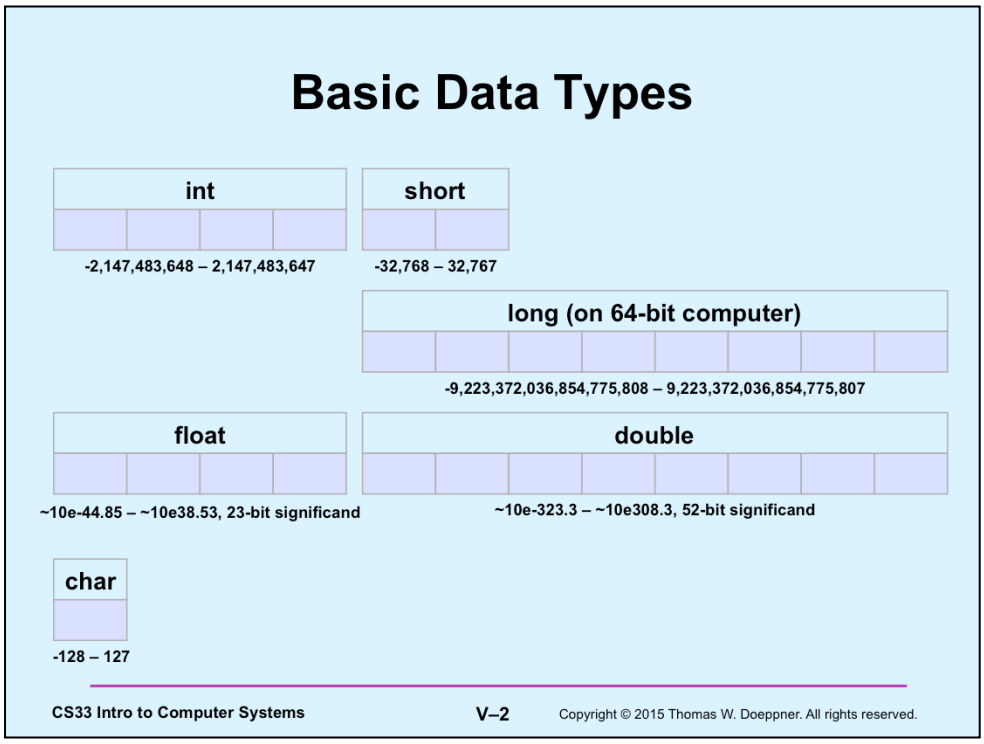
CS 33

Introduction to C
Part 5



The floating-point representation is as supported on the Intel X86 architecture. Note that the exponent is base 2, so that the limits given are approximate. We will discuss the representation of the basic data types in much more detail soon.

Characters

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

ASCII is appropriate for English. English-speaking missionaries devised the written form 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 thus has no characters with diacritical marks and works for English, Swahili, and very few other languages.

Who cares!!

You should care …

(but not in this course)

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. Many Asian languages require far more than eight bits.

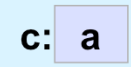
This table is a bit confusing: it's presented in column-major order, meaning that it's laid out in columns. Thus the value of the character '0' is 48, the value of '1' is 49, the value of '2' is 50, the value of '3' is 51, etc.

*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';
```



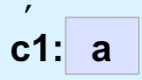

```
char *s = "string";
```

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

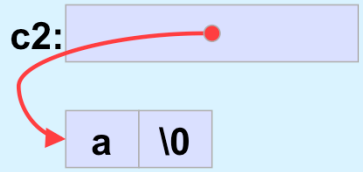
```
char c1 = 'a';
char *c2 = "a";
```

Yes!!

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



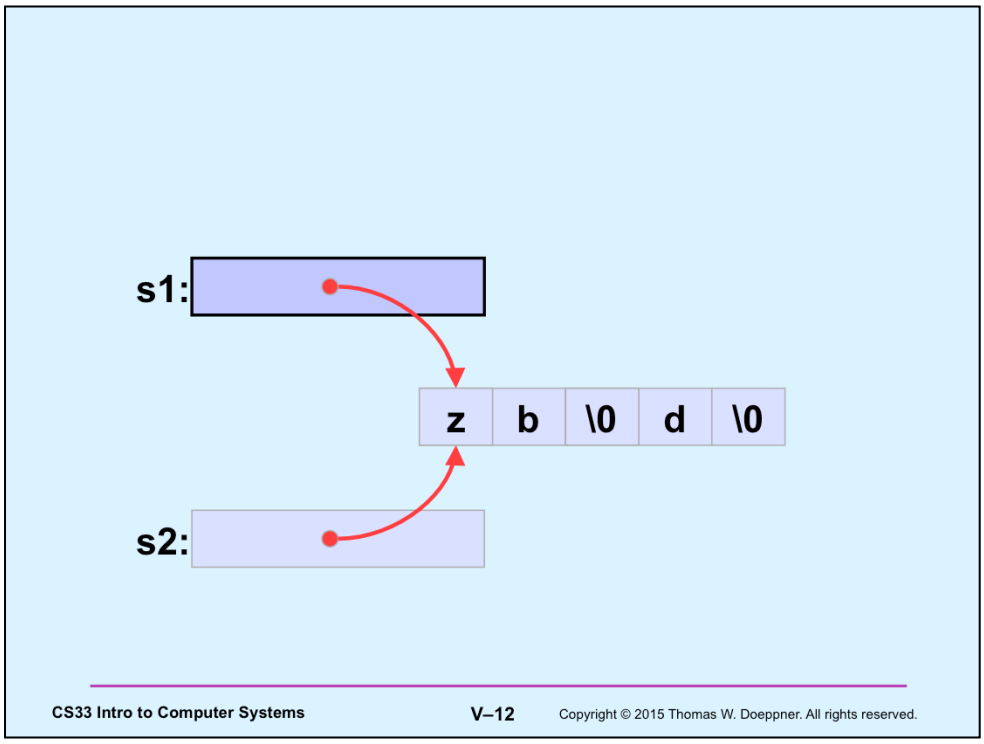

```
char *c2 = "a";
```



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';
```


The answer, of course, is that it 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 1

```
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 procedure returns -1 if s1 is precedes s2, 0 if they are identical, and 1 if s2 precedes 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
```


These will be useful in upcoming assignments.

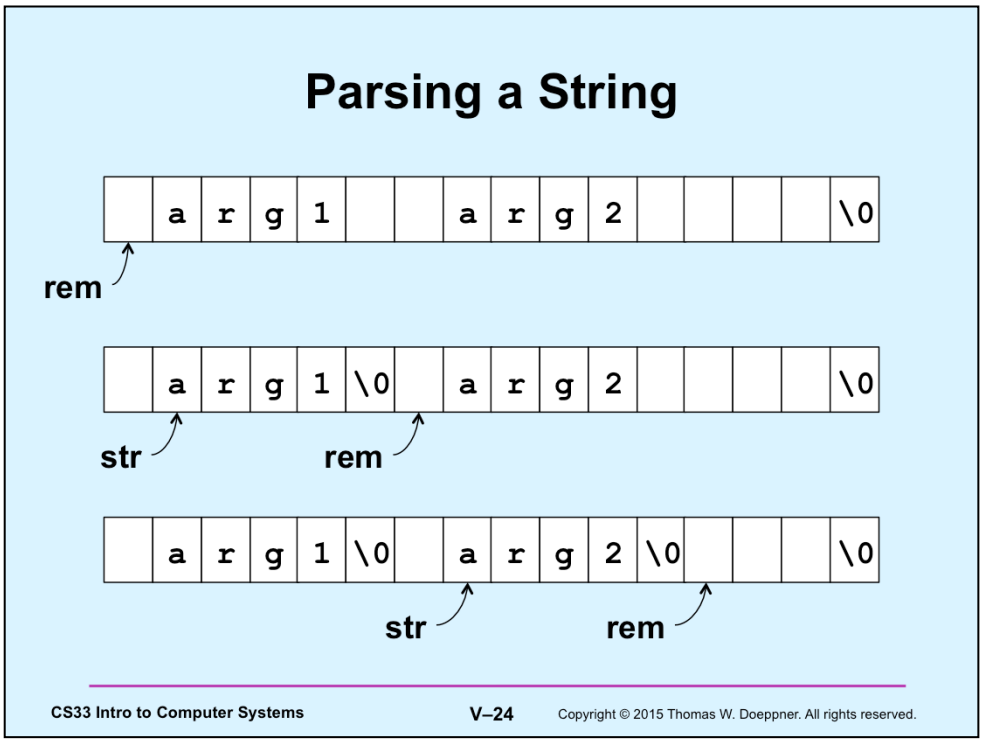
Quiz 2

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



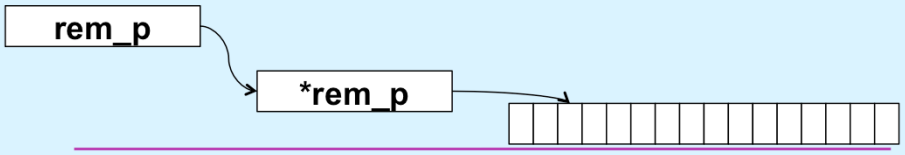
Suppose we have a string of characters typed into the command line of a shell. We'd like to parse this string to pull out individual words (to be used as arguments to a command); these words are separated by one or more characters of white space. Starting with a pointer to this string (*rem*), we call a function that returns a pointer to the first word (*str*), and updates *rem* to point to the remainder of the string. We call it again to get the second word, etc.

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 of entirely 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



Note that the argument is a char **. Thus we are passing *getfirstword* a pointer to a variable that points to the string. This allows us to change this variable inside of *getfirstword* so that it points to a different location in the string.

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;
}
```