Pointer and Memory (2)

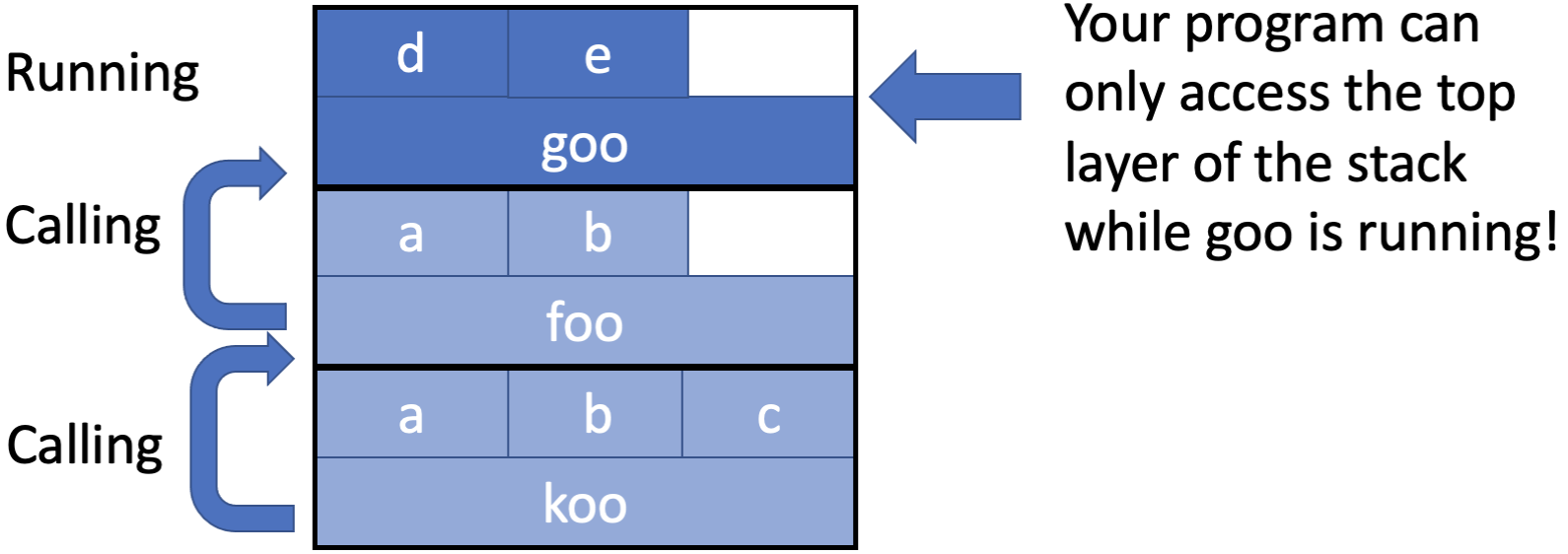
Song Liu (song.liu@bristol.ac.uk)
GA 18, Fry Building,
Microsoft Teams (search "song liu").

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 in the current program is called "stack".
- When a function is called, **its data is added to the top of the stack**. 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.

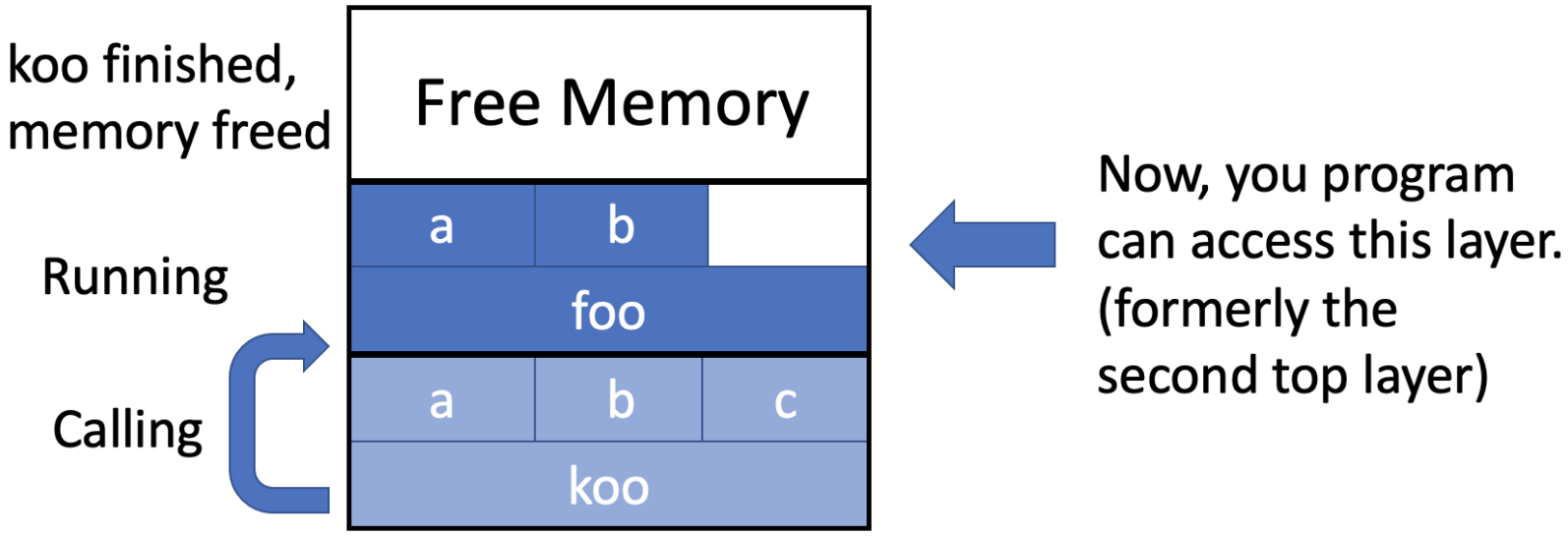
Revision: Stack Memory Allocation

Consider a situation where function `koo` calls `foo` and `foo` calls `goo`. Below is the stack while `goo` is running.



Revision: Stack Memory Allocation

When `goo` finishes running, its memory is freed.



When `foo` finishes running, its memory will be freed too and only variables in `koo` will be accessible.

Revision: Stack Memory Allocation

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

- The memory allocation and release are **all automatically handled by the OS**.
- However, there is only **a limited stack space for each program** (determined by the OS). If a single function occupies a large memory space, or the call stack gets too "tall", we may run out of stack memory and an execution error will be raised by the OS.

Example: Array in Stack Memory

```
#include<stdio.h>

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

- In this example, `array` is stored in the stack memory.
- Our code tells the complier: " `array` contains 4 elements. Therefore, allocate `4*sizeof(int)` byes in the **stack memory** for the `array` variable. "

Example: Array in Stack Memory

```c
#include<stdio.h>
void main(){
    // int array[??????];
    // I do not know how big my array is,
    // when writing the code.
}
```

- However, what if we do not know how big the array is when writing the code?
    - For example, `array` records customers' ratings of my store. I do not know how many customers I will have each day 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*.
    - * This situation has been changed in recent years.

Dynamic Memory Allocation

- If we cannot determine how big the array is before compilation, how do we allocate the memory for the array?
- 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

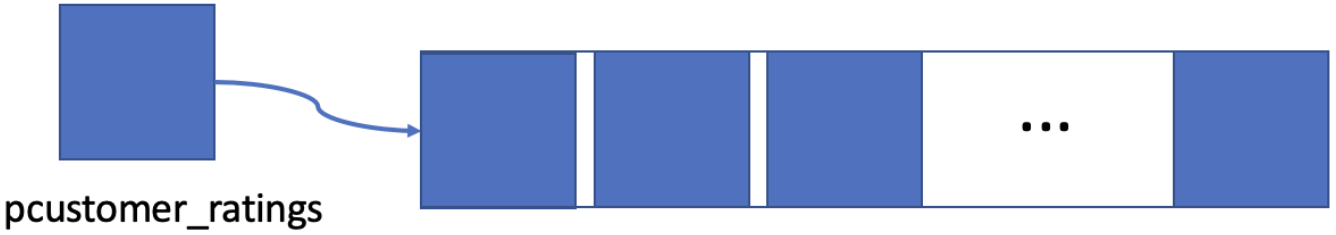
- Heap memory is a part of the **virtual memory**.
- Your program can allocate heap memory to store a variable while it is running.
    - The size of the allocated memory does **not** have to be known before compilation.
- You need to use pointers to access the heap memory.

Customer Rating Example

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

void main(){
    printf("How many customers do we have today?\n");
    // read from keyboard, will talk about later
    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.
  - In the example, `malloc` allocated `num_customers * sizeof(int)` bytes of 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.
  - See the previous lecture.
- For now, this array contains garbage values (the array has not yet been initialized!).

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 we allocated!!**
    - 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`.

Heap vs. Stack Memory

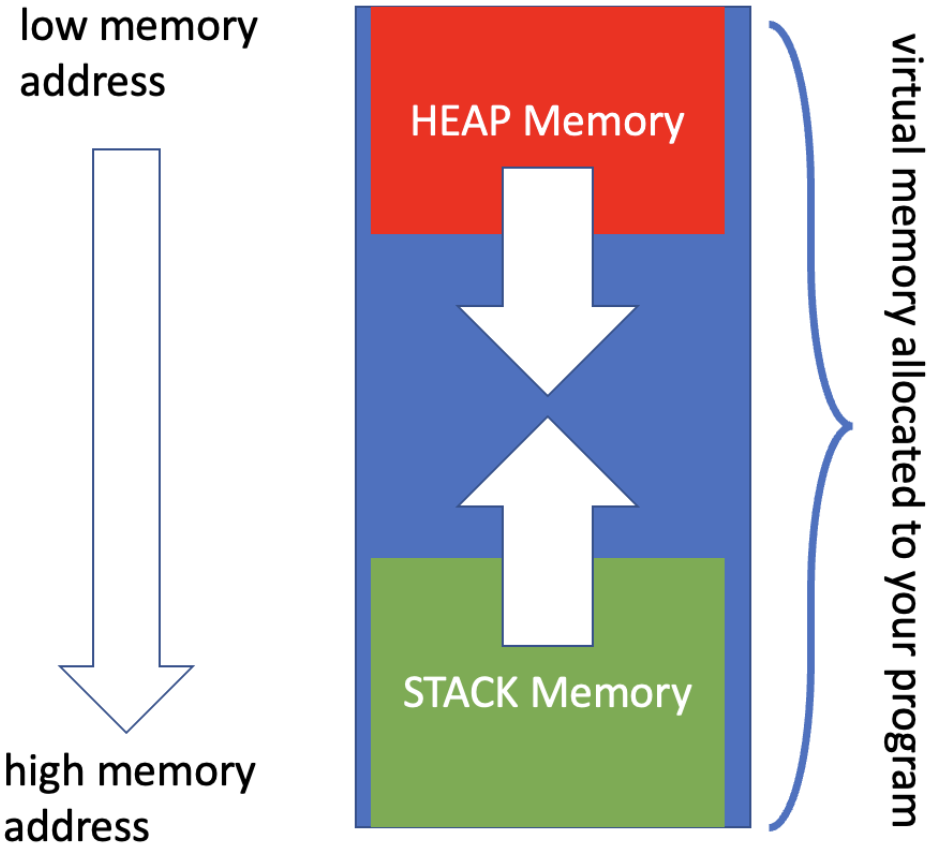
- Heap and Stack Memory are both parts of virtual memory, but they differ in **allocation**, **management** and **variable visibility**.
- Static vs. Dynamic Allocation
    - Stack Memory stores variables declared in functions whose sizes are already known **before compilation**.
    - Heap Memory stores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

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) );
```

Notice `*` is changed to `,`!

Allocate and Clear Heap Memory using `calloc`

```c
#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));
    // 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.
- To avoid copying array, OS may "attach" a new block of virtual memory at the end of the old array, as if our old array has "grown" in size.
- 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.
- At the end of the day, the manager 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 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`.
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 using a pointer.
    - 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.

Homework 1

0. Compile and run the customer rating 2.0.
1. Add appropriate code to the customer rating 2.0, so before quitting:
    i. It displays "Today's average customer rating is X.YZ", where X.YZ is the average rating. Print two digits after the decimal point.
2. Add appropriate code to the customer rating 2.0, so it checks for illegal input each time user types a number:
    i. Rating must be between 0-5.
    ii. If an illegal input is detected, ignore this rating and get ready for the next rating input.

Homework 2: Appending (submit)

1. If you are familiar with Python arrays, you may be jealous about its ability to "append to an array". For example:

```
>>> a = ['harry', 'hermione', 'ron']
>>> a.append('dumbledore')
['harry', 'hermione', 'ron', 'dumbledore']
```

2. Write a function

   ```
   int *append(int *array, int len, int a)
   ```

   Takes input `array`, its length `len` and an integer `a`. `append` returns a copy of `array` containing all elements in the original `array` with the additional `a` at the end.

4. Example:

```
int array[] = {1,2,3,4,5,6,7};
int len = 7;
int *newarray = append(array,len,8);
// newarray contains 1 2 3 4 5 6 7 8
```

5. Use the skeleton code provided to you.
6. Think before you do: when returning an array, should I return an array pointing to a **stack** variable or a **heap** variable?
7. Write pseudo code and discuss with your group members.

Homework 3: Slicing (submit)

1. If you are familiar with Python arrays, you may be jealous about its ability to "slice an array". For example:

```
>>> a = ['harry', 'hermione', 'ron', 'dumbledore', 'voldemort']
>>> a[1:3]
['hermione', 'ron']
```

2. Let us implement it using C, which will provide you with some insights about how the slicing is implemented.
3. Write a function

   ```
   int* slice(int *array, int start, int end)
   ```

   Takes inputs `array`, `start` and `end` and return a copy of sliced `array`, including elements only from `array[start]` to `array[end-1]`.

4. Example:

```
int array[] = {1,2,3,4};
newarray = slice(array, 1, 3);
//new array contains elements 2,3.
```

5. Use the skeleton code provided to you.
6. Think before you do: when returning an array, should I return an array pointing to a **stack** variable or a **heap** variable?
7. Write pseudo code and discuss with your group members.

Homework 4: Image Viewer (submit)

Finish the image viewer task from last week's tutorial.