HIGH-LEVEL PROGRAMMING I

Character strings

by Prasanna Ghali

char data type

- Represents single character or glyph

Z is character variable

```
char Z       = 'Z';
char m       = 'm';
char nine    = '9';
char space   = ' ';
char newline = '\n';
char tab     = '\t';
char quote   = '\'';
char slash   = '\\';
char plus    = '+';
char star    = '*';
```

'Z' is character constant

ASCII representation (1/4)

- Internally, C represents each char type as integer using an encoding scheme - ASCII is most commom
- ASCII (and other encoding schemes) assume:
  - Digits are sequentially numbered
  - Lowercase Latin letters are sequentially numbered
  - Uppercase Latin letters are sequentially numbered

ASCII representation (2/4)


```
char ucA  = 'A'; // ASCII value 65
char ucZ  = 'Z'; // ASCII value 90
char lca  = 'a'; // ASCII value 97
char lcz  = 'z'; // ASCII value 122
char zero = '0'; // ASCII value 48
char nine = '9'; // ASCII value 57
printf("%c: %i | %c: %i\n", ucA, ucA, ucZ, ucZ);
printf("%c: %i | %c: %i\n", lca, lca, lcz, lcz);
printf("%c: %i | %c: %i\n", zero, zero, nine, nine);
```

ASCII representation (3/4)

- Programmers often take advantage of C representing each `char` as integer

```
bool areEqual      = 'A' == 'A'; // true
bool earlierLetter = 'a' < 'f';  // true
bool laterLetter   = 'x' > 'z';  // false
char ucB           = 'A' + 1;
char lcx           = 'y' - 1;
int numuc_letters  = 'Z' - 'A' + 1;
int numlc_letters  = 'z' - 'a' + 1;
int num_digits     = '9' - '0' + 1;
```

ASCII representation (4/4)

- Programmers often take advantage of chars representing characters as small integers

```
// print lowercase characters in sequence
for (char ch = 'a'; ch <= 'z'; ++ch) {
  printf("%c", ch);
}
```

Character handling

- `<ctype.h>` header provides two kinds of functions:
  - character classification functions like `isdigit`
  - character case-mapping functions like `toupper`

Common <ctype.h> functions (1/2)

Function	Description
`isalpha(ch)`	true if *ch* is Latin character
`islower(ch)`	true if *ch* is `'a'` through `'z'`
`isupper(ch)`	true if *ch* is `'A'` through `'Z'`
`isspace(ch)`	true if *ch* is tab, space, new line
`isdigit(ch)`	true if *ch* is `'0'` through `'9'`
`toupper(ch)`	return uppercase equivalent of letter *ch*
`tolower(ch)`	return lowercase equivalent of letter *ch*

Common <ctype.h> functions (2/2)


```c
#include <ctype.h>

int is_latin = isalpha('a'); // true
is_latin     = isalpha('$'); // 0

int bigF     = isupper('f'); // 0
bigF         = toupper('f'); // 'F'
bigF         = toupper('#'); // '#'

int digit    = isdigit('0'); // true
```

Character I/O

- Variety of functions: `putchar`, `putc`, `fputc`, `getchar`, `getc`, `fgetc`

Character I/O: File copy


```
int ch;
while ((ch = getchar()) != EOF) {
  putchar(ch);
}
```

```
int ch;
while ((ch = fgetc(stdin)) != EOF) {
  fputc(ch, stdout);
}
```

```
int ch;
while ((ch = getc(stdin)) != EOF) {
  putc(ch, stdout);
}
```

Character I/O: Line counting


```
int ch, line_cnt = 0;

while ((ch = fgetc(stdin)) != EOF) {
  line_cnt = ('\n' == ch) ? line_cnt+1 : line_cnt;
}
printf("Line count: %d\n", line_cnt);
```

What is a string?

- *String* data structure is collection or grouping of characters
- Important in computer science because many computer applications are concerned with manipulation of textual data in addition to numerical data
- C implements strings using array of char elements

Character array (1/2)

- *Character array* is array whose components are of type char

```
char city[] = {'s', 'e', 'a', 't', 't', 'l', 'e'};
```

index	0	1	2	3	4	5	6
city	's'	'e'	'a'	't'	't'	'l'	'e'

```
char city[] = {'s', 'e', 'a', 't', 't', 'l', 'e'};
for (int i = 0; i < sizeof(city); ++i) {
  printf("%c", city[i]);
}
printf("\n");
```

Character array (2/2)

- In C, *not all* character arrays are strings

Character string

- *Character string* (aka *c-string*) is character array plus one extra character – the *null character* – to mark end of string
  - Null character is byte with all bits zeroed out and is represented by character `'\0'`
  - Null character is represented in ASCII by symbol NUL

```
char city[] = {'s', 'e', 'a', 't', 't', 'l', 'e', '\0'};
```

String literals (1/3)

- String literal (or string constant) is sequence of characters enclosed in double quotes

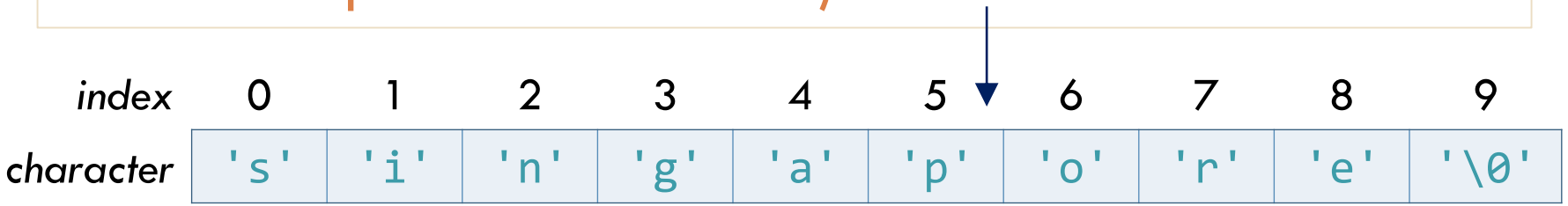
```
"singapore"
```

type is `char [10]`

But since we're dealing with string literal, treat it as:

```
char const str[10]
```

`sizeof("singapore")` evaluates to `10` (number of characters plus null character)



String literals (2/3)

evaluates to 10

```
for (int i = 0; i < sizeof("singapore"); ++i) {
  printf("%c", "singapore"[i]);
}
printf("\n");
```

String literals (3/3)

- Empty string: `""`

type is `char [1]`
`sizeof("")` evaluates to 1
(zero characters plus null character)

index	0
character	`'\0'`

New line escape sequence

- Each `'\n'` causes cursor to advance to next line

```
printf("Candy\nIs dandy\nBut liquor\nIs quicker.\n --Ogden Nash\n");
```

- Output of printing string literal to standard output:

```
Candy
Is dandy
But liquor
Is quicker.
 --Ogden Nash
```

Multiline string literals (1/2)

- Backslash \ can be used to continue string literal from one line to next

```
printf("When you come to a fork in the road, take it.\
Yogi Berra\n");
```

- Both lines will be printed to standard output as single line

Multiline string literals (2/2)

- Better way to deal with multiline string literals is to take advantage of rule that says two adjacent string literals combined into single string

```
printf("When you come to a fork in the road, take it."
       " Yogi Berra\n");
```

Operations on string literals (1/3)

- Since string literals are char arrays, they can be subscripted

```
char ch = "seattle"[2];
```

ch is initialized with 'a'

Operations on string literals (2/3)

- Don't try to change string literals – they're *literals*!!!

```
"digipen"[4] = 'd';
```

Program behavior is undefined.
The program may crash and burn

Operations on string literals (3/3)

- Function to convert decimal number between 0 and 15 to hexadecimal representation

```
char digit_to_hex(int digit) {
  return "0123456789ABCDEF"[digit];
}
```

String literal vs. character constant

- String literal with single character *not same* as character constant
- What is difference between `"a"` and `'a'`?
- What is difference between arguments in following function calls?

```
printf("a");

printf('a');
```

`printf` is passed integer value of `'a'` as address of 1st element of array of null-terminated `char`s causing undefined behavior [more likely a hard crash of program

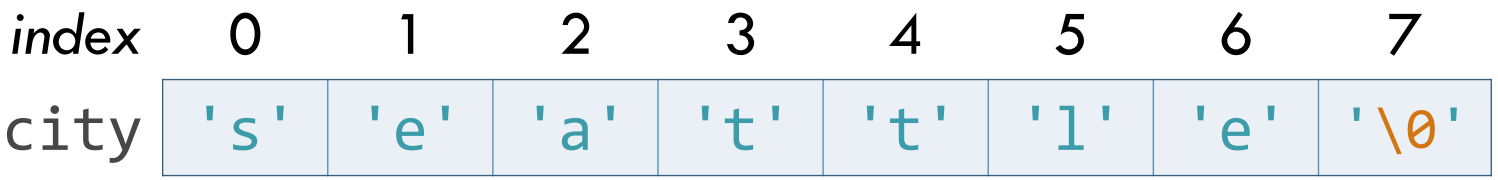
String variable

- Array variable that holds sequence of null-terminated characters
  - Array that holds sequence of characters is ***not*** a string
- To store N characters in string, require array variable's size be at least N+1

Defining and initializing string variables (1/4)

- String is array of null-terminated characters

```
char city[] = {'s', 'e', 'a', 't', 't', 'l', 'e', '\0'};
```




```
char city[] = {'s','e','a','t','t','l','e','\0'};
for (int i = 0; city[i] != '\0'; ++i) {
  fputc(city[i], stdout);
}
```

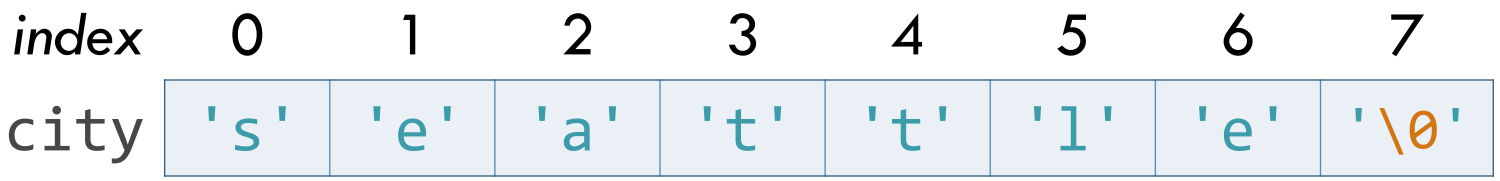
Defining and initializing string variables (2/4)

- C provides convenient shorthand to initialize string variables `char city[] = "seattle";`

Not a string literal!!!
In initializer context, `"seattle"` is shorthand for:

```
{'s', 'e', 'a', 't', 't', 'l', 'e', '\0'};
```

`city`'s memory storage will be no different:



Defining and initializing string variables (3/4)

- Initializer cannot be less than number of characters but can be same

```
char str[7] = "digipen";
```

- Since no room to add null character, compiler will not try to add one
- More convenient to let compiler determine number of characters from initializer and add one more for null character

```
char str[] = "digipen";
```

Defining and initializing string variables (4/4)

- If array size is larger than characters in initializer, then additional elements are zeroed out

```
char str[10] = "digipen";
```

str

'd'	'i'	'g'	'i'	'p'	'e'	'n'	'\0'	'\0'	'\0'

String I/O

- ~~`gets`~~, `fgets`, `puts`, `fputs`
  - Think of `gets` as deprecated

fgets and fputs

- Display contents of file to standard output

```
FILE *file = fopen("instructions.txt", "r");
if (file == NULL) {
  printf("Unable to open file!!!\n");
  exit(EXIT_FAILURE);
}

#define SIZE 81
char buf[SIZE];
while (fgets(buf, SIZE, file) != NULL) {
  fputs(buf, stdout);
}
```

Strings and functions

- No different than ordinary arrays except that string length is inherent in strings

```
// returns number of characters
// not including null character
int str_len(char const str[]) {
  int len = 0;
  while (str[len] != '\0') {
    ++len;
  }
  return len;
}
```

String handling

- `<string.h>` header provides string handling functions
  - Among most often used functions

Common <string.h> functions (1/2)

Function	Description
`strlen(str)`	Returns # of characters in *str* not including null-character
`strcmp(str1, str2)` `strncmp(str1, str2)`	Compares two strings; returns `0` if identical, `<0` if *str1* comes before *str2* lexicographically, `>0` if *str1* comes after *str2*. `strncmp` stops comparing after at most *n* characters
`strchr(str, ch)` `strrchr(str, ch)`	Character search: returns a pointer to first occurrence of *ch* in *str*, or NULL if *ch* was not found in *str*. `strrchr` finds the last occurrence.
`strcpy(dst, src)` `strncpy(dst, src, n)`	Copies characters in *src* to dest, including null-terminating character. Assumes enough storage in *dst*. Strings must not overlap. `strncpy` stops after at most *n* characters, but doesn't add null-terminating character.

Common <string.h> functions (2/2)

Function	Description
`strcat(dst, src)` `strncat(dst, src, n)`	Concatenate *src* onto end of *dst*. `strncat` stops concatenating after at most *n* characters. Always add a null-terminating character.
`strspn(str, accept)` `strcspn(str, reject)`	`strspn` returns length of initial part of *str* which contains only characters in *accept*. `strcspn` returns length of initial part of *str* which doesn't contain any characters in *reject*.