Dynamic Memory

Static Memory

- Define variables when we write code
- When we write the code we decide
  - What the type of the variable is
  - How big array sizes will be
  - etc.
- These cannot change when we run the code!

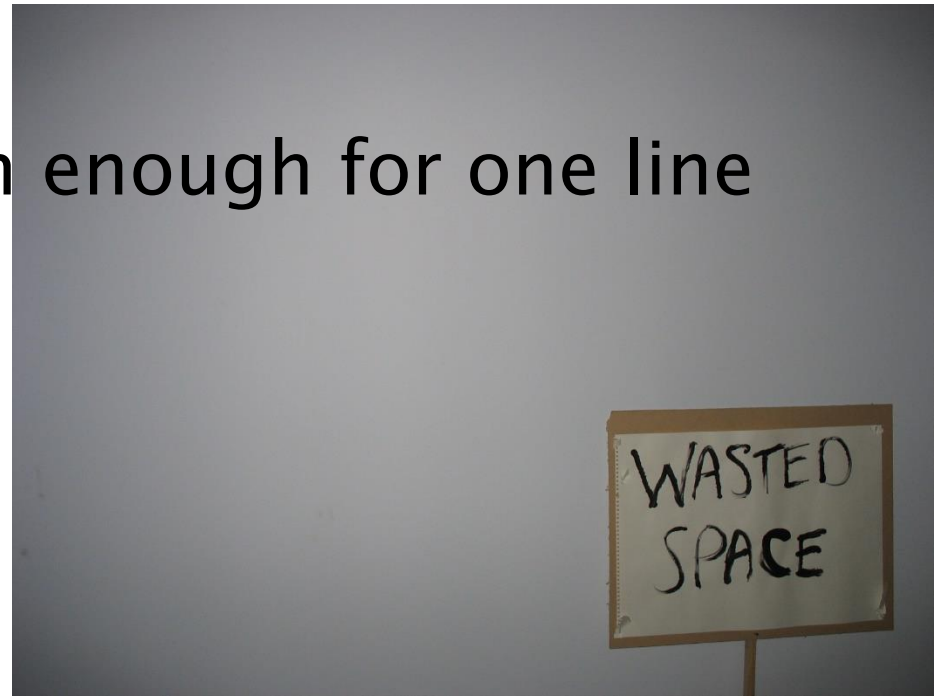
- For example:

```
char buffer[100]="This is a test.";
```

Implications of Static Memory

- Imposes limits on what our programs can handle
  - readLine(&buffer[0]) can't handle a line that's more than 99 chars long!
- Forces us to allocate enough space for the worst case
  - Waste space for the average case!

```
char buffer[4096]; // More than enough for one line
```



Dynamic Memory

- Standard library function call to request new memory

```
#include <stdlib.h>

void * malloc(int size);
```

Number of Bytes requested

Address of space returned
NULL if no space is available
Type is pointer to nothing.

What does malloc mean?

- (Abbreviation for "memory allocation")
- Operating system "owns" a portion of the address spaced called the "HEAP" – a heap of memory
- When you invoke malloc, the operating system finds a portion of the heap large enough to hold the number of bytes you requested
- By returning the address of that memory to you, the Operating System is granting control of that memory to you!
- Operating system is guaranteeing that no one other than your program will use that memory!

What's in malloc'ed memory?

- Malloc does not initialize memory for you!
- You get whatever happens to in memory when malloc completes

- Alternative: calloc
  - void *calloc(int num,int size);
  - Allocs "num" contiguous elements of size bytes each
  - Initializes everything to zero

The malloc "contract"

- You are guaranteed sole use of malloc'ed memory
- Nothing outside of your program will read or write that memory
- When you are finished using that memory, you must give it back to the operating system!

```
char * buffer=(char *)malloc(300); // get 300 bytes from heap

// use buffer here

free(buffer);  // return buffer 300 bytes to the heap
```

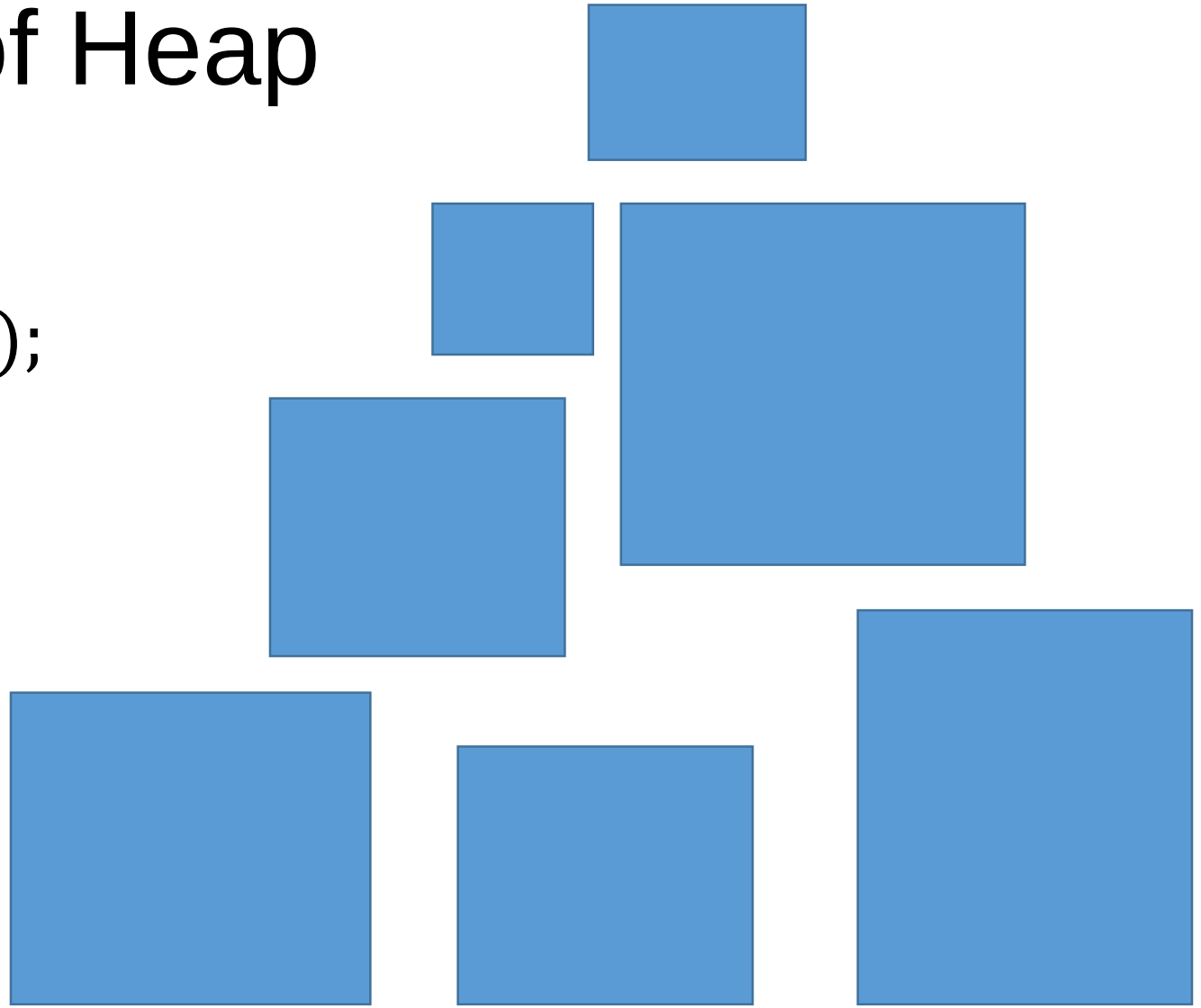
Graphic View of Heap

```
char * x=malloc(1200);

char * y=malloc(900);

free(y);

free(x);
```



Dynamic Node Allocation/Free

```
struct node * makeNode(int value) {
        struct node * np=
                (struct node *)malloc(sizeof(struct node));
        np->value=value;
        np->next=NULL;
        return np;
}
void freeNode(struct node * np) { free(np); }
```

The C “sizeof” operator/function

- Argument can be:
  - Type
  - Variable (or expression)

- Returns : number of bytes required for that type or for a variable in bytes

sizeof(char)==1, sizeof(int)==4, sizeof(num[4])==16

sizeof(struct node)==8 (int value; struct node *next)

Why Dynamic

- Get exactly as much memory as you need
  - No program limits
  - No wasted space
- Get memory as many times as you need
  - e.g. memory for each node in a list
  - Don’t have to guess how many nodes you will need
  - Don’t care how many nodes you need!

strdup

```
char * strdup(char * from);
```

- Combination of malloc and strcpy

```
char * strdup(char *from) {
        char *to=
            malloc(strlen(from)+1);
        strcpy(from,to);
        return to;
}
```

- Need to free result!

```
char buffer[4096];
while (!feof(stdin)) {
        buffer=getLine();
        char *ln=strdup(buffer);
        …
}
for(…)
        free(ln);
}
```

Why is “free” important?

- As long as you “own” memory, no-one else can use it
- If you don’t free, eventually, nothing is left in the heap
  - malloc then returns NULL pointers

- small print… when your program exits, any space you have malloc’ed is freed.
- It’s not uncommon to run for days and days
- Do you turn off your laptop? Many programs start when you turn on your laptop, and don’t stop until you turn off your laptop.

- Be a good citizen… free your malloc’ed memory

Problem: Referencing Free'd Memory

```
char *buffer=(char *)malloc(300); // get 300 bytes
strcpy(buffer,"This is a test"); // use it
free(buffer);

strcpy(buffer,"This was a test");
```

returns space to heap
does not change
the value of buffer!

writes to memory I no longer own!
May work, but cause other problems
May cause segmentation violation

VALGRIND

- Memory Leak: Memory that has been malloc'ed, but not free'd
- Special program: "valgrind"
  - monitors each malloc
  - monitors each free
  - Reports on mallocs that have no corresponding free when program exits
  - run as: "valgrind --leak-check=full ./program arg1 arg2 <input.txt
  - Also reports on references to free'd memory
  - Also reports on array bounds violations

Alternative: Garbage Collection

- Need to know when programmer is using memory
  - Use of pointers introduce aliases
  - Therefore, pointers and garbage collection don't go together
- Periodically stop program execution for garbage collection
- "Automatically" free any memory that the program is no longer using.
  - Requires significant analysis to ensure you don't throw away something useful
- Adds about 10% performance penalty
- Benefit: Allows programmers to be sloppy housekeepers

Resources

- Programming in C, Chapter 16 (Dynamic Memory Allocation)
- Wikipedia Memory Management https://en.wikipedia.org/wiki/Memory_management
- valgrind home http://valgrind.org/
- Dynamic Memory Allocation Tutorial http://randu.org/tutorials/c/dynamic.php