

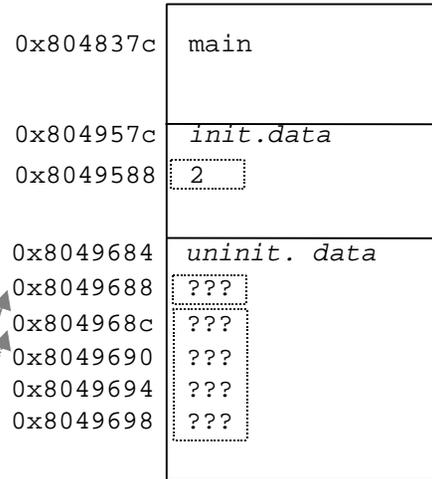
Static Allocation

- Recall: static allocation happens at compile time based on variable definitions.

```
int x = 2;
int a[4];
int *b;
```

```
int main() {}
```

SYMBOL TABLE:				
main	0x804837c	.text	f9	
x	0x8049588	.data	04	
b	0x8049688	.bss	04	
a	0x804968c	.bss	10	



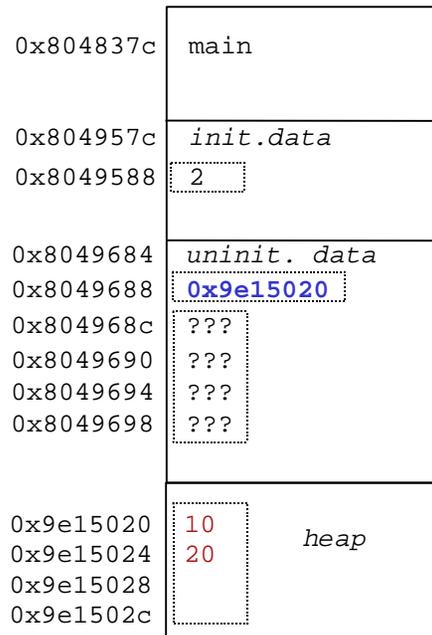
Dynamic Memory Allocation

- In Java,
 - Set `s`; // Memory is allocated for pointer `s`
 - // Memory is allocated for object
 - `s = new HashSet();`
- In C,
 - `int *a;` /* Memory is allocated for pointer `a` */
 - /* Memory is allocated for `a` to point to */
 - `a = (int *)malloc(10 * sizeof(int));`

Dynamic Allocation

```
int x = 2;
int a[4];
int *b;

int main() {
    b = (int *)malloc( 4 *
                      sizeof(int));
    b[0] = 10;
    b[1] = 20;
}
```



SYNOPSIS

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

```
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

DESCRIPTION

malloc() allocates `size` bytes and returns a pointer to the allocated memory. The memory is not cleared.

free() frees the memory space pointed to by `ptr`, which must have been returned by a previous call to **malloc()**, **calloc()** or **realloc()**. Otherwise, or if **free(ptr)** has already been called before, undefined behaviour occurs. If `ptr` is **NULL**, no operation is performed.

malloc

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

- Some things you haven't seen yet:

```
void *
```

- A generic pointer type that can point to memory of any type.

```
size_t
```

- A type defined by the standard library as the type returned by `sizeof`.
- The type is unsigned int.

5

malloc

- Usually cast the return value of `malloc` to the type you want.

```
int *i = (int *)malloc(sizeof(int));  
char *c = (char *)malloc(NAME_SIZE);
```

- `sizeof` works on types, and knows type of expressions.

```
double *d = (double *)malloc(5*sizeof(*d));
```

- Be careful to allocate the correct number of bytes.
- E.g., `int *i = (int *)malloc(1); /*wrong*/`
– allocates 1 byte, not 1 int.

6

NULL pointers

- A function that returns a block of memory might fail to do so, in which case it returns a null pointer.
- `NULL` is a pre-processor variable defined in `stdlib.h` (included from `stdio.h`) and other places
 - it is usually defined to be 0 (no program allocates anything at address 0x0)

7

De-allocating memory

```
int *a = (int *)malloc(10 * sizeof(int));  
int b[10];  
...  
a = b;
```

- What is wrong with the last line? It compiles and runs fine.
- We have lost the pointer to the memory region allocated in the first line, so that space is now tied up until the program terminates.
- **Memory leak.**

8

free()

- Before removing the last pointer to a memory region, you must explicitly deallocate it.

```
int *a = (int *)malloc(10 * sizeof(int));
int b[10];
...
free(a);
a = b;
```

Is a NULL after the free statement? → No, free cannot change the value of a parameter
--

9

Dangling pointers

```
int *a = (int *)malloc(10 * sizeof(int));
...
free(a);
printf("%d\n", a[0]); /* Error */
```

- Dereferencing a pointer after the memory it refers to has been freed is called a “dangling pointer”.
- Behaviour is undefined.
 - appear to work
 - bogus data
 - program crash

10

Arrays of pointers

- Most obvious use is to get an array of strings.

```
#define SIZE 4
char **strs = (char **)malloc(3*sizeof(char *));

for(i = 0; i < 3; i++) {
    strs[i] = (char *)malloc(SIZE);
}
strs[0] = strncpy(strs[0], "209", SIZE);
strs[1] = strncpy(strs[1], "369", SIZE);
```

11

Tips

- Use a debugger and start to figure out what valid addresses look like.
- Check return values from library functions.
- Watch out for common errors:
 - forgetting to allocate memory when a pointer is declared.

12