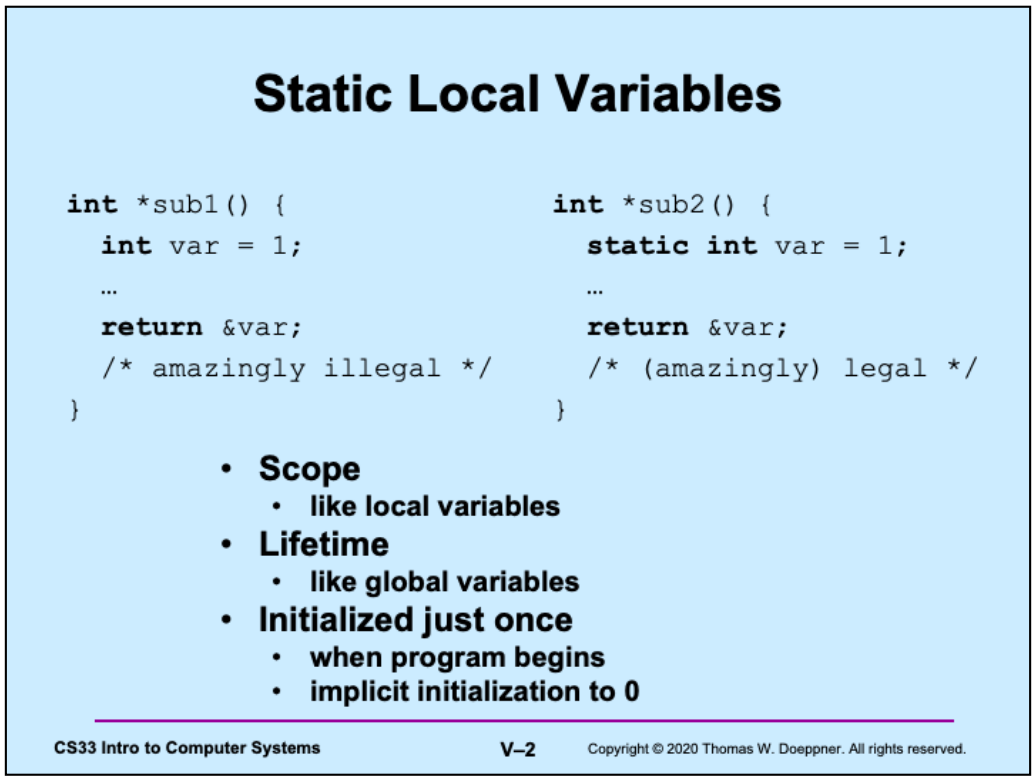
CS 33

Introduction to C
Part 5



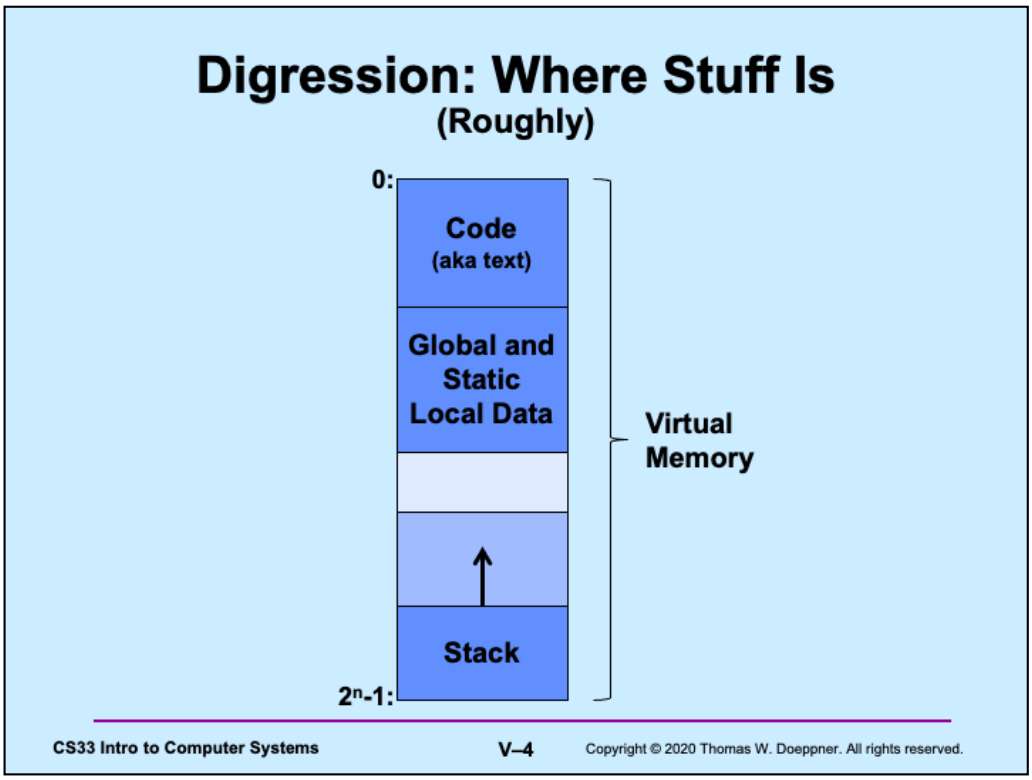
Static local variables have the same scope as other local variables, but their values are retained across calls to the procedures they are declared in. Like global variables, uninitialized static local variables are implicitly initialized to zero. Initialization happens just once, when the program starts up. Thus in *sub2*, *var* is set to 1 when the program starts, and not every time *sub2* is called.

Not a Quiz!

```
int sub() {
  static int svar = 1;
  int lvar = 1;
  svar += lvar;
  lvar++;
  return svar;
}

int main() {
  sub();
  printf("%d\n", sub());
  return 0;
}
```

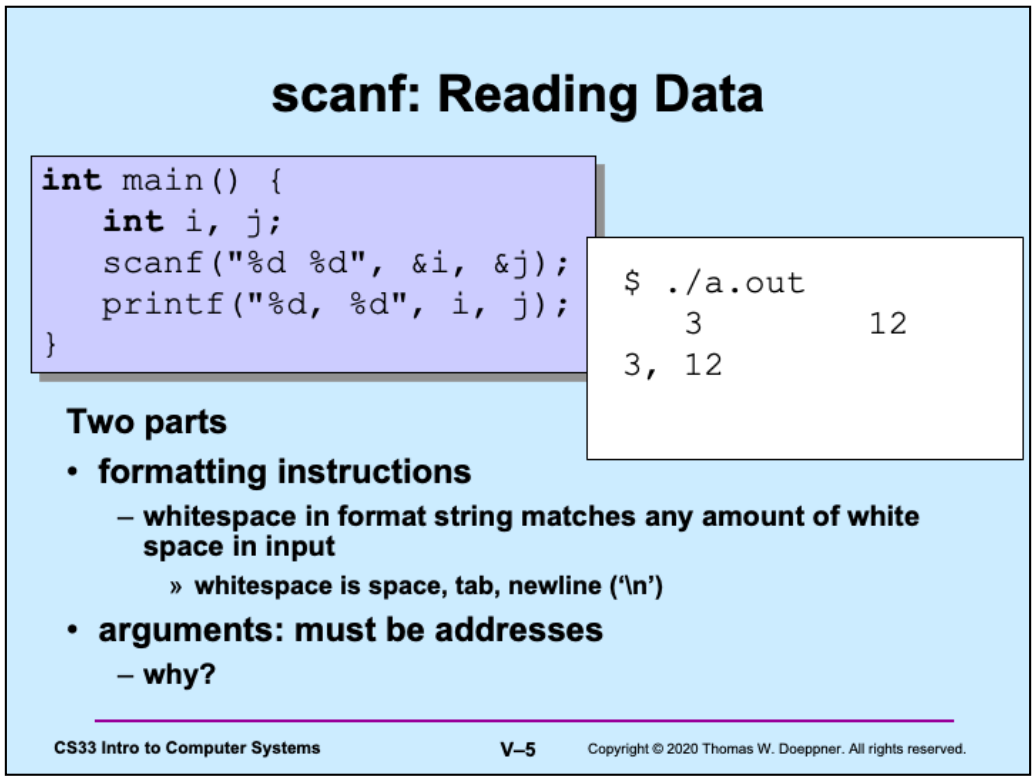
What is printed?

a) 2
b) 3
c) 4
d) 5



Let's step back and revisit our concept of virtual memory. All of a program, both code and data, resides in virtual memory. We begin to explore how all of this is organized. This is neither a complete nor a totally accurate picture, but serves to explain what we've seen so far. Executable code (also known, historically, as text) resides at the lower-addressed regions of virtual memory. After it comes a region of memory that contains global and static local data. At the high-addressed end of the address space is memory reserved for the stack. The stack itself starts at the high end of this region and grows (in response to function calls, etc.). If the end of the stack reaches the end of the region of memory reserved for it, a segmentation fault occurs and the program terminates.

This is clearly very rough. As we learn more about how computer systems work, we'll fill in more and more of the details.



The function *scanf* is called to read input, doing essentially the reverse of what *printf* does. Its first argument is a format string, like that of *printf*. Its subsequent arguments are pointers to locations where the input should be copied (after format conversion as specified in the format string). Note that we must have pointers for these arguments, not simple values, since arguments are passed by value. (Make sure you understand why this is important!)

The format conversion done is the reverse of what *printf* does. For example, *printf*, given the *%d* format code, converts the machine representation of an integer into its string representation in decimal notation. *scanf* with the same format code takes the string representation of a number in decimal notation and converts it to the machine representation of an integer.

#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;
```


Be careful with how arguments are used! Note the use of parentheses in the second version.

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;
```


Note that when we refer to members of a structure via a pointer, we use the "->" notation rather than the "." notation.

structs 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;
}
```


This doesn't work, since it returns a pointer to result that would not be in scope once the procedure has returned. Thus the returned pointer would point to an area of memory with undefined contents.

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;
}
```


This works fine: the caller provides the location to hold the result.

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*

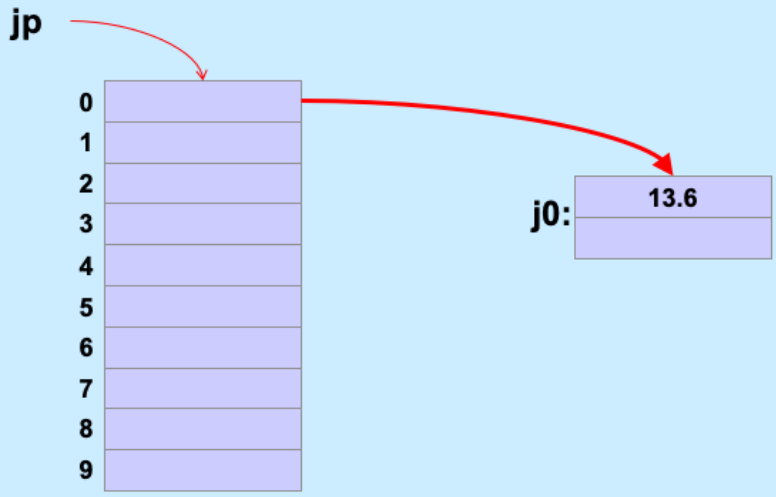
```
struct ComplexNumber j[10];
j[0].real = 8.127649;
j[0].imag = 1.76e18;
```

Arrays, Pointers, and *structs*

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


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


Subscripting (i.e., the "[]" operator) has a higher precedence than the "*" operator. Thus jp is an array of pointers to *struct ComplexNumber*s.

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

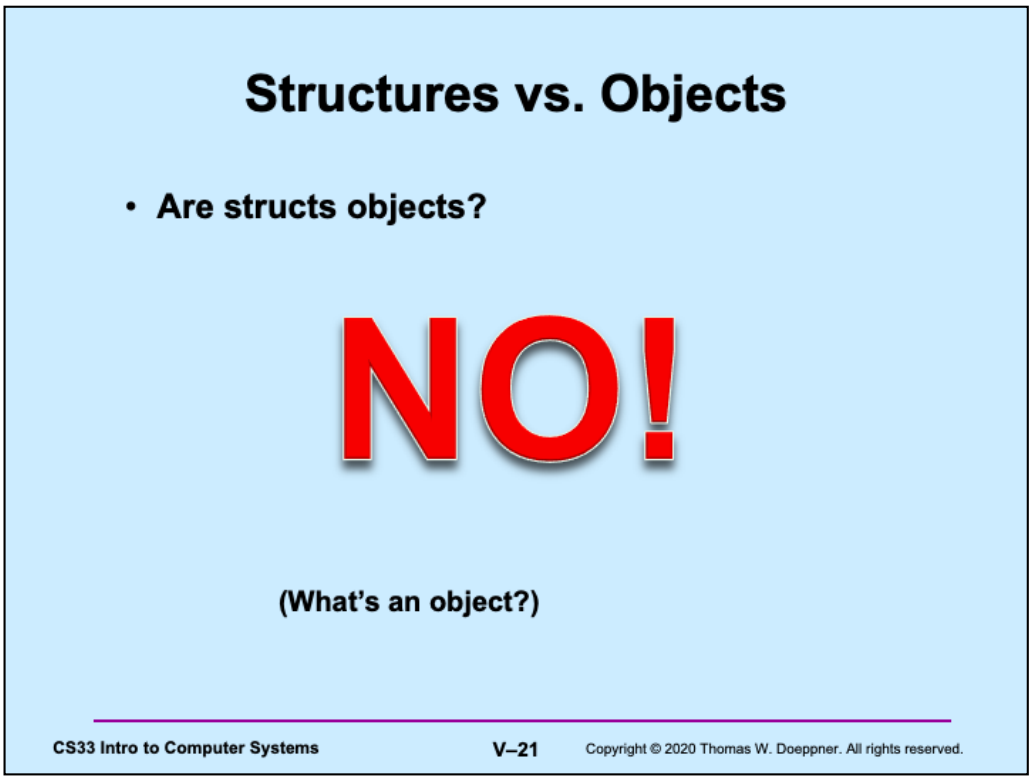
```
struct list_elem {
   int val;
   struct list_elem *next;
} a, b;

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




```
for (;;)
  printf("C does not have objects!\n");
```

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
```


This seems pretty weird at first glance. But keep in mind that the name of an array refers to the address its first element, and does not represent the entire array. But the name of a structure refers to the entire structure.

A Bit More Syntax …

- **Constants**

```
const double pi =
  3.1415926535897932 38;

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

More Syntax …

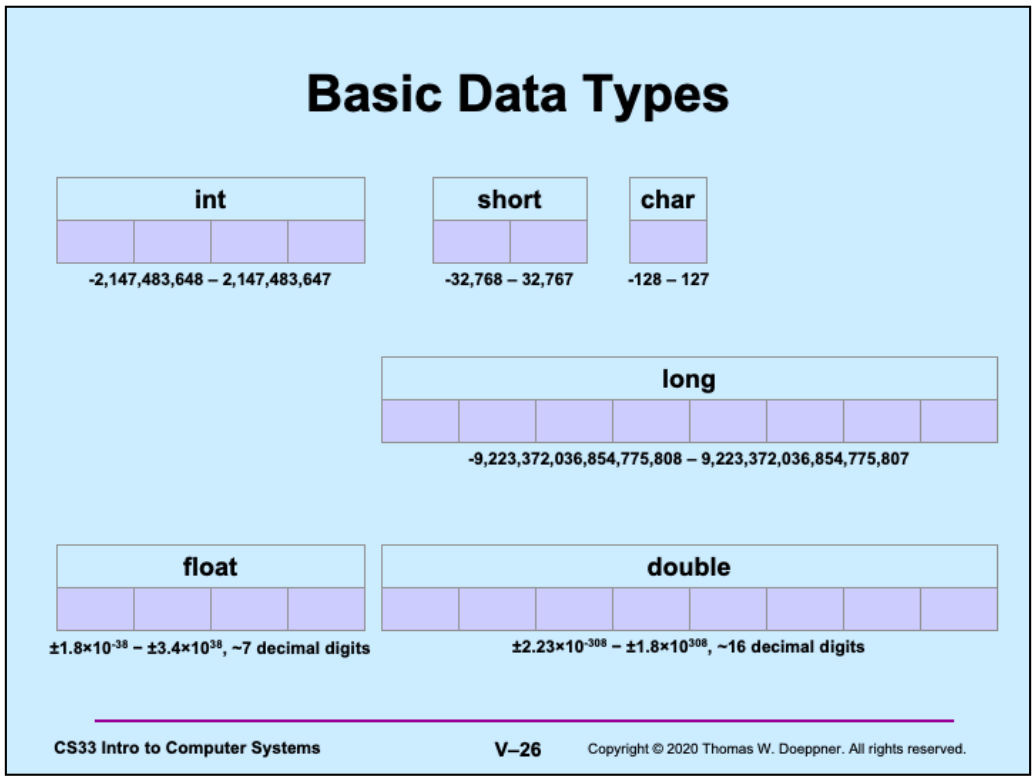
```
const int six = 6;
int nonconstant;
const int *ptr_to_constant;
int *const constant_ptr = &nonconstant;
const int *const constant_ptr_to_constant = &six;

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


Note that constant_ptr_to_constant's value may not be changed, and the value of what it points to may not be changed.

And Still More …

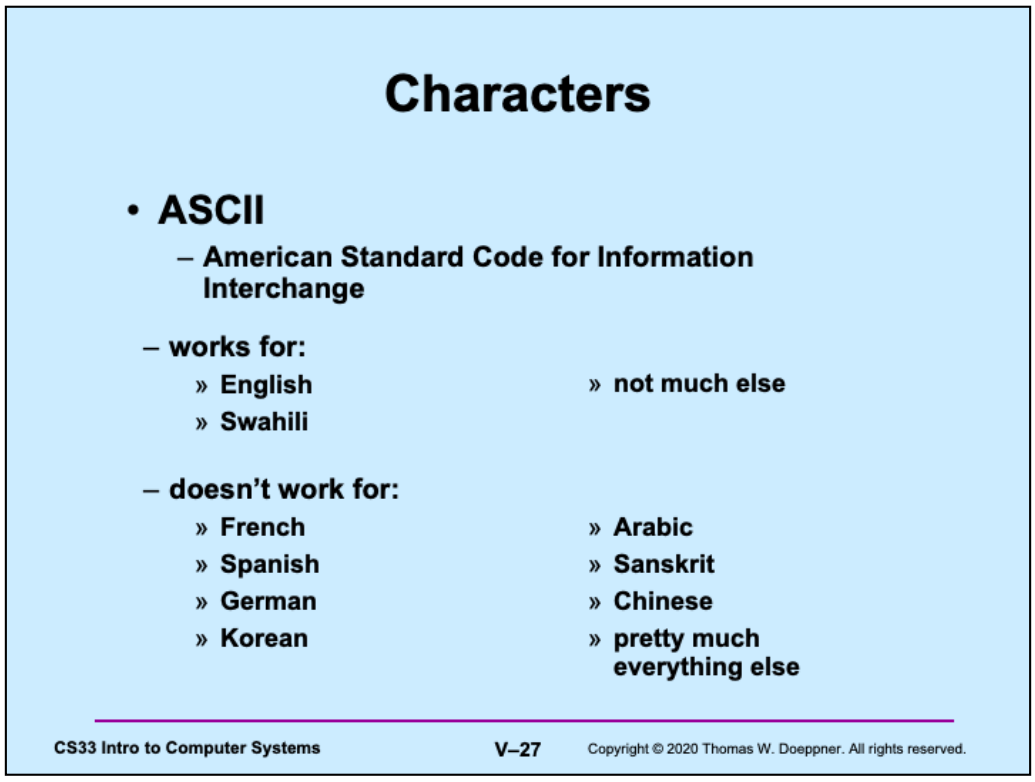
- **Array initialization**

```
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}};
```



The char, short, int, and long data types come in both signed and unsigned versions. What's shown here are the signed versions. We'll discuss the unsigned versions in an upcoming lecture.

Note that the exponent used for float and double is base 2, so that the limits given here are approximate. We will discuss the representation of the basic data types in much more detail soon.



ASCII is appropriate for English. European colonial powers devised written forms of some languages, such as Swahili, using the English alphabet. What differentiates the English alphabet from those of other European languages is the absence of diacritical marks. ASCII has no support for characters with diacritical marks and works for English, Swahili, and very few other languages. (Swahili may be written in either a latin script, which can be represented in ASCII, as well as an arabic script, which doesn't have a standard ASCII representation. See https://www.omniglot.com/writing/swahili.htm.)

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

The Unicode standard first came out in 1991. It defines a number of character encodings. UTF-8, in which each character is represented with one to four bytes, is the most commonly used, particularly on web sites. Being variable in length, its decoding requires more computation than fixed-width character encodings. Unicode also defines some fixed-with encodings, but these require more space than variable-width encodings.

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		

ASCII uses only seven bits. Most European languages can be coded with eight bits (but not seven). Many Asian languages require far more than eight bits.

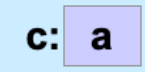
This table is a bit confusing: it's presented in column-major order, meaning that it's laid out in columns. Thus the value of the character '0' is 48, the value of '1' is 49, the value of '2' is 50, the value of '3' is 51, etc. Note that there are no printable characters in the "20" column.

*char*s as Integers

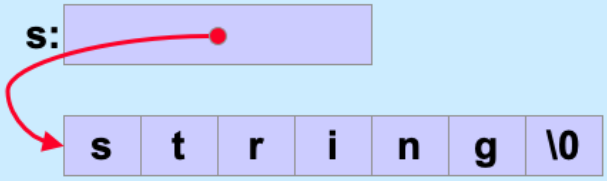
```
char tolower(char c) {
  if (c >= 'A' && c <= 'Z')
      return c + 'a' - 'A';
  else
      return c;
}
```

Character Strings

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




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

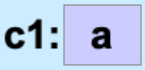


Is there any difference between *c1* and *c2* in the following?

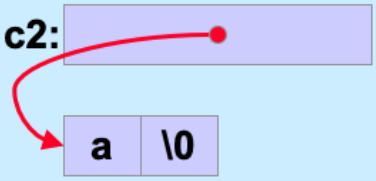
```
char c1 = 'a';
char *c2 = "a";
```

Yes!!

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



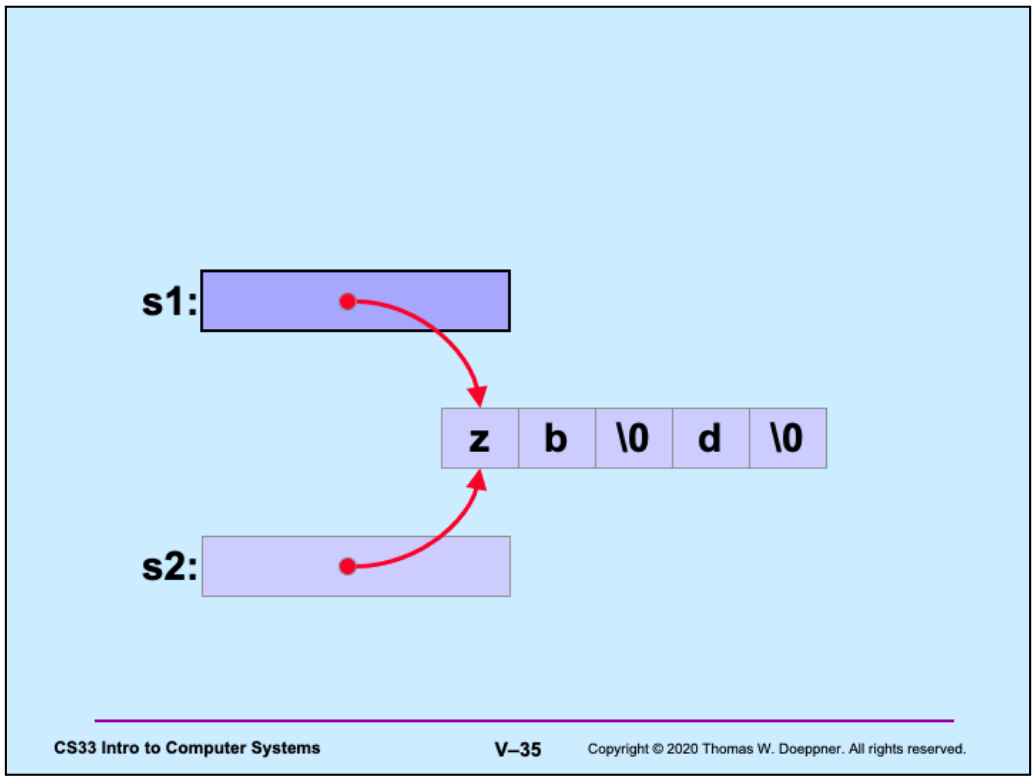

```
char *c2 = "a";
```



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

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


Note that the declaration of s1 results in the allocation of 5 bytes of memory, into which is copied the string “abcd” (including the null at the end).



Note that if either s1 or s2 is printed (e.g., "printf("%s", s1)), all that will appear is "zb" — this is because the null character terminates the string. Recall that s1 is essentially a constant: its value cannot be changed (it points to the beginning of the array of characters), but what it points to may certainly be changed.

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
```

String constants are stored in an area of memory that's made read-only, thus any attempt to modify them is doomed. In the example, s1 is a pointer that points to such a read-only area of memory. This is unlike what was done two slides ago, in which the string in read-only memory was copied into read-write memory pointed to by s1.

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;
// ?
```


The answer to the first question is no: the assignment will be considered a syntax error, since the value of s2 is constant. What we really want to do is copy the array pointed to by s1 into the array pointed to by s2.

It would not work if s2 were declared simply as a pointer. The original s2, declared as an array, has 5 bytes of memory associated with it, which is sufficient space to hold the string that's being copied. Thus the original s2 points to an area of memory suitable for holding a copy of the string. The second s2, being declared as simply a pointer and not given an initial value, points to an unknown location in memory. Copying the string into what s2 points to will probably lead to disaster.

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!

The answer, of course, is that the first for loop doesn't work, since there's not enough room in the array referred to by s2 to hold the contents of the array referred to by s1. Note that "&&" is the AND operator in C.

The correct way to copy a string is shown in the code beginning with the second for loop, which checks the length of the target: it copies no more than 4 bytes plus a null byte into s2, whose size is 5 bytes.

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
```


sizeof(s1) yields the size of the variable s1, which, on a 64-bit architecture, is 8 bytes.

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
$
```


sizeof(s) is 5 because 5 bytes of storage were allocated to hold its value (including the null).

sizeof(s1) is 8 because it's a pointer to a char, and pointers occupy 8 bytes.

sizeof(s2) is 12 because 12 bytes of storage were allocated for it.

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?
}
```


Note that comparing s1 and s2 simply compares their numeric values as pointers, it doesn't take into account what they point to.

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 for loop finds the first position at which the two strings differ. The rest of the code then determines whether the two strings are identical (if so, they must be of the same length), and if not, it determines which is less than the other. The function returns -1 if s1 is precedes s2, 0 if they are identical, and 1 if s2 precedes s1.