

Pointer Operations!



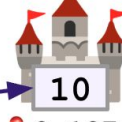
Addressing: `&` is the **address-of** operator, and the expression `&variable` will tell us where `variable` lives in memory by producing its 8-byte address.

```
int jerryland = 10;
int* intPtr = &jerryland;
```

intPtr

0x107

jerryland



enchanted
Troccoli
Forest



KarelWorld



Dereferencing *pointer

Referencing &variable

Dereferencing: `*` is the **magic wand of dereferencing**: `*ptr` obtains the value being pointed at, like entering or clicking a URL. Recall that pointers hold address values.

```
*intPtr = 40; // new value!
```

Like `intPtr[0]`,
`*intPtr` is the 4-byte
value starting at `0x107`

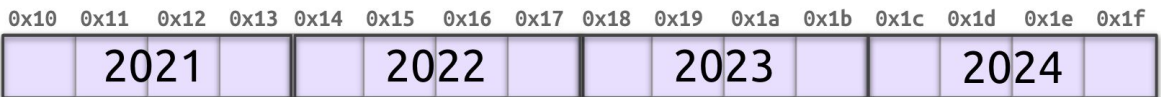
intPtr

0x107



Pointer arithmetic works in terms of the **size of the data type being pointed to**

→ `+1` for an `int*` means advance one `int`, so **4 bytes**

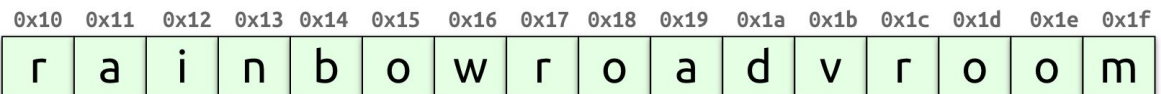


int* ptr

int* ptr+1

int* ptr+2

→ `+1` for a `char*` means advance one `char`, so just **1 byte**



char* ptr

char* ptr+4

char* ptr+9

Pointer Arithmetic



C is Pass by Value



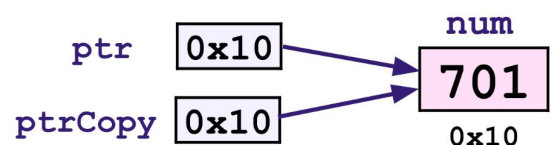
When passing parameters, **C passes a copy of whatever is passed.**

- ☐ To change values, and have those changes persist outside the function, **we must**  **pass a pointer to the value** (i.e. its address in memory).
- ☐ For instance, to change an `int`, we have to pass in an `int*` (pointer to an `int`)
- ☐ To change a `char*` (string), we have to pass in a `char**` (pointer to string).

Short Example

```
function mischief(int* ptrCopy) {
    *ptrCopy = 701;
}

int main() {
    int num = 107;
    int* ptr = &num;
    mischief(ptr);
}
```



Though the pointer is itself duplicated, it's **still pointing to the same value** in memory, and so **we can directly modify that value!**



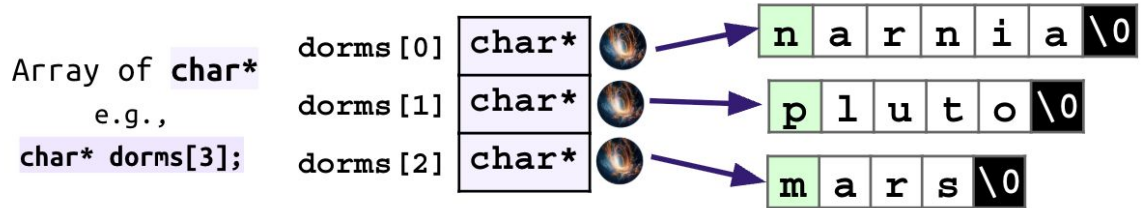
Heated Rivalry: Arrays, Pointers



- When passing **arrays as parameters**, they decay into a **pointer to the first element**
- A common bug is calling `sizeof()` on an array that's decayed into an 8-byte pointer.

```
char dorm_name[100];
enchanted ... \n
inhabit(char* dorm_name);
```

- Array of pointers** = array where each element is a pointer to some data.
- For example, an **array of strings** stores **pointers to their first characters**, NOT the characters themselves. Those can be stored elsewhere in memory

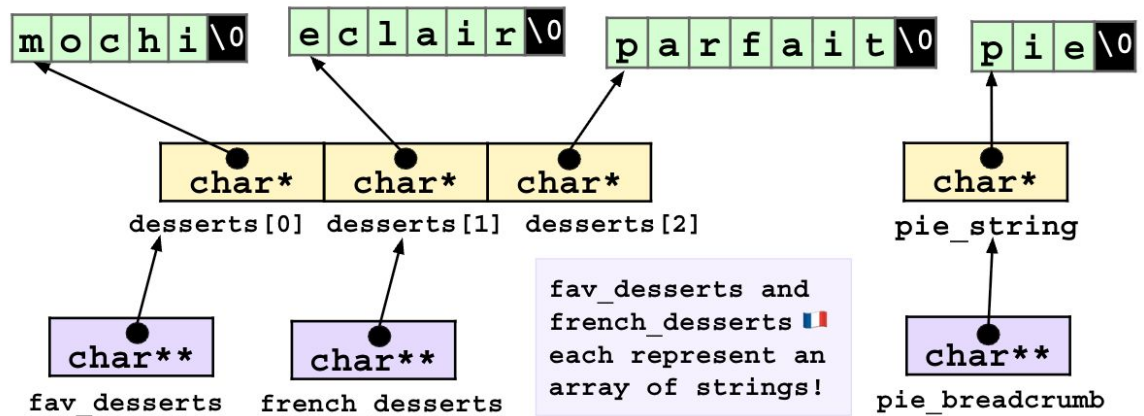


From my experience, `char*` vs `char**` is the #1 mixup on CS107 midterms (it's tricky!)

With a strings task, I like to illustrate **any characters or char buffers on the top layer!**

I then illustrate any `char*` (strings) on the 2nd layer, which point to chars on top layer.

Lastly, I illustrate any double pointers, `char**` (pointers to `char*`), on the bottom layer.



```
// string array + standalone string
char* desserts[3];
char* pie_string = "pie";
```

```
desserts[0] = "mochi";
desserts[1] = "eclair";
desserts[2] = "parfait";
```

```
char** fav_desserts = desserts;
// or equivalently, &desserts[0]
char** french_desserts = desserts + 1;
char** pie_breadcrumb = &pie_string;
```

Sinners II: Smoke Heap and Stack



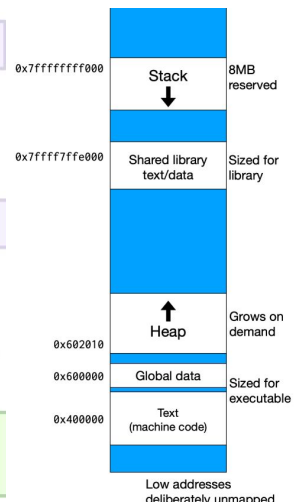
What are the general **benefits of heap memory** over stack memory?

- Heap memory is **more plentiful** + we can **resize memory** after its initial allocation!
- Heap memory **persists outside a function call** (unlike stack memory, which is erased when the call is over), so we have better control over its memory lifetime.

What are the general **disadvantages of heap memory** over stack memory?

- More opportunities for errors**, such as type errors, forgetting to free all allocations before the program is done, allocating wrong size, pointer errors, etc.
- Stack allocation is faster, more convenient** (automatic allocation and deallocation, can declare and initialize in one step), and has superior type safety.

Programs typically use both in combination, but **generally stick to stack allocation unless we need dynamic allocation**, e.g., may need to resize later.



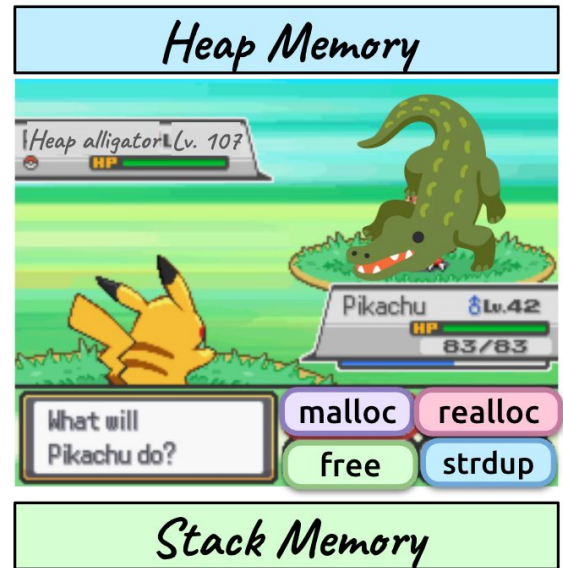
New Character: Heap Alligator!



- ❑ **Stack memory** is used for function call stack frames, local variables, etc.
- ❑ **Heap memory** is a shared memory oasis for dynamic memory allocation!

- ❑ We can't freely give ourselves memory on the heap at random addresses 🙌
- ❑ **Imagine the heap as being guarded fiercely by the heap alligator!** 🐊 (secret identity to be revealed on Assign6)

- ❑ To obtain heap memory, we need to 🗨️ communicate with the heap alligator!
- ❑ **The alligator understands requests such as malloc, realloc, free, and can set aside precious heap memory for us!**



Heap Allocator Methods (Main Trio)



void* malloc(size_t size);

- ❑ **Input the number of raw bytes** to be allocated on the heap, i.e. the request size.
- ❑ **Returns base address** of freshly allocated memory.

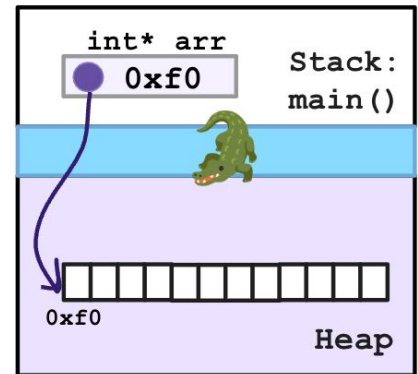
void* realloc(void *ptr, size_t new_size);

- ❑ **Input pointer to heap memory** given by malloc, and a **new total size to reallocate**.
- ❑ **Returns base address** of resized memory region.
- ❑ May move to new address if can't expand in-place! (will ferry the existing memory over)

void free(void* ptr);

- ❑ **Accepts pointer to heap** from malloc, realloc, etc. and **recycles the memory back** to heap.

```
int* arr = malloc(
    3 * sizeof(int));
```



🦋 **Note:** Don't forget to free your heap memory before the program ends! Otherwise a memory leak!

Other Amazing Allocator Methods!



char* strdup(const char* str);

- ❑ A convenience function for **duplicating C strings onto the heap**.
- ❑ Example with duplicating 🌹 "rose" on right:

```
char[5] flower;
strcpy(flower, "rose");
char* new_flower = strdup(flower);
```

void* calloc(size_t count, size_t size);

- ❑ An allocation function that's just like **malloc**, except that it **zeroes out** the allocated memory!
- ❑ Example to create array of 3 integers set to 0:

```
int* all_zeroes = calloc(3, sizeof(int));
```

