Lecture #2: More C Program Basics

Reading: Holmes, Chapters 1–3, 6–8 (continued from Lecture #1); also Chapter 5
Handouts: More Sample Basic C Programs; Operator Precedence and Associativity

1. Greetings and felicitations
2. Read a line
   - show why it skips input newline
   - briefly touch on array declaration
3. Pointers: quick look
   - what a pointer is
   - walk through its use here
4. Function parameters
   - call by value vs. call by reference (C is ALWAYS the former)
   - how to fake the former
   - walk through this one, with pictures
5. Passing arrays and pointers
   - mention function prototypes (types of args critical)
   - walk through ptogram, showing what happens

More Sample Basic C Programs

This program reads an input line, and retains the newline; normally you would have a function to do this (there are some in the Standard I/O Library, for instance) but it does show how to use character arrays and pointers. This version uses arrays.

```
#include <stdio.h>

#define     BUFSIZE     1024  /* maximum input line length */

/*
 * echo the input back to the output, but print the number
 * of characters read at the beginning of the line
 */
void main(void)
{
      char line[1024];  /* input buffer */
      int c;                  /* input character */
      int i = 0;        /* counter in while loop */

      do{
            /*
             * load up to max-1 chars into the buffer buf
             */
            i = 0;
            while(i < BUFSIZE-1 && (c = getchar()) != EOF &&
                                                      c != '\n')
                  line[i++] = c;
            /*
             * end the string with a newline and return success
             */
            if (c == '\n')
                  line[i++] = c;
            /* tack on the string terminator */
            line[i] = '\0';

            /*
             * print it out
             */
            if (i != 0)
                  printf("%3d %s", i, line);

      } while (c != EOF);

      /*
       * say goodbye!
       */
      exit(0);
}
```

This is exactly the same program, but uses a pointer to put elements into the buffer:

```
#include <stdio.h>

#define     BUFSIZE     1024  /* maximum input line length */

/*
 * echo the input back to the output, but print the number
 * of characters read at the beginning of the line
 * this uses pointers
 */
void main(void)
{
      char line[BUFSIZE];     /* input buffer */
      char *l;                /* pointer into input buffer */
      int c;                  /* input character */

      do{
            /*
             * load up to BUFSIZE-1 chars into the buffer
             */
            l = &line[0];
            while(l < &line[BUFSIZE-1] &&
                        (c = getchar()) != EOF &&
                                          c != '\n')
                  *l++ = c;
            /*
             * end the string with a newline and return success
             */
            if (c == '\n')
                  *l++ = c;
            /* tack on the string terminator */
            *l = '\0';

            /*
             * print it out
             */
            if (l != line)
                  printf("%3d %s", l - line, line);

      } while (c != EOF);

      /*
       * say goodbye
       */
      exit(0);
}
```

This shows how C does call by reference (the same as a PASCAL **var** parameter):

```
#include <stdio.h>

/*
 * swap two elements (no pointers)
 */
void npswap(int x, int y)
{
      int temp;   /* temporary */

      /*
       * show what they are on entry
       */
      printf("begin npswap: elem1 is %d, elem2 is %d\n", x, y);

      /*
       * do the swap
       */
      temp = x;
      x = y;
      y = temp;

      /*
       * show they were exchanged
       */
      printf("  end npswap: elem1 is %d, elem2 is %d\n", x, y);
}

/*
 * swap two elements (pointers)
 */
void pswap(int *x, int *y)
{
      int temp;   /* temporary */

      /*
       * show what they are on entry
       */
      printf("begin  pswap: elem1 is %d, elem2 is %d\n", *x, *y);

      /*
       * do the swap
       */
      temp = *x;
      *x = *y;
      *y = temp;

      /*
       * show they were exchanged
       */
      printf("  end  pswap: elem1 is %d, elem2 is %d\n", *x, *y);
}
```

```
/*
 * this is to show that everything is call by value
 * so you need to simulate call by reference
 * it uses swapping
 */
void main(void)
{
      int elem1 = 4;          /* first element */
      int elem2 = 5;          /* second element */

      /*
       * say what you're starting with
       */
      printf("     initial: elem1 is %d, elem2 is %d\n",
                                                elem1, elem2);

      /*
       * use the non-pointer swap and print the results
       */
      npswap(elem1, elem2);
      printf("after npswap: elem1 is %d, elem2 is %d\n",
                                                elem1, elem2);

      /*
       * use the pointer swap and print the results
       */
      pswap(&elem1, &elem2);
      printf("after  pswap: elem1 is %d, elem2 is %d\n",
                                                elem1, elem2);

      /*
       * say goodbye
       */
      exit(0);
}
```

How do you pass arrays, and how do you reference them? This little goodie shows that arrays and pointers can be passed indiscriminately, so long as the prototypes and definitions are consistent.

```
#include <stdio.h>

#define MAXLINE   1000  /* maximum allowed line length */

/*
 * prototypes -- all are different functions that do the same
 * thing: copy a string from old to new
 */
void str1copy(char new[], char old[]);
void str2copy(char new[], char old[]);
void str3copy(char new[], char old[]);
void str4copy(char *new, char *old);

/*
 * echo a line from the input to the output
 */
void main(void)
{
      char buf[MAXLINE];      /* input buffer */
      char new[MAXLINE];      /* output buffer */

      /*
       * read a line, then write it out
       */
      while(printf("type away!> "),
                        fgets(buf, MAXLINE, stdin) != NULL){
            /* #1 to: copy it to the new buffer and print it */
            str1copy(new, buf);
            printf("1> ");
            fputs(new, stdout);
            /* #2: copy it to the new buffer and print it */
            str2copy(new, buf);
            printf("2> ");
            fputs(new, stdout);
            /* #3: copy it to the new buffer and print it */
            str3copy(new, buf);
            printf("3> ");
            fputs(new, stdout);
            /* #4: copy it to the new buffer and print it */
            str4copy(new, buf);
            printf("4> ");
            fputs(new, stdout);
      }

      /*
       * say goodbye!
       */
      exit(0);
}
```

```
/*
 * make a copy of a string
 *
 * arguments:     char old[]  original string
 *                char new[]  buffer for copy
 *
 * returns: noting, but on return new contains a copy of old
 *
 * exceptions:    none, but length of old and new are not checked
 */
void str1copy(char new[], char old[])
{
      int i;            /* counter in a for loop */

      /*
       * do it char by char
       */
      for(i = 0; old[i] != '\0'; i++)
            new[i] = old[i];
      /* tack on the terminator */
      new[i] = '\0';
}

/*
 * make a copy of a string
 *
 * arguments:     char old[]  original string
 *                char new[]  buffer for copy
 *
 * returns: noting, but on return new contains a copy of old
 *
 * exceptions:    none, but length of old and new are not checked
 */
void str2copy(char new[], char old[])
{
      int i;            /* counter in a for loop */

      /*
       * do it char by char
       */
      for(i = 0; old[i]; i++)
            new[i] = old[i];
      /* tack on the terminator */
      new[i] = '\0';
}
```

```
/*
 * make a copy of a string
 *
 * arguments:     char old[]  original string
 *                char new[]  buffer for copy
 *
 * returns: noting, but on return new contains a copy of old
 *
 * exceptions:    none, but length of old and new are not checked
 */
void str3copy(char new[], char old[])
{
      int i;            /* counter in a for loop */

      /*
       * do it char by char
       */
      for(i = 0; new[i] = old[i]; i++)
            ;
}

/*
 * make a copy of a string
 *
 * arguments:     char *old   pointer to original string
 *                char *new   pointer to buffer for copy
 *
 * returns: noting, but on return new contains a copy of old
 *
 * exceptions:    none, but length of old and new are not checked
 */
void str4copy(char *new, char *old)
{
      /*
       * do it char by char
       */
      while(*new++ = *old++)
            ;
}
```

This program , split over three pages, shows how to open, close, and read from a file using the Standard I/O Library; it also shows how to use static variables to keep information across function calls.

```
#include <stdio.h>

#define MAXFILENAMESIZE 1024  /* max length of file name */
#define IN_WORD   1     /* currently inside a word */
#define NOTIN_WORD      0     /* currently not in a word */
```

```
/*
 * count the number of lines, words, and chars in the input
 * a word is a maximal sequence of nonspace characters, so
 * the quote "+++ --- hi bye 879+3" has 5 words ("+++", "---",
 * "hi", "bye", and "879+3")
 */
void wc(char *filename, FILE *fp)
{
      register int c;               /* input char */
      static int totnl = 0;   /* total line count */
      static int totnw = 0;   /* total word count */
      static int totnc = 0;   /* total char count */
      register int nl, nw, nc;      /* line, word, char count */
      register int state;           /* in or not in a word? */

      /*
       * initialize
       */
      nl = nw = nc = 0;
      state = NOTIN_WORD;

      /*
       * handle input a char at a time
       */
      while((c = getc(fp)) != EOF){
            /* got another character */
            nc++;
            /* is it a newline? */
            if (c == '\n')
                  nl++;
            /* is it a word separator? */
            if (c == ' ' || c == '\t' || c == '\n')
                  /* YES -- change state */
                  state = NOTIN_WORD;
            else if (state == NOTIN_WORD){
                  /* NO -- we're now in a word; update */
                  /* the counter and state if need be  */
                  state = IN_WORD;
                  nw++;
            }
      }

      /*
       * increment the cumulative counts
       */
      totnl += nl; totnw += nw; totnc += nc;

      /*
       * announce the results and quit
       */
      printf("%6d\t%6d\t%6d\t%s\n", nl, nw, nc, filename);
      printf("%6d\t%6d\t%6d\tcumulative\n", totnl, totnw, totnc);
}
```

```
/*
 * read a file name from the standard input,
 * open it, and count characters, words, and lines
 */
void main(void)
{
      char fname[MAXFILENAMESIZE];  /* the file name(s) */
      FILE *fp;               /* file pointer */

      /*
       * read one file name per line, and do the dirty deeds
       * then get the next one, …, until EOF
       */
      while(printf("file name> "),
                  fgets(fname, MAXFILENAMESIZE, stdin) != NULL){
            /* open the file */
            if ((fp = fopen(fname, "r")) == NULL){
                  fprintf(stderr, "Could not open file %s\n", fname);
                  continue;
            }
            /* count the characters, etc. */
            wc(fname, fp);
            /* now close it */
            (void) fclose(fp);
      }

      /*
       * say goodbye!
       */
      exit(0);
}
```

Operator Precedence and Associativity

This table, or one similar to it, can be found in almost every book on C. It's worth keeping this with you as you program, because some of the precedences are not what you might expect:

precedence	***class***	***operators***	***associativity***
highest	primary	`()  []  ->  .`	left to right

	unary	`~  ++  --  !`
		`(`*`type`*`)  sizeof`
		`&` (address)
		`*` (pointer)
		`+` (positive sign)
		`-` (negative sign)
			right to left

	multiplicative	`*  /  %`	left to right
	additive	`+  -`	left to right
	shift	`<<  >>`	left to right
	relational	`<  <=  =>  >`	left to right
	equality	`==  !=`	left to right
	bitwise and	`&`	left to right
	bitwise exclusive or	`^`	left to right
	bitwise or	`|`	left to right
	logical and	`&&`	left to right
	logical or	`||`	left to right
	conditional	`? :`	right to left

	assignment	`=  +=  -=`
		`*=  /=  %=`
		`&=  ^=  |=`
		`<<=  >>=`
			right to left

lowest	comma	`,`	left to right