C PROGRAMMING
Lecture 6

1st semester 2021 - 2022

Input/Output

- Most programs require interaction (input or output.
- I/O is not directly supported by C.
- I/O is handled using standard library functions defined in <stdio.h>.
  - stdio.h header file contains function declarations for I/O.

Standard I/O Library

- Streams and FILE objects
- Three types of buffering
- Open a stream
- read/write a stream
- Positioning a stream
- Formatted I/O

Streams and FILE Objects

- Standard I/O
  - File stream driven - FILE*
  - Associate a file with a stream and operate on the stream
  - 3 pre-defined streams
    - stdin
    - stdout
    - stderr

Buffering

- Goal of buffering is to minimize the number of read and write calls.
- Three types of buffering
  - Fully buffered – Block Buffered
  - Line buffered
  - Unbuffered
- ANSI C requirements
  - stderr must never be fully buffered
  - stdin and stdout are fully buffered if the device they use is not interactive

Buffering

- On most UNIX/Linux systems
  - stderr is always unbuffered
  - Streams for terminals are line buffered
  - All other streams are fully buffered

Setting Buffer Type

- setbuf

```
void setbuf(FILE *stream, char *buf);
```

- Enable or disable buffering on a stream
  - If buf is NULL buffering is disabled
  - Buffer must be of size BUFSIZ as defined in <stdio.h>

Setting Buffer Type

- setvbuf

```
int setvbuf(FILE *stream, char *buf, int mode , size_t size);
```

  - Can set all 3 types of buffering depending on value of *mode*
    - _IOFBF - Fully buffered
    - _IOLBF - Line buffered
    - _IONBF - Non buffered
  - If buf is NULL, system will create its own buffer automatically
  - buf is ignored if mode is _IONBF
  - `setbuf(fp, NULL)` is the same as `setvbuf(fp, buf, _IONBF, size)`

Opening a Stream

```
FILE *fopen(const char *path,const char *mode);
FILE *freopen(const char *path,const char *mode,FILE *stream);
FILE *fdopen(int fildes, const char *mode);
```

- fopen opens the file given by *path*.
- freopen opens a file on a specified stream, closing the stream first if it is already open
- fdopen opens a stream from a file descriptor. Useful for streams that don't have a regular file such a pipes

Opening a Stream

- mode
  - r or rb
  - w or wb
  - a or ab
  - r+ or r+b or rb+
  - w+ or w+b or wb+
  - a+ or a+b or ab+

Reading and Writing a Stream

- Three ways to read and write
  - One character at a time
  - One line at a time
  - Direct (binary) I/O
- Character at a time

```
int getc(FILE *stream);
int fgetc(FILE *stream);
int getchar(void);
```

Reading and Writing a Stream

- Handling errors

```
int feof(FILE *stream);
int ferror(FILE *stream);
void clearerr(FILE *stream);
```

- "Unreading"
  - We can put a single character back into the stream

```
int ungetc(int c, FILE *stream);
```

Reading and Writing a Stream

```
int putc(int c, FILE *stream);
int fputc(int c, FILE *stream);
int putchar(int c);
```

- Returns c if ok, EOF on error

- getc and putc may be more efficient than fgetc and fputc because macros do not have the overhead of calling a function

Reading and Writing a Stream

- Line at a time I/O
  - Read

```
char *gets(char *s);
char *fgets(char *s,int size,FILE *stream);
```

  - Write

```
int fputs(const char *s,FILE *stream);
int puts(const char *s);
```

Example – v1

```
#include <stdio.h>
int main ()
{
   FILE *fp;
   int i, c, cnt=0;
   char buff[1000];
   fp = fopen("file1.txt","r");
   if(fp == NULL)
   {
      perror("Error in opening file\n");
      return(-1);
   }
```

Example – v1

```
 do{
        for (cnt=0;(c=fgetc(fp)) !='\n'&& c!=EOF;cnt++){
                buff[cnt] = c;
        }
        buff[cnt] = '\0';
        for(i=cnt-1;i>=0;i--)
                printf("%c", buff[i]);
        printf("\n");
        cnt=0;
        if( feof(fp) )
                break ;
   }while(1);
   fclose(fp);
   return(0);
}
```

Example – v2

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main ()
{
   FILE *fp;
   int c;
   char buff[1000];
   fp = fopen("file1.txt", "r");
   while (1) {
        if (fgets(buff,1000, fp) == NULL) break;
        for(c=(int)strlen(buff);c>=0;c--)
                printf("%c",buff[c]);
  }
return 0;
}
```

Reading and Writing a Stream

- Direct (binary) I/O

```
size_t fread(void *ptr,size_t size,size_t nmb, FILE *stream);
```

  - Read *nmb* items of size *size* from *stream* and store them at the location pointed to by *ptr*

```
size_t fwrite(const void *ptr,size_t size,size_t nmb,FILE
   *stream);
```

  - Write *nmb* items of size *size* to *stream* from the location pointed to by *ptr*
- Useful for reading/writing arrays and structs
- Returns number of items written

Example

```
#include <stdio.h>
#include <string.h>
int main (int argc, char* argv[]) {
    char linie[1000], *p, *q;
    int propozitii = 0;
    FILE* f = fopen(argv[1], "r");
    for (;; ) {
        p = fgets(linie, 1000, f);
        if (p == NULL) break;
        for (p = linie, q = NULL;; ) {
            q = strchr(p, '.');
            if (q == NULL) break;
            propozitii++;
            p = q + 1;
        }
```

Example

```
 for (p = linie, q = NULL;; ) {
            q = strchr(p, '!');
            if (q == NULL) break;
            propozitii++;
            p = q + 1;
        }
        for (p = linie, q = NULL;; ) {
            q = strchr(p, '?');
            if (q == NULL) break;
            propozitii++;
            p = q + 1;
        }
    }
    fclose(f);
    printf("In \"%s\" are %d sentences\n", argv[1], propozitii);
    return 0;
}
```

Positioning a Stream

```
int fseek(FILE *stream, long offset, int whence);
```

- whence must be one of the constants SEEK_SET, SEEK_CUR, or SEEK_END, to indicate whether the offset is relative to the beginning of the file, the current file position, or the end of the file, respectively.
- Returns 0 if ok, non zero on error

```
long ftell(FILE *stream);
```

- Returns current offset if ok, or -1 on error

```
void rewind(FILE *stream);
```

- Same as fseek(fp, 0L, SEEK_SET)

Example

```
#include <stdio.h>
int main ()
{
   FILE *fp;
   fp = fopen("file.txt","w+");
   fputs("The first  string written", fp);
   fseek( fp, 4, SEEK_SET );
   fputs("second", fp);
   fclose(fp);
   return(0);
}
```

Formatted I/O

- Output

```
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char
  *format, ...);
```

- Input

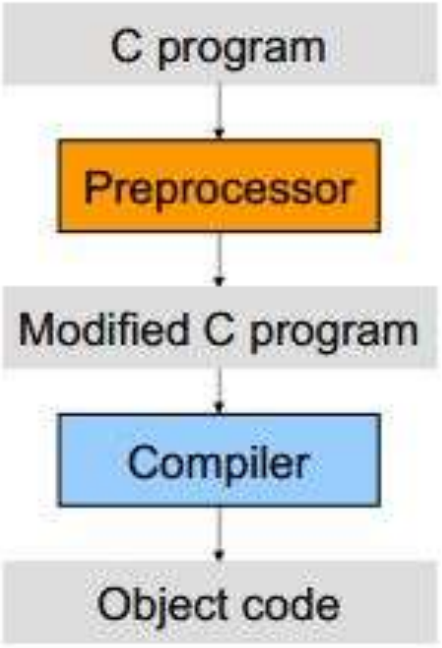
```
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str,const char *format, ...);
```

Example

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

int main()
{
   int mark;
   char student_firstname[50], student_lastname[50], in_string[100];

   strcpy( in_string, "John Doe 9" );
   sscanf( in_string, "%s %s %d", student_firstname,
student_lastname, &mark );

   printf("Student %s %s received %d at the exam;\n",
student_firstname, student_lastname, mark );

   return(0);
}
```

C Preprocessor

- Before a C program is compiled, the source code file is first handled by a preprocessor
- The output of preprocessor (C code) is the actual input for the compiler



Preprocessor Role

- The actions:
  - Removes comments and whitespaces from the source code
  - Acts according to preprocessor directives
  - Expands some kind of characters
  - ! Does not compile code
- Preprocessor directive:
  - Command recognized only by preprocessor
  - Each directive prefixed by #

Preprocessor Role

- Directives and facilities provided for:
  - File inclusion
    - #include
  - Macro expansion
    - #define
  - Conditional compilation
    - #if, #ifdef, #ifndef, …

File Inclusion

- Specifies that the preprocessor should read in the contents of the specified file
  - usually used to read in type definitions, prototypes, etc.
  - proceeds recursively
    - #includes in the included file are read in as well
- Two forms:
  - #include <*filename*>
    - searches for filename from a predefined list of directories
    - the list can be extended via "gcc –I *dir*"
  - #include "*filename*"
    - looks for *filename* specified as a relative or absolute path

File Inclusion

- a header file has file-extension '**.h**'
  - these header files typically contain "public" information
    - type declarations
    - macros and other definitions
    - function prototypes
  - Most of the time the public information associated with a code file **f.c** will be placed in a header file **f.h**
  - these header files are included by files that need that public information

File Inclusion – Examples

```
/* Load system header for stdio library */
#include <stdio.h>

/* Load system header for math library */
#include <math.h>

/* Load user header file */
#include “file_header.h"

/* Loads user header file */
#include "../My_Lib/mylib.h"
```

File Inclusion

- What happens if we try to include the same file multiple times?

Macros

- A macro is a symbol that is recognized by the preprocessor and replaced by the macro body
- A preprocessor macro is used for:
  - Defining a symbolic name for a constant value
  - Defining an alias for something else
- To define a macro, use this directive:

  #define mname mtext

  mname → macro name, formal parameters allowed

  mtext → substitution text
- Examples:

```
#define   BUFFERSZ    1024
#define   WORDLEN     64
```

- The replacement is not done if mname appears within a character string or is part of a longer identifier

Macros

- The preprocessor expands macros in the C code
- Wherever it sees ***`mname`*** in the source file, it substitutes ***`mtext`*** in its place
- A macro definition can contain previously defined macros
- Macros are not the same as C variables! No storage is created in memory; macros only exist at compile-time
- An old C programming convention rules that macro names be fully capitalized to differentiate them from C variable names; this is NOT a language rule, hence NOT enforced by the C compiler!
- macros can be defined by the compiler:

gcc –D macroName

gcc –D macroName=definition

Macros - Example

Original file

```
/* EUR rate in RON */
/* Working hours per week */
#include <stdio.h>
#define RATE1 4.485
#define HRS_WK1 40

int main (void)
{
        float rate2 = 4.50;
        float hrs_wk2 = 35;
        float pay1, pay2;

        /* Weekly pay in RON */
        pay1 = RATE1 * HRS_WK1;
        pay2 = rate2 * hrs_wk2;

        printf("RATE1=%f\n",RATE1);
        return 0;
}
```

After expansion

```
[stdio.h definitions]

int main (void)
{
        float rate2 = 4.50;
        float hrs_wk2 = 35;
        float pay1, pay2;

        pay1 = 4.485 * 40;
        pay2 = rate2 * hrs_wk2;

        printf("RATE1=%f\n",4.485);
        return 0;
}
```

Macros replacement - Example

- Replacement text can be **any** set of characters, meaning zero or more
- Example:

```
#define then
```

causes all occurrences of then to be removed from the source file.

```
if(a>b) then
    max=a;
else
    max=b;
```

Parametrized Macros

- Macros can have parameters
  - these resemble functions in some ways:
    - macro definition ~ formal parameters
    - macro use ~ actual arguments
  - Form:

    #define macroName(arg$_1$, …, arg$_n$) replacement_list

    IMPORTANT: no space between macro name and (
  - Example:

```
#define  deref(ptr)        *ptr
#define  max(x,y)          x > y ? x : y
```

Macros or Functions?

- Macros may be (sometimes) faster
  - don't incur the overhead of function call/return
  - however, the resulting code size is usually larger
    - this can lead to loss of speed
- Macros are "generic"
  - parameters don't have any associated type
  - arguments are not type-checked
- Macros may evaluate their arguments more than once
  - a function argument is only evaluated once per call

Macros or Functions?

- Macros and functions may behave differently if an argument is referenced multiple times:
  - a function argument is evaluated once, before the call
  - a macro argument is evaluated each time it is encountered in the macro body.
- Example

```
int dbl(x)
{
    return x+x;
}
…
i=10; j=dbl(i++);
```

```
#define DBL(x) x+x
…
i=10; j=dbl(i++);
```

Macro Properties

- Macros may be nested
  - in definitions, e.g.:

    #define Pi 3.1416

    #define Twice_Pi 2*Pi
  - in uses, e.g.:

    #define double(x) x+x

    #define Pi 3.1416

    …

    if (x > double(Pi)) …
- Nested macros are expanded recursively

Macros - Pitfalls

```
#define MAX 10
#define LOWERMAX MAX – 1
maxvalue=LOWERMAX*5
```

What is the value of maxvalue?

How can this be avoided?

Macros - Pitfalls

```
#define square(n) n*n
y=square(10);
```

Result?

Now, consider

```
x=10;
y=square(x+1);
```

How can this be fixed?

Macros - Pitfalls

Same macro as on previous slide

y=square(10)/square(5)

Solution?

Conditional Compilation

- the selection of lines of source code to be compiled and those to be ignored
- If you have the sequence

  #ifdef *name*
  program text
  #else
  more program text
  #endif

the code that is to be compiled depends on whether a preprocessor macro by that *name* is defined or not.

Conditional Compilation

- If it is (that is, if there has been a #define line for a macro called *name*), then ``*program text*'' is compiled and ``*more program text*'' is ignored. If the macro is not defined, ``*more program text*'' is compiled and ``*program text*'' is ignored.
- This looks a lot like an if statement, but it behaves completely differently:
  - an if statement controls which statements of the program are executed at run time,
  - #ifdef controls which parts of the program actually get compiled.

Conditional Compilation

- Just as for the if statement, the #else in an #ifdef is optional.
- There is a corresponding directive #ifndef, which compiles code if the macro is *not* defined (although the ``#else clause'' of an #ifndef directive will then be compiled if the macro *is* defined).
- There is also an #if directive which compiles code depending on whether a compile-time expression is true or false.

Conditional Compilation

- Conditional compilation is useful in two general classes of situations:
  1. You are trying to write a portable program, but the way you do something is different depending on what compiler, operating system, or computer you're using.

```
#ifdef unix
    unlink(filename);
#else
    remove(filename);
#endif
```

Conditional Compilation

2. You want to compile several different versions of your program, with different features present in the different versions. You bracket the code for each feature with #ifdef directives, and (as for the previous case) arrange to have the right macros defined or not to build the version you want to build at any given time. This way, you can build the several different versions from the same source code.

```
#define DEBUG
#ifdef DEBUG
        printf("x is %d\n", x);
#endif
```

Conditional Compilation

- Use conditional compilation to ensure that a header file is “really included” at most once
  - header file’s responsibility to protect itself from multiple-inclusion problems
  - uses a conditional-compilation directive **#ifndef**
  - in effect sets a flag when a file is included so we don’t include it again
  - relies on convention (should be used as stated)

Bitwise Operations

- Many situation, need to operate on the bits of a data word
  - Register inputs or outputs
  - Controlling attached devices
  - Obtaining status
- ”The combination of pointers and bit-level operators makes C useful for many low level applications and can almost replace assembly code. (Only about 10 % of UNIX is assembly code the rest is C!!.)” *Dave Marshall*

Bitwise Operations

- **&** – AND :results **1** if both operand bits are **1**
- **|** – OR :results **1** if either operand bit is **1**
- **^** – Exclusive OR :results **1** if operand bits are different
- **~** – Complement :each bit is reversed
- **<<** – Shift left : multiply by 2 (see later)
- **>>** – Shift right : divide by 2 (see later)

Bitwise Operations

- **DO NOT** confuse & with &&: & is bitwise AND, && logical AND. Similarly for | and ||.
- ~ is a unary operator -- it only operates on one argument to right of the operator.
- The shift operators perform appropriate shift by operator on the right to the operator on the left. The right operator must be positive. The vacated bits are filled with zero (***i.e.*** There is **NO** wrap around).

Bitwise Shift Operations

- If an int can represent all values of the original type, the value is converted to an int ; otherwise, it is converted to an unsigned int. These are called the integer promotions. All other types are unchanged by the integer promotions.
- Shift operations shift the bits in an integer to the left or right
- Operands may be of any integer type (including char).
- The result has the type of the left operand after promotion.

Bitwise Shift Operations

- i << j : shifts the bits in i to the left by j places; for each bit that is “shifted off” the left end of i, a 0 bit enters at the right.
- i >> j : shifts the bits in i to the right by j places; if i is of an unsigned type or if the value of i is nonnegative, 0s are added at the left as needed; if i is negative, the result is platform-dependent.
- Operands may be of any integer type, but use unsigned for portability

Bitwise Shift Operations

- For example: *x* << 2 shifts the bits in *x* by 2 places to the left.

  if *x* = 00000010 (binary) or 2 (decimal)
  then:
  x>>=2 => x=00000000 or 0 (decimal)

- Also: if *x* = 00000010 (binary) or 2 (decimal)
  then:
  x<<=2 => x=00001000 or 8 (decimal)

- Therefore a shift left is equivalent to a multiplication by 2.
- Similarly a shift right is equal to division by 2
- **NOTE**: Shifting is much faster than actual multiplication (*) or division (/) by 2. So if you want fast multiplications or division by 2 ***use shifts***.

Precedence

- The bitwise shift operators have lower precedence than the arithmetic operators, which can cause surprises:
- i << 2 + 1 means i << (2 + 1), not (i << 2) + 1
- Each of the ~, &, ^, and | operators has a different precedence:

Highest: ~
&
^

Lowest: |

Examples:

i & ~j | k means (i & (~j)) | k

i ^ j & ~k means i ^ (j & (~k))

Using parentheses helps avoid confusion.