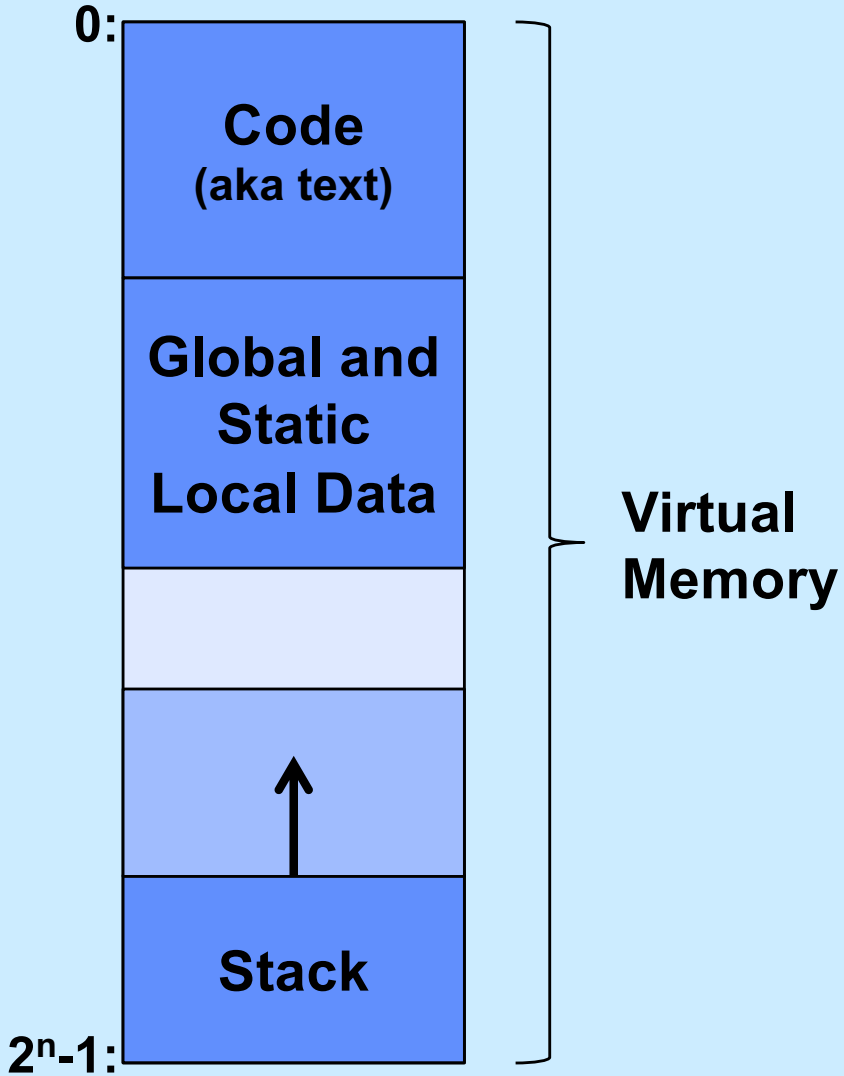
CS 33

Introduction to C
Part 5

Digression: Where Stuff Is
(Roughly)



scanf: Reading Data

```
int main() {
    int i, j;
    scanf("%d %d", &i, &j);
    printf("%d, %d", i, j);
}
```

```
$ ./a.out
   3          12
3, 12
```

Two parts

- **formatting instructions**
  - **whitespace in format string matches any amount of white space in input**
    - » **whitespace is space, tab, newline (‘\n’)**
- **arguments: must be addresses**
  - **why?**

#define (again)

```
#define CtoF(cel) (9.0*cel)/5.0 + 32.0
```

Simple textual substitution:

```
float tempc = 20.0;
float tempf = CtoF(tempc);
// same as tempf = (9.0*tempc)/5.0 + 32.0;
```

Careful ...

```
#define CtoF(cel) (9.0*cel)/5.0 + 32.0

float tempc = 20.0;
float tempf = CtoF(tempc+10);
// same as tempf = (9.0*tempc+10)/5.0 + 32.0;
```

```
#define CtoF(cel) (9.0*(cel))/5.0 + 32.0

float tempc = 20.0;
float tempf = CtoF(tempc+10);
// same as tempf = (9.0*(tempc+10))/5.0 + 32.0;
```

Structures

```
struct ComplexNumber {
        float real;
        float imag;
};

struct ComplexNumber x;
x.real = 1.4;
x.imag = 3.65e-10;
```

Pointers to Structures

```
struct ComplexNumber {
        float real;
        float imag;
};

struct ComplexNumber x, *y;
x.real = 1.4;
x.imag = 3.65e-10;
y = &x;
y->real = 2.6523;
y->imag = 1.428e20;
```

*struct*s and Functions

```
struct ComplexNumber ComplexAdd(
            struct ComplexNumber a1,
            struct ComplexNumber a2) {
        struct ComplexNumber result;
        result.real = a1.real + a2.real;
        result.imag = a1.imag + a2.imag;
        return result;
}
```

Would This Work?

```
struct ComplexNumber *ComplexAdd(
            struct ComplexNumber *a1,
            struct ComplexNumber *a2) {
        struct ComplexNumber result;
        result.real = a1->real + a2->real;
        result.imag = a1->imag + a2->imag;
        return &result;
}
```

How About This?

```
void ComplexAdd(
        struct ComplexNumber *a1,
        struct ComplexNumber *a2,
        struct ComplexNumber *result) {
  result->real = a1->real + a2->real;
  result->imag = a1->imag + a2->imag;
  return;
}
```

Using It ...

```
struct ComplexNumber j1 = {3.6, 2.125};
struct ComplexNumber j2 = {4.32, 3.1416};
struct ComplexNumber sum;

ComplexAdd(&j1, &j2, &sum);
```

Arrays of *structs*

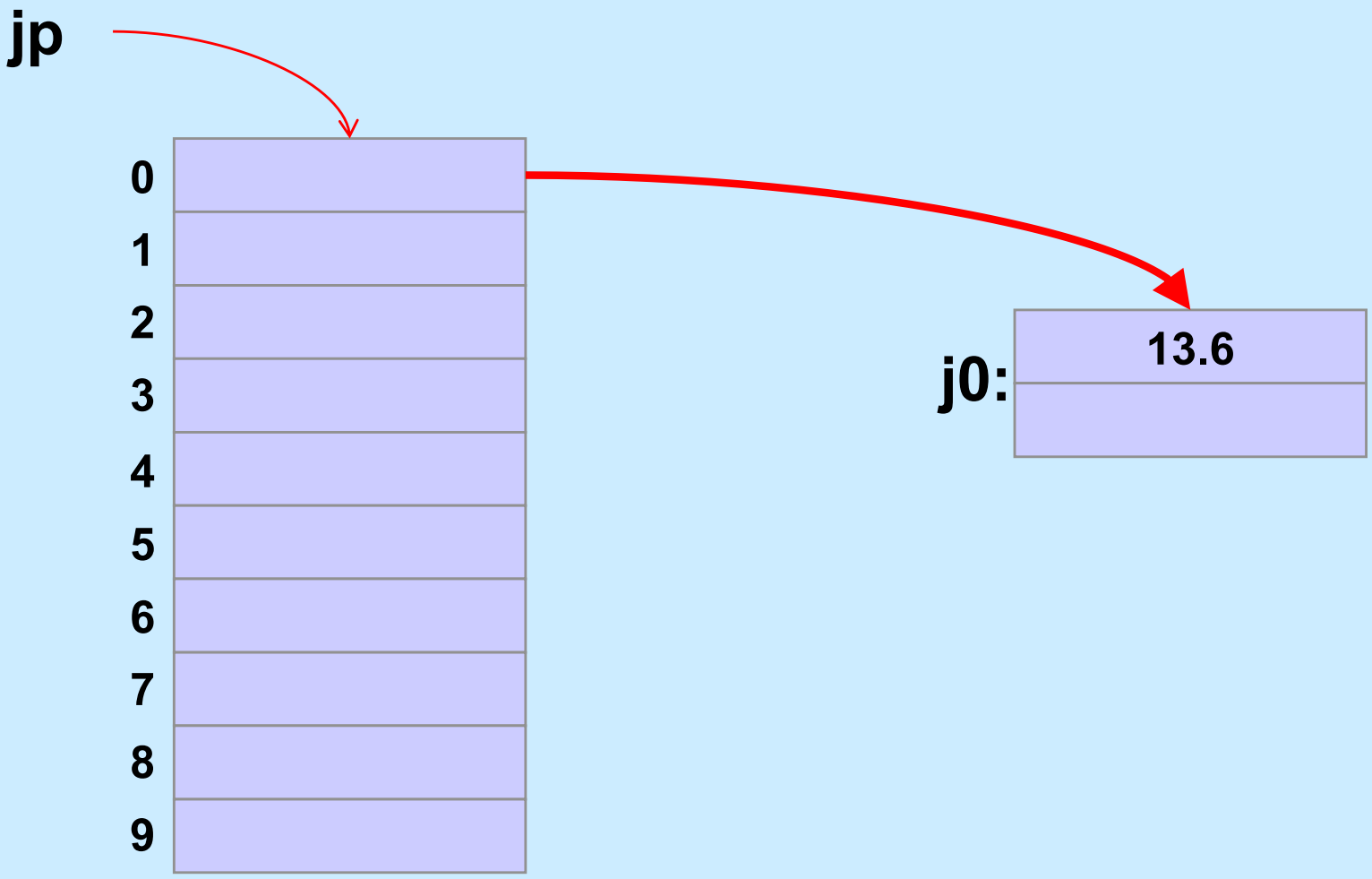
```
struct ComplexNumber j[10];
j[0].real = 8.127649;
j[0].imag = 1.76e18;
```

Arrays, Pointers, and *structs*

```
/* What's this? */
struct ComplexNumber *jp[10];
```

```
struct ComplexNumber j0;
jp[0] = &j0;
jp[0]->real = 13.6;
```

Memory View



Quiz 1

```
struct list_elem {
    int val;
    struct list_elem *next;
} a, b;

int main() {
    a->val = 1;
    a->next = &b;
    b->val = 2;
    printf("%d\n", a->next->val);
    return 0;
}
```

- **What happens?**
  a) **syntax error**
  b) **seg fault**
  c) **prints something and terminates**

Quiz 2

```
struct list_elem {
    int val;
    struct list_elem *next;
} a, b;

int main() {
    a.val = 1;
    a.next = &b;
    b.val = 2;
    printf("%d\n", a.next.val);
    return 0;
}
```

- **What happens?**
  a) **syntax error**
  b) **seg fault**
  c) **prints something and terminates**

Quiz 3

```
struct list_elem {
    int val;
    struct list_elem *next;
} a, b;

int main() {
    a.val = 1;
    b.val = 2;
    printf("%d\n", a.next->val);
    return 0;
}
```

- **What happens?**
  a) **syntax error**
  b) **seg fault**
  c) **prints something and terminates**

Quiz 4

```
struct list_elem {
    int val;
    struct list_elem *next;
} a, b;

int main() {
    a.val = 1;
    a.next = &b;
    b.val = 2;
    printf("%d\n", a.next->val);
    return 0;
}
```

- **What happens?**
  a) **syntax error**
  b) **seg fault**
  c) **prints something and terminates**

Structures vs. Objects

- **Are structs objects?**

(What's an object?)

Structures Containing Arrays

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

int A1[6], A2[6];

A1 = A2;
    // not legal: arrays don't know how big they are

S1 = S2;
    // legal: structures do
```

A Bit More Syntax …

- **Constants**

```
const double pi =
  3.14159265358979238;

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

More Syntax …

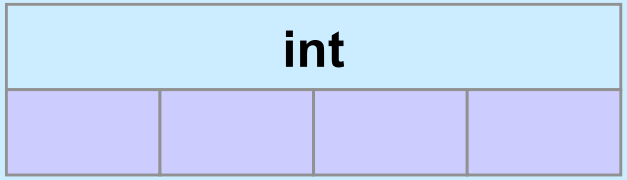
```
const int six = 6;
int nonconstant;
const int *ptr_to_constant;
int *const constant_ptr = &nonconstant;
const int *const constant_ptr_to_constant = &six;

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

And Still More …

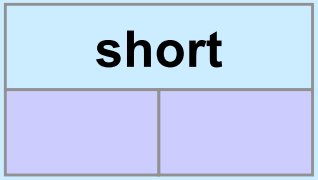
- **Array initialization**

```
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}};
```

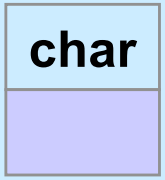
Basic Data Types



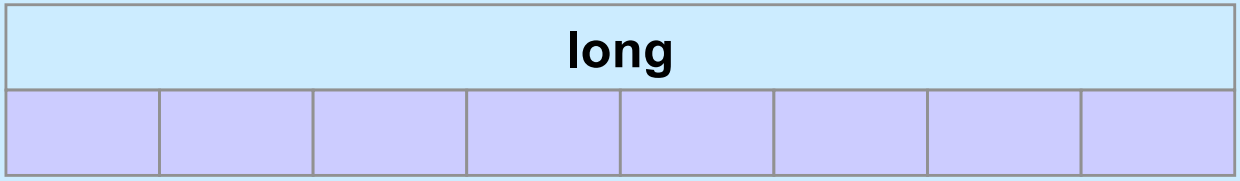
-2,147,483,648 – 2,147,483,647



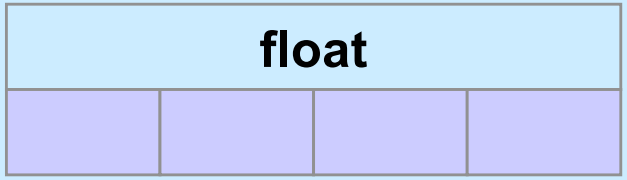
-32,768 – 32,767



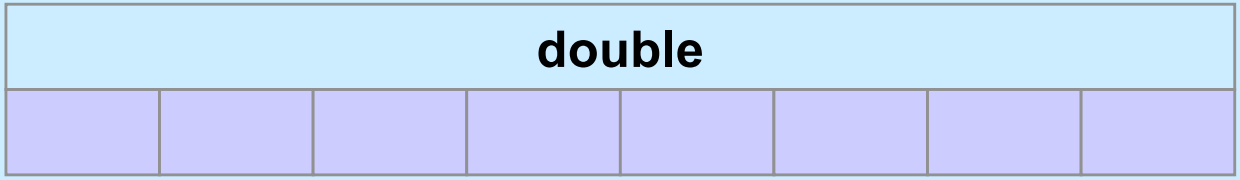
-128 – 127



-9,223,372,036,854,775,808 – 9,223,372,036,854,775,807



$\pm 1.8 \times 10^{-38} - \pm 3.4 \times 10^{38}$, ~7 decimal digits



$\pm 2.23 \times 10^{-308} - \pm 1.8 \times 10^{308}$, ~16 decimal digits

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

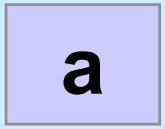
	00	10	20	30	40	50	60	70	80	90	100	110	120
0:	\0	\n			(	2	<	F	P	Z	d	n	x
1:		\v			)	3	=	G	Q	[	e	o	y
2:		\f		sp	*	4	>	H	R	\	f	p	z
3:		\r		!	+	5	?	I	S	]	g	q	{
4:				"	,	6	@	J	T	^	h	r	|
5:				#	-	7	A	K	U	_	i	s	}
6:				$	.	8	B	L	V	`	j	t	~
7:	\a			%	/	9	C	M	W	a	k	u	DEL
8:	\b			&	0	:	D	N	X	b	l	v	
9:	\t			'	1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

*char*s as Integers

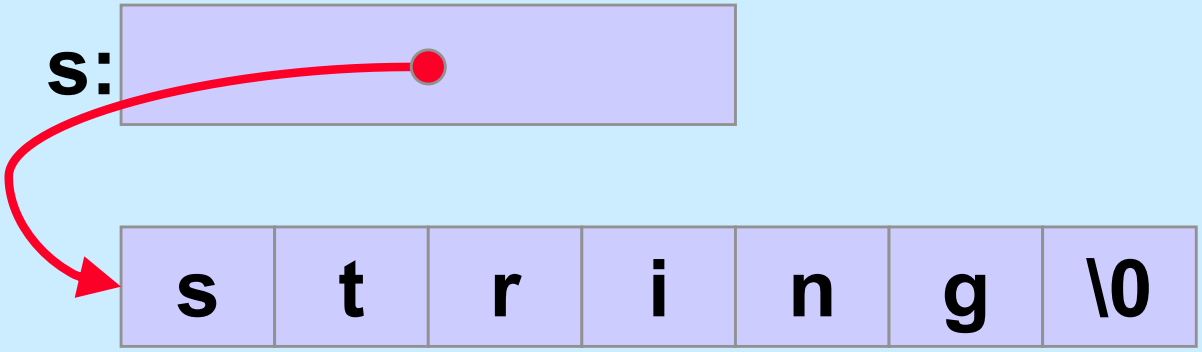
```
char tolower(char c) {
  if (c >= 'A' && c <= 'Z')
      return c + 'a' - 'A';
  else
      return c;
}
```

Character Strings

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




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

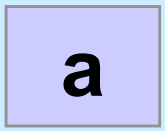


Is there any difference between *c1* and *c2* in the following?

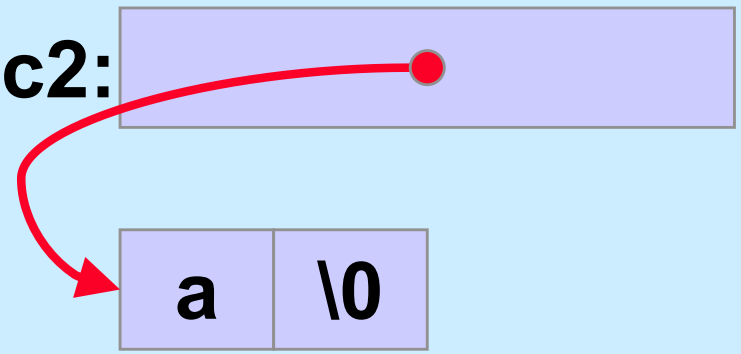
```
char c1 = 'a';
char *c2 = "a";
```

Yes!!

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

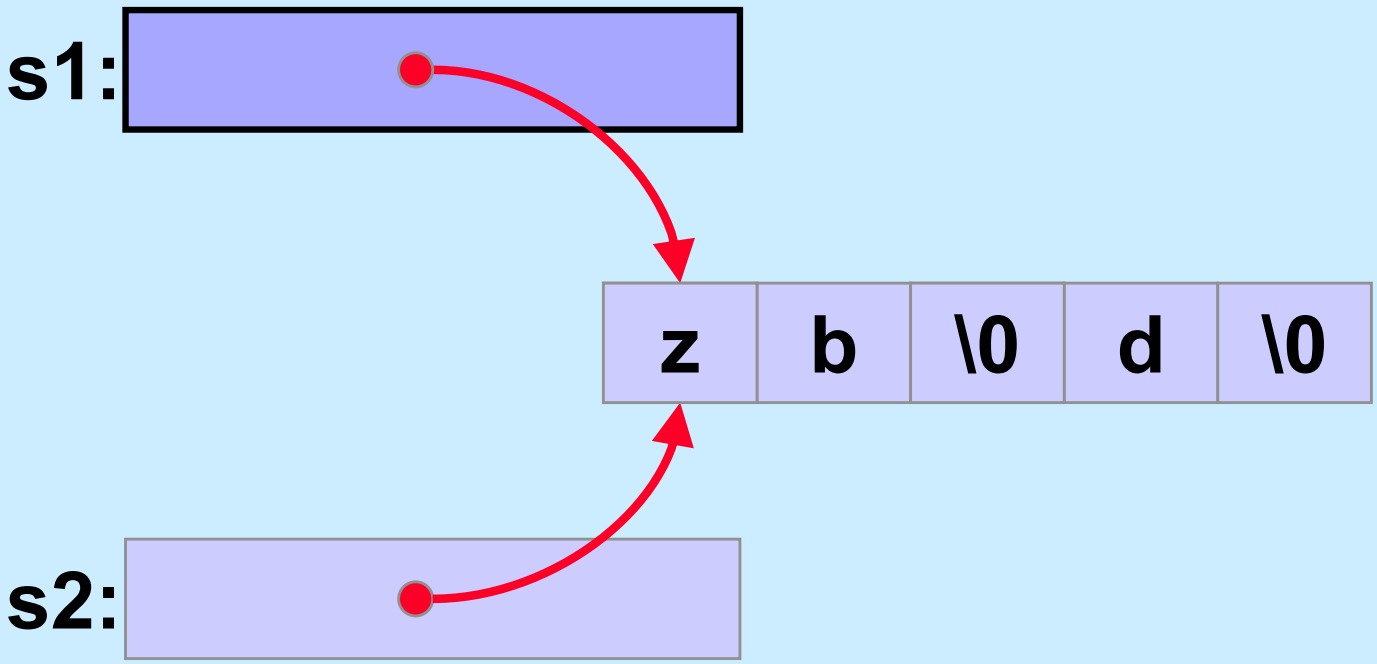



```
char *c2 = "a";
```



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
```

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 5

```
void proc(char s[16]) {
    printf("%d\n", sizeof(s));
}
```

What's printed?

a) 8
b) 15
c) 16
d) 17

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);
  // return 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);
      // returns length of initial portion of s
      // consisting entirely of bytes from accept

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

Quiz 6

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

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

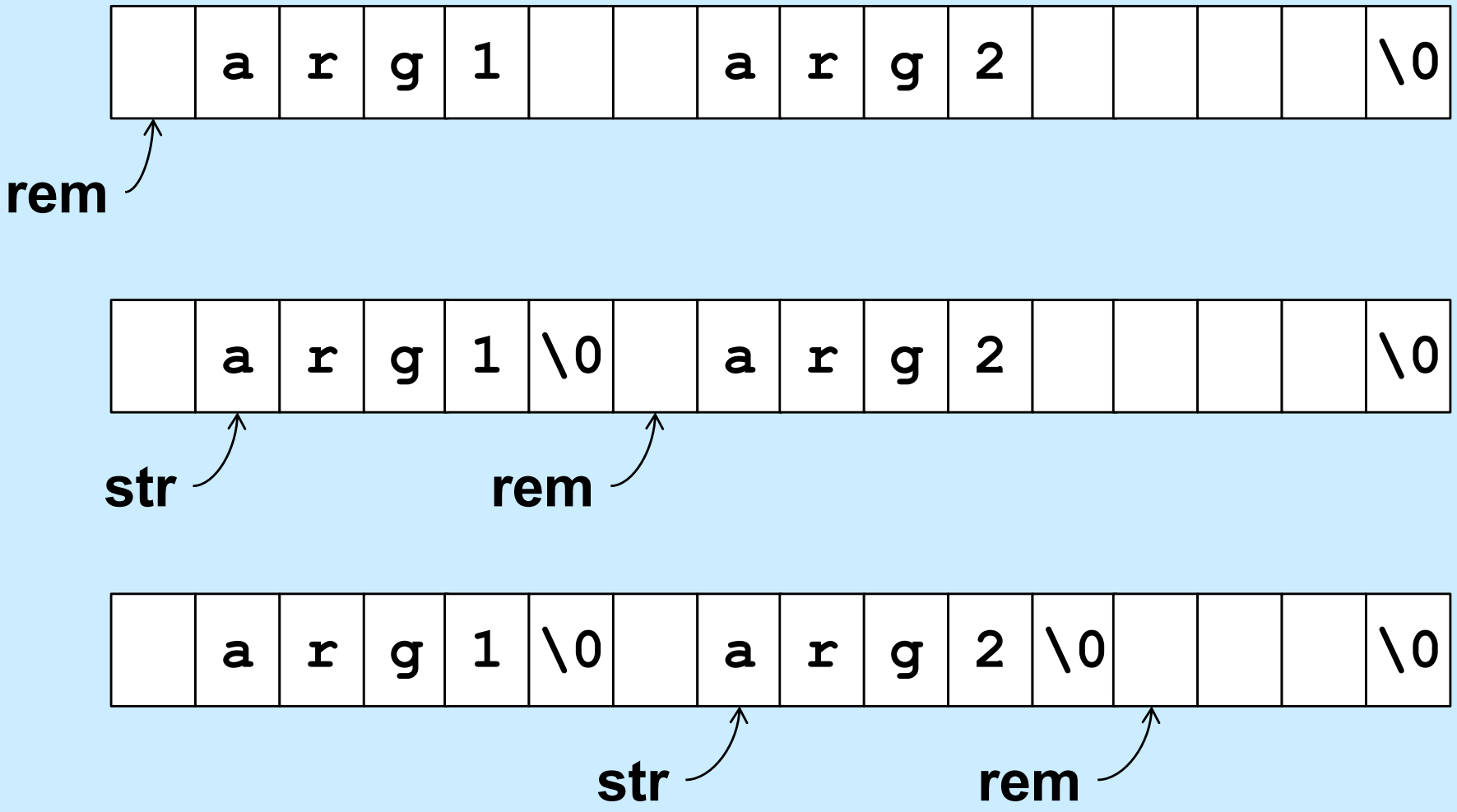
This code:

a) **is a great example of well written C code**

b) **has syntax problems**

c) **might seg fault**

Parsing a String

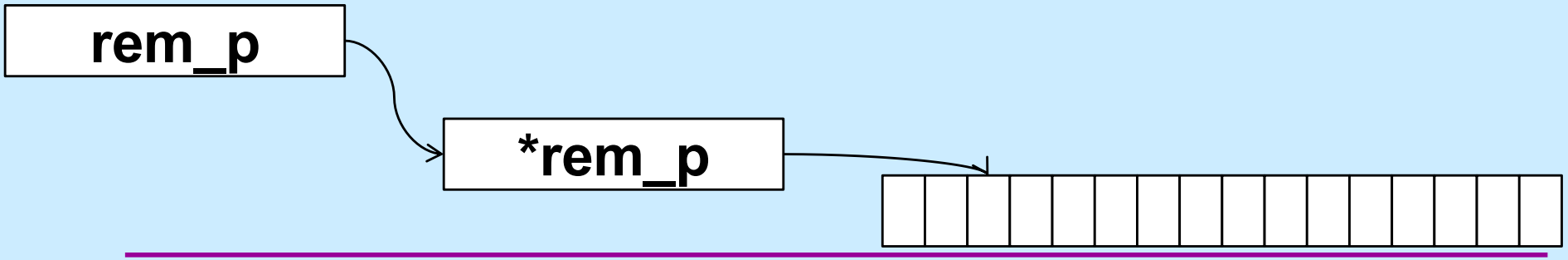


Design of *getfirstword*

- char *getfirstword(char **rem_p)
  - returns
    - » pointer to null-terminated first word in *rem_p

      or
    - » NULL, if *rem_p is a string entirely of whitespace
  - *rem_p modified to
    - » point to character following first word in *rem_p if within bounds of string

      or
    - » NULL if next character not within bounds



Using *getfirstword*

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

Output:

```
arg0
arg1
arg2
arg3
```

Code

```
char *getfirstword(char **rem_p) {
  char *str = *rem_p;
  if (str == NULL)
    return NULL;
  int len = strlen(str);
  int wslen =
    strspn(str, " \t\n");
      // initial whitespace
  if (wslen == len) {
    // string is all whitespace
    return NULL;
  }
  str = &str[wslen];
    // skip over whitespace
  len -= wslen;
  int wlen =
    strcspn(str, " \t\n");
      // length of first word
  if (wlen < len) {
    // word ends before end of
    // string: terminate
    // it with null
    str[wlen] = '\0';
    *rem_p = &str[wlen+1];
  } else {
    // no more words
    *rem_p = NULL;
  }
  return str;
}
```