CS 2261: Media Device Architecture - Week 5 - part 2

Overview

- Swap -- just a bit more
- C "Strings"
  - Chars
  - printf / sprintf
- Arrays of character pointers
- C Structs

Question

```
int main() {
   int x = 3;
   int y = 72;
   int *px = &x;
   int *py = &y;
   *px = 7;
   py = px;
   x = 12;
   printf("%d %d\n", *px, *py);
}
```

What is the output?

1. 3 72
2. 72 3
3. 7 12
4. 12 7
5. 3 3
6. 72 72
7. [✓ 12 12]
8. 12 72
9. 72 12

Swap Function

```
void swap(u16 *a, u16 *b){
  u16 temp = *a;
  *a = *b;
  *b = temp;
}
int main() {
  u16 x = 1;
  u16 y = 2;
  swap(&x, &y);
}
```

Swap Pointers Function?

```
int main() {
  u16 arr1[] = {1,2,3};
  u16 arr2[] = {4,5,6,7};
  u16 *p1 = arr1;
  u16 *p2 = arr2;
  swap_pointers(&p1, &p2);  // the addresses of the pointers
  for(i=0; i<4; i++) {
    printf("%d ", p1[i]);  // The printf(ormat) stdio function
  }
  for(i=0; i<3; i++) {
    printf("%d ", p2[i]);
  }
}
// printed output: 4567123
```

What does Swap Pointers Look like?

```
void swap_pointers(int *a, int *b) {   /* Choice 1 */
   int t;
   t = *a;
   *a = *b;
   *b = t;
}
void swap_pointers(int **a, int **b) { /* Choice 2 */
   int *t;
   t = *a;
   *a = *b;
   *b = t;
}
void swap_pointers(int **a, int **b) { /* Choice 3 */
   int **t;
   *t = *a;
   *a = *b;
   *b = *t;
}
```

C Strings

C Strings

"There are no strings in C." -- Bill Leahy

C Strings

"There are no strings in C." -- Bill Leahy

There really is *no string type in C.*

"But… you just used one back there in printf!?"

C Strings

"There are no strings in C." -- Bill Leahy

There really is *<u>no string type in C.</u>*

"But… you just used one back there in printf!?"

Nope. It only *looked* like a string.

Strings

- Many languages include built-in support for Strings

- Typical examples include
  - Basic
  - Scheme/Lisp/Clojure
  - Python
  - Java
  - Probably every one you've ever used before this class.

- "Built-in" here implies behind the scenes details are managed by the compiler and/or the run-time environment

C Strings

- There are no Strings in C.
- There is an agreed upon convention to store string like data structures in arrays of characters.
  - Part of the convention includes terminating strings with a null character.
- In many cases "string" operations in C look just like those found in other languages.
- Do not be deceived. This is C...memory must be managed properly (by you)!

Characters

```
char a = 'a';
char a = 97;
char a = 0x61;
```

These are all the same. Things between single quotes, get replaced by their ASCII encoded number.

Don't do!:

```
char abc = 'abc';
// This is undefined (but compiles...!)
```

String Comparison: Python vs. C

Python:

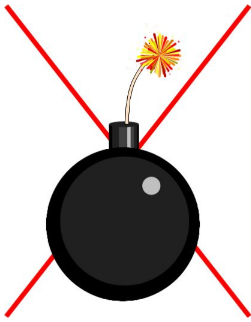
```
a = "Hello "
b = "World!"
a = a + b
print(a)

Output: Hello World!
```

C:

```
char *a = "Hello ";
char *b = "World!";
a = a + b;

printf("%s\n", a);
```



Compiler error: invalid operands to binary + (have ‘char *’ and ‘char *’)

String Comparison: Python vs. C

Python:

```
a = "Hello "
b = "World!"
a = a + b
print(a)

Output: Hello World!
```

C:

```
char *a = "Hello ";
char *b = "World!";
a = a + b;

printf("%s\n", a);
```

Compiler error: invalid operands to binary + (have ‘char *’ and ‘char *’)

```
// second attempt
char a[] = "Hello ";
char b[] = "world";
strcat(a, b);
printf("%s\n", a);
```

Segmentation fault (core dumped)

What is Python doing?

I don't know. Do we care right now?

What is C doing? strcat?

- Following is the declaration for strcat() function:
  - `char *strcat(char *dest, const char *src)`
  - Parameters
    - dest − This is pointer to the destination array, which should contain a C string, and should be large enough to contain the concatenated resulting string.
    - src − This is the string to be appended. This should not overlap the destination.

`const` is a keyword that leads to a type we can't modify. In this case, it's saying the pointer points to something we can't modify.

Better?

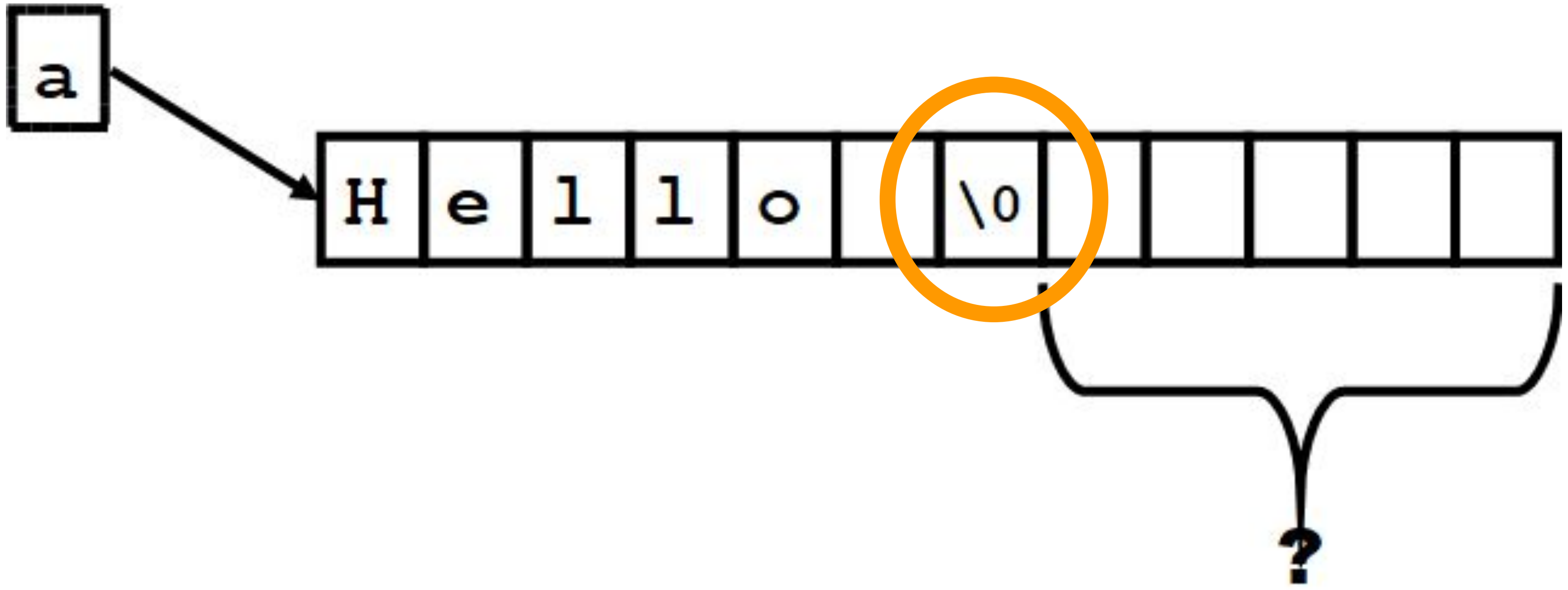
```
char a[12] = "Hello ";
char b[] = "World!";
strcat(a, b);
printf("%s\n", a);
```

```
Output: Hello World!
```

Success!

Not quite

```
char a[12] = "Hello ";
```



Strings

- We implement string-like data structures using character arrays / pointers
  - char s1[10]; /* Definitely not a string! */
    - Static s1 would be pretty close though.
  - char *s2; /* Definitely not a string! */
  - char s3[10] = "foo"; /* Not a string -- closer */
  - char *s4 = "bar"; /* Not a string -- really close */
- What *acts* like a string?
  - A pointer to a character followed by a sequence of characters terminating in a null byte
- char s5[10] = "foo";

```
'f' 'o' 'o' '\0' '??' '??' '??' '??' '??' '??'
```

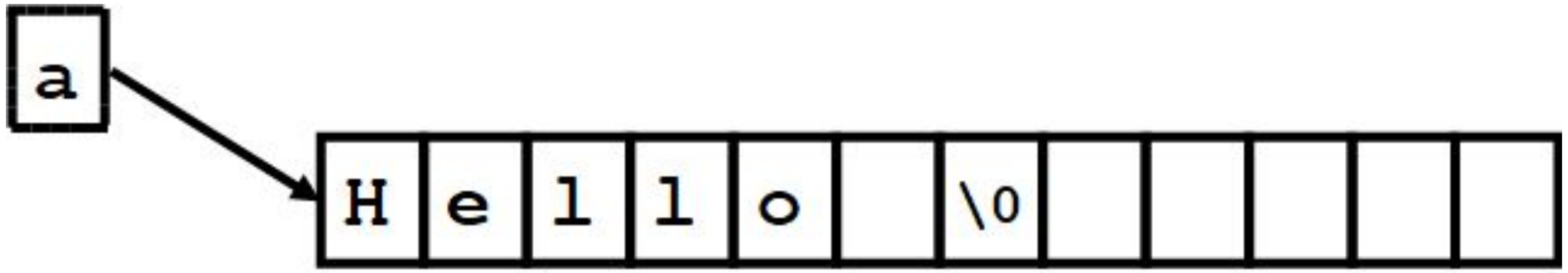
\0

- (char)0 (aka '\0', aka 0b00000000) is the *null character* .
  - Not to be confused with '0' (which is 0b00110000), or with the NULL pointer.
  - \ is used for escape sequences to type character not on the keyboard: '\n', '\t', '\\', '\'', '\a', etc.

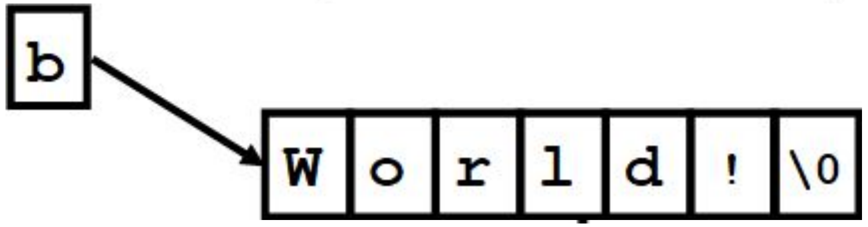
- This is the convention for ending a string in C, so the compiler throws one in there after the last character we said to give it.

Why?

```
char a[12] = "Hello ";
```

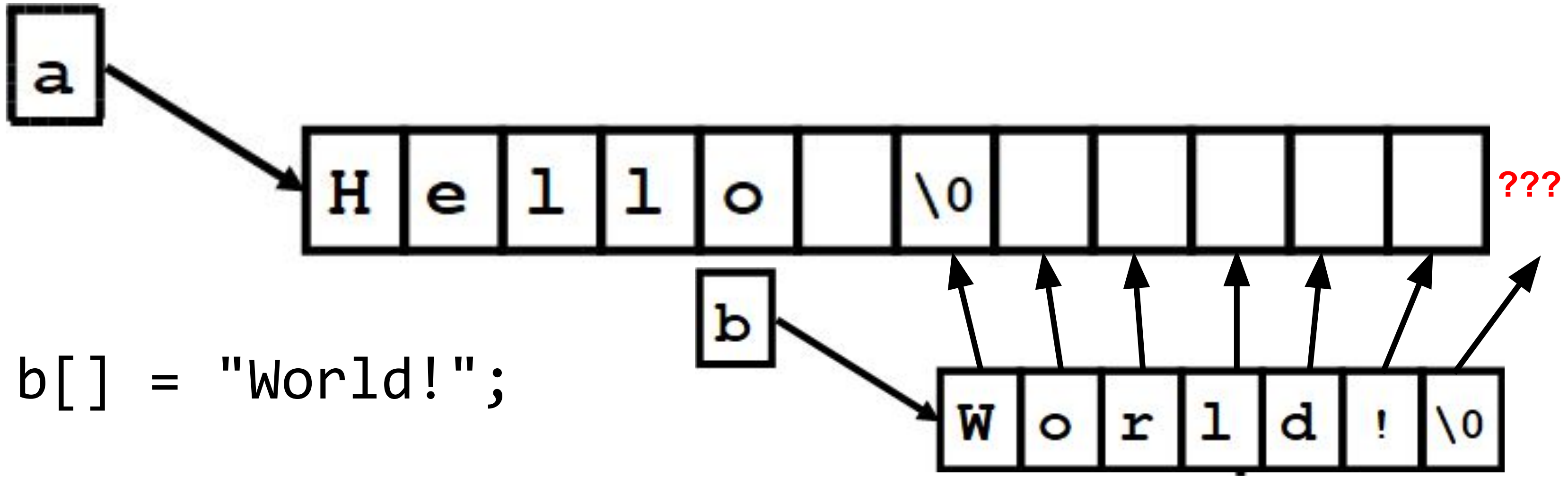


```
char b[] = "World!";
```



Why?

```
char a[12] = "Hello ";
```

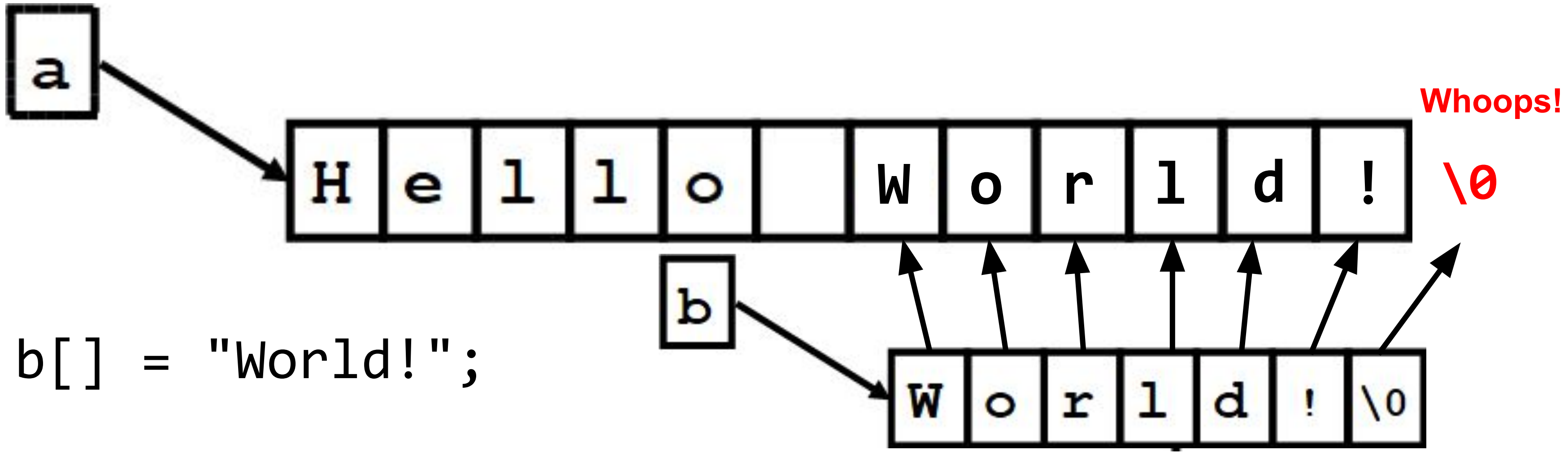


```
char b[] = "World!";

strcat(a, b);
// copies char by char from src to dest, starting at
// the first '\0' found in dest, and the first char
// in src.
// It stops when it sees the first '\0' in src.
```

Why?

```
char a[12] = "Hello ";
```



```
char b[] = "World!";
```

```
strcat(a, b);
// copies char by char from src to dest, starting at
// the first '\0' found in dest, and the first char
// in src.
// It stops when it sees the first '\0' in src.
```

The fix

```
char a[13] = "Hello ";  // needs space for 12 visible
char b[] = "World!";    //   plus the \0
strcat(a, b);
printf("%s\n", a);

Output: Hello World!
```

Potential Pitfall: Character Array vs Character Pointer

```
char a[] = "DEF";
vs
char *b = "ABC";

a[1] = 'e'; // works
b[1] = 'b'; // seg. fault
```

- Why?
  - Every "string literal" is a preallocated sequence of characters (with static storage duration) you're not allowed to mess with (const char *), in a special constant memory area.
  - char b[] = is the special case. It copies the values to the stack/static memory area from the constant version.

What's this little guy doing?

- `printf("%s\n", a);`
  - Signature:
    `int printf(const char *format, ...)`
  - Prints to stdout using C formatting rules.
  - %s substitutes characters from the provided (char *) until the first '\0' (where it stops).
  - Returns the number of characters printed (or a negative number if it fails).

- `sprintf(char *buffer, const *format, ...)`
  - Just like printf, except it prints to the provided buffer (replacing whatever was inside it initially).

- There's a whole family of these guys you can look into (fprintf…).

Careful!

```
char *foo;
printf("%s", foo);
```

What will this print?!?

Don't accidentally print something without a '\0' in it.

Also, don't accidentally squash your '\0'!

Uncareful String Example

```
char s1[10];
s1[0] = 'f';
s1[1] = 'o';
s1[2] = 'o';
printf("%s\n", s1);
```

Sample output

```
     fooâ-•VGot 2
     ,_VGot 2
     Got 20
     _VGot 2
     _Got 16
     Got 1
```

"Quiz"

```
char s1[10] = "foo";
char *s2 = "superduper";

sizeof(s1) = ?
  1)  4
  2) 10
  3) Depends
  4) 11
```

[✓ 2) 10]

"Quiz"

```
char s1[10] = "foo";
char *s2 = "superduper";

sizeof(s2) = ?
```

[✓ 1) 4]
2) 10
3) Depends
4) 11

"Quiz"

```
char s1[10] = "foo";
char *s2 = "superduper";

sizeof(*s2) = ?
  1)  1
  2)  4
  3) 10
  4) 11
```

"Quiz"

```
sizeof("Hello") = ?
```

```
6
```

"Quiz"

```
   char s1[10] = "foo";
   char *s2 = "superduper";
Legal?
   s2 = s1;


Yes.
```

"Quiz"

```
  char s1[10] = "foo";
  char *s2 = "superduper";
```

Legal?

```
  s1 = s2;
```

No!

"Quiz"

```
  char s1[10] = "foo";
  char *s2 = "superduper";
Legal?
  s1[2] = 'x';

Yes!
```

"Quiz"

```
   char s1[10] = "foo";
   char *s2 = "superduper";
Legal?
   s2[0] = 'Z';

No!
```

"Quiz"

```
  char s1[10] = "foo";
  char *s2 = "superduper";
Legal?
  s2 = s1;
  s2[0] = 'Z';

Yes!
```

"Quiz"

```
char s1[10] = "foo";
char *s2 = "superduper";
```

Does `s2 = s1` copy the strings?

No!

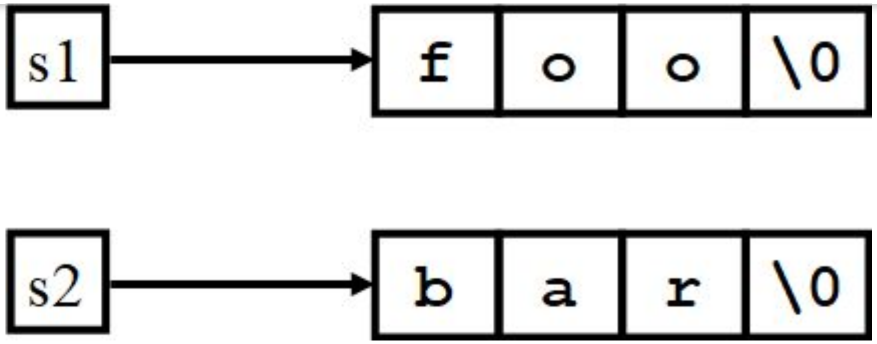
But it looks like it copies strings

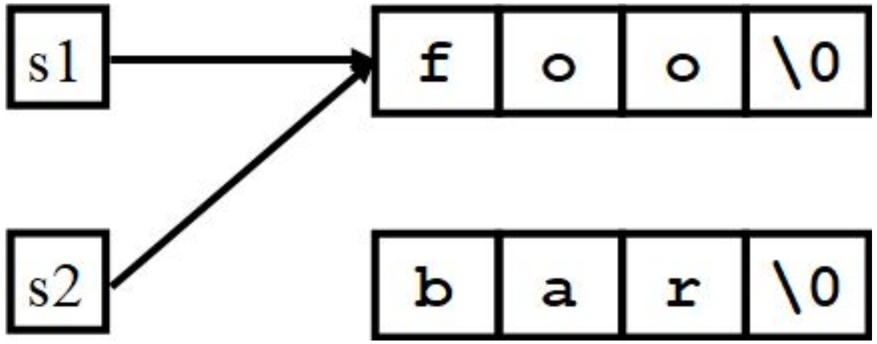
```
char s1[10] = "foo";
char *s2 = "superduper";
s2 = s1;

printf("%s\n", s1);
printf("%s\n", s2);
```

What prints?

s2 = s1;
What happens?





Arrays of Pointers

```
char *month_name(int n)
{
   static char *name[] = {
      "Illegal month",
      "January", "February", "March",
     "April", "May", "June",
      "July", "August", "September",
      "October", "November", "December"
   };
   return (n<1 || n>12)? name[0]: name[n];
}
```

Arrays of Pointers

- A block of memory (in the constant area) is initialized like this:

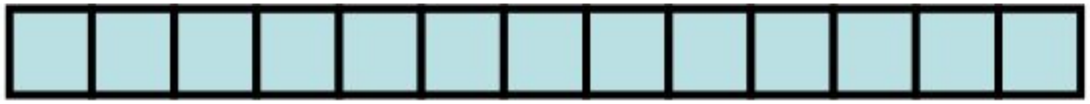
```
Illegal month\0January\0February\0March\0April\0May\0June\0July\0August\0September\0October\0November\0December\0
```

- Slightly more readably:

```
Illegal month\0January\0February\0March\0April\0
May\0June\0July\0August\0September\0October\0
November\0December\0
```

Arrays of Pointers

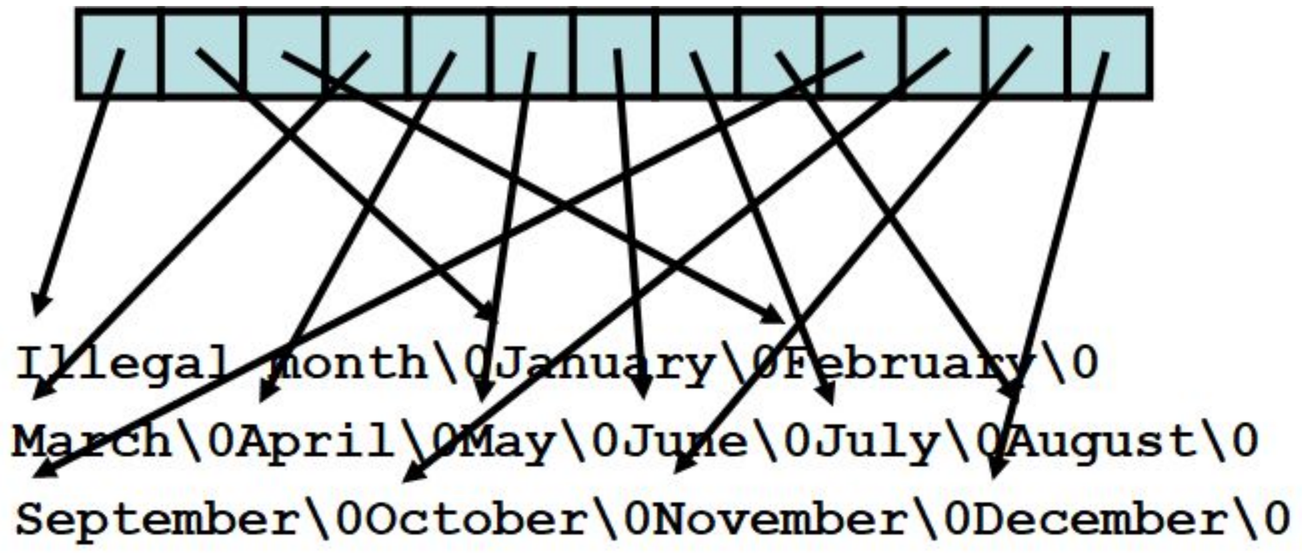
- An array is created in the static area which will hold 13 character pointers



```
Illegal month\0January\0February\0
March\0April\0May\0June\0July\0August\0
September\0October\0November\0December\0
```

Arrays of Pointers

- The pointers are initialized like so


Illegal month\0January\0February\0
March\0April\0May\0June\0July\0August\0
September\0October\0November\0December\0

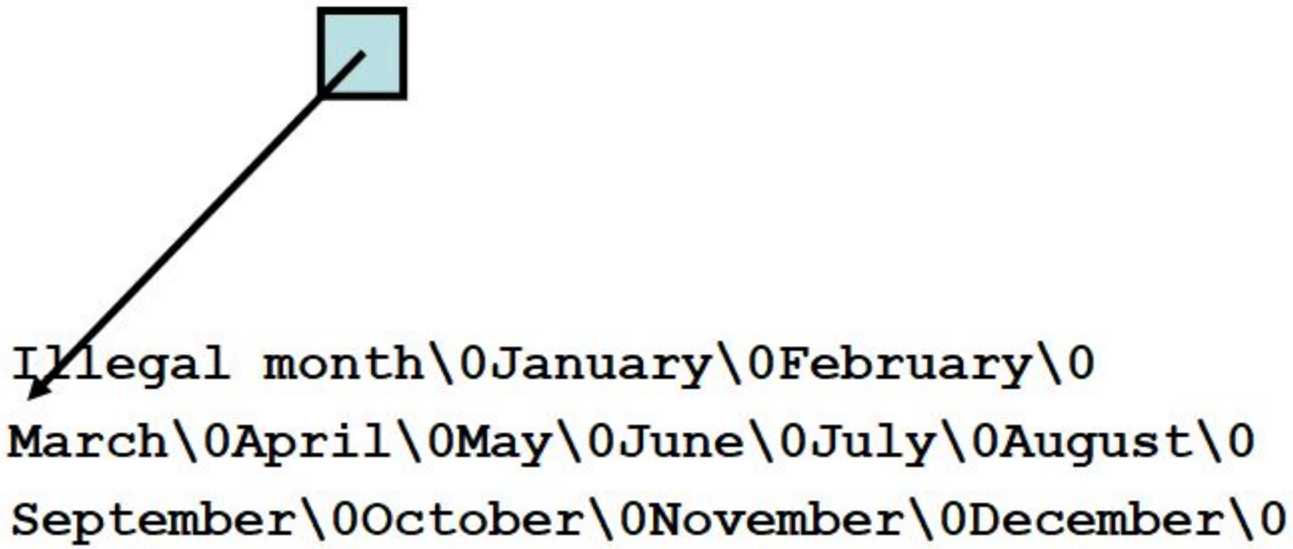
Arrays of Pointers

- So when the function is called:

```
printf("The third month is %s\n", month_name(3));
```

The line: `return (n<1 || n>12)? name[0]: name[n];`

- Gives us back the address (a pointer to) the 'M'



```
Illegal month\0January\0February\0
March\0April\0May\0June\0July\0August\0
September\0October\0November\0December\0
```

Structs

We can declare our own types!

Arrays are a chunk of memory, all of one type.

Structs are a chunk of memory that can contain multiple different types (with named fields, even!).

As close to OO as you can get in C.

  No inheritance. No polymorphism.

  Weak Encapsulation. Some abstraction.

C Structs

Trivia: The struct data type in C was derived from the ALGOL 68 (Algorithmic Language 1968) struct data type.

Define one (sort of):

```
struct tag_name {
   type member1;
   type member2;
   /* declare as many members as desired, but the entire structure size must be known to the compiler. */
};
```

C Structs

More useful definition:

```
typedef struct tag_name {
   type member1;
   type member2;
} struct_alias;
```

e.g.:

```
typedef struct student {
   unsigned char age;
   char name[128];//size must be known! max here is 127
} Student;

Now you can declare one either way:
struct student s1;
Student s2;
```

Sizeof?

```
typedef struct student {
   unsigned char age;
   char name[128];//size must be known! max here is 127
} Student;
sizeof(Student); // ???
```

What do we expect?

It's 129 (1 char, plus an array of 128 chars)

Struct Initialization

```
Student s1 = { 20, "George Burdell" };
Student s2 = { .age = 20, .name = "George Burdell" };
Student s3 = s1;


Struct s0; // what does this do?
```

Depends on where you are.

- Static, it would set everything to "zero" (NULL, '\0', 0, etc.).
- Dynamically, it isn't guaranteed to clear anything, just says "this space is yours for now.", with whatever was already in it.

Unlike arrays... You can assign them to other structs (copying ensues)

```
Student s1 = { 20, "George Burdell, Jr." };
Student s2;
Student s3 = { 57, "George Burdell, Sr." };



s2 = s1;


s1 = s3;


What's what now?
```

Accessing Data Mambers

```
Student s1 = { 20, "George Burdell, Jr." };

s1.age;
s1.name;
```

Pointers to Structs

```
Student s1 = { 20, "George Burdell, Jr." };
Student *s1ptr = &s1;

Access via the arrow operator:
s1ptr->name;
s1ptr->age;


This is the same as (*s1ptr).name, (*s1ptr.age), etc.

Why not *s1ptr.name and *s1ptr.age?
```

Not recursive

Struct cannot contain another struct of the same type.

But it ***can*** contain a pointer to another struct of the same type (like with a link-list or a tree data structure).