2.22 C Strings and pointer const correctness

Pointers and const

Strings as char pointers

EE108 – Computing for Engineers

1

Overview

Aims

- Learn how to use C Strings (character arrays/pointers) and the const keyword with pointers

Learning outcomes – you should be able to…

- Declare and use C strings in char pointer form
- Make basic use of const to indicate read-only for pointers

2

pointer const correctness

3

3

Pointers and const – why?

4

In C programming there are usually two types of text strings that we care about

- Constant strings (such as string literals) that should never change – these are sometimes called immutable strings
  - Generally you don't need to modify a string if you are printing it, searching through it, matching it, etc.
- Modifiable strings that our code should be able to modify – these are called mutable strings
  - Generally we need to modify strings when converting them in any way (e.g. lower to upper case), when reading in a string from an external source, etc.

Because of these differences it is important to distinguish between mutable and immutable strings

For this reason, we start with a discussion of the const keyword in conjunction with pointers…

4

Pointer const correctness

5

The **const** keyword actually means **read-only** rather than constant

- you cannot change the value of something that is declared const (although the value might potentially change by some other means)

When dealing with pointers there are two things to consider separately

- The pointer value itself (i.e. what it points at)
- The pointee value (i.e. what's pointed at by the pointer)

If the **pointer is const** (read only)

- you cannot change where it points after it has been initialized

If the **pointee value is const** (read only)

- you cannot modify the pointee value

24 November 2020

5

Different types of const

6

Most typical case: **pointer modifiable** and **pointee value modifiable**

```
int arr[] = { 100, 110, 120, 130 };
int x = 42;
int *ptr;

ptr = &x; // point at x

// pointee is modifiable so we can modify pointee value
*ptr = 43; // pointee (i.e. x) value is now 43

// pointer itself is modifiable so we can change what
// it points at
ptr = &arr[2]; // point at element 2 of array, arr
x = *ptr; // x is now the value of pointee (i.e. 120)

// NOTE: that we were able to change our pointer from
// pointing at an ordinary int variable to an element of
// an int array because in both cases all we need is a
// pointer to int
```

24 November 2020

6

Different types of const

7

Also commonly used: **pointer is modifiable** (so we can change what pointer points at) but **pointee is const** (i.e. read only, so we cannot modify pointee value)

```
int arr[] = { 100, 110, 120, 130 };
int x = 42;
const int *ptrToConst;

ptrToConst = &x; // point at x

// pointee is read only so we cannot modify the pointee
// value
// *ptrToConst = 43; // Illegal

// pointer itself is writable so we can change what
// it points at
ptrToConst = &arr[2]; // point at element 2 of arr

// pointee is read-only so it is OK to get/read its
// value
x = *ptrToConst; // x is now the value of pointee
                 // (i.e. 120)
```

24 November 2020

7

Different types of const

8

Less common: **pointer is const** (i.e. read only, so we cannot change what it points at) but the **pointee value is modifiable**

```
int arr[] = { 100, 110, 120, 130 };
int x = 42;
// NOTE 1: the const comes after the * to indicate the
//         pointer is const (read only)
// NOTE 2: because the pointer is const, where it points
//         must be set in an initializer
int * const constPtr = &x; // point at x

// pointee is modifiable so we can modify pointee value
*constPtr = 43; // pointee (i.e. x) value is now 43

// pointer itself is read only so we cannot change it
// constPtr = &arr[2]; // illegal - cannot change what
                       // it points at
```

24 November 2020

8

Different types of const

9

pointer is const (i.e. read only, so we cannot change what it points at) and the **pointee value is const** (i.e. read only)

```
int arr[] = { 100, 110, 120, 130 };
int x = 42;

// NOTE: because the pointer is const, where it points
//       must be set in an initializer
const int * const constPtrToConst = &arr[2]; // point at element 2
                                             // of arr

// when pointee is read-only it is still OK to get/read its
// value
x = *constPtrToConst; // x is now the value of pointee (i.e. 120)


// pointee is read-only so we cannot modify pointee value
// *constPtrToConst = 43; // illegal - cannot change pointee value

// pointer itself is read only so we cannot change it
// constPtr = &arr[2]; // illegal - cannot change what
                       // it points at
```

24 November 2020

9

Summary Guidelines

10

In a pointer definition

- Everything **to left of the * refers to the pointee type**. Therefore if the pointee should be read-only, put const somewhere before the *. Normally you put const as the first word of the type.
- Everything **to the right of the * refers to the pointer itself**. If the pointer should be read-only, put the const after the *.

```
int intArray[10] = { 1, 2, 3, 4 };
char str[20] =
  { 'H', 'E', 'L', 'L', 'O', '\0' };

// pointer read-write,
// pointee read-write
int * pInt = intArray;
char * pStr = str;

// pointer read-write,
// pointee read-only
const int * pInt = intArray;
const char * pStr = str;
```

```
int intArray[10] = { 1, 2, 3, 4 };
char str[20] =
  { 'H', 'E', 'L', 'L', 'O', '\0' };

// pointer read-only,
// pointee read-only
const int * const pInt = intArray;
const char * const pStr = str;

// pointer read-only,
// pointee read-write
int * const pInt = intArray;
char * const pStr = str;
```

24 November 2020

10

Self test questions

11

Declare an array to hold 4 floating point values, 10.0, 11.1, 12.2, 13.3.

Declare an array to hold 5 char values: 'a', 'b', 'c', 'd', and 'e'

Declare a pointer that should point at the floating point values and treat them as read-only. Initialize the pointer

Declare a pointer that should point at the characters but allow them to be modified. However the pointer itself must not be allowed to point anywhere else. Initialize the pointer

24 November 2020

11

Visualizing pointers and const

12

24 November 2020

12

C Strings and pointers

14

13

Char and single quotes

15

In C and related languages an individual printable character is stored in a variable of type char.

A char is really just a small, typically unsigned, integer value between 0 and 255 (or 0x00 to 0xFF in hex).

How does a number, like hex 0x61 get rendered? The answer is that, when printing chars, we normally interpret their values as **ASCII codes** (see table on right). For example, a char containing the value 0x51 would be printed as a Q character.

So that we don't have to remember ASCII codes, the C language converts any printable character surrounded by **single quotes** to it's corresponding ASCII code.

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

```
char c; // c will be used to hold small integer values suitable for printing

c = 'Q';  // c now contains the number 0x51 – ASCII code for Q
// c = Q; // illegal unless Q happens to be a valid variable name
c = '$';  // c now contains the number 0x24 – ASCII code for $
c = ' ';  // c now contains the number 0x20 – ASCII code for SPACE
c = '\n'; // c now contains the number 0x0A – ASCII code for newline
```


14

Char arithmetic

16

15

Strings and double quotes

17

NUL terminator

- In C, a string is an array of characters terminated by a special character called the NUL character or NUL terminator.
- It is written '\0' and has ASCII code 0

There is a special shorthand for strings of characters: **specify the contents of a string by surrounding the text by double quotes**

- C converts any text surrounded by double quotes to an array of characters, replacing each printable character by it's corresponding ASCII code and adding a NUL terminator character at the end
- E.g. "hello"
  is equivalent to the character array contents
  `{ 'h', 'e', 'l', 'l', 'o', '\0' }`
  which is equivalent to the numbers
  `{ 0x68, 0x65, 0x6C, 0x6C, 0x6F, 0x00 }`

16

Strings, char arrays, and char pointers

18

Strings whose contents can change (e.g. because they are being read from serial input), **must be stored in a character array**

- If the size of the string can change, the array must be big enough to hold the largest the string can be (and error checking is needed to make sure you prevent overflowing the array bounds)
- As usual a function that uses a char array can take a char array or char pointer parameter

Literal strings which should never be changed should generally be referred to using a **const char pointer** instead of an array

24 November 2020

17

Contd.

19

String as character array, initialized with a string literal

```
// allocate an array with space for 6 characters and initialize it
// with characters from string literal
char str1[] = "hello";
char str2[10] = "there"; // first 5 chars, then '\0', remainder all 0
```

The characters in str1 and str2 can be modified at any time. Both str1 or str2 are array names and cannot be changed to refer to anything else

⚠ Don't declare a string as a (read-write) char pointer to string literal such as below. This implies the ability to change the characters or the literal which doesn't make sense.

```
// allocate a pointer to a string and initialize it to point at a
// particular string literal
char *str3 = "hello"; // never do this
```

A pointer to a string literal should indicate that the characters of the literal (the pointee) are read-only by using const in the declaration as follows.

```
const char *str4 = "hello"; // indicates that string pointed at by
                            // str4 cannot change its contents
```

Note that the pointer is read-write so it may be changed to point at some other read-only chars if needed.

24 November 2020

18

Functions and strings (params and return values)

20

Generally strings can be treated like arrays

- The nul character used to terminate a string means that the end of the string or length of the string can always be identified
- Therefore we usually don't need to supply a separate length parameter

Functions that take string parameters:

- For read only strings that you will not modify: use a `const char *`
  - The caller may pass in a string literal or a character array
- For strings that will be modified in the function: use `char *`
  - The caller must not pass a string literal as a modifiable string
  - If the modification may change the length of the string, it is generally useful to pass the maximum allowed length of the string as an input parameter. (This length is usually the length of the array that will hold the modified characters.)

Functions that return strings:

- If returning a read-only string: declare a `const char *` return type
- If returning a modifiable string: declare a `char *` return type

24 November 2020

19

Function declaration example

21

```
// function declarations…

// fun1 takes a single input string whose contents cannot be modified
void fun1(const char *inputReadOnlyString);

// fun2 takes a modifiable string – this form is only
// suitable if the string length won't be changed – otherwise
// supply a maximum length also
void fun2(char *inoutWritableString);

// fun3 takes a modifiable string. A maximum length is specified so
// the function knows how many elements of the string it can write to
void fun3(char *outWritableString, int len);

// returns a writable string – this is not very common
char *fun4();

// returns a read only string – this is reasonably common
const char *fun5();

// WARNING: never return an array/string declared as an automatic
// local variable as the array/string will cease to exist when the
// function ends.
```

24 November 2020

20

Function call examples

22

```
// to call the funtions on previous slide…
#define LEN 10
char         writableBuf[LEN + 1] = "hello"; // the +1 allows space for the nul terminator
const char * readOnlyString       = "there";
char       * readWriteStr;

// fun1 doesn't modify its param, so can pass in either string
fun1(writableBuf);
fun1(readOnlyString);

// fun2 modifies its param so can only pass in writable string.
// Since fun2 has to figure out length from the parameter value
// it receives, it cannot use the full size of writableString.
fun2(writableBuf);
// fun2(readOnlyString); // won't compile

// fun3 modifies the contents of writableString and knows the length, // so it can use the
full array size
fun3(writableBuf, LEN);

// returns a writable string
readWriteStr = fun4();
readOnlyString = fun4(); // assigning read-write to read-only is safe

// returns a read only string
readOnlyString = fun5();
```

4 November 2020

21

Function definition example 1

23

```
// fun1 takes a single input string whose contents cannot be modified
// e.g. just print each character

void fun1(const char *inputReadOnlyString) {
  const char * p = inputReadOnlyString;

  while (*p != '\0')
    Serial.print(*p++);
}

// fun2 takes a modifiable string – this form is only suitable if the string length
// won't be changed – otherwise supply a maximum length also
// e.g. add 1 to each char < 0x60

void fun2(char *inoutWritableString) {
  char *p;

  for (p = inoutWritableString; *p != '\0'; p++)
    if (*p < 0x60)
      *p += 1;
}
```

24 November 2020

22

Function definition example 2

24

```
#define STRLEN 5

// fun3 takes a modifiable string. A maximum length is specified so
// the function knows how many elements of the string it can write to
// e.g. replace len characters with dots
void fun3(char *outWritableString, int len) {
  char *p;

  for (p = outWritableString; p < outWritableString + len; p++)
    *p = '.';
}

// fun5 returns a read only string - this is reasonably common

const char *fun5() {
  static char randomWord[STRLEN+1] = {0}; // must be static to prevent it disappearing!!
  int i;

  for (i=0; i < STRLEN; i++)
    randomWord[i] = 'a' + random(26);

  return randomWord;
}
```

24 November 2020

23

Iterating over a string (also seen on prev 2 slides)

25

```
// function which iterates over but does not modify the string
// (hence we use a read-only string parameter)
int countLowercaseVowels(const char *str) {
  const char* p;
  int count = 0;

  // this idiom for checking end of string is very common in C
  for (p=str; *p != '\0'; p++) {
    if (*p == 'a' || *p == 'e' || *p == 'i' || *p == 'o' || *p == 'u')
      count++;
  }
  return count;
}

// function which iterates over and modifies the string
// the string is returned as a convenience - we could just as easily
// have defined the function to have a void return type
char * replaceLowercaseVowels(char *str) {
  char* p;

  for (p=str; *p != '\0'; p++) {
    if (*p == 'a' || *p == 'e' || *p == 'i' || *p == 'o' || *p == 'u')
      *p = '.';
  }
  return str; // return the (now modified) string that was passed in
}
```

24 November 2020

24

Contd.

26

```
// using the functions on the previous slide

#define STR_LEN 10

void loop() {
  const char * fixedString = "hello world";
  char varString[STR_LEN + 1] = "howdy";

  // pass in const string to function taking a const string
  Serial.print("fixedString vowel count: ");
  Serial.println( countLowerCaseVowels(fixedString) );

  // pass in const string to function taking a non-const string – BAD!
  replaceLowercaseVowels(fixedString); // BAD!

  // pass in non-const string to function taking a const string - OK
  Serial.print("varString vowel count: ");
  Serial.println( countLowerCaseVowels(varString) );

  // pass non-const string to function taking a non-const string – OK
  // because replaceLowercaseVowels returns a string we can do…
  Serial.println( replaceLowercaseVowels(varString) );
  // …instead of…
  // replaceLowercaseVowels(varString);
  // Serial.println(varString);
}
```

24 November 2020

25

Self test questions

27

Q1. Declare a string buffer to hold up to 10 characters that will be read in from the serial port.

Q2. Declare a variable that can refer to a string literal or read-write string but will only be used for printing.

24 November 2020

26

Self test questions

28

Q3. Declare a variable that can refer to a read-write string and initialize it to refer to the string buffer from Q1.

Q4. Use the variable from Q3 to change the 3rd character of the string so that it would print as the number 5.

24 November 2020

27

Useful char functions in the standard library

29

#include <ctype.h>

int isalpha(int c) – is c an alphabetic character (by ascii code)

int isdigit(int c) – is c a digit (0-9) character (by ascii code)

int isxdigit(int c) – is c a hex digit (0-9, A-F) character (by ascii code)

int isalnum(int c) – is c an alphabetic or numeric character (by ascii code)

int iscntrl(int c) – is c a control character (by ascii code), form feed, bel, newline, etc

int ispunct(int c) – is c a punctuation character (by ascii code)

int isspace(int c) – is c a whitespace character (by ascii code), e.g. space, newline, tab

int isupper(int c) – is c an uppercase character (by ascii code)

int islower(int c) – is c a lowercase character (by ascii code)

Example:

```
char c = random(128); // random ASCII value

if (isalpha(c) && islower(c))
  doSomething();
```

24 November 2020

28

Contd.

30

#include <ctype.h>

int toupper(int c) – convert c to an uppercase character (by ascii code)

int tolower(int c) – convert c to a lowercase character (by ascii code)

Example:

```
char c = 'A' + random(26); // random uppercase letter

c = tolower(c); // convert character in c to lowercase equivalent
                // and save new (lowercase) character ascii code in c
```

24 November 2020

29

Examples using the char functions

31

```
// function which iterates over but does not modify the string
// (hence we use a read-only string parameter)
int countAlphabeticChars(const char *str) {
  const char* p;
  int count = 0;

  for (p=str; *p != '\0'; p++) {
    if (isalpha(*p)) // isalpha returns true is character is alphabetic
      count++;
  }
  return count;
}

// function which iterates over and capitalizes all alphabetic chars.
// the string is returned as a convenience - we could just as easily
// have defined the function to have a void return type
char * strToUpper(char *str) {
  char* p;

  for (p=str; *p != '\0'; p++) {
    if (isalpha(*p))
      *p = toupper(*p);
  }
  return str; // return the (now modified) string that was passed in
}
```

24 November 2020

30

Useful string functions in the standard library

32

#include <string.h>

strcpy and **strncpy** – copy one string to another*

strcat – concatenates strings by appending one to the other*

strlen – determine the length of a string

strcmp and **strncmp** – compare two strings

strstr – find first occurrence of a substring in a string (e.g. find "tom" in "atom")

Etc.

* the destination must be a writable character array that's big enough to hold the modified/new string incl. nul terminator

24 November 2020

31

Some examples using standard library string functions

33

```
#include <string.h>

#define STRLEN 20

void loop() {
  char         writableBuf[STRLEN+1] = "hello";
  const char * readOnlyString   = "there";
  int len;

  len = strlen(writableBuf);
  len = strlen(readOnlyString);
  …
  // compare strings, 0 means 0 differences, i.e. a match
  if (strcmp(writableBuf, readOnlyString) == 0)
    doSomething();
  // compare only first 3 charactes of the two strings
  if (strncmp(writableBuf, readOnlyString, 3) == 0)
    doSomething();
  …
  // concatenate readOnlyString to end of writableBuf
  strcat(writableBuf, readOnlyString);

 // copy literal string over current contents of writableBuf
  strcpy(writableBuf, "new string");
}
```

24 November 2020

32

serialPrintf and serial input

34

33

Printf – (note: cannot use directly on Arduino)

35

The standard I/O library in C defines functions for printing (and reading input) that we typically don't use as much in very small embedded systems, but you should know them:

- printf – print formatted to standard output (Console)
- fprintf – print formatted to a file
- sprintf – print formatted to a string

Using printf as the example:

- **int printf(const char *format, …)**
- The format defines the format/layout of text to printed and the … represents zero or more additional arguments whose values will be substituted into the format at the appropriate places, e.g.

```
char * username = …; // specific value unimportant
int number = …; // specific value unimportant

printf("hello number %d from %s\n", n, username);
// might print something like:
//
//   hello number 12 from Rudi
```

Warning: designed for desktop - doesn't work on Arduino

34

serialPrintf

36

In the EE108 library we have a function based on printf that acts almost identically called serialPrintf which uses the serial output to console. This does work on Arduino.

The allowed format substitutions are as follows:

- %c – print a single character
- %d or %i – a single signed integer,
  - for short int use %hd or %hi, for long int use %ld or %li
- %u – an unsigned integer, short and long modifiers as above
- %x – an unsigned hexadecimal integer, short and long modifiers as above
- %% - print the % character
- Lookup up printf format for additional info about left and right justified numbers etc.
- **IMPORTANT: Floating point values are NOT supported by serialPrintf**

Special characters begin with a backslash

- \n – the newline character
- \r – the carriage return character (not usually needed)
- \t – the tab character

24 November 2020

35

serialPrintf example

37

The following example code shows the basic usage of serialPrintf.

```
char * username = "Alice";
unsigned short number = 63;

serialPprintf("hello %s: the number was %hu decimal and %hx in hex\n", username, n, n);
// might print something like:   hello Alice: the number was 63 decimal and 3F in hex
```

Long format strings like this can use up too much RAM on the Arduino, so there is a custom compiler extension, the F() macro, which allows you to specify that the (literal) string should be stored in program flash memory rather than RAM, e.g.

```
char * username = "Alice";
unsigned short number = 63;

serialPrintf(F("hello %s: the number was %hu decimal and %hx in hex\n"), username, n, n);
// might print something like:   hello Alice: the number was 63 decimal and 3F in hex
```

24 November 2020

36

Reading serial input on arduino

38

Serial.available() – returns how many characters are available to be read

Serial.read() – reads a single character (often the most useful as you can examine each character as it is read and take action accordingly)

Serial.readBytes(buffer, len) – reads a specified number of characters into a character array buffer

There are other functions, but they are not so useful to us

24 November 2020

37

Basic serial input usage (blocking version)

39

```
void loop() {
  Serial.print("\n\nType some characters and hit return...\nreceived:");

  while (true) { // infinite loop, so we'll break out manually later
    char c;

    // blocking version - wait here until at least one character has been entered
    while (Serial.available() == 0)
      continue;

    // OK, we now have at least 1 character (we might have more)
    c = Serial.read(); // read 1 character

    // check for a newline and exit the while-loop if appropriate
    if (c == '\n' || c == '\r')
      break; // break out of our infinite while-loop so that we can exit and restart the loop function

    // make alphabetic characters lowercase
    Serial.print(isalpha(c) ? tolower(c) : c);
  }

  Serial.println();
  // no need for any superloop delay since we wait for characters to
  // be received in the while loop above
}
```

24 November 2020

38

Basic serial input usage (non-blocking version)

40

```
void setup() {
  ...
  Serial.print ("\n\nType some characters and hit return...\nreceived:");
}

void loop() {
  char c;

  // process a single character if at least one is ready
  if (Serial.available()) {  // (change if to while to process all available characters)
    // OK, we now have at least 1 character (we might have more)
    c = Serial.read(); // read 1 character

    // check for a newline and exit the while-loop if appropriate
    if (c == '\n' || c == '\r') {
      Serial.print ("\n\n\nType some characters and hit return...\nreceived:");
    } else {
      Serial.print(isalpha(c) ? tolower(c) : c); // make alphabetic characters lowercase
    }
  }

  // TODO: other real time tasks could go here
  delay(SUPERLOOP_MS);
}
```

24 November 2020

39