入 system 的位址後，若 chunk 內容為 “/bin/sh”，則在呼叫 free 時會變成執行 system(“/bin/sh”)



```
void
__libc_free (void *mem)
{
    mstate ar_ptr;
    mchunkptr p; /* chunk corresponding to mem */

    void (*hook) (void *, const void *) = atomic_forced_read (__free_hook);
    if (__builtin_expect (hook != NULL, 0))
    {
        (*hook) (mem, RETURN_ADDRESS (0));
        return;
    }
    ...
}
```

\$ Exploitation goal

__malloc_hook, __realloc_hook

- ▶ __malloc_hook 與 __realloc_hook 不為 NULL 時，內容可以作為 function pointer 取代 malloc / realloc 的行為
- ▶ 寫入 one gadget，在呼叫 malloc / realloc 時被 trigger

```
u1f383@u1f383:/
```

```
$ void *
__libc_malloc (size_t bytes)
{
    mstate ar_ptr;
    void *victim;

    _Static_assert (PTRDIFF_MAX <= SIZE_MAX / 2,
                   "PTRDIFF_MAX is not more than half of SIZE_MAX");

    void *(*hook) (size_t, const void *)
        = atomic_forced_read (__malloc_hook);
    if (__builtin_expect (hook != NULL, 0))
        return (*hook)(bytes, RETURN_ADDRESS (0));
    ...
}
```

__malloc_hook

```
u1f383@u1f383:/
```

```
$ void *
__libc_realloc (void *oldmem, size_t bytes)
{
    mstate ar_ptr;
    INTERNAL_SIZE_T nb; /* padded request size */

    void *newp; /* chunk to return */

    void *(*hook) (void *, size_t, const void *) =
        atomic_forced_read (__realloc_hook);
    if (__builtin_expect (hook != NULL, 0))
        return (*hook)(oldmem, bytes, RETURN_ADDRESS (0));
    ...
}
```

__realloc_hook



Exploitation tech

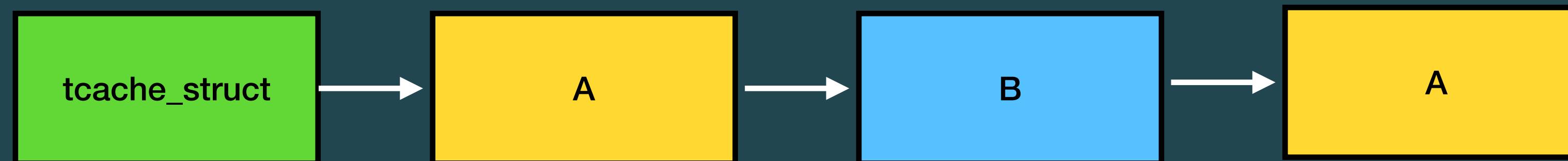
\$ Exploitation tech

- ▶ 由於攻擊手法會根據功能跟情況有所不同，在此只介紹 how2heap repo 所記錄 glibc-2.31 的攻擊方法中，常見的三種利用方式
 - ⦿ Tcache poisoning
 - ⦿ Fastbin attack (fastbin dup)
 - ⦿ Overlapping chunks

\$ Exploitation tech

Tcache poisoning

- ▶ **Explanation** - 使用 double free 讓 tcache 當中存在兩個相同的 chunk，並利用修改 fd 的方式，將對應位址視為 chunk 分配給 user
 - ⌚ Tcache 拿 chunk 時並不會檢查 chunk size 是否合法，因此常會拿 __free_hook 寫 system
- ▶ **Protection1** - 當釋放 chunk 時，如果 chunk + 8 (key) 位置的值與當前 heap 的 &tcache_struct 相等，則會遍歷所有 entry，檢查是否有相同的 chunk，確保沒有 double free 的發生



\$ Exploitation tech

Tcache poisoning

```
u1f383@u1f383: ~ % $ static void _int_free (... , mchunkptr p) { ... tcache_entry *e = (tcache_entry *) chunk2mem (p); if (__glibc_unlikely (e->key == tcache)) { tcache_entry *tmp; LIBC_PROBE (memory_tcache_double_free, 2, e, tc_idx); for (tmp = tcache->entries[tc_idx]; tmp; tmp = tmp->next) if (tmp == e) malloc_printerr ("free(): double free detected in tcache 2"); ... }
```

1. 將要釋放的 chunk 視為 tcache entry，
並檢查 key 欄位是否為 tcache struct 的位址

2. 如果是的話，則遍歷所有 entry，檢查此 chunk 使否已經存在

\$ Exploitation tech

Tcache poisoning

- ▶ Protection2 - 當取出 chunk 時，會檢查對應大小的 counter 是否大於 0，如果是的話才會取出 tcache_struct 當中指向的第一塊 chunk

```
$ void *_libc_malloc (size_t bytes)
{
    ...
    size_t tbytes;
    if (!checked_request2size (bytes, &tbytes)) { ... }
    size_t tc_idx = cslice2tidx (tbytes);
    ...
    if (tc_idx < mp_.tcache_bins
        && tcache
        && tcache->counts[tc_idx] > 0)
    {
        return tcache_get (tc_idx);
    }
    ...
}
```

1. 取得請求大小對應到的 index

2. 檢查對應 index 是否還有 chunk

\$ Exploitation tech

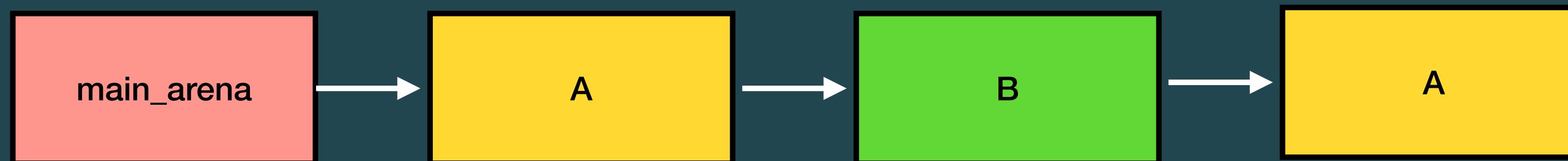
Tcache poisoning

- ▶ **Bypass Protection1** - 透過 UAF 或是 heap overflow，修改 chunk 的 key 欄位
- ▶ **Bypass Protection2**
 - ⦿ 拿到 tcache_struct 的 chunk 後修改 counts 欄位成非 0 的值
 - ⦿ 多次 free 相同的 chunk

\$ Exploitation tech

Fastbin Attack

- ▶ **Explanation** - 使用 double free 讓 tcache 當中存在兩個相同的 chunk，並利用修改 fd 的方式，將對應位址視為 chunk 分配給 user
- ▶ **Protection1** - 當釋放 chunk 時，檢查 fastbin 的第一個 chunk 與此 chunk 是否相同，確保沒有 double free 的發生 (link)
- ▶ **Protection2** - 在取得 chunk 時，會檢查請求大小是否與被回傳的 chunk 的大小相同



\$ Exploitation tech

Fastbin Attack

```
u1f383@u1f383:~ % $ static void *_int_malloc (... , size_t bytes)
{
    ...
    if (!checked_request2size (bytes, &nb)) { ... }
    if ((unsigned long) (nb) <= (unsigned long) (get_max_fast ()))
    {
        idx = fastbin_index (nb);
        mfastbinptr *fb = &fastbin (av, idx);
        mchunkptr pp;
        victim = *fb;

        if (victim != NULL)
        {
            *fb = victim->fd;
            if (__glibc_likely (victim != NULL))
            {
                size_t victim_idx = fastbin_index (chunksize (victim));
                if (__builtin_expect (victim_idx != idx, 0))
                    malloc_printerr ("malloc(): memory corruption (fast)");
            }
        }
    }
}
```

1. 取得 chunk size，並檢查是否位於 fastbin 的範圍內

2. 取得第一個 fastbin chunk，存在變數 **victim**

3. 檢查 victim 對應到的 fastbin index 是否與當前請求的 size 相同，
舉例來說，請求 0x30 但是卻拿到 0x40 大小的 chunk

\$ Exploitation tech

Fastbin Attack

- ▶ **Bypass Protection1** - 能利用 free(a); free(b); free(a); 的方式繞過檢查
- ▶ **Bypass Protection2**
 - ⦿ 因為 library address 都為 **0x7f** 開頭，而 0x70 為合法的 fastbin size，因此可以通過 fastbin 的保護機制
 - ⦿ 而 __malloc_hook 上方會有 library address，因此可以先取得上方的 chunk，在寫 **one gadget** 到 __malloc_hook 來做 exploit

\$ Exploitation tech

Overlapping chunks

- ▶ **Explanation** - 透過修改 chunk size，讓 chunk 在被釋放時 trigger **consolidation**，使得正在使用的 chunk 與已經釋放的 chunk 有部分重疊，也就代表
 - ⦿ 使用中的 chunk 可以更改 freed chunk 中的 fd、bk
 - ⦿ freed chunk 在被分配時，會分配到與使用中的 chunk 相同的區塊，可以修改敏感資料
- ▶ **Consolidation** - 當釋放記憶體時，若檢查到相鄰的 chunk 沒有被使用，會將其合併成一塊更大的 freed chunk
 - ⦿ 為 glibc 用來減少 heap fragmentation 的機制
 - ⦿ Chunk 在被合併前，會先透過 **unlink** 的機制來離開原本的 bin

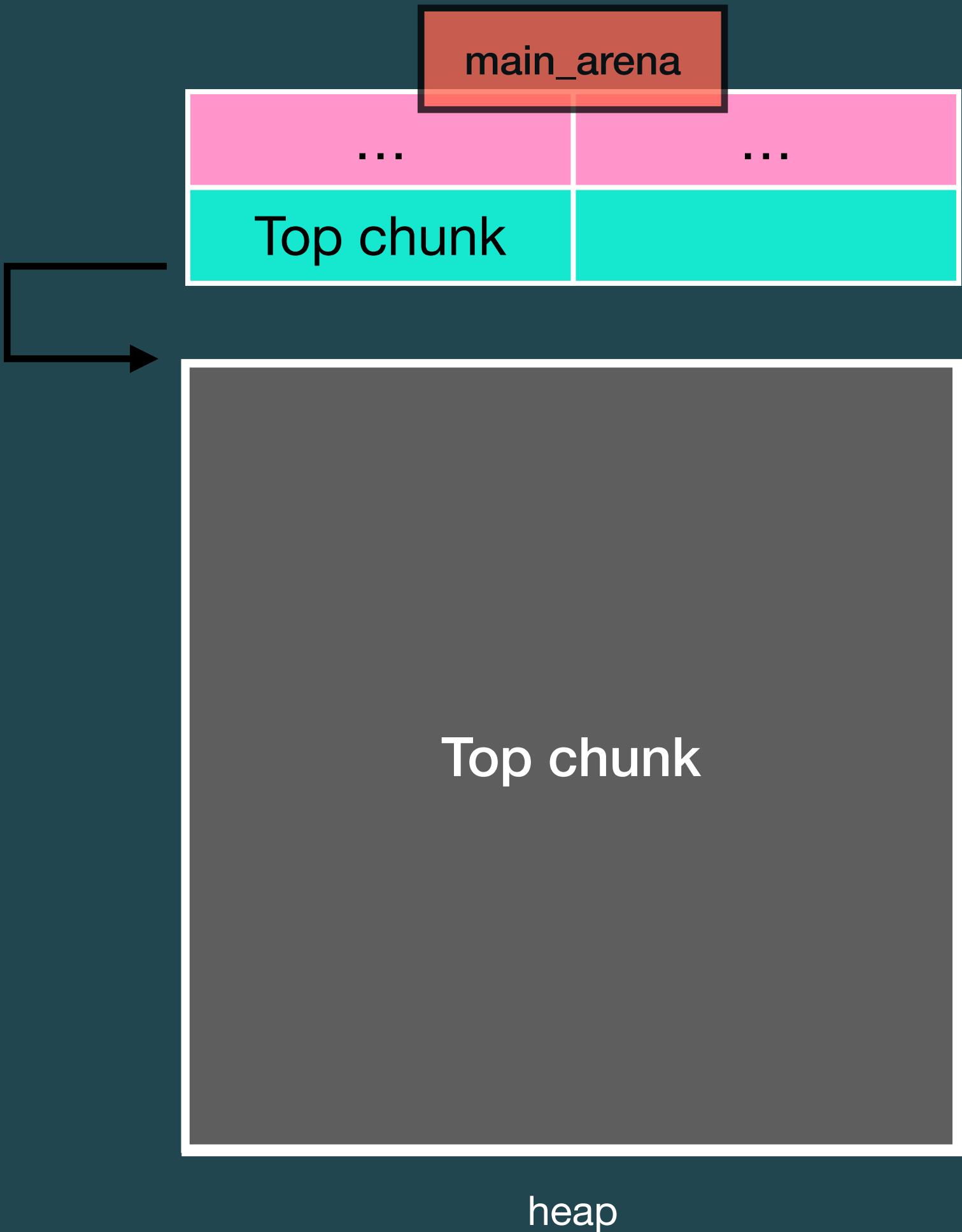
\$ Exploitation tech

Overlapping chunks

```
分配大小不會進 tcache 的 chunk A
$ ./a
A = malloc(0x410);
B = malloc(0x10);

*(A-1) = 0x421 + 0x20; // 0x20 == B 的 chunk size
free(A); // consolidate to top chunk

A = malloc(0x430);
total = (0x430 / 8);
A[total - 2] = 0xdeadbeef;
B[0] == 0xdeadbeef;
```



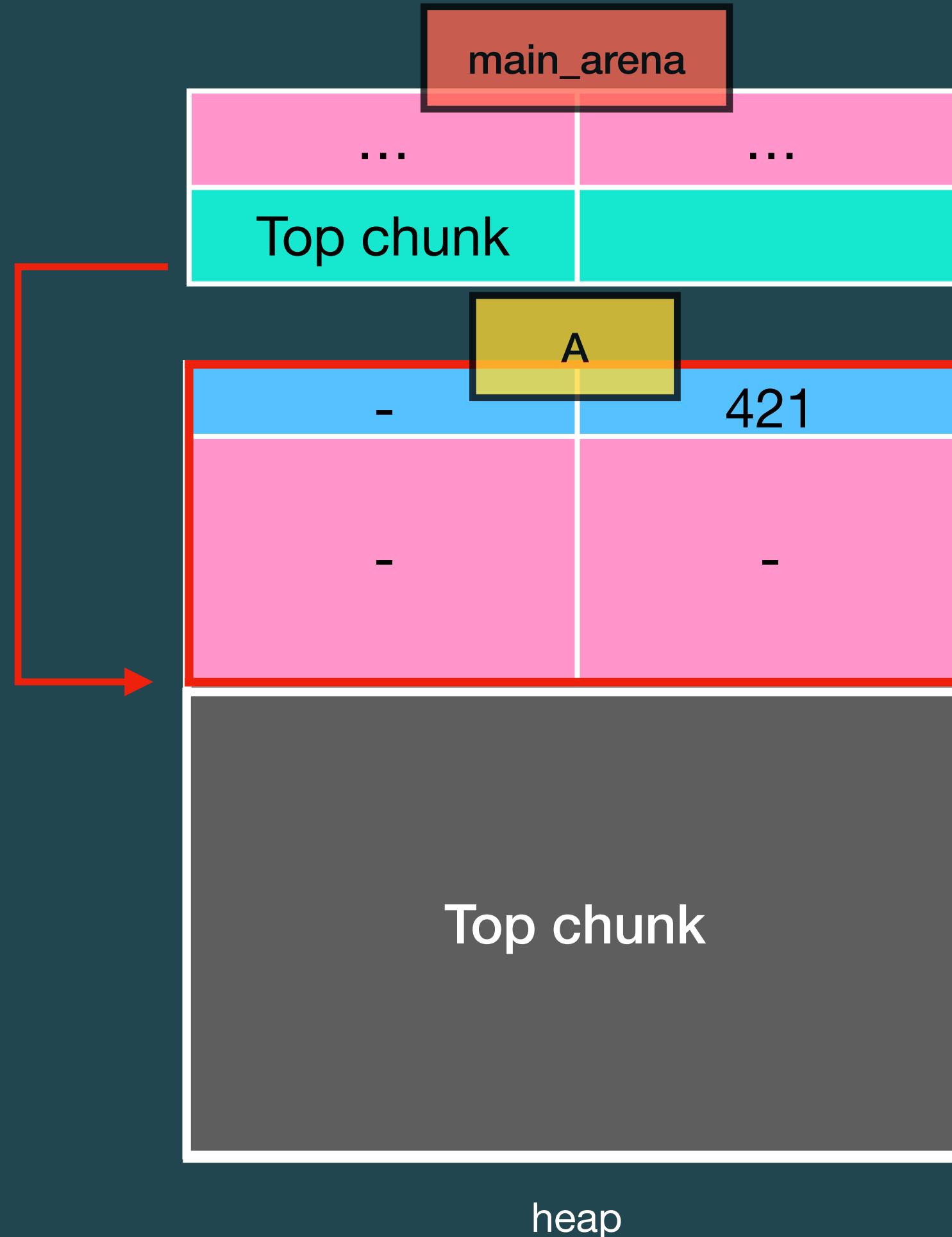
\$ Exploitation tech

Overlapping chunks

```
u1f383@u1f383:/ $ A = malloc(0x 分配要被 overlap 的 chunk B
B = malloc(0x10);

*(A-1) = 0x421 + 0x20; // 0x20 == B 的 chunk size
free(A); // consolidate to top chunk

A = malloc(0x430);
total = (0x430 / 8);
A[total - 2] = 0xdeadbeef;
B[0] == 0xdeadbeef;
```



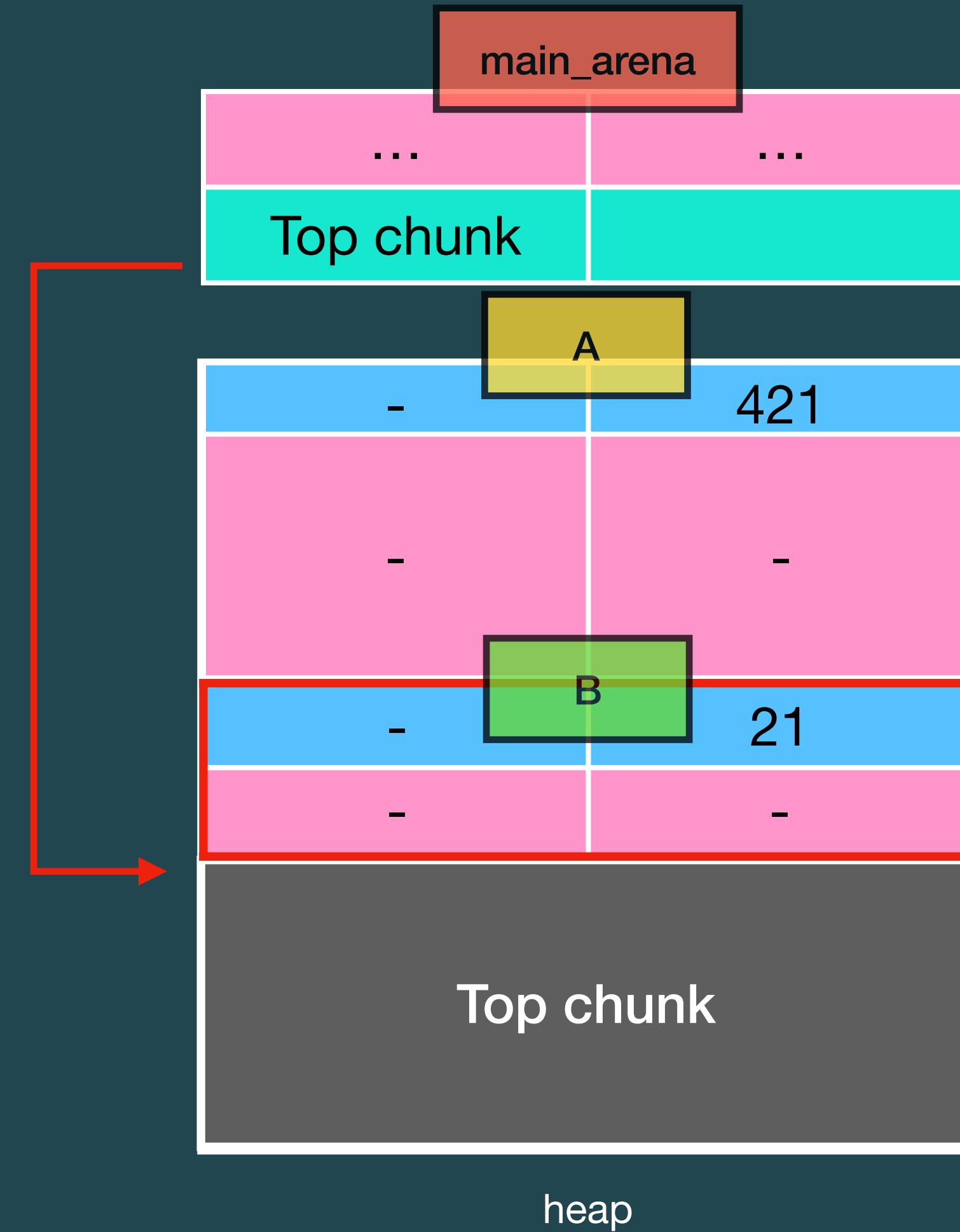
\$ Exploitation tech

Overlapping chunks

```
u1f383@u1f383:/
```

```
A = malloc(0x410);
B = 修改 chunk size , 剛好涵蓋整個 chunk B
*(A-1) = 0x421 + 0x20; // 0x20 == B 的 chunk size
free(A); // consolidate to top chunk

A = malloc(0x430);
total = (0x430 / 8);
A[total - 2] = 0xdeadbeef;
B[0] == 0xdeadbeef;
```



\$ Exploitation tech

Overlapping chunks

```
u1f383@u1f383:/
```

```
$ static void _int_free (... , mchunkptr p, ...)

...
size = chunksize (p);
...
else if (!chunk_is_mmapped(p)) {
...
/* consolidate backward */
if (!prev_inuse(p)) {
    // ... some operation
    unlink_chunk (av, p);
}

...
if (nextchunk != av->top) {
    nextinuse = inuse_bit_at_offset(nextchunk, nextsize);
    /* consolidate forward */
    if (!nextinuse) {
        unlink_chunk (av, nextchunk);
        size += nextsize;
    }
}
...
} else { /* consolidate with top chunk */
    size += nextsize;
    set_head(p, size | PREV_INUSE);
    av->top = p;
    check_chunk(av, p);
}
...
}
```

1. 當前一塊 chunk 沒有在使用，則 trigger consolidate，並且用 unlink_chunk 來離開 bin

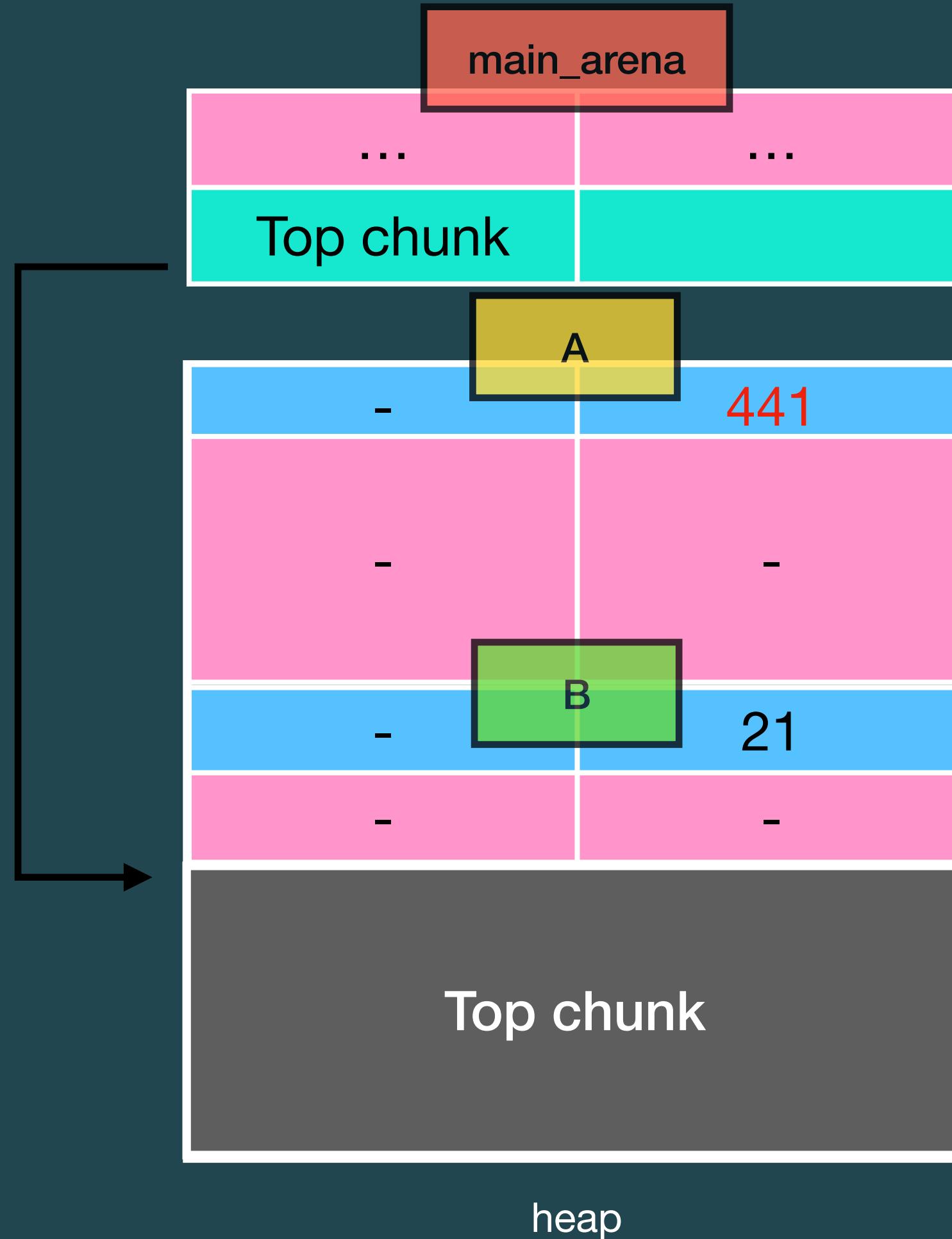
2. 下一塊 chunk 沒在用，並且不是 top chunk 的話，一樣 trigger consolidate

3. 如果下一塊是 top chunk，則與 top chunk 做 consolidate

\$ Exploitation tech

Overlapping chunks

```
u1f383@u1f383:/ $ ./overlapping_chunks  
釋放 chunk A 時，檢查 chunk A 的下個 chunk 為 top chunk，  
因此 trigger consolidation，merge A 與 top chunk  
free(A); // consolidate to top chunk  
  
A = malloc(0x430);  
total = (0x430 / 8);  
A[total - 2] = 0xdeadbeef;  
B[0] == 0xdeadbeef;
```



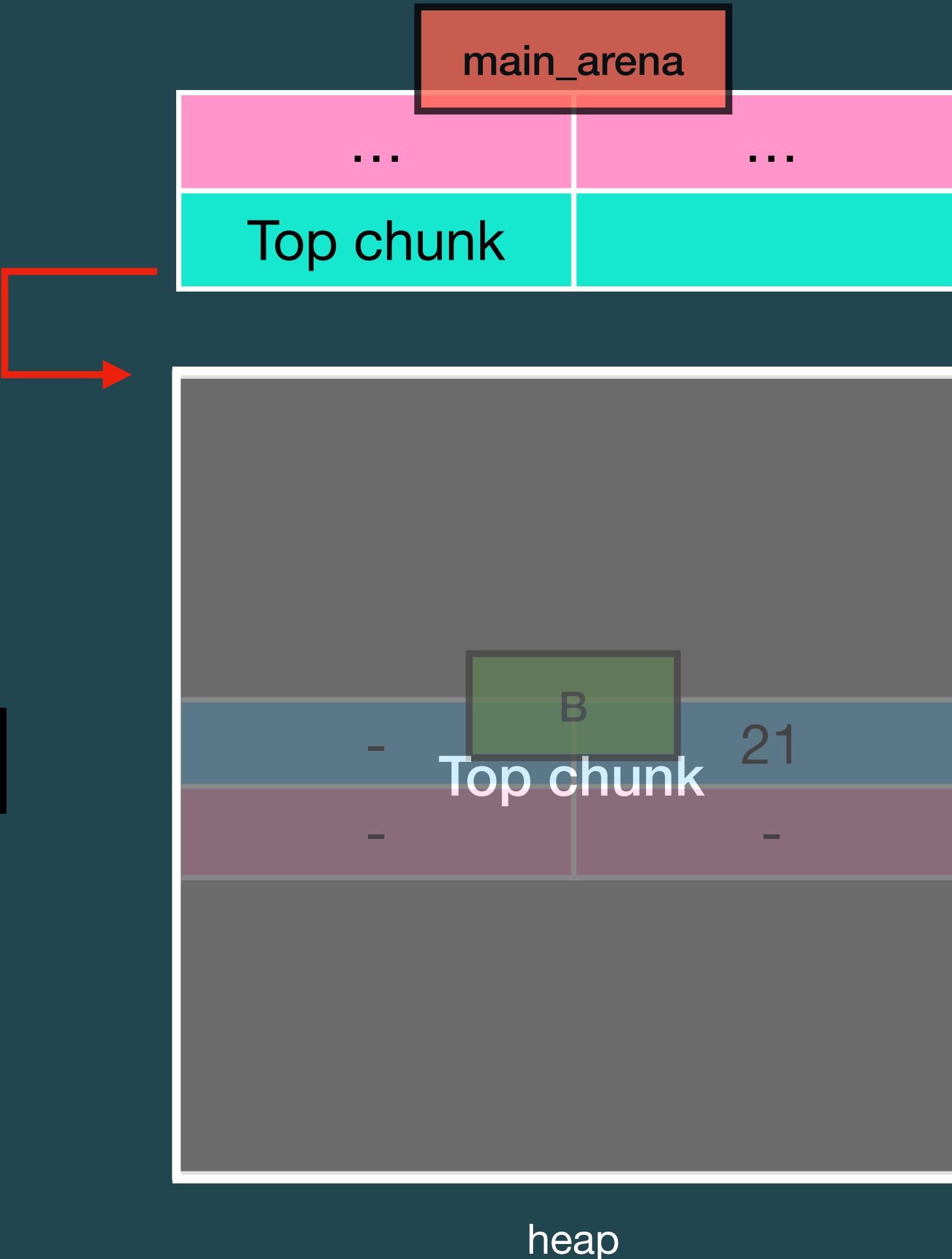
\$ Exploitation tech

Overlapping chunks

```
u1f383@u1f383:/ $ A = malloc(0x410);  
B = malloc(0x10);  
  
*(A - 1) = 0x121 + 0x20; // 0x20 == B 的 chunk size
```

此時 B 已經包含在 top chunk，分配 0x440 大小的 chunk 會包含 chunk B

```
A = malloc(0x430);  
total = (0x430 / 8);  
A[total - 2] = 0xdeadbeef;  
B[0] == 0xdeadbeef;
```



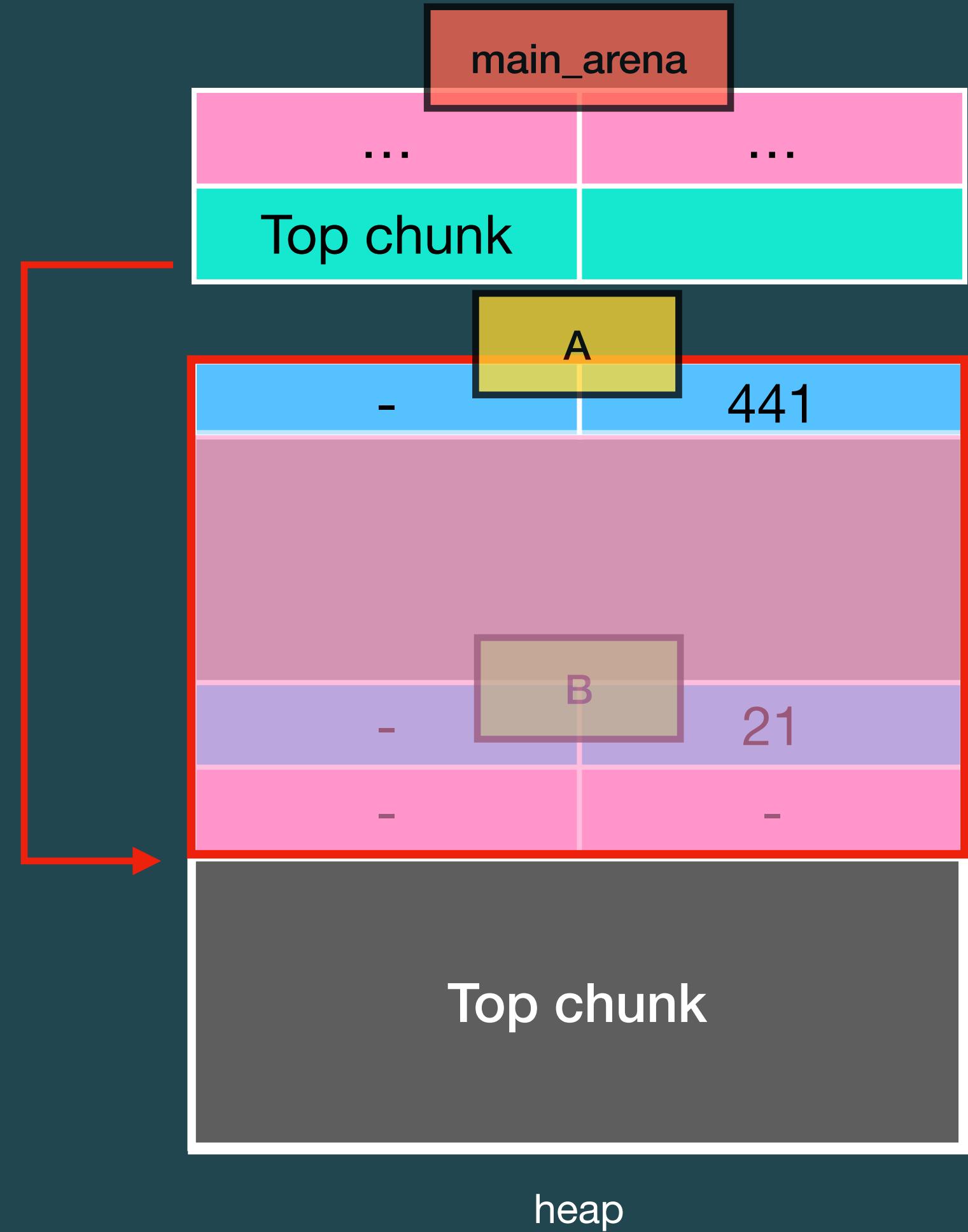
\$ Exploitation tech

Overlapping chunks

```
u1f383@u1f383:/
```

```
$ A = malloc(0x410);
B = malloc(0x10);

*(A-1) = 0x421 + 0x20; // 0x20 == B 的 chunk size
free(A); // consolidate to top chunk
當修改 chunk A 時，chunk B 也會受到影響
total = (0x430 / 8);
A[total - 2] = 0xdeadbeef;
B[0] == 0xdeadbeef;
```

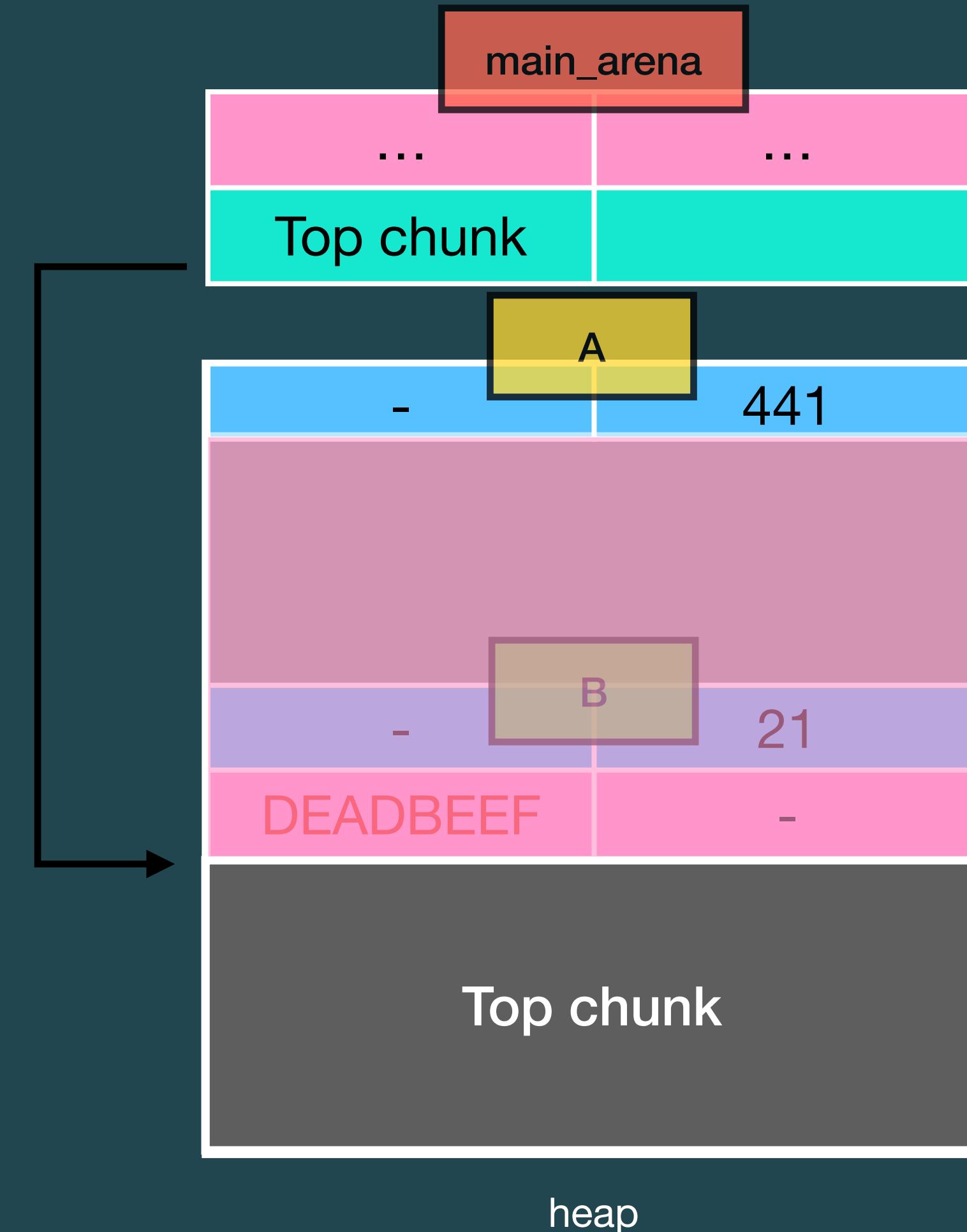


\$ Exploitation tech

Overlapping chunks

```
u1f383@u1f383:/ $  
$ A = malloc(0x410);  
B = malloc(0x10);  
  
*(A-1) = 0x421 + 0x20; // 0x20 == B 的 chunk size  
free(A); // consolidate to top chunk  
  
A = malloc(10);  
total = 10;  
A[total] = 0xdeadbeef;  
B[0] == 0xdeadbeef;
```

此範例中單純蓋寫 chunk B 的資料，而 size 或是 fd 也是很好的利用點





Safe linking

\$ Safe linking

Introduction

- ▶ 在 glibc 2.32 引入，會將 tcache 的 next 與 fastbin 的 fd 的 pointer 與所在位址的 $\gg 12$ 做 XOR
- ▶ 會檢查第一塊 tcache chunk 的位址是否對齊 0x10
- ▶ 下圖 P 為 chunk 存放的 pointer value ; L 為 chunk 位址

$$\begin{aligned}P &:= 0x0000BA\textcolor{red}{9876543}210 \\L &:= 0x0000BA\textcolor{red}{9876543}180 \\ \\P \oplus L \gg 12 &= \begin{array}{c} 0x0000BA\textcolor{red}{9876543}210 \\ \oplus \\ 0x00000000BA\textcolor{red}{9876543} \end{array} \\ \\P' &:= P \oplus (L \gg 12) = 0x0000BA\textcolor{red}{93DFD35753}\end{aligned}$$

\$ Safe linking

Introduction

```
u1f383@u1f383:/
```

```
$
```

```
#define PROTECT_PTR(pos, ptr) \
    ((__typeof__(ptr))(((size_t)pos) >> 12) ^ ((size_t)ptr)) \
#define REVEAL_PTR(ptr) PROTECT_PTR(&ptr, ptr)
```

```
#define aligned_OK(m) (((unsigned long)(m)&MALLOC_ALIGN_MASK) == 0)
```

```
static __always_inline void
tcache_put(mchunkptr chunk, size_t tc_idx)
{
    ...
    e->next = PROTECT_PTR(&e->next, tcache->entries[tc_idx]);
    ...
}

static __always_inline void *
tcache_get(size_t tc_idx)
{
    ... tcache_entry *e = tcache->entries[tc_idx];
    if (__glibc_unlikely(!aligned_OK(e)))
        malloc_printerr("malloc(): unaligned tcache chunk detected");
    tcache->entries[tc_idx] = REVEAL_PTR(e->next);
    ...
}
```

Mangle / demangle pointer

Check alignment

Use in tcache

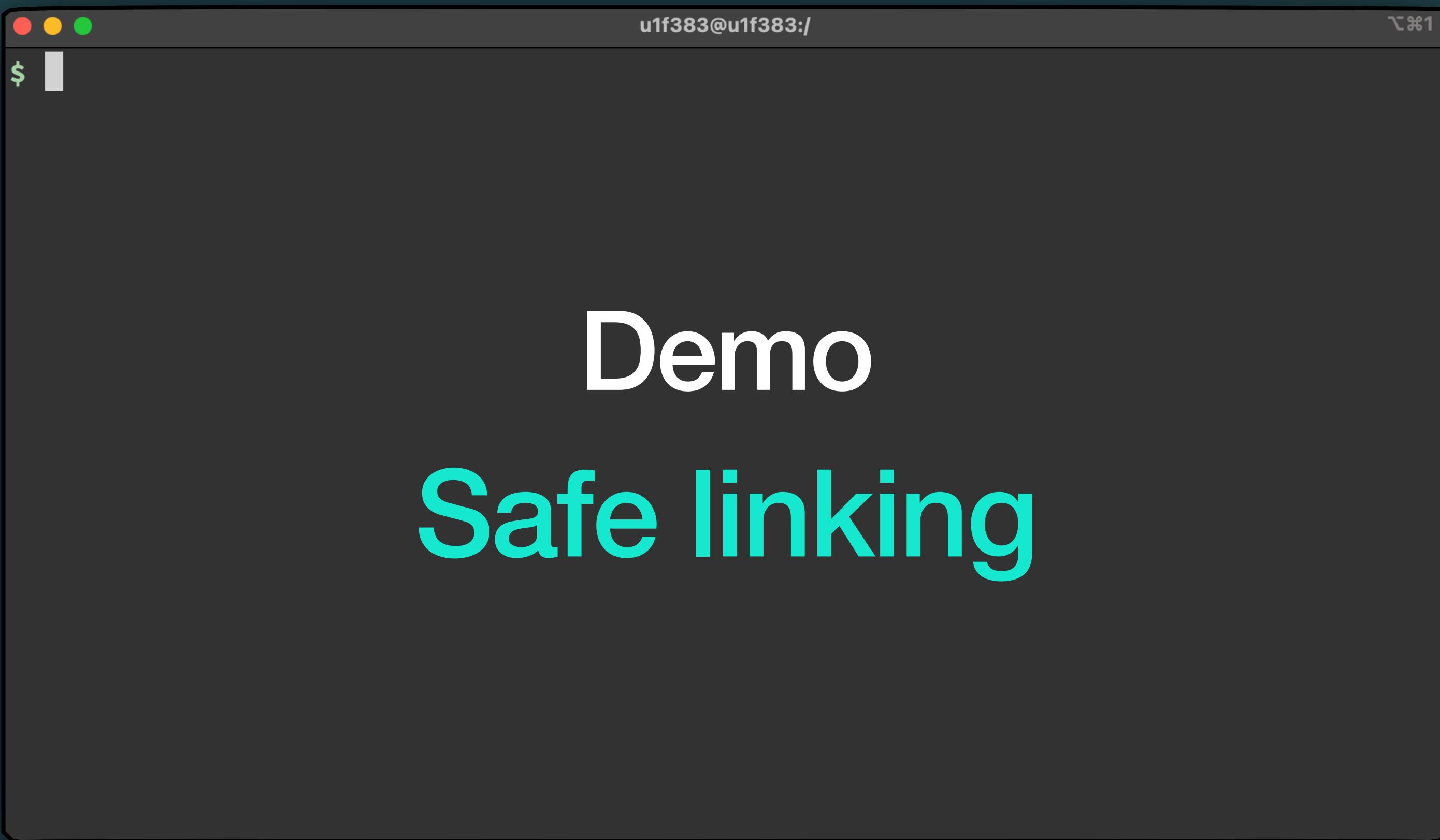
\$ Safe linking

Bypass

- ▶ 如果 tcache chunk 的 next 欄位為 NULL，則 xor 就是直接存放 heap >> 12
 - ⦿ 從 tcache 取出 chunk 時並不會清空 next 欄位，因此 chunk 會殘留 heap >> 12 的值
- ▶ 若可以控制 `tcache_perthread_struct`，其內部會直接存放沒有做 XOR 的 heap pointer，也可以拿來利用

\$ Safe linking

Bypass



Demo
Safe linking



How2heap tech

\$ How2heap tech

	2.31	2.32	2.34
fastbin_dup	✓	✓	✓
fastbin_reverse_into_tcache	✓	✓	✓
house_of_botcake	✓	✓	✓
house_of_einherjar	✓	✓	✓
house_of_lore	✓	✓	✓
house_of_mind_fastbin	✓	✓	✓
large_bin_attack	✓	✓	✓
mmap_overlapping_chunks	✓	✓	✓
overlapping_chunks	✓	✓	✓
poison_null_byte	✓	✓	✓
tcache_house_of_spirit	✓	✓	✓
tcache_poisoning	✓	✓	✓
tcache_stashing_unlink_attack	✓	✓	✓
unsafe_unlink	✓	✓	✓
decrypt_safe_linking		✓	✓

\$ How2heap tech

Tcache stashing (fastbin_reverse_into_tcache)

```
u1f383@u1f383:/
```

```
$ size_t victim_idx = fastbin_index(chunksize(victim));
  if (__builtin_expect(victim_idx != idx, 0))
    malloc_perr("malloc(): memory corruption (fast)");
  check_reallocoed_chunk(av, victim, nb);
  /* While we're here, if we see other chunks of the same size,
   * stash them in the tcache. */
  size_t tc_idx = csize2tidx(nb);
  if (tcache && tc_idx < mp_.tcache_bins)
  {
    mchunkptr tc_victim;
    /* While bin not empty and tcache not full, copy chunks. */
    while (tcache->counts[tc_idx] < mp_.tcache_count && (tc_victim = *fb) != NULL)
    {
      if (SINGLE_THREAD_P)
        *fb = tc_victim->fd;
      else
      {
        REMOVE_FB(fb, pp, tc_victim);
        if (__glibc_unlikely(tc_victim == NULL))
          break;
      }
      tcache_put(tc_victim, tc_idx);
    }
    void *p = chunk2mem(victim);
    alloc_perturb(p, bytes);
    return p;
```

1. 在從 fastbin 取得 chunk 並回傳前，會檢查 tcache 是否滿了

2. 如果還沒滿，就取出來剩下的 fastbin chunk 放到 tcache，直到 tcache 滿了或 fastbin 沒了

\$ How2heap tech

Smallbin stashing (tcache_stashing_unlink_attack) & House of lore

```
u1f383@u1f383:/
```

```
$
```

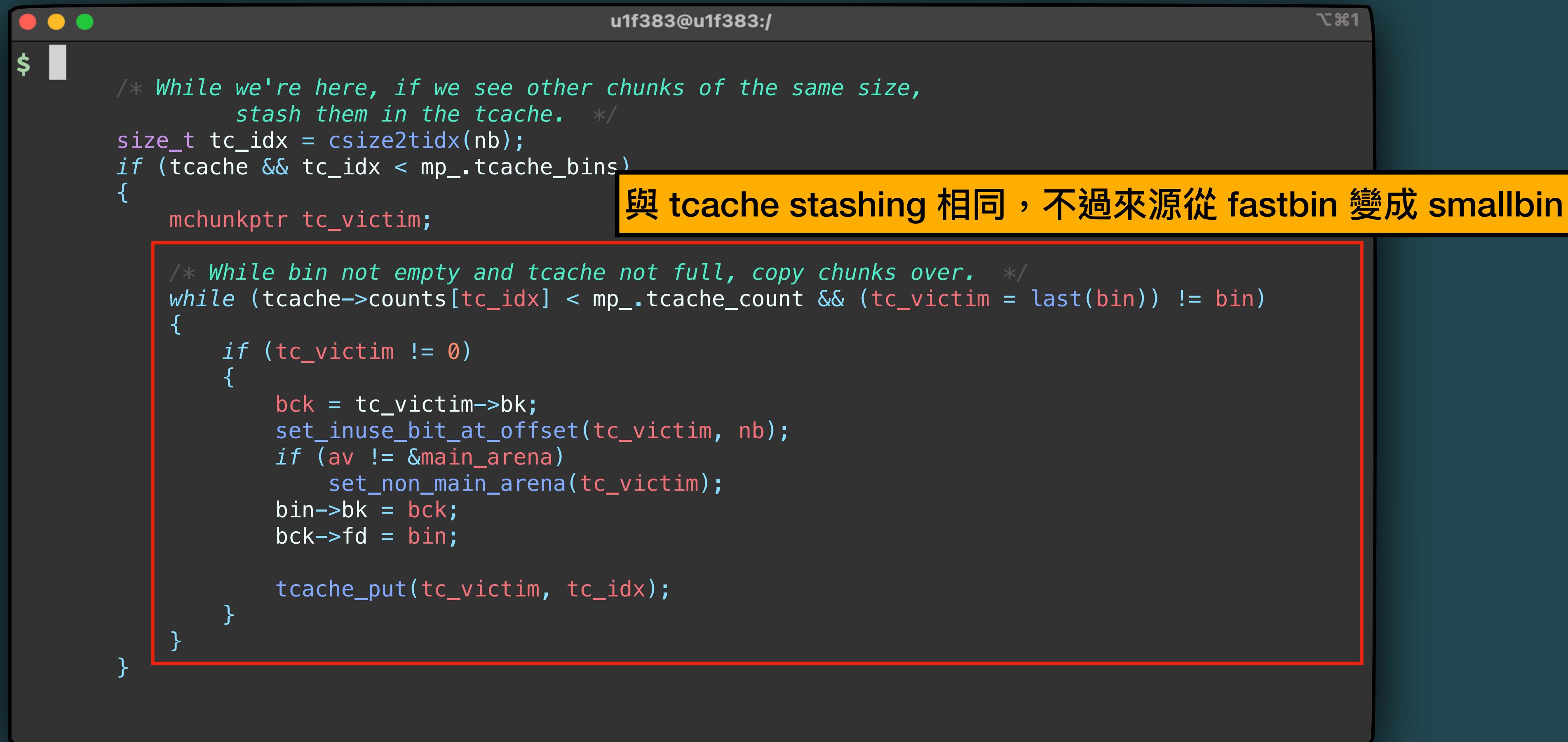
```
if (in_smallbin_range(nb))
{
    idx = smallbin_index(nb);
    bin = bin_at(av, idx);

    if ((victim = last(bin)) != bin)
    {
        bck = victim->bk;
        if (__glibc_unlikely(bck->fd != victim))
            malloc_printerr("malloc(): smallbin double linked list corrupted");
        set_inuse_bit_at_offset(victim, nb);
        bin->bk = bck;
        bck->fd = bin;
    }
    ...
}
```

從 smallbin 取出 chunk 時，會檢查 `victim->bk->fd == victim`，
也就是自己的下一塊 chunk 的 fd 是否指向自己

\$ How2heap tech

Smallbin stashing (tcache_stashing_unlink_attack) & House of lore



The screenshot shows a terminal window with the title "u1f383@u1f383:/". The code is written in C and handles the stashing of chunks in a tcache. A red box highlights the main loop of the function, and a yellow box contains a note about the source of the stashed chunks.

```
/* While we're here, if we see other chunks of the same size,
   stash them in the tcache. */
size_t tc_idx = csize2tidx(nb);
if (tcache && tc_idx < mp_.tcache_bins)
{
    mchunkptr tc_victim;

    /* While bin not empty and tcache not full, copy chunks over. */
    while (tcache->counts[tc_idx] < mp_.tcache_count && (tc_victim = last(bin)) != bin)
    {
        if (tc_victim != 0)
        {
            bck = tc_victim->bk;
            set_inuse_bit_at_offset(tc_victim, nb);
            if (av != &main_arena)
                set_non_main_arena(tc_victim);
            bin->bk = bck;
            bck->fd = bin;

            tcache_put(tc_victim, tc_idx);
        }
    }
}
```

與 tcache stashing 相同，不過來源從 fastbin 變成 smallbin

\$ How2heap tech house_of_mind_fastbin

```
u1f383@u1f383:/
```

```
$
```

若 chunk 不屬於 main_arena，則對應的 arena 從記憶體位址來判斷

```
#define heap_for_ptr(ptr) \  
    ((heap_info *) ((unsigned long) (ptr) & ~(HEAP_MAX_SIZE - 1)))  
#define arena_for_chunk(ptr) \  
    (chunk_main_arena (ptr) ? &main_arena : heap_for_ptr (ptr)->ar_ptr)
```

```
void  
__libc_free (void *mem)  
{  
    ...  
    ar_ptr = arena_for_chunk (p);  
    _int_free (ar_ptr, p, 0);  
}
```

p 為 mem 的 chunk 開頭，而取得 arena pointer ar_ptr 後，會將其做為參數傳入 _int_free()

\$ How2heap tech large_bin_attack

```
u1f383@u1f383:/
```

```
$ if (in_smallbin_range(size)) { ... }
else {
    victim_index = largebin_index(size);
    bck = bin_at(av, victim_index);
    fwd = bck->fd;
}

/* maintain large bins in sorted order
if (fwd != bck)
{
    /* Or with inuse bit to
    size |= PREV_INUSE;
    /* if smaller than smallbin
    assert(chunk_main_arena);
    if ((unsigned long)(size) < (unsigned long)chunksize_nomask(bck->bk))
    {
        fwd = bck;
        bck = bck->bk;

        victim->fd_nextsize = fwd->fd;
        victim->bk_nextsize = fwd->fd->bk_nextsize;
        fwd->fd->bk_nextsize = victim->bk_nextsize->fd_nextsize = victim;
    }
}
```

1. bck 為 main_arena (av) 對應的 largebin 位址，而 fwd 則是第一塊 largebin chunk

2. 如果 largebin 內有 chunk 的話

3. 請求的 chunk size 小於 largebin 當中最小的 chunk，經過 pointer 的更新後回傳給使用者

\$ How2heap tech unsafe_unlink

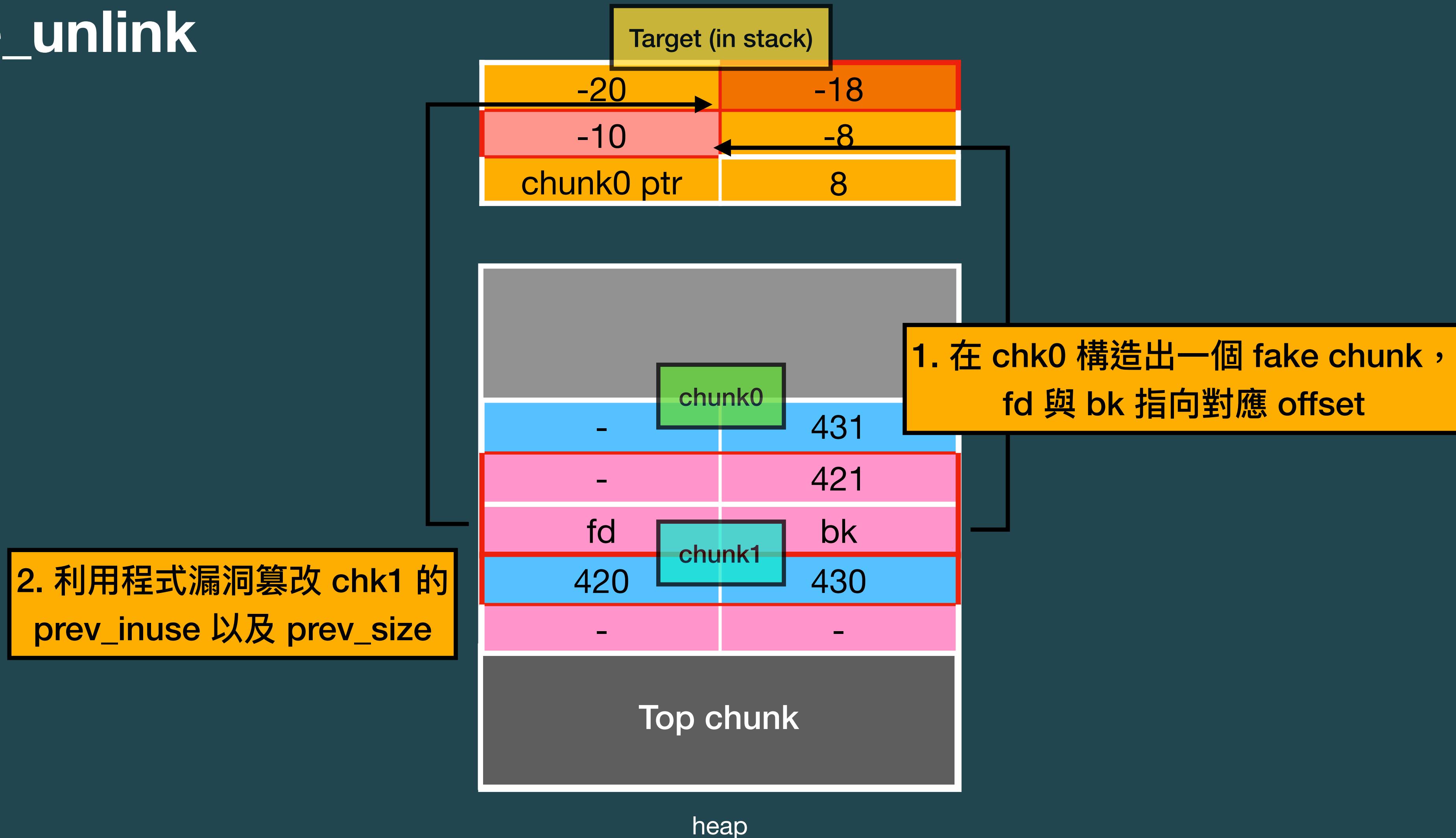
```
u1f383@u1f383:/  
$ static void unlink_chunk(mstate av, mchunkptr p)  
{  
    if (chunksize(p) != prev_size(next_chunk(p)))  
        malloc_printerr("corrupted size vs. prev_size");  
  
    mchunkptr fd = p->fd;  
    mchunkptr bk = p->bk;  
  
    if (__builtin_expect(fd->bk != p || bk->fd != p, 0))  
        malloc_printerr("corrupted double-linked list");  
  
    fd->bk = bk;  
    bk->fd = fd;  
    if (!in_smallbin_range(chunksize_nomask(p)) && p->fd_nextsize != NULL) {  
        if (p->fd_nextsize->bk_nextsize != p || p->bk_nextsize->fd_nextsize != p)  
            malloc_printerr("corrupted double-linked list (not small)");  
  
        if (fd->fd_nextsize == NULL) {  
            if (p->fd_nextsize == p)  
                fd->fd_nextsize = fd->bk_nextsize = fd;  
            else {  
                fd->fd_nextsize = p->fd_nextsize;  
                fd->bk_nextsize = p->bk_nextsize;  
                p->fd_nextsize->bk_nextsize = fd;  
                p->bk_nextsize->fd_nextsize = fd;  
            }  
        }  
        else {  
            p->fd_nextsize->bk_nextsize = p->bk_nextsize;  
            p->bk_nextsize->fd_nextsize = p->fd_nextsize;  
        }  
    }  
}
```

1. 檢查下個 chunk 的 prev_size 是否等於自己 && 前一個的下一個要是自己 && 後一個的上一個要是自己

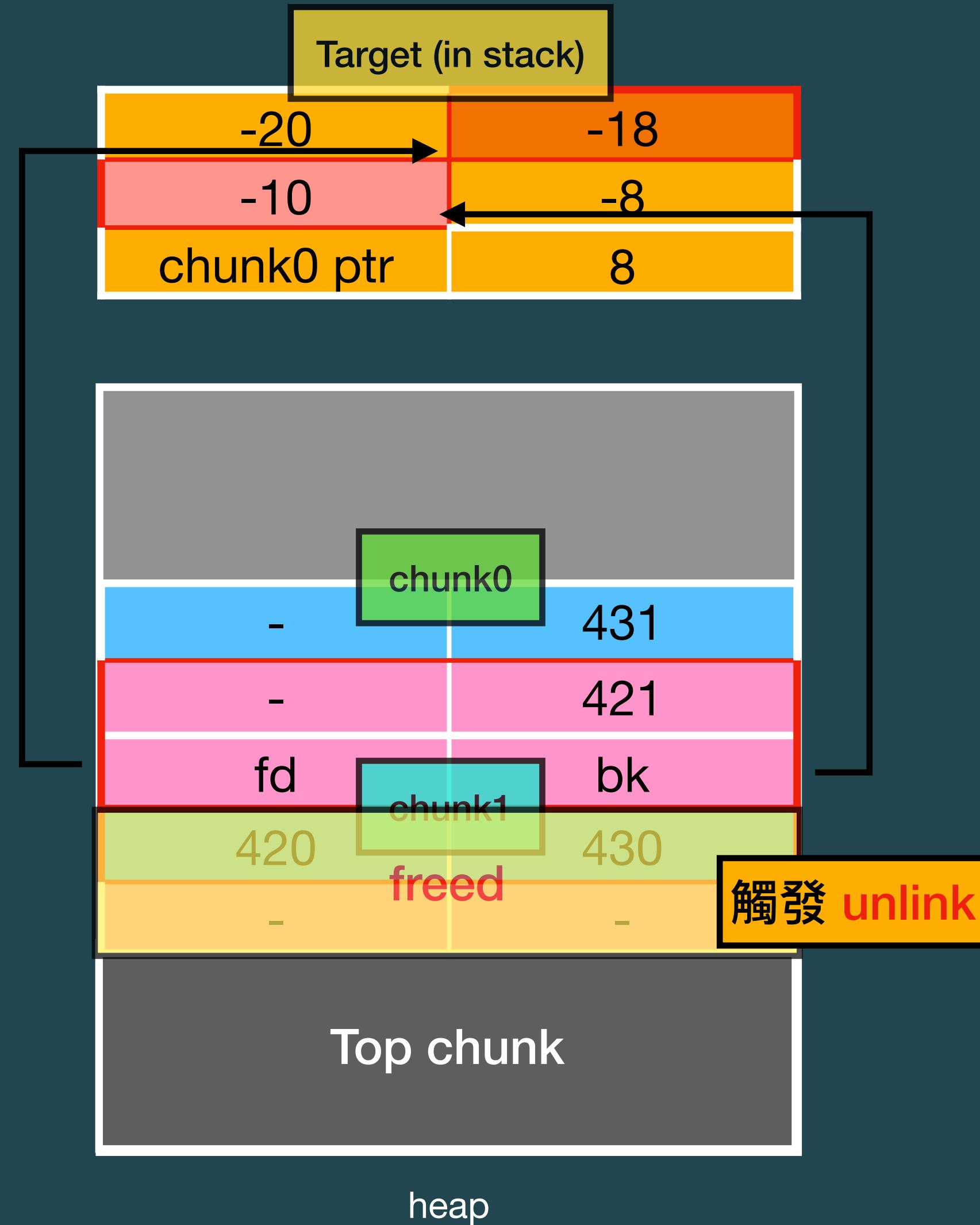
2. smallbin 或 unsorted bin 就單純更新 fd bk 即可

3. largebin 就需要額外更新 fd_nextsize 與 bk_nextsize

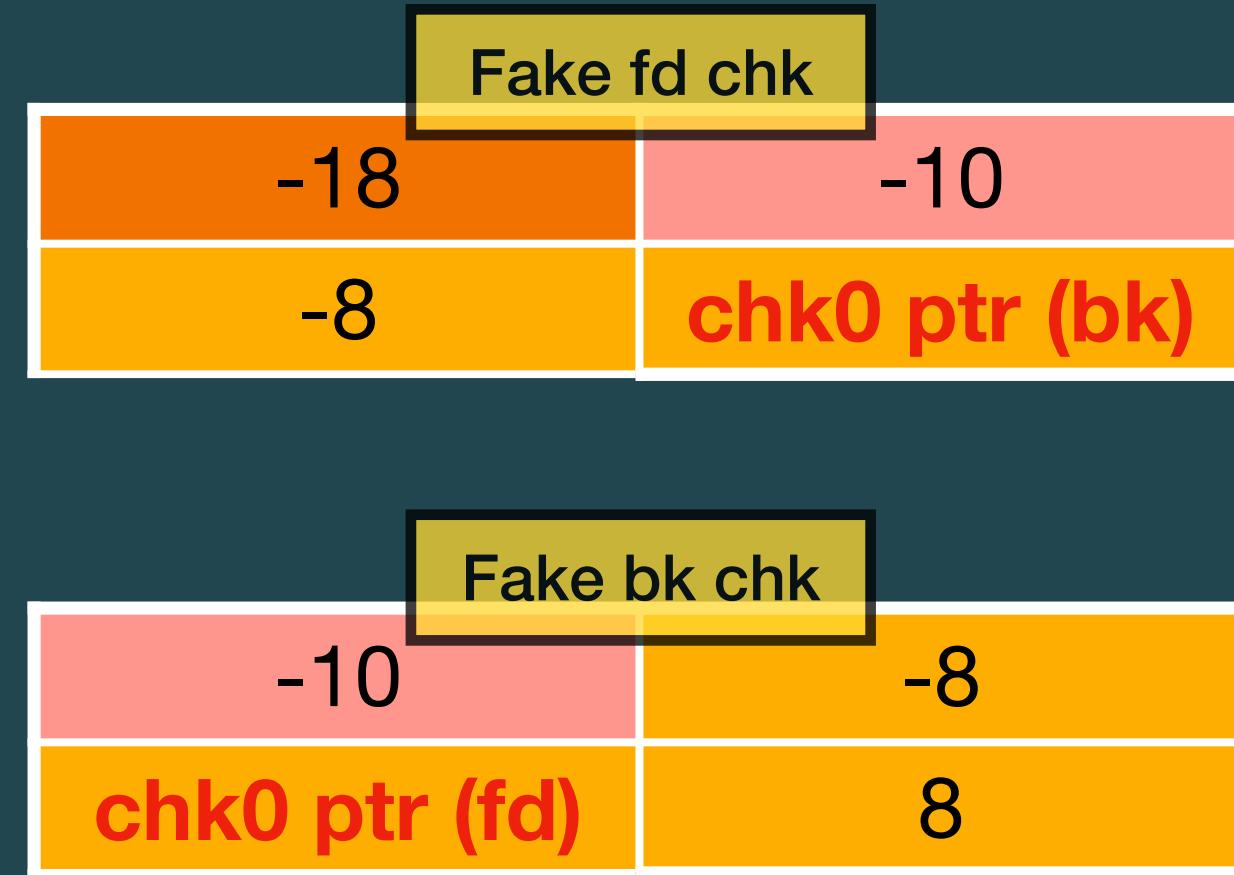
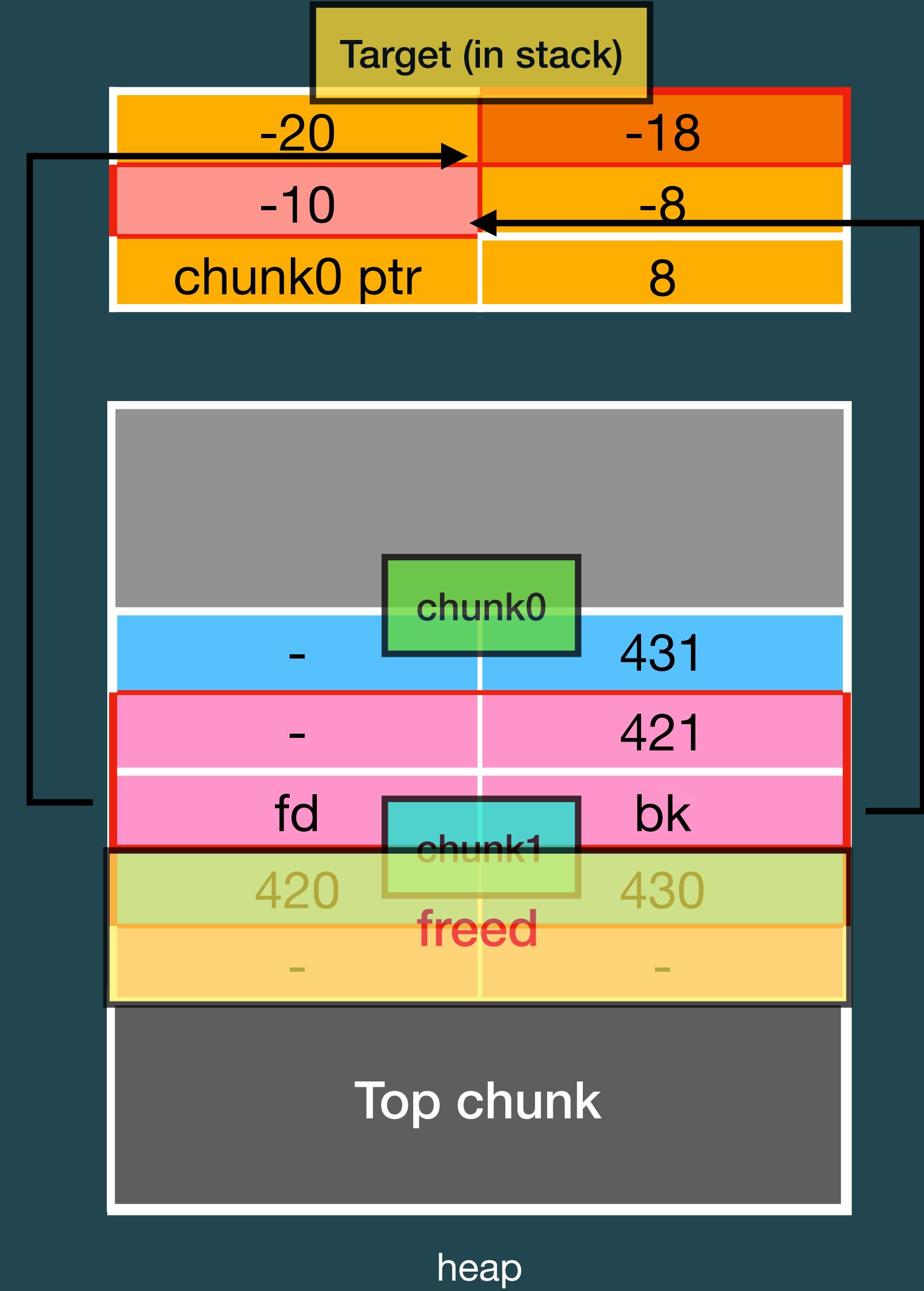
\$ How2heap tech unsafe_unlink



\$ How2heap tech unsafe_unlink



\$ How2heap tech unsafe_unlink



1. 通過 **fd->bk** 與 **bk->fd** 的檢查

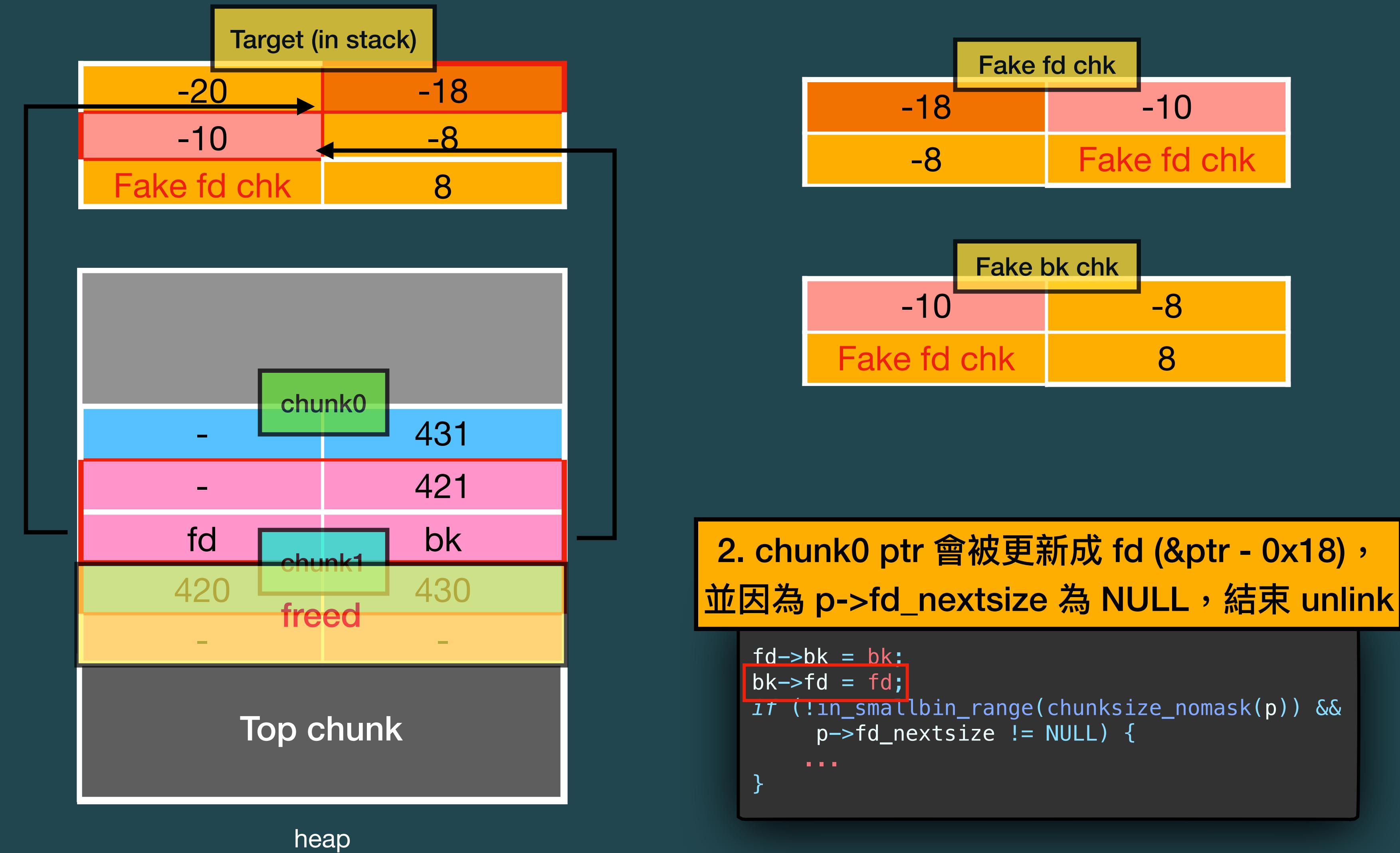
```
if (chunksize(p) != prev_size(next_chunk(p)))
    malloc_printerr("corrupted size vs. prev_size");

mchunkptr fd = p->fd;
mchunkptr bk = p->bk;

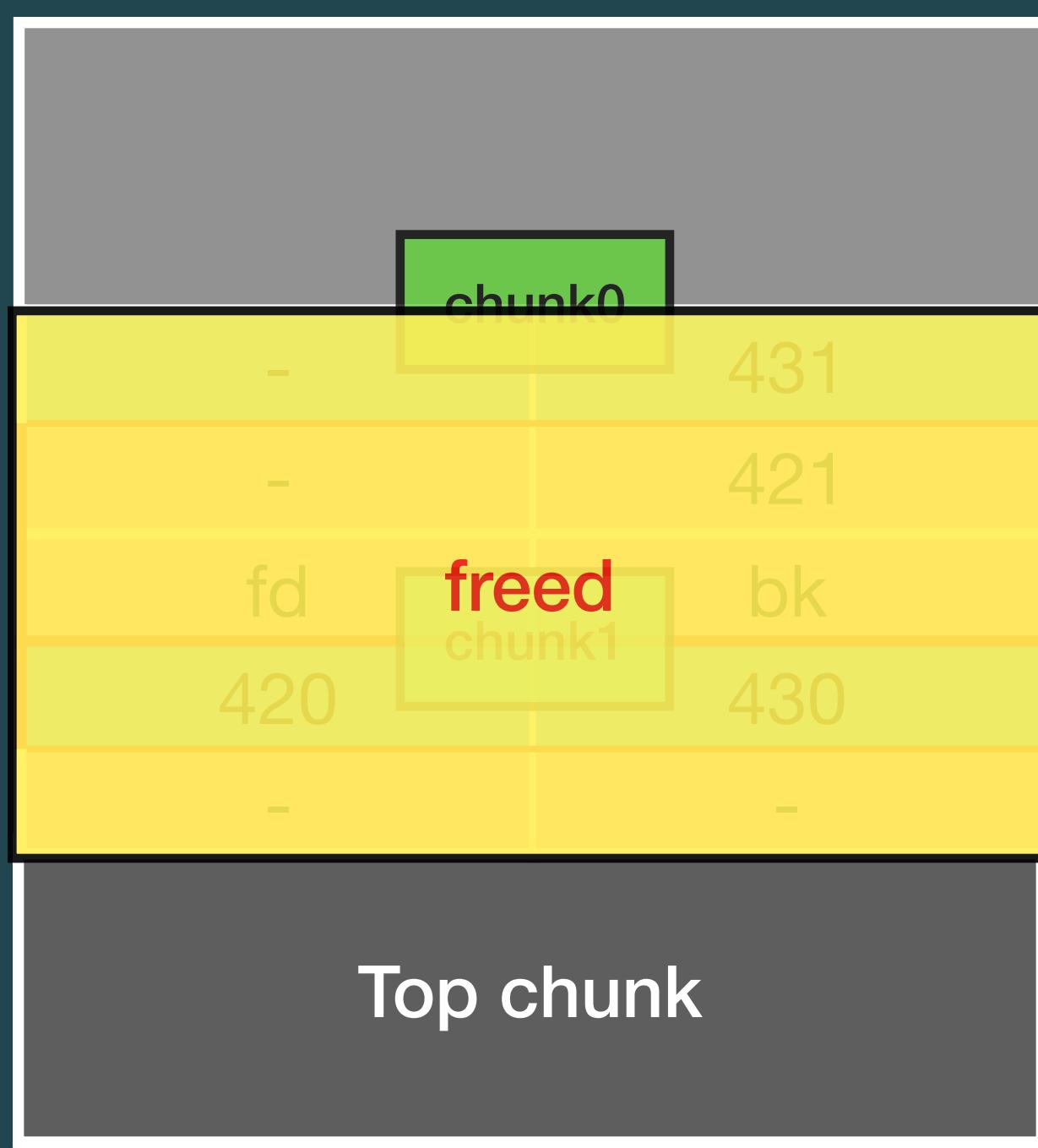
if (__builtin_expect(fd->bk != p || bk->fd != p, 0))
    malloc_printerr("corrupted double-linked list");
```

\$ How2heap tech

unsafe_unlink



\$ How2heap tech unsafe_unlink



heap

\$ How2heap tech







Appendix

\$ Appendix

Tools

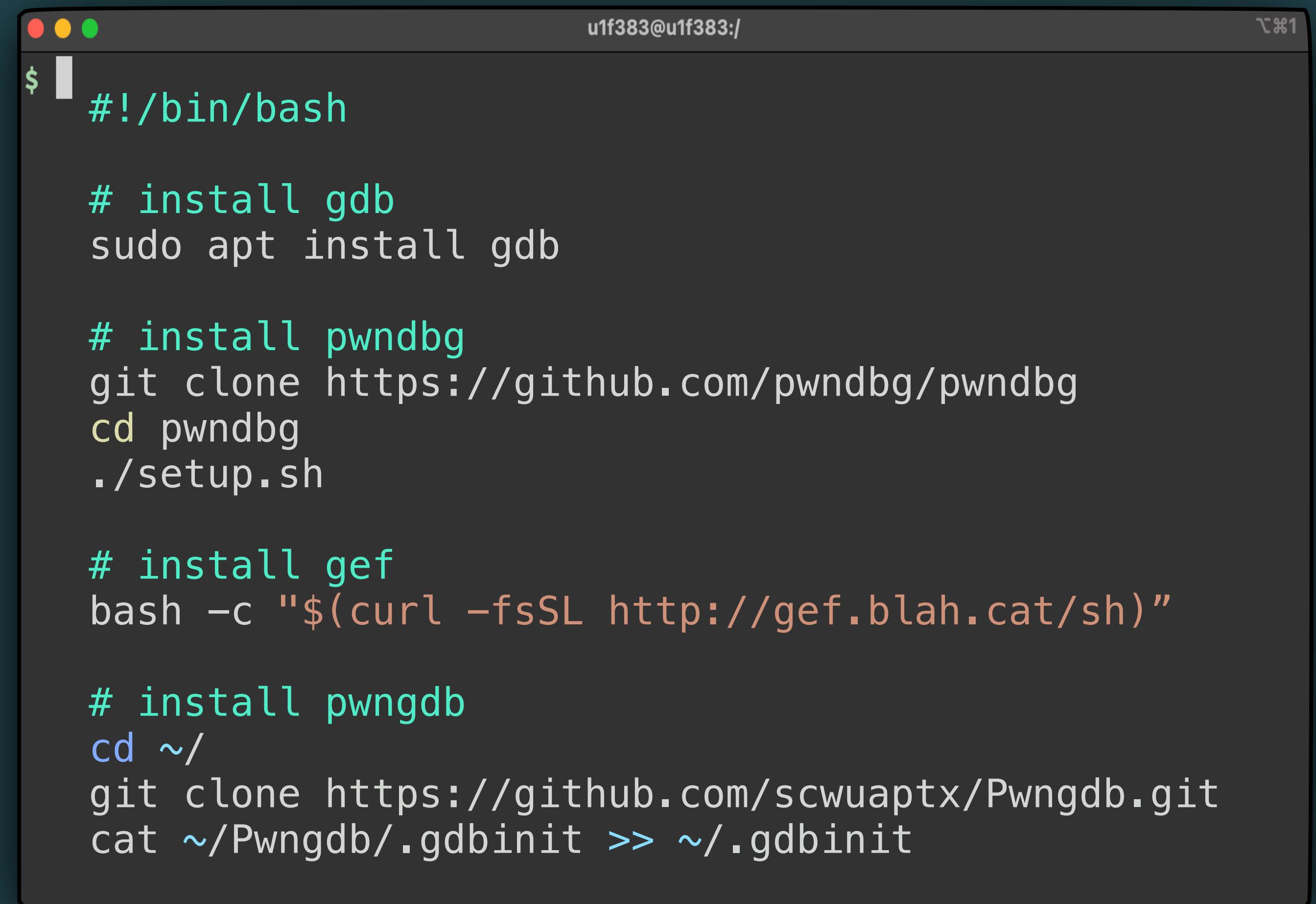
- ▶ *gdb* - the GNU debugger

- ▶ gdb plugins

- ⦿ gef

- ⦿ pwndbg

- ⦿ pwngdb



```
#!/bin/bash

# install gdb
sudo apt install gdb

# install pwndbg
git clone https://github.com/pwndbg/pwndbg
cd pwndbg
./setup.sh

# install gef
bash -c "$(curl -fsSL http://gef.blah.cat/sh)"

# install pwngdb
cd ~/
git clone https://github.com/scwuaptx/Pwngdb.git
cat ~/Pwngdb/.gdbinit >> ~/.gdbinit
```

```
u1f383@u1f383:/
```

```
$ #### Normal #####
b *<addr>      # break at addr
start            # break at main automatically
r                # run program
si               # exec one insn and step into
ni               # exec one insn and go next
s                # exec one line code and step into
n                # exec one line code and go next
x/30gx <addr>  # print content as 8 bytes, 30 times, hex format
x/10i <addr>   # print content as instruction, 10 lines, hex format
p <variable>    # print var's value

#### Advanced #####
xinfo <addr>      # extended information for virtual address
vmmmap           # virtual memory map
telescope         # recursively deference pointer (default $sp)
bt               # backtrace
p *(struct Test *) <addr> # print content in addr as struct Test
ctx              # refresh current status
fin              # finish current function
```

0 > 1 > docker

[REGISTERS]

```
RAX 0x55b2bb4aa149 (main) ← endbr64
RBX 0x55b2bb4aa170 (_libc_csu_init) ← endbr64
RCX 0x55b2bb4aa170 (_libc_csu_init) ← endbr64
RDX 0x7ffc0182ee28 → 0x7ffc0182f84c ← 'PYTHONIOENCODING=UTF-8'
RDI 0x1
RSI 0x7ffc0182ee18 → 0x7ffc0182f842 ← '/tmp/test'
R8 0x0
R9 0x7f4a65933d50 ← endbr64
R10 0x0
R11 0x0
R12 0x55b2bb4aa060 (_start) ← endbr64
R13 0x7ffc0182ee10 ← 0x1
R14 0x0
R15 0x0
RBP 0x0
RSP 0x7ffc0182ed28 → 0x7f4a657480b3 (__libc_start_main+243) ← mov edi, eax
RIP 0x55b2bb4aa149 (main) ← endbr64
```

[DISASM]

```
► 0x55b2bb4aa149 <main> endbr64
 0x55b2bb4aa14d <main+4> push rbp
 0x55b2bb4aa14e <main+5> mov rbp, rsp
 0x55b2bb4aa151 <main+8> lea rdi, [rip + 0xeac]
 0x55b2bb4aa158 <main+15> call puts@plt <puts@plt>
 0x55b2bb4aa15d <main+20> mov eax, 0
 0x55b2bb4aa162 <main+25> pop rbp
 0x55b2bb4aa163 <main+26> ret
 0x55b2bb4aa164 nop word ptr cs:[rax + rax]
 0x55b2bb4aa16e nop
 0x55b2bb4aa170 <__libc_csu_init> endbr64
```

[SOURCE (CODE)]

```
In file: /tmp/test.c
1 #include <stdio.h>
2
3 int main()
► 4 {
5     puts("OWO");
6     return 0;
7 }
```

[STACK]

00:0000	rsp 0x7ffc0182ed28 → 0x7f4a657480b3 (__libc_start_main+243) ← mov edi, eax
01:0008	0x7ffc0182ed30 → 0x7f4a6594f620 (_rtld_global_ro) ← 0x50a2f00000000
02:0010	0x7ffc0182ed38 → 0x7ffc0182ee18 → 0x7ffc0182f842 ← '/tmp/test'
03:0018	0x7ffc0182ed40 ← 0x100000000
04:0020	0x7ffc0182ed48 → 0x55b2bb4aa149 (main) ← endbr64
05:0028	0x7ffc0182ed50 → 0x55b2bb4aa170 (_libc_csu_init) ← endbr64
06:0030	0x7ffc0182ed58 ← 0xbdfe5f8fb270b7df
07:0038	0x7ffc0182ed60 → 0x55b2bb4aa060 (_start) ← endbr64

[BACKTRACE]

```
► f 0 0x55b2bb4aa149 main
  f 1 0x7f4a657480b3 __libc_start_main+243
```

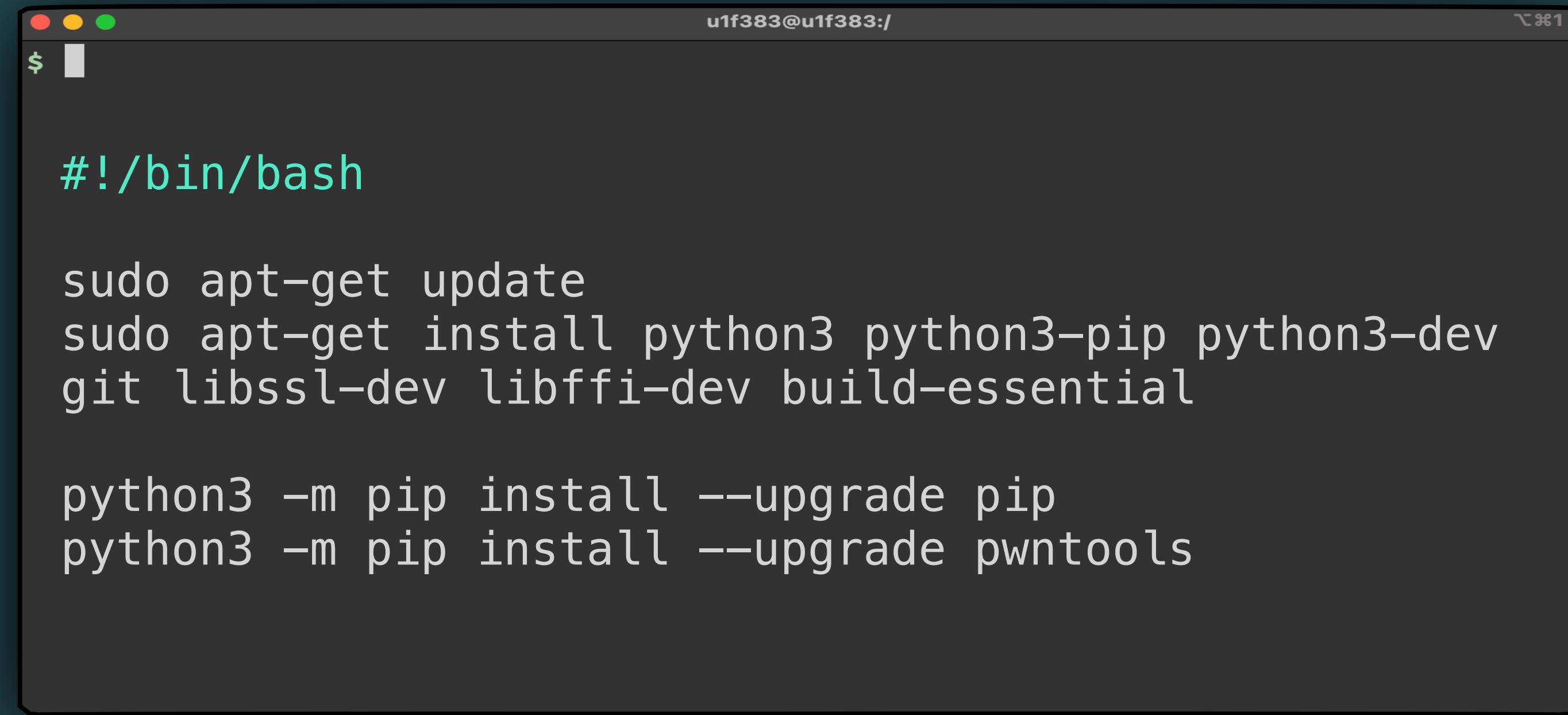
pwndbg>

gdb + pwndbg

\$ Appendix

Tools

- ▶ pwntools - an python exploit development library



A screenshot of a terminal window titled "u1f383@u1f383:/". The window contains the following command-line session:

```
#!/bin/bash

sudo apt-get update
sudo apt-get install python3 python3-pip python3-dev
git libssl-dev libffi-dev build-essential

python3 -m pip install --upgrade pip
python3 -m pip install --upgrade pwntools
```

```
u1f383@u1f383:/ ~%1  
$ !#/usr/bin/python3  
  
from pwn import *  
import sys  
  
# specified arch  
context.arch = 'amd64'  
# specified terminal  
context.terminal = ['tmux', 'splitw', '-h']  
  
if len(sys.argv) != 2:  
    exit(1)  
  
HOST = 'localhost'  
PORT = 9999  
BINARY = './test'  
  
# ./exp.py remote  
# ./exp.py local  
if sys.argv[1] == 'remote':  
    # connect to remote server  
    r = remote(HOST, PORT)  
elif sys.argv[1] == 'local':  
    # run program in local  
    r = process(BINARY)  
else:  
    exit(1)  
  
r.interactive()
```

template

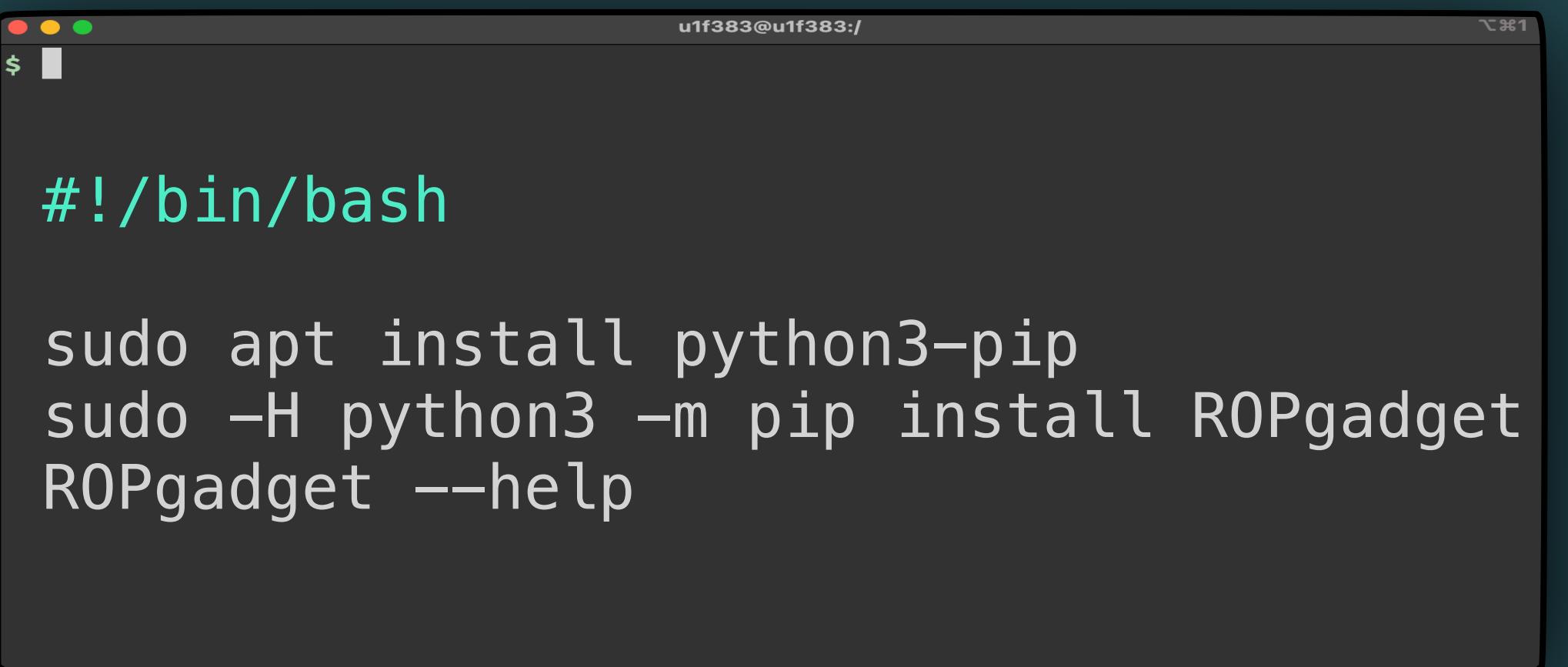
```
u1f383@u1f383:/ ~%1  
$ !#/usr/bin/python3  
  
... # init  
  
r.send('A') # "A"  
r.sendline('A') # "A\n"  
  
r.recvline() # recv until line  
r.recvuntil('A') # stop until recv 'A'  
  
r.sendafter('A', 'B') # after recv 'A', send 'B'  
r.sendlineafter('A', 'B') # after recv 'A', send 'B\n'  
  
# little endian  
# b'\x11\x00\xff\xee\xdd\xcc\xbb\xaa'  
p64(0xAABBCCDDEEFF0011)  
  
# big endian  
# 0xAABBCCDDEEFF0011  
u64(b'\x11\x00\xff\xee\xdd\xcc\xbb\xaa')  
  
r.interactive()
```

send / recv

\$ Appendix

Tools

- ▶ ROPgadget - search helpful ROP gadgets
 - ⦿ Included in `pwntools` by default
- ▶ `ROPgadget --binary ./test --only "pop|ret"`
 - ⦿ Search gadgets only contain `pop XXX ; ret`
- ▶ `ROPgadget --binary ./test --string "/bin/sh"`
 - ⦿ Search string “/bin/sh” in binary

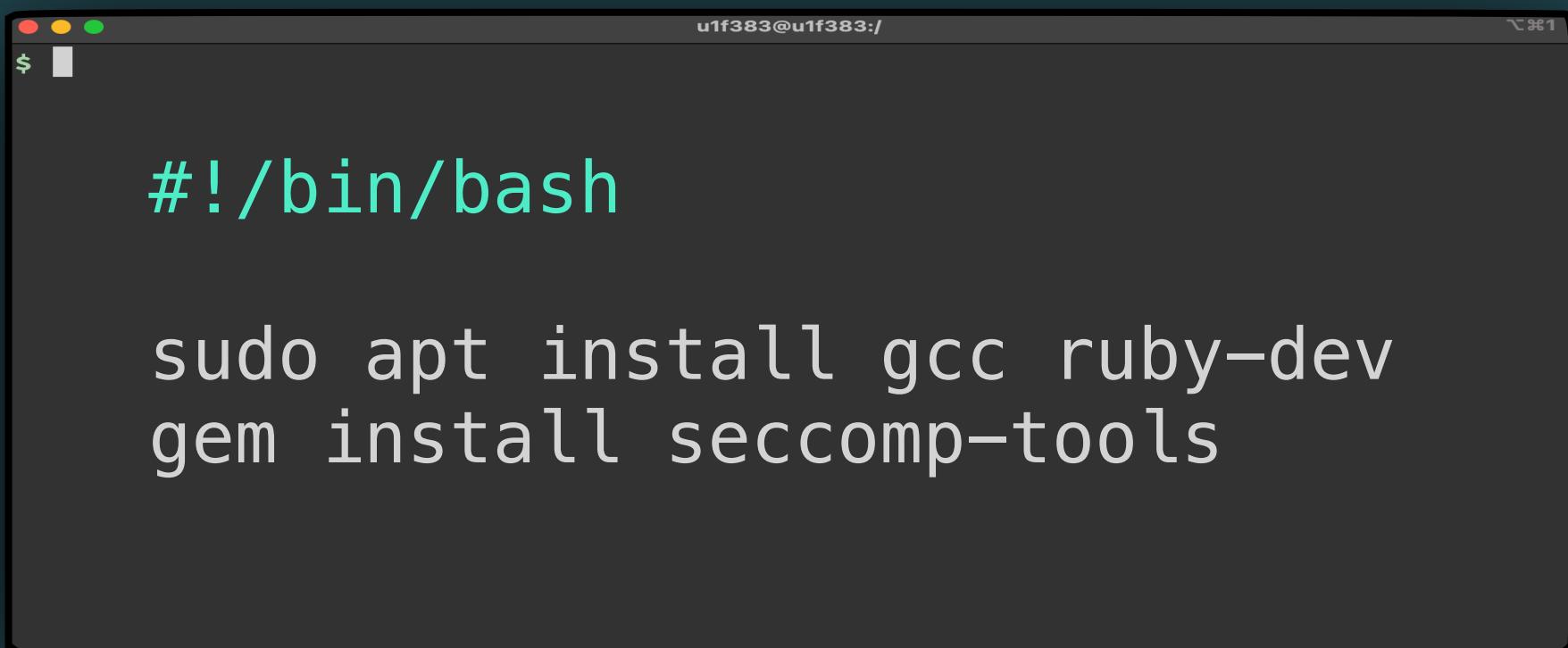


```
#!/bin/bash
sudo apt install python3-pip
sudo -H python3 -m pip install ROPgadget
ROPgadget --help
```

\$ Appendix

Tools

- ▶ seccomp-tools - powerful tools for seccomp analysis
- ▶ *seccomp-tools* dump ./test

A screenshot of a terminal window with a dark background and light text. The window title bar says "u1f383@u1f383:/". The prompt shows a dollar sign (\$) followed by a vertical bar (|). The text inside the terminal is:

```
#!/bin/bash
sudo apt install gcc ruby-dev
gem install seccomp-tools
```

\$ Appendix

Tools

```
22:31:55 ➤ u1f383@OWO ➤ /tmp ➤
$ seccomp-tools dump ./test
line  CODE  JT  JF      K
=====
0000: 0x20 0x00 0x00 0x00000004 A = arch
0001: 0x15 0x00 0x05 0xc000003e if (A != ARCH_X86_64) goto 0007
0002: 0x20 0x00 0x00 0x00000000 A = sys_number
0003: 0x35 0x00 0x01 0x40000000 if (A < 0x40000000) goto 0005
0004: 0x15 0x00 0x02 0xffffffff if (A != 0xffffffff) goto 0007
0005: 0x15 0x00 0x01 0x0000003c if (A != exit) goto 0007
0006: 0x06 0x00 0x00 0x7fff0000 return ALLOW
0007: 0x06 0x00 0x00 0x00000000 return KILL
```

only `sys_exit` is allowed

seccomp-tools output

\$ Appendix

Tools

- ▶ One gadget - a good one gadget finder

```
#!/bin/bash

sudo apt install gcc ruby-dev
gem install one_gadget
```

- ▶ *one_gadget /lib/x86_64-linux-gnu/libc.so.6*
- ▶ *one_gadget --level8 /lib/x86_64-linux-gnu/libc.so.6*

```
0xe6c84 execve("/bin/sh", rsi, rdx)
constraints:
[rsi] == NULL || rsi == NULL
[rdx] == NULL || rdx == NULL

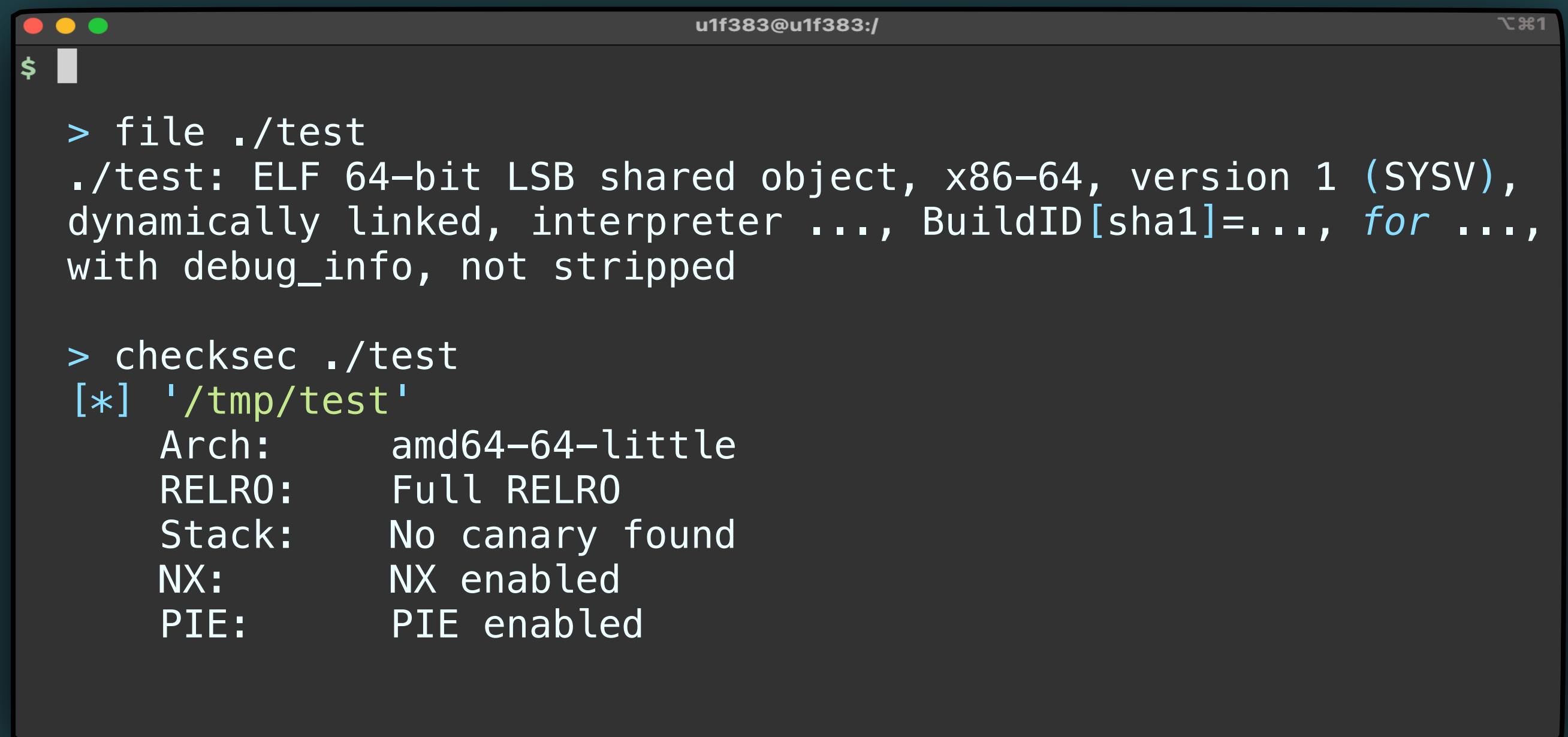
0xe6e73 execve("/bin/sh", r10, r12)
constraints:
address rbp-0x78 is writable
[r10] == NULL || r10 == NULL
[r12] == NULL || r12 == NULL

0xe6e76 execve("/bin/sh", r10, rdx)
constraints:
address rbp-0x78 is writable
[r10] == NULL || r10 == NULL
[rdx] == NULL || rdx == NULL
```

\$ Appendix

Tools

- ▶ *file* - determine file type
- ▶ *checksec* - a python/linux script to check binary secure settings
 - ⌚ Included in pwntools by default

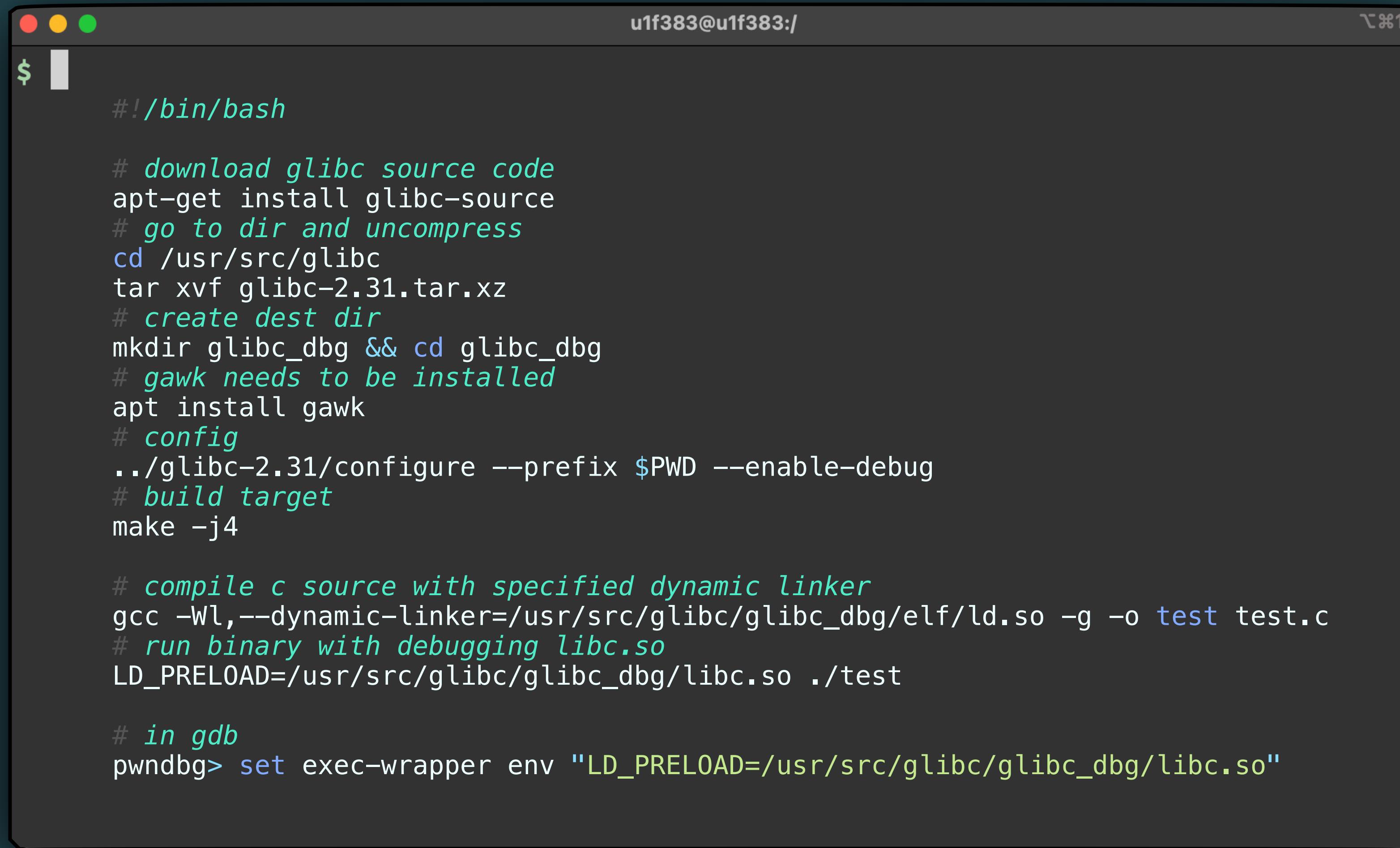


A terminal window showing two commands: `file ./test` and `checksec ./test`. The `file` command output indicates the file is an ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter ..., BuildID[sha1]=..., for ..., with debug_info, not stripped. The `checksec` command output provides detailed security analysis:

Setting	Value
Arch	amd64-64-little
RELRO	Full RELRO
Stack	No canary found
NX	NX enabled
PIE	PIE enabled

\$ Appendix

Debug glibc with source code



A terminal window titled "u1f383@u1f383:" showing a shell script for debugging glibc. The script includes comments explaining each step: downloading source code, extracting it, creating a debug directory, installing gawk, configuring the build, building the target, compiling a test program with a specified dynamic linker, running the binary with debugging symbols, and finally setting up a debugger wrapper.

```
#!/bin/bash

# download glibc source code
apt-get install glibc-source
# go to dir and uncompress
cd /usr/src/glibc
tar xvf glibc-2.31.tar.xz
# create dest dir
mkdir glibc_dbg && cd glibc_dbg
# gawk needs to be installed
apt install gawk
# config
../glibc-2.31/configure --prefix $PWD --enable-debug
# build target
make -j4

# compile c source with specified dynamic linker
gcc -Wl,--dynamic-linker=/usr/src/glibc/glibc_dbg/elf/ld.so -g -o test test.c
# run binary with debugging libc.so
LD_PRELOAD=/usr/src/glibc/glibc_dbg/libc.so ./test

# in gdb
pwndbg> set exec-wrapper env "LD_PRELOAD=/usr/src/glibc/glibc_dbg/libc.so"
```

\$ Appendix

Debug glibc with source code

```
0x55bc0c3d5050 <printf@plt>    endbr64
0x55bc0c3d5054 <printf@plt+4>   bnd jmp qword ptr [rip + 0x2f75]    <printf>
                                ↓
▶ 0x7f556d985b00 <printf>        endbr64
0x7f556d985b04 <printf+4>       sub   rsp, 0xd8
0x7f556d985b0b <printf+11>      mov   r10, rdi
0x7f556d985b0e <printf+14>      mov   qword ptr [rsp + 0x28], rsi
0x7f556d985b13 <printf+19>      mov   qword ptr [rsp + 0x30], rdx
0x7f556d985b18 <printf+24>      mov   qword ptr [rsp + 0x38], rcx
0x7f556d985b1d <printf+29>      mov   qword ptr [rsp + 0x40], r8
0x7f556d985b22 <printf+34>      mov   qword ptr [rsp + 0x48], r9
0x7f556d985b27 <printf+39>      test  al, al

In file: /usr/src/glibc/glibc-2.31/stdio-common/printf.c
23
24 /* Write formatted output to stdout from the format string FORMAT. */
25 /* VARARGS1 */
26 int
27 __printf (const char *format, ...)
▶ 28 {
29   va_list arg;
30   int done;
31
32   va_start (arg, format);
33   done = __vfprintf_internal (stdout, format, arg, 0);
```

gdb with source code

\$ Appendix

Pwnbox with docker

```
#!/bin/bash

set -e
if [ -z "$1" ]; then
    echo "Usage:";
    echo "Build environment: ./snippet build";
    echo "Up pwnbox daemon: ./snippet up";
    echo "Get shell: ./snippet shell";
    echo "Down pwnbox daemon: ./snippet down";
    exit 0
fi

if [ $1 == "build" ]; then
    mkdir pwnbox
    docker build -t pwnbox .
elif [ $1 == "up" ]; then
    docker run -it -d --cap-add=SYS_PTRACE --name pwnbox -v `pwd`/pwnbox:/pwnbox pwnbox
elif [ $1 == "shell" ]; then
    docker exec -it pwnbox fish
elif [ $1 == "down" ]; then
    docker stop pwnbox
fi
```

How to use

```
FROM ubuntu:20.04
MAINTAINER u1f383

ENV DEBIAN_FRONTEND=noninteractive
ENV LC_ALL=en_US.UTF-8

RUN apt update && \
    apt install -yq gcc && \
    apt install -yq gdb && \
    apt install -yq git && \
    apt install -yq ruby-dev && \
    apt install -yq vim-gtk3 && \
    apt install -yq fish && \
    apt install -yq glibc-source && \
    apt install -yq make && \
    apt install -yq gawk && \
    apt install -yq bison && \
    apt install -yq libseccomp-dev && \
    apt install -yq tmux && \
    apt install -yq wget && \
    apt install -yq locales && \
    locale-gen en_US.UTF-8

# compile glibc-2.31
RUN cd /usr/src/glibc && \
    tar xvf glibc-2.31.tar.xz && \
    mkdir glibc_dbg && \
    cd glibc_dbg && \
    ../glibc-2.31/configure --prefix $PWD --enable-debug && \
    make -j4

# install pwndbg
RUN git clone https://github.com/pwndbg/pwndbg ~/pwndbg && \
    cd ~/pwndbg && \
    ./setup.sh

# install Pwngdb
RUN git clone https://github.com/scwuaptx/Pwngdb.git ~/Pwngdb && \
    cat ~/Pwngdb/.gdbinit >> ~/.gdbinit && \
    sed -i "s/source ~\`/peda\`/peda.py\`/g" ~/.gdbinit

RUN pip3 install pwntools==4.4.0
RUN gem install seccomp-tools one_gadget
RUN ln -s /usr/local/lib/python3.8/dist-packages/bin/ROPgadget /bin/ROPgadget
RUN echo "set-option -g default-shell /bin/fish" > /root/.tmux.conf

CMD ["/bin/fish"]
```

Dockerfile

\$ Appendix

Resources

- ▶ [bootlin-glibc](#) - glibc source code cross referencer
- ▶ [x64 syscall table](#) - x64 syscall table in linux 4.7
- ▶ [libc.rip](#) - use symbol offset to find libc version
- ▶ [glibc-all-in-one](#) - scripts to download & debug & complie glibc
- ▶ [how2heap](#) - heap exploitation tech collection