CS 261 – We are expected to be familiar with C. It’s not that different from C++, but there are differences.

Why do we use C? It’s an easy imperative language. So, we can focus on the important concepts now and just go forward with that.

Avoid baggage of the OOP baggage – classes, inheritance, polymorphism, and function overloading. These muddy the waters. And we also lose the garbage collection and reference types and we have to do memory management and do pointers. The pointers are the toughest thing. Pointers are mostly harmless, and are fairly easy.

Every program in C has a main

void main (int argc, char \*\*argv) { } – these are command line arguments.

Command Line Arguments:

Argc = 3; argv[0] = compueRectangleArea, arg = 10, 22

Long arrSum(int arr[], unsigned int) { unsigned int I; long sum = 0; … in the new C, variables can be declared anywhere. However, we should put our variables in the beginning

Struct Gate {

Int type;

Struct Gate \*left; struct Gate \*right;

We don’t have classes in the new C environment.

We define functions and in order to use the structure and function, based on an imperative or procedural basis. We have to use the parameters. To print a function and variable call, we might have to do some extra work. We do some extra method work and then pass the structure around to operate on it. It’s not all that different, just a bit more involved.

Global:

Variable declared outside of any function (DON’T USE THESE!) They can cause side effects and overwrites which are annoying as hell.

LOCAL:

Variable declared inside of function. (In ANSI C, local variable must be listed first.)

See the slide for the code.

POINTER REVIEW:

C is pass by value – no pass by reference. But pointers simulate this.

Void foo (int a)

{

A = a+2;

}

Void main (….)

{

Int b = 6;

Foo(b);

Printf(“b = %d\n”, b);

}

A pointer is simply a value that can refer to another location in memory. In other words, its value is an address in memoiry! Declaring a pointer(\*) – use the star.

int \*pVal.

pVal = 0; just use the assignment operator

Get the address or pointer of a stored value(&), and int a = 5; pVal = &a; and dereferencing is pointer. \*pVal = 4; /\*Assignment\*/

Int b = \*pVal;

Double \*ptr;

Double pi, e

Ptr= &pi;

And we can look at the address. See the slide.

23 is the address of pi. And the ptrs are arrows. PTR is pi, and the arc of the arrow ending in Pi.

23 will then get value 3.14159; and the dereference drops the pointer, assigns the value and clears the pointer for new information. A ptr gets a new address of e, and then re-assigns the pointer. C has placeholders and we need to use a format specifier.

Remember %p for pointers, %g for doubles. And there is comma and each of the variables. And replace those placeholders. In the particular case, we see the 0x203, and the 2.718, 3.141, 2.718.

Pass by reference and the vectors and addresses. It’s a common idiom. However, this is not really used in C so much as it is in higher OOP. But it is used, so be prepared.

And this is a common practice in the course.

Dynamic Memory Allocation is struct Gate \*p; and p->type = 4; And the P is a pointer to struct gate, and the operating system allocates enough space for the pointer itself. And the use in the slide will cause a seg fault. YOU NEED TO DECLARE A TYPE.

Static allocations, and if we are going to be static at compile, then we can just use reference and statically allocate space. And then, in terms of dynamic allocation, we have to do our own memory allocation and deallocation in C.

UGH! This is a pain. So, we give it a malloc(sizeof(struct whatever));

We also want to use the assert keyterm and make sure we get something that doesn’t end up non-initialized.

Remember:

NO NEW OPERATION

USE malloc(num-of-bytes)

Malloc always returns a pointer

And Use Sizeof for safety.

If the OS cannot handle it sizewise, it’ll pass a null.

Remember to use assert:

#include < assert.h >

Arrays are a bit more fun and Arrays are pointers in C.

Void foo(double d[]) { d[0] = 3.14159 }

Double data[4]; and data[0] = 42.0

Foo(data);

Printf(what is data[0]? %g”, data[0]);

Before C99, there was no Boolean type. And prior to this, there was integer values for Booleans. This was pointers and such. We can use that to simulate the lines of the code. We did the same thing in CS271.

Compilation process - preprocessors remove comments, replaces directives with the machine code.

The compiler turns it into machine level code (so it looks like assembly.)

Assembler turns it into machine language, and the linker creates the final executable file. And then we see the libraries and .o files here. .exe.

Separation of interface and implementation – the C source files are the .h and .c files. Interface files (.h) are the header and have the function prototypes.

Implementation files (\*.c) have implementation source.

Prototypes is function header but no body (promise that code will be linked in later) – and the function prototypes need to be terminated with a semi-colon. It allows us to hide implementation.

We use things like #ifndef to make sure that things are seen only once. Thus we don’t repeatedly include things. This ensures that header files are included once. And we don’t want experience errors.

As long as a compiler can link to the .o files, it will pulls the file in.

Lecture on structures – we see a structure, and it has an id and character.

See the static structure – it’s got a lot of code, but is effective and straightforward.

There is a lot of duplicate code – which is problematic as it creates a lot of extra work. Wherever we can find ways to simplify things, then we can go forward and use functions. This reduces some stuff.

But the plus-ultra will be the use of pointers and such.