{C}

Lecture 12
Dynamic Memory Allocation

Memory: the C Story

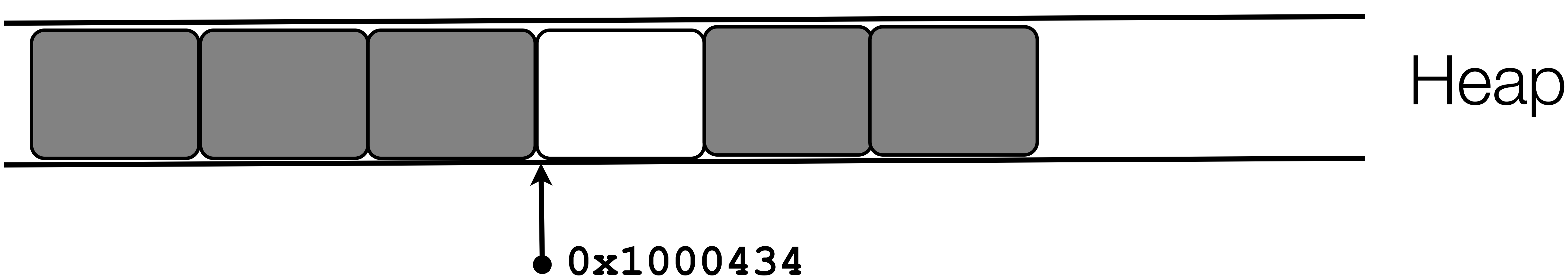
C offers a story both simpler and more complex than Java

Memory is a sequence of bytes, read/written by providing an address

Addresses are values manipulated using arithmetic & logic operations

Memory can be allocated:

- Statically
- Dynamically on the stack
- Dynamically on the heap



Static and Stack allocation

- Static allocation with the keyword `static`
- Stack allocation automatic by the compiler for local variables
- `printf` can display the address of any identifier

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

static int sx;
static int sa[100];
static int sy;

int main() {
   int lx;
   static int sz;

   printf("%p\n", &sx);    0x100001084
   printf("%p\n", &sa);    0x1000010a0
   printf("%p\n", &sy);    0x100001230
   printf("%p\n", &lx);    0x7fff5fbff58c
   printf("%p\n", &sz);    0x100001080
   printf("%p\n", &main);  0x100000dfc
```

Static and Stack allocation

Any value can be turned into a pointer

Arithmetics on pointers allowed

Nothing prevents a program from writing all over memory

```
static int sx;
static int sa[100];
static int sy;

int main() {
  for(p= (int*)0x100001084;
      p <= (int*)0x100001230;
      p++)
 {
    *p = 42;
  }
  printf("%i\n",sx);
  printf("%i\n",sa[0]);
  printf("%i\n",sa[1]);
```

```
42
42
42
```

Memory layout

The OS creates a process by assigning memory and other resources

C exposes the layout as the programmer can take the address of any element (with &)

Stack:

- *keeps track of where each active subroutine should return control when it finishes executing; stores local variables*

Heap:

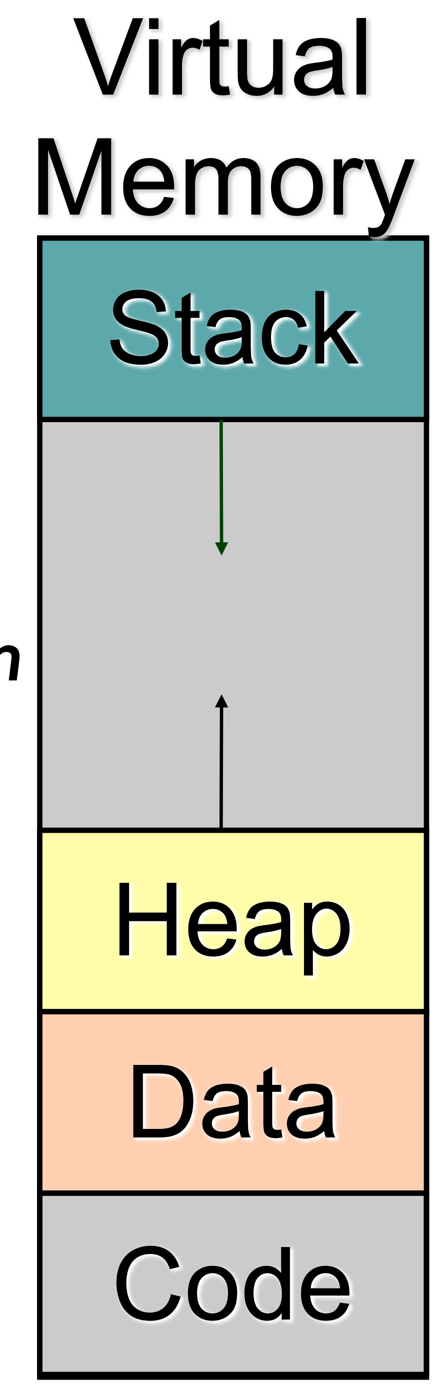
- *dynamic memory for variables that are created with malloc, calloc, realloc and disposed of with free*

Data:

- *global and static variables*

Code:

- *instructions to be executed*



Dynamically Allocated Memory

A simple dynamic allocation pattern is to ask the OS for a chunk of memory large enough to store *all* data needed

`mmap` is a general Linux utility that "memory maps" a piece of address space for use by the application

The downside is that the programmer must keep track of how memory is used

```
int main() {
 int* x; int* start;
 double* y;
 start = (int*)
         mmap ( NULL, 5*sizeof(int),
                PROT_READ | PROT_WRITE, MAP_PRIVATE |
                MAP_ANONYMOUS, 0, 0 );
  x = start;
  *x = -42;
  x++;
  y = (double*) x;
  y++;
  x = (int*) y;
  *x = 42;
 printf("%i\n", start[0]);  // -42
 printf("%i\n", start[1]);  // 0
 printf("%i\n", start[2]);  // 0
 printf("%i\n", start[3]);  // 42
 printf("%i\n", start[4]);  // 0
 y = (double*) start;
 *y = 2.1;
 printf("%f\n",              // 2.100000
     *(double*)start);
}
```

mmap

- Manipulates process page table, a data structure used to map virtual to physical addresses
- Allocates addresses on demand
- Can fail if process runs out of address space
- Commonly used to allow memory operations to implicitly access files

Dynamic memory management


```
#include <stdlib.h>

void* calloc(size_t n, size_t s)
void* malloc(size_t s)
void  free(void* p)
void* realloc(void* p, size_t s)
```

Allocate and free dynamic memory

malloc(size_t s)

Allocates `s` bytes and returns a pointer to the allocated memory.

Memory is not cleared

Returned value is a pointer to alloc'd memory or `NULL` if the request fails

You must cast the pointer

```
p = (char*) malloc(10); /* allocated 10 bytes */
if(p == NULL) { /*panic*/ }
```

CAN FAIL, CHECK THE RETURNED POINTER NOT NULL

`calloc(size_t n, size_t s)`

Allocates memory for an array of `n` elements of `s` bytes each and returns a pointer to the allocated memory.

The memory is set to zero

The value returned is a pointer to the allocated memory or `NULL`

```
p = (char*) calloc(10,1); /*alloc 10 bytes */
if(p == NULL) { /* panic */ }
```

CAN FAIL, CHECK THE RETURNED POINTER NOT NULL

What's the difference between `int array[10]` and `calloc(10,4)`

`free(void* p)`

- Frees the memory space pointed to by `p`, which *must* have been allocated with a previous call to `malloc`, `calloc` or `realloc`
- If memory was not allocated before, or if `free(p)` has already been called before, undefined behavior occurs.
- If `p` is `NULL`, no operation is performed.
- `free()` returns nothing

```
char *mess = NULL;
mess = (char*) malloc(100);
…
free(mess);   *mess = 43;
```

FREE DOES NOT SET THE POINTER TO NULL

`realloc(void* p,size_t s)`

Changes the size of the memory block pointed to by `p` to `s` bytes

Contents unchanged to the minimum of old and new sizes

Newly alloc'd memory is uninitialized.

Unless `p==NULL`, it must come from `malloc`, `calloc or realloc`.

If `p==NULL`, equivalent to `malloc(size)`

If `s==0`, equivalent to `free(ptr)`

Returns pointer to alloc'd memory, may be different from `p`, or NULL if the request fails or if `s==0`

If fails, original block left untouched, i.e. it is not freed or moved

`memcpy(void*dest,const void*src,size_t n)`

Copies `n` bytes from `src` to `dest`

Returns `dest`

Does not check for overflow on copy

```
char buf[100];
char src[20] = “Hi there!”;
int type = 9;
memcpy(buf, &type, sizeof(int)); /* copy an int */
memcpy(buf+sizeof(int), src, 10); /*copy 10 chars */
```

`memset(void *s, int c, size_t n)`

Sets the first `n` bytes in `s` to the value of `c`

- (`c` is converted to an `unsigned char`)

Returns `s`

Does not check for overflow

```
memset(mess, 0, 100);
```

Memory Allocation Problems

Memory leaks

- Alloc'd memory not freed appropriately
- If your program runs a long time, it will run out of memory or slow down the system
- Always add the free on all control flow paths after a malloc

```
void *ptr = malloc(size);
/*the buffer needs to double*/
size *= 2;
ptr = realloc(ptr, size);
if (ptr == NULL)
   /*realloc failed, original address in ptr
    lost; a leak has occurred*/
   return 1;
```

Memory Allocation Problems

Use after free

- Using dealloc'd data
- Deallocating something twice
- Deallocating something that was not allocated

Can cause unexpected behavior. For example, malloc can fail if "dead" memory is not freed.

More insidiously, freeing a region that wasn't malloc'ed or freeing a region that is still being referenced

```
int *ptr = malloc(sizeof (int));
free(ptr);
*ptr = 7; /* Undefined behavior */
```

Memory Allocation Problems

Memory overrun

- Write in memory that was not allocated
- The program will exit with segmentation fault
- Overwrite memory: unexpected behavior

```
int* y= …
int* x= y+10
for(p= x; p >= y;p++)
{
    *p = 42;
}
```

Memory Allocation Problems

Fragmentation

- The system may have enough memory but not in contiguous region

```
int* vals[10000];

int i;
for (i = 0; i < 10000; i++)
  vals[i] = (int*) malloc(sizeof(int*));

for (i = 0; i < 10000; i = i + 2)
  free(vals[i]);
```

Checklist

`NULL` pointer at declaration

Verify `malloc` succeeded

Initialize alloc'd memory

free when you *malloc*

`NULL` pointer after free

Allocator Requirements

- Must be able to service arbitrary interleaving of malloc() and free() requests.
  - malloc() must return a pointer to a region of contiguous memory greater than or equal to the requested size, or it must return NULL
  - Contents of memory uninitialized
- The allocator cannot control or schedule the number or shape of requests (i.e., it is not allowed to reorder or buffer malloc requests)
- Allocated blocks must be aligned and cannot be moved

It is desirable that blocks allocated close in time are located close in space