CSE / ENGR 142 Programming I

Pointers and Output Parameters

Chapter 6

- 6.1 Output Parameters
- 6.2 Multiple calls to functions with output parameters
- 6.3 Scope of Names
- 6.4 Passing Output Parameters to other functions
- 6.6, 6.7 Debugging and common programming errors

Review: Function Terminology

```
0! is 1
1! is 1
2! is 1 * 2
3! is 1 * 2 * 3
...
```

function name

```
int factorial ( int n ) {
    int product, i ;
    product = 1 ;
    for ( i = n ; i > 1 ; i = i - 1 ) {
        product = product * i ;
    }
    return (product) ;
}
```

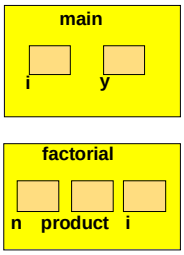
[formal] parameter

local variables

return type & value

Review: Local Variables

```
int
main(void)
{
    int i, y ;

    i = 3 ;
    y = factorial (i + 1) ;
    return (0) ;
}
```



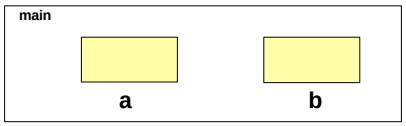
Local Variables: Summary

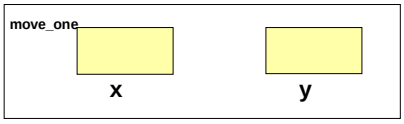
- Formal parameters and variables declared in a function are local to it:
  - cannot be directly accessed by other functions
- Allocated (created) on function entry.
- De-allocated (destroyed) on function return.
- Formal parameters are initialized by copying value of actual parameter.
- *Reminder: no global variables in 142!*

Call by Value

```
void move_one ( int x, int y ) {
    x = x - 1;
    y = y + 1;
}

int main ( void ) {
    int a, b ;
    a = 4 ;   b = 7 ;
    move_one(a, b) ;
    printf("%d %d", a ,b);
    return (0);
}
```

Trace

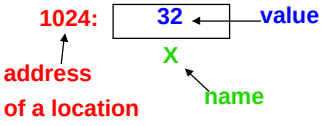




Values vs. Locations

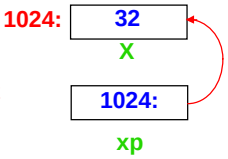
Problem: for *move_one (a,b)* to do what we want, it needs access to the locations of *a* and *b* as well as to their values.

Recall: variables name memory locations, which hold values.



New Type: Pointer

A pointer contains a reference to another variable; that is, the pointer contains the address of a variable.

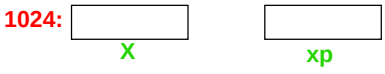


Declaring and Using a Pointer

```
int  x;          /* declares an int variable */
int  * xp;       /* declares a pointer to int */
```

If somehow, xp gets the address of x, then:

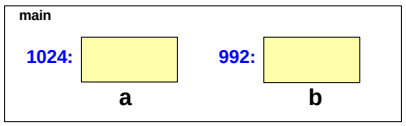
```
*xp = 0;         /* Assign integer 0 to x */
*xp = *xp + 1; /* Add 1 to x */
```

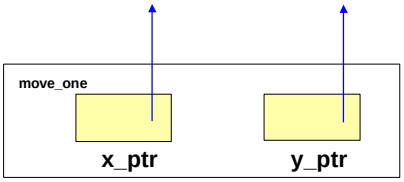


Pointer Solution to *move_one*

```
void move_one ( int  * x_ptr, int  * y_ptr ) {
    *x_ptr = *x_ptr - 1;
    *y_ptr = *y_ptr + 1;
}

int main ( void ) {
    int a, b ;
    a = 4 ;  b = 7 ;
    move_one( &a , &b ) ;
    printf("%d %d", a, b);
}
```

Trace





Addresses and Pointers

Three new types:

- **int *** "pointer to int"
- **double *** "pointer to double"
- **char *** "pointer to char"

Two new (unary) operators:

- **&** "address of"
  - & can be applied to any variable (or param)
- ***** "location pointed to by"
  - * can be applied only to a pointer

Vocabulary

- **Dereferencing or indirection:**
  - –following a pointer to a memory location
- **Output parameter:**
  - –a pointer parameter of a function
  - –can be used to provide a value ("input") as usual, and/or store a changed value ("output")
  - –Don't confuse with printed output (printf)

scanf Revisited

```
int x,y,z;
printf("%d %d %d", x, y, x+y);
```

What about *scanf*?

```
scanf("%d %d %d", x, y, x+y);
```
NO!

```
scanf("%d %d", &x, &y);
```
YES! (why?)

Why Use Pointers?

- **In CSE142, only used for output parameters:**
  - functions that need to change their actual parameters, e.g., *move_one*
  - to get multiple "return" values
    - e.g., *scanf ( )*
- *In advanced programming, pointers are used to create dynamic data structures.*

Sort Two Integers

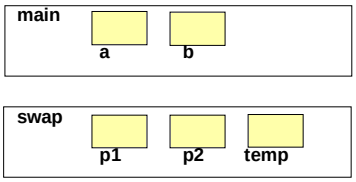
```
/* read in and sort 2 integers */

int c1, c2, temp ;
printf ( "Enter  2 integers: " ) ;
scanf ( "%d%d", &c1, &c2 ) ;
/*At this point the 2 values may be in either order*/
if ( c2 < c1 ) {     /* swap if out of order */
   temp= c1 ;
   c1    = c2 ;
   c2    = temp ;
}
/*At this point c1 <= c2 (guaranteed) */
```


swap as a Function

```
void swap ( int * p1, int * p2)  {
    int temp ;
    temp  = *p1 ;
    *p1    = *p2 ;
    *p2    = temp ;
}
```

```
int a, b ;
a = 4; b = 7;
…
swap (&a, &b) ;
```


Trace



Aliases

A way to think about pointer parameters:

*p1 and *p2 act like aliases for the variables in the call of swap.

When you change *p1 and *p2 you are changing the values of the variables in the call.

To set up these aliases you need to use &a, &b in the call.

Otherwise, calls are like Xerox copies (except for arrays which also use aliases)

Sorting

Problem: Sort 3 integers

Three-step algorithm:

1. *Read in three integers: x, y, z*
2. *Put smallest in x:*

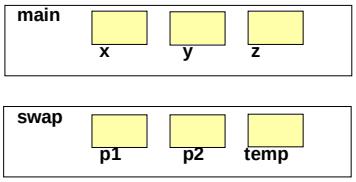
   Swap x, y if necessary; then swap x, z, if necessary.
3. *Put second smallest in y:*

   Swap y, z, if necessary.

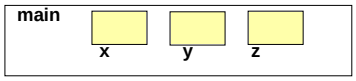
Sort 3 Integers

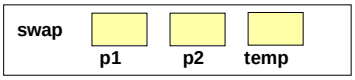
```
int main (void) {
int x, y, z, scanStatus ;
…
scanStatus = scanf("%d%d%d", &x, &y, &z) ;
if scanStatus == 3 {
  if ( x > y )  swap(&x, &y) ;
  if ( x > z )  swap(&x, &z) ;
  if ( y > z )  swap(&y, &z) ;
…
```

Trace



Trace





sort3 as a Function

```
/* interchange values as needed to establish */
/* *xp <= *yp <= *zp  */
void sort3 (int *xp, int *yp, int *zp)  {
    if ( *xp > *yp )  swap(xp, yp) ;
    if ( *xp > *zp )  swap(xp, zp) ;      ← NO &s!
    if ( *yp > *zp )  swap(yp, zp) ;
}

int main(void) {
    int x, y, z ;
    ...  /*scanf the values, then: */
    sort3(&x, &y, &z) ;
    ...
}
```


Why no & in swap call?

Real reason

xp and yp are **already** pointers that point to the variables that we want to swap

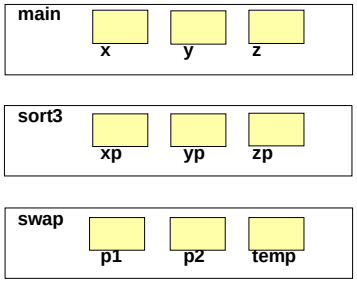
Alternative explanation using alias idea

*xp and *yp are aliases for the variables we want to swap

We want to allow swap to use aliases for *xp and *yp so we should use &(*xp) and &(*yp) in the call

BUT xp==&(*xp) **and** yp==&(*yp) **!!!!**

Trace

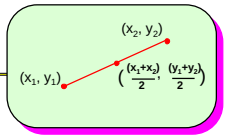


C is "strongly typed"

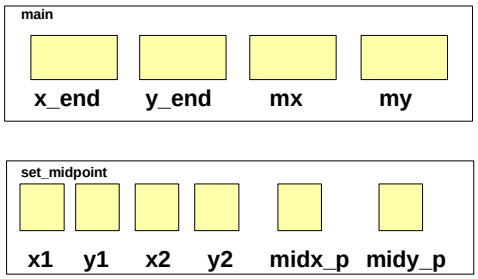
```
int i;  int * ip;
double x; double * xp;
...
x = i;          /* no problem */
i = x;          /* not recommended */

ip = 30;        /* No way */
ip = i;         /* Nope */
ip = &i;        /* just fine */
ip = &x;        /* forget it! */
xp = ip;        /* bad */
&i =  ip;       /* meaningless */
```


Midpoint Of A Line

```
/* Given 2 endpoints of a line, "return" coordinates of midpoint */

void set_midpoint(
   double x1, double y1,          /* 1st endpoint          */
   double x2, double y2,          /* 2nd endpoint          */
   double * midx_p, double * midy_p )  /* Pointers to midpoint */
{
   *midx_p = (x1 + x2) / 2.0;
   *midy_p = (y1 + y2) / 2.0;
}

double x_end, y_end, mx, my;
...
set_midpoint(0.0, 0.0, x_end, y_end, &mx, &my);
```



Trace



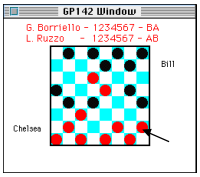
Example: Coordinates

Board Coordinates

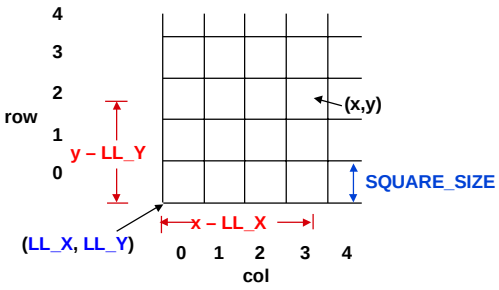
- row, column

Screen Coordinates

- x, y
- used by graphics package



Coordinate Conversion

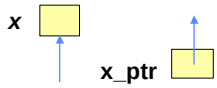


Coordinate Conversion

```
#define LL_X     40
#define LL_Y     20
#define SQUARE_SIZE  10

void screen_to_board (
   int screenx, int screeny,    /* coordinates on screen */
   int * row_p,  int * col_p)   /* position on board     */
{
   *row_p =  (screeny - LL_Y) / SQUARE_SIZE;
   *col_p  =  (screenx - LL_X) / SQUARE_SIZE;
}

screen_to_board (x, y, &row, &col);
```

Pointers vs. Values

	in caller	in callee
Declaration:	*int x*	*int * x_ptr*
To get the address of x:	*&x*	*x_ptr*
To get the value of x:	*x*	**x_ptr*



& in scanf again

```
void Read_a_or_b(char *chp) {
    char temp;
    printf("Enter an 'a' or a 'b'.\n");
    scanf("%c", &temp);
    while ( temp != 'a' && temp != 'b') {
       printf ("\nNope, it must be 'a' or 'b'!\n");
       scanf("%c", &temp);
    }
    *chp = temp;
}

int main(void) {
    char ch_ab;
    Read_a_or_b(&ch_ab);
    ...
```

chp temp

ch_ab

& in scanf again

Moral:
Wrong rule: "always use &'s in scanf"
Right rule: "always use addresses in scanf"

```
void Read_a_or_b(char *chp) {

    printf("Enter an 'a' or a 'b'.\n");
    scanf("%c", chp);
    while ( *chp != 'a' && *chp != 'b') {
       printf ("\nSorry, try again\n");
       scanf("%c", chp);
    }

}
int main(void) {
    char ch_ab;
    Read_a_or_b(&ch_ab);
    ...
```

chp

No '&' !

ch_ab

Additional Slides

- These are copied from the end of the Iteration slides

Event-Driven Programming

- Modern programs tend to be "event-driven"
  - Program starts, sets itself up.
  - Program enters a loop, waiting for some event or command to happen:
    - mouse click, key click, timer, menu selection, etc.
  - Program performs operation ("handles" the event or command)
  - Program goes back to its wait loop
- GP142 programs follow this model

Simple Command Interpreter

Repeatedly read in "commands" and handle them.

Input (symbolized by single characters)

a -- execute command A by calling *A_handler( )*

b -- execute command B by calling *B_handler( )*

q -- quit

Pseudocode for main loop:

```
get next command
if a, execute command A
if b, execute command B
if q, signal quit
```

Command Interpreter Loop Control Schema

repeat until quit signal

use variable "done" to indicate when done

```
set done to false
while not done
    body statements
    if quit command, set done to true
```


Command Interpreter main ( )

```
#define FALSE 0
#define TRUE  1

int main(void) {
    char command;
    int done;

    done = FALSE;
    while (! done){
                /* Input command from user */
        command = ReadCommand( );

        switch (command){
        case 'A':
        case 'a':
            A_handler();        /* Execute command A */
            break;
        case 'B':
        case 'b':
            B_handler();        /* Execute command B */
            break;
        case 'Q':
        case 'q':
            done = TRUE;        /* quit */
            break;
        default:
            printf("Unrecognized command\n");
        }
    }
    return 0;
}
```


do ... while

- Sometimes we want a loop to execute its body at least once before checking its condition
  - The command interpreter loop is a good example!
- In C, we can do this with a ***do ... while*** loop
- Similar to regular ***while***, but condition is written (and tested) after the loop body

```
do  {
   ...  /*loop body */
   }
   while (condition);
```


Command Interpreter main ( ) with do...while

```
#define FALSE 0
#define TRUE  1

int main(void) {
    char command;
    int done;

    done = FALSE;
    do  {
        command = ReadCommand( );

        switch (command){
        case 'A':
        case 'a':
            A_handler();        /* Execute command A */
            break;
        case 'B':
        case 'b':
            B_handler();        /* Execute command B */
            break;
        case 'Q':
        case 'q':
            done = TRUE;        /* quit */
            break;
        default:
            printf("Unrecognized command\n");
        }
    } while (! done);
    return 0;
}
```


Additional Slides

- These slides were adapted from the first Functions unit and/or the Files, Libraries, and I/O unit

Some HW3 Mysteries Revealed...

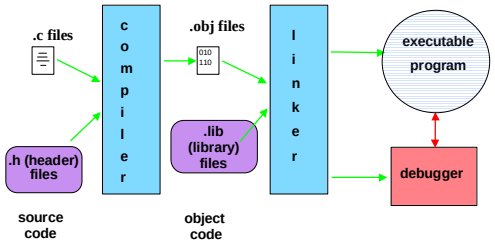
Remember this from hw3_orig.c?

```
/* Function prototypes for the functions contained in HW3.lib */
void ExitProgram (void);  /* Call this to...
int ReadMonth (void); /* Prompts user and ...
int GetWeekday(int month, int day, int year); /* Returns...
```

What exactly are prototypes, and what exactly are libraries?

Let's start by reviewing the process of building a C program

Compilers, Linkers, etc.



Files Used in Compiling

- **Source Files**
  - –**.c files:** C programs and functions
  - –**.h ("header") files:** fragments of C code
  - ***real-world projects may contain hundreds of source files!***
- **Compiled Files** (system-dependent names)
  - –**object files:** compiled C code ready to link
  - –**libraries:** collections of compiled C functions
  - –**executable files:** linked machine-language, ready to load into memory

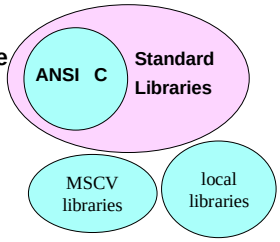
Libraries

Files of compiled, pre-written functions

Why?

- **Reuse existing code**
- **Enhance portability**
- **Hide system dependencies**

Examples: hw3lib, standard C I/O lib



Function Prototypes

- **A dilemma: If one .c file wants to call a function from a library (or from another .c file)**
  - –**the compiler needs to know: function name, return type, parameter types**
- **This is done with a "prototype" for the function**
  - –**Looks much like the start of a function definition**

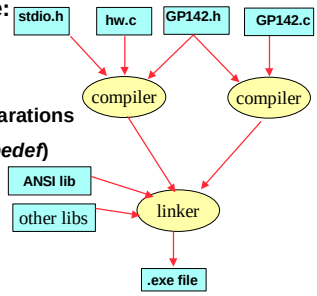
int GetWeekday(int month, int day, int year);

Using Prototypes

- **The prototypes can be coded directly into the .c program that uses them**
  - –**This is what we did for the hw3lib functions**
- **It is more common to create a .h file for each library or each shared .c file**
  - –**Example: stdio.h contains prototypes for the I/O functions of the ANSI C library**
- **Large programs typically consist of multiple .c files, *each* with a corresponding .h file**

More About Header (.h) Files

- **Fragments of C code:**
  - –**Function Prototypes**
  - –**Symbolic Constants**
  - –**Global Variable Declarations**
  - –**Type Definitions (*typedef*)**



Another Use For Prototypes

- **The ordering rule for C identifiers requires that the function names be declared before they can be used (in a call).**
- **Defining all the functions in the proper order is sometimes impossible or inconvenient**
  - **function A calls B, *and* B calls A**
  - **A calls B, but we would rather write B after A in the program**
- **Solution: *Put a prototype for the function near the top of the program, then define the function anywhere convenient later on***
  - **order is now unimportant**