SYS2/CCCP: The C Memory Model, Pointers and References

For Python and Java Programmers

Dr David Griffin

A Confession

Right at the beginning of the course, these slides said C has 2 fundamental data types

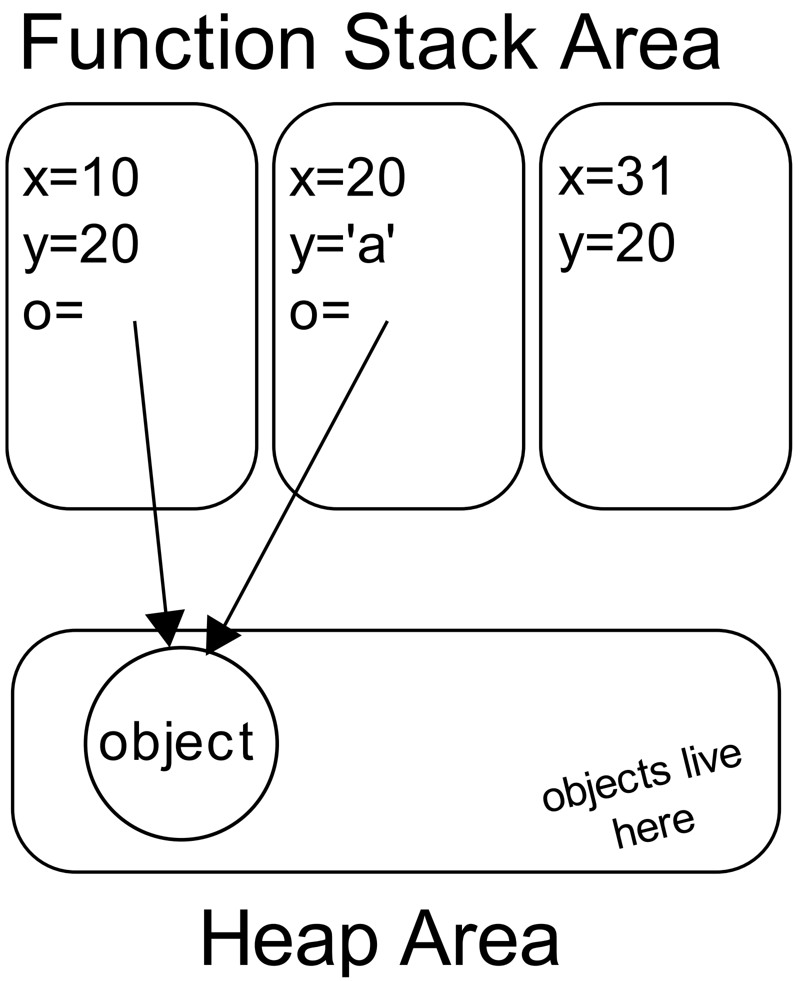
That was a lie

C has 3 fundamental types

- Ints
- Floats
- Pointers

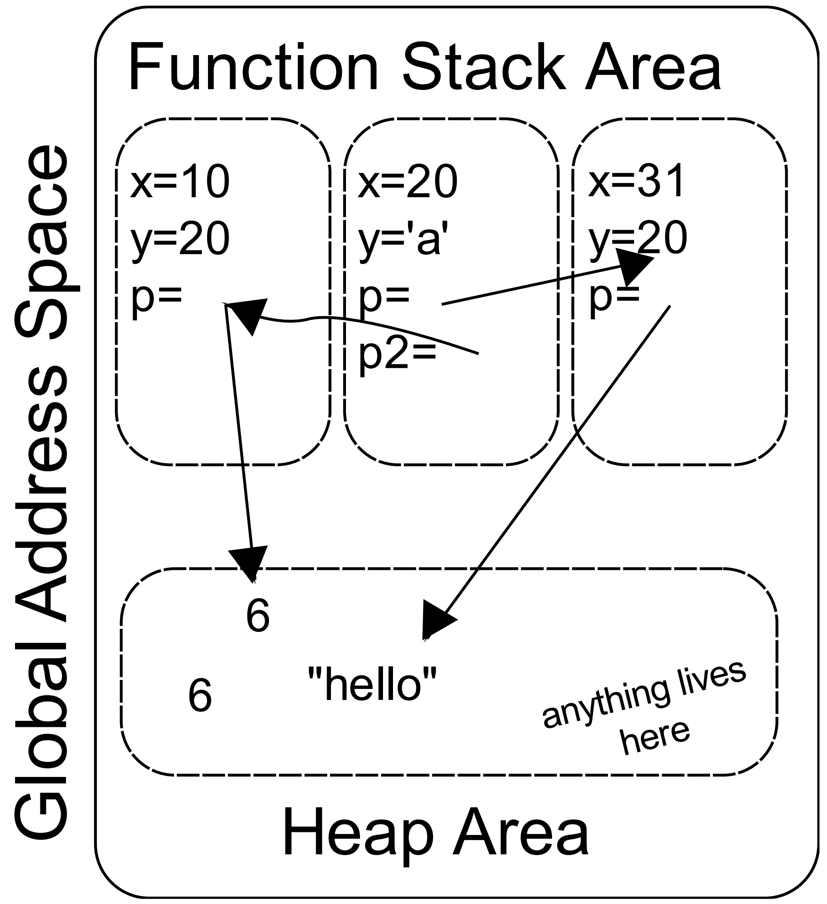
Pointers can be quite tricky to understand, so they were left off that list until we'd seen them used a few times

The Java Memory Model



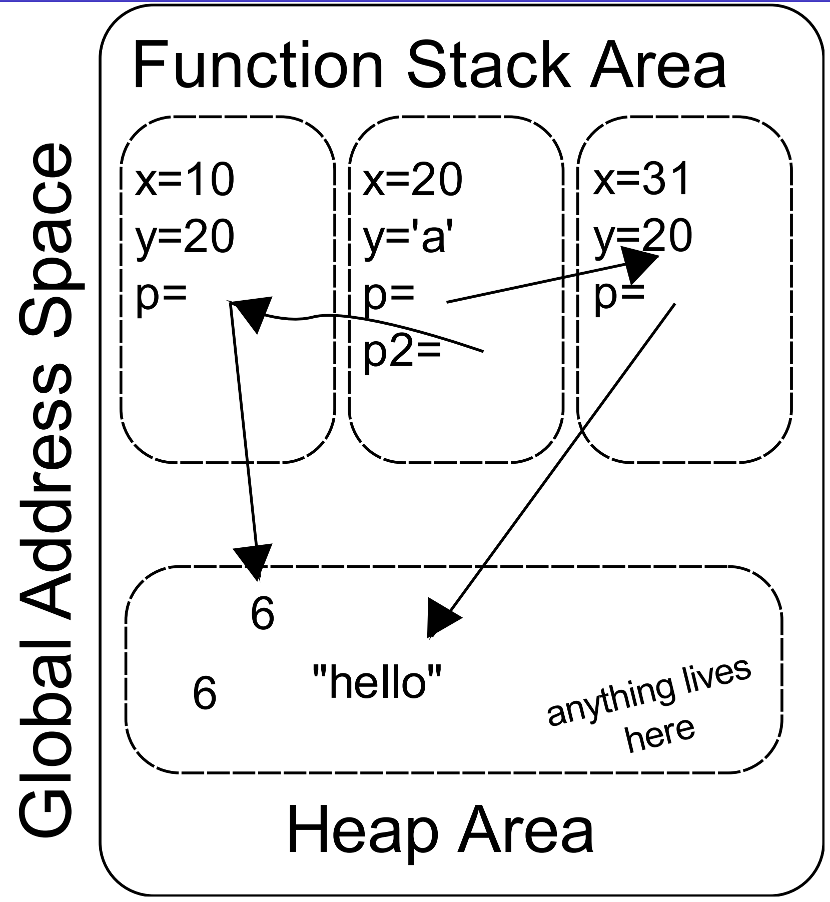
- Variables are allocated in either the stack, or the heap, and they shall never mix
- Ints/Strings other types will be on the stack if inside a function
- The “true” value of Java Objects is always on the heap and local references can be created
- Java uses Pass-By-Value for most variables which are not Java objects
    - Implies that if you pass a string to a function, that string will remain the same in the calling function
- For objects, Java still uses Pass-By-Value, but it passes a reference
    - So the function follows the reference and can modify the object

The C Memory Model



- C has a stack and a heap, like Java, but everything resides in a *global address space*
- If you have the address of a variable, you can read/write to it
  - So if you give the address of a variable on the stack to a function, it can modify that variable
  - This is what the & (reference) operator does – it gives the address of a variable, and it's how scanf writes user input to variables
- This means that in C any variable can be passed-by-value, and any variable can be passed-by-reference
- In C, memory address variables are called *pointers*

Pointers



A pointer is created by appending * to the type of the variable during its declaration

- `int* x; // pointer to an int`
- `char* y; // pointer to a char`

`sizeof(x) == sizeof(y)` for any two pointers, as the size of a memory address is constant on one machine

- sizeof doesn't follow the pointer to get its value

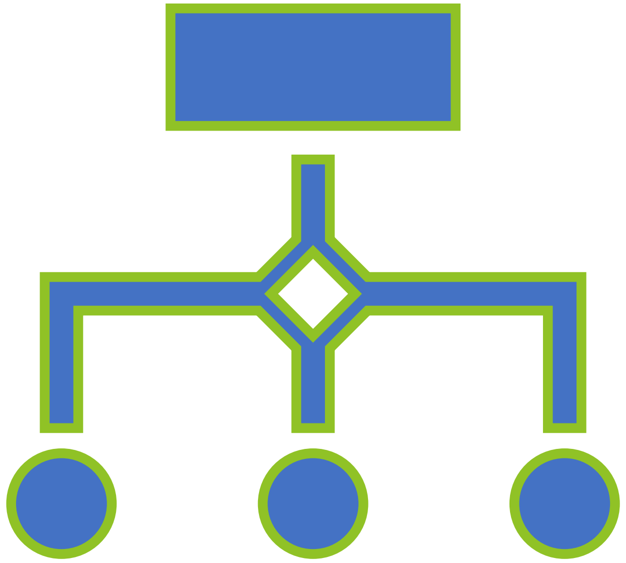
To get the value of a pointer, use the `*` operator or the `[0]` subscript. This is called *dereferencing a pointer*.

- `(*x == 5)   /* true if the value x points to is 5 */`
- `(x[0] == 5) // alternate syntax`

Arrays and Pointers

- It might seem odd that `[0]` can be used to dereference a pointer
- BUT an array is just a slightly fancier pointer
  - The only difference is that `sizeof` can determine the size of an array
- If you look in C documentation you'll find that I've been lying about the function definitions for some functions
  - I said `strlen` was `int strlen(char[] str);`
  - It's actually `int strlen(const char* str);`
- This is why if you pass a function an array, it can write into it without being "given" the array using & - because the array was *always* just a memory address

Pointers and const



- If you pass a pointer into a function, you're giving that function the power to modify the variable that is being pointed at
- However, if you want to pass an array into a function, you must give it the pointer
  - As that's all an array is
- A lot of the time, you might not even want to modify the array, and doing so would be a bug
- `const` pointers, like in `strlen`, are the solution
- With a `const` pointer, a function declares that it will not modify the data the pointer points to
  - Which lets your compiler catch bugs

Assigning to Pointers

A pointer is a memory address along with some type information for what the data at that memory address represents

It can be assigned to like any other variable

You can make a pointer point to a variable by using the `&` operator to extract the memory address

```
int x = 4;
int* x_ptr = &x;
```

DO NOT dereference a pointer that hasn't been set to point at something

Best case scenario: You dereference a null pointer and the program crashes

Worst case scenario: ??? Who knows. If the pointer hasn't been initialised at all, anything could happen.

Compilers do their best to warn you about this

Pointer Arithmetic

- You can add and subtract numbers from pointers
- C is aware of the type of the pointer, so it will add the correct number of bytes to move by the given number of elements
  - `int* x; x + 1;` moves the `sizeof(int)` bytes (normally 4)
- Only do pointer arithmetic when you know the layout of allocated memory

```
// ints are 4 bytes

int arr[3] = {1, 2, 3};

int* ptr = arr;


if(*(ptr + 2) == 3) {

  printf("Pointer
arithmetic works")

}
```

void pointers

`void*` are the one place where you can type a variable void

This works because the *actual* variable is a memory address. A void pointer just doesn't know what it points at.

Used to indicate either the type is unknown, or the type doesn't matter

- For example, in `fread`/`fwrite`

Casting

- Casts are important when handling void pointers, but they can do other things too.
- Casting is the process of converting a variable to another type
- Casts are written as prefixing the type in brackets
  - `int x = (int) y; // explicitly force y to be interpreted as an int even if it is not an int`
  - `int* z = (int*) &void; // force the address of void into an int pointer`
- When casting pointers, no conversion of the data happens. You change the way the data is looked at.
  - `int x = 20;`
  - `char * c = (char*) &x;  // c[0] = 1st byte of x, c[1] = 2nd byte of x…`
- Every pointer can be thought of as a subclass of a void pointer, so you don't have to cast to void*
  - But you do have to cast the other way to retrieve data

Unions

- Want to use memory manipulation to get multiple views onto some data?
- Use Unions instead
- Syntax is like a struct, but all variables go to the same memory
- A union is the size of the largest element, so it can't ever save memory
- More useful if you use unions of structs than basic data types.

```
union intfloat {
 int x;
 float y;
} if;

union maybesigned {
  int signed;
  unsigned int unsigned;
};
```

Unions

- Can use a Union to access the individual bytes of a variable
- What does this program print?
  - 0
  - 1

```
union intbytes {
  int x;
  unsigned char[4] s;
} ib;
int main(){
  ib.x = 1;
  char c = ib.x[0];
  printf("%d", &c);
}
```

Unions

- Can use a Union to access the individual bytes of a variable
- What does this program print?
  - 0 – unlikely
  - 1 – probably
- Depends on if the computer is big or little endian
- Most computers (x86, x64, most ARM) are little endian

```
union intbytes {
  int x;
  unsigned char[4] s;
} ib;
int main(){
  ib.x = 1;
  char c = ib.x[0];
  printf("%d", &c);
}
```

Pointers and IO

C will let you write a pointer to a binary file

If you do this, congratulations! C won't dereference the pointer, so the data won't get saved. Just a memory address, that probably won't exist when the data is loaded.

Instead, you must follow pointers manually to make sure you save everything.

Care must be taken with pointers inside structs

Pointers and Structs

These work as you would expect

However, if you have a pointer to a struct there is a special syntax for accessing elements of that struct

As `*(struct_ptr).element` is a bit confusing to read

`struct_ptr->element` dereferences the struct pointer and accesses the element

Null Pointers

- A pointer which doesn’t point to anything
- These point at the memory address 0
    - So in C Truthiness, a Null Pointer is False
- Very useful for indicating that there’s nothing there
- But if there’s a chance your pointer might be Null, you should check before dereferencing the pointer.

The C Heap

- Variables created inside of functions allocate the memory to hold that variable on the stack
- The stack has finite size, so large amounts of data should be allocated on the heap
- Also, on the heap we can dynamically allocate memory
  - i.e. we don't have to keep the data in a variable and it will still persist – we just need to keep a pointer to the memory
  - This pointer might also be in a heap allocated data structure, allowing us to have dynamically sized data structures
- If data is on the heap, memory will be set aside for that data until it is manually deallocated
  - Unlike in Python and Java, with use Garbage Collection to release memory
  - Note that if we lose all references to the data then we won't be able to deallocate the memory. This is called a memory leak and is a bad thing.

Allocating Memory on the Heap

- `void* malloc(int size);`
  - size = number of bytes to allocate
- `void* calloc(int n, int size);`
  - n = number of elements to allocate
  - size = size of one element (from `sizeof(type)`)
- Returns a `void*` - you should always use a cast to convert this void* to a pointer of your data type
- `calloc` initialises the allocated memory to 0
- `malloc` does not do any initialisation and so is faster, but your code must initialise the memory with your own data
- It's generally preferable to use `calloc`, as it makes bugs more predictable

Releasing memory

- `free(void* ptr)`
- Releases the memory pointed to by `ptr` back to the heap
- Only works with pointers created by `malloc/calloc`

Putting it All Together: A Dynamic Linked List

- In C, all of our data structures have had fixed sizes
- Using Structs, Pointers and the heap, we can create a dynamically sized data structure
    - This example is a linked list of integers
- We can read this list by following the chain of next pointers
- If we see a next pointer which is Null (0), then we've reached the end of the list
- Linked lists like this have linear traversal time, but constant insertion/deletion time

```
struct list_element {
          struct list_element* next;
          int content;
};


struct list_element* next;
struct list_element* current;
current = calloc(1, sizeof(struct list_element));
for (int x = 0; x < 10; x++){
  next = (struct list element*) calloc(1,
sizeof(struct list_element));
  current->next = next;
  current->content = x;
  current = next;
};
```

Conclusions

Everyone here should have learned about the

The C Memory Model	Pointers and References	How to allocate/deallocate memory on the heap	Why you might want to do this

This completes the introduction to all the basic bits of C. The next lecture will pull all of this together into a concrete example.

Assignment

This lectures assignment looks at creating and using pointers for stack variables

And then variables on the heap

And then extending the dynamic linked list that was used as an example here

This should get you familiar with pointers in C, which are the most significant difference between C and Java or Python