4010-350 Personal SE

Functions, Arrays, Strings and Files

Functions in C

- Syntax like Java methods but w/o public, abstract, etc.
- As in Java, all arguments (well, **most** arguments) are passed by *value*.
- Example:

```
void try_swap( int x, int y ) {
    int t = x ;
    x = y ;
    y = t ;
}
```

- Doesn't work:
  - `x` and `y` are copies of the arguments in the caller.
  - Changing the copy has *no effect* in the caller.

Functions in C

- Functions must be declared before use:
  - *Declare* means specify name, return value, and argument types.
- Indeed, in C *everything* must be declared before use!

Functions in C

- Functions must be declared before use:
  - *Declare* means specify name, return value, and argument types.
- Indeed, in C *everything* must be declared before use!

```
extern int min(int x, int y) ;  // Declaration of min
static int max(int x, int y) ;  // Declaration of max

int max_div_min(int x, int y) {
   return max(x, y) / min(x, y) ;
}

int min(int x, int y) {     // Definition of min
    return (x <= y) ? x : y ;
}

static int max(int x, int y) {  // Definition of max
    return (x >= y) ? x : y ;
}
```

Functions in C

extern: defined elsewhere (possibly this file).

- Functions must be declared before use:
  - *Declare* means specify name, return value, and argument types.
- Indeed, in C *everything* must be declared before use!

```
extern int min(int x, int y) ;  // Declaration of min
static int max(int x, int y) ;  // Declaration of max

int max_div_min(int x, int y) {
   return max(x, y) / min(x, y) ;
}

int min(int x, int y) {     // Definition of min
    return (x <= y) ? x : y ;
}

static int max(int x, int y) {  // Definition of max
    return (x >= y) ? x : y ;
}
```

Functions in C

static: defined and known only in this C source file.

- Functions must be declared before use:
  - *Declare* means specify name, return value, and argument types.
- Indeed, in C *everything* must be declared before use!

```
extern int min(int x, int y) ;  // Declaration of min
static int max(int x, int y) ;  // Declaration of max

int max_div_min(int x, int y) {
   return max(x, y) / min(x, y) ;
}

int min(int x, int y) {     // Definition of min
    return (x <= y) ? x : y ;
}

static int max(int x, int y) {  // Definition of max
    return (x >= y) ? x : y ;
}
```

Arrays in C

- Generic form: *type name*[*size*] ;
- Examples:

```
#define MAX_SAMPLES (100)
int samples[MAX_SAMPLES] ;
```

Arrays in C

- Generic form: *type name*[*size*] ;
- Examples:

```
#define MAX_SAMPLES (100)
int samples[MAX_SAMPLES] ;
```

Array of 100 integers.
Indices run 0 .. 99

NO SUBSCRIPT CHECKS!

NOTE THE USE OF SYMBOLIC CONSTANT!

Arrays in C

- Generic form: *type name*[*size*] ;
- Examples:

```
#define MAX_SAMPLES (100)
int samples[MAX_SAMPLES] ;
```

Simple summation of array values.

```
int sum = 0 ;
int i ;

for ( i = 0 ; i < MAXSAMPLES ; ++i ) {
    sum += samples[ i ] ;
}
```

Arrays in C

```
#define DIMENSION (50) ;
double m1[DIMENSION][DIMENSION] ;
```

Arrays in C

A matrix or a 2 dimensional array

Access by `m1[ i ][ j ]`

```
#define DIMENSION (50) ;
double m1[DIMENSION][DIMENSION] ;
```

Arrays in C

Matrix multiplication to show use of double indices.

```
#define DIMENSION (50) ;
double m1[DIMENSION][DIMENSION] ;
double m2[DIMENSION][DIMENSION] ;
double product[DIMENSION][DIMENSION] ;
```

```
int i, j, k ;

for ( i = 0 ; i < DIMENSION ; ++i ) {
    for ( j = 0 ; j < DIMENSION ; ++j ) {
        product[ i ][ j ] = 0.0 ;
        for ( k = 0 ; k < DIMENSION ; ++k ) {
            product[i][j] += m1[i][k] * m2[k][j] ;
        }
     }
}
```

Arrays in C

- Arrays are passed to functions by *reference*.

Arrays in C

- Arrays are passed to functions by *reference*.
- Changes to the array contents in the function will be visible to the caller, e.g., array copy.

Arrays in C

- Arrays are passed to functions by *reference*.
- Changes to the array contents in the function will be visible to the caller, e.g., array copy.

```
void acopy( int to[], int from[], size ) {
    int i ;

    for( i = 0 ; i < size ; i++ ) {
        to[ i ] = from[ i ] ;
    }
}
```

Arrays in C

- Arrays are passed to functions by *reference*.
- Changes to the array contents in the function will be visible to the caller, e.g., array copy.

```
void acopy( int to[], int from[], size ) {
    int i ;

    for( i = 0 ; i < size ; i++ ) {
        to[ i ] = from[ i ] ;
    }
}
```

Need not, but may, give the array size.

Arrays in C - Review

- Array size fixed at definition time.
- Good practice (that is, ***OUR*** practice) is to use symbolic constants to define array sizes.
- Array indices are integers.
- Legal indices run from 0 to ***arraysize*** - 1
- C will *not* prevent you from indexing outside the bounds of the array (no subscript checks).
- Arrays are passed by reference.

Strings in C

- A string is just an array of chars:

```
char welcome[] = "Hello" ;  // C permits this
```

- This is an array of 8-bit bytes holding ASCII characters.

Strings in C

- A string is just an array of chars:

```
char welcome[] = "Hello" ;  // C permits this
```

- This is an array of 8-bit bytes holding ASCII characters
- The array in memory looks like this:

'H'	'e'	'l'	'l'	'o'	'\0'

- Whoa! What's that *last* character????

Strings in C

- A string is just an array of chars:

```
char welcome[] = "Hello" ;  // C permits this
```

- This is an array of 8-bit bytes holding ASCII characters
- The array in memory looks like this:

'H'	'e'	'l'	'l'	'o'	'\0'

- Whoa! What's that *last* character????
- In C, proper strings *must* be terminated with a NUL (0) character.
- We *always* need an *extra byte* to hold the terminator!

Strings in C

- There are 10 (that's two) basic ways to represent strings of characters:
  - Use a designated terminator character, like C.
  - Keep a separate integer with the string length.

Strings in C

- There are 10 (that's two) basic ways to represent strings of characters:
  - Use a designated terminator character, like C.
  - Keep a separate integer with the string length.
- Second way requires at least two bytes to process reasonable sized strings.
  - Memory in 1970 cost approx. $85 / kilobyte.
  - Gasoline cost 36¢/gallon - those were the days!
  - Thus every byte counts!

Strings in C

- There are 10 (that's two) basic ways to represent strings of characters:
  - Use a designated terminator character, like C.
  - Keep a separate integer with the string length.
- Second way requires at least two bytes to process reasonable sized strings.
  - Memory in 1970 cost approx. $85 / kilobyte.
  - Gasoline cost 36¢/gallon - those were the days!
  - Thus every byte counts!
- Besides, with C, strings are no different from other arrays.

Strings in C

- There are 10 (that's two) basic ways to represent strings of characters:
  - Use a designated terminator character, like C.
  - Keep a separate integer with the string length.
- Second way requires at least two bytes to process reasonable sized strings.
  - Memory in 1970 cost approx. $85 / kilobyte.
  - Gasoline cost 36¢/gallon - those were the days!
  - Thus every byte counts!
- Besides, with C, strings are no different from other arrays.

Strings in C

- Assume we are reading and processing lines of text, where at most the first 80 characters of a line are useful
- How would we declare an array to hold the line as a string?

Strings in C

- Assume we are reading and processing lines of text, where at most the first 80 characters of a line are useful
- How would we declare an array to hold the line as a string?

```
#define MAXLINE (80)
char line[ MAXLINE + 1 ] ;    // 1 extra character for the NUL
```

Strings in C

- Assume we are reading and processing lines of text, where at most the first 80 characters of a line are useful
- How would we declare an array to hold the line as a string?

```
#define MAXLINE (80)
char line[ MAXLINE + 1 ] ;    // 1 extra character for the NUL
```

- How would we read in such a line?

Strings in C

- Assume we are reading and processing lines of text, where at most the first 80 characters of a line are useful
- How would we declare an array to hold the line as a string?

```
#define MAXLINE (80)
char line[ MAXLINE + 1 ] ;    // 1 extra character for the NUL
```

- How would we read in such a line?

```
void readline( char line[], int maxsize ) {
    int i = 0 ;
    int ch ;

    for ( ch = getchar() ; ch != '\n' && ch != EOF ; ch = getchar() ) {
        if ( i < maxsize ) {
            line[ i++ ] = ch ;
        }
    }
    line[ i ] = '\0' ;
    return ;
}
```

Strings in C

- How can we copy one string to another?
- Modify acopy to strcpy:

```
void strcpy( char sto[], char sfrom[] ) {
    int i ;

    for ( i = 0 ; sto[ i ] = sfrom[ i ] ; ++i )
        ;
}
```

Strings in C

- How can we copy one string to another?
- Modify acopy to strcpy:

```
void strcpy( char sto[], char sfrom[] ) {
    int i ;

    for ( i = 0 ; sto[ i ] = sfrom[ i ] ; ++i )
        ;
}
```

Copy the ith character.
If this was a NUL, exit the loop.

String Library

```
#include <string.h>
int strlen( char str[] ) ;
```

Note: `strlen("Hello") == 5`

```
void strcpy( char sto[], char sfrom[] ) ;
void strncpy( char sto[], char sfrom[], unsigned n );
```

Note: Copies 'n' characters to 'sto' from 'sfrom', padding with '\0' as necessary.

Note: If 'sfrom' is too long to fit in 'sto', then 'sto' will *NOT* be NUL terminated.

```
int strcmp( char str1[], char str2[] ) ;
```

Note: comparison is in dictionary order.

Note: returns -1, 0, 1 if 'str1' is less than, equal to, or greater than 'str2', respectively.

Basic File Operations in C

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

fopen - open named file & return a "handle":

```
FILE * fopen( char name[], char mode[] ) ;
```

Basic File Operations in C

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

fopen - open named file & return a "handle":

```
FILE *fopen( char name[], char mode[] ) ;
```

FILE * is a pointer to a structure.

You need not know the details to use it.

Just consider it an internal "handle" for the file.

Basic File Operations in C

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

fopen - open named file & return a "handle":

```
FILE * fopen( char name[], char mode[] ) ;
```

The name (pathname) of the file as a string.

This can be a constant string or a properly terminated character array.

Basic File Operations in C

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

fopen - open named file & return a "handle":

```
FILE * fopen( char name[], char mode[] ) ;
```

The way you want the file opened.

The two modes we may use are:

- "r" - open an existing file for reading.
- "w" - open an a file for writing - create if necessary.

Basic File Operations in C

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

fclose - close an open file:

```
int fclose( FILE *handle ) ;
```

Return value is 0 for success, EOF for any error.

We may simply ignore the return value.

Basic File Operations in C

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

`fclose` - close an open file:

```
int fclose( FILE *handle ) ;
```

Handle (from **fopen**) of the file you want to close.

Files automatically close when the program exits.

Character I/O on Files in C

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

fgetc - read a character (like getchar)

```
int fgetc( FILE *handle ) ;
```

fputc - write a character (like putchar)

```
int fputc( int ch, FILE *handle ) ;
```

fprintf - formatted output (like printf)

```
int fprintf( FILE *handle, char fmt[], ... ) ;
```

Command Line Arguments

The full declaration of main is:

```
int main( int ac, char **argv ) ;
```

Command Line Arguments

The full declaration of main is:

```
int main( int ac, char **argv ) ;
```

ac = argument count (the number of command line arguments).

ac >= 1, as the program name is the 0th argument.

Command Line Arguments

The full declaration of main is:

```
int main( int ac, char **argv ) ;
```

ac = argument count (the number of command line arguments).

Includes the program name as the 0th argument.

Example: ac == 5

gcc	-o	myprog	main.c	util.c
0	1	2	3	4

Command Line Arguments

The full declaration of main is:

```
int main( int ac, char **argv ) ;
```

`argv` = the argument vector - allows access to the arguments
it's a pointer, but don't worry - treat it like a 2D array.
`argv[ i ]` is i^{th} argument as a string (array).
`argv[ i ][ j ]` is the jth character of the ith argument.

Example – Echo Arguments

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

int main( int ac, char **argv ) {
    int i ;

    printf( "Program name = %s\n", argv[0] ) ;

    for( i = 1 ; i < ac ; ++i ) {
        printf( "argv[%d] = %s ", i, argv[i] ) ;
        printf( "and its length is %d\n", strlen( argv[i] ) ) ;
    }

    return 0 ;
}
```

Example – Copy Files

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

static void usage( char progname[] ) ;
static void copy( char source[], char dest[] ) ;

int main( int ac, char **argv ) {
    FILE *infile ;
    FILE *outfile ;

    if ( ac != 3 ) {
        usage( argv[0] ) ;
        exit(1) ;
    }

    copy( argv[1], argv[2] ) ;
    return 0 ;
}
```

Example – Copy Files

```
static void usage( char progname[] ) {
    printf( "Usage: %s in_file out_file\n", progname ) ;
}

static void copy( char source[], char dest[] ) {
    FILE *inf =  fopen( source, "r" ) ;
    FILE *outf = fopen( dest, "w" ) ;

    if ( ! inf || ! outf ) {  // 0 => bad handle => open error
        exit(1) ;
    }

    int ch ;
    for( ch = fgetc( inf ) ; ch != EOF ; ch = fgetc( inf ) ) {
        fputc( next_char, outf ) ;
    }
}
```