Strings in C

Strings

A string is an Null-Terminated Character Array

Allocate space for a string just like any other array:

```
char outputString[16];
```

Space for string must contain room for terminating zero

Special syntax for initializing a string:

```
char outputString[] = "Result = ";
```

…which is the same as:

```
outputString[0] = 'R';
outputString[1] = 'e';
outputString[2] = 's';
...
outputString[9] = '\0'; // Null terminator
```

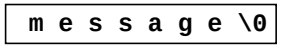

String Declaration

What's the difference between:

- **char amessage[ ] = "message"**
- **char *pmessage = "message"**

Answer:

- **char amessage[ ] = "message" // single array**



- **char *pmessage = "message" // pointer and array**

String Declaration (contd ..)

- Variable *amessage*
  - Cannot be reassigned to point to another string
  - The array of characters are located on the stack and can be mutated i.e. you can change the string contents
- Variable *pmessage*
  - It is pointer that points to first location of the string
  - Pointer can be reassigned to point to another string
  - Cannot change the string contents (immutable)
    - The string is placed in a "read only" section

I/O with Strings

char * outputString = “Hello”;

char inputString [100];

Printf and scanf use "%s" format character for string

Printf -- print characters up to terminating zero

```
printf("%s", outputString);
```

Scanf -- read characters until whitespace,
store result in string, and terminate with zero

```
scanf("%s", inputString);
```

Why no & operator?

atoi function in <stdlib.h>

- **int atoi (const char * str)**
  - Interprets the contents of string as an integral number, which is returned as an int value
  - First discards as many whitespace characters as necessary until the first non-whitespace character is found
    - Takes an optional initial *plus* or *minus* sign followed by as many numerical digits as possible, and interprets them as a numerical value
  - The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.
  - If *str* is not a valid integral number, or if no such sequence exists because either *str* is empty or it contains only whitespace characters, no conversion is performed and 0 is returned.

Strings

Although there is no string data type in C, C has library <string.h> that can perform actions on strings.

All the functions in <string.h> have parameters or return values as

- **character arrays terminated with null character**
- **const char * i.e. declare a pointer to a const (string constant)**

E.g. of String function

- **strlen – returns string length**
- **strcpy – copy one string to another location**

String Length - Array Style

```
int strlen(char str[])
{
  int i = 0;
  while (str[i] != '\0') {
    i++;
  }
  return i;
}
```

String Length - Pointer Style

```
int strlen(const char* str)
{
  int i = 0;
  while (*str != '\0') {
    i++;
    str++;
  }
  return i;
}
```

Note: array and pointer declarations interchangeable as function formal parameters because the whole array is never actually passed to a function (the address of the array is)

Usage of strlen

Output:

H
e
l
l
o

```
#include <string.h>
#include <stdio.h>
int main(){
 char array[] = "Hello";
  for(i = 0; i <  strlen(array);i++){
    printf("%c\n",array[i]);
 }
}
```


Side Note on *const* keyword

- const int x; // constant int
  x = 2; // illegal - can't modify x
- const int* pX; // changeable pointer to constant int
  *pX = 3; // illegal - can't use pX to modify an int
  pX = &someOtherIntVar; // legal - pX can point somewhere else
- int* const pY; // constant pointer to changeable int
  *pY = 4; // legal - can use pY to modify an int
  pY = &someOtherIntVar; // illegal - can't make pY point anywhere else
- const int* const pZ; // const pointer to const int
  *pZ = 5; // illegal - can't use pZ to modify an int
  pZ = &someOtherIntVar; // illegal - can't make pZ point anywhere else

CONFUSING!!!

String Copy - Array Style

```
void strcpy(char dest[], char src[])
{
  int i = 0;
  while (src[i] != '\0') {
    dest[i] = src[i];
    i++;
  }
  dest[i] = '\0'
}
```

String Copy - Array Style #2

```
void strcpy(char dest[], char src[])
{
  int i = 0;
  while ((dest[i] = src[i]) != '\0') {
    i++;
  }
}
```


String Copy - Pointer Style

```
void strcpy(char* dest, const char* src)
{
  while ((*dest = *src) != '\0') {
    dest++;
    src++;
  }
}
```


String Copy - Pointer Style #2

```
void strcpy(char* dest, const char* src)
{
  while ((*dest++ = *src++) != '\0') {
    // nothing
  }
}
```

Difficult to read

- "Experienced C programmers would prefer…" - K&R
- However confusing: try avoid this type of code

What happens if `dest` is too small?

- Bad things…

C String Library

C has a limited string library

- All based on null-terminated strings
- #include <string.h> to use them

Functions include

1. `int strlen(const char* str)`
2. `void strcpy(char* dest, const char* src)`
3. `int strcmp(const char* s1, const char* s2)`
   - Returns 0 on equal, -1 or 1 if greater or less
     - If not equal, then sign is based on difference between bytes in the location the strings differ in.
   - Remember, 0 is false, so equal returns false!

More String Library Functions

1. `strcat(char* dest, const char* src)`
   - string concatenation (appending two strings)
   - No plus (+) operator for string concatenation
2. `strncpy(char* dest, const char* src, size_t n)`
3. `strncmp(const char* s1, const char* s2, size_t n)`
4. `strncat(char* dest, const char* src, size_t n)`

Note: Use strn* (instead of str*) as it provides length.

Plus many more…

When manipulating text in C

Use char array to hold strings

Extremely important:

- **Array size must be max string length + 1**
- **why??**

Use the string library (string.h) when possible instead of creating your own functions

Scanf return value

The scanf function returns an integer, which indicates the number of successful conversions performed.

- **This lets the programer check whether the input stream was in the proper format.**
- **This also includes any literals in format string i.e. must match literals in the conversion process**

Example: (in scanf1.c)

```
scanf("%d /%d /%d", &bMonth, &bDay, &bYear);
```

Input Stream	*Return Value*
02/16/69	3
02/16 69	2
02 16 69	1

Scanf Bad Input

Remember that characters are added to a buffer (temporary storage) and given to input stream (keyboard) only when the "Enter" key is pressed (buffered streaming)

//Example in scanf2.c

```
#include <stdio.h>

int main(){
  int check = 0;
  int i = 0;
  while(i != 1){
    printf("Enter number\n");
    if((check = scanf("%d",&i)) != 1){
      printf("Error in input, must be a number");
    }
  }
}
```

What happens when you enter letter or float response to the prompt for an integer?

- **Stuck in a while loop, why??**

Scanf Bad Input (contd..)

Why?

1	2	.	4	5	'\n'

The picture shows the stream of input characters after the first call to scanf was complete. Here is what the first call did:

- **Read the '1', saw that it was a digit and can be used as part of an int**
- **Read the '2', saw that it was a digit and can be used as part of an int**
- **Read the '.', saw that it was not a digit and could be not be used as part of an int. The '.' was put back on the input stream (in our e.g. stdin) so that it could be read by the next input operation**
- **So how do we take care of this ?**

Scanf bad input (contd..)

Need to clear/disable buffering with the input stream (in our example stdin).

void setbuf (FILE * *stream* , char * *buffer*);

- **Causes the character array pointed to by the *buffer* parameter to be used instead of an automatically allocated buffer.**
- **If the specified *buffer* is NULL it disables buffering with the *stream***
  - **E.g. setbuf(stdin, NULL)**
- **Important: Use *setbuf()* function after a stream has been opened but before it is read or written.**