CS 33

Introduction to C
Part 6

Structures Containing Arrays

```
struct Array {
   int A[6];
} S1, S2;

int A1[6], A2[6];

A1 = A2;
   // not legal: array variables refer to the
   // addresses of the first elements

S1 = S2;
   // legal: structure variables refer to contents
   // of the entire structure
```

A Bit More Syntax …

- **Constants**

```
const double pi =
  3.14159265358979238;

area = pi*r*r;      /* legal */
pi = 3.0;           /* illegal */
```

More Syntax …

```
const int six = 6;
int nonconstant;
const int *ptr_to_constant;
int *const constant_ptr = &nonconstant;
const int *const constant_ptr_to_constant = &six;

ptr_to_constant = &six;
   // ok
*ptr_to_constant = 7;
   // not ok
*constant_ptr = 7;
   // ok
constant_ptr = &six;
   // not ok
```

And Still More …

- **Array initialization**

```
int FirstSixPrimes[6] = {2, 3, 5, 7, 11, 13};
int SomeMorePrimes[] = {17, 19, 23, 29};
int MoreWithRoomForGrowth[10] = {31, 37};
int MagicSquare[][] = {{2, 7, 6},
                       {9, 5, 1},
                       {4, 3, 8}};
```

Characters

- **ASCII**
  - **American Standard Code for Information Interchange**
  - **works for:**
    - **English**
    - **Swahili**
    - **not much else**
  - **doesn't work for:**
    - **French**
    - **Spanish**
    - **German**
    - **Korean**
    - **Arabic**
    - **Sanskrit**
    - **Chinese**
    - **pretty much everything else**

Characters

- Unicode
  - support for the rest of world
  - defines a number of encodings
  - most common is UTF-8
    - » variable-length characters
    - » ASCII is a subset and represented in one byte
    - » larger character sets require an additional one to three bytes
  - not covered in CS 33

ASCII Character Set

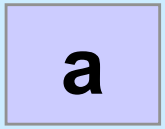
	00	10	20	30	40	50	60	70	80	90	100	110	120
0:	\0	\n			(	2	<	F	P	Z	d	n	x
1:		\v			)	3	=	G	Q	[	e	o	y
2:		\f		sp	*	4	>	H	R	\	f	p	z
3:		\r		!	+	5	?	I	S	]	g	q	{
4:				"	,	6	@	J	T	^	h	r	|
5:				#	-	7	A	K	U	_	i	s	}
6:				$	.	8	B	L	V	`	j	t	~
7:	\a			%	/	9	C	M	W	a	k	u	DEL
8:	\b			&	0	:	D	N	X	b	l	v	
9:	\t			'	1	;	E	O	Y	c	m	w	

*char*s as Integers

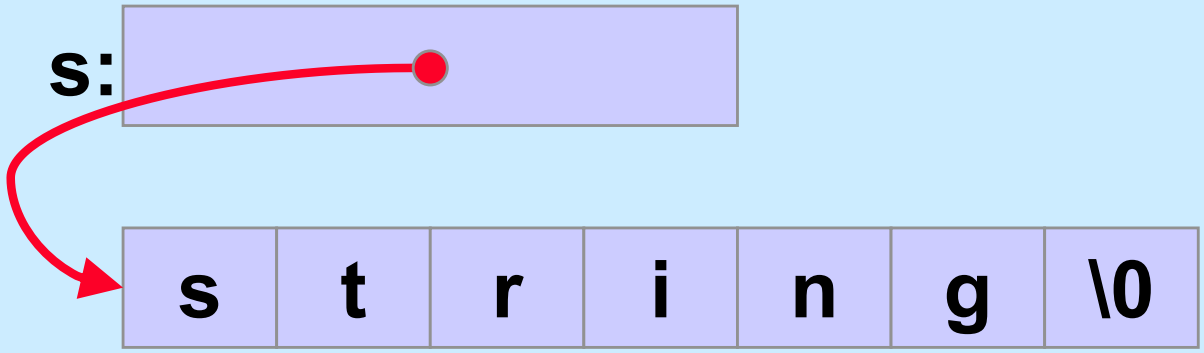
```
char tolower(char c) {
  if (c >= 'A' && c <= 'Z')
      return c + 'a' - 'A';
  else
      return c;
}
```

Character Strings

```
char c = 'a';
```




```
char *s = "string";
```

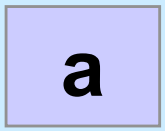


Is there any difference between *c1* and *c2* in the following?

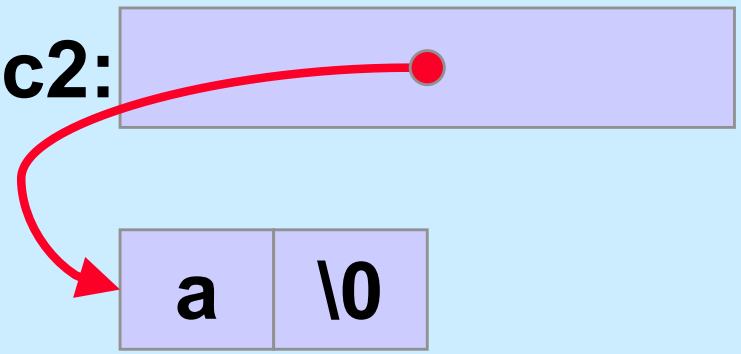
```
char c1 = 'a';
char *c2 = "a";
```

Yes!!

```
char c1 = 'a';
```

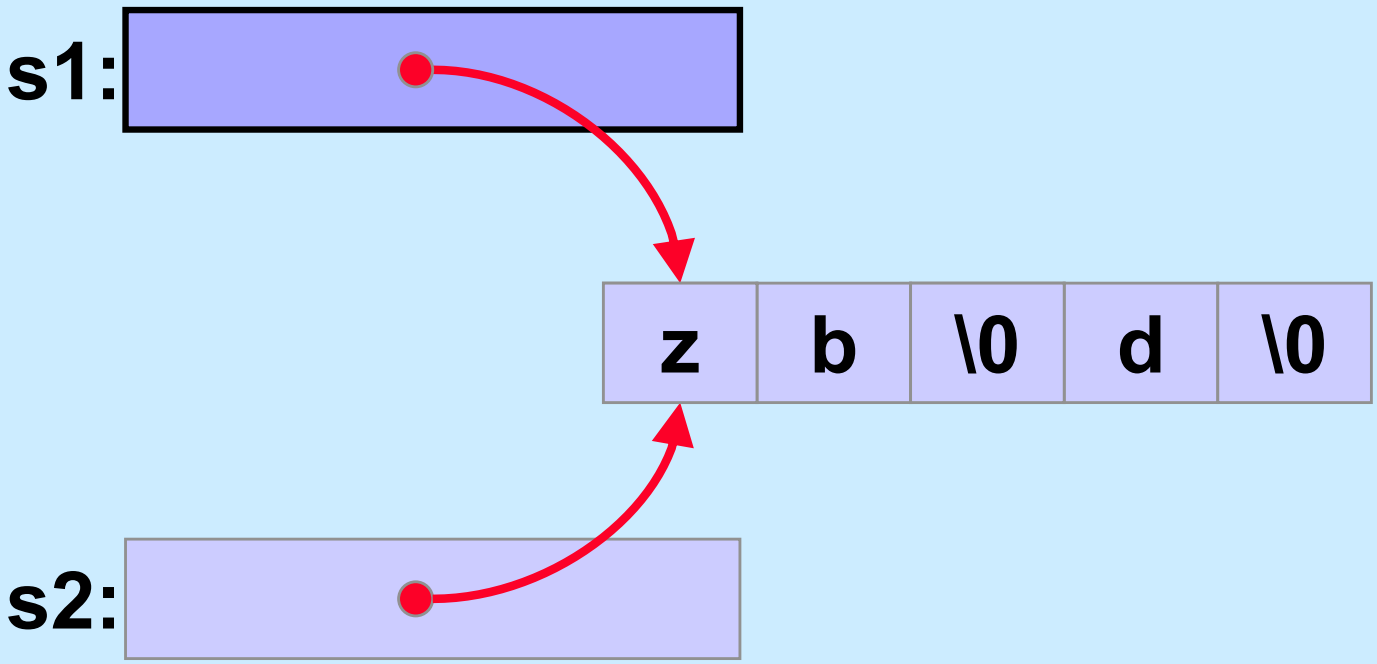



```
char *c2 = "a";
```



What do *s1* and *s2* refer to after the following is executed?

```
char s1[] = "abcd";
char *s2 = s1;
s1[0] = 'z';
s2[2] = '\0';
```



Weird …

Suppose we did it this way:

```
char *s1 = "abcd";
char *s2 = s1;
s1[0] = 'z';
s1[2] = '\0';
```

```
% gcc -o char char.c
% ./char
Segmentation fault
```

Copying Strings (1)

```
char s1[] = "abcd";
char s2[5];

s2 = s1;   // does this do anything useful?

// correct code for copying a string
for (i=0; s1[i] != '\0'; i++)
  s2[i] = s1[i];
s2[i] = '\0';

// would it work if s2 were declared:
char *s2;
// ?
```

Copying Strings (2)

```
char s1[] = "abcdefghijklmnopqrstuvwxyz";
char s2[5];

for (i=0; s1[i] != '\0'; i++)
  s2[i] = s1[i];
s2[i] = '\0';
```

Does this work?

```
for (i=0; (i<4) && (s1[i] != '\0'); i++)
  s2[i] = s1[i];
s2[i] = '\0';
```

Works!

String Length

```
char *s1;

s1 = produce_a_string();
// how long is the string?

sizeof(s1); // doesn't yield the length!!

for (i=0; s1[i] != '\0'; i++)
  ;
// number of characters in s1 is i
// (not including the terminating '\0')
```

Size

```
int main() {
    char s[] = "1234";
    printf("%d\n", sizeof(s));
    proc(s, 5);
    return 0;
}

void proc(char s1[], int len) {
    char s2[12];
    printf("%d\n", sizeof(s1));
    printf("%d\n", sizeof(s2));
}
```

```
$ gcc -o size size.c
$ ./size
5
8
12
$
```

Quiz 1

```
void proc(char s[9]) {
    printf("%d\n", sizeof(s));
}
```

What's printed?

a) 7
b) 8
c) 9
d) 10

Comparing Strings (1)

```
char *s1;
char *s2;

s1 = produce_a_string();
s2 = produce_another_string();
// how can we tell if the strings are the same?

if (s1 == s2) {
  // does this mean the strings are the same?
} else {
  // does this mean the strings are different?
}
```

Comparing Strings (2)

```
int strcmp(char *s1, char *s2) {
  int i;
  for (i=0;
       (s1[i] == s2[i]) && (s1[i] != 0) && (s2[i] != 0);
       i++)
    ; // an empty statement
  if (s1[i] == 0) {
    if (s2[i] == 0) return 0; // strings are identical
    else return -1; // s1 < s2
  } else if (s2[i] == 0) return 1; // s2 < s1
  if (s1[i] < s2[i]) return -1; // s1 < s2
  else return 1;  // s2 < s1;
}
```

The String Library

```
#include <string.h>

char *strcpy(char *dest, char *src);
  // copy src to dest, returns ptr to dest
char *strncpy(char *dest, char *src, int n);
  // copy at most n bytes from src to dest
int strlen(char *s);
  // returns the length of s (not counting the null)
int strcmp(char *s1, char *s2);
  // returns -1, 0, or 1 depending on whether s1 is
  // less than, the same as, or greater than s2
int strncmp(char *s1, char *s2, int n);
  // do the same, but for at most n bytes
```

The String Library (more)

```
size_t strspn(const char *s, const char *accept);
      // return length of initial portion of s
      // consisting entirely of bytes from accept

size_t strcspn(const char *s, const char *reject);
      // return length of initial portion of s
      // consisting entirely of bytes not from
      // reject
```

Quiz 2

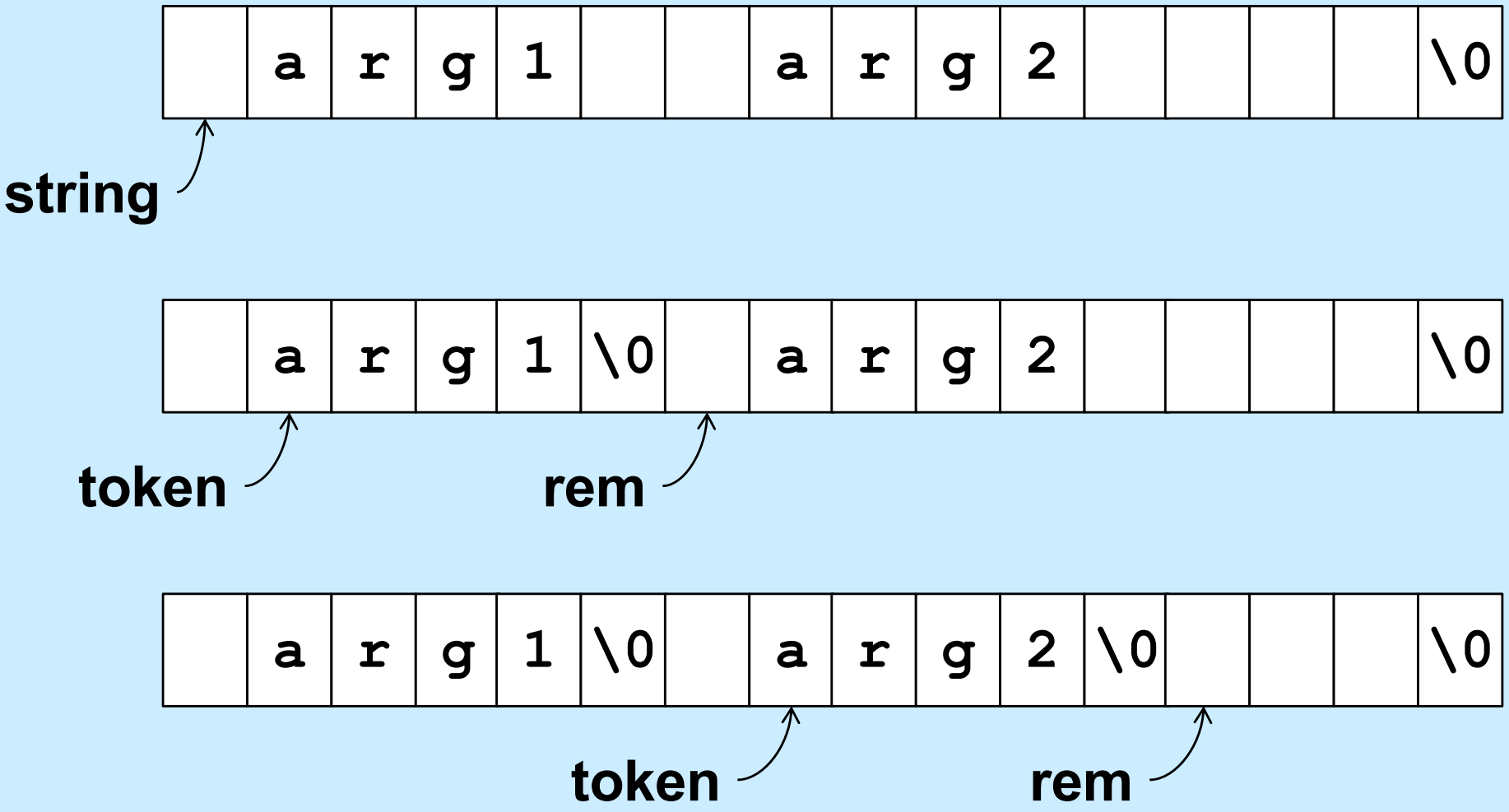
```
#include <stdio.h>
#include <string.h>

int main() {
  char s1[] = "Hello World!\n";
  char *s2;
  strcpy(s2, s1);
  printf("%s", s2);
  return 0;
}
```

This code:

a) has syntax problems
b) might seg fault
c) is a great example of well written C code

Parsing a String



Designing the Parse Function

- It modifies the string being parsed
  - puts nulls at the end of each token
- Each call returns a pointer to the next token
  - how does it know where it left off the last time?
    - how is *rem* dealt with?

Design of *strtok*

- **char** *strtok(**char** *string,
  const char *sep)
  - **if *string* is non-NULL, *strtok* returns a pointer to the first token in *string* (and keeps track of where the next token would be)**
  - **if *string* is NULL, *strtok* returns a pointer to the token just after the one returned in the previous call (and keeps track of where the next token would be), or returns NULL if there are no more tokens**
  - **tokens are separated by any non-empty combination of characters in *sep***

Using *strtok*

```
int main() {
  char line[] = "  arg0   arg1 arg2   arg3   ";
  char *str = line;
  char *token;
  while ((token = strtok(str, " \t\n")) != NULL) {
    printf("%s\n", token);
    str = NULL;
  }
  return 0;
}
```

Output:

```
arg0
arg1
arg2
arg3
```

strtok Code part 1

```
char *strtok(char *string, const char *sep) {
  static char *rem = NULL;
  if (string == NULL) {
    if (rem == NULL) return NULL;
    string = rem;
  }
  int len = strlen(string);
  int slen = strspn(string, sep);
      // initial separators
  if (slen == len) {
    // string is all separators
    rem = NULL;
    return NULL;
  }
```

strtok Code part 2

```
  string = &string[slen]; // skip over separators
  len -= slen;
  int tlen = strcspn(string, sep); // length of first token
  if (tlen < len) {
    // token ends before end of string: terminate it with 0
    string[tlen] = '\0';
    rem = &string[tlen+1];
  } else {
    // there's nothing after this token
    rem = NULL;
  }
  return string;
}
```

Numeric Conversions

```
short a;
int b;
float c;

b = a;    /* always works */
a = b;    /* sometimes works */
c = b;    /* sort of works */
b = c;    /* sometimes works */
```

Implicit Conversions (1)

```
float x, y=2.0;
int i=1, j=2;

x = i/j + y;
  /* what's the value of x? */
```

Implicit Conversions (2)

```
float x, y=2.0;
int i=1, j=2;
float a, b;

a = i;
b = j;
x = a/b + y;
  /* now what's the value of x? */
```

Explicit Conversions: Casts

```
float x, y=2.0;
int i=1, j=2;

x = (float)i/(float)j + y;
  /* and now what's the value of x? */
```

Purposes of Casts

- **Coercion**

```
int i, j;
float a;
a = (float)i/(float)j;
```

modify the value appropriately

- **Intimidation**

```
float x, y;
    // sizeof(float) == 4
swap((int *)&x, (int *)&y);
```

it's ok as is (trust me!)

Quiz 3

- **Will this work?**

```
double x, y; //sizeof(double) == 8

...
swap((int *)&x, (int *)&y);
```

a) **yes**
b) **no**