Lecture 11 examples

January 7, 2019

1 Strings, Structures and function pointers

1.1 Strings

In C, string is an array of characters, terminated by a speciall character, the NULL. Such strings are refered to as Null-terminated strings.

Let us recall the things we know about characters * Occupies 1 byte * Each caracter has an ASCI code * Some are whitespace - we do not see them * characters use ' ' not " "

```
In [ ]: #include <stdio.h>

        int main()
        {
            char c;
            printf("The size of char is %ld\n", sizeof(char));
            //1 byte stores ints from 0 to 255 (256 characters)
            for(int i=50; i<55; ++i) // Set the range to what you want
            {
                printf("%d %c\n", i, i);
            }
        }
```

Let us see where is the Null character '\0'

```
In [ ]: #include <stdio.h>

        int main()
        {
         //there is a special character! '\0'
          for(int i=0; i<256; ++i)
          {
            if(i=='\0')
              printf("%d %c\n", i, i); // the zero char
          }
        }
```

The speciall Null character is '\0' and is the first in the ASCII table. This character is important to us. It will be used to mark termination of a string. It is used for: * Determining the length of a string * Copying one string to another * Appending (concatenating) one string to another * Any other operation on strings

We will now write a program storing multiple characters inside an array of characters - A word or a sentence. The last entry is

In [ ]:
```
//Write a program storing multiple characters inside an array of characters - A word o
#include <stdio.h>

int main(){
  char word[256]; //an array of characters, let us see what is inside

  word[0] = 'H';
  word[1] = 'e';
  word[2] = 'l';
  word[3] = 'l';
  word[4] = 'o';
  word[5] = ' ';
  word[6] = 'W';
  word[7] = 'o';
  word[8] = 'r';
  word[9] = 'l';
  word[10] = 'd';
  word[11] = '\0';

  for(int i=0; i<15; ++i) // Print character by character
  {
    printf("%c", word[i]);
  }
  printf("\n");

  printf("%s\n", word); // Print as a string

  for(int i=11; i<256; ++i) // Manipulate the string
  {
    word[i]=55; // this is 7!
  }
  word[20] = '\0'; // Add a Null character
  printf("%s\n", word);//And print as a string
}
```

How many '7' was printed and why?
Let now have another example:

In [ ]:
```
#include <stdio.h>
#include <stdlib.h>

int main(){
  char a = 'a'; //this is a single char
```

```
  char b = '\0'; //so is this

  printf("%c %c --\n", a, b);

  char tc[20];
  tc[0] = 'a';
  for(int i=0; i<20; ++i)
  {
    tc[i] = rand()%10+50;
    printf("%c ", tc[i]);
  }
  tc[19] = '\0';
  printf("\n%s\n", tc);

  tc[3] = b; // '\0' is the termination character
  printf("%s\n", tc);
}
```

1.1.1 Initialization of strings

Initialize strings in a more convinient way: * As a static array, entries of which we can modify, but which can not be reasigned * As a pointer pointing at a static array, which we can not modify, but we can reasign the pointer to a different address

```
In [ ]: #include <stdio.h>
        #include <stdlib.h>

        int main(){
          // A static array a[]
          char a[] = "The cat is black!";// {'a','s'}
          printf("%s\n", a);

          a[1] = 33;
          printf("%s\n", a);
          //a = "aaa"; // We can not do this!

          // As a pointer pointing at a fixed, static array
          char *p = "This cat is white!";
          printf("%s\n", p);
          printf("%p\n", p);

          //We can not modify elelents sicne array is fixed at compilation
          //p[1] = 33; //can not modify the value like that
          // We can reassign the address
          p = a;
          printf("%s\n", p);
          printf("%p\n", p);
```

```
  p="aaaaa";
  printf("%s\n", p);
  printf("%p\n", p);
}
```

Note that each time address to which *p* pointed changed! The main message here is that manipulating strings is smoewhat difficult. Do not worry, we will deal with the subject by learning string copping function!

1.1.2 Reading strings:

first with scanf(), but only up to a first whitespace character

In [ ]:
```
#include <stdio.h>

int main(){
  char a[256];

  scanf("%s", a);

  printf("%s\n", a);
}
```

An alternative is tu use **fgets** function:
char *fgets (char *str, int size, FILE* file);
the function reads a string of data from *FILE* input, of size *size* and stores it in a buffer *str*.
The source freom which we read is more general than a simple file *FILE*, it can be a standard input (stdin - the keybord).

In [ ]:
```
#include <stdio.h>

int main(){
  char a[10];

  fgets( a, 10, stdin ); // Read from keyboard

  printf("%s\n", a);
  printf("%c %p\n", a[0], a);
}
```

As above, but the size of a buffer is determined on runtime, and dynamic allocation is used.

In [ ]:
```
#include <stdio.h>
#include <stdlib.h>

int main(){
  int n;
  scanf("%d\n", &n);
  char *p=(char*)malloc(n*sizeof(char));
```

```
  fgets( p, n, stdin );

  printf("%s\n", p);
  printf("%c %p\n", p[0], p);

  free(p);
}
```

Finally fgets, makes it easy to read from a file. Here we read a C source file and print the content:

```
In [ ]: #include <stdio.h>
        #include <stdlib.h>

        int main(){
          FILE *f=fopen("read4.c", "r");

          char line[1000];
          for(int i=0; i<15; ++i) //How to see if file has ended?
          {
            fgets( line, 1000, f );
            printf("%s", line);
          }
          fclose(f);
        }
```

1.1.3 String manipulation

There is a siute of functions designed for operations on strings, in order to use those we need to include a new header: **string.h**, it gives access to the follwing functions: * Comparison: int strcmp (const char *s1, const char* s2); Returns 0 if s1 and s2 are the same; less than 0 if s1s2. * String concatenate: char *strcat (char* dest, const char *src);* Copy: char *strcpy (char* dest, const char *src);* Length of a string: int strlen (const char *s);* char* strchr(s1, ch); Returns a pointer to the first occurrence of character ch in string s1. * char* strstr(s1, s2); Returns a pointer to the first occurrence of string s2 in string s1.

Let us start with "our" implementation of the string compare function, try to analyze how it works:

```
In [ ]: #include <stdio.h>

        int mystrcmp(char *s1, char *s2)
        {
          for(int i=0; 1; ++i)
          {
            if(s1[i] - s2[i] != 0)
              return s1[i] - s2[i];
            if(s1[i] == '\0' || s2[i] == '\0' ) break;
          }
```

```
  return 0;
}

int main(){
  char a[] = "111";
  char key[]="aaa";
  int res = mystrcmp( a, key );
  printf("%d\n", res);
}
```

- String comparison with **strcmp()**

In [ ]:
```
#include <stdio.h>
#include <string.h>

int main(){
  char a[] = "ABCb";//65 66 67
  char b[] = "ABCa";//97 98 99

  int res = strcmp(b, a);
  printf("res is: %d\n", res);
}
```

- String concatenate with **strcat()**

In [ ]:
```
#include <stdio.h>
#include <string.h>

int main(){
  char a[10] = "ABC";//65 66 67
  char b[10] = "abc";//97 98 99

  printf("%s \n", a);
  printf("%s \n", b);
  strcat(b, a);       //make sure the size of a is enough!
  printf("%s \n", a);
  printf("%s \n", b);
}
```

- String copy with **strcpy()** - mind that we had trouble manipulating strings, this fuction will be helpfull to us.

In [ ]:
```
#include <stdio.h>
#include <string.h>

int main(){
  char a[] = "AsssBC";//65 66 67
  char b[] = "abffffdsadsadc";//97 98 99
```

```
    printf("String a: %s \n", a);
    printf("String b: %s \n", b);

    strcpy( b, a ); // Copy a to b
    printf("String a: %s \n", a);
    printf("String b: %s \n", b);
  }
```

- String length with **strlen()**

In [ ]:
```
#include <stdio.h>
#include <string.h>

int main(){
  char a[] = "abc";
  char b[] = "ABCDEF";

  printf("%s \n", a);
  printf("%s \n", b);

  int l1=strlen(a);
  int l2=strlen(b);

  printf("Length of a: %d, length of b: %d\n", l1, l2);
}
```

- Find character in a string with **strchr(s1, ch)** - Returns a pointer to the first occurrence of character ch in string s1.

In [ ]:
```
#include <stdio.h>
#include <string.h>

int main(){
  char a[] = "abcde";

  char *p = strchr(a, 'c');

  printf("%p\n", p);
  if(p == NULL){
      printf("Not found\n");
  }
  else{
      printf("%c %p %ld\n", *p, p, p-a);
      printf("%s\n", p);//!!
  }
}
```

- Find a string in a string with **strstr(s1, s2)** - Returns a pointer to the first occurrence of string s2 in string s1.

In [ ]:
```
#include <stdio.h>
#include <string.h>

int main(){
  char a[] = "Passing data to and from functions with pointers";
  char key[]="data";
  int l=strlen(key);

  char *p = strstr(a, key);
  if(*p!='\0')
    *(p+l) = '\0';

  printf("%c %p %ld\n", *p, p, p-a);
  printf("%s\n", p);
}
```

1.2 Structures

Are used to group data and allow for better code organization. Up to now whe have been using **simple** or **primitive** data types. I.e. such that represented a single data (int, double char ...). In case wee needed multiple data we used arrays. With structures we can create **compound** or **composite** data types.

- Usage of functions allowed to organize functionality
- Composite data types allow to organize data

In [ ]:
```
struct structure_name {
    member_type member_mane ;
    member_type member_mane ;
    //...
    member_type member_mane ;
} one or more structure variables ;
```

To access *members* of a structure use . or -> in case of pointers

In [ ]:
```
#include <stdio.h>
#include <string.h>

struct Robot { // Our first structure!
    char name[50];
    double x ,y;
};

int main(){
    printf("Robot program\n");

    struct Robot r1;
    strcpy(r1.name, "R2D1");
    r1.x = 0;
```

```
    r1.y = 0;

    printf("Name: %s, Position %lf, %lf\n", r1.name, r1.x, r1.y);

    struct Robot *pr = &r1;
    printf("%p Name: %s, Position %lf, %lf\n", pr, pr->name,
    pr->x, pr->y);

    printf("%ld", sizeof(struct Robot)); // ???
}
```

Add a function that perform operation on a structure Robot

In [ ]:
```
#include <stdio.h>
#include <string.h>

struct Robot {
    char name[50];
    double x ,y;
} r1 ; // Define a variable of type struct Robot

void print_robot(struct Robot *r)
{
    printf("Robot name: %s\n", r->name);
    printf("Position x=%lf y=%lf\n", r->x, r->y);
}

int main () {
    // r1 allready defined and global
    strcpy( r1.name, "aaaa" );
    r1.x = 8.0; r1.y = 0;

    struct Robot r3;
    strcpy( r3.name, "bbb" );
    r3.x = 1.0; r3.y = 0;

    struct Robot *p = &r3 ;
    p->y=5.0;

    print_robot(&r1);
    print_robot(&r3);
}
```

1.3 Pointers to functions

Do functions have addresses? **Yes!**
Have two functions and print their addresses:

```
In [ ]: #include <stdio.h>
        #include <string.h>

        void f1()
        {
            printf("Hello from f1 %p\n", f1);
        }

        void f2()
        {
            printf("Hello from f2 %p\n", f2);
        }

        int main () {
            printf("Address of f1 is %p\n", f1);
            printf("Address of f2 is %p\n", f2);

            f1();
            f2();
        }
```

So functions have addresses and we can pass those addresses, as we would pass address of *normal* variables. Or in other words, we can make function accept other functions as arguments!

```
In [ ]: #include <stdio.h>
        #include <string.h>

        void f1()
        {
            printf("Hello from f1 %p\n", f1);
        }

        void f2()
        {
            printf("Hello from f2 %p\n", f2);
        }

        // This function accepst an argument of type void
        //that is a function with empty argument list
        void funcaller( void ff(void) )
        {
            printf("A call from a funcaller: ");
            ff(); // A call to the argument function
        }

        int main () {
            printf("Address of f1 is %p\n", f1);
            printf("Address of f2 is %p\n", f2);
```

```
    funcaller(f1);
    funcaller(f2);
}
```

Note the argument list of the *funcaller(void ff(void))* and the way we used it in lines 26 and 27.

This ability of C language is very important to us. We now gained the ability to write generic functions. I.e. such that can work with a multitude of other functions, as long as the **interface** to the function is maintained. (In the above example we could have used any function that is of type *void* and needs no arguments.)

We conclude with an example. Let us assume we are designing a function to calculate integrals of other functions (integrate some f(x) for the values of x ranging from a to b). We will not perform any inegration (would you be able to propose a valid algorithm?), but instead write a general *interface* for such a function.

In [ ]:
```c
//%cflags:-lm

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

//Some inegration function
// f is of type double and accepts a single argument
double integrator(double a, double b, double f(double))
{
    printf("%lf %lf\n", f(a), f(b));
}

double fun1(double x){
    return x*x - 2*x + 5;
}

double fun2(double x){
    return sin(x) * cos(x) * exp(2.0*x);
}

int main () {
    printf("Use integrator with fun1\n");
    integrator(4, 5, fun1);
    printf("Use integrator with fun2\n");
    integrator(4, 5, fun2);
    printf("Use integrator with some other functions\n");
    integrator(4, 5, sin);
    integrator(4, 5, cos);
    //integrator(4, 5, pow); we can not do this!
}
```