The Heap

Steven R. Bagley

Recap

- Data is stored in variables
- Can be accessed by the variable name
- Or in an array, accessed by name and index
  `a[42] = 35;`
- Variables and arrays have a type
  `int`, `char`, `double`, etc.
- Create our own data `structures`

Drawing Pictures

- Write C that outputs PostScript to draw pictures
- Create functions that print the commands
- Less chance of mistakes in the output
- The meaning of our program will be clearer
- Can use our `struct`s as parameters

Clearer, because we will see function calls that mean something (e.g. MoveToPoint, AddLineToPoint) rather than a series of printf programs

Reading Path from File

- Could also read the points from a file
- Until we get the line ‘`stop`’
- Use `fscanf()` and see whether it works
- Or we could use `fgets()` to read a line
- Compare with ‘`stop`’
- If not, use `sscanf` to process the string

And loop...
Go implement
Show some examples

Out of Memory

- Problem with this routine
- What if the number of points in the file is greater than the size of the array?
- Program will CRASH!
- Could stop when we fill the array
- But that would leave half the picture undrawn

Compile-time Memory

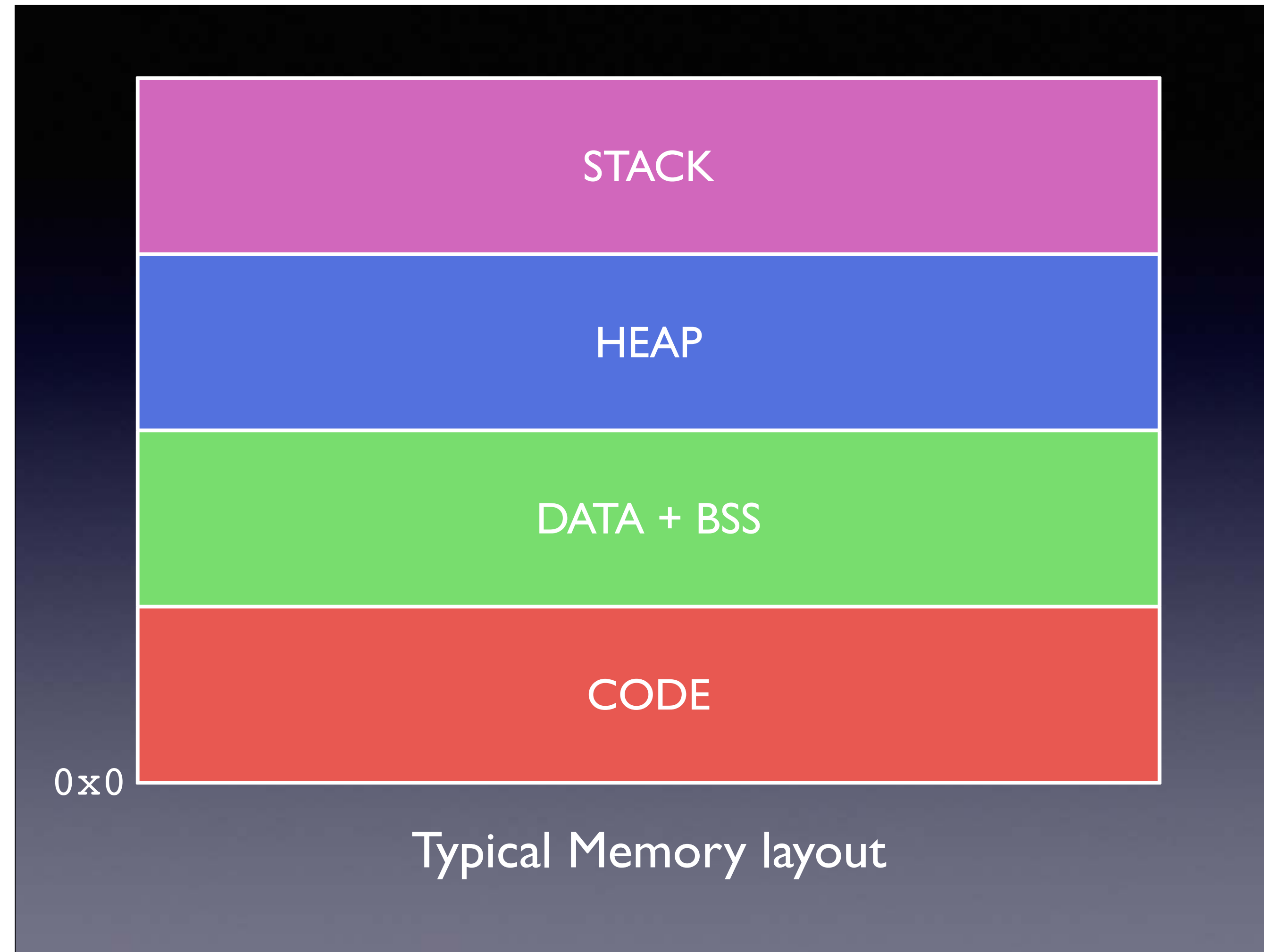
- Problem is that we set the size of the array at compile-time
- Size is fixed as the program runs
- And we don't know what the size of the input is here
- Have to guess, and hope we have enough space

Run-time Memory

- It would be better if we could set the size of the array at run-time
- When we know the size we need
  - Perhaps make the first line of file the number of points
  - Rather than looking for ‘`stop`’
- C lets us do this, but its manual…

A Program's Memory

- Need to understand the way a program manages memory
- Although all memory is identical, the program sees it as several different sections
- These sections are used for various purposes



Explain typical memory layout of a program

Code -- machine code
Data + BSS -- global variables, (plus other bits and pieces such as predefined strings)
Stack -- local variables (grows downward -- top of stack below bottom!)
Heap -- This is where objects live!

Allocating Memory

- Normally, our variables are created on the *stack*, and are local to the function
- Get a pointer to it by using `&`, the *address-of* operator
- Can also allocate memory on *the heap*
- The heap is the rest of available memory…
- Only have a pointer to it, no direct access

`malloc()`

- C lets us allocate memory by using the `malloc()` function
- Returns a pointer to the block of memory
- Need to specify how many *bytes* we want to allocate

`sizeof()`

- Could calculate the size manual
- `int` is 4 bytes, `float` 4 byes, `char` 1 byte…
- Add them up, so a struct point would be 8 bytes
- Error prone
- C provides the sizeof operator that does it automatically — `sizeof(struct point)`

Can ask for the size of any type

`malloc()` and casting

- `void *malloc(size_t size)`
  allocate `size` bytes and return a pointer to the address
- Note `void *` is a pointer to something, not nothing
- Need to cast the pointer to the right type before we can use it…

Casting

- Cast a value to another type by putting the type name in brackets before it e.g. to convert a `float` to `int`
```
float pi = 3.1415927;
int ipi = (int)pi;
```
- Will truncate values…
- Can also cast pointers
- This is very dangerous and very powerful

Allocating memory

- `int *p = (int *)malloc(sizeof(int));`
- Find the size of a `int`
- `malloc` that many bytes
- Cast the `void *` pointer to a `int *`
- And store in `p`

This generally safe, although handle with care
Always think why am I casting this?
Don't cast just to make the errors and warnings go away

Using allocated memory

- Can use this allocated int like any other pointer to an `int`
- But *have* to access it via the pointer
- Cannot obtain a variable name for it
- Can also allocate `structs` in this fashion

`free()`

- When we've finished with memory, we need to `free()` it
- Otherwise, it can't be used for anything else until the program quits
- But can't free it until we've finished with it (the program has no more pointers with it)
- Otherwise, accessing it will cause problem

`free()`

- `void free(void *ptr)`
- Pass it the pointer to the memory you want freeing
- Must be allocated via `malloc()`
- Afterwards, the variable containing the pointer should be cleared as it is no longer valid

Allocating `struct`

- `struct point *p = (struct point *) malloc(sizeof(struct point));`
- Find the size of a `struct point`
- `malloc` that many bytes
- Cast the `void *` pointer to a `struct point *`
- And store in `p`

This generally safe, although handle with care
Always think why am I casting this?
Don’t cast just to make the errors and warnings go away

Heap `structs`

- Can use these allocated blocks just like anything else we've pointed at

```
p->x = 42.0;
p->y = 36.0;
struct point pt = *p;
```

- Allocating `struct`s on the heap is used a lot
- Most data is stored on the heap

Arrays on the Heap

- Arrays are represented by a pointer to the base of the array
- Each element in the array is laid out in memory sequentially
- The array operator `[]` works just as well on a pointer as on an array

Arrays on the Heap

- If we can `malloc` space for one thing
- Then we can `malloc` space for two things
- Or three, or four etc…
- Treat that pointer as the base of the array

Arrays on the Heap

- Find out how big one thing is using `sizeof()`
- If we want to allocate space for `n` things
- Multiple `sizeof()` by `n`
- Gives us the number of bytes to allocate
- So an array of `42 ints`:

```
int *p = malloc(42 * sizeof(int));
```

Solving our pictures

- Don't decide array size at compile-time
- Create it at run-time using `malloc()`
- But how big should it be?
- Four options…

Size of Path Array

- Option One — read through the file twice
- First time through, count lines until `stop`
- Then allocate memory using `malloc`
- Then use `fseek` to go back to the beginning and read again
- Problem — might be reading from something that can't be seeked (e.g `stdin`)

Size of Path Array

- Option Two — copy data if array gets full
- Allocate the array of a certain size
- If we fill it, allocate a new larger array
- And copy the data over using `memcpy`
- Works, but can leave memory fragmented and slows program while copying

Size of Path Array

- Option three — Cheat…
- Rather than putting stop at the end
- Make the first line contain the number of elements
- Read that
- Use value to allocate the array

Size of Path Array

- Option Four — Don’t use an Array
- The problem we have is that an array’s size is fixed when created
- If it needs to grow, we need to create a new one and copy the data
- So use a different data structure that doesn’t have this limitation, such as…

The Linked List…