C Supplementary

Terry Marris September 2012

Memory Management

Input from the keyboard has always been an interesting problem. With *scanf()* you need to manage any characters left in the keyboard buffer. With *gets()* your program crashes if the input is longer than expected. We see how to fix these problems by writing our own string input functions, and, in doing so, see how and when *malloc()* and *free()* are used.

malloc() allocates memory. *free()* releases memory previously allocated with *malloc()*.

String Input

getString()

We start with a fundamental function to get a string input from the keyboard.

```
/* getString: reads up to maxLength-1 characters plus '\0'
   into string from keyboard */
int getString(char string[], int maxLength)
{
  char c;
  int i = 0;

  while ((c = getchar()) != '\n') {
    if (i < maxLength - 1) {
      string[i] = c;
      i++;
    }
  }
  string[i] = '\0';
  return i;
}
```

getString() has two parameters. *string[]* is an array of char. It is an incomplete type because its size, the number of elements it contains, is defined elsewhere e.g. in *main()*. *maxLength* is the maximum length of the string, including the null character, '\0'. So, if *maxLength = 4* then *string* could contain *1 2 3 /0*. We assume *string[]* is large enough. You might call *getString()* like this:

```
char name[10];
getString(name, 10);
```

getString() has two variables. *char c* stores a single character at a time. *int i* indexes the array of *char* named *string*.

The loop header

```
while ((c = getchar()) != '\n') {
```

says: get the next character from the keyboard and assign it to the variable *c*. If it is not the newline character, *'\n'*, then loop. If it is the newline character then stop looping.

The loop is

```
while ((c = getchar()) != '\n') {
  if (i < maxLength - 1) {
    string[i] = c;
    i++;
  }
}
```

Providing index *i* is less than *maxLength-1*, store the character in the *string* array at location *i*, then add 1 to *i*. If *i* is not less than *maxLength-1*, do nothing.

The loop ensures that any characters left in the keyboard buffer after *maxLength-1* characters have been read and stored, are themselves read and ignored

When the loop terminates, the null character, *'\0'*, is assigned to the array and the number of characters in the array, *i*, is returned.

```
string[i] = '\0';
return i;
```

Exercise 1

1 Write a program to test the *getString()* function shown above.

readString()

The next function, *readString()*, displays a prompt and reads up to 255 characters, possibly more, from the keyboard, and returns the string read.

```
/* readString: displays prompt, reads up to 255 characters,
   possibly more, plus '\0', from the keyboard and returns it
*/
char *readString(char *prompt)
{
  char string[BUFSIZ];
  char *str;

  printf("%s ", prompt);
  getString(string, BUFSIZ);

  str = (char *)malloc(strlen(string) + 1);
  if (str == NULL)
    error("readString: out of memory");
  strcpy(str, string);
  return str;
}
```

To use this function you could write

```
char *name = readString("Please enter your name: ");
```

We look at the function line-by-line. The variables are

```
char string[BUFSIZ];
char *str;
```

string[BUFSIZ] is an array of *char*. *BUFSIZ* is at least 256 and is defined in *stdio.h*. Notice there is no *E* in *BUFSIZ*. *str* is a pointer to *char*. It could contain the address of the first character in a string. At this point we do not know how big the string is, we do not know how many characters it contains.

The next two lines

```
printf("%s ", prompt);
getString(string, BUFSIZ);
```

display the prompt and make a call to *getString()*. The function *getString()*, discussed above, receives an array of *char* to store the string input, and the maximum size of the string.

The next line makes a call to *malloc()*.

```
str = (char *)malloc(strlen(string) + 1);
```

malloc() is defined in *stdlib.h* *malloc()* allocates a block of memory and returns a pointer to the block of memory it has allocated. How big is the block? Well, in this case it is the length of the string previously retrieved by *getString()* plus 1. Plus 1 is for the end-of-string character, '\0'.

Technically, *malloc()* returns a *pointer to void*. This means it can be used for any kind of object. We use the cast operator, *(char *)*, to convert the pointer returned to be of type pointer to *char* because *str* is defined to be of type pointer to *char*.

Now, if *malloc()* cannot find a block of memory of the requested size, it returns the null pointer, *NULL*. *NULL* is defined in *stddef.h*. If *malloc()* returns *NULL* we have an error situation. So we write

```
if (str == NULL)
  error("readString: out of memory");
```

There is no returning from the *error()* function.

```
/* error: prints error message and terminates program
   execution */
void error(char* message)
{
  printf("Fatal error from %s\n", message);
  exit(EXIT_FAILURE);
}
```

exit() and *EXIT_FAILURE* are both defined in *stddef.h.* *exit()* immediately stops the program from running and returns control to the program's environment.

Anyway, back to the address of a block of memory stored in *str*. The next line of the function copies the string obtained by *getString()* into the block of memory referenced by *str*.

```
strcpy(str, string);
```

Finally, the function returns the address of the block of memory just filled by *strcpy()*.

```
return str;
```

reclaimMemory()

malloc() puts you, the programmer, in charge of allocating memory to variables. And, being responsible, you release the memory when no longer required so that it may be re-used.

```
/* reclaimMemory: reclaims memory previously allocated with
   malloc() */
void *reclaimMemory(void *ptr)
{
  free(ptr);
  ptr = NULL;
  return ptr;
}
```

To use this function you could write

```
char *name = readString("Please enter your name: ");
......
reclaimMemory(name);
```

You may notice that the function parameter is *void **, and we are passing a *char ** to it. That is not a problem. Any pointer can be cast to a *void ** and back again.

The two most important lines in *reclaimMemory()* are

```
  free(ptr);
  ptr = NULL;
```

free() de-allocates a block of memory previously allocated by *malloc()*. *ptr* contains the address of that block. After calling *free()* you should ensure that the pointer contains the address of nothing important because (a) *free()* does not do that and (b) if your program tries to access the memory that has been freed, there is no guarantee that the program will behave correctly. Further, the pointer passed to *free()* must contain the address of a block of memory previously allocated by *malloc()* (or by *calloc()* or by *realloc()*).

So what have we got? We have a pair of functions, *readString()* and *getString()*, that together reads a string from the keyboard of any length up to a maximum defined by *BUFSIZ*. Neat or what? All you need to do is to include these functions in every program that reads input from the keyboard.

Here is the entire program.

```
/* readstring.c - implements a variable length string */

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

/* error: prints error message and terminates program
   execution */
void error(char* message)
{
  printf("Fatal error from %s\n", message);
  exit(EXIT_FAILURE);
}


/* reclaimMemory: reclaims memory previously allocated with
   malloc() */
void *reclaimMemory(void *ptr)
{
  free(ptr);
  ptr = NULL;
  return ptr;
}


/* getString: reads up to maxLength-1 characters plus '\0'
   into string from keyboard */
int getString(char string[], int maxLength)
{
  char c;
  int i = 0;

  while ((c = getchar()) != '\n') {
    if (i < maxLength - 1) {
      string[i] = c;
      i++;
    }
  }
  string[i] = '\0';
  return i;
}
```

```
/* readString: displays prompt, reads up to 255 characters,
   possibly more, plus '\0', from the keyboard and returns it
*/
char *readString(char *prompt)
{
  char string[BUFSIZ];
  char *str;

  printf("%s ", prompt);
  getString(string, BUFSIZ);

  str = (char *)malloc(strlen(string) + 1);
  if (str == NULL)
    error("readString: out of memory");
  strcpy(str, string);
  return str;
}


int main()
{
  char *name;
  char *bike;

  name = readString("Rider?");
  bike = readString("Bike?");
  printf("%s rides %s\n", name, bike);
  reclaimMemory(name);
  printf("After reclaiming memory rider is %s\n", name);

  return 0;
}
```

Program run

When to use malloc() and for what

You would use *malloc()* when you cannot predict precisely what the user wants to do. Who is the user? Probably not a programmer! For example, the user might want to enter a very large string (e.g. San Benedetto del Tronto) or a very small string (e.g. Rome). You could use a large array of *char* which you declare at *compile time*, to contain user input. But that could waste a lot of space, or it could be not large enough if the user wanted to enter something even larger. So, you let the user decide how big the string is *at run time*. You would use *malloc()* to reserve memory for variables when required when the program is running.

Exercise 2

1 Design, write and test a function, *char *duplicateString(const char *str)*, that returns a duplicate copy of its string parameter.

Answers

Exercise 2

1

```
/* error: displays error message, halts program execution */
void error(char *message)
{
  printf("Fatal error from %s: ", message);
  exit(EXIT_FAILURE);
}


/* duplicateString: returns a string duplicate of parameter
   str */
char *duplicateString(const char *str)
{
  char *dupStr;

  dupStr = (char *)malloc(sizeof(str) + 1);
  if (dupStr == NULL)
    error("duplicateString: out of memory");
  strcpy(dupStr, str);
  return dupStr;
}
```