# Applied Programming

# **Input-Output in C**

# Command lines

- "C" has a standard entry point called "main"
  - Provides the ability to pass **N string** arguments
- Prototype:
  int main(int argc, char *argv[], char*env[])
  - argc – the number of arguments on the line
  - argv – pointer to the read only argument strings
  - env – pointer to the read only environment strings

(not used in this class)

- There will always be at least ONE parameter on non-embedded systems
  - arg[0] - the name of the program

# Command lines

- Given: int main(int argc, char *argv[]) **and**
  myprog one 2 3.0

- What will happen?
  - argc = 4 Note: An integer
  - argv[0] = "myprog"
  - argv[1] = "one"
  - argv[2] = "2" Note: A STRING!
  - argv[3] = "3.0" Note: A STRING!

- To use argv[2] and argv[3] you must convert from string to integer (atoi) or float (atof)

# Command parsing bugs

- Your user WILL enter MORE or FEWER parameters than you request, your code must handle it.
  - ALWAYS CHECK **argc** FIRST!

- Assume that you are writing code that will perform simple mathematics on 2 numbers.
  - E.g. Math num1 op num2
    - Math 1 + 2
    - Math 1 2
    - Math 1 +2

- "Dumb" parsing code

```
Num1 = atoi(argv[1]);        /* converts to an integer */
Op   = argv[2];              /* just copies the pointer */
Num2 = atoi(argv[3]);
```

# Parsing: Math 1 + 2

- "Dumb" parsing code

```
Num1 = atoi(argv[1]);
Op   = argv[2];
Num2 = atoi(argv[3]);
```

- Works great, all the variables have the values we hoped for. ☺

# Parsing: Math 1 2

- "Dumb" parsing code

```
Num1 = atoi(argv[1]);
Op   = argv[2];
Num2 = atoi(argv[3]);
```

- The user forgot the operator so there are only 2 parameters, not three! This code will CRASH!
  - Num1 – ok
  - Op – will point to "2"
  - Num3 will core dump because there isn't any argv[3]

- We needed to check argc FIRST to make sure all the parameters we need are there.

# Parsing: Math 1 +2

- "Dumb" parsing code

```
Num1 = atoi(argv[1]);
Op   = argv[2];
Num2 = atoi(argv[3]);
```

- The user didn't put a space between the "+" and the "2" so there are only 2 parameters! This code will CRASH!
  - Num1 – ok
  - Op – will point to "+2",
  - Num3 will core dump because there isn't any argv[3]

- Again, we need to check argc FIRST to make sure all the parameters we need are there.

# Sample code

```
/* Prints out argv and env variables */
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[], char *env[]) {
  int i; /* counter */
  /* Dump the arguments */
  for (i = 0; NULL != argv[i]; ++i){
      printf("arg[%2d]= %s\n", i, argv[i]);  }
  printf("\n\n");

  /* Dump the environment */;
  for (i = 0; NULL != env[i]; ++i){
      printf("env[%2d]= %s\n", i, env[i]);   }
  return 0;
}
```

# Input/Output in C

- **All input-output operations are abstracted as *reading and writing to files***
  - **the concept of stream did not exit yet**

- **C provides the following *standard I/O files* (defined in `<stdio.h>` )**

| | file name | | device "attached to" |
|---|---|---|---|
| ▪ **Input file:** | stdin | — | ***keyboard*** |
| ▪ **Output file:** | stdout | — | *standard* ***display*** |
| ▪ **Error file:** | stderr | — | *error* ***display*** |

# I/O Functions in C

- **The Standard I/O library:** <stdio.h>
- **Functions for output:** printf() **family**
  - printf() — **to standard output (file `stdout`)**
  - fprintf() — **to any file**
  - sprintf() — **to a string**

- **Functions for input:** scanf() **family**
  - scanf() — **from standard input (file `stdin`)**
  - fscanf() — **from any file**
  - sscanf() — **from a string**

# Output with `printf`

- **Prototype:**

```
int printf(const char *format, ...)
```

- **Usage:**

```
printf("format-string", variable-list);
```

***Important***

- **`printf` *returns* and `int` with the *number of characters printed*. In case of *error* it returns a *negative value***

**I/O functions can be called with "any" number of arguments. The number of arguments depends on the `"format-string"` specifications.**

**Warning: C does not care if the format-string matches the number of arguments given.**

# printf:format-string

- **Contains *3 types of specifications* (each of which is optional) *enclosed in double quotes***
  1. **Literal text: Characters to be printed to the output "literally".**
  2. **Escaped text: Special characters to be translated into *control characters***
  3. **Formatting string: Used to specify the *type and format* of variables to be printed**
     - **type: signed or unsigned integers, float, char, string, etc.**
     - **format: field width, alignment, decimals places, etc.**

# Escaped Text

- **Similar to Python, backslash (“\”) indicates escaped text (\ is sometimes called the “escape character”)**
  - **Used for “reserved” and “special” characters (in the old days of tele types - `tty`)**
  - **Example: `\n` is probably the mostly used one**

```
printf (“Here is my text!\n”);
```

- **Common Special Characters (similar to Python)**
  - **`\\`: Backslash**
  - **`\n`: New line (i.e., CR-LF pair)**
  - **`\t`: Horizontal Tab**
  - **`\'`: Single quote**
  - **`\"`: Double quote**
  - **`\?`: Question mark**
  - **`%%`: Percent sign (seems an odd choice)**

# Formatting String

- **Starts with a percent sign ("%") [like Python]**
  - **Value converted according formatting descriptor**
  - **C has many formatting descriptors**
- **Common Formatting Descriptors**
  - **`%d`: int**
  - **`%u`: unsigned int**
  - **`%c`: char**
  - **`%s`: string ( i.e., *pointer to char array*)**
  - **`%f`: float or double (*)**
  - **`%x`: hexadecimal, using lowercase letters**
  - **`%X`: Hexadecimal, using uppercase letters**
  - **`%ld`: long int**
  - **`%lu`: unsigned long int**

# Formatting String

- **More parameters are available to format the output *field width*, *number of decimals*, etc. (see examples)**
- **Examples:**
  - printf("%8d",123456): **8 "places" wide (right justified)**

    XXXXXXXX

      123456
  - printf("%6.2f",1.236): **float 6 "places" wide (including the dot), rounded to 2 digits after decimal point (point takes up one place)**

    XXX.XX

      1.24
  - printf("%06X",1236) : **Hexadecimal, 6 "places" wide, padded with "0"s to fill the width**

    0003E8

**Tip: To format hexadecimals use "`0x%04X`" to avoid confusion**

# Printf() Example

```
/* Example: Output in C  ex1_printf.c      *
 * printing strings                        *
 * Author: Juan C. Cockburn                */
#include <stdio.h>
int main(){
int    a1 = 67;
float  f1 = 1.45F;
double d1 = 15.63145;

printf("**** Using printf ********\n\n");
printf("Addresses of Variables \n");
printf("int a1    %p\n",(void*)&a1);
printf("float f1  %p\n",(void*)&f1);
printf("double d1 %p\n\n",(void*)&d1);
printf("Values of variables \n");
printf("%%d for int   : %d\n",a1);
printf("%%c for char  : %c\n",a1);
printf("%%f for float : %f\n",f1);
printf("%%e for scientific : %e\n",d1);
printf("shorter vararg list: a1=%d, f1=%f,
          d1=%f\n",a1,f1);
printf("longer  vararg list: a1=%d,
         f1=%f, d1=%f\n",a1,f1,d1,a1);
printf("wrong format types:  a1=%f,
         f1=%u, d1=%c\n",a1,f1,d1);
return(0);}
```

```
**** Using printf ********

Addresses of Variables
int a1    0x7fff54d6c0fc
float f1  0x7fff54d6c0f8
double d1 0x7fff54d6c0f0

Values of variables
%d for int   : 67
%c for char  : C
%f for float : 1.450000
%e for scientific : 1.563145e+01
shorter vararg list: a1=67,
         f1=1.450000, d1=0.000000
longer  vararg list: a1=67,
         f1=1.450000, d1=15.631450
wrong format types:
         a1=1.450000, f1=67, d1=C
```

# Input with `scanf()`

- **Prototype:**

```
int scanf(const char *format, ...) ;
```

**Usage:**

```
scanf("format-string", variable-list);
```

***Important***

- **`scanf()` *returns* an int with the *number of successful conversions* performed.**

- **Notes:**
  - **The *address of the variable* is passed into `scanf()`, instead of the value of the variable.**
  - **The format-string specifications are *almost the same* (as in `printf`) but the *control (escaped) characters are not used***

# Scanf() Example

What is wrong with this program?

```
#include <stdio.h>                          /* ex2a_scan.c */
const double mi2km = 1.609;
double convert(double mi) {return (mi * mi2km);}
int main()
   {
   double miles;
   printf("Miles to Km Conversion (enter negative number to end)\n");
    do {
       printf("Input distance in miles: ");
       scanf("%f", &miles);
       printf("\n%f miles = %f km\n",
                              miles, convert(miles));
   } while (miles > 0);
   return 0;
}
```

```
 Miles to Km Conversion (enter negative number to end)
Enter Miles: 2.45
  2.4500 miles =   3.9421 km
Enter Miles: 1
  1.0000 miles =   1.6090 km
Enter Miles: -1
```

# Example

```
#include <stdio.h>                          /* ex2a_scan.c */
const double mi2km = 1.609;
double convert(double mi) {return (mi * mi2km);}
int main()
   {
   double miles;
   printf("Miles to Km Conversion (enter negative number to end)\n");
   do {
      printf("Input distance in miles: ");
     if (scanf("%f", &miles) <= 0)
        {
        printf("Error, nothing parsed\n");
        return(-99);
        }
      printf("\n%f miles = %f km\n", miles, convert(miles));
   } while (miles > 0);
   return 0;
   }
```

**Without this change, the program will infinite loop when alpha text is entered! E.G. “abcd”**

**ALWAYS CHECK RETURN CODES!!!**

# Scanf – bugs 1

Error checking removed

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
FILE *DataFile = NULL;
char String[10];
DataFile = fopen("test.txt", "r");
while (EOF != fscanf(DataFile, "%s", String)) {
   if (strlen(String) >= 10) {
      printf("Error: Too long: %d\n", strlen(String));
      continue;
   }
printf("String '%s' %d\n", String, strlen(String));
}
```

Test.txt
12345678
123456789AB

stdout
String '12345678' 8
Error: Too long: 11

Our buffer is 10
One string is 11+NULL (or 12)
Q: Where did the extra bytes go?
A: After the end of our String array, corrupting the stack! Scanf CAN do any length checking!

# Scanf – bugs 2

Fixed Bug #1

```
#include <stdio.h>          /* A safer fscanf() */
#include <string.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
FILE *DataFile = NULL;
char buffer [31] = {"xxxxxxxxxx          ----------"};
char *String;
String = &buffer [10];
DataFile = fopen("test.txt", "r");
printf("Before read= '%s'\n", buffer);
while (EOF != fscanf(DataFile, "%9s", String
   printf("After read=: '");
   for (int i = 0; i < sizeof(buffer); i++) { printf("%c", buffer[i]);}
   printf("'\n");
   printf("String '%s' %d\n", String, (int) strlen(String));
}
return(0); }
```

**Test.txt**
**12345678**
**123456789AB**

```
Before read= 'xxxxxxxxxx          ----------'
After read=: 'xxxxxxxxxx12345678  ----------'
String '12345678' 8
After read=: 'xxxxxxxxxx123456789 ----------'
String '123456789' 9
After read=: 'xxxxxxxxxxAB 456789 ----------'
String 'AB' 2
```

**Note: Data NEVER exceeds our ‘9’ limit BUT parse can be WRONG**

# Dynamic scanf string checking

- Hardcoding a buffer size is wrong
  - Not supportable
- Dynamically create the checking string
- And check the length!

```
char String[MAX_BUFF_SIZE+2]; /* the fscanf buffer */
char formatStr [32];          /* Build dynamic length */

/* Build a dynamic format string like    %255s    */
sprintf(formatStr, "%c%d%c", '%', MAX_BUFF_SIZE+1, 's');

while (EOF != fscanf(InputFile, formatStr, String)){
  if (strlen(String) >= MAX_BUFF_SIZE)       {
       printf("Error: Input data too long\n");
       exit(99);}}
```

# Scanf – bugs 3

Error checking removed

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
FILE *DataFile = NULL;
int num, v1, v2, v3;
DataFile = fopen("test.txt", "r");
while ((num=fscanf(DataFile,"%d%d%d",&v1,&v2,&v3))>0){
  printf("%d parameters %d %d %d\n", num, v1, v2, v3);
} }
```

Test.txt

```
1 2 3
4 5 6 7
8 9
```

Q: Did this do what you expected?
A: No, it never told me about the extra long or short lines!

stdout

```
3 parameters 1 2 3
3 parameters 4 5 6
3 parameters 7 8 9
```

Note: "%d%d%d“, %d %d %d“ or "%d  %d  %d“ produce the same results.

# Using `scanf()`

**Not robust - ONLY Useful when your data is "clean"**

- Example: `scanf("%d",&height);`
  - `scanf()` will make only minor considerations when expecting values of a particular type
  - ***Non-integer values*** entered to `stdin` will result in `height` being set to the ***value 0***
  - ***Whitespaces*** (space, tab and enter characters) will be ***ignored*** by `scanf()`

- **`Use the buffer size when reading pure strings "%255s"`**

# I/O with general files

- **To use files declare the file handle as a pointe to file: `FILE*`**
- **Some C functions in <stdio.h>**

```
FILE *fopen()  — open file
int fclose()   — close file
int fseek()    — move the file pointer
int fflush()   — flush output buffer
int feof()     — check end-of-file
```

- **Important:**

**Functions in the standard library that use a pointer to `FILE` are *usually buffered.***

# Buffered I/O

**Operating systems don't immediately write data out to the final device**

- **Why do I care?**
  - **If your system crashes you will loose some of your output**
  - **If you are debugging your code you MIGHT NOT SEE YOUR OUTPUT!**

```
If you REQURE the data, flush the buffer!
e.g.
fprintf(stderr,“Message I want\n");
fflush(stderr);
```

# C File open with `fopen()`

- **Syntax, declare: `FILE* file_p`**

```
file_p = fopen("file name", "file mode");
```

- **Valid file modes**
  - `r` read text file (`r+` , read & write)
  - `w` write text file (`w+`, write & read)
  - `a` append to text file
  - `rb` read binary file
  - `wb` write binary file
  - `ab` append to binary file

- Note: If you forget to **fclose** a FILE* you get a **memory leak** AND you could run out of file handles (access)

# Read Files with `fread()`

- **Read from handle into buffer**

```
nRead = fread(void *buffer, int esize, int elem, FILE *handle);
```

- **Notes:**
  - **Reads "elem * esize", make esize '1' so you read bytes**
  - **buffer must be at least "esize*elem" bytes**
  - **Reads "lines" in ASCII mode (up to /n)**

- **E.g.: read up to 255 bytes from handle and put it in buffer**

```
num = fread(buff, 1, 255, handle);
```

# fread()- text mode

```
nRead = fread(void *buffer, int esize, int elem, FILE *handle);
```

- Reads "up to a line" of data from the file
  - If nRead is less than your buffer size, you know you got all the data
  - If nRead equals your buffer size then you don't know.
    - Always make your text read buffers a little larger then you need.

# Manage Files with `fseek()`

- **Used to "move around" in a file**

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

- **Whence values:**
  - **SEEK_SET** **Beginning file**
  - **SEEK_CUR** **Current position in file**
  - **SEEK_END** **End of file**

- **E.g.: re-start at the beginning of a file**

  **rc = fseek(handle, 0, SEEK_SET);**

# Example: Reading from File

```
/* Example – ex_fileio.c */
#include <stdio.h>  /* for fopen and fprintf */
#include <stdlib.h> /* for exit() */
   ...
FILE *ifp;                    /* input file pointer */
char fname[256]="in_file“; /*double quotes 4 strings*/
ifp = fopen(fname, "r");
if (NULL == ifp) { /* could not open file */
   fprintf(stderr,"Can't open file in_file!\n");
   exit(1);
}
...
while (!feof(ifp)) { /* read until end-of-file */
...
}
fclose(ifp);       /* close it when you are done */
```

# File status with `stat()`

- **Used to "see stuff" about a file**

```
int = stat(const char *filename, struct stat);
```

- **Returns "0" for success or specific errors**
  - **See Google for the details**

- **E.g.: (quick check to see if a file exists)**

```
struct stat file_status;
if (stat("file.txt", &file_status) != 0) {
    fprintf (stderr,"File not found\n",); }
```

# C Functions & the compiler model

Reminder: General form of all C functions:

<retType> FunName(<parm1>, (<parm2>, …);

RetType - The type of data the function will return
e.g. int, float, void, int *, etc

FunName - The name of the function
e.g. sin, log, etc

Parm1 - Parameters (variables) passed into the function. Zero to N parameters are allowed.
Normally a "fixed" list with a symbolic variable names
e.g. int angle, int *angle, float radian

# Important Facts

- Processors have a one or more general purpose registers
  - Registers are a fixed size (length)
  - Registers are 10X or more faster than RAM
- The general programmer model assumes variables are kept in RAM
  - But the compiler tries to keep copies of variables in registers for performance.
- C was developed when RAM was slow and cache memory was rare.
  - This affected the way "C" **sees** the world ☺

# C compiler model

- "C" wants to use registers to pass variables and use registers to return values

  <retType> FunName(<parm1>, (<parm2>, …);

  retType, ParmX - Want to be fixed length registers

- In most "C" implementations the first few processor registers are used to pass the first few parameters
  - Additional parameters are put on the stack

- In most "C" implementations the return value is put in one or two registers

# C compiler model

- Most “C” native types are short and match the registers size of the machine
  - Char - smaller than most registers
  - Int/long - 1-2x the register size
  - Float/double - 2-4x the register size

- A well written C compiler SHOULD support other sizes
  - History has shown this is often not true

# “C” Compiler Upshot

- Don’t pass things in to or out of a C function if it is more than 2-4x the register size
  - Don’t pass long things as parameters
  - Don’t return long things from functions
  - Use pointers

- Don’t assume de-referencing pointers to long objects will product good data
  - E.g.  longObj = *longObj_p
  - Use Memcpy() instead

# Short Data Example

```
/* Good example */
int x, y;
int *x_p;

x_p = &x;          /* Get the pointer to x */
y = *x_p           /* C will use the pointer  to
                    access x and then copy the 4
                    bytes into y, works great! */
```

# Long Data Example 1

```
struct longStruct{ int x; int y; int z; } ;

struct longStruct  x, y;
Struct longStrucct *x_p;

x_p = &x;          /* Get the pointer to x */
y = *x_p           /* C should use the pointer  to
                    access x and then copy the 12
                    bytes into y, it might work */
```

# Long Data Example 2

```
struct longStruct{ int x; int y; int z; } ;

struct longStruct  x, y;
struct longStrucct *x_p;

x_p = &x;          /* Get the pointer to x */
/* always works */
memcpy(&y, x_p, sizeof(longStruct));
```

# Exercise 1

- We are running on a 32 bit machine and given:

```
int x [100];
int y = 100;
int z [y];
```

What are the sizes of x, y & z?

- 32 bits = 4 bytes. The dimension "100" is known at compile time, so x is 100*4 => 400 bytes
- y is 4 bytes
- The dimension "y" is not known at compile time, so z is 4 bytes.

# Exercise 2

- What is wrong with the following code?

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

int main(int argc, char *argv) {
   FILE *fp;                    /* input file pointer */
   fp = fopen(“in_file.txt”, "r");
   while (!feof(fp)) {
      /* read until end-of-file */
   }
Return(0);
}
```

  - No checking after the fopen
  - Did not fclose(fp), this will cause a memory leak

# Exercise 3

- What is wrong with the following fragment?

```
char buff [255];
num = fread(buff, 1, 255, handle);
if (num < sizeof(buff)) { printf("Read all");}
else                    { printf ("Didn't read all";}
```

- sizeof(buff) will ALWAYS RETURN 4 (or 8)
  - you needed to code:

```
#define MAXBUFF (255)
char buff [MAXBUFF];
num = fread(buff, 1, MAXBUFF, handle);
if (num < MAXBUFF) { printf("Read all");}
else               { printf ("Didn't read all";}
```

# Exercise 4

- Given: int main(int argc, char *argv[])
  myprog one 2
- Code;

```
progNameP = argv[0];
cmdStrP   = argv[1];
myInt     = atoi(argv[2]);
myFloat   = atof(argv[3]);
```

- What will happen?
  - Code lines 1-3 will do what you expect
  - **Line 4 will SEGFAULT and CRASH**
  - argv is 3 – there are only 2 passed parameters and arg[0]

- **You MUST verify the argv index with argc BEFORE using it.**

# Exercise 5

- You want to read raw (binary) data from a 255 pixel wide sensor.
  - Write the fopen()/fread() code fragment?

```
#define MAXBUFF (255)      /* Never hardcode buffers */
FILE *fp;                  /* input file pointer */
char buff [MAXBUFF];

fp = fopen("in_file.txt", "rb");   /* rb is KEY */
if (NULL == fp) {printf("open error); exit(99);}

while (!feof(fp)) {        /* read until end-of-file */
  num = fread(buff, 1, MAXBUFF, fp);
  if (num != MAXBUFF) { printf("len error"); exit(98);}
}
fclose(fp);
```

# Appendix

# Ex_fileio.c

```
/*************************************************************
 * fileio - Illustrates the use of files, detection of EOF,
 *          values returned by fscanf and printing to stderr
 *
 * compile with: gcc -Wall -ansi -pedantic ex_fileio.c -o fileio
 * run with:      ./fileio 2&>1
 *
 * Author: Dr. Juan C. Cockburn (jcck@ieee.org)
 * Revised: 2/1/2014 JCCK
 *************************************************************/

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

int main(){
  /* declare local variables */
  char name[256];     /* string to hold name */
  unsigned int id;    /* id number */
  FILE *ifp;          /* input file pointer */
  int ok;             /* successfull conversions by scanf */
  int lcv=1;          /* line counter variable */
  char fname[20]="in_file"; /* file name */
  ifp = fopen(fname, "r");
  if (NULL == ifp) { /* always check */
    fprintf(stderr,"Can't open file in_file!\n");
    exit(1);
  }
```

```
/* Read file and print contents to stdout */
  while (!feof(ifp)) { /* read until end-of-file */
    ok=fscanf(ifp,"%s %u", name,&id);
    fprintf(stderr,"successfull conversions in line %d:%d\n",lcv,ok);
    if (2 != ok) {
      if (ok != -1) { /* returned by EOF */
        fprintf(stderr,
          "error in %s, line %d, expecting \"name id\"\n",fname,lcv);
        }
      break;
    }
    printf("%d, %s %d\n", lcv++, name, id);
  }
  /* close file */
  fflush(ifp);
  fclose(ifp);
  return 0;
}
```