# Enum, Typedef, Structures and Unions

## CS 113: Introduction to C

Instructor: Saikat Guha

Cornell University

Spring 2007, Lecture 8

# Complex Types

- Different sized integers
  - `int`: machine-dependent
  - `char`: 8-bits
  - `int8_t`: 8-bits signed
  - `int16_t`: 16-bits signed
  - `int32_t`: 32-bits signed
  - `int64_t`: 64-bits signed
  - `uint8_t, uint32_t, ...`: unsigned
- Floating point numbers
  - `float`: 32-bits
  - `double`: 64-bits

# Complex Types

- Enumerations
  (user-defined `weekday`: `sunday`, `monday`, `...`)
- Structures (user-defined combinations of other types)
- Union (same data, multiple interpretations)
- Function types (and function pointers)
- Arrays and Pointers of the above

# Enumerations

```
enum days {mon, tue, wed, thu, fri, sat, sun};
// Same as:
// #define mon 0
// #define tue 1
// ...
// #define sun 6
```

```
enum days {mon=3, tue=8, wed, thu, fri, sat, sun};
// Same as:
// #define mon 3
// #define tue 8
// ...
// #define sun 13
```

# Enumerations

```
enum days day;
// Same as:    int day;

for(day = mon; day <= sun; day++) {
    if (day == sun) {
        printf("Sun\n");
    } else {
        printf("day = %d\n", day);
    }
}
```

# Enumerations

- Basically integers
- Can use in expressions like ints
- Makes code easier to read
- Cannot get string equiv.
- caution: `day++` will always add 1 even if enum values aren't contiguous.

# Structures

```
struct mystruct {
    char name[32];
    int age;
    char *addr;
};
```

# Structures

```
void foo(void) {
    struct mystruct person;                 // uninitialized

    struct mystruct person2 = {             // initialization
        .name = {'f','o','o','\0'},
        .age = 22,
        .addr = NULL
    };

                                            // struct pointer
    struct mystruct *pptr =
     (struct mystruct *)malloc(sizeof(struct mystruct));

    ...
```

# Structures

```
struct mystruct {
    char name[32];
    int age;
    char *addr;
};
```

```
...

person.age = 10;                    // direct access
person.addr = (char *)malloc(64);

pptr->age = 24;                     // indirect access
strncpy(pptr->name,"foo",32);       // through pointer
pptr->addr = NULL;

...
```

# Structures

- Container for related data
- Chunks of memory; syntactic sugar for easy access.
- May have empty gaps between members

  (see `#pragma packed`)
- You'll need to use for linked-list assignment

# Unions

```
union myunion {
    int x;
    struct {
        char b1;
        char b2;
        char b3;
        char b4;
    } b;
};

union myunion num;

num.x = 1000;
num.b.b1 = 5;
```

# Unions

- Same memory space interpreted as multiple types
- Useful for plugins, sclicing network packets etc.

# Function Pointers

```
int min(int a, int b);
int max(int a, int b);

int foo(int do_min) {
    int (*func)(int,int);          // declaring func. ptr

    if (do_min)
        func = min;
    else
        func = max;

    return func(10,20);            // indirect call
}
```

# Function Poninters

- Points to a function
- Has a *-type of the function it points to

# Renaming Types

- Complex types inconvenient to write over and over
  - `(enum day *)malloc(sizeof(enum day)`
  - `(struct foo *)malloc(sizeof(struct foo)`
  - `(union bar *)malloc(sizeof(union bar)`
  - `(int (*)(int,int))((void *)min)`

## Renaming Types

```
typedef long old type newtype
```

```
typedef enum day day_t;
typedef struct foo foo_t;
typedef int (fptr_t)(int,int);
```