iClicker Attendance

Please click on A if you are here:

A. I am here today.

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!

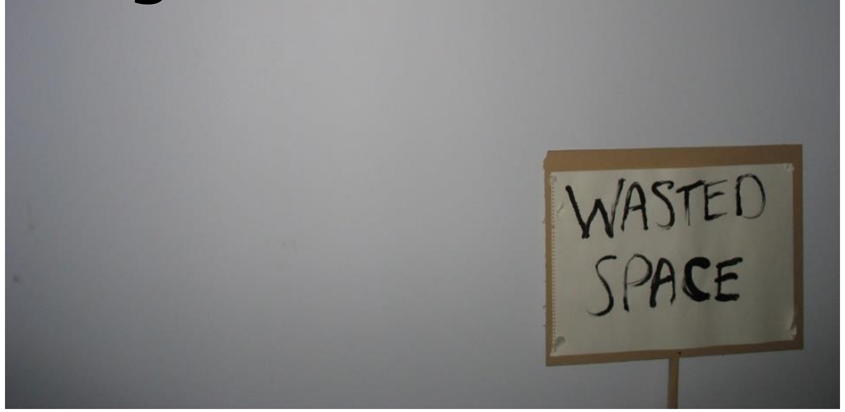
- For example:

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

Implications of Static Memory

- Imposes limits on what our programs can handle
  - char buffer[100];
  - 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 guarantees 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 is 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”

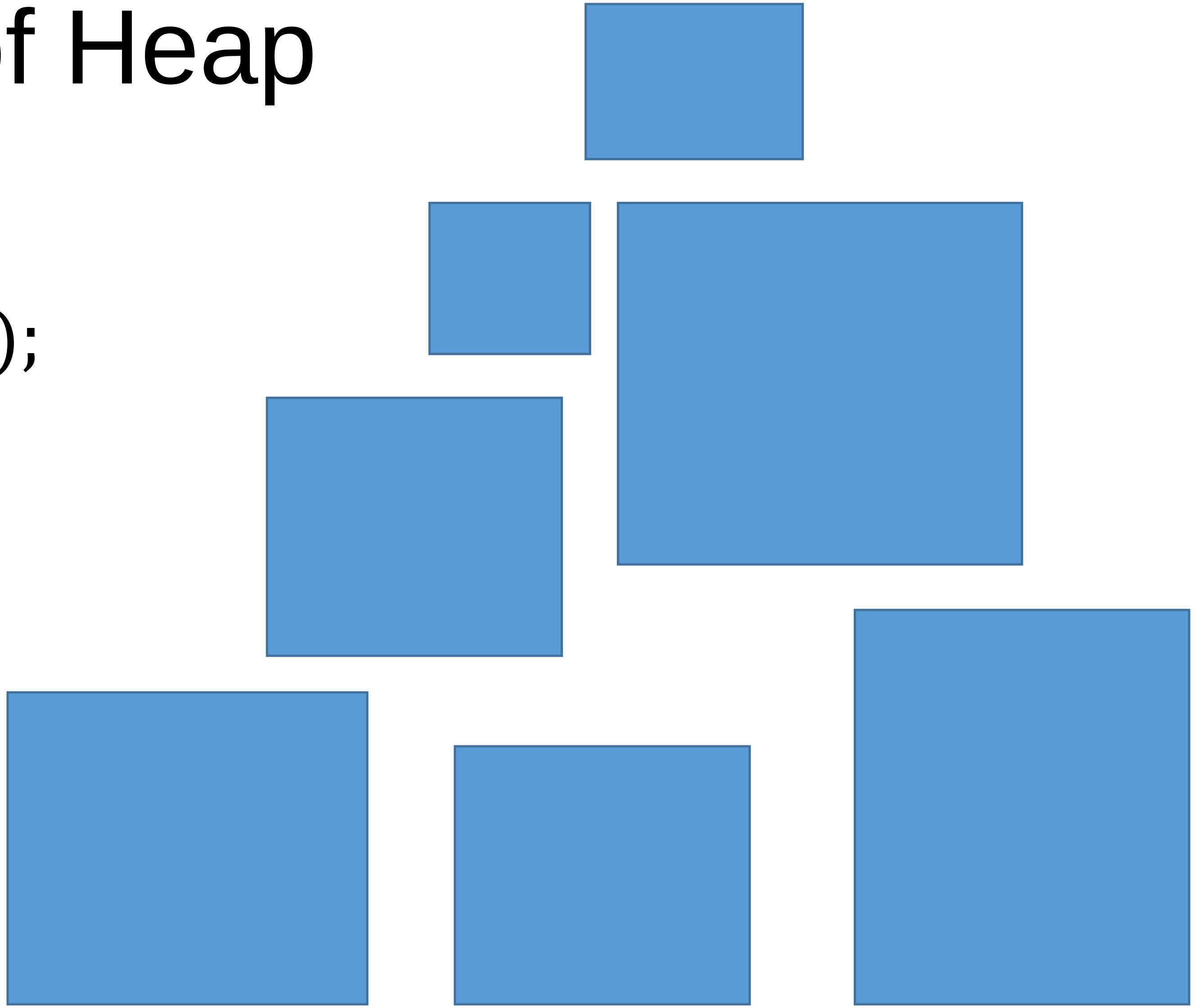
- You are guaranteed sole use of malloc’ed memory
- Nothing outside of your program will read or write that memory
- When you are finished using that memory, you must give it back to the operating system!

```
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);
```

Using Dynamic Memory for Strings

- We can use the library function fgets to read from a file until the next newline
  - fgets puts a null terminator after the newline
  - fgets returns a NULL address at the end of the file
  - fgets requires you to provide memory to write to
- Use "malloc" to provide the result to the user

A typical "nextLine" function

```
char * nextLine(FILE *codeFile) {
	static char buf[4096];
	if (fgets(buf,sizeof(buf),codeFile)==NULL) return NULL;
	char * line=malloc(strlen(buf)+1);
	strcpy(line,buf);
	return line;
}
```

Using "nextLine"

```
int ln=1;
while(1) {
	char * line=nextLine(codeFile);
	if (line==NULL) break;
	printf("%3d: %s",ln++,line);
	free(line);
}
```

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 line
  - Don't have to guess how big the line is
  - Don't care how many lines you need!

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: Dangling Pointers

```
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

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);
}
```

"nextLine" with strdup

```
char * nextLine(FILE *codeFile) {
	static char buf[4096];
	if (fgets(buf,sizeof(buf),codeFile)==NULL) return NULL;
	return strdup(buf);
	// char * line=malloc(strlen(buf)+1);
	// strcpy(line,buf);
	// return line;
}
```

Using Dynamic Memory: Dynamic Array

- Suppose we want an array of integers, but we don't know how many.
  - We want to add new values to the end of the array
  - We want to be able to get or put data into known indexes of the array
- Proposal… keep track of three data items:
  - number of integers we can use
  - number of integers we are using
  - pointer to an array of integers

iClicker Question

- What data structure should we use?

> - number of integers we can use
> - number of integers we are using
> - pointer to an array of integers

A. An integer
B. A float
C. An array of integers
D. An array of pointers
E. A structure

A structure for a dynamic array

```
struct dynArrayStruct {
	int max; // Number of integers at *data
	int used; // Number of integers we are using
	int *data; // Pointer to an array of integers
};
```

Structure vs. Structure Pointer

- We could pass the entire structure in as an argument
  - Need to copy 2 ints and a pointer – 16 bytes
  - Doesn't allow us to update the caller's view of the structure!
  - Therefore need to return the structure, but we may want to return other data!
- It's better to pass a pointer to the structure
  - Only copies a pointer – 8 bytes
  - Allows the functions to update the structure values
  - No extra return required

Creating a Dynamic Array

- Need a function to
  - Create a new instance of the dynArrayStruct structure
    - Including reserving memory for the structure itself!
  - Initialize all the fields
  - Return a pointer to the structure

newDynArray()

```
struct dynArrayStruct * newDynArray() {
	struct dynArrayStruct *n=
		malloc(sizeof(struct dynArrayStruct));
	n->max=16;
	n->used=0;
	n->data=(int *)malloc(sizeof(int)*n->max);
	return n;
}
```

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)

Need an add function

- Allow the user to add a new value
- If the dynamic array is not large enough to hold a new value
  - Make a new temporary array that is double the size
  - Copy the old array values to the new array
  - Free the memory for the old array
  - Update the dynamic array structure
- Now, the array is big enough…
  - put the users value into the array
  - Increment the number of used elements of the array

Dynamic Array "add" function

```
void add(struct dynArrayStruct *da,int val) {
   if (da->used>=da->max) { // At the limit... need to grow the array
      int *temp=malloc(sizeof(int)*2*da->max); // Double the size
      memcpy(temp,da->data,sizeof(int)*da->max); // Copy old data to new
      free(da->data); // Free old data
      da->data=temp; // Copy old data to new
      da->max*=2;
   }

   da->data[da->used]=val;
   da->used++;
}
```

Dynamic Array get and put functions

```
int get(struct dynArrayStruct *da,int index) {
	assert(index<da->used && index>=0);
	return da->data[index];
}

void put(struct dynArrayStruct *da,int index,int val) {
	assert(index<da->used &&index>=0);
	da->data[index]=val;
}
```

Array Bounds Checking

- Dynamic arrays can perform array bounds checking!

```
assert(index<da->used && index>=0);
```

Freeing a Dynamic Array

```
void freeDynArray(struct dynArrayStruct *da) {
	free(da->data);
	free(da);
}
```

Example Dynamic Array Use

- See useDyn.c

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
    - (Not available on Cygwin)

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

More examples of malloc and free

- verilog2.c (Project 3)
  - malloc is invoked to provide space for identifiers
  - No need to free because we are only working with one file
- verilog3.c (Project 4)
  - malloc is invoked to provide space for identifiers
  - malloc is used to keep memory for each pin, net, instance, and module
  - When a module is freed, free all the data for all pins, nets, instances in the module.

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