CS107, Lecture 11
The Heap, Continued

Reading: K&R 5.6-5.9 or Essential C section 6 on the heap

Poll Check-in

How are you doing / how is everything going so far? Feel free to respond however you'd like!

Respond on PollEv:
pollev.com/cs107
or text CS107 to 22333
once to join.

CS107 Topic 3

How can we effectively manage all types of memory in our programs?

Why is answering this question important?

- Shows us how we can pass around data efficiently with pointers (last time)
- Introduces us to the heap and allocating memory that we manually manage (today)
- Helps us better understand use-after-free vulnerabilities, a common bug (today)

assign3: implement a function using resizable arrays to read lines of any length from a file and write 2 programs using that function to print the last N lines of a file and print just the unique lines of a file. These programs emulate the **tail** and **uniq** Unix commands!

Learning Goals

- Learn about the differences between the stack and the heap and when to use each one
- Become familiar with the **malloc**, **calloc**, **realloc** and **free** functions for managing memory on the heap
- Understand use-after-free vulnerabilities and vulnerability disclosure

Lecture Plan

- **Recap:** The Heap So Far
- Freeing Memory
- **Practice:** Pig Latin + Valgrind
- **realloc**

```
cp -r /afs/ir/class/cs107/lecture-code/lect11 .
```

Lecture Plan

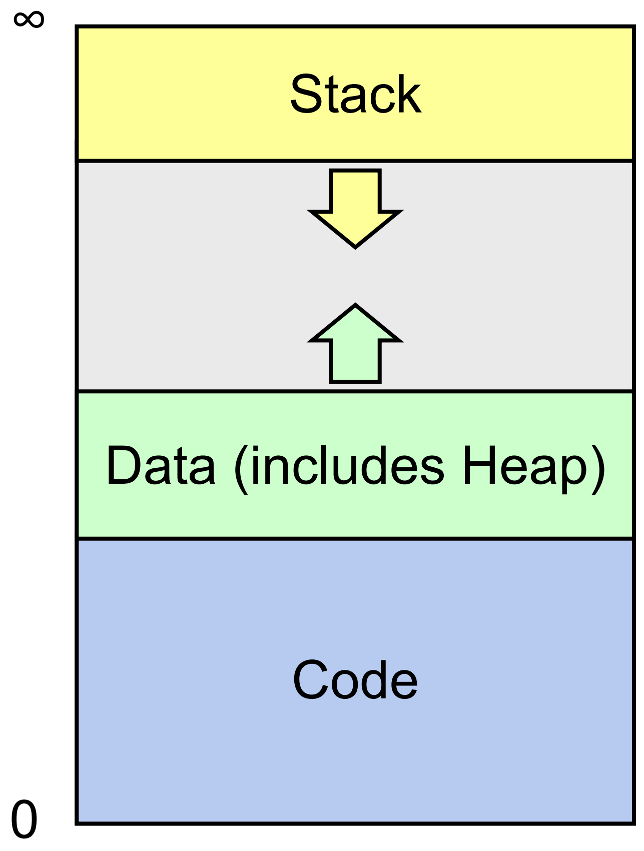
- **Recap: The Heap So Far**
- Freeing Memory
- **Practice:** Pig Latin + Valgrind
- **realloc**

```
cp -r /afs/ir/class/cs107/lecture-code/lect11 .
```

Memory Layout

- The **heap** is a part of memory that you can manage yourself.
- The **heap** is a part of memory below the stack that you can manage yourself. Unlike the stack, the memory only goes away when you delete it yourself.
- Unlike the stack, the heap grows **upwards** as more memory is allocated.

The heap is **dynamic memory** – memory that can be allocated, resized, and freed during **program runtime**.



Working with the heap

Working with the heap consists of 3 core steps:

1. Allocate memory with malloc/realloc/strdup/calloc
2. Assert heap pointer is not NULL
3. Free when done (more today!)

The heap is **dynamic memory**, so you may encounter many **runtime errors**, even if your code compiles!

`malloc`

```
void *malloc(size_t size);
```

To allocate memory on the heap, use the **`malloc`** function (“memory allocate”) and specify the number of bytes you’d like.

- This function returns a pointer to *the* ***starting address*** *of the new memory*. It doesn’t know or care whether it will be used as an array, a single block of memory, etc.
- **`void *`**means a pointer to generic memory. You can set another pointer equal to it without any casting.
- The memory is *not* cleared out before being allocated to you!
- If `malloc` returns `NULL`, then there wasn’t enough memory for this request.

Always assert with the heap

Let's write a function that returns an array of the first **len** multiples of **mult**.

```
int *array_of_multiples(int mult, int len) {
    int *arr = malloc(sizeof(int) * len);
    assert(arr != NULL);
    for (int i = 0; i < len; i++) {
        arr[i] = mult * (i + 1);
    }
    return arr;
}
```

- If an allocation error occurs (e.g. out of heap memory!), malloc will return NULL. This is an important case to check **for robustness**.
- **assert** will crash the program if the provided condition is false. A memory allocation error is significant, and we should terminate the program.

calloc and strdup

```
void *calloc(size_t nmemb, size_t size);
```

`calloc` is like `malloc` that **zeros out** the memory for you. It takes two parameters, which are multiplied to calculate the number of bytes (`nmemb * size`).

However, we want to default to **malloc** and use **calloc** only when we need to zero out the space.

```
char *strdup(char *s);
```

`strdup` is a convenience function that returns a **null-terminated**, heap-allocated string with the provided text, instead of you having to **`malloc`** and copy in the string yourself. (like **malloc** + **strcpy**).

Lecture Plan

- **Recap:** The Heap So Far
- **Freeing Memory**
- **Practice:** Pig Latin + Valgrind
- **realloc**

```
cp -r /afs/ir/class/cs107/lecture-code/lect11 .
```

Cleaning Up with `free`

```
void free(void *ptr);
```

- If we allocated memory on the heap and no longer need it, it is our responsibility to **free** it.
- To do this, use the **`free`** command and pass in the *starting address on the heap for the memory you no longer need.*
- Example:

```
char *bytes = malloc(4);
…
free(bytes);
```

Free

```
void free(void *ptr);
```

When you free an allocation, you are freeing up what it *points* to. You are not freeing the pointer itself. You can still use the pointer to point to something else.

```
char *str = strdup("hello");
...
free(str);
str = strdup("hi");
```

free details

Even if you have multiple pointers to the same block of memory, each memory block should only be freed **once**.

```
char *bytes = malloc(4);
char *ptr = bytes;
…
free(bytes);   ⇦ ✅
…
free(ptr);     ⇦ ❌ Memory at this address was already freed!
```

You must free the address you received in the previous allocation call; you cannot free just part of a previous allocation.

```
char *bytes = malloc(4);
char *ptr   = malloc(10);
…
free(bytes);        ⇦ ✅
…
free(ptr + 1);      ⇦ ❌
```

Cleaning Up

You may need to free memory allocated by other functions if that function expects the caller to handle memory cleanup.

```
char *str = strdup("Hello!");
…
free(str);    // our responsibility to free!
```

Cleaning Up

You may need to free memory allocated by other functions if that function expects the caller to handle memory cleanup.

```
char *create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char) * (num + 1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;  // caller must free
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);
    free(str);
    return 0;
}
```

Memory Leaks

A **memory leak** is when you do not free memory you previously allocated.

```
char *str = strdup("hello");
str = strdup("hi"); // memory leak!  Lost previous str
```

Memory Leaks

A **memory leak** is when you do not free memory you previously allocated.

- Your program should be responsible for cleaning up any memory it allocates but no longer needs.
- If you never free any memory and allocate an extremely large amount, you may run out of memory in the heap!
- However, memory leaks rarely (if ever) cause crashes.
- We recommend not to worry about freeing memory until your program is written. Then, go back and free memory as appropriate.
- Valgrind is a very helpful tool for finding memory leaks!
- Tip: free as soon as you are done with a block of memory

Lecture Plan

- **Recap:** The Heap So Far
- Freeing Memory
- **Practice: Pig Latin + Valgrind**
- **realloc**

```
cp -r /afs/ir/class/cs107/lecture-code/lect11 .
```

Example: Pig Latin

Let’s write a program that can convert text to Pig Latin! Simplified Pig Latin rules:

- If the word starts with a vowel, append “way”: *apple* -> *appleway*
- Otherwise, move all starting consonants to the end and append “ay”: *bridge* -> *idgebray*

We want to write a function **char *pig_latin(const char *in)** that returns the Pig Latin version of the given string.

- Good use case for heap allocation – array size is unknown until we convert it to Pig Latin! We’ll create and return a heap-allocated string.
- The caller must free the string when it is done.

Example: Pig Latin

We will also see an example of how to uncover memory leaks using Valgrind.

```
valgrind --leak-check=full --show-leak-kinds=all [program info here]
```

Demo: Pig Latin + Valgrind

`pig_latin.c`

Memory Leaks vs. Memory Errors

Memory Leak: we didn’t free all heap memory we allocated. Rarely causes functionality issues, can generally be resolved after program is functionally complete.

Memory Error: we used memory in an invalid way (accessing memory that doesn’t belong to us, read uninitialized memory, etc.). Can cause gnarly functionality issues, make sure to resolve immediately!

Lecture Plan

- **Recap:** The Heap So Far
- Freeing Memory
- **Practice:** Pig Latin + Valgrind
- realloc

```
cp -r /afs/ir/class/cs107/lecture-code/lect11 .
```

realloc

```
void *realloc(void *ptr, size_t size);
```

- The **realloc** function takes an existing allocation pointer and enlarges to a new requested size. It returns the new pointer.
- If there is enough space after the existing memory block on the heap for the new size, **realloc** simply adds that space to the allocation.
- If there is not enough space, **realloc** *moves the memory to a larger location*, frees the old memory for you, and *returns a pointer to the new location*.

realloc

```
char *str = strdup("Hello");
assert(str != NULL);
…

// want to make str longer to hold "Hello world!"
char *addition = " world!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
assert(str != NULL);

strcat(str, addition);
printf("%s", str);
free(str);
```

realloc

- realloc only accepts pointers that were previously returned by malloc/etc.
- Make sure to not pass pointers to the middle of heap-allocated memory.
- Make sure to not pass pointers to stack memory.

Cleaning Up with `free` and `realloc`

You only need to free the new memory coming out of `realloc`—the previous (smaller) one was already reclaimed by `realloc`.

```
char *str = strdup("Hello");
assert(str != NULL);
…
// want to make str longer to hold "Hello world!"
char *addition = " world!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
assert(str != NULL);
strcat(str, addition);
printf("%s", str);
free(str);
```

Stack and Heap

- Generally, unless a situation requires dynamic allocation, stack allocation is preferred. Often both techniques are used together in a program.
- Heap allocation is a necessity when:
  - you have a very large allocation that could blow out the stack
  - you need to control the memory lifetime, or memory must persist outside of a function call
  - you need to resize memory after its initial allocation

Recap

- **Recap:** The Heap So Far
- Freeing Memory
- **Practice:** Pig Latin + Valgrind
- realloc

Next time: C Generics

Lecture 11 takeaway: We can allocate memory on the heap to manage it ourselves. We manipulate heap memory via pointers. We must free memory when we are done with it, and can resize prior allocations with realloc.

Structs

Structs

A *struct* is a way to define a new variable type that is a group of other variables.

```
struct date {               // declaring a struct type
    int month;
    int day;                // members of each date structure
};
…

struct date today;                              // construct structure instances
today.month = 1;
today.day = 28;

struct date new_years_eve = {12, 31};   // shorter initializer syntax
```

Structs

Wrap the struct definition in a **typedef** to avoid having to include the word **struct** every time you make a new variable of that type.

```
typedef struct date {
    int month;
    int day;
} date;
…

date today;
today.month = 1;
today.day = 28;

date new_years_eve = {12, 31};
```

Structs

If you pass a struct as a parameter, like for other parameters, C passes a **copy** of the entire struct.

```
void advance_day(date d) {
	d.day++;
}

int main(int argc, char *argv[]) {
	date my_date = {1, 28};
	advance_day(my_date);
	printf("%d", my_date.day); // 28
	return 0;
}
```

Structs

If you pass a struct as a parameter, like for other parameters, C passes a **copy** of the entire struct. **Use a pointer to modify a specific instance.**

```
void advance_day(date *d) {
     (*d).day++;
}

int main(int argc, char *argv[]) {
     date my_date = {1, 28};
     advance_day(&my_date);
     printf("%d", my_date.day); // 29
     return 0;
}
```

Structs

The **arrow** operator lets you access the field of a struct pointed to by a pointer.

```
void advance_day(date *d) {
      d->day++;         // equivalent to (*d).day++;
}

int main(int argc, char *argv[]) {
      date my_date = {1, 28};
      advance_day(&my_date);
      printf("%d", my_date.day); // 29
      return 0;
}
```

Structs

C allows you to return structs from functions as well. It returns whatever is contained within the struct.

```
date create_new_years_date() {
     date d = {1, 1};
     return d;          // or return (date){1, 1};
}

int main(int argc, char *argv[]) {
     date my_date = create_new_years_date();
     printf("%d", my_date.day); // 1
     return 0;
}
```

Structs

sizeof gives you the entire size of a struct, which is the sum of the sizes of all its contents.

```
typedef struct date {
      int month;
      int day;
  } date;

int main(int argc, char *argv[]) {
      int size = sizeof(date);   // 8
      return 0;
}
```

Arrays of Structs

You can create arrays of structs just like any other variable type.

```
typedef struct my_struct {
     int x;
     char c;
} my_struct;

…

my_struct array_of_structs[5];
```

Arrays of Structs

To initialize an entry of the array, you must use this special syntax to confirm the type to C.

```
typedef struct my_struct {
     int x;
     char c;
} my_struct;

…

my_struct array_of_structs[5];
array_of_structs[0] = (my_struct){0, 'A'};
```

Arrays of Structs

You can also set each field individually.

```
typedef struct my_struct {
     int x;
     char c;
} my_struct;

…
my_struct array_of_structs[5];
array_of_structs[0].x = 2;
array_of_structs[0].c = 'A';
```

Extra Practice

Goodbye, Free Memory

Where/how should we free memory below so that all memory is freed properly?

```c
char *str = strdup("Hello");
assert(str != NULL);
char *ptr = str + 1;
for (int i = 0; i < 5; i++) {
  int *num = malloc(sizeof(int));
  *num = i;
  printf("%s %d\n", ptr, *num);
}
printf("%s\n", str);
```

Recommendation: Don’t worry about putting in frees until **after** you’re finished with functionality.

Memory leaks will rarely crash your CS107 programs.

Goodbye, Free Memory

Where/how should we free memory below so that all memory is freed properly?

```
char *str = strdup("Hello");
assert(str != NULL);
char *ptr = str + 1;
for (int i = 0; i < 5; i++) {
  int *num = malloc(sizeof(int));
  *num = i;
  printf("%s %d\n", ptr, *num);
  free(num);
}
printf("%s\n", str);
free(str);
```

Recommendation: Don’t worry about putting in frees until **after** you’re finished with functionality.

Memory leaks will rarely crash your CS107 programs.

strcat_extend

Write a function that takes in a heap-allocated **str1,** enlarges it, and concatenates **str2** onto it.

```
char *strcat_extend(char *heap_str, const char *concat_str) {
    (_________________(1)__________________);
    heap_str = realloc(___(2A)___,___(2B)___);
    (_________________(3)__________________);
    strcat(___(3A)___, ___(3B)___);
    return heapstr;
}
```

Example usage:

```
char *str = strdup("Hello ");
str = strcat_extend(str, "world!");
printf("%s\n", str);
free(str);
```

strcat_extend

Write a function that takes in a heap-allocated **str1,** enlarges it, and concatenates **str2** onto it.

```
char *strcat_extend(char *heap_str, const char *concat_str) {
    int new_length = strlen(heap_str) + strlen(concat_str) + 1;
    heap_str = realloc(heap_str, new_length);
    assert(heap_str != NULL);
    strcat(heap_str, concat_str);
    return heapstr;
}
```

Example usage:

```
char *str = strdup("Hello ");
str = strcat_extend(str, "world!");
printf("%s\n", str);
free(str);
```