Programming with C

Terry Marris November 2010

14 Arrays of Char

The programmer got his rise. Three cheers for the boss. Hip hip array.

In previous chapters we looked at the primitive data types, *int*, *double* and *char.* Now we begin looking at data structures. The first data structure we shall look into is arrays.

14.1 Arrays of Char

We have already met an array of *char*:

```
char string[10];
```

This declaration says that *string* is an array of *char* that can contain up to 10 characters including the end-of-string character, null or '\0'.

string

0	1	2	3	4	5	6	7	8	9
T	i	m		B	u	r	r	\0	

The size of the array is 10 (it has 10 cells or elements). The length of the string stored there, *Tim Burr*, is eight, that is the number of characters excluding the null character but including the space character between *Tim* and *Burr*.

The integers 0, 1, 2, .. 9 are known as *indices*. Indices address each element of the array. So

string[0] == 'T'
string[3] == ' '
string[7] == 'r'
string[8] == '\0'

string[9] has no defined value. And *string[10]* does not exist; there is no index with value 10.

The next program, shown below, finds the length of a string.

```
/* stringlength.c: finds the length of a string */

#include <stdio.h>
#include "utility.h"

int main()
{
  char string[10];
  int length = 0;
  int i;

  printf("Name? ");
  getString(string, 10);

  for (i = 0; string[i] != '\0'; i++)
    length++;

  printf("length of name entered is %d\n", length);
  return 0;
}
```

The specification of *getString()* is contained in *utility.h*, which we introduced in the chapter on Modules.

The *for* loop is the standard idiom for processing arrays.

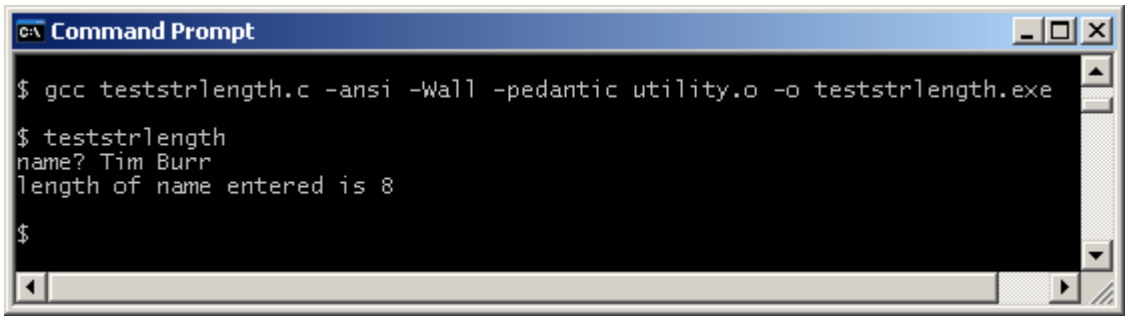
```
char string[10];
int length = 0;
int i;
...
for (i = 0; string[i] != '\0'; i++)
  length++;
```

Initially, *length* and *i* are both set to zero. We loop while the value of *string[i]* is not the null character. For each time round the loop we increment (add 1 to) *length*, and we increment *i*;

14.2 Arrays as Parameters

We really ought to write a *function* that returns the length of a string. This is done in the next program.

```
/* teststrlength.c: tests the stringLength function */

#include <stdio.h>
#include "utility.h"

/* stringLength: returns the length of the string */
int stringLength(char string[])
{
  int i;

  for (i = 0; string[i] != '\0'; i++)
    ;
  return i;
}

int main()
{
  char string[10];

  printf("name? ");
  getString(string, 10);
  printf("length of name entered is %d\n", stringLength(string));
  return 0;
}
```



In the function itself we have dispensed with the *length* variable because the array index variable *i* does the same job.

```
/* stringLength: returns the length of the string */
int stringLength(char string[])
{
  int i;

  for (i = 0; string[i] != '\0'; i++)
    ;
  return i;
}
```

The parameter, *char string[ ]*, says that the size of the array is determined elsewhere.

```
int main()
{
...
char string[10];
  ...
stringLength(string);
```

The size is defined in *main()*, and so, when the string is passed to the *stringLength()* function it size is already known.

What is actually passed is the *address* of the first element of the array. This is done automatically without any special intervention from us programmers. So, a pointer is passed in *string* to *stringLength()*.

```
int stringLength(char string[])
```

So we could write, for the function header,

```
int stringLength(char *string)
```

and this would have the same effect.

14.3 Arrays as Return Types

We might like to write array1 = array2, but this is not allowed in C. To make duplicate of an array we are obliged to copy it element by element. Our next function makes and returns a duplicate copy of its *string* parameter.

```
/* stringDuplicate: returns a duplicate copy of string */
char *stringDuplicate(const char *string)
{
  char *duplicate;
  int i;

  duplicate = malloc(strlen(string) + 1);  /* +1 for '\0' */
  for (i = 0; string[i] != '\0'; i++)
    duplicate[i] = string[i];
  duplicate[i] = '\0';
  return duplicate;
}
```

The function header

```
char *stringDuplicate(const char *string)
```

says *stringDuplicate()* returns a pointer to *char*, and, looking at the *return* statement, it is an array of *char* that is returned. The parameter is *(const char *string)*, *const* because we intend *string* to remain unchanged, *char** because *string* is an array of *char*.

We make *duplicate*, a pointer to *char*, large enough to hold the contents of *string*.

```
char *duplicate;
...
duplicate = malloc(strlen(string) + 1);  /* +1 for '\0' */
```

malloc(), for memory allocator, reserves the requested amount of space and assigns it to *duplicate*. The requested amount of space is the length of the *string*, plus one for the end-of-string null character. *strlen()* is defined in *string.h*. *malloc()* is defined in *stdlib.h*. We shall have more to say about *malloc()* when we come to consider structures.

Then we copy *string* into *duplicate*, element by element.

```
  for (i = 0; string[i] != '\0'; i++)
    duplicate[i] = string[i];
```

The assignment operator, **=**, works fine with the primitive data types such as *char*, and with pointers (and with structures - but we deal with that in a future chapter).

We remember to terminate *duplicate* with the null character, *'\0'* because many string handling functions rely on finding it.

```
duplicate[i] = '\0';
```

And, finally, we return *duplicate*.

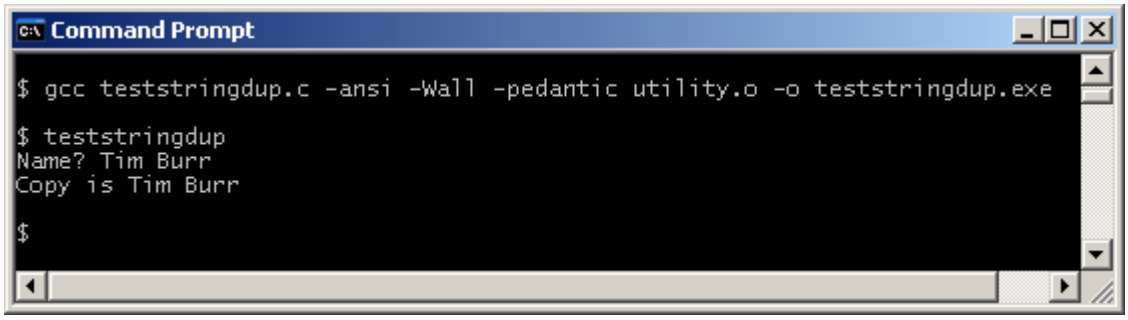
In *main()* we have

```
char *copy;
...
copy = stringDuplicate(string);
```

copy receives the value returned by *stringDuplicate*, the address of the first element in the array. We remember that *stringDuplicate* reserved memory space with *malloc()* for the array of lower case characters. *copy* points to the same space.

Shown below is the entire program and its run.

```
/* teststringdup.c: tests stringDuplicate() */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utility.h"

/* stringDuplicate: returns a duplicate copy of string */
char *stringDuplicate(const char *string)
{
  char *duplicate;
  int i;

  duplicate = malloc(strlen(string) + 1);  /* +1 for '\0' */
  for (i = 0; string[i] != '\0'; i++)
    duplicate[i] = string[i];
  duplicate[i] = '\0';
  return duplicate;
}

int main()
{
  enum { stringSize = 10 };
  char string[stringSize];
  char *copy;

  printf("Name? ");
  getString(string, stringSize);
  copy = stringDuplicate(string);
  printf("Copy is %s\n", copy);
  return 0;
}
```



14.4 Reverse an Array

We see how to reverse the contents of an array of char e.g. from this

string

0	1	2	3	4	5	6	7	8
T	i	m		B	u	r	r	\0

to this

str

0	1	2	3	4	5	6	7	8
r	r	u	B		m	i	T	\0

Index *i* addresses *string*, and index *j* addresses *str*, *string*'s reverse.

Initially, *i = 0* and *j = strlen(string) - 1* i.e. 7.

We copy *string[i]* into *str[j]*, then increment *i* and decrement *j*, until *string[i]* == *'\0'*.

```
for (i = 0; string[i] != '\0'; i++) {
  str[j] = string[i];
  j--;
}
```

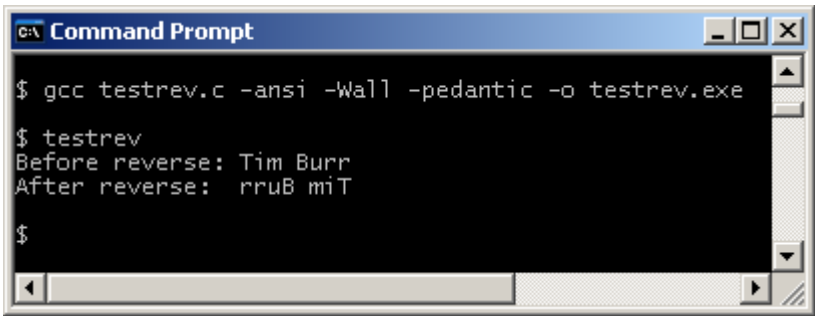
Finally, we copy the terminating null character into *str*.

Here is the entire function.

```
/* reverseString: reverses the contents of string */
char *reverseString(const char *string)
{
  char *str = malloc(strlen(string) + 1);
  int i, j;

  j = strlen(string) - 1;
  for (i = 0; string[i] != '\0'; i++) {
    str[j] = string[i];
    j--;
  }
  str[i] = '\0';
  return str;
}
```

And here is a complete program and its run.

```
/* testrev.c: tests a function that reverses string contents */

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

/* reverseString: reverses the contents of string */
char *reverseString(const char *string)
{
  char *str = malloc(strlen(string) + 1);
  int i, j;

  j = strlen(string) - 1;
  for (i = 0; string[i] != '\0'; i++) {
    str[j] = string[i];
    j--;
  }
  str[i] = '\0';
  return str;
}

int main()
{
  char string[] = "Tim Burr";
  char *rev;

  printf("Before reverse: %s\n", string);
  rev = reverseString(string);
  printf("After reverse:  %s\n", rev);
  return 0;
}
```



We use *reverseString()* in the function to convert an integer into an array of *char* (see §14.5 below). We include *reverseString()* in *utility.c*.

14.5 Integer to Array of char

We want to turn an integer such as 3456 into the array of char

string

0	1	2	3	4	5	6	7	8
3	4	5	6	\0				

Why? Well it may be useful when we come to consider files of text. But first we see how to converts a single digit integer into a *char*.

Part of the American Standard Code for Information Interchange (ASCII) is shown below.

Code (decimal)	*Digit char*
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9

We can see that the character digit '0' is represented by 48 (in decimal, i.e. base 10), '1' by 49, '2' by 50, ... We remember that *char*s are stored as their ASCII value and you cannot do sums with *chars*. But you can do sums with their integer equivalents.

Look at the digit char '5'. Its ASCII value is 53. Subtract from that the ASCII value for '0', and we get 53 - 48 = 5, an integer.

```
'5' - '0' = 53 - 48
          = 5
```

So, to convert a digit *char* into an integer we subtract '0' from the *char*

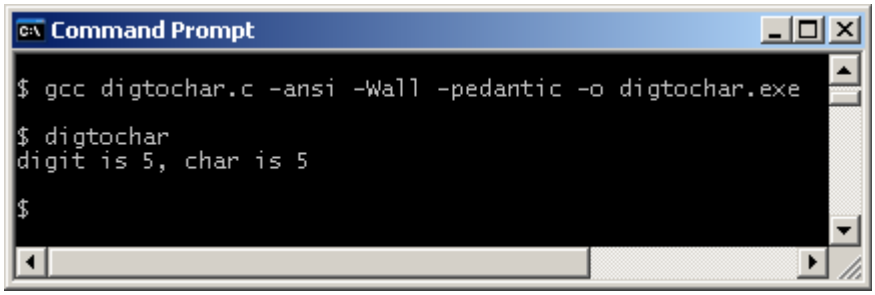
Conversely, to convert a single digit integer into a *char* we add '0' to the integer.

```
5 + '0'  = 5 + 48
         = 53
```

53 maps to '5'.

The program shown below illustrates the point.

```
/* digtochar.c: digit to char */
#include <stdio.h>

int main()
{
  int d = 5;
  char c;
  c = d + '0';
  printf("digit is %d, char is %c\n", d, c);
  return 0;
}
```



Now we see how to convert an integer such as 3456 into an array of *char*. We start with an example integer, and repeatedly use the mod % and div / operators to extract the last digit and remove it from the integer.

num = 3456
rem = num % 10 = 6
put 6 in the array

num = num / 10 = 345
rem = num % 10 = 5
put 5 in the array

num = num / 10 = 34
rem = num % 10 = 4
put 4 in the array

num = num / 10 = 3
rem = num % 10 = 3
put 3 in the array

num = num / 10 = 0
num == 0 so finish.

We end up with a string with the digits in the wrong order.

string

0	1	2	3	4	5	6	7	8
6	5	4	3	\0				

so we reverse them using *reverseString()*.

string

0	1	2	3	4	5	6	7	8
3	4	5	6	\0				

In the chapter on Functions we saw that the largest integer on my system is 2147483647, that is 10 digits altogether. Taking a possible minus sign and the null end-of-string character into account, we need an array with 12 elements to store an *int*.

```
/* intToString: converts an integer to a string */
char *intToString(int n)
{
  int i, sign;
  char *string = malloc(12);
```

If the number, *n*, is negative, we store its inverse after preserving the original number in *sign*.

```
  sign = n;
  if (n < 0)
    n = -n;
```

Then we repeatedly use the mod % and div / operators to strip out the least significant, right-most digit, convert it to a *char*, and store the *char* in the array.

```
  do {
    string[i] = n % 10 + '0';
    i++;
    n = n / 10;
} while (n > 0);
```

If the original number, n, was negative we record the minus sign in the array.

```
  if (sign < 0) {
    string[i] = '-';
    i++;
  }
```

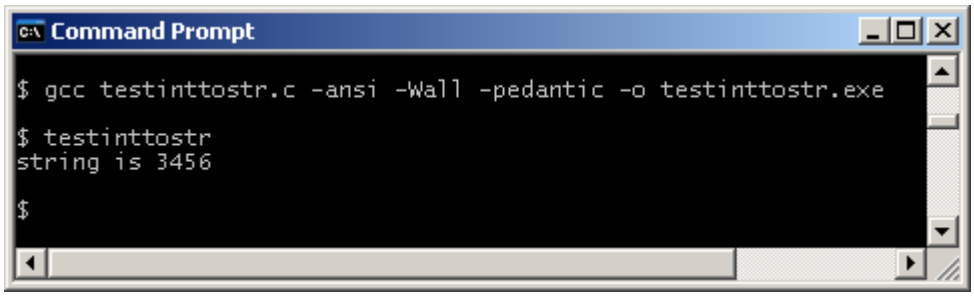
We append the null end-of-string character and, remembering that the digits were generated in the wrong order, we return the array with its contents reversed.

```
  string[i] = '\0';
  return reverseString(string);
}
```

Here is the entire program and an example run.

```
/* testinttostr.c: tests integer to string function */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utility.h"

/* reverseString: reverses the contents of string */
char *reverseString(const char *string)
{
  char *str = malloc(strlen(string) + 1);
  int i, j;

  j = strlen(string) - 1;
  for (i = 0; string[i] != '\0'; i++) {
    str[j] = string[i];
    j--;
  }
  str[i] = '\0';
  return str;
}
```

```
/* intToString: converts an integer to a string */
char *intToString(int n)
{
  int i, sign;
  char *string = malloc(12);

  sign = n;
  if (n < 0)
    n = -n;
  i = 0;
  do {
    string[i] = n % 10 + '0';
    i++;
    n = n / 10;
  } while (n > 0);

  if (sign < 0) {
    string[i] = '-';
    i++;
  }
  string[i] = '\0';
  return reverseString(string);
}

int main()
{
  char *string = intToString(3456);
  printf("string is %s\n", string);
  return 0;
}
```



We include both *reverseString()* and *intToString()* in *utility.c*.

Having done all that hard work, to further your understanding of arrays of char, there is a very simple way to convert an integer to a string. Use *sprintf()*.

sprintf() is defined in *stdio.h*. It constructs a string from its numeric arguments, each of which is converted by the familiar conversion specifications such as *%d* and *%f*.

sprintf(string, conversionSpecifications, numericArgumentList)

Here is an example program to show you what I mean.

```
/* intotstr.c: uses sprintf() to convert an int to a string */

#include <stdio.h>

int main()
{
  int integer = 7;
  char str[16];

  sprintf(str, "%d", integer);
  printf("The lucky number is %s\n", str);
  return 0;
}
```

14.6 Random Password Generator

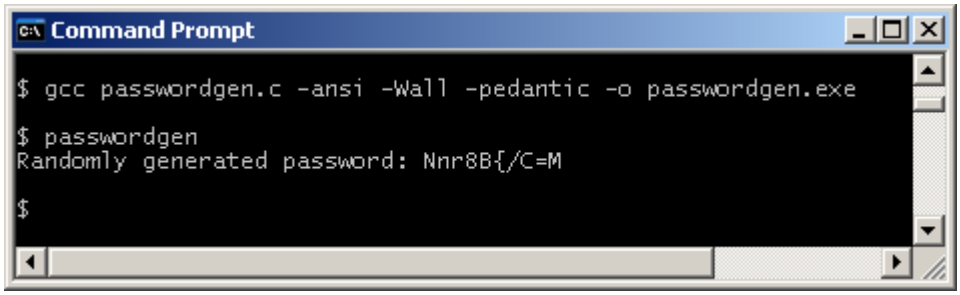
We develop a programp that creates 10-character randomly-generated password. First, we present the ASCII table for the common keyboard characters.

code	*char*	*code*	*char*	*code*	*char*	*code*	*char*	*code*	*char*
33	!	52	4	71	G	90	Z	109	m
34	"	53	5	72	H	91	[	110	n
35	#	54	6	73	I	92	\	111	o
36	$	55	7	74	J	93	]	112	p
37	%	56	8	75	K	94	^	113	q
38	&	57	9	76	L	95	_	114	r
39	'	58	:	77	M	96	`	115	s
40	(	59	;	78	N	97	a	116	t
41	)	60	<	79	O	98	b	117	u
42	*	61	=	80	P	99	c	118	v
43	+	62	>	81	Q	100	d	119	w
44	,	63	?	82	R	101	e	120	x
45	-	64	@	83	S	102	f	121	y
46	.	65	A	84	T	103	g	122	z
47	/	66	B	85	U	104	h	123	{
48	0	67	C	86	V	105	i	124	|
49	1	68	D	87	W	106	j	125	}
50	2	69	E	88	X	107	k	126	~
51	3	70	F	89	Y	108	l		

code is the integer equivalent of the keyboard character, *char*.
We see that the ASCII codes range from 33 up to 126 inclusive. 126 - 33 = 93. If we generate a randon number in the range 0..93, then add 33 to it, we get a random letter. Check. If random number = 0, ASCII code is 0 + 33 = 33 i.e. !. If random number = 93, ASCII code is 93 + 33 = 126 i.e. ~.

```
/* passwordgen.c: generates a 10 character random password */

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

```
int main()
{
  int number, seed, i;
  char password[12];

  printf("Randomly generated password: ");
  seed = time(NULL);
  srand(seed);

  for (i = 0; i < 10; i++) {
    number = rand() % 93 + 33;
    password[i] = number;
  }
  password[i] = '\0';
  printf("%s\n", password);
  return 0;
}
```



Exercise 14.1

1. Write and test a function that will convert an entire array of *char* to lower case. Include the function in *utility.c*.

2. Write and test a function that will convert an entire array of *char* to upper case. Include the function in *utility.c*.

3. Write and test a version of *readString()*, named *readStr()*, that has just one integer parameter, size, and returns the string read from the keyboard. Include the function in *utility.c*.

4. Two arrays of char are equal if they have precisely the same elements. Write and test a function that returns 1 (true) if its two array of *char* arguments are identical, 0 (false) otherwise. Include the function in *utility.c*.

5. Write and test a function that compares two arrays of *char* for equality when case is ignored. Include the function in *utility.c*.

6. Write and test a function that compares two arrays of char when case is ignored. The function should return -1 if the first array comes before the second in alphabetical or lexical order, 0 if they are identical, and +1 if the first array comes after the second in lexical order. Include the function in *utility.c*.

We have taken a first look at arrays, in particular, arrays of *char*. **Next** we continue looking at arrays, in particular, arrays of numbers.

Bibliography

Kernighan B and Ritchie D *The C Programming Language* Prentice Hall 1988 pp 64, 143
Mark Williams Company *ANSI C A Lexical Guide* Prentice Hall 1988