Lecture 4
C Programming Language

Binary File I/O

- (See Section 8.8 of [IACU]
- The main functions for binary file I/O are fread, fwrite, fopen, fclose.

```
size_t fread(void *ptr,size_t s,size_t n, FILE * stream);
size_t fwrite(void *ptr, size_t s, size_t n, FILE *stream);
```

Example:

```
FILE *fp;
char x[500];
if ((fp=fopen(“pic.bmp”,”rb”))==0)
{
        fprintf(stderr,”Can’t open file\n”);
        exit(1);
}
fread(x,10,1,fp);
fclose(fp);
```

String I/O

- puts writes a string to standard output
  gets reads a string from standard output,
  and interprets the newline character as '\0'.
  `int puts(char *);  char *gets(char *);`
- fgets and fputs are the same functions for streams (files).
  ```
  char *fgets(char *s,int size,FILE * fp);
  int fputs (char *s,FILE *fp);
  ```
- Other commands for strings:
  ```
  #include <stdio.h>
   …..
      FILE *fp;
      float version;
      char name = "John Doe";
      char buff[1000];
      printf("%s",name);
      fprintf(fp,"%s",name);
      fscanf(fp,"%s",&buf);
      sprintf(buf,"%s %f",name,version);
  ```

sprintf, sscanf, strlen

- int sprintf(char *s, const char *format,…);
  int sscanf(const char *s, const char *format, …);
  string versions of the functions printf() and scanf().   Example:
  char str1[] = “1 2 3 go”, str2[100], tmp[100];
  int a,b,c,;
  sscanf(str1,”%d%d%d%s”, &a, &b, &c, tmp);
  sprintf(str2,“%s %s %d %d %d\n”,tmp, tmp, a ,b, c);
  printf(“%s”,str2);
- String Length: strlen(s) returns length of s:
  size_t strlen(char const *s);
  /* const here means that s cannot be changed
  from within strlen */
  (size_t is defined in /usr/include/stddef.h as unsigned int - caution !!!)

Caution !

- Make sure that the destination array is large enough to hold the string:
  Example:
  char * strcpy(char *dst, char const *src);
  char message[] = “Original message”
  strcpy(message, “Different”);
  /* lost bits */
  strcpy(message, “A different message”);
  /* overwrite memory */ Same in:
  strcat, strncpy /* when s > d */
- **All string library functions** consider ‘\0’ as the end of a string and check for it themselves (don’t use strlen).
  If no ‘\0’ exists at the end of the string, the output of some functions will be wrong.
  char dst[100], s[] = “abc”;
  s[3] = ‘d’; /* overwrites ‘\0’ */
  strcpy(dst,s); /* oops !!! */ Same in:
  strncpy, strncat, strncmp /* when s > d */

strcmp, strtok

- ***int strcmp(char const *s1, char const *s2);***
  Beginners mistake:
  if (strcmp(a, b)) …
  If a == b then strcmp(a,b) == 0 !!!
  If a > b strcmp(a,b) > 0 (and < 0 if a<b)
  (return value is not 1 or -1)

- ***char *strtok(char *s1, const char *s2);***
  searches for tokens in s1 using token-separators given in s2.   Example:

```
void print_tokens(char *line)
{
   char whitespace[] =" \t\f\r\v\n";
   char *token;

   for (token=strtok(line,whitespace); token !=NULL;
                            token=strtok(NULL,whitespace))
        printf("Next token is %s\n", token);
}
```

String Library functions

- char *strcat(s,cs) Concatenates a copy of cs to end of s; returns s.
  char *strncar(s,cs,n) Concatenates a copy of at most n characters of cs to end of s; returns s;

  char *strcpy(s,cs) Copies cs to s including \0. returns s.
  char *strncpy(s,cs,n) Copies at most n characters of cs to s; returns s; pads with \0 if cs has less than n characters

  char *strtok(s,cs) Finds tokens in s delimited by characters in cs.
  size_t strlen(cs) Returns length of cs (excluding \0)
  int strcmp(cs1,cs2) Compares cs1 and cs2; returns negative, zero, or positive value for cs1 <,==, or > cs2 respectively
  int strncmp(cs1,cs2,n) Compares first n characters of cs1 and cs2; returns as in strcmp.

  char *strchr(cs,c) Returns pointer to first occurrence of c in cs
  char *strrchr(cs,c) Returns pointer to last occurrence of c in cs
  char *strpbrk(cs1,cs2) Returns pointer to first char in cs1 and cs2
  char *strstr(cs1,cs2) Returnss pointer to first occurrence of cs2 in cs1
  ==> The 4 above functions return NULL if search fails
  size_t strspn(cs1,cs2) Returns length of prefix of cs1 consisting of characters from cs2
  size_t strcspn(cs1,cs2) Returns length of prefix of cs1 consisting of characters not in cs2

- Note: This table is from page 140 in [IACU]
  More exciting string functions in /usr/include/string.h ...

Const type qualifier

- const int SIZE = expression;
- There are no changes after initialization
- When is const preferred over enum or macro?
  - Its value is decided at run time.
  - It is used where its address (& operator) is required.
  - It must be recognized by the compiler/ debugger.
  - In trying to force a function not to modify an array argument, or any argument that is passed by its address.
- Example:

```
int foo(const int arr[],int exp) {
        ….
        Arr[j] = exp;
        /* The compiler may shout here !*/
}
```

typedef declarations

- C provides a facility for creating new data type **names**
- typedef int Length;   /* Definition */
  Length len, arr[SIZE];   /* Use */
  Now len is of type int and arr of type array of int.
  Another example:
  typedef char * String; /* String is a string… */
- typedefs are far from being macros
- const int BUF_SIZE = 8;
  const int SIZE = 100;
  typedef char Buf[BUF_SIZE];
  Buf buffer, buf_array[SIZE];
  Now buffer is an array (of size BUF_SIZE).
        buf_array is an array (of size SIZE) of arrays (of BUF_SIZE).
  ⇨ equivalent to:
        char buf_array[SIZE][BUF_SIZE];

Why **typedef** ?

- **Easy modification of data types**
  Example:
  Certain **int** variables are used for carrying flags. Later, the software became more complicated, and we want to change these variables into type **long**.
  Had these variables been declared of type **Flag** (with "typedef int Flag;"), all can be done by modifying the typedef statement.

- **Readability, Documentation**
  Example:
  Meaningful names for data types:
  In the example above, wherever we see the declaration "Flag var;" we understand that 'var' is going to be used as a "flag carrier".

Scoping rules

- An identifier declared in one part of the program can only be accessed (used) from a certain area in the program. This area is determined by the identifier’s **scope**.
- **Block Scope:**

```
1: {
       int k;
                 …
                 2:{ …..
                            }
       …
   } ==> k is defined within block 1
```

Any statement within block 1 or 2 can use k, any statement outside block 1 cannot. Also, within block 1, k overrides any other identifier k previously declared.

Scoping rules

- **File Scope:**

```
int k;
1: {
        …
                2:{ …..
                        }
 }
3: {
       ……
   }
```

Any identifier declared outside of all blocks has file scope: it can be accessed anywhere within the file in which it was declared. (Note that #include statements treat the identifiers declared in the header file as if they were declared in the including file).

Scoping rules

- **Prototype Scope:**
  void func1(int count);
  This scope applies only to argument names declared in function prototypes - the names of the arguments in the declaration don’t matter, they don’t have to match the formal parameters in the function definition.

- **Function Scope:**
  ```
  void func1(int counter);
  {
          …
   }
  ```
  The identifier counter is only accessible within the function block.

Scoping rules - Example

-
  ```
  {
     int a = 1, b = 2, c = 3;
     printf("%d %d %d\n",a, b, c);
     {
         int b = 4;
         float c = 5.0;
         printf("%d %d %g\n", a, b, c);
         a = b;
         {
            int c;
            c = b;
            printf("%d %d %d \n", a, b, c);
          }
         printf("%d %d %g\n", a, b, c);
     }
     printf("%d %d %d \n", a, b, c);
  }
  ```

Automatic variables and variables scoping

- **Internal** variables - declared inside a function or a block.
  These variables are **local**, or private, to the block in which they are declared and cannot be accessed from the outside.
  They are usually **automatic**: they only come into existence when the function or block is entered and are destroyed automatically after exit from the block (function).
  Static variables - internal variables that are created and initialized at compile time. They retain their value even after exit from the block (function).
- Example: A function that keeps track of how many times it is called:
  ```
  static int my_count = 0;
  my_count++;
  ```

Automatic variables and variables scoping - cont.

- **External** or **Global** variables - variables declared outside functions, and can be used and changed by several functions
- External variables always exist and retain their value until the entire program is terminated
- Provide a way for function to communicate
- a function can use a variable if it was
  - defined earlier in the file
  - declared extern earlier in the file
  - declared extern in the function
- Example:

```
/*file1.c*/
        int  my_count = 0;
        char * foo();
         …
        foo();
```

```
/*file2.c*/
extern int  my_count;
extern char *foo();
…
my_count++;
```

Declaration vs. Definiton

- **Declarations** specify information for the compiler. Example:
```
extern int x; /* external variable declaration*/
float square(float); /* function prototype
                          declaration */
typedef int * Link; /* typedef declaration */
```
- When a declaration also caused storage allocation, it is called a **definition.** Example:
```
int x;
Link y;
```
- In C, a variable can be defined only once but can be declared many times.

assert

- A macro supplied in C, that enables useful debugging tools:
  void assert(int expression);
  Can be found in assert.h.
- Example:
  ```
  int a,b,c;
  ...
  scanf("%d %d", &a, &b);
  …
  c = some_func(a,b);
  assert( c > 0);
  ```
- If the expression passed to assert is false, the system will abort execution and print an error message including the expression, the name of source file and line number of the assertion. Without the assertion, the program might continue to run and fail later.

core dump

- When a program has a run-time error it generates a file named core
- The core file stores a copy of the memory at the point when the program exits due to error
- note: core files are usually quite large so it’s a good idea to delete them afterwards
- Using the proper commands of gdb will help you track down where the program had a run-time error