

# CSE 374 Lecture 13

The Heap



- Homework 3 due tonight
- Review previous lectures via website
  - Additional reading is also linked there
- Office Hours This Week
  - Monday: Cynthia 11-12
  - Wednesday: Simon 4-5
  - Thursday: Dixon 3-4
  - Friday: Keith 11-12
    - *Schedule via calendly for Megan (weekdays) & Andrew (weekends)*

# I/O : Printf, scanf

Printf and scanf are two I/O functions, prototyped in `stdio.h`

- Printf (print-format)
- `int printf(const char *format, ...)`
- 'Format' is a string that can contain format tags
- + additional arguments to match tags
- Number of arguments better match number of %
- Corresponding arguments better have the right types (%d, int; %f, float; %e, float (prints scientific); %s, \0- terminated char\*; ... Compiler might check, but not guaranteed
  - ◆ best case scenario: you crash
- `printf("%s: %d %g\n", p, y+9, 3.0)`
- scanf (gets input, formatted)
- `int scanf(const char *format, ...)`
- 'Format' is a string that can contain format tags
- + additional arguments to match tags - should be pointers to the right data type so input can be stored in them
- `scanf("%d %s", &n, str);`
- `scanf("%*s %d", &a);`
  - ◆ %\*s ignores string until space, then reads in an integer

# The stack

Stack stores active functions & local variables

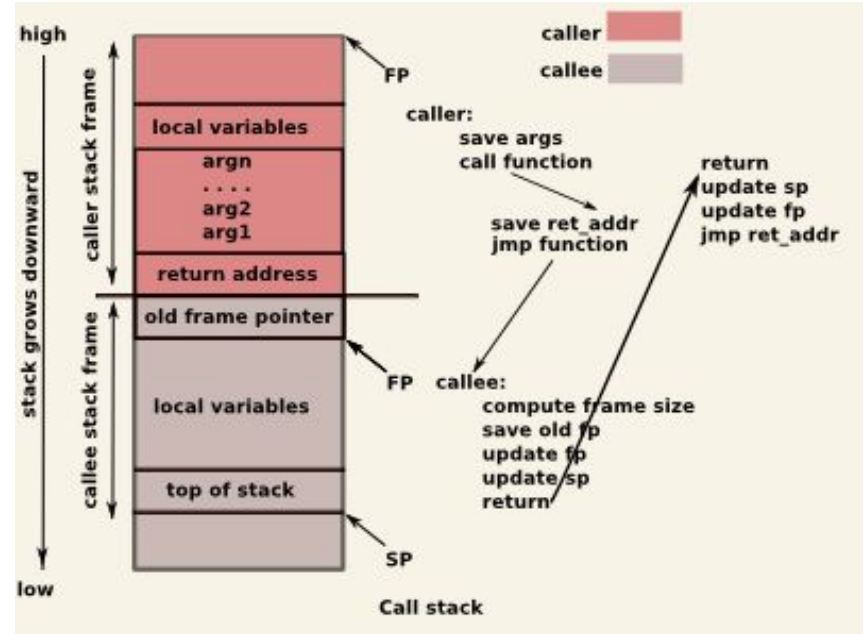
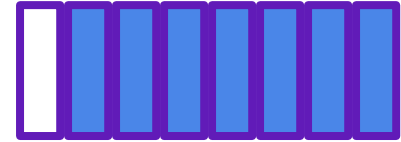
Frames deleted when function returns

Local variables do not persist

Local variables must have defined size

Can not make run-time adjustments

<- stack



# The heap

- Gives us flexible space
- Allocated at run time, with current space requirements
- Persistent until specifically free-ed
- User allocates memory with `malloc`

heap ->



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

## Allocate memory block

Allocates a block of size bytes of memory, returning a pointer to the beginning of the block.

Returns NULL in failure -> *should always check for NULL before using a pointer*

# Malloc

- malloc is used in a specific way: `(T*) malloc (e * sizeof(T))`
  - User doesn't need to know `sizeof(T)` - use `sizeof` instead of '16'.
- Returns a pointer to memory large enough to hold an array of length `e` with elements of type `T`
- malloc returns an untyped pointer `(void*)`; the cast `(T*)` tells C to treat it as a pointer to a block of type `T`
- If allocation fails (extremely rare, but can happen), returns `NULL`. Programs must always check.

# Initialization

Malloc does not initialize

***Must go set initial values manually***

Calloc:

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

**Allocate and zero-initialize array**

Allocates a block of memory for an array of num elements, each of them size bytes long, and initializes all its bits to zero.

\*Malloc is faster



var = 555

Or

\*varptr=555

var

# Half-way done with memory management!

- We can now allocate memory of any size and have it “live” forever
  - ◆ For example, we can allocate an array and use it indefinitely
  - ◆ Unfortunately, computers do not have infinite memory so “living forever” could be a problem

## Garbage Collection

- Java solution: Conceptually objects live forever, but the system has a garbage collector that finds unreachable objects and reclaims their space
- C solution: You explicitly free an object’s space by passing a pointer to it to the library function free
  - ◆ Managing heap memory correctly is hard in complex software and is the disadvantage of C-style heap allocation

# Freeing memory

Dynamically allocating memory

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

## Deallocate memory block

A block of memory previously allocated by a call to [malloc](#), [calloc](#) or [realloc](#) is deallocated, making it available again for further allocations.

If `ptr` does not point to a block of memory allocated with the above functions, it causes *undefined behavior*.

If `ptr` is a *null pointer*, the function does nothing. Notice that this function does not change the value of `ptr` itself, hence it still points to the same (now invalid) location.

# Example

```
int * p = (int*)malloc(sizeof(int));  
p = NULL; /* LEAK! - lost address */  
int * q = (int*)malloc(sizeof(int));  
free(q);  
free(q); /* Best case: crash */  
int * r = (int*)malloc(sizeof(int));  
free(r);  
int * s = (int*)malloc(sizeof(int));  
*s = 19;  
*r = 17; /* Best case: crash */
```

If `foo` returns a pointer, can the caller free the memory? (Who owns that pointer?)

If `bar` gets two pointers, can it free one, or both?

# Rules

- For every run-time call to malloc there should be one runtime call to free
- If you “lose all pointers” to an object, you can’t ever call free (a leak)!
  - ◆ Think hard before re-assigning a pointer; where is it pointing
- If you “use an object after it’s freed” (or free it twice), you used a dangling pointer!
- Note: It’s possible but rare to use up too much memory without creating “leaks via no more pointers to an object”

# Valgrind

`$valgrind program arguments`

```
$valgrind ./dangling --leak-check-full
```

Tool used to analyze memory usage (and other things)

Catches pointer errors during execution

Prints summary of heap usage, including details of memory leaks

---

# So why didn't we free our array in reverse.c?

The process (a running program) has a single address space for code, globals, the stack, & the heap

When the process exits, the entire address space is freed.

OK to rely on this in many cases

However, good practice to provide mechanism to free any memory allocated in a package, allowing potential clients to release code if desired

```
/* Header stuff */
/* Return a new string with the contents of s backwards */
char * reverse(char * s) {
    char * result = NULL;          /* the reversed string */
    int L, R;  char ch;
    int strsize = strlen(s)+1; // CHANGED
    result = (char *)malloc(strsize); // CHANGED
    strncpy(result, s, strsize); // CHANGED

    L = 0;  R = strlen(result) - 1; // CHANGED
    while (L < R) {
        ch = result[L];
        result[L] = result[R];
        result[R] = ch;
        L++; R--;
    }
    return result;
}

/* Main:  also doesn't free the memory pointed to by result */
```

