Introduction To C Programming

Lesson 08

Characters and Strings

character functions

- int getchar()
  - reads a character from stdin
  - beware
    - char c;
    - c = getchar() /* disaster waiting to happen */
- int putchar(int c)
  - writes a character to stdout

ctype.h

- int isdigit(int c);
- int isalpha(int c);
- int isalnum(int c);
- int isxdigit(int c); /* is c a hex digit */
- int islower(int c);
- int isupper(int c);
- int tolower(int c);
- int toupper(int c);

ctype .h

- int isspace(int c);
- int iscntrl(int c);
- int ispunct(int c);
- int isprint(int c);
- int isgraph(int c);

Strings

- Recall that in C a string is a null terminated array of char
- A string literal is any sequence of characters enclosed in double quotes

  “Hello”

- It is stored in memory in contiguous byte locations with a terminating ‘\0’ at the end

Strings

- A literal string is an *expression* in C
- A string has a value and it may be evaluated
- The value that a string resolves to is the address of where it is stored in memory
- Therefore, a literal string is a symbolic representation of an address; it is a pointer and it's type is **`char *`** which can also be expressed as **`char[]`**

String sizes

- Logical Length
  - How many characters currently in the string
- Physical Length
  - Max characters the string may ever contain
- Physical length must be at least one greater than logical length to allow for the terminating null character

```
Char str[256] = “Hello”;
// str logical length == 5
// str physical length == 256
// str could contain a string no greater than 255
// characters
```

Example

```
#include <stdio.h>
int main(void){
    int   i;

    for (i = 0; i < 5; ++i)
        putchar(“Hello”[i]);
    putchar(‘\n’);
    return 0;
}
```

Output

```
**************************************************
Output:
Hello
```

Strings

- We’ve seen how you can initialize a char array:
  `char  p1[] = “Hello one”;`
- Consider this declaration:
  `char  *p2 = “Hello two”;`
- A string literal is a pointer (to char), so it can be assigned to another pointer; p2 now holds the address where “Hello two” is stored
- Again, p1 is a pointer *constant*, and p2 is a pointer *variable*

Strings

- The only real difference between p1 and p2 is that p1 cannot have its value changed, and p2 can

`*(p1 + 1)` valid `*p1++` invalid

`*(p2 + 1)` valid `*p2++` valid

- A more correct definition of a string is:

  a string is a pointer to a char

Strings

- For this prototype: **`int  hypo(char *)`**
- An experienced programmer knows that a string literal is a valid argument
- With these declarations:

```
char  mess1[] = “This is mess1”;
char  *mess2 = “This is mess2”;
```

- These calls are all valid:

```
hypo(“This is a mess”); hypo(mess1);
hypo(mess2);
```

Arrays of Strings

- It is possible to have an array of strings:

```
char* msg[]={“File not found”,
             “Write protected”,
             “Read error”};
msg[0] points to “File not found”
msg[1] points to “Write protected”
msg[2] points to “Read error”
printf(“error: %s\n”, msg[1]);
```

Strings

- Inputting a string requires 2 steps:
  1. Allocate enough space in memory to store the string
  2. Use an input function

- No one forgets to use the input function, but it’s common to forget to allocate space

Strings

- Examples:

```
char  fname[80]; /* space allocated */
scanf("%s", fname); /* input func */
```

- The biggest problem comes when using pointers

```
char  *fname; /* no space allocated */
scanf("%s", fname);
```

Strings

- Even though it is declared, **`fname`** is not pointing to usable space
- When scanf() executes, it could place the the string anywhere in memory, including possibly overwriting the program code or data, which is catastrophic
- You must allocate space at compile time by using a declaration such as char fname[40];

gets()

- gets() is a string input function. The prototype is:

```
char * gets(char *)
```

- gets() reads characters from stdin and puts them in the memory area pointed to, until it encounters a newline or an EOF
- Replaces newline with terminating ‘\0’
- If successful it returns a pointer to where it stored the chars, and a NULL otherwise

gets()

- Example:

```
char  name[80];

printf("Please enter your name: ");
gets(name);
printf("Hello %s, how are you?", name);

/* what happens if the user enters a string
** longer than 80 chars? */
```

puts()

- puts() is used to print strings to stdout.

```
int puts(const char *);
```

- puts() writes the string pointed to by its argument to stdout. When it reaches the null terminator is discards it and writes out a newline character.
- If successful, it returns the number of characters printed. If unsuccessful, it returns and EOF.

```
#include <stdio.h>

int main(void)
{
    char   *greeting = “Hello all!”;

    puts(greeting);
    return 0;
}
```

String Functions

`unsigned int strlen(const char*)`

- Returns the number of characters in a string

```
printf("The length of the string is %d",
       strlen("Hello"));
```

```
Output:
The length of the string is 5
```

String Functions

```
char* strcat(char*, const char*);
```

- Copies the string in the second argument to the end of the string in the first argument. Returns a pointer to the first string

```
char  header[255] = “This is a ”;
char  trailer[] = “test”;
puts(header);
strcat(header, trailer);
puts(header);
```

String Functions

```
int strcmp(const char *, const char *);
```

- Tests the equality of two strings. It returns 0 if they are equivalent, a negative value if the first string is less than the second one, and a positive value if the first one is greater than the second one.
- To understand the need for this function, consider the example on the next slide:

Example

```
#include <stdio.h>
int main(void) {
    char answer[80];
    puts(“Please enter yes or no”);
    gets(answer);
    if (answer == “yes”)
        do_something();
```

Example II

```
else if (answer == “no”)
    do_something_else();
else
    puts(“You didn’t answer”);
    return 0;
}
```

String Functions

- The program appears to be reasonable but in the world of C it is not. Comparing two strings in this manner doesn't work, because you are comparing *addresses*. Even a statement like this is likely to fail:

```
if (“Test” == “Test”)
```

- The compiler is not required to have both of these literal strings in the same place in memory

Example I

```
#include <stdio.h>
int main(void) {
    char answer[80];
    puts(“Please enter yes or no”);
    gets(answer);
    if (strcmp(answer, “yes”) == 0)
        do_something();
```

Example II

```
    else if (strcmp(answer, “no”) == 0)
        do_something_else();
    else
        puts(“You didn’t answer”);
    return 0;
}
```

String Functions

`char* strcpy(char*, const char*)`

- Copies the contents of the second string into the memory location of the first string.

```
char  buffer[80];
char  name[255];
puts(“Please enter your name”);
gets(buffer);
strcpy(name, buffer);
printf(“Hello %s\n”, name);
```

String Functions

```
int atoi(const char *);
```

- Converts the string of digits into its numerical value

atoi(“2356”) returns 2356

atoi(“34abc”) returns 34

atoi(“xy45”) returns an undefined value

More string functions

- int sprintf(char* s, const char* format);
- int sscanf(char* s, const char* format);