

# Alocação dinâmica de memória

Bib: The C Programming Language (cap. 8.7)

Bib: Computer Systems: A Programmer's Perspective (cap. 9.9)

## Programação em Sistemas Computacionais

João Pedro Patriarca ([joao.patriarca@isel.pt](mailto:joao.patriarca@isel.pt)), Gabinete F.0.23 do edifício F

ISEL, ADEETC, LEIC

# Agenda

---

- Introdução
  - Motivação, exemplo, características e API da biblioteca standard do C
- Exemplo de uma implementação
  - Interface programática e modelo de dados
  - Processo de alocação
  - Processo de libertação
  - Caso de utilização

# Agenda

---

- Introdução
  - Motivação, exemplo, características e API da biblioteca standard do C
- Exemplo de uma implementação
  - Interface programática e modelo de dados
  - Processo de alocação
  - Processo de libertação
  - Caso de utilização

# Motivação

---

- Aloca o espaço em memória estritamente necessário
- Aloca apenas quando precisa
- Liberta quando não precisa mais

# Exemplos

---

- Estruturas de dados dinâmicas: listas, árvores binárias, ...
  - A capacidade da coleção adapta-se à medida das necessidades, portanto não precisa considerar, à partida, uma capacidade máxima
- Representação de objetos de tipos compostos, cujo número é variável
  - Por exemplo, criação de um conjunto de objetos com base no processamento de dados com origem do *standard input*, ou ficheiro
- Exemplos de exercícios
  - Coletânea de exercícios mantida pelo prof. Ezequiel

# Comparação entre tipos de alocação

---

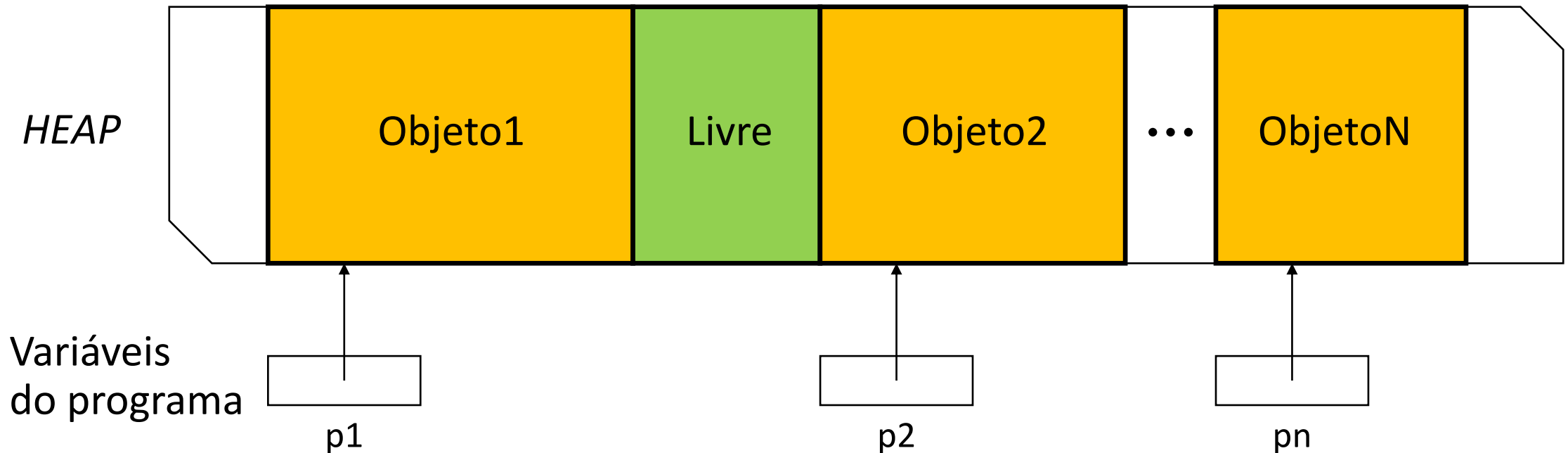
Tipo de alocação	Tempo de vida	Região em memória
Dinâmica	Dependente do utilizador (alocação e libertação explícita*)	<i>Heap</i>
Automática	Execução da função (alocação e libertação automática)	<i>Stack</i>
Estática	Execução do programa	Secções .data e .bss

\* Na alocação dinâmica, se se perder a referência do espaço alocado não mais será possível aceder-lhe ou libertá-lo

# Região para alocação dinâmica

---

- Conjunto de nós alocados ou livres de uma região de memória denominada por *Heap*



# Interface programática da biblioteca standard do C

---

```
#include <stdlib.h>

// malloc() allocates size bytes and returns a pointer to the allocated
// memory. The memory is not initialized. If size is 0 or there is no
// memory to allocate, then malloc() returns NULL...
void *malloc(size_t size);
// free() frees the memory space pointed to by ptr. If free(ptr) has
// already been called before or ptr have not been returned by a previous
// call to allocator functions, undefined behavior occurs. If ptr is NULL,
// no operation is performed.
void free(void *ptr);
// calloc() allocates memory for nmemb elements of size bytes each and
// returns a pointer to the allocated memory. The memory is set to zero...
void *calloc(size_t nmemb, size_t size);
// realloc() changes the size of the memory block pointed to by ptr to
// size bytes...
void *realloc(void *ptr, size_t size);
```



# Agenda

---

- Introdução
  - Motivação, exemplo, características e API da biblioteca standard do C
- Exemplo de uma implementação
  - Interface programática e modelo de dados
  - Processo de alocação
  - Processo de libertação
  - Caso de utilização

# Interface programática (1 de 2)

---

```
#ifndef _DYN_ALLOC_H
#define _DYN_ALLOC_H

void    init_heap(void);           // Inicia a infra estrutura
void *  xmalloc(size_t size);     // Aloca size bytes
void    xfree(void * ptr);        // Liberta o espaço alocado

#endif/*_DYN_ALLOC_H*/
```

# Interface programática (2 de 2)

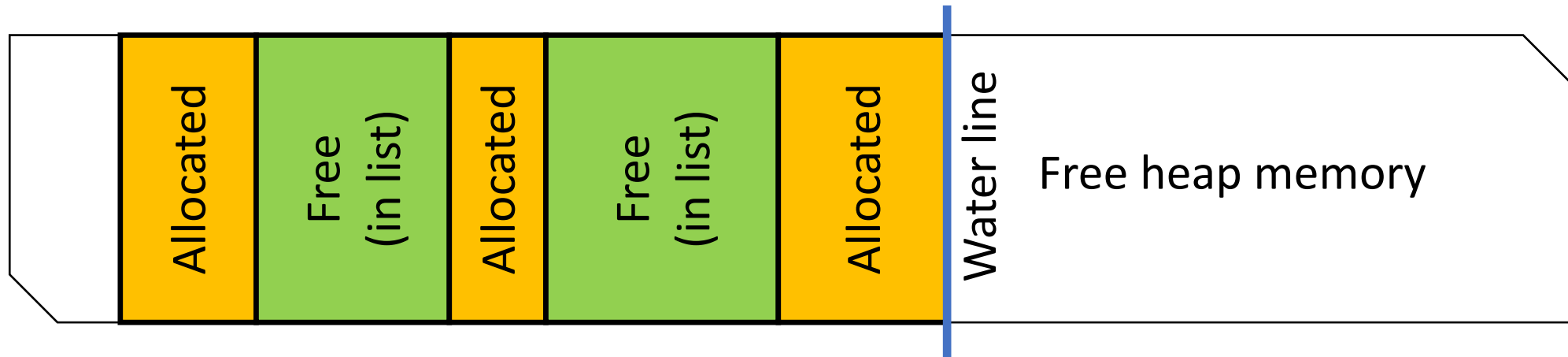
```
#ifndef _DYN_DBG_H
#define _DYN_DBG_H
/*****
 * Algoritmos para apresentar o estado do heap:
 * 1. print_free_list: percorre a lista de blocos livres apresentando
 *    o estado de cada nó.
 * 2. print_implicit_list: percorre lista implícita com todos os blocos
 *    livres e ocupados.
 * 3. print_heap_state: apresenta o estado do heap.
 *****/
void print_free_list(void);
void print_implicit_list(void);
void print_heap_state(void);
void print_all(void);

#endif/*_DYN_DBG_H*/
```

# Gestão do *Heap*

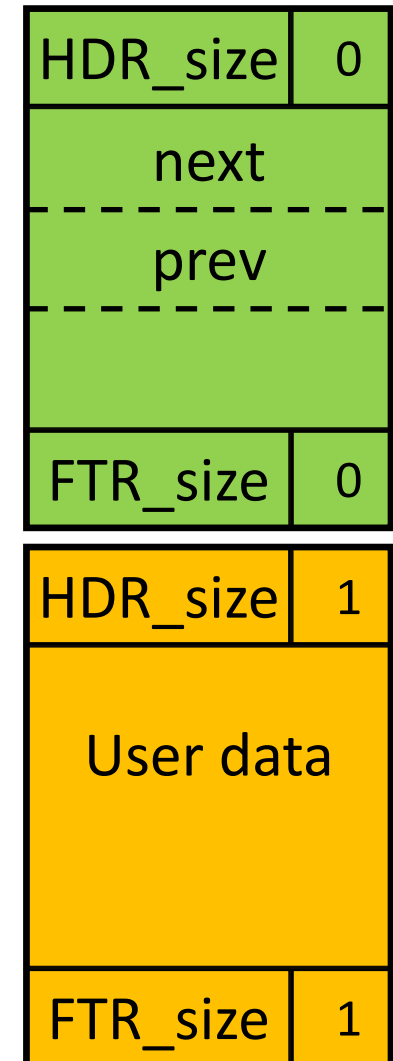
---

- Noção de linha de água
  - Reduz o número de blocos na lista de blocos livres
  - No processo de alocação, solicita um novo bloco da região livre (*free heap memory*) caso não exista um bloco que satisfaça a dimensão na lista de blocos livres
  - Na libertação, devolve o bloco libertado à região livre caso esteja junto à linha de água



# Caracterização de um bloco livre e de um bloco alocado

- Todos os blocos (livres e alocados) são constituídos por um cabeçalho e por um rodapé
  - Contêm a dimensão do bloco e se está livre (0) ou alocado (1)
  - Formam uma lista implícita duplamente ligada
  - Nesta implementação, considera-se dimensão igual a 4 bytes
- Um bloco livre contém dois ponteiros para o manter numa lista duplamente ligada, apenas de blocos livres
  - A dimensão mínima de um bloco tem de suportar o cabeçalho, rodapé e dois ponteiros
- A dimensão total do bloco alocado depende da dimensão solicitada na alocação
  - A dimensão do bloco é múltipla de 8 bytes para arquiteturas de 64 bits, tal como o ponteiro para o primeiro byte de *User data*



# Processo de alocação

---

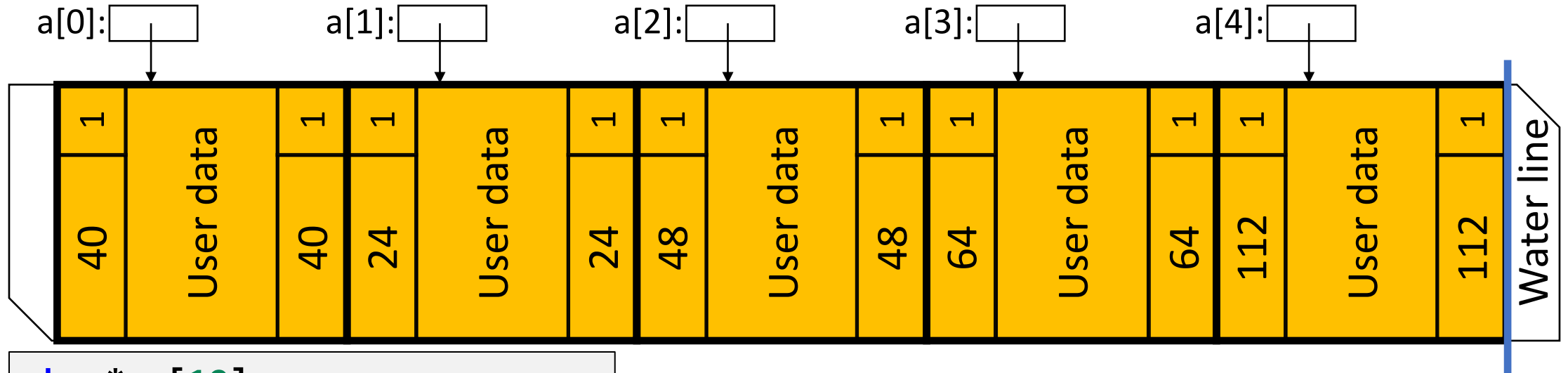
1. Procura na lista de blocos livres por um bloco com dimensão que satisfaça o espaço solicitado
  - Devolve **NULL**, se não existir qualquer bloco livre que satisfaça a dimensão solicitada
2. Retira o bloco da lista e fragmenta-o caso a dimensão do bloco sobrante seja igual ou superior à dimensão mínima de um bloco
  - O bloco sobrante é adicionado à lista de blocos livres
  - ✓ Otimiza a utilização do recurso memória
  - x A fragmentação reduz a dimensão de blocos livres
3. Marca o bloco como alocado e devolve um ponteiro para o primeiro byte útil do bloco (primeiro byte após o cabeçalho)

# Processo de libertação

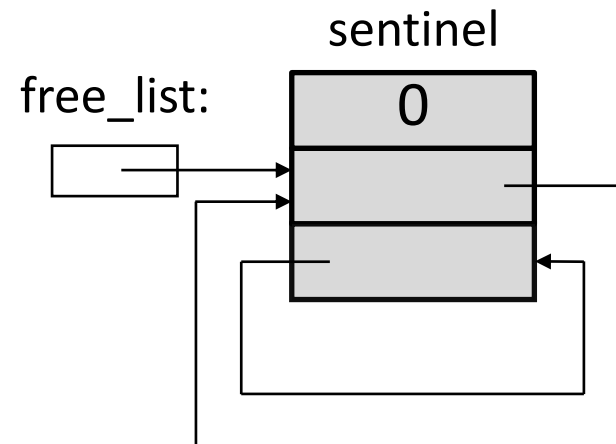
---

1. Marca o bloco como livre
2. Verifica se os blocos vizinhos em memória estão livres
  - Em caso afirmativo, remove o bloco livre adjacente da lista de blocos livres e junta-o com o bloco a libertar formando um novo bloco
  - ✓Previne a fragmentação de memória
3. O bloco é adicionado à lista de blocos livres

# Caso de utilização (1 de 8)

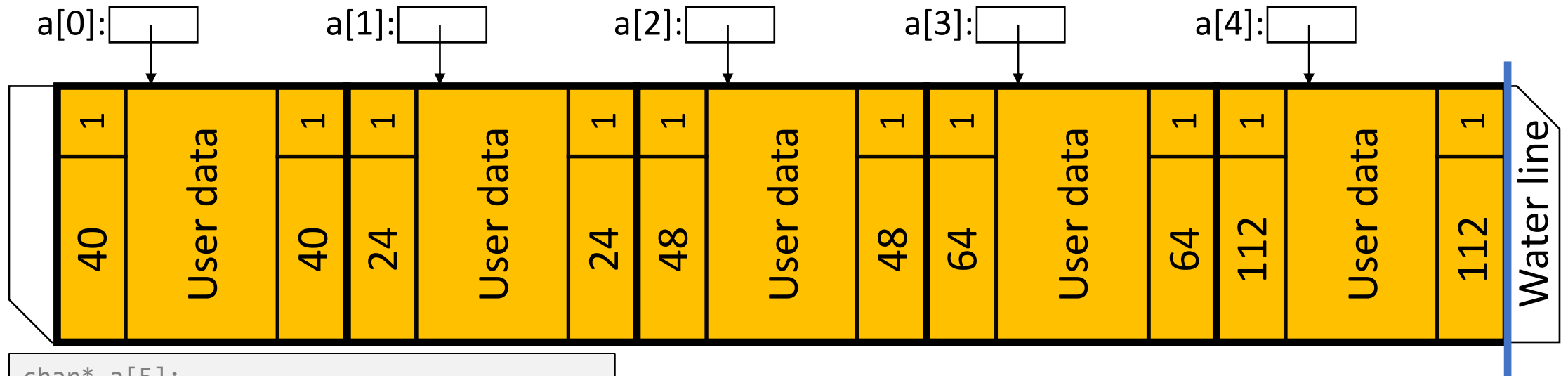


```
char* a[10];  
a[0] = (char*)xmalloc(32);  
a[1] = (char*)xmalloc(2);  
a[2] = (char*)xmalloc(33);  
a[3] = (char*)xmalloc(50);  
a[4] = (char*)xmalloc(100);
```

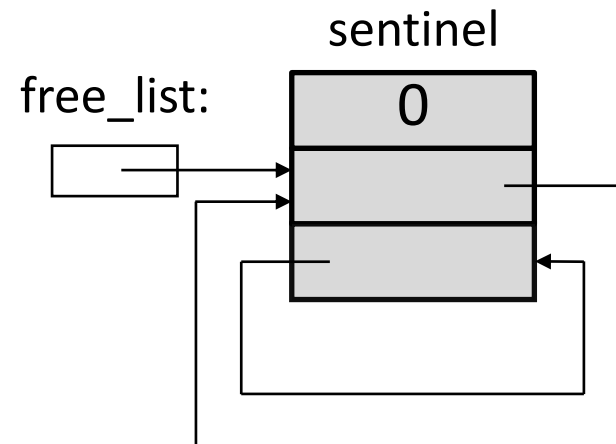




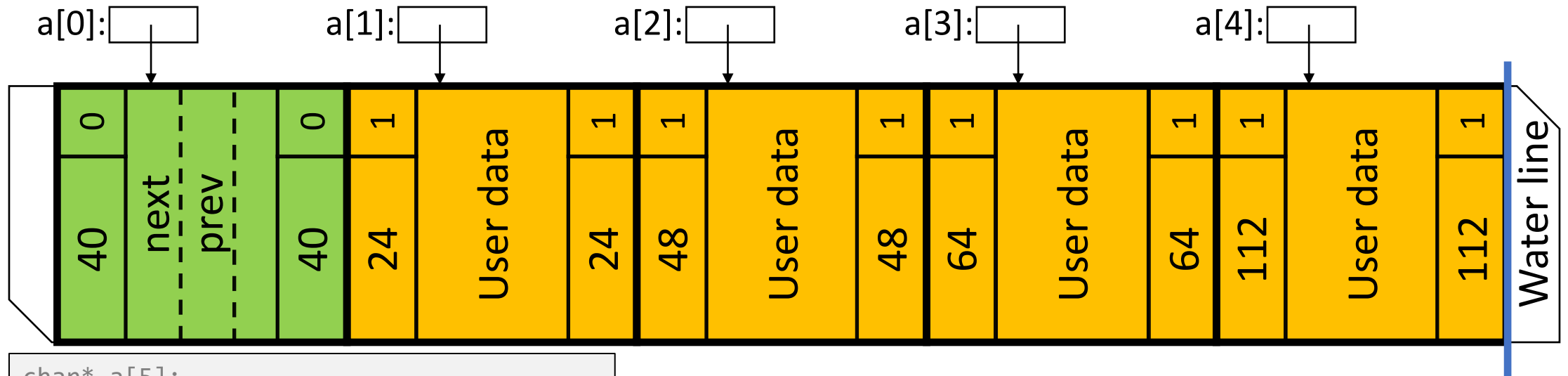
# Caso de utilização (2 de 8 – antes)



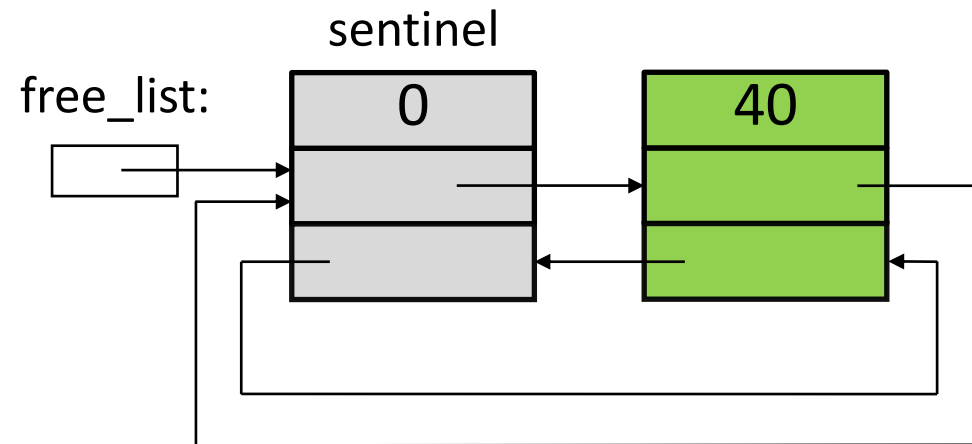
```
char* a[5];  
a[0] = (char*)xmalloc(32);  
a[1] = (char*)xmalloc(2);  
a[2] = (char*)xmalloc(33);  
a[3] = (char*)xmalloc(50);  
a[4] = (char*)xmalloc(100);  
xfree(a[0]);
```



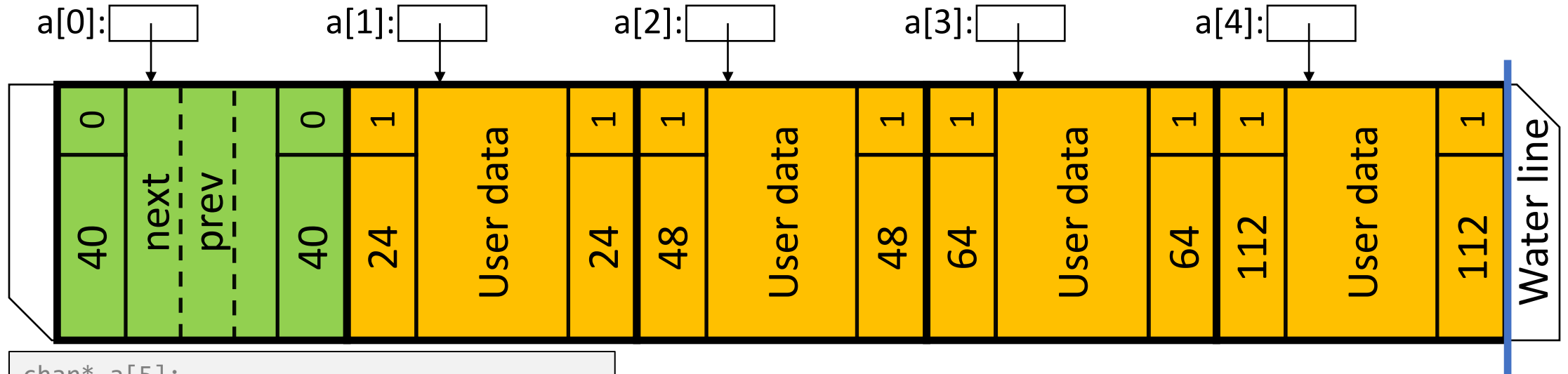
# Caso de utilização (2 de 8 – depois)



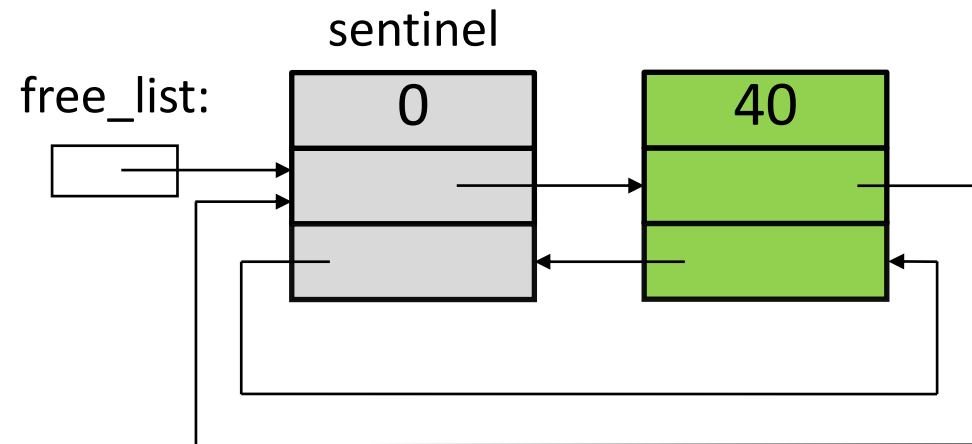
```
char* a[5];  
a[0] = (char*)xmalloc(32);  
a[1] = (char*)xmalloc(2);  
a[2] = (char*)xmalloc(33);  
a[3] = (char*)xmalloc(50);  
a[4] = (char*)xmalloc(100);  
xfree(a[0]);
```



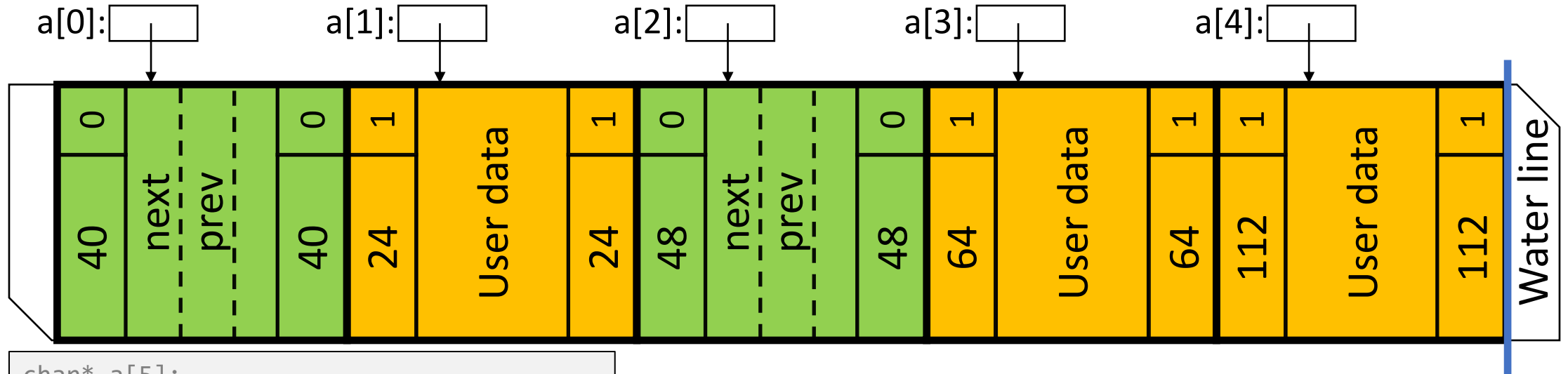
# Caso de utilização (3 de 8 – antes)



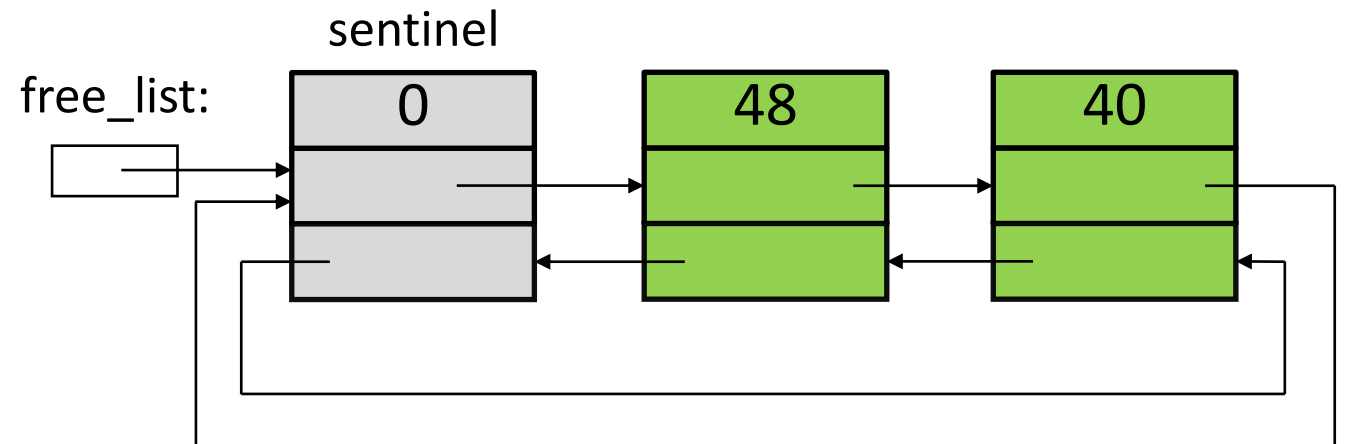
```
char* a[5];  
a[0] = (char*)xmalloc(32);  
a[1] = (char*)xmalloc(2);  
a[2] = (char*)xmalloc(33);  
a[3] = (char*)xmalloc(50);  
a[4] = (char*)xmalloc(100);  
xfree(a[0]);  
xfree(a[2]);
```



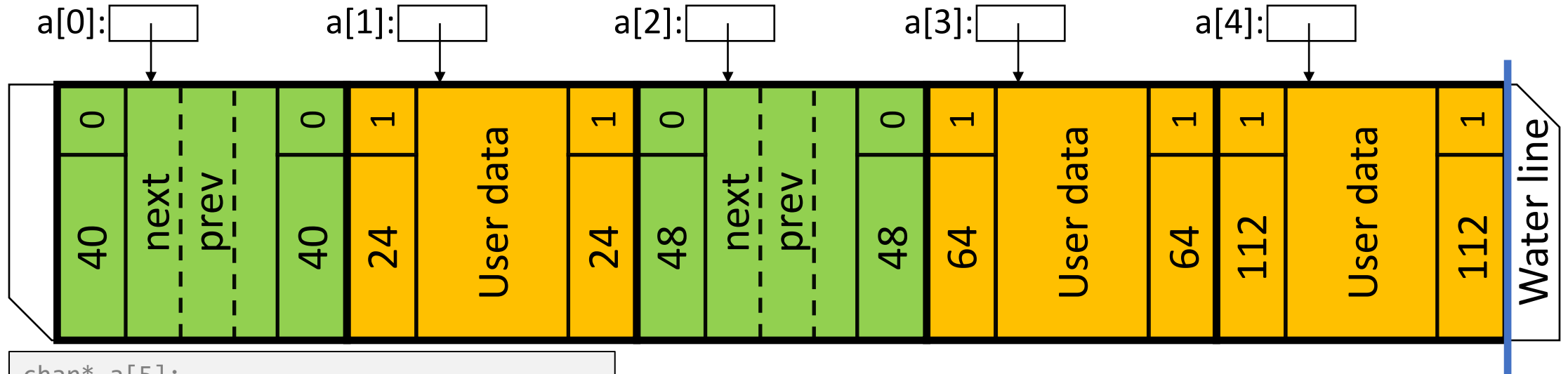
# Caso de utilização (3 de 8 – depois)



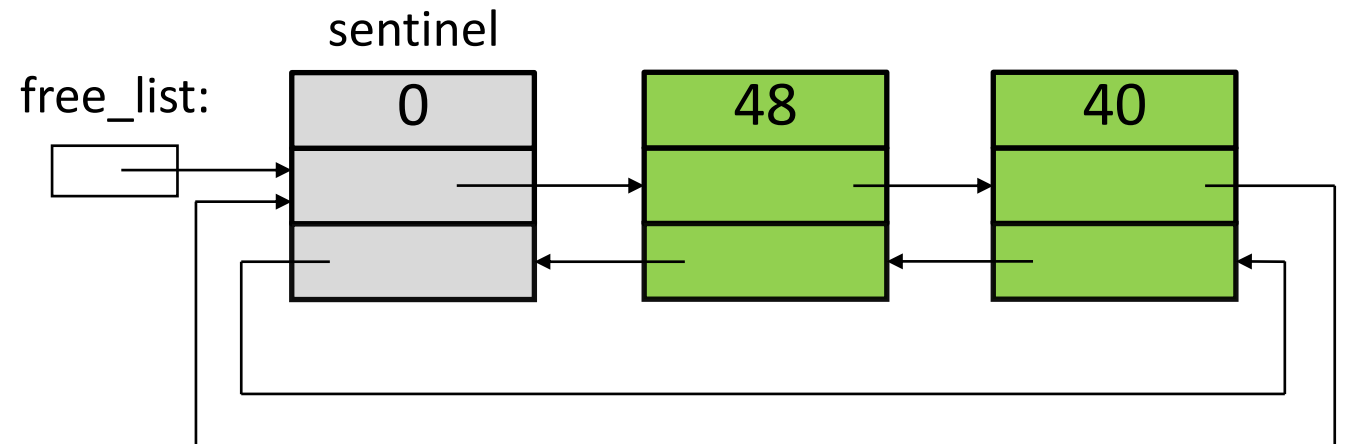
```
char* a[5];  
a[0] = (char*)xmalloc(32);  
a[1] = (char*)xmalloc(2);  
a[2] = (char*)xmalloc(33);  
a[3] = (char*)xmalloc(50);  
a[4] = (char*)xmalloc(100);  
xfree(a[0]);  
xfree(a[2]);
```



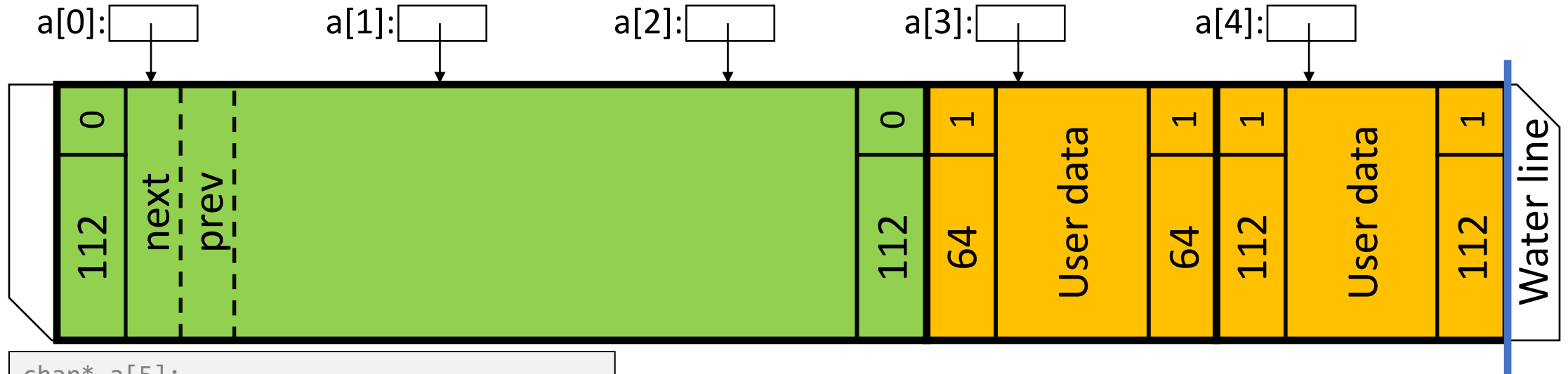
# Caso de utilização (4 de 8 – antes)



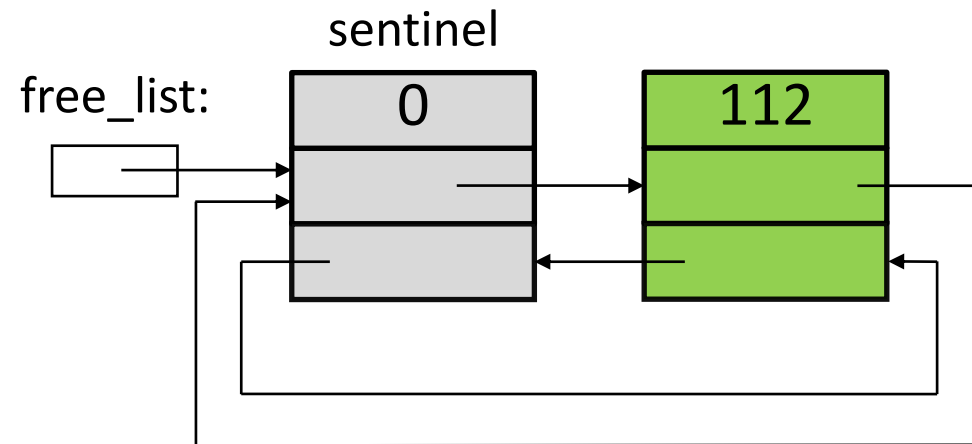
```
char* a[5];  
a[0] = (char*)xmalloc(32);  
a[1] = (char*)xmalloc(2);  
a[2] = (char*)xmalloc(33);  
a[3] = (char*)xmalloc(50);  
a[4] = (char*)xmalloc(100);  
xfree(a[0]);  
xfree(a[2]);  
xfree(a[1]);
```



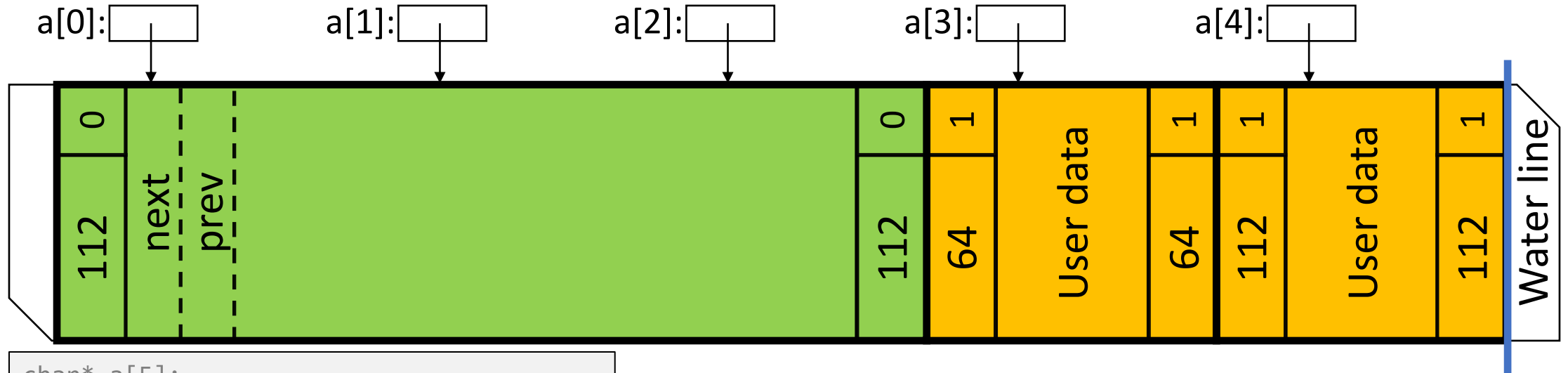
# Caso de utilização (4 de 8 – depois)



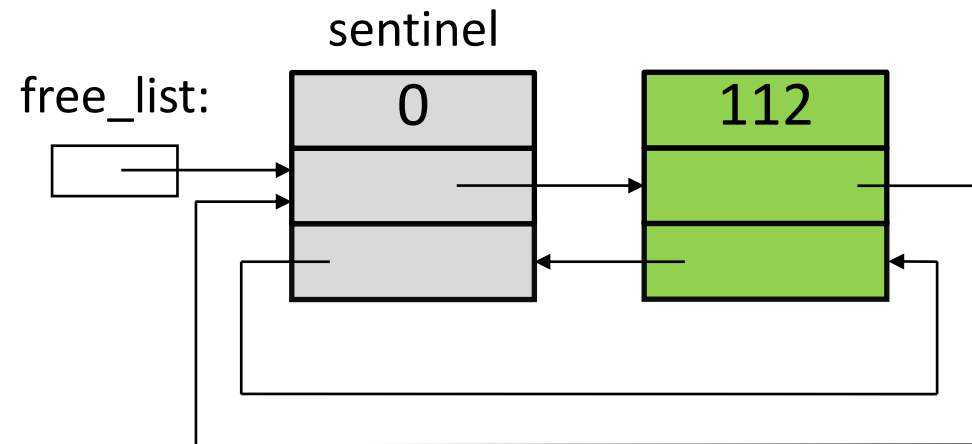
```
char* a[5];  
a[0] = (char*)xmalloc(32);  
a[1] = (char*)xmalloc(2);  
a[2] = (char*)xmalloc(33);  
a[3] = (char*)xmalloc(50);  
a[4] = (char*)xmalloc(100);  
xfree(a[0]);  
xfree(a[2]);  
xfree(a[1]);
```



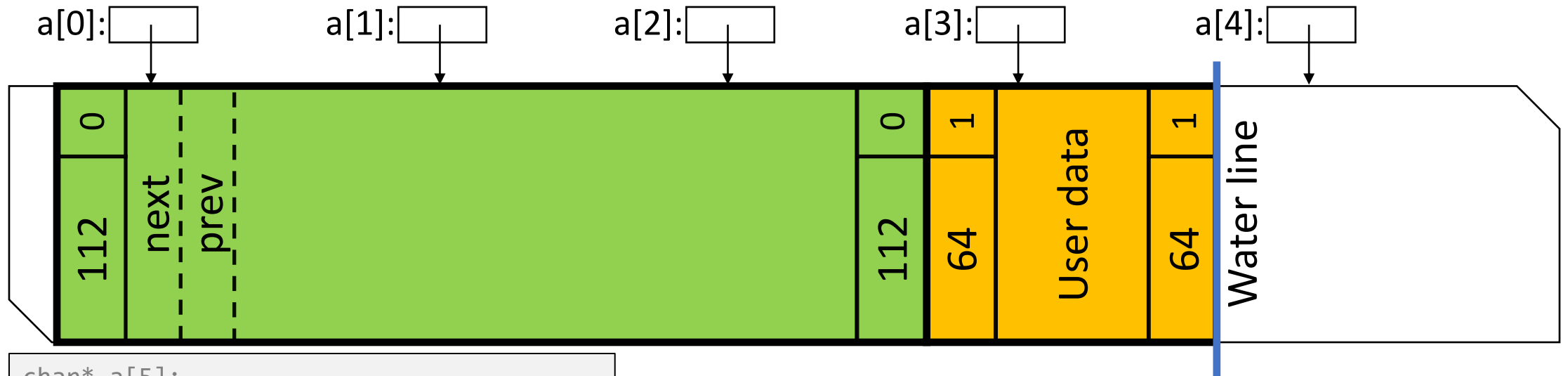
# Caso de utilização (5 de 8 – antes)



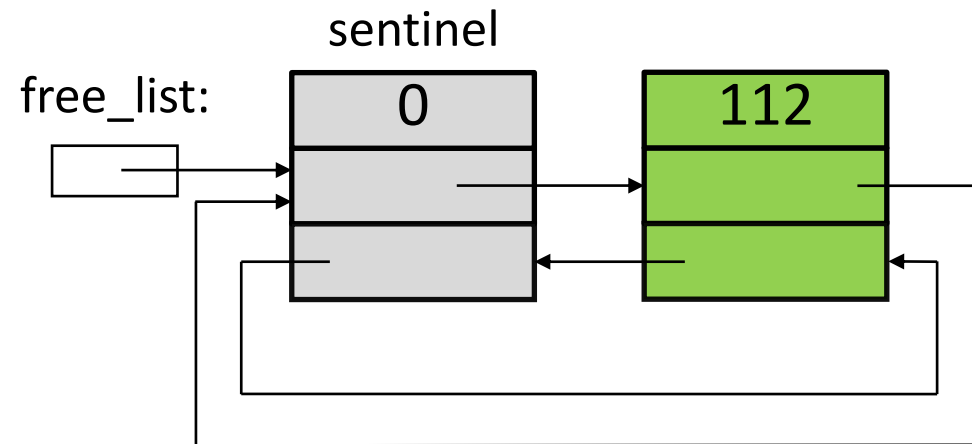
```
char* a[5];  
a[0] = (char*)xmalloc(32);  
a[1] = (char*)xmalloc(2);  
a[2] = (char*)xmalloc(33);  
a[3] = (char*)xmalloc(50);  
a[4] = (char*)xmalloc(100);  
xfree(a[0]);  
xfree(a[2]);  
xfree(a[1]);  
xfree(a[4]);
```



# Caso de utilização (5 de 8 – depois)

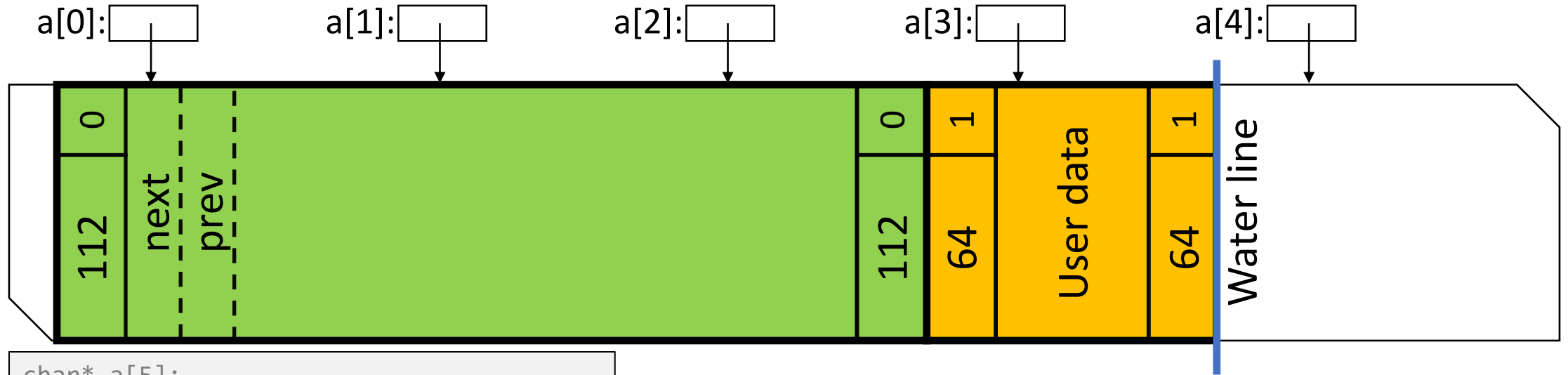


```
char* a[5];  
a[0] = (char*)xmalloc(32);  
a[1] = (char*)xmalloc(2);  
a[2] = (char*)xmalloc(33);  
a[3] = (char*)xmalloc(50);  
a[4] = (char*)xmalloc(100);  
xfree(a[0]);  
xfree(a[2]);  
xfree(a[1]);  
xfree(a[4]);
```

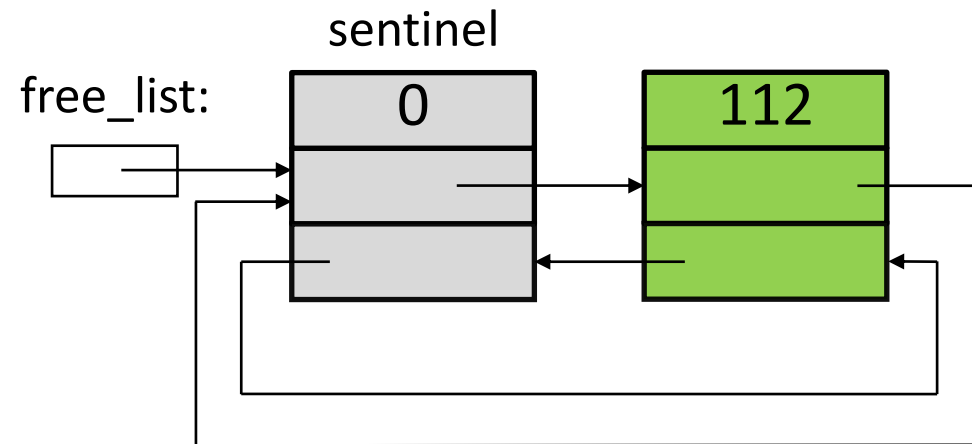




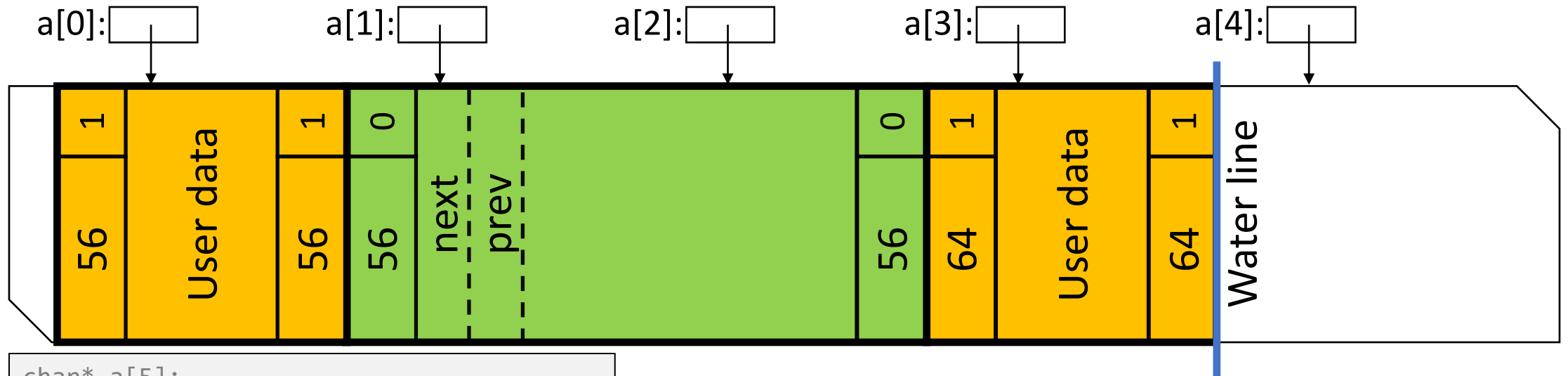
# Caso de utilização (6 de 8 – antes)



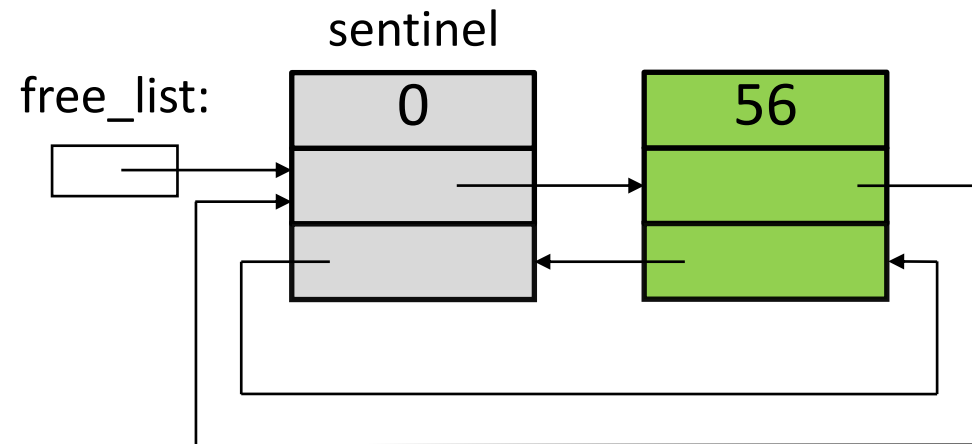
```
char* a[5];  
a[0] = (char*)xmalloc(32);  
a[1] = (char*)xmalloc(2);  
a[2] = (char*)xmalloc(33);  
a[3] = (char*)xmalloc(50);  
a[4] = (char*)xmalloc(100);  
xfree(a[0]);  
xfree(a[2]);  
xfree(a[1]);  
xfree(a[4]);  
a[0] = (char*)xmalloc(45);
```



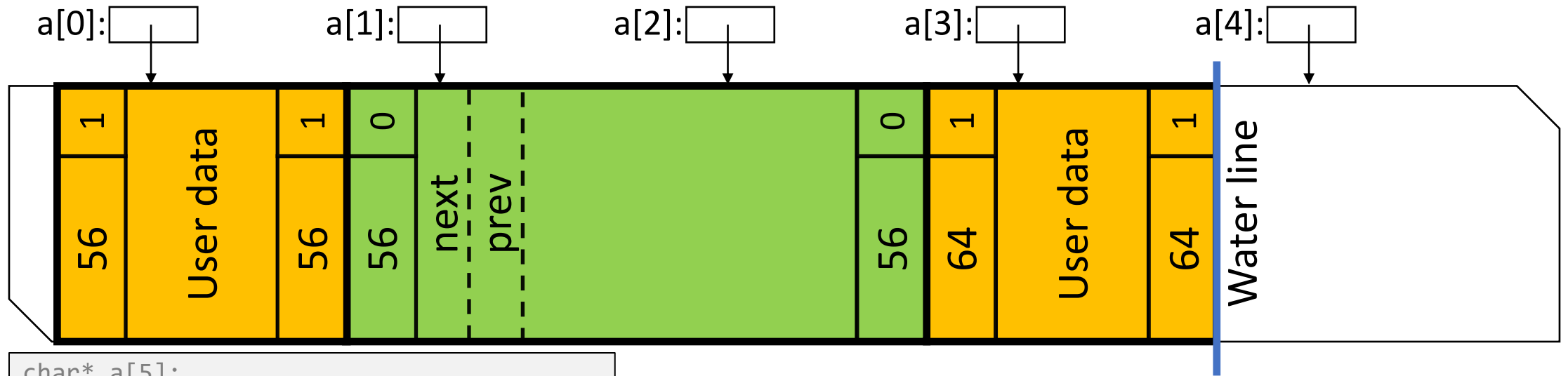
# Caso de utilização (6 de 8 – depois)



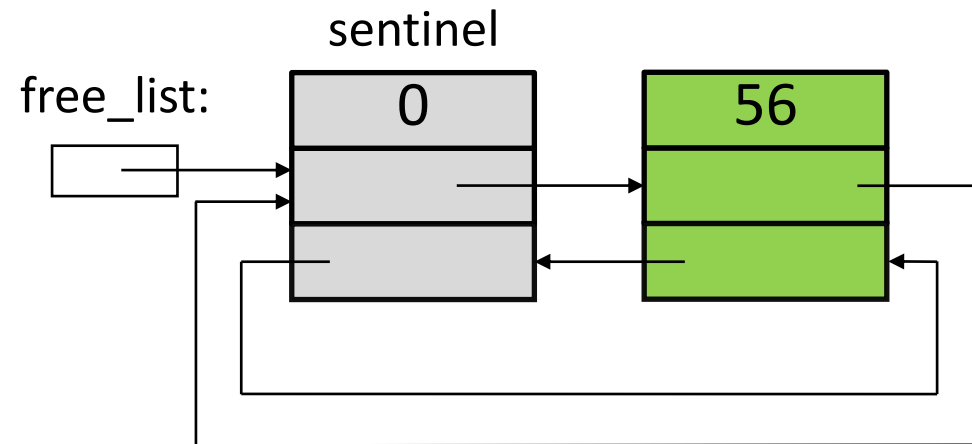
```
char* a[5];  
a[0] = (char*)xmalloc(32);  
a[1] = (char*)xmalloc(2);  
a[2] = (char*)xmalloc(33);  
a[3] = (char*)xmalloc(50);  
a[4] = (char*)xmalloc(100);  
xfree(a[0]);  
xfree(a[2]);  
xfree(a[1]);  
xfree(a[4]);  
a[0] = (char*)xmalloc(45);
```



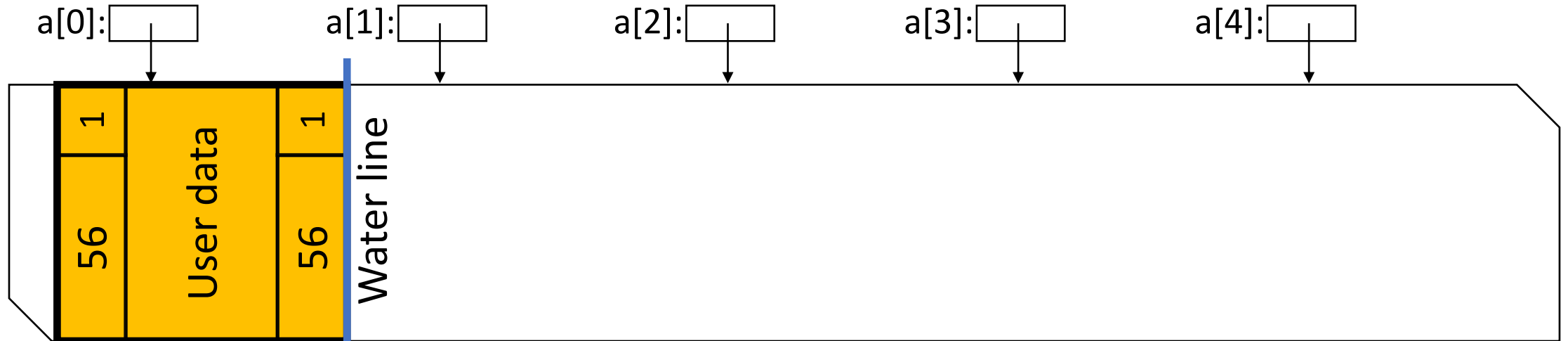
# Caso de utilização (7 de 8 – antes)



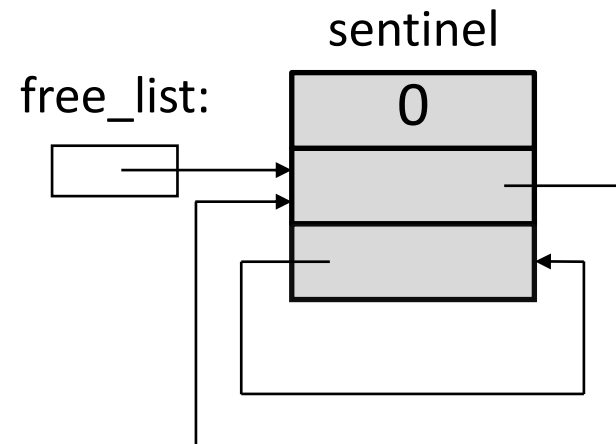
```
char* a[5];  
a[0] = (char*)xmalloc(32);  
a[1] = (char*)xmalloc(2);  
a[2] = (char*)xmalloc(33);  
a[3] = (char*)xmalloc(50);  
a[4] = (char*)xmalloc(100);  
xfree(a[0]);  
xfree(a[2]);  
xfree(a[1]);  
xfree(a[4]);  
a[0] = (char*)xmalloc(45);  
xfree(a[3]);
```



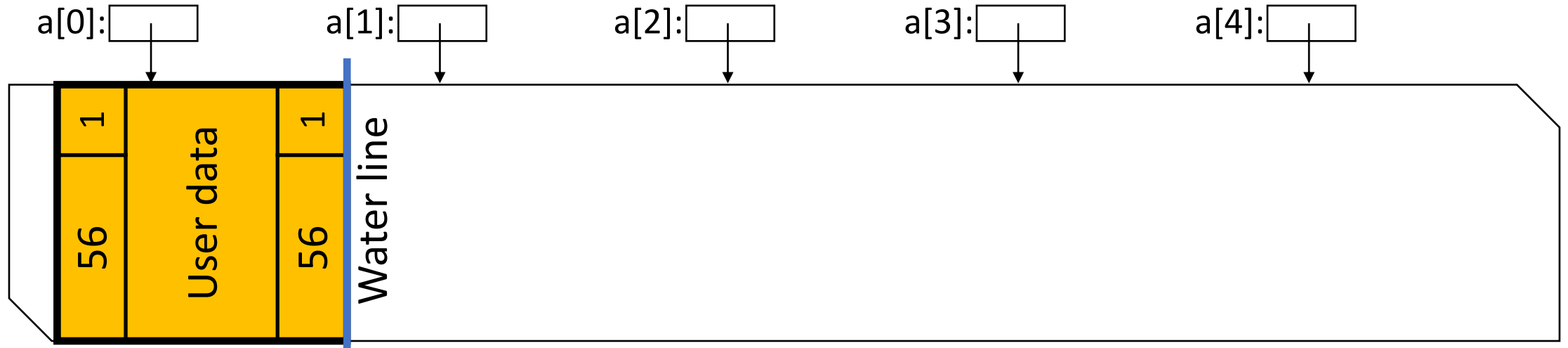
# Caso de utilização (7 de 8 – depois)



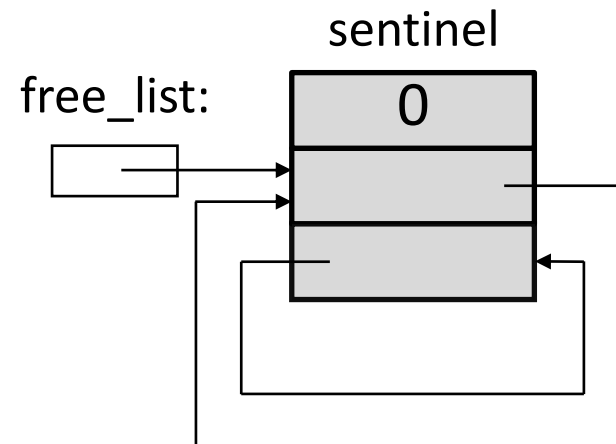
```
char* a[5];  
a[0] = (char*)xmalloc(32);  
a[1] = (char*)xmalloc(2);  
a[2] = (char*)xmalloc(33);  
a[3] = (char*)xmalloc(50);  
a[4] = (char*)xmalloc(100);  
xfree(a[0]);  
xfree(a[2]);  
xfree(a[1]);  
xfree(a[4]);  
a[0] = (char*)xmalloc(45);  
xfree(a[3]);
```



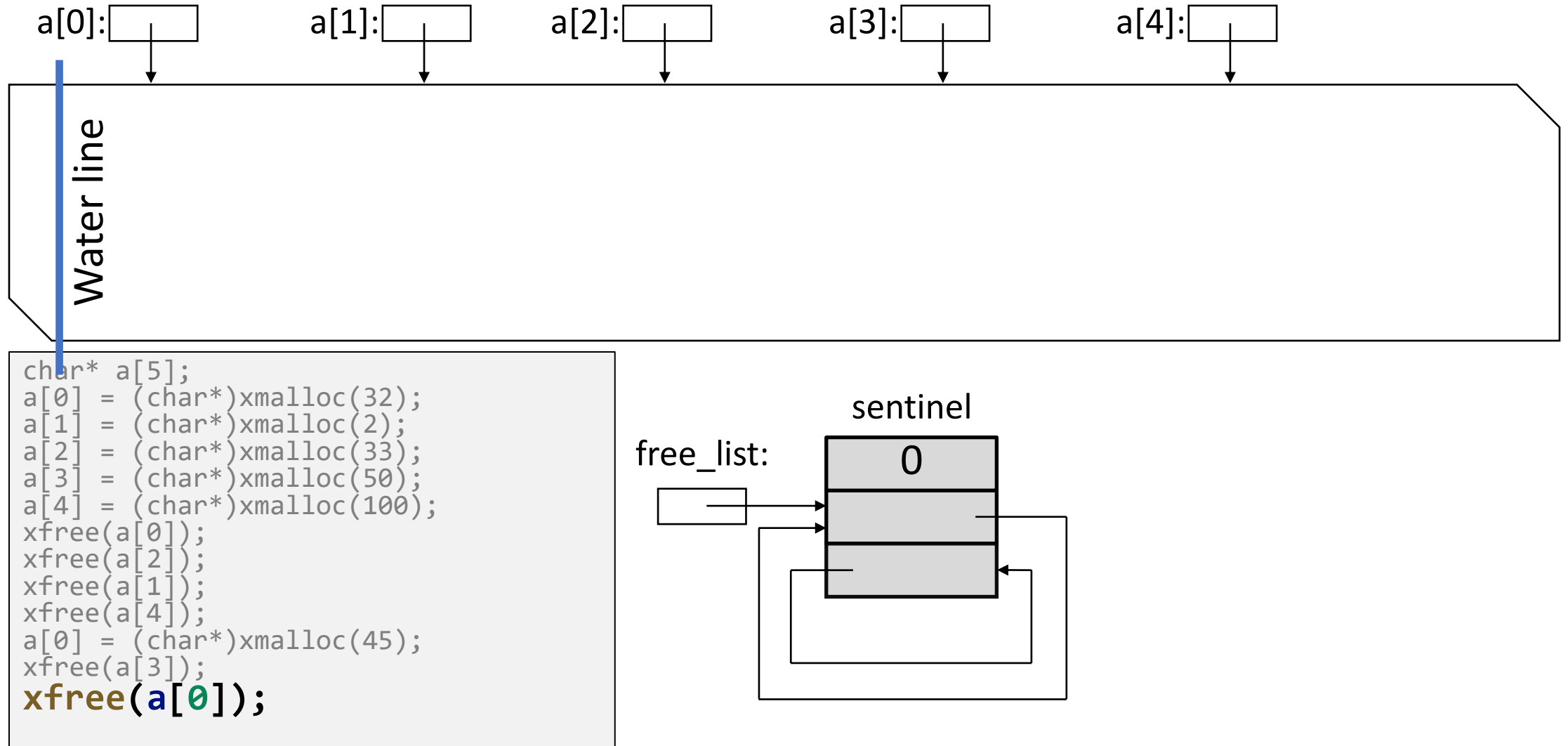
# Caso de utilização (8 de 8 – antes)



```
char* a[5];  
a[0] = (char*)xmalloc(32);  
a[1] = (char*)xmalloc(2);  
a[2] = (char*)xmalloc(33);  
a[3] = (char*)xmalloc(50);  
a[4] = (char*)xmalloc(100);  
xfree(a[0]);  
xfree(a[2]);  
xfree(a[1]);  
xfree(a[4]);  
a[0] = (char*)xmalloc(45);  
xfree(a[3]);  
xfree(a[0]);
```



# Caso de utilização (8 de 8 – depois)



# Exercício 1

---

O tipo **dblock\_node\_t** é usado para definir listas simplesmente ligadas de blocos de dados com dimensão variável, armazenados em instâncias do tipo **dblock\_t**. A função **split\_dblock** divide o bloco de dados passado com o argumento **dblock** em blocos com dimensão **size**, retornando-os numa lista ligada que respeita a ordem dos dados no bloco original; o último bloco da lista ficará com a dimensão que restar. A função **split\_dblock** deverá libertar toda memória utilizada no argumento **dblock**. A função **join\_blocks** cria um bloco único com a concatenação de todos os blocos de dados que constam na lista passada com o argumento **dblist** e liberta toda a memória utilizada pelo argumento **dblist**. Implemente as duas funções em linguagem C.

Nota: Para efeitos da libertação da memória considere: (1) as instâncias do tipo **dblock\_t** usam a memória com o critério definido pela função **join\_dblocks**, e; (2) a lista de instâncias de **dblock\_node\_t** usa a memória com os critérios definidos na função **split\_dblock**.

```
typedef struct dblock { size_t length; unsigned char *raw_data; } dblock_t;

typedef struct dblock_node { struct dblock_node *next; dblock_t dblock; }
dblock_node_t;

dblock_node_t *split_dblock(dblock_t *dblock, size_t size);

dblock_t *join_dblocks(dblock_node_t *dblist);
```

## Exercício 2

---

Considere que se pretende converter uma lista de cartões bancários organizada em lista ligada para uma organização em *array* de ponteiros. O tipo **Card** contém os dados de um cartão: número, estado e respetivo titular. O tipo **node\_card\_t** é um tipo auxiliar usado para construir listas simplesmente ligadas de cartões; o campo **next** indica o elemento seguinte e o campo **card** aponta para uma instância de **Card** com os dados do cartão. O tipo **array\_card\_t** armazena um *array* de ponteiros para os dados dos cartões, acompanhado da respetiva dimensão (**ncard**).

Programe em linguagem C as funções **list\_to\_array**, que converte uma lista simplesmente ligada de cartões num *array* de cartões e **array\_to\_list** que realiza a operação contrária. Durante o processo de conversão, a memória utilizada pelas estruturas de dados originais deve ser libertada.

Nota: Quando se pretende usar estruturas com um campo do tipo *array* com dimensão variável, o compilador de C permite declarar o *array* sem se especificar a dimensão, desde que o *array* seja o último campo da estrutura. Este campo do tipo *array* pode ser referido normalmente no código, contudo não é tido em consideração na determinação de **sizeof** da estrutura.

```
typedef struct card {
    unsigned long number: 50;
    unsigned long state: 3;
    unsigned long hlen: 8;
    char holder[];
} card_t;

typedef struct node_card {
    struct node_card *next;
    struct card *card;
} node_card_t;

typedef struct array_card {
    unsigned int ncard;
    struct card *array[];
} array_card_t;

array_card_t *list_to_array(
    node_card_t *list
);

node_card_t *array_to_list(
    array_card_t *array
);
```