CS 261 - Summer 2011

An Introduction To C

Why C?
Because...
You have learned Java
A very high level language
Which is ok for the median
But makes it easier to do high level
Now you need to learn
A low level language
That some of the best fun and other languages
And lets you more flexibility

What you lose
Choose between easy/complex
Some other things
Why C?

Because...

You have learned Java
A relatively high level language
Which restricts your access to memory
But makes it easy to deal with objects
Because...

You have learned Java
A relatively high level language
Which restricts your access to memory
But makes it easy to deal with objects

Now you need to learn
A low level language
That serves as the basis for many other languages
And lets you mess with memory
You have learned Java
A relatively high level language
Which restricts your access to memory
But makes it easy to deal with objects

Now you need to learn
A low level language
That serves at the basis for many other languages
And lets you mess with memory
(For better or for worse)
What you lose

Classes, inheritance and polymorphism among other things

Instead we use functions to encapsulate our code and arrays and structures to
What you lose

Classes, inheritance and polymorphism among other things

Instead we use functions to encapsulate our code

and arrays and structures to keep track of our data
So, About These Functions

They look like Java's methods but they do not exist in a class

```python
return-type function-name (arguments)
{
    declarations;  /* must come first!! */
    function-body;
}
```

Variables in here are local
Variables out here (not in a function) are **Global**

(Just like this huge font, you should try to use global variables as little as possible)
When you are passing things to functions they are passed by value, meaning you are passing a copy of the thing.

So if you pass something to a function and change it...
So if you pass something to a function and change it, you only changed a copy. The original data remains unchanged.
When you are passing things to functions they are passed by value, meaning you are passing a copy of the thing.

So if you pass something to a function and change it.

You only changed a copy. The original remains unchanged.
Dealing with Data

In C we will use structs to keep track of our data.

A struct is not unlike a class with only public data fields and no methods.
will use structs to keep track data.

A struct is not unlike a class with only public data fields and no methods

When you declare a variable you must use the struct keyword

struct arrayBag foo;

No constructors... do it all yourself.

Then to access things you use the dot notation

myStruct.myField;

(but not always, more in the pointers section)
Pointers

Remember how I said to use this font size sparingly? This is important enough to use it.
Remember how I said
enough to use it.

Unsurprisingly pointers point to things

Im a pointer to the next slide.

(it's over there)
There is a large, and very important, difference between the pointer and the thing being pointed to.

You can change BOTH and they do DIFFERENT THINGS
There is a large, and very important, difference between the pointer and the thing being pointed to.

You can change BOTH and they do DIFFERENT THINGS

Consider a sign
You can change BOTH and they do DIFFERENT THINGS

Consider a sign

that tells me where Kelly is

I could repaint the sign and have it show me.
that tells me where Kelly is

I could repaint the sign and have it show me the way to the MU. I am changing WHAT it is pointing at

OR
the sign and have it show me MU. I am changing WHAT it

OR

I can change Kelly, say I add a bunch more tables and chairs to it, the sign still points to Kelly, but now I can actually find a table to study at.
I could repaint the sign and have it show me the way to the MU. I am changing WHAT it is pointing at.

OR

I can change Kelly, say I add a bunch more tables and chairs to it, the sign still points to Kelly, but now I can actually find a table to study at.

Changing the pointer to point at something else entirely. If I follow the sign, I won't get to Kelly.

Changing the thing the pointer is point at. The pointer still takes me to Kelly. But now I can find a seat.
Syntax

int x;

int * y;
Syntax

A number

int x;

int * y;

A sign that shows me where a number is
If I do this:
\[ x = 5; \]
Everything is fine. Nothing is ruined.
is fine. Nothing is ruined.

But what if I do this:

\[ y = 5; \quad // \text{remember int } * \ y \]
Nothing is fine. Nothing is ruined.

But what if I do this:

```
y = 5;  //remember int * y
```

Everything is NOT fine and we may have just ruined everything.

(Good job)
The star "*"

```c
int * px;  // A pointer to an int

*px = 5;
( This is an OK thing to do)
```

This star is dereferencing px.
(It's like we followed the sign and we are doing work on Kelly)
and symbol amfers

The amphi stand "&"

double a;
double * pa;
a = 5;
pa = &a;

Now points at a. The address of a.
( location in memory)
The arrow "->"

```c
struct myStruct a;
struct myStruct * p;
p = &a;
a.count = 7;
p -> count = 7;
```

These do the same thing. The top operates directly on the struct. The bottom follows the sign to the struct the operates on it.

This is the same as (*p).count;
Great, now I can break things in more ways... Why is this good again?

Remember
Great, now I can break things in more ways... Why is this good again?

Remember

We don't want to copy BIG things.

(I'm a pointer to a big thing, pass me instead)

So we pass by reference.
(Im a pointer to a big thing, pass me instead)
So we pass by reference.

```c
void myFunction(struct myBigStruct * p) {
    p->someValue = 3.14;
}
```
We don't want to copy big things.

So we pass by reference.

```c
void myFunction(struct myBigStruct * p)
{
    p->someValue = 3.14;
}
```

We are working with a pointer to myBigStruct.

It turns out this is the only way to modify

We don't want to copy big things.

So we pass by reference.

```c
void myFunction(struct myBigStruct * p) {
    p->someValue = 3.14;
}
```

We are working with a pointer to myBigStruct.

It turns out this is the only way to modify something passed into a function. Otherwise we are just working on a copy and not changing the original.

An array is just a pointer and some memory. Don't worry about this for now.
An array is just a pointer and some memory.

Don't worry about this for now. Just know that it is impossible to pass an array by value.
Memory

In C memory is your responsibility.

malloc giveth and
free taketh away
When we want to give something memory we need to malloc it.

```c
struct myStruct * p = (struct myStruct *) malloc(sizeof(struct myStruct));
```

A pointer to a struct

```c
assert (p != 0)
```

How much memory to allocate

This makes sure you actually got your memory.

```c
free(p);
```
assert (p != 0)

free(p);

Much easier
assert (p != 0)

This makes sure you actually got your memory.

free(p);

Much easier

Match your mallocs and frees

Otherwise you get a memory leak because your program won't give back its memory.

The preprocessor

A Preprocessor scans C programs before they are compiled. You have no choice, it's a part of all.
The preprocessor

A Preprocessor scans C programs before they are compiled. Used to make symbolic constants, conditionally include code

```c
#define max 423
#if (max < 3)
```
# define max 423
# if (max < 3)
...
# endif
But what if I have a bunch of files?

Ensuring declarations seen only one

/* interface file for foo.h */
#define BigComplexName
#define BigComplexName
...
#define endif

/* that way, 2nd time does nothing */