CSE / ENGR 142 Programming I

Pointers and Output Parameters

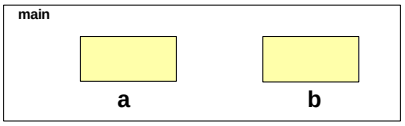
Chapter 6

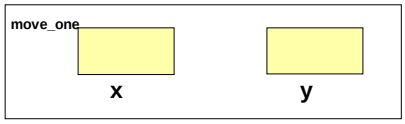
- 6.1 Output Parameters
- 6.2 Multiple calls to functions with output parameters
- 6.3 Scope of Names
- 6.4 Passing Output Parameters to other functions
- 6.6, 6.7 Debugging and common programming errors

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



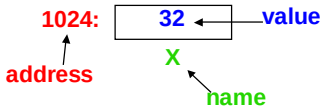


Values vs. Locations

Recall: variables name memory locations, which hold values.

move_one (a,b) **needs access to the locations of *a* and *b* as well as to their values.**

How?



New Type: Pointer

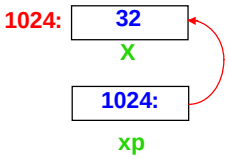
A pointer contains a reference to another variable; that is, the pointer contains the address of a variable.

```
int  x;
int  *xp;
```

pointer to int

```
xp = &x ;
```

address of x



Using a Pointer

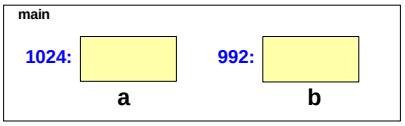
Access location pointed to by pointer

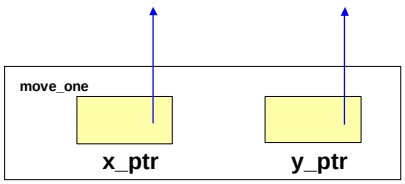
```
xp = &x;        /* Assign address of x to xp */
*xp = 0;        /* Assign integer 0 to x */
*xp = *xp + 1; /* Add 1 to x */
```

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

Pointers Abstractly

```
int x ;
int * p ;
p = &x;
...
(x == *p)   True
(p == &x)   True
```

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?

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

Sort Two Integers

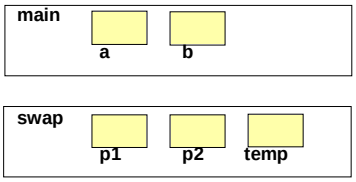
```
/* read in and sort 2 integers */

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:***

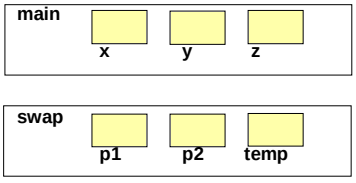
   Swap x, y if necessary; then swap x, z, if necessary.
3. ***Put second smallest in y:***

   Swap y, z, if necessary.

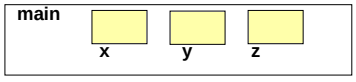
Sort 3 Integers

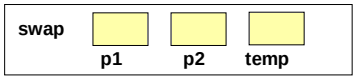
```
int main (void) {
int x, y, z ;
...
scanf("%d%d%d", &x, &y, &z) ;
if ( x > y )  swap(&x, &y) ;
if ( x > z )  swap(&x, &z) ;
if ( y > z )  swap(&y, &z) ;
...
```


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 ;
    ...
    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

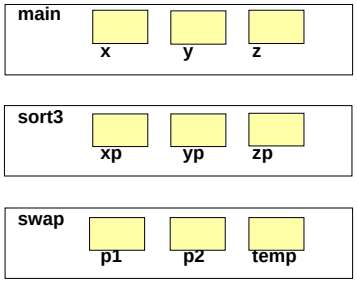
Alternative explanation using alias idea

*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



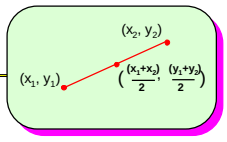
C is "strongly typed"

```
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 */
```

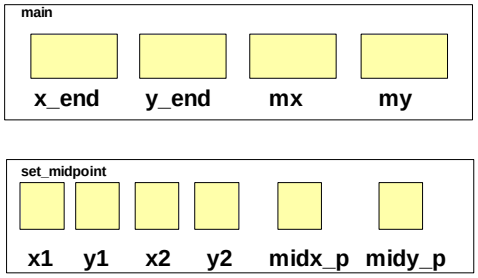

Midpoint Example

/* 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

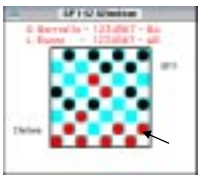
Board Coordinates

row, column

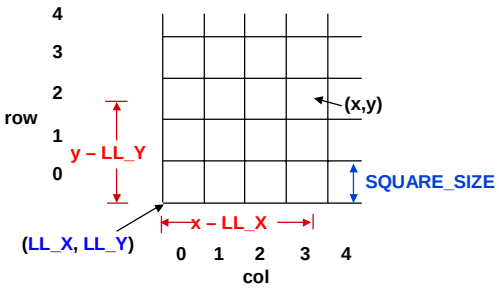
Screen Coordinates

x, y

used by graphics package



Coordinate Conversion



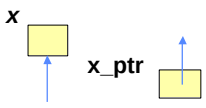
Coordinate Conversion

```
#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);
```

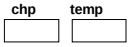

Pointers vs. Values

Declaration:	*int x*	*int * x_ptr*
To get the pointer:	*&x*	*x_ptr*
To get the value:	*x*	**x_ptr*



& in scanf again

```
void scan_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;
    scan_a_or_b(&ch_ab);
    ...
```



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

& in scanf again

```
void scan_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;
    scan_a_or_b(&ch_ab);
    ...
```

chp

No '&' !

ch_ab