Phys4051: C-Lecture 7 / 7

- 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

Debugging

Preprocessors
LabWindows Debug
Utilities: Trace & Variable View

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)

- Preprocessors are executed before the program is compiled.
- They specify how the program should be compiled.
- They cannot be changed during the program execution.
- All preprocessors start with the pound sign (#) and do not have a semicolon at the end!

Preprocessors: General (2)

- Applications:
  - Write programs for different operating systems
  - Debugging!
- 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
```
**Preprocessors: #if, #elif, #else, #endif Example 1**

```c
#define LABWINDOWS 0

#if LABWINDOWS
    #include <ansi_c.h>
#else
    #include <stdlib.h>
    #include <stdio.h>
    #include <math.h>
#endif
```

**Preprocessors: #if, #elif, #else, #endif Example 2**

```c
#define TEST 1
#if TEST
    #define ASIZE 10
#else
    #define ASIZE 100000
#endif

double VoltIn[ ASIZE ];
```

**Preprocessors: #if, #elif, #else, #endif Example 3**

```c
#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
  - logic condition never changes during program execution
  - logic condition is checked only during program compilation

- 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( )` is the same as `printf( )`, except you can specify the output stream as being: the monitor, a file, the printer or even a modem.
  - `sprintf( )` is the same as `printf( )`, except it prints to a string.
C-Output Functions: printf() Text Only

*Program Segment:
printf("What is");
printf("next");
printf("?")

*Resulting Output:
What isnext?

printf(): Text with Escape Sequence

*Program Segment:
printf("What\n\tnext\n?" );

*Resulting Output:
What
tnext
?

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

```c
printf("ST1 %TS ST2", var_name);
```

Where:
- **ST1**: Optional String1
- **TS**: Type Specifier (required)
- **ST2**: Optional String2
- **var_name**: The name of variable

Example:
```c
printf("y in Hex is: %X\n", y);
```

Output:
```c
y in Hex is: F1AB
```

printf(): Type Specifiers

Numerical Variables:
- `d` (signed) short
- `u` unsigned short
- `ld` (signed) long
- `lu` unsigned long
- `i` signed int
- `f` float (decimal notation)
- `e` float (scientific notation)
- `lf` double (decimal notation)
- `le` 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:
```c
int sx;
sx = -200.0 /3;
printf("double: %le int: %d ",
      (double) sx, sx);
printf("char: %c Hex: %X Error: %u",
      (char) abs(sx), sx, sx);
```

Output:
```c
double: -6.600000e+01 int: -66 char: B Hex: FFFFFFFBE Error: 4294967230
```
printf(): Specifying the Precision

The *printf* precision of a numerical variable can be specified like this:

```c
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>-66.6666666666667</strong></em></td>
</tr>
<tr>
<td><em><strong>%12.12f</strong></em></td>
<td><em><strong>-66.666666666667</strong></em></td>
</tr>
<tr>
<td><em><strong>%12.11f</strong></em></td>
<td><em><strong>-66.66666666667</strong></em></td>
</tr>
<tr>
<td><em><strong>%12.10f</strong></em></td>
<td><em><strong>-66.6666666667</strong></em></td>
</tr>
<tr>
<td><em><strong>%12.9f</strong></em></td>
<td><em><strong>-66.666666667</strong></em></td>
</tr>
<tr>
<td><em><strong>%12.8f</strong></em></td>
<td><em><strong>-66.66666667</strong></em></td>
</tr>
<tr>
<td><em><strong>%12.7f</strong></em></td>
<td><em><strong>-66.6666667</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:

```c
char course[] = "Phys5122";
printf("%s", course + 4);
```

Output:

```c
???
```
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:
  ```c
  char varray[256];
  printf(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\n", sText);
        SetCtrlVal(panelHandle, PANEL_TEXTBOX, sText);
        break;
    }
    return 0;
}
```

sprintf(): LabWindows Example Callback Function

* 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: ");
  scanf("%i", &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 File I/O

ASCII Files vs. Binary Files
File Open and Close
Read and Write
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: ASCII Data Storage

- Store 12345, 128, 12346... in ASCII format:
  - 1 = 49 = 0x31 (in ASCII)
  - 2 = 50 = 0x32 (in ASCII) etc.
  - , = 44 = 0x2C (in ASCII)

Data Files: Binary Data Storage

- Store 12345, 128, 12346... in Binary format:
  - 12345 = 0x3039
  - 128 = 0x0080
  - 12346 = 0x303A
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:
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...)
```

where `arg1` is the file stream pointer and `arg2`... are identical to arguments of the `printf()` function.

**Example:**

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

Reading: fscanf()

**fscanf( ) Syntax:**

```c
fscanf( arg1, arg2...)
```

where `arg1` is the file stream pointer and `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()

**fwrite( ) Syntax:**

```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
Reading From a Binary Data File: Program Segment

```c
#include MAX 100
FILE *fin;
double datain[MAX];
fin = fopen("DaData.dat", "rb");
fread( datain, sizeof(double),
      MAX, fin );
fclose( fin );
```

Conclusions

**Questions?**

- Java and C++
- Object Oriented Programming (OOPs)
  - Methods (Similar to Functions)
  - Events
  - Properties (Similar to Variables)
  - Objects and Classes