CS 33

Introduction to C
Part 5

Scope and For Loops (1)

```
int A[100];
for (int i=0; i<100; i++) {
  // i is defined in this scope
  A[i] = i;
}
```


It's often convenient to declare a for loop's index variable in the for loop, as shown in the slide.

Scope and For Loops (2)

```
int A[100];
initializeA(A);
for (int i=0; i<100; i++) {
  // i is defined in this scope
  if (A[i] < 0)
    break;
}
if (i != 100)
  printf("A[%d] is negative\n", i);
```

syntax error: reference to *i* is out of scope.

But be careful – the scope of such an index variable does not extend outside of the for loop.

Lifetime

```
int count;

int main() {
   func();
   ...
   func(); // what's printed by func?
   return 0;
}

int func() {
   int a;
   if (count == 0) a = 1;
   count = count + 1;
   printf("%d\n", a);
   return 0;
}
```

```
% ./a.out
1
-38762173
```


Even though **a** is given a value the first time **func** is called, on **func**'s second invocation **a** is not given a value and thus the result that's printed is "undefined". This is because the lifetime of **a** is just for the length of time its scope is active, which is from when the execution of **func** starts to when **func** returns. The **a** in the next invocation of **func** is different from the previous **a**.

Lifetime (continued)

```
int main() {
   func(1); // what's printed by func?
   return 0;
}
int a;
int func(int x) {
   if (x == 1) {
      a = 1;
      func(2);
      printf("%d\n", a);
   } else
      a = 2;
   return 0;
}
```

```
% ./a.out
2
```


In this case, **a** is global and thus the value set for it in one invocation of **func** is still there for the next invocation – the lifetime of **a** is that of the program itself.

Lifetime (still continued)

```
int main() {
   func(1); // what's printed by func?
   return 0;
}

int func(int x) {
   int a;
   if (x == 1) {
      a = 1;
      func(2);
      printf("a = %d\n", a);
   } else
      a = 2;
   return 0;
}
```

```
% ./a.out
1
```


Here **a** is local again. **func** is called (recursively) from within itself: the recursive invocation of **func** modifies a different **a** than is used in the first invocation. Thus, the value printed is 1.

Lifetime (more ...)

```
int main() {
   int *a;
   a = func();
   printf("%d\n", *a); // what's printed?
   return 0;
}

int *func() {
   int x;
   x = 1;
   return &x;
}
```

```
% ./a.out
23095689
```


When a function returns, its local variables become out of scope and no longer active – the lifetime of local variables is from the instant the function is called to when it returns. Thus, a pointer to a local variable refers to an undefined value if the variable is of a function invocation that is no longer active.

Lifetime (and still more ...)

```
int main() {
   int *a;
   a = func(1);
   printf("%d\n", *a); // what's printed?
   return 0;
}

int *func(int x) {
   return &x;
}
```

```
% ./a.out
98378932
```

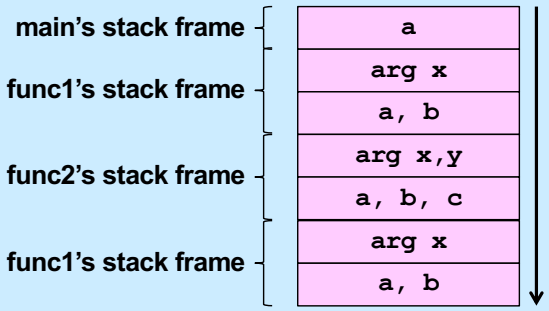

Similarly, the lifetime of function arguments is the same as the lifetime of the function.

Rules

- Global variables exist for the duration of program's lifetime
- Local variables and arguments exist for the duration of the execution of the function
  - from call to return
  - each execution of a function results in a new instance of its arguments and local variables

Implementation: Stacks

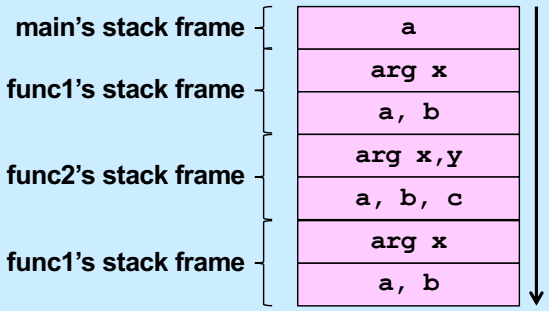
```
int main() {
   int a;
   func1(0);
   ...
}
int func1(int x) {
   int a,b;
   if (x==0) func2(a,2);
   ...
}
int func2(int x, int y) {
   int a,b,c;
   func1(1);
   ...
}
```



Function calling in C (and in most other languages) is implemented on stacks. Associated with an invocation of a function is a stack frame, which contains, among other things, its arguments and local variables. When a function is called, a stack frame for it is pushed onto the stack. When it returns, its stack frame is popped off the stack.

Implementation: Stacks

```
int main() {
   int a;
   func1(0);
   ...
}
int func1(int x) {
   int a,b;
   if (x==0) func2(a,2);
   ...
}
int func2(int x, int y) {
   int a,b,c;
   func1(1);
   ...
}
```



Each of the functions returns to its caller.

Quiz 1

```
void func(int a) {
   int b=2;
   if (a == 1) {
      func(2);
      printf("%d\n", b);
   } else {
      b = a*(b++)*b;
   }
}
int main() {
   func(1);
   return 0;
}
```

- **What's printed?**
  - **a) 0**
  - **b) 1**
  - **c) 2**
  - **d) 4**

Static Local Variables

```
int *sub1() {
  int var = 1;
  …
  return &var;
  /* amazingly illegal */
}
```

```
int *sub2() {
  static int var = 1;
  …
  return &var;
  /* (amazingly) legal */
}
```

- **Scope**
  - **like local variables**
- **Lifetime**
  - **like global variables**
- **Initialized just once**
  - **when program begins**
  - **implicit initialization to 0**

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**, is set to 1 when the program starts, and not every time **sub2** is called.

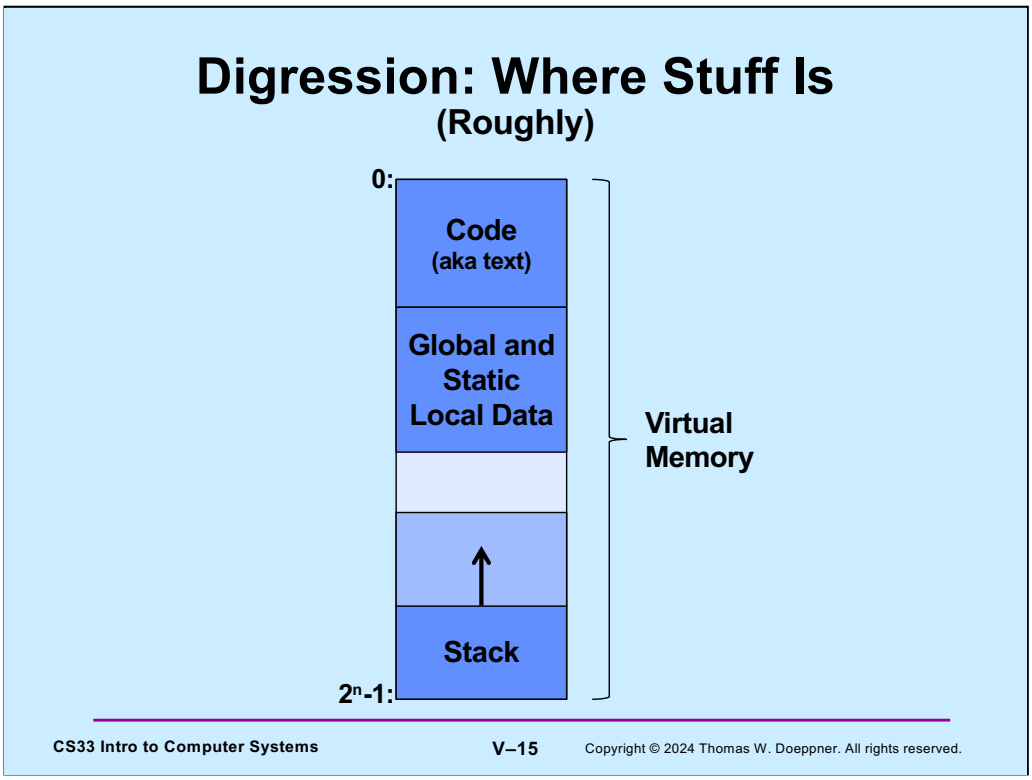
Quiz 2

```
int sub() {
  static int svar = 2;
  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 it is organized. What's shown in the slide 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.

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?**

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.

Conditional Compilation

```
#ifdef DEBUG
  #define DEBUG_PRINT(a1, a2) printf(a1,a2)
#else
  #define DEBUG_PRINT(a1, a2)
#endif
```

```
int buggy_func(int x) {
   DEBUG_PRINT("x = %d\n", x);
     // printed only if DEBUG is defined
   ...
}
```


One can define DEBUG simply by using the statement

#define DEBUG

or by supplying the flag –D=DEBUG to gcc.

Note that in addition to #ifdef (which should be read "if defined"), there's also #ifndef (which should be read "if not defined"). Thus, the code in slide could also be written

```
#ifndef DEBUG
  #define DEBUG_PRINT(a1, a2)
#else
  #define DEBUG_PRINT(a1, a2) printf(a1,a2)
#endif
```

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;
}
```


This works, but note that the entirety of the struct arguments are copied to the function, and the entirety of the result is copied back to the caller. This is no big deal here, but, for larger structures, it might be a problem (due to the time required to copy the data).

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;
```

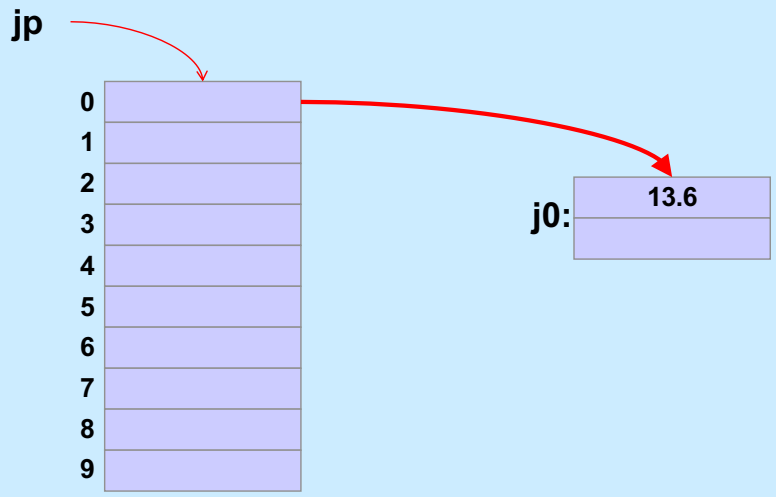
Arrays, Pointers, and *structs*

```
/* 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 ComplexNumbers**.

Memory View



Quiz 3

```
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) **prints something and terminates**
  b) **seg fault**
  c) **syntax error**

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) **prints something and terminates**
  b) **seg fault**
  c) **syntax error**

Quiz 5

```
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) **prints something and terminates**
  b) **seg fault**
  c) **syntax error**

Quiz 6

```
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) prints something and terminates
  b) seg fault
  c) syntax error

Structures vs. Objects

- **Are structs objects?**

NO!

(What's an object?)


```
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: array variables refer to the
   // addresses of the first elements

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


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}};
```

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**

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 either as 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.

Some of the characters require some explanation. '\a' is the alarm or bell character: it rings a bell. '\b' is the backspace character. '\t' is the horizontal tab character (usually referred to just as "tab"). '\n' is the newline character. '\v' is the vertical tab character. '\f' is the form-feed character, and '\r' is the carriage-return character. Some of these characters are rarely, if ever, used.

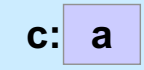
*char*s as Integers

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

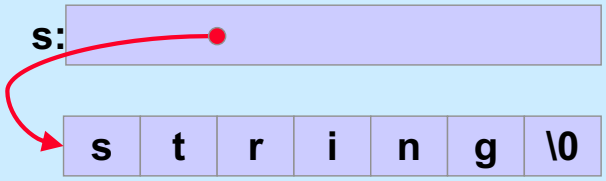

A variable of type **char** may be thought of as an 8-bit **int**.

Character Strings

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




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

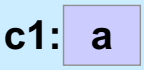


Is there any difference between *c1* and *c2* in the following?

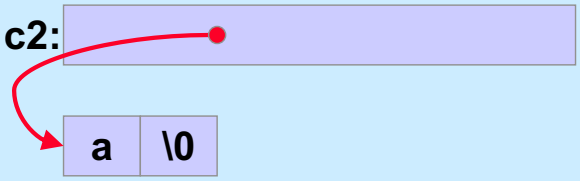
```
char c1 = 'a';
char *c2 = "a";
```

Yes!!

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



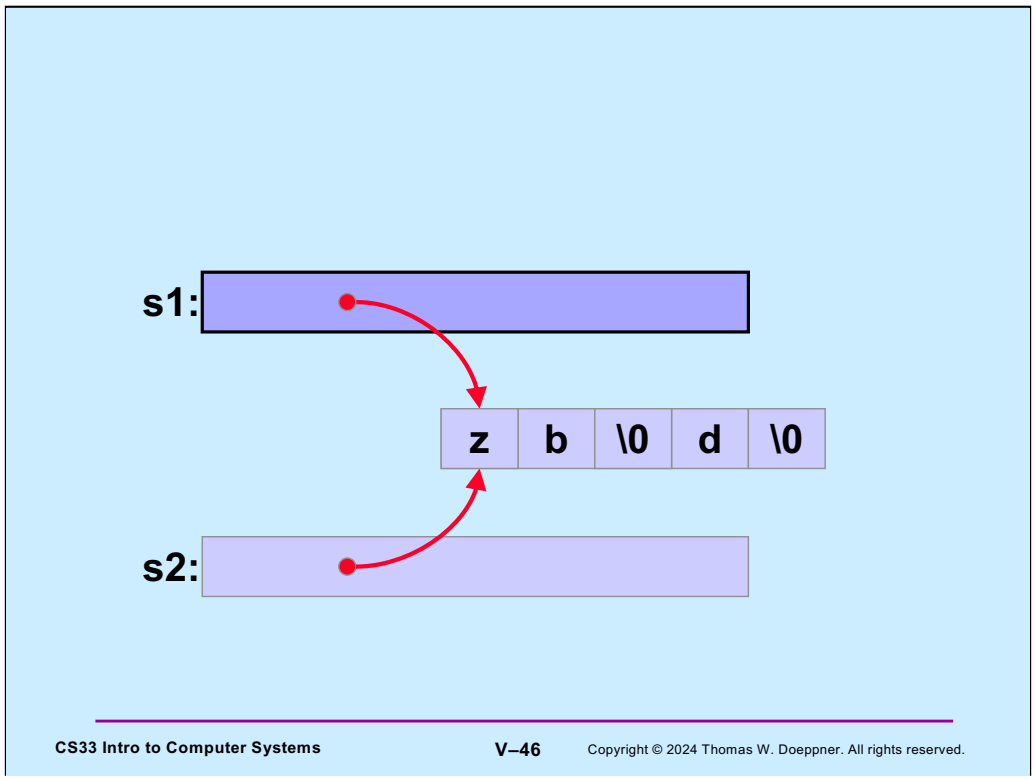

```
char *c2 = "a";
```



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 read-only, ensuring that they really are constants; 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**.