

Section 3

Strings

1

strings are versatile

not just visible character strings like literals, but foundation for lowlevel access of memory

C has no built-in, native string type like other langs -- therefore no operators, no varying-length. everything done with function-calls

strictly, strings are a composite array type (array of characters), however for introduction, we can consider as a monolithic type.

look at constants first, then variables.

```
/* str-1.c : string storage representation */
```

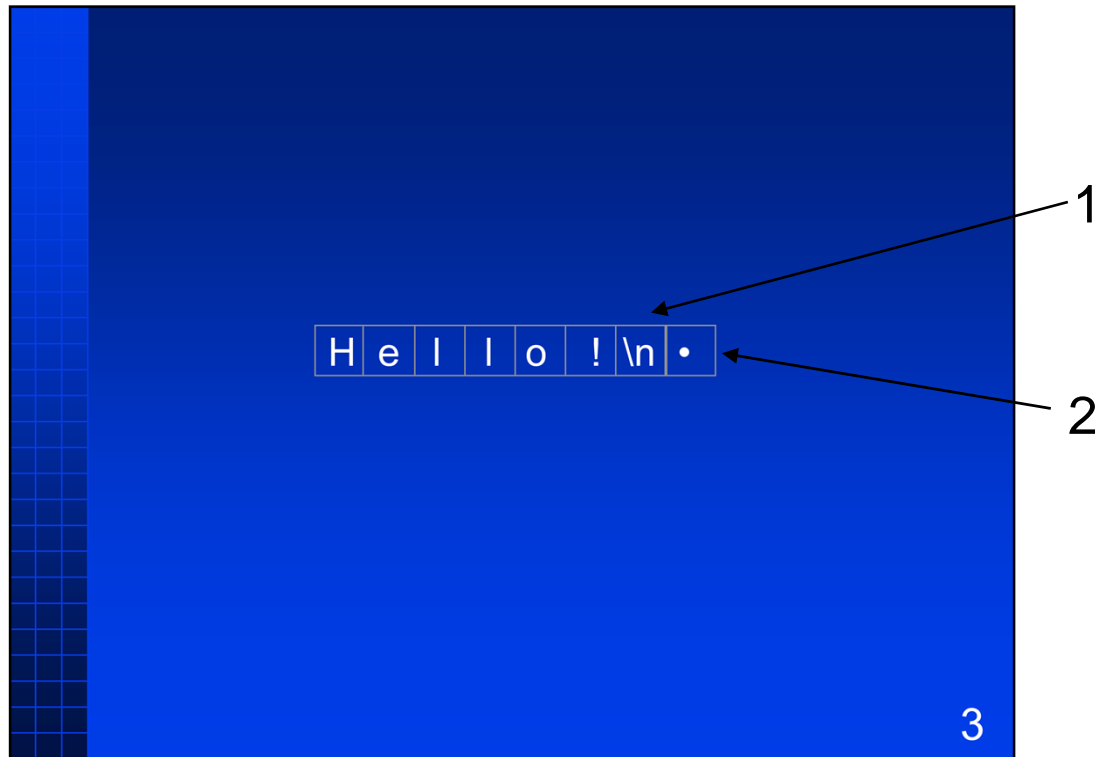
```
#include <stdio.h>
```

```
main()  
{  
    printf( "Hello!\n" );  
}
```

Hello!

2

Like the very first program, displays a string literal.
But how is the string represented in memory?



string takes 8 characters/bytes of storage: 6 visible, one for newline and one funny thing at end:

1) used to represent newline char that is escaped at compile-time. actually might be more, but we'll treat as one.

2) null character (character with character-set code of 0). nullchar is used to mark the end of the string (will elaborate on this later).

Note length is not stored anywhere. length is implicit -- end marked by null, have to count every char from beginning to nullchar. have to do this every time -- computing length is non-trivial operation

```
/* str-1a.c : string constants */
```

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

```
#define GREETING "Hello!\n"
```

```
main()
```

```
{
```

```
    printf( GREETING );
```

```
}
```

Hello!

H	e	l	l	o	!	\n	•
---	---	---	---	---	---	----	---

4

Same program, printf string is defined as a constant.
Note that the \n is in the string -- *it's just a character*

```
/* str-1b.c : string constants */
```

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

```
#define GREETING "Hello!"
```

```
main()
```

```
{
```

```
    printf( "%s\n", GREETING );
```

```
}
```

Hello!

H	e	l	l	o	!	•
---	---	---	---	---	---	---

5

Slightly different -- the constant has no `\n`.

Instead, use a `printf` control-string with a string directive `%s`

like other `%` directives, substitutes values from parameter list; for strings, copy character-for-character

separates message content from control information

```
/* str-1c.c : string constants */

#include <stdio.h>

#define FIRST  "John"
#define INITIAL "O."
#define LAST   "Doe"

main()
{
    printf( "My name is %s %s %s.\n",
           FIRST, INITIAL, LAST );
}
```

6

some more string literal manipulation

My name is John O. Doe.

J o h n •

O . •

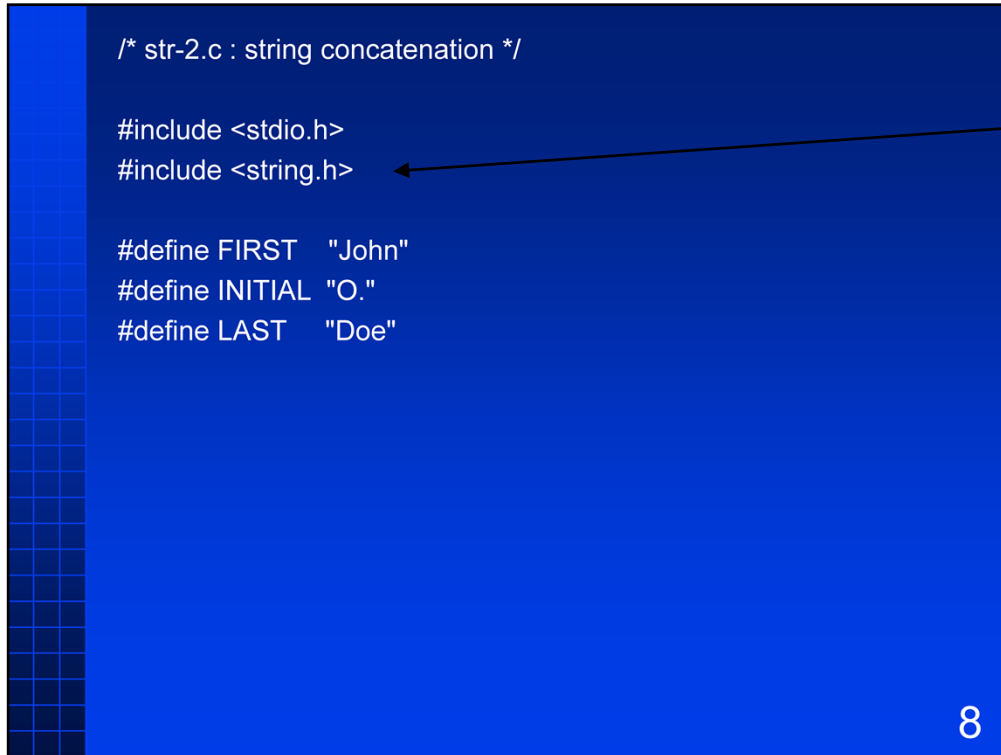
D o e •

Output of proceeding, storage representation

```
/* str-2.c : string concatenation */

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

#define FIRST  "John"
#define INITIAL "O."
#define LAST   "Doe"
```



(flip forward and back, two-slide sequence)

Now, more stuff: string variables and operations.

1) standard header file <string.h>

- defines standard functions for string manipulation
- as noted, use libraries rather than built-in (this eg is simple string concatenation) [C designed to be efficient at calling]
- null char is significant -- string functions generally treat null as end-of-string


```

main()
{
    char name[ 50 ];

    strcpy( name, FIRST );
    strcat( name, " " );
    strcat( name, INITIAL );
    strcat( name, " " );
    strcat( name, LAST );
    printf( "My name is %s.\n", name );
}

```

My Name is John O. Doe.

9

2) declaration of a string variable: informally, name is a 50-character string
 - can hold string from 0 to 50 characters

3) strcpy "string copy" -- make a copy of a string (destination, must be a variable; source, can be a variable or a constant or literal)

- method: character-by-character copy until null char is encountered. null char is copied.

4) strcat "string concatenate" -- append the source string (2nd parameter) to the destination (first parameter)

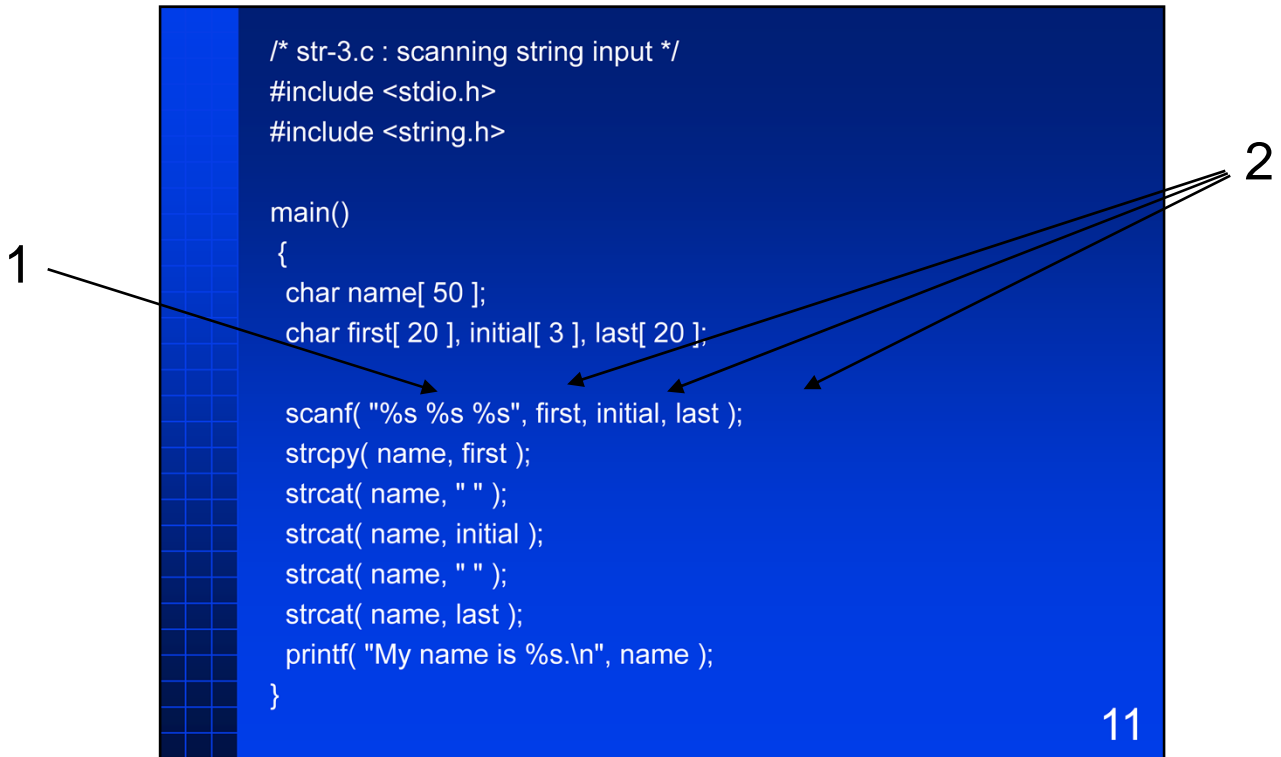
- method: find end of destination by finding nullchar, then char-by-char copy. nullchar end of first is removed (overwritten), then new nullchar is placed at end of concatenation.

Values of the character array

											...	
J	o	h	n	
J	o	h	n		
J	o	h	n		O	
J	o	h	n		O	
J	o	h	n		O	.			D	o	e	.

10

build up string in pieces



Using scanf to read string values

1) same idea as numbers; %s directive for strings

2) no & here:

- easy explanation: its a rule (strings have no &)
- hard explanation: & creates a reference, array names are already a reference [the expression value of an integer variable name is the contents of the variable, the expression value of an array variable name is the address of the array]

reading strings skips whitespace (blanks, tabs, newlines) => no blank-embedded strings

John O. Doe

My name is John O. Doe.

first: J o h n .

initial: O .

last: D o e .

12

sample input

```
/* str-4.c : string comparison */
```

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

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

```
main()
```

```
{
```

```
    char str1[ 50 ];
```

```
    char str2[ 50 ];
```

```
    scanf( "%s %s", str1, str2 );
```

13

C supports concept of relational comparisons between strings (strings can be equal or not; relation can be ordered: one string can be less than another)

Ordering based on binary values of character-set -- ordering is system-dependent. these days, code pages, internationalization, dbcs make character-string ordering more complicated than it used to be.

Program here reads two string with scanf and then compares them

```
if( strcmp( str1, str2 ) == 0 )
{
    printf( "The strings %s %s are equal.\n", str1, str2 );
}
else
{
    printf( "The strings %s %s are not equal.\n", str1, str2 );
}
}
```

- Also: strncmp, strcoll

14

In keeping with C philosophy, not part of language, no operators.

library function “strcmp”:

returns 0 if equal

returns negative if lhs < rhs

returns positive if lhs > rhs

note that this is not a “string equal” function, does not return true/false

for relations, shorter is lesser (null char is less than anything else)

strncmp: restrict number of characters in strings

strcoll: compare according to locale collating sequence

this this

The strings this this are equal.

this that

The strings this that are not equal.

the these

The strings the these are not equal.

samples, look at storage following

Diagram illustrating string comparison using character arrays. The strings are represented as arrays of characters in boxes, followed by an ellipsis and a final box, likely representing a null terminator.

String 1: t h i s

String 2: t h i s

String 3: t h i s

String 4: t h a t

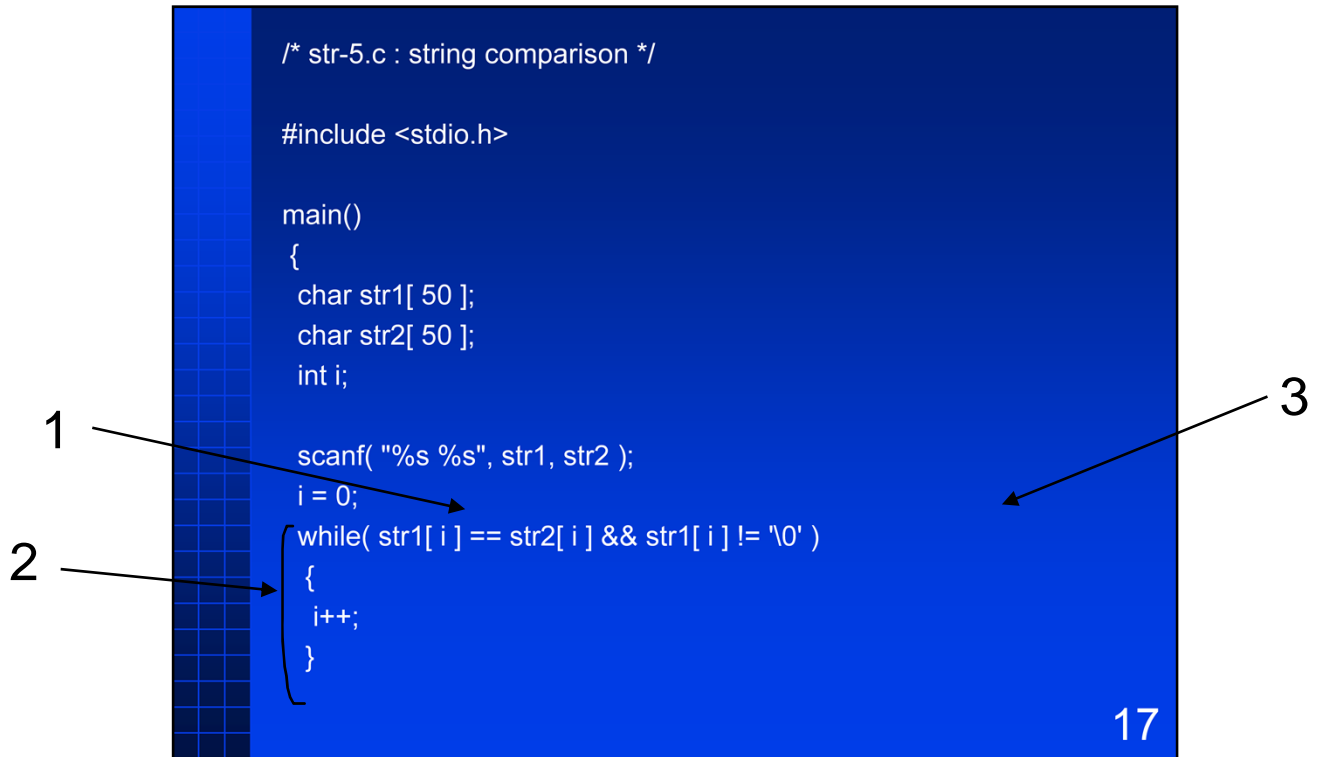
String 5: t h e

String 6: t h e s e

16

1. equal
2. not equal (this > that, returns +ve, i > a)
3. not equal (the < these, returns -ve, nullchar < s)

strcmp family compares character-by-character, whitespace is significant (eg leading, trailing blanks)



An example of character-by-character manipulation: determine if two strings are equal (subset of what `strcmp` does).

A bit of a look-ahead to array processing, since strings are really arrays of characters, but

1) array subscripting operation: selects *i*-th element from string.

note *i* starts at 0

2) loop structure: as long at same characters and not end-of-string, advance to next character (increment *i*)

3) null character constant: apostrophes (single-quotes) instead of doubles

at the end of this loop, strings are not equal; or end-of-string 1

if end-of-string 2, then we quit because not equal (unless also eos1)

Note possibilities for auto-increment?

```
if( str1[ i ] == str2[ i ] )
{
    printf( "The strings %s %s are equal.\n",
           str1, str2 );
}
else
{
    printf( "The strings %s %s are not equal.\n",
           str1, str2 );
}
}
```

18

we've stopped advancing through the loop, either

- end of string1
- two strings not equal

check to see which.

if eos1, then if equal then also eos2 and strings are equal

if str2 is shorter, would have quit because not equal

Lexical concatenation

```
/* str-6.c : lexical concatenation */
#include <stdio.h>
#include <string.h>

main()
{
    char long_const[ 200 ];

    strcpy( long_const, "Here is a string constant that "
                "will be very very long. Often, "
                "such strings are difficult to "
                "enter with text editors." );
    printf( "Long_const is %d characters long\n",
            strlen( long_const ) );
}
```

1

19

long constants hard to enter in some text-editors (less of a concern these days in windowed systems)

compile-time concatenation of constants -- adjacent string literals

1) strlen -- part of standard string library, returns number of characters from beginning up to but not including null char, number of "visible" characters (misnomer?)



Long_const is 117 characters long

20

output

String Initialization

```
/* str-7.c : string initialization */
#include <stdio.h>
#include <string.h>

main()
{
    char month_january[ ] = "January";
    char name[ 50 ] = "unknown";

    printf( "Name [%s]? ", name );
    scanf( "%s", name );
}
```

1

2

21

can specify initial values for strings at compile time
called an initializer:

- 1) first example, string length unspecified, compiler will compute. most applicable to constant (maybe should use constant, like previous example)
- 2) second example specifies length and causes that much storage to be reserved. initial value must fit within that space (but need not be that long)