Pointer and Memory (2)

Song Liu (song.liu@bristol.ac.uk)

GA 18, Fry Building,

Microsoft Teams (search "song liu").

Revision: Stack Memory

- When the function is being executed on the CPU, its data (such as variables declared in the function) are temporarily stored in the memory.
- The memory region for storing function data is called "stack".
- When a function is called, **its data is added to the top of the stack**, so you program can access them.
- When a function finishes its execution, **its data is removed from the stack and the space it occupies is freed** for future calls of functions.

Ice Cream

Stack Memory: Pros & Cons

Stack is **a highly efficient memory allocation/release mechanism**.

- You do not need to free up stack memory used by your program.
  - Operating system (OS) cleans it up after you.
- However, the amount of stack memory must be determined at the compilation time!
  - OS must know how much stack memory your program uses before your program runs!
  - Allocating memory at compilation time is called **static memory allocation**.

Example: Array in Stack Memory

- `array` declared in a function is stored in the stack memory.

```c
#include<stdio.h>

void main(){
    int array[] = {1, 2, 3, 4};
    // do some operations on array
}
```

- Our code tells the complier: " `array` contains 4 elements. Allocate `4*sizeof(int)` bytes in the **stack memory** for the `array` variable. "
- `sizeof(type)` operator gives number of bytes for `type`.
    - `sizeof(int)` is 4. `sizeof(double)` is 8.

Problem: Dynamic Array Allocation

- However, what if we do not know how big the array is when writing the code?

```
#include<stdio.h>
void main(){
    // int array[????];
    // I do not know hoa big the array is.
}
```

- For example, `array` records customers' ratings of my store. I do not know how many customers I will have at the programming stage.
- The compiler cannot allocate stack memory for us if we do not know the size of our array when writing the code*.
- Memory allocation at **runtime** is called **Dynamic Memory Allocation**.

Dynamic Memory Allocation

- We need a mechanism to **dynamically allocate memory spaces** for variables whose sizes cannot determined before compilation.
- In C programming language, variables that requires dynamic memory allocation are stored in the **heap memory**.
    - Heap memory is a part of the **virtual memory**.

Heap Memory: Pros & Cons

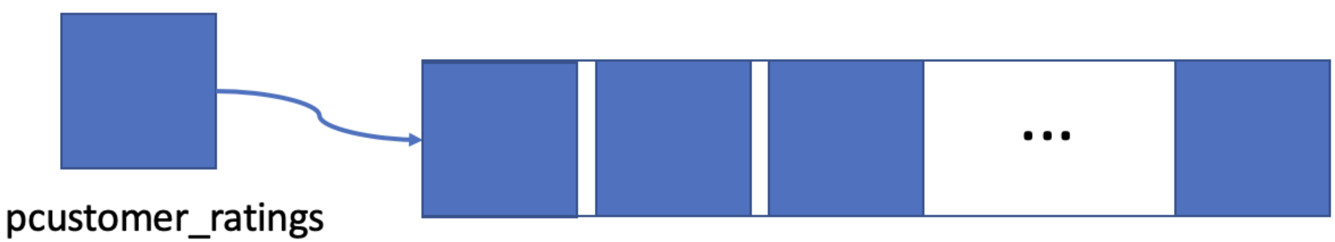
- Your program can allocate heap memory to store variables while it is running.
  - The size of the allocated memory does **not** have to be known before compilation.
- However, you have to **manually allocate and free heap memory**.
  - Allocate Heap Memory, `malloc`.
  - Free Heap Memory, `free`.
  - They are provided in the **header file** `stdlib.h`.

Customer Rating Example

```c
#include<stdio.h>
#include<stdlib.h> //must have!

void main(){
    printf("How many customers do we have today?\n");
    // read from keyboard,
    int num_customers;
    scanf("%d", &num_customers);
    // allocate heap memory for an array
    // depending on user's input
    int *pcustomer_ratings =
            malloc(num_customers * sizeof(int) );

    // do something with the new array
    // e.g., initialize the array with ratings
    // then calculate the average score.

    //release the heap memory
    free(pcustomer_ratings);
}
```

Dissecting Customer Rating Example

- Usage: `ptr_to_memory = malloc(size_of_memory)`
  - The argument of `malloc` function is the number of bytes heap memory desired.
  - `malloc` allocated `num_customers * sizeof(int)` bytes of **contiguous heap memory**.
- `malloc` function returns **a pointer** points to the starting address of the allocated memory.
  - This address is stored in `pcustomer_ratings`.



Dissecting Customer Rating Example

```
int *pcustomer_ratings =
    malloc(num_customers * sizeof(int) );
```

- After this statement, `pcustomer_ratings` can be used **as if** it is an `int` array with `num_customers` elements.
  - `pcustomer_ratings[2]` is the 3rd elem. in the array.
  - Recall, the pointer points to the first elem. of the array <=> array name.
- After being created, this array contains **garbage values** (the array has not yet been initialized!).
  - Same as you create an array in stack memory.

Dissecting Customer Rating Example

- Before our program finishes, we use `free` function releases the heap memory that were allocated to us.
- Usage: `free(pointer_to_memory)`
- **We are responsible to release all heap memory which were allocated to our program!!**
  - If we keep allocating heap memory but do not release them, our program will slowly but gradually exhaust all available memory in the system, causing performance degradation over time.
  - This kind of resource mismanagement is referred to as memory leak.

Memory Leak

- Memory leak will negatively impact user's experience and is a problem very difficult to trace.
- Therefore, programmers should be very careful when allocating heap memory and always use `malloc` and `free` **in pairs**.
- In your final project, we will reduce 5% for each unpaired `malloc` and `free`.

Stack vs. Heap Memory

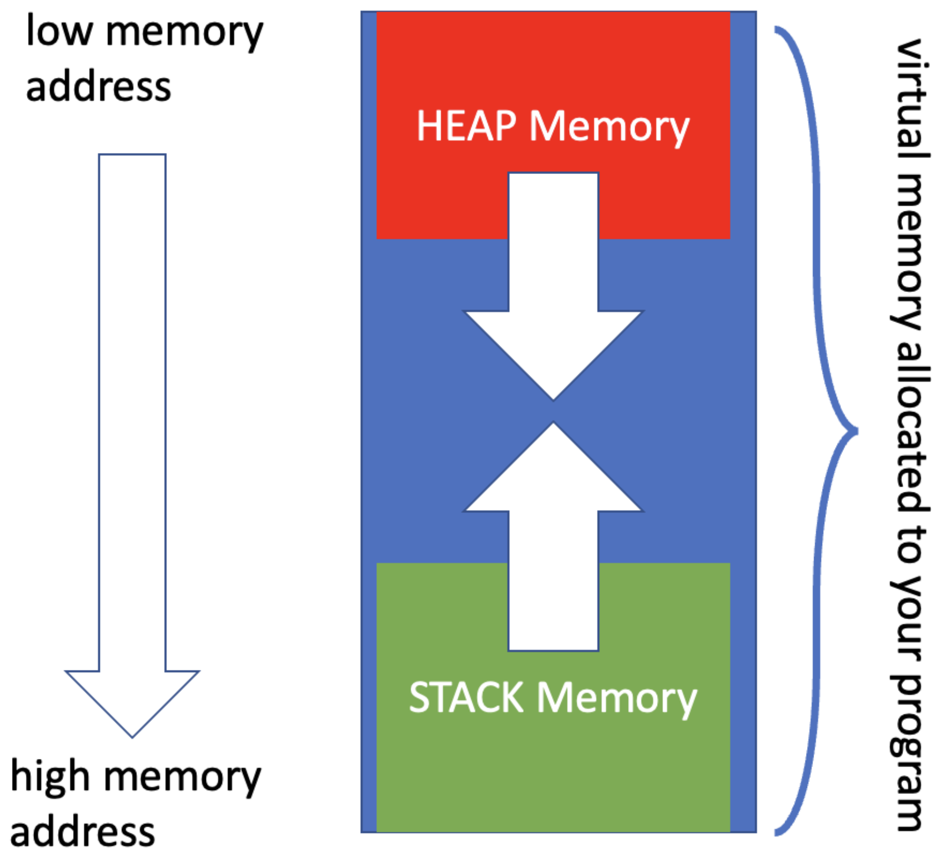
- Heap and Stack Memory are both parts of virtual memory, but they differ in **allocation**, **management**.
- Static vs. Dynamic Allocation
  - Stack Memory stores variables declared in functions whose sizes are already known **at compilation**.
  - Heap Memory stores variables whose sizes are determined **at the runtime**.

Heap vs. Stack Memory

- Automated vs. Manual Management
  - OS manages stack memory for us. We do not need to allocate and free stack memory.
  - We need to manually allocate/release the heap memory for each variable.

Layout of Virtual Memory

Heap and Stack memory occupies different segments of your virtual memory and grows toward different directions.



- Heap memory grows toward bigger memory addresses
- Stack memory grows toward smaller memory addresses

Allocate and Clear Heap Memory using `calloc`

- When allocating heap memory using `malloc`, we get an array that contains garbage values.
  - **C never initializes variables for us**!
  - What if we want to initialize memory with zeros as soon as it is allocated?
- Replace

```
int *pcustomer_ratings =
    malloc(num_customers * sizeof(int) );
```

  with

```
int *pcustomer_ratings =
    calloc(num_customers, sizeof(int) );
```

  - Usage: `ptr = calloc(num_elem, size_of_each)`

Allocate and Clear Heap Memory using `calloc`

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

void main(){
    printf("How many customers do we have today?\n");
    int num_customers;
    scanf("%d", &num_customers);
    // allocate and CLEAR heap memory
    int *pcustomer_ratings =
            calloc(num_customers, sizeof(int) );

    for (int i = 0; i < num_customers; i++)
    {
        printf("%d ", pcustomer_ratings[i]);
    }
    // prints out 0 0 0 0 0 0 0 ...
    free(pcustomer_ratings);
}
```

Reallocating Heap Memory using `realloc`

- What if you need to **resize** your array?
- You can do:
  - allocate a new array with the new size
  - copy from the old array to the new array.
  - free the heap memory occupied by the old array
- `realloc` does these things for you **automatically**!
- Usage: `ptr_to_new = realloc(ptr_to_old, new_size)`

Reallocating Heap Memory using `realloc`

Below we expand a 10-element array to a 20-element array.

```
#include<stdio.h>
#include<stdlib.h>
void main(){
    // we start with a 10-element array.
    int *array =  malloc(10*sizeof(int));
    // do something with array...
    // Expand it to a 20-element array.
    array =  realloc(array, 20*sizeof(int));
    // free the heap memory of the NEW array!
    free(array);
}
```

- `array` after the second statement points to the new array!!

Reallocating Heap Memory using `realloc`

- However, using `realloc` to grow an array may not be the most efficient thing to do:
  - `realloc` copies from the old array to the new array.
  - If the old array is big, this cost is not negligible.
- We will introduce a different solution to this "growing array" problem in the future.

Case Study: Customer Rating 2.0

- Imagine a program taking customer's rating in real time.
- Customers provide their ratings one at time.
- Our program writes customers ratings into an array.
- The manager can enter a secret code "1234", the program displays today's average rating, exits.

Case Study: Customer Rating 2.0

Problem: We **never** know how many customers we will encounter today.
Solution: We **dynamically** allocate a small array at the beginning, "grow" it using `realloc` as we encounter more and more customers.

(High-level) Pseudo Code

1. Creating `array` with length `len` in heap memory.
2. Initialize `count = 0`.
3. Repeat:
    - Take customer's rating R.
    - If `R == secret_code`
        - `break`;
    - If `count == len`
        - use `realloc` to expand `array` to `len + 10`
        - `len = len + 10`
    - `array[count] = R`;
    - add `count` by 1;
4. Compute and display average of `array`.
5. Free `array`.

Customer Rating 2.0

```c
#include<stdio.h>
#include<stdlib.h>
void main(){
    int len = 10, count = 0;
    //start with some provisional heap memory
    int *pratings =  malloc(len*sizeof(int));
    while(1){//loop forever until reach "break" statement.
        printf("How do you feel about our service?\n");
        printf("input rating 0-5, type secret code to quit.\n");
        int rating=0; scanf("%d", &rating);

        if(rating == 1234){break;} //end loop
        // expand the array if we have reached the max capacity
        if(count == len){
            pratings = realloc(pratings, (len + 10)*sizeof(int));
            len = len + 10;
        }
        pratings[count] = rating;
        count++;
    }
    //... compute average ratings and display
    free(pratings);
}
```

Conclusion

1. You can allocate heap memory **dynamically** when the program is running.
2. `malloc` can allocate heap memory.
    - You can access the allocated memory as if it is an array.
    - You must `free` the allocated memory after using it.
3. Use `calloc` to allocate and clear the memory. Use `realloc` to resize the allocated memory.

Lab 1

1. Download the files and place them in the labpack.
2. Read `rating.c`
    i. familiar with the usage of `malloc` , `free` .
    ii. Change the `malloc` function used in this file with `calloc` , and check if the newly allocated array has been initialized to zero.
3. Read `expand.c`
    i. familiar with the usage of `realloc` .

Lab 2 (submit)

1. Open `image2d.c` and follow instructions on the following slides.

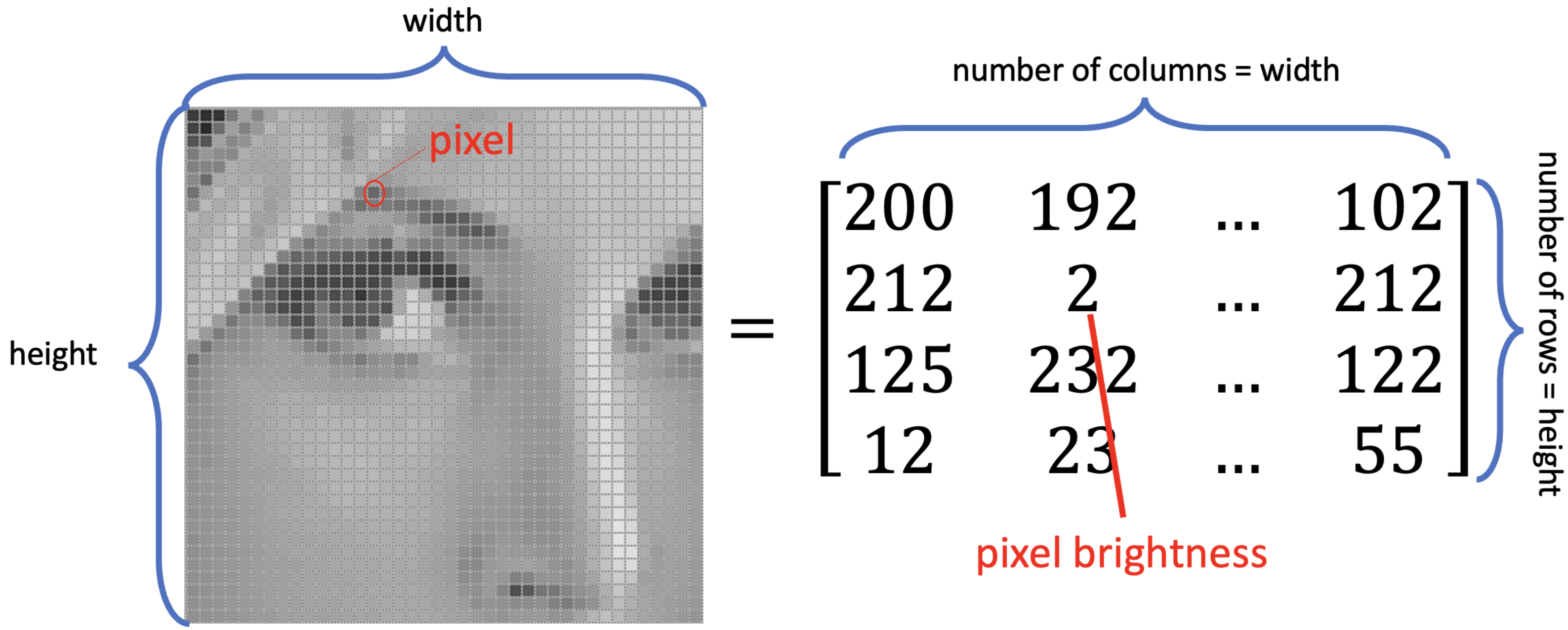
How Computer Stores/Displays Images?

Song Liu (song.liu@bristol.ac.uk)

GA 18, Fry Building,

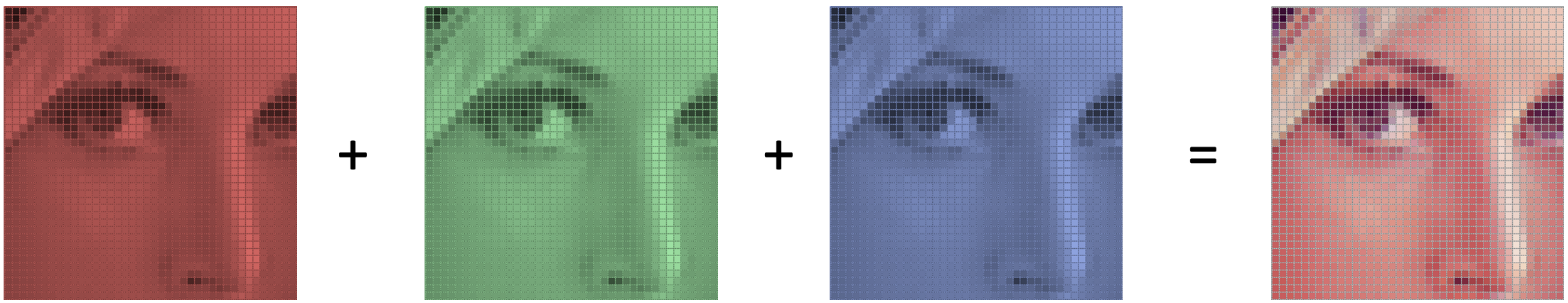
Microsoft Teams (search "song liu").

Images are Matrices



- Grayscale images are expressed as matrices in computer.
- A pixel in the image corresponds to an element in the matrix.
- Each element of the matrix indicate the brightness of a pixel. There are usually 256 levels of brightness, 0 is darkest while 255 is brightest.

Colored Images are Matrices too



- One colored image is expressed as **three** individual matrices:
  - Three matrices indicate brightness in Red, Green and Blue tones (RGB).
  - Computer can display a colored image by stacking three images together.

Images Files are Flattened Matrices

- **Image files** usually store images as flattened matrices.
    - Many file system (such as tape) can only support sequential read/write.
    - Recall, a row-major flattend matrix is

$$\begin{bmatrix} 1, & 2 \\ 3, & 4 \end{bmatrix} \implies [1, 2, 3, 4].$$

    - Check Lecture 6, Task 3 for more details.
- Knowing these facts, we can build a "textual image viewer" using C programming language.

Building an Image Viewer

- Suppose you have obtained an `int` array `a` with length `M*N`.
  - It contains a flattened matrix `[12, 232, ..., 254]`
  - Let the "unflattend" matrix be $A \in \mathbb{N}^{m \times n}$.
  - Matrix A represents an image with width `N` and height `M`.
- To visualize your image, simply print out an M by N matrix replacing the integer $A_{i,j}$ with a single character according to the following rules.
  - if $A_{i,j} \leq 85$, print empty space ` `.
  - if $85 < A_{i,j} \leq 170$, print character `I`.
  - if $170 < A_{i,j} \leq 255$, print character `M`.

Building an Image Viewer

Now, modify code in `image2d.c` so that it prints out the image stored in `mysterious.dat`.

Hint: If you cannot see the image, try to zoom out by pressing ctrl + -