Phys4051: Last C-Lecture

- Debugging:
  - Preprocessors: #if, #else, #endif
  - LabWindows: Breakpoints and Variables
- Output: printf()
- Input: scanf() and gets()
- File I/O
  - ASCII vs. Binary
  - Open, Close, Read, Write Data
- Summary & Conclusions

Maximum Function Version 1

- Write a function that returns the largest value stored in an array with \( n \) elements.
- The array is type “double.”
- The largest value is returned by the function itself.
- NO GLOBAL VARIABLES!

Solution MaxFunction V1

double MyMaxVal( double* a, int n)
{
  int i;
  double dmax = a[0];
  for( i = 1; i < n; i++)
    if (a[i] > dmax)
      dmax = a[i];
  return dmax;
}

Maximum Function Version 2

- Write a function that returns the largest value in an array with \( n \) elements.
- The array is type “double.”
- The largest value is returned through a function argument by reference!
- NO GLOBAL VARIABLES!

Solution MaxFunction V2

void MyMaxRef( double* a, int n, double* mx)
{
  int i;
  double dmax = a[0];
  for( i = 1; i < n; i++)
    if (a[i] > dmax)
      dmax = a[i];
  *mx = dmax;
}

Maximum Function

- Call and use (for example, print) the value returned by the previous two functions
Solution

```c
#include <stdio.h>

double MyMaxVal(double* a, int n); 
void MyMaxRef(double* a, int n, double* mx);

main()
{
    int i;
    static double m, r[MAX]; //assign values to r (not shown)
    printf("%E\n", MyMaxVal(r, MAX));
    MyMaxRef(r, MAX, &m);
    printf("%E\n", m);
}
```

Final F02 (1/3)

4) (10 Points)

a) Write a stand-alone ANSI-C function that returns the average value for an array of type double with n elements.

Final F02 (3/3)

* Continue this way until the first \( m - 100 \) elements have been averaged. (Note: \( m-100 \) is used to avoid exceeding the limits of the array.)
* Finally, the function must return how many sets of 100 elements were averaged.

The array is of type double and all averaging calculations must be carried out by calling the function you wrote for part a!

Final F02 (2/3)

b) Write another stand-alone ANSI-C function that compacts a large array with m elements by a factor of 100 by averaging “chunks” of 100 array elements together in the following manner:

Take the first 100 elements of the array, calculate the corresponding average and then store this result in the very first element of the original array.

Take the second set of 100 elements, i.e., array elements 100 to 199, average them and store them at the second position of the original array.

Solution

```c
int AveChunks1(double *a, int n, int m)
{
    int i, count = 0;
    for(i = 0; i < n-m; i += m)
    {
        a[count] = MyAve(a+i, m);
        count++
    }
    return count;
}
```

Bug: Definition (from WIRED, 7.02 Feb. 99 p.60)

*Jargon Watch: Issue*

Microspeak for “technical problem.” Allegedly, Microsoft employees are instructed not to use the word *bug*. Acceptable terms also include “known issues,” “intermittent issues,” “design side effects,” and “undocumented behaviors.”
Preprocessors: General (1)
*x* Preprocessors are executed before the program is compiled.
*x* They specify how to the program should be compiled.
*x* They can not be changed during the program execution.
*x* All preprocessors start with the pound sign (#) and do not have a semicolon at the end!

Preprocessors: General (2)
*x* Applications:
- Write programs for different operating systems
- Debugging!
*x* Preprocessor Examples:
- `#include`
- `#define`
- `#if, #elif, #else, #endif`

Preprocessors: #if, #elif, #else, #endif Syntax
```
#if CONSTANT1
  statements
#elif CONSTANT2
  statements
#elif CONSTANT3
  statements
#else
  statements
#endif
```

Example 1
```
#define LABWINDOWS 0
#if LABWINDOWS
  #include <ansi_c.h>
#else
  #include <stdlib.h>
  #include <stdio.h>
  #include <math.h>
#endif
```

Example 2
```
#define TESTING 0
void MyFunction( void );
#if TESTING
main(){
    MyFunction();
}
#endif
void MyFunction( void )
    { .......... // more statements...}
```

Preprocessors: #if vs. (regular) if
```
#if CONSTANT
  logic condition must be a constant
#else
  logic condition never changes during program execution
#endif
```
```
if(logic condition)
  logic condition can be a variable, an expression or (rarely) a constant
  logic condition usually changes during program execution
  logic condition is checked each time during program execution
```
Debugging in LabWindows

- Setting breakpoints and stepping through the code
- Viewing the variables

Debugging: Setting Breakpoints

Note the red diamond in the left column indicates a breakpoint.

Debugging: Stepping Through the Program

- Go (continue)
- Go to cursor
- Step Into
  - Proceed statement by statement
- Step Over
  - Execute the function but proceed to the next statement
- Finish Function

Debugging: Displaying the Program Variables (2)

Open the Variables window to display them.

Debugging: Displaying the Program Variables (3)

- The “Variables-Window” displays all variables and their current value.

Debugging: Displaying the Program Variables (4)

- Short Cut: View Variable Value
  - 1) Highlight the variable in your C code
  - 2) Select the View Variable Value button
  - 3) The variable window will be displayed
### C-Output Functions

**printf, fprintf, sprintf**

- **printf( )**
  - Prints to "stdout," the standard output stream, which is the monitor.
- **fprintf( )**
  - Same as printf( ), except you can specify the output stream as being: the monitor, a file, the printer or even a modem.
- **sprintf( )**
  - Same as printf( ), except it prints to a string.

### C-Output Functions: printf() Text Only

- **Program Segment:**
  ```c
  printf("What is");
  printf("next");
  printf("?");
  ```
- **Resulting Output:**
  ```
  What isnext?
  ```

### printf(): Text with Escape Sequence

- **Program Segment:**
  ```c
  printf("What \n\t next \n ?");
  ```
- **Resulting Output:**
  ```
  What  
  next  
  ?
  ```

### printf(): Escape Sequence Characters

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>Alert, Bell</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal Tab</td>
</tr>
<tr>
<td>\n</td>
<td>Newline</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage Return</td>
</tr>
</tbody>
</table>

### printf(): Text and Variables

- **Program Segment:**
  ```c
  printf("ST1 %TS ST2", var_name);
  ```
- **Where:**
  - \*ST1: Optional String1
  - \*TS: Type Specifier (required)
  - \*ST1: Optional String2
  - \*var_name: The name of variable
- **Example:**
  ```c
  printf("y in Hex is: %X\n", y);
  ```
- **Output:**
  ```
  y in Hex is: F1AB
  ```
printf(): Type Specifiers

- Numerical Variables:
  - i (signed) int, short, long
  - u unsigned int short, long
  - f float, double (decimal notation)
  - e float, double (scientific notation)

- Characters, Strings and Others:
  - c character
  - s character array (string)
  - x hex notation (any variable)
  - p pointer

printf(): Example 1: Multiple Variables

Program Segment:
```
int sx;
sx = -200.0 / 3;
printf("double: %e int: %i ",
       (double) sx, sx);
printf("char: %c Hex: %X Error: %u",
       (char) abs(sx), sx, sx);
```
Output:
double: -6.600000e+01 int: -66 char: B Hex: FFFFFFBE Error: 4294967230

printf(): Specifying the Precision

- The *printf* precision of a numerical variable can be specified like this:
  ```
  printf("%14.8f", val);
  ```
  where:
  - the number right after the percentage sign indicates the total number of digits (or blank spaces) to be printed,
  - and the number before the type specifier indicates how many digits after the decimal point will be printed.

printf(): Specifying the Precision: Example

<table>
<thead>
<tr>
<th>Format String</th>
<th>Resulting Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><em><strong>%12.14f</strong></em></td>
<td><em><strong>-66.66666666666667</strong></em></td>
</tr>
<tr>
<td><em><strong>%12.13f</strong></em></td>
<td><em><strong>-66.66666666666667</strong></em></td>
</tr>
<tr>
<td><em><strong>%12.12f</strong></em></td>
<td><em><strong>-66.66666666666667</strong></em></td>
</tr>
<tr>
<td><em><strong>%12.11f</strong></em></td>
<td><em><strong>-66.66666666666667</strong></em></td>
</tr>
<tr>
<td><em><strong>%12.10f</strong></em></td>
<td><em><strong>-66.66666666666667</strong></em></td>
</tr>
<tr>
<td><em><strong>%12.9f</strong></em></td>
<td><em><strong>-66.66666666666667</strong></em></td>
</tr>
<tr>
<td><em><strong>%12.8f</strong></em></td>
<td><em><strong>-66.66666666666667</strong></em></td>
</tr>
<tr>
<td><em><strong>%12.7f</strong></em></td>
<td><em><strong>-66.6666666666667</strong></em></td>
</tr>
<tr>
<td><em><strong>%12.6f</strong></em></td>
<td><em><strong>-66.6666666666667</strong></em></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><em><strong>%12.1f</strong></em></td>
<td><em><strong>-66.7</strong></em></td>
</tr>
<tr>
<td><em><strong>%12.0f</strong></em></td>
<td><em><strong>-67</strong></em></td>
</tr>
</tbody>
</table>

printf(): Strings

- To print a string, provide a pointer to the "char" array that contains the string

Program Segment:
```
char course[] = "Phys5122";
printf("%s", course + 4);
```
Output:
```
???
```

sprintf()

- Instead of printing to the screen, sprintf() will "print" its output to a string
- Except for the first argument, which is a pointer to the char array for the output string, sprintf() is identical to printf().

Example:
```
char varray[ 256 ];
sprintf( varray, "%Voltage: \%E\n", v1);
```
**sprintf(): LabWindows Example Callback Function**

```c
CVICALLBACK ToTextbox (int panel, int control, int event, void *callbackData, int eventData1, int eventData2)
{
    switch (event)
    {
        static char sText[200];
        case EVENT_COMMIT:
            sprintf(sText,"Addr. of 'sText' is: 0x%p
            SetCtrlVal (panelHandle, PANEL_TEXTBOX, sText);
            break;
        } return 0;
    }
}
```

**Output:**
Note the textbox control with its sliders.

---

**C-Input: scanf(), fscanf()**

- To read a (single) numerical value use:
  - `scanf()` for the default input, i.e., keyboard,
  - `fscanf()` for file input
- Usage: very similar to `printf()` function
- Example:
  ```c
  int x;
  printf("Enter a value: ");
  fflush( stdin ); //bug in LW compiler
  scanf("%d", &x);
  printf("\nValue read is: %d", x);
  ```

---

**C-Input: gets(), fgets()**

- To read in an entire character string from the keyboard use:
  - `gets()`
- Example:
  ```c
  char name[ 200 ];
  printf("Enter your name: ");
  gets( name );
  printf("\nName entered:%s\n", name);
  ```

---

**Data Files: ASCII vs. Binary Storage**

- ASCII vs. Binary Storage
  - not much difference for text
  - very different for numbers because it depends on the method used to encode the data
- Binary files are usually "width" delimited
- ASCII files are usually character delimited using commas, tabs, blanks, etc.
Data Files: Opening & Closing

Three steps for accessing data files:
1. Data File is Opened
2. Data is Read / Written to the File
3. Data File is Closed

Step 1 and 3 are identical for ASCII and Binary files.

Opening a Data File: fopen() (1)

1: Declare a file stream pointer
2: Call the fopen() function. The function returns the address for the file stream pointer

Example:
```c
FILE *fptr; //declare a file ptr
fptr = fopen("Data1.txt","wt");
```

Opening a Data File: fopen() (2)

fopen( arg1, arg2)
- arg1: a pointer to a string containing the filename or a literal, for example: "c:\myfolder\data.txt"
- arg2: activity and filetype constants:
  - wt Open an ASCII file for writing
  - rt Open an ASCII file for reading
  - at Open an ASCII file for appending
  - wb Open a Binary file for writing
  - rb Open a Binary file for reading
  - ab Open a Binary file for appending

Closing a Data File: fclose()

Syntax:
```c
fclose( fptr );
```

where fptr refers to the file stream pointer returned by the fopen() function.

Writing to and Reading from an ASCII Data File

Writing: fprintf( )

fprintf( ) Syntax:
```c
fprintf( arg1, arg2... )
```
- arg1: the file stream pointer
- arg2... are identical to arguments of the printf() function

Example:
```c
fprintf(fptr, "%d\n", x);
```

Reading: fscanf( )

fscanf( ) Syntax:
```c
fscanf( arg1, arg2... )
```
- arg1 is the file stream pointer
- arg2... are identical to arguments of the scanf() function

Example:
```c
fscanf(fptr, "%d", &x);
```

Writing to and Reading from a Binary Data File

Writing: fwrite( )

Reading: fread( )

Syntax: (identical for fwrite and fread):
```c
fwrite( arg1, arg2, arg3, arg4 );
```
- arg1: a pointer to the data buffer
- arg2: the size of each data element (in bytes)
- arg3: the number of data elements
- arg4: the file stream pointer