Topics in C

A few loose ends…

Pointer Syntax….

- One more piece of syntax…
  - For something you already know how to do!
- Consider a structure…

```
typedef struct Mono {
   int coef;
   int deg;
   struct Mono* next;
} Mono;
```

- And a declaration…

```
Mono* ptr = (Mono*)malloc(sizeof(Mono));
```

- The following three forms are equivalent!

Array Bracket Style	Dereference Style	Arrow Style
`ptr[0].coef = 1;`	`(*ptr).coef = 1;`	`ptr->coef   = 1;`

Union Types

- C supports a special kind of values
  - union
  - Also called (in other languages) sum types or even variants
- Purpose
  - Have the ability for a value to take different form / types
  - Variants are mutually exclusive

Example!

```
typedef union uTag {
   int   iVal;
   float fVal;
   char* sVal;
} UType;

UType  u;
u.iVal = 5;
printf("as an int?   %d\n",u.iVal);
printf("as a float?  %f\n",u.fVal);
printf("as a string? %s\n",u.sVal);
```

```
src (master) $ ./a.out
as an int?   5
as a float?  0.000000
Segmentation fault: 11
```

What is happening?

- A value of union type can be only one of its members!
  - You are either an int
  - or a float
  - or a string!
- But never more than one thing!
  - **This is polymorphism in C.**
- Storage ?
  - Only enough space to hold the largest attribute
  - All attributes occupy the same memory location

Therefore….

- When you do…

```
typedef union uTag {
   int   iVal;
   float fVal;
   char* sVal;
} UType;

UType  u;
u.iVal = 5;
printf(“as an int?   %d\n”,u.iVal);
printf(“as a float?  %f\n”,u.fVal);
printf(“as a string? %s\n”,u.sVal);
```

- You store an integer…
- That you try to interpret as a float …. or as a string!
  - (that does not work!)

Proper usage…

- Use within a structure type
  - With a union
  - With an extra field that tells you the ***kind*** of value in the union!

Example

```
typedef struct tagged_Value {
   int   tag;          /* This is the tag reporting the kind of value*/
   union uTag {        /* all 3 “alternatives” use the same memory   */
      int   iVal;
      float fVal;
      char* sVal;
   } u;
} UType;
#define INTEGER_TAG 0  /* tag to code for an integer */
#define FLOAT_TAG   1  /* tag to code for a float */
#define STRING_TAG  2. /* tag to code for a string */
```

```
void printUType(UType* x) {
   switch(x->tag) {
      case INTEGER_TAG: printf(“%d\n”,x->u.iVal);break;
      case FLOAT_TAG:   printf(“%f\n”,x->u.fVal);break;
      case STRING_TAG:  printf(“%s\n”,x->u.sVal);break;
   }
}
…
UType  x;
x.tag = INTEGER_TAG; /* x holds an integer */
x.u.iVal = 5;        /* x holds the integer 5 */
printUType(&x);      /* Polymorphic printing invoked */
```

Bit Fields

- This is a mechanism to optimize storage inside structures…
- Scenario:
  - Consider that I need to store in a struct fields with:
    - age: a value between 0 and 128
    - gender: (Male/Female/Neutral)
    - Species: (Humans, Rakata, Hutts, Wookies, Jawas,Ewoks,Gungans, Neimoidians)

```
struct Census {
   int age;
   int gender;
   int species;
};
```

12 bytes!
Can we do better?

Bit Fields

- This is a mechanism to optimize storage inside structures…
- Scenario:
  - Consider that I need to store in a struct fields with:

7 Bits • age: a value between 0 and 128

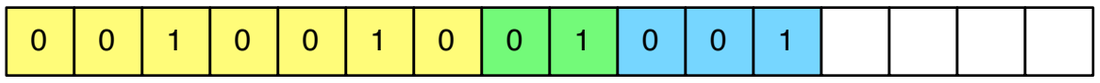
2 Bits • gender: (Male/Female/Neutral)

3 Bits • Species: (Humans, Rakata, Hutts, Wookies, Jawas,Ewoks,Gungans, Neimoidians)

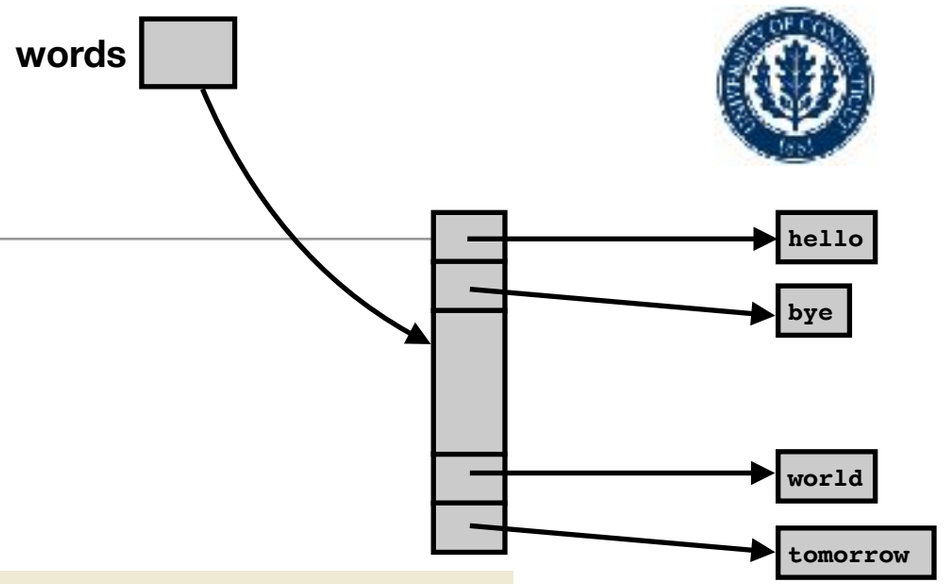
000 001 010 011 100 101 110 111

```
struct Census {
   short age     : 7;
   short gender  : 2;
   short species: 3;
};
```

Total: 12 bits, or 2 bytes!
That's 83% smaller!



Function Pointers…



- You already saw an example!
  - Calling quickSort builtin C function

```
int stringCompare(const void* s1,const void* s2)
{
   return strcmp(*(char**)s1,*(char**)s2);
}

Tree* readDictionary(char* fname)
{
   FILE* f = fopen(fname,"r");
   …
   char** words = (char**)malloc(sizeof(char*)*nbWords);
   …
   qsort(words,nbWords,sizeof(char*),stringCompare);
   …
}
```

What is happening?

- We are passing to qsort a “thing” called **stringCompare**
- **stringCompare** is defined earlier as a function!
- Purpose
  - We have a *generic* quickSort implementation
  - We *genericize* by passing a comparator function adapted to the type of elements in the array
- In practice
  - The qsort implementation calls the comparator to rank elements
  - A lot like comparators in Java!

Understanding qsort?

- Here is the prototype

```
void qsort(void *base,
           size_t nel,
           size_t width,
           int (*compar)(const void *, const void *));
```

- qsort takes…
  - base: the address of the array as an untyped pointer
  - nel: the number of elements in the array
  - width: the size (in byte) of ONE element of the array
  - compar: a **pointer** to a function capable of comparing two values

Implementing “Slow Sort”

```
#include <stdio.h>
#include <stdlib.h>

void ssort(void *base,size_t nel,size_t width,
           int (*compar)(const void *, const void *))
{
   int i,j,k;
   char tmp[width];
   for(i=0;i < nel ;i++) {
      for(j=i;j < nel;j++) {
         if (compar(base+i*width,base+j*width) > 0) {
            char* ei = base + i * width;
            char* ej = base + j * width;
            for(k=0;k<width;k++) tmp[k] = ei[k];
            for(k=0;k<width;k++) ei[k]  = ej[k];
            for(k=0;k<width;k++) ej[k]  = tmp[k];
         }
      }
   }
}
```

Using Slow Sort

```
int compareInt(int* a,int* b)
{
    return *a - *b;
}

int main()
{
    int* t = (int*)malloc(sizeof(int)*32);
    int i;
    for(i=0;i<32;i++)
        t[i] = abs(random()) % 1000;
    for(i=0;i<32;i++)
        printf("t[%d] = %d\n",i,t[i]);
    ssort(t,32,sizeof(int),(int(*)(const void*,const void*))compareInt);
    for(i=0;i<32;i++)
        printf("t[%d] = %d\n",i,t[i]);
    return 0;
}
```

Other uses ?

- You can store pointers to functions anywhere….
- Including in arrays…
- And arrays stored in structures!

You can simulate Object Oriented Languages!

[if curious ask offline!]

Note to the wise….

- Pointers to functions are…

Stateless **Lambdas**!

See?… just like in 1729

You can use them for all sorts of things!

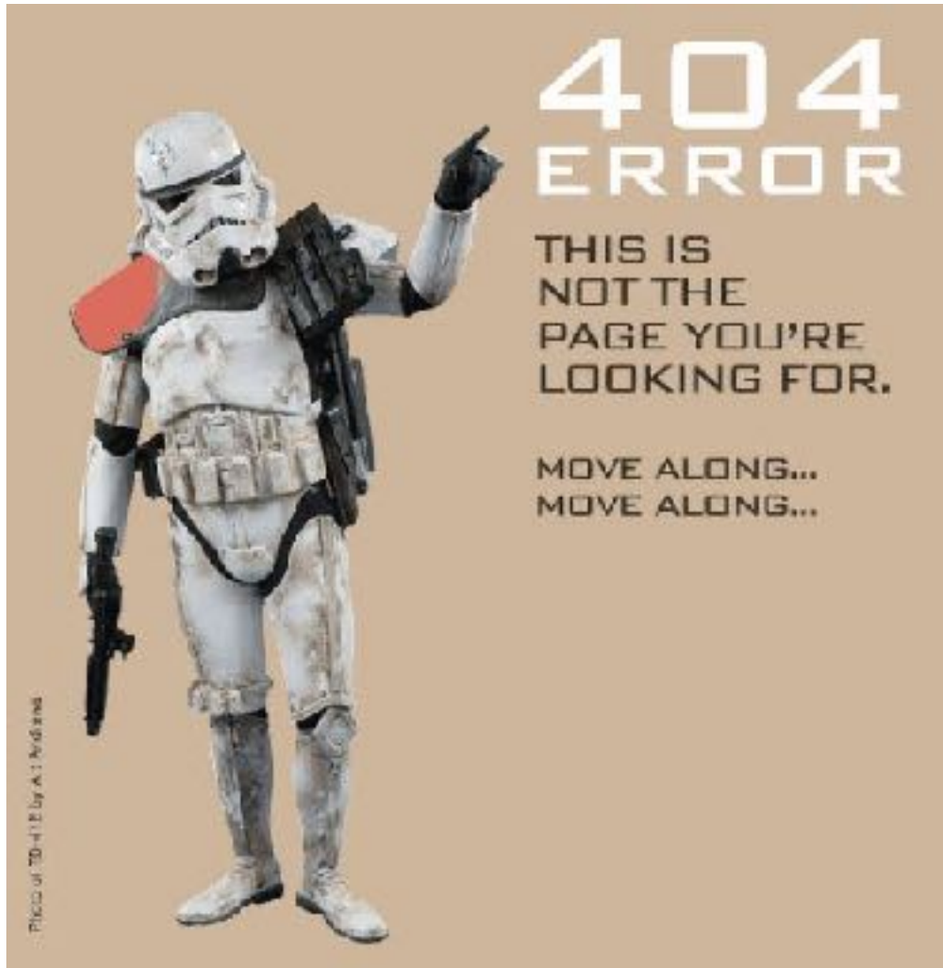
And you *already* know how!

Errors ?

- Remember the “errno” discussion?
  - Most C library function can “fail”
  - When they do, they return a flag reporting failure… (-1)
  - And they set a global variable to report the exact error code

errno

- Check manual page
  - man errno
  - #include <errno.h>



Caveat

- This is not reentrant
- Whenever possible In multithreaded code
  - Avoid functions that set errno
  - Prefer reentrant versions when available
- If there is no way around it, you will need “locking”
  - More on this when we cover threads!