Computer Architecture

Programming in C

Memory allocation

Agenda

- Explicit allocation: the `malloc` function
- Why do we need `malloc`?
- Example: linked list
- Exercises

Explicit allocation of memory

```c
#include<stdio.h>  //necessary to use printf, sizeof
#include<stdlib.h> //necessary to use malloc, free

int main(){
    int *p;
    //allocate a region of ten times the size of int
    //store the address of the allocated region in p
    p = malloc(10*sizeof(int));
    p[0] = 38;
    p[1] = 42;
    p[2] = p[1] + 3;
    for(int i=0; i<10; i++) printf("%d ", p[i]);
    printf("\n");

    free(p);

    return 0;
}
```

`malloc(10 * sizeof(int));`

- calls the function `malloc` defined in the library `stdlib`
- sends as argument the size of memory to allocate
- an equivalent instruction: `malloc(sizeof(int[10]));`

The function `malloc` does two things:

- it allocates a region in memory of the required size (in a single block, and not initialized).
- it returns as a pointer the address of the first byte of the allocated region.

Typically, the instruction:
`p = malloc(10 * sizeof(int));` stores the address of the allocated memory in the pointer `p`.

The `free(p);` deallocates the memory region allocated by `malloc`.

Memory leak

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

int main(){
    int *p;

    p = malloc(10 * sizeof(int));
    p[0] = 1;
    p[1] = 2;

    //reassign p to a new allocated region
    p = malloc(10 * sizeof(int));
    //p is overwritten; the address of the
    //first malloc is lost.
}
```

The code in the figure will result in a memory leak:

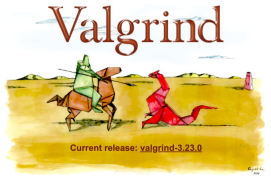
- The address of the region allocated by the first `malloc` has been lost because `p` is overwritten by the address of the region allocated by the second `malloc`.

It will is **impossible** to free this lost region before the end of the program execution!!!

- If too much memory allocated and not free ⇒ not enough memory, resulting in performance degradation, or even crashes.

To avoid memory leaks, it is essential to always **pair** `malloc` **with** `free`.

Solving memory leak issues



- Most modern operating systems (e.g., Linux, macOS, Windows) automatically reclaim the memory that was allocated by the program, even if the program did not explicitly free it.
  - But it is not solve the memory leak:
    - for long-running programs (e.g., a server or background service), a memory leak can gradually consume more and more memory, leading slowdowns or crashers due to memory exhaustion.
    - Leaking memory means the program is not managing its resources properly. It should be solved during the development, not the OS.
- Embedded systems may not have capabilities to handle leaks, so, memory leaks can have severe consequences.
- Detecting memory leak issues: Valgrind is a powerful tool for detecting memory issues in programs.

Failure of allocation

```c
#include<stdio.h>  //necessary to use printf, sizeof
#include<stdlib.h> //necessary to use malloc, free

int main(){
    int *p;

    p = malloc(10*sizeof(int));
    //allocation can fail when not enough memory is available
    //malloc will returns a NULL if so.
    if(p==NULL) {
        printf("function...: error of allocation.\n");
        exit(EXIT_FAILURE);
    }
    p[0] = 38;
    p[1] = 42;
    p[2] = p[1] + 3;
    for(int i=0; i<10; i++) printf("%d ", p[i]);
    printf("\n");
    free(p);
    return 0;
}
```

Allocation can fail when not enough memory is available.

- In that case, the function `malloc` returns `NULL` (the “null” pointer)
- In principle, one should always treat the case of failure of allocation
- `exit(EXIT_FAILURE);` interrupts the program with the predefined value `EXIT_FAILURE` transmitted to the shell (which will then handle the error).

Variants of malloc

```c
#include<stdio.h>  //necessary to use printf, sizeof
#include<stdlib.h> //necessary to use malloc, free

int main(){
    int *p;
    //malloc will not initialize the values
    //in the allocated memory
    p = malloc(10*sizeof(int));
    for(int i=0; i<10; i++) printf("%d ", p[i]);
    printf("\n");
    free(p);
    //calloc will initialize the values to be 0
    //in the allocated memory
    p = calloc(10, sizeof(int));
    for(int i=0; i<10; i++) printf("%d ", p[i]);
    printf("\n");
    free(p);
    //realloc modifies the size of the allocated region of memory
    p = malloc(10*sizeof(int));
    for(int i=0; i<10; i++) p[i]=i;
    p = realloc(p, 20*sizeof(int));
    for(int i=0; i<20; i++) printf("%d ", p[i]);
    free(p);
    return 0;
}
```

- `p=calloc(10, sizeof(int));`
  - Equivalent to `p=malloc(10*sizeof(int));` except that the allocation memory is initialized with 0 values.
- `p=malloc(10*sizeof(int));`
  `p=realloc(p, 20*sizeof(int));`
  - modifies the size of the allocated region of memory (keep the data for a larger size, truncate them for a smaller size)
  - the reallocation may occur at a different address (in that case, the old region is freed, the value of p changes and the content of the old memory is copied to the new location)

Return type of malloc and generic pointers

```
5  int main(){
6      char *p;
7      p = malloc(8); //8 bytes
8      for(int i=0; i<8; i++){
9          p[i] = 97+i;
10     }
11     for(int i=0; i<8; i++) printf("%d ", &p[i]);
12     free(p);
13     printf("\n");
14     int *q;
15     q = malloc(8); //8 bytes;
16     //equal to malloc(2*sizeof(int))
17     for(int i=0; i<2; i++){
18         q[i] = i;
19     }
20     for(int i=0; i<2; i++) printf("%d ", &q[i]);
21     free(q);
22     return 0;
23 }
```

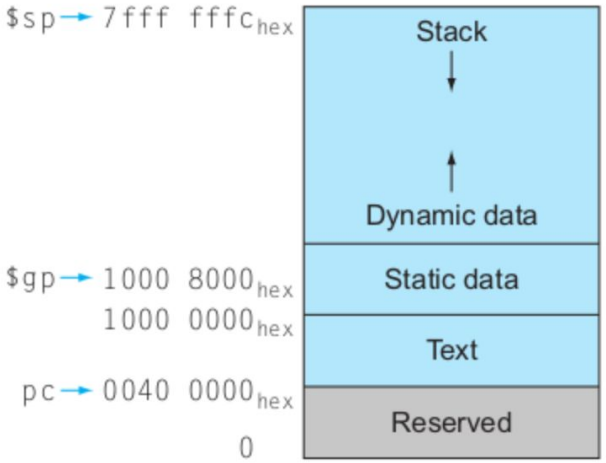
The printed addresses increment by 1

The printed addresses increment by 4

- generic pointer (`void *`) is a pointer that has no specific type. It can be assigned to any other pointer type without requiring an explicit cast.
- `malloc` returns a generic pointer, so it can be assigned to any pointer type.
  - It allocates a block memory of the request size (in bytes) and returns a pointer to the beginning of the block.
- We can set a explicit cast to the `malloc` by

```
int *p;
p = (int *)malloc(8);
```

Stack memory and Heap memory (dynamic data)



a typical memory layout for a C program:

- Stack
- Heap (a.k.a.,dynamic data)
- Static data
- Text segment (program instructions)
- Reserved (for OS buffer)

Stack memory is a memory buffer for small, fixed-size variable that have a known lifetime (e.g., local variables, function parameters)

- it is automatically managed, faster to allocate and deallocate than heap memory.

Heap memory is a memory buffer set aside for dynamic allocation, meaning that the size of the memory can be determined and allocated at runtime.

- It is manually managed, and is suitable for data that persists beyond function calls.

Stack grows downward, while heap grows upward.

The difference between Heap and Stack

Feature	Heap memory	Stack memory
Allocation / Deallocation	Manual (malloc/free)	Automatic (push/pop by the stack)
Lifetime	Exists until manually freed or program termination	Exists only during the function’s execution
Size	Typically larger (limited by available system memory)	Typically smaller (limited stack size, e.g., a few MB)
Performance	Slower (due to dynamic management)	Faster (automatic, managed by the system)
Use cases	Large data, dynamically size data, data that needs to persist	Small, short-lived data (e.g., local variables)

When do we need malloc to allocate memory?

`malloc` is a function for allocating memory on the heap. `malloc` is needed in the following scenarios:

- When you need memory beyond the lifetime of a function
- When you don’t know the size of data at compile time
- When you need to allocate large data set
- …

Need a local variable beyond the lifetime

```c
int* createArray(int size) {
    // Allocate memory on the stack
    int arr[size];
    //initialize arr
    for(int i=0; i<size; i++){
        arr[i]=i;
    }
    //arr won't be available after
    //the execution of the function
    return arr;
}

int main(){
    int *array;
    int size=5;
    array = createArray(size);
    for(int i=0; i<size; i++){
        printf("%3d", array[i]);
    }
}
```

Wrong!

```c
int* createArray(int size) {
    // Allocate memory on the heap
    int* arr = (int*)malloc(size * sizeof(int));
    // This pointer can be used even after the function returns
    return arr;
}
```

Correct!

Local variables are (automatically) allocated on the stack, which will be deallocated after the execution of the function.

- If the function returns a value, the value will be copied to the outside of the function.
- But if the function returns a pointer (i.e., memory address), it results in problems. (Actually, warnings will be raised but the code can still be compiled!!!)

The size of data is only known at runtime

```
32  int main(){
33      int n;
34      printf("Enter the number of elements: ");
35      scanf("%d", &n);
36      // Size is determined at runtime
37      int* arr = (int*)malloc(n * sizeof(int));
38      //Note: int arr[n] also works in this scenario
39      //int arr[n]
40  }
```

If the size of the date can only be determined during the runtime, you can use malloc to allocate the memory.

Allocating very large dataset

```
42  int large_array(int size){
43      int array[size];
44      return 0;
45  }
46
47  int main(){
48      int size=100000000;
49      large_array(size);
50      return 0;
51  }
```

Allocating a large array that might exceed stack limits.

- The stack has a limit size (typically 1-8 MB)
- In the example, the function large_array wants to allocate $1x10^9$ int type elements, which is

  $1x10^8x4$ bytes = $(4 \times 10^8)/(1024*1024)$ = 381.47 MB

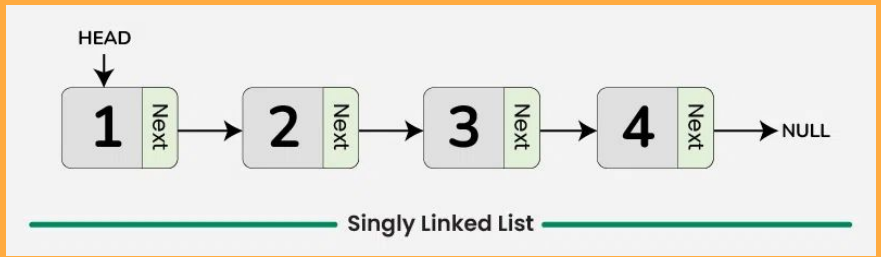
  far more than 8 MB.

Example: construct a linked list

A linked list is a sequence of nodes where each node contains two parts:

- **Data**: The value stored in the node.
- **Pointer**: A reference to the next node in the sequence.

Unlike arrays, linked lists **do not** store elements in **contiguous** memory locations. Instead, each node points to the next, forming a chain-like structure and to access any element (node), we need to first sequentially traverse all the nodes before it.




```
4   struct Node {
5       int data;
6       struct Node* next;
7   };
```

We can define a structure `Node` to represent the element in the linked list.

Also, list operations are needed:

- `create_node`
- `append`
- `insert`
- `delete`

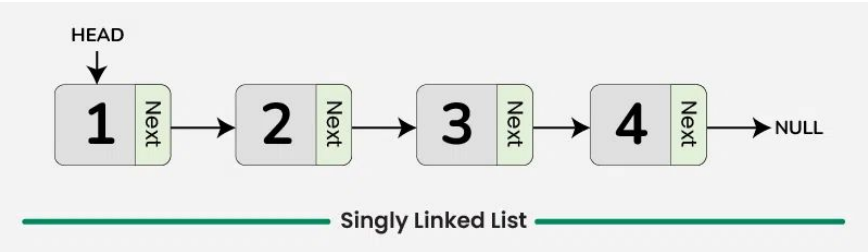
Example: linked list

- In `create_node`, the `malloc` allocates the memory for `newNode` in Heap, so, the `newNode` will persist after the execution of `creat_node`.
- In `append`, it takes the address of the `list` as a parameter, not the address stored in `list` because the `list` is initialized as `NULL`, and we want to change the value stored in `list`, not the value stored in `NULL` when appending the first node to the list.

Example: the "insert" of a linked list

```
34  void insert(struct Node ** list, int data, int index){
35      struct Node* insertedNode = create_node(data);
36      if (*list == NULL) { *list = insertedNode; return;}
37      struct Node* previousNode = *list;
38      for(int i=0; i<index-1; i++){
39          previousNode = previousNode->next;
40      }
41      struct Node* currentNode = previousNode->next;
42      insertedNode->next = currentNode;
43      previousNode->next = insertedNode;
44      return;
45  }
```

Incompleted!



To insert a node in a list at the specified position, there are three steps;

1. find the node in front of the specified position;
2. replace the next pointer with the inserted node; (connecting the previous node with the inserted node)
3. Set the next node of the inserted node to be the current node at the specified position.

Note: Similar to `append`, `insert` takes the address of `list` as a parameter, so, it can handle the case when `list` is empty. In the example, a for loop is used to get the node in front of the position. But the code is incompleted, some corner cases are missed! Can you tell what they are?

Example: the “insert” of a linked list

```
34  void insert(struct Node ** list, int data, int index){
35      struct Node* insertedNode = create_node(data);
36      int len = length(*list);
37      // index = index<(len-1)?index:len-1;
38      if (*list == NULL) { *list = insertedNode; return;}
39      struct Node* previousNode = *list;
40      if(index==0){
41          insertedNode->next = previousNode;
42          *list = insertedNode;
43          return;
44      }
45      if(index>1 && index<len){
46          for(int i=0; i<index-1; i++){
47              previousNode = previousNode->next;
48          }
49          struct Node* currentNode = previousNode->next;
50          insertedNode->next = currentNode;
51          previousNode->next = insertedNode;
52          return;
53      } else {
54          struct Node* node = *list;
55          while(previousNode->next!=NULL){ previousNode = previousNode->next;}
56          previousNode->next = insertedNode;
57      }
58  }
```

Corner cases:

- The list is empty (`NULL`)
- When the position is at the beginning of the list (i.e., `index=0`)
- When the position is at the end of the list (i.e., `index` > length of list -1)
- Any other else?

Example: delete a node at a specified position

To delete a node in a list at the specified position, it needs two steps;

1. find the node (`previousNode`) in front of the specified position;
2. replace the next pointer of the `previousNode` with the next node of the node being deleted; (disconnecting)
3. deallocate the node to be deleted (`free`)

```
72  void delete(struct Node **list, int index){
73      if (*list == NULL) printf("the list is empty\n");
74      struct Node* previousNode = *list;
75      for(int i=0; i<index-1; i++){
76          previousNode = previousNode->next;
77      }
78      struct Node* currentNode = previousNode->next;
79      previousNode->next = currentNode->next;
80      free(currentNode);
81  }
```

Incompleted!

Note: In the example, delete takes the address of `list` (i.e., `**list`) as a parameter. But it is also fine to if we set the parameter to be `*list`, i.e., `void delete(struct Node *list, int index)` One reason to use `**list` is for the consistency of the format of methods.

Example: delete a node at a specified position

```
83  void delete(struct Node **list, int index){
84      if (*list == NULL) {printf("The list is empty!\n"); return;}
85      int len = length(*list);
86      if (index>len-1) {printf("The index is out of range!\n"); return;}
87      struct Node* previousNode = *list;
88      if (index==0){
89          *list = previousNode->next;
90          free(previousNode);
91          return;
92      }
93      if(index>1 && index<len){
94          for(int i=0; i<index-1; i++){
95              previousNode = previousNode->next;
96          }
97      }
98      struct Node* currentNode = previousNode->next;
99      previousNode->next = currentNode->next;
100     free(currentNode);
101 }
```

Corner cases:

- The list is empty (`NULL`)
- When the position is at the beginning of the list (i.e., `index=0`)
- When the position is at out of the list (i.e., `index < 0` or `index >` length of list -1)

Exercise 1

Write a function which takes as argument a character string s and returns either:

- the string consisting of the first word of s when s is not empty
- the empty string otherwise.

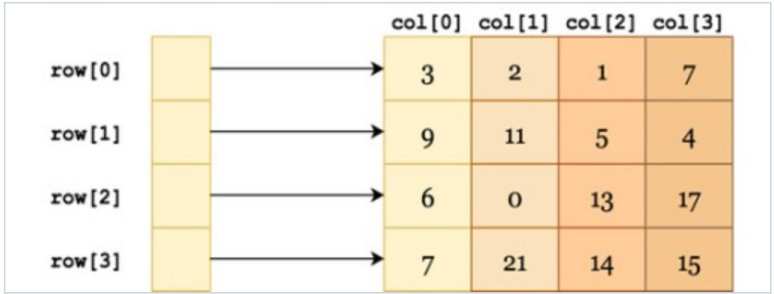
```c
int main(){
    char string[] = "   hello world";
    char * firstWord;

    firstWord = first_word(string);
    //todo:
    //define a function first_word;
    //it takes a string as argument;
    //it returns the first word of the string;
    //if the string is empty, return an empty string.
    printf("%s\n", firstWord);
}
```

Hints:

- The number of the characters in the first word is unknown to the function;
- To get the first word, one needs to find the start position and the length of the word.
- There can be spaces between words.

Exercise 2: matrix

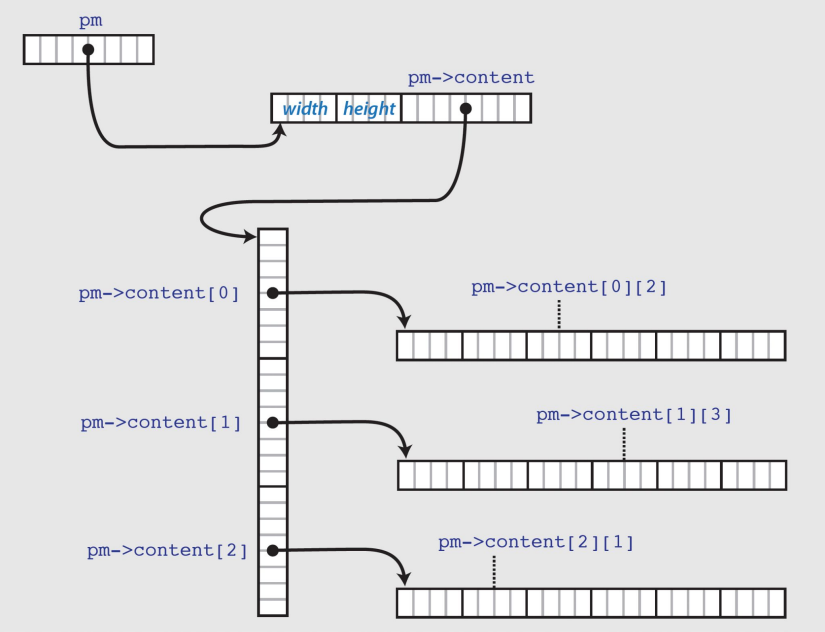
- A matrix is a collection of numbers arranged into a fixed number of rows and columns.
- We can define a structure to represent a matrix (i.e., a 2-d array).




```
5   struct matrix {
6       int width;  //number of columns
7       int height; //number of rows
8       int **content; //array of the address
9   };
```

- The field `content` is the address of an array of address (an array of pointers to `int`)
- The type of the fields is “pointer to pointer to `int`” or `int **`
- Each element of this field is the address of an array containing one of the rows of the matrix.

Exercise 2: matrix




```
11  struct matrix *allocate_matrix(int width, int height){
12      struct matrix * pm;
13      //allocate the structure
14      pm = malloc(sizeof(struct matrix));
15      pm->width = width;
16      pm->height = height;
17      //allocate the array of row address
18      pm->content = malloc(height*sizeof(int * ));
19      //allocate the rows
20      for(int i=0; i<height; i++){
21          pm->content[i] = malloc(width*sizeof(int));
22      }
23      return pm; //return the address of the addressed matrix
24  }
```

`pm->content`: the address of the matrix
`pm->content[i]`: the address of the `i-th` row
`pm->content[i][j]`: the content of the `(i, j)` position

Exercise 2: matrix

Complete the `clone_matrix`:

- It returns a pointer of a matrix structure.
- It copies the structure and the data to the new matrix.

```c
struct matrix *clone_matrix(struct matrix *pm){
    struct matrix *pmc;
    //Todo:
    //allocate the structure of the clone
    //and its array of addresses
    //allocate the rows, copy the rows of the original
    return pmc;
}
int main(){
    struct matrix * myMatrix;
    int columns=4, rows=3;
    myMatrix = allocate_matrix(columns, rows);
    //initialization
    for(int i=0; i<rows; i++){
        for(int j=0; j<columns; j++){
            myMatrix->content[i][j]=i*columns+j;
        }
    }
    struct matrix * myMatrixClone;
    myMatrixClone = clone_matrix(myMatrix);
    for(int i=0; i<rows; i++){
        for(int j=0; j<columns; j++){
            printf("%3d", myMatrixClone->content[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Remark

A similar treatment of binary trees of integers is possible:

```
struct node {
int key;
struct node *left;
struct node *right;
};
```

A tree is a pointer to type node

- the `NULL` pointer when the tree is empty
- a pointer to the root node of the tree when the tree is nonempty

The functions on the lists are naturally defined by recursion: recursion becomes somewhat necessary to manipulate trees.