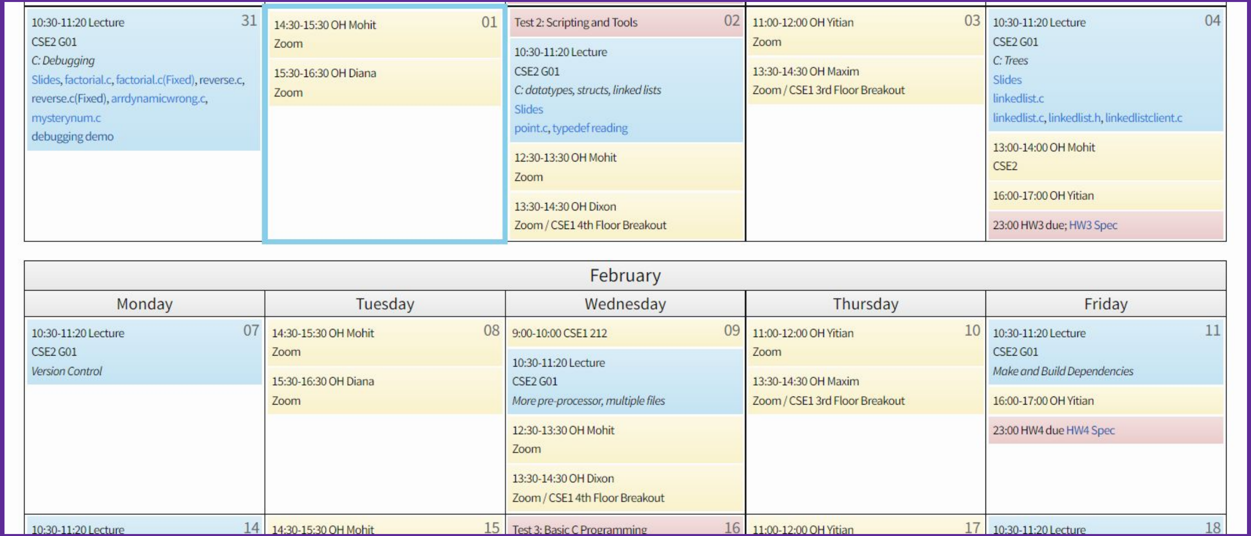
CSE 374 Lecture 13

Typedefs, structs, data structures

Test Due Tonight



31 10:30-11:20 Lecture CSE2 G01 *C: Debugging* Slides, factorial.c, factorial.c(Fixed), reverse.c, reverse.c(Fixed), arrdynamicwrong.c, mysterynum.c debugging demo	01 14:30-15:30 OH Mohit Zoom 15:30-16:30 OH Diana Zoom	02 Test 2: Scripting and Tools 10:30-11:20 Lecture CSE2 G01 *C: datatypes, structs, linked lists* Slides point.c, typedef reading 12:30-13:30 OH Mohit Zoom 13:30-14:30 OH Dixon Zoom / CSE1 4th Floor Breakout	03 11:00-12:00 OH Yitian Zoom 13:30-14:30 OH Maxim Zoom / CSE1 3rd Floor Breakout	04 10:30-11:20 Lecture CSE2 G01 *C: Trees* Slides linkedlist.c linkedlist.c, linkedlist.h, linkedlistclient.c 13:00-14:00 OH Mohit CSE2 16:00-17:00 OH Yitian 23:00 HW3 due; HW3 Spec

February

Monday	Tuesday	Wednesday	Thursday	Friday
07 10:30-11:20 Lecture CSE2 G01 *Version Control*	08 14:30-15:30 OH Mohit Zoom 15:30-16:30 OH Diana Zoom	09 9:00-10:00 CSE1 212 10:30-11:20 Lecture CSE2 G01 *More pre-processor, multiple files* 12:30-13:30 OH Mohit Zoom 13:30-14:30 OH Dixon Zoom / CSE1 4th Floor Breakout	10 11:00-12:00 OH Yitian Zoom 13:30-14:30 OH Maxim Zoom / CSE1 3rd Floor Breakout	11 10:30-11:20 Lecture CSE2 G01 *Make and Build Dependencies* 16:00-17:00 OH Yitian 23:00 HW4 due HW4 Spec
14 10:30-11:20 Lecture	15 14:30-15:30 OH Mohit	16 Test 3: Basic C Programming	17 11:00-12:00 OH Yitian	18 10:30-11:20 Lecture

Datatypes in C

- Void: a placeholder
- Numbers: int, short, long, double, float, ... (signed, unsigned)
- char: really a very short int (1 byte) interpreted as a printable character
- Pointers (T*): int, char, double, char*, ...
- Arrays (T[]): int arr[], char arr[], char* arr[], …
  - Implicit promotion to pointer when passed as an argument to a function or returned from a function
- Booleans? Not defined in C
  - 0 or NULL is always considered "false" and anything else is true
- Advanced: Union T, Enum E, Function pointers

Typedef

Not really a new type - just creating an alias for an existing type

typedef <type> <name>;

In C, strings are "char*", but if I wanted to actually provide the name "string", I could!

```
typedef char* string;
int main(int argc, string *argv) {
    string s = "hello, world!";
    printf("%s\n", s);
}
```

Type-casting *(converting one type to another)*

- **Syntax: `(t) e` where `t` is a type and `e` is an expression (sameas Java)**
- **If e is a numeric type and t is a numeric type, this is a conversion**
  - **To wider type, get same value**
  - **To narrower type, may not (will get mod)**
  - **From floating-point to integer, will round (may overflow)**
  - **From integer to floating-point, may round (but int to double is exact on most machines)**

```
main() {
    int sum = 17, count = 5;
    double mean;
    mean = (double) sum / count;
    printf("Value of mean : %f\n", mean );
}
```

Implicit casting

- When necessary the compiler automatically converts from one type to another (more general) type
  - Promotes to integers, then to larger integers, then to floating point
  - During arithmetic
  - R-value converted to L-value

For details:
https://www.oreilly.com/library/view/c-in-a/0596006977/ch04.html

Pointer-casting

If `e` has type `t1*`, then `(t2*)e` is a (pointer) cast.

- You still have the same pointer (index into the address space).
  - Nothing "happens" at run-time.
- Just "getting around" the type system - can write any bits anywhere you want.

```
void evil(int **p, int x) {
    int *q = (int*)p;
    *q = x; }
void f(int **p) {
    evil(p,345);
    **p = 17; // writes 17 to address 345 Best case - crash
}
```

Structs

- New datatypes
  - a record, containing one or more fields
  - Stored adjacently in memory
- Like Java class, except no methods
- Access a field S.f
- If S* Ps then *Ps.f
  - shortcut S->f

```
struct person_info {

    char * name;

    int age;

}
```

Struct-tags

Has type struct person_info

‘Person_info’ is a struct tag, not a type

Can use typedef to rename

```
struct person_info {

    char * name;

    int age;

}
```

Struct-tags

```
typdef struct person_info person_info;
```

Has type struct person_info

‘Person_info’ is a struct tag, not a type

Can use typedef to rename

```
struct person_info {
    char * name;
    int age;
}
```

Struct-tags

Has type struct person_info

‘Person_info’ is a struct tag, not a type

Can use typedef to rename

```
typdef struct person_info {
   char * name;
   int age;
} person_info;
```

Parameters / Arguments

Reminder:

Function parameters initialized with a copy of corresponding argument

If the argument is a pointer, the parameter value will point to the same thing, of course

Arrays are passed as pointers (remember?)

(Demo: point.c)

Even with a struct a copy is created

Since this won't change the original struct, it is more common to use a pointer to the struct

Avoids copying large objects

Allows manipulation of original object (can write functions like Java methods)

But, sometimes, want to pass-by-value. *THINK!!*

Typedef struct ex.

```
typedef int int32;  // use int32 for portability
typedef struct point {  // type tag optional (sortof)
   int32 x, y;
} Point2d;   // Point2d is synonym for struct point
typedef Point2d * ptptr;  // pointer to Point2D
Point2d p; // var declaration
ptptr ptlist; // declares pointer
```