

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

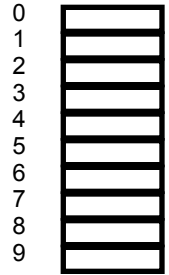
6.6, 6.7 Debugging and common programming errors

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);  
}
```

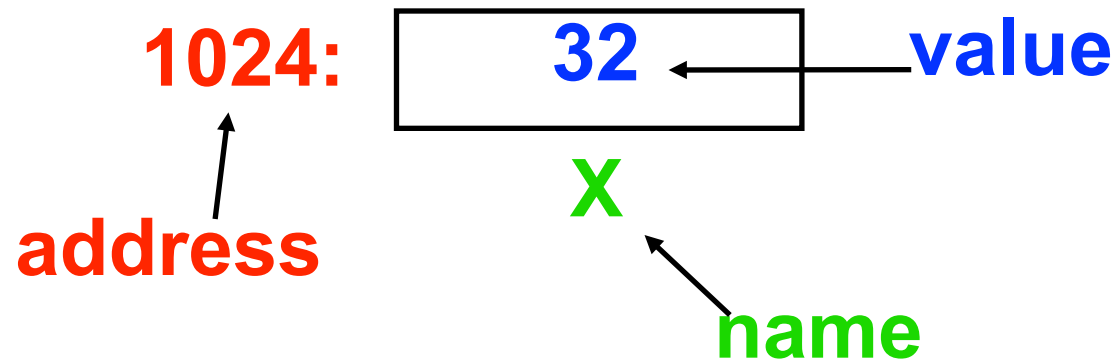
Values vs. Locations



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*;
pointer to int

int **px*;

px = **&***x*;

address of x

1024:



x



1024:

px

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”

Pointers vs. Values

Declaration:

int x

*int * ptr_x*

To get the pointer:

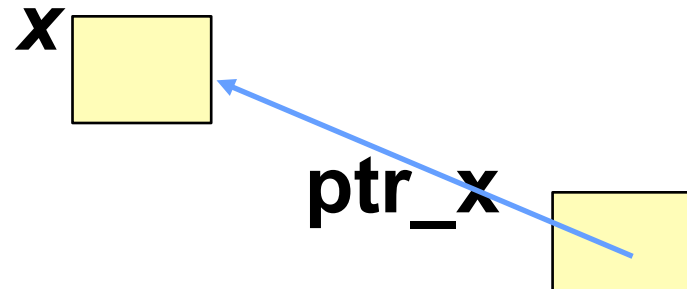
&x

ptr_x

To get the value:

x

**ptr_x*



Using a Pointer

Access location pointed to by pointer

`px = &x;` */* Assign address of x to px */*

`*px = 0;` */* Assign integer 0 to x */*

`*px = *px + 1;` */* Add 1 to x */*

1024:



x



px

Pointer Solution to *move_one*

```
void move_one ( int * ptr_x, int *  
ptr_y ) {  
    *ptr_x = *ptr_x - 1;  
    *ptr_y = *ptr_y + 1;  
}
```

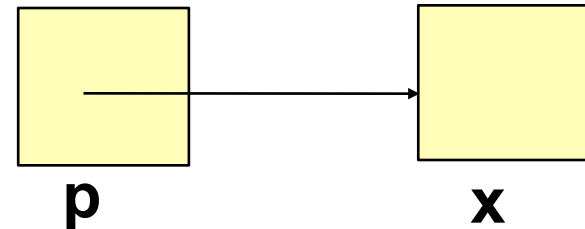
```
int main ( void ) {  
    int a, b ;  
    a = 4 ; b = 7 ;  
    move_one( &a , &b ) ;  
    printf(“%d %d”, a, b);  
}
```

Pointers Abstractly

```
int x ;  
int * p ;  
p = &x;  
...
```

$(x == *p)$
True –
comparing
values!


$(p == \&x)$
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)

Review: * and &

-  **is *overloaded*** : it has two meanings depending on where it is.
 - between a type and a variable name : it declares a pointer variable.

```
int * pi; // a new variable named pi of type "pointer to int"
```
 - in front of a pointer variable : it *dereferences* the pointer
 - `*pi = 2;` // assign 2 into *the place where pi points*

Review: * and &

- **&** is *overloaded* : it has two meanings depending on where it is.
 - in front of a regular (non-pointer) variable :returns the address of the variable

```
int x;  
int * pi; // a new variable named pi of type "pointer to int"  
pi = &x; // pi now points to x
```
 - in C++, & has even more meanings, especially in function prototypes
 - we will not use these in this class

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);
```

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*

Sort Two Integers Revisited

```
/* 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 ;
```

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;
```

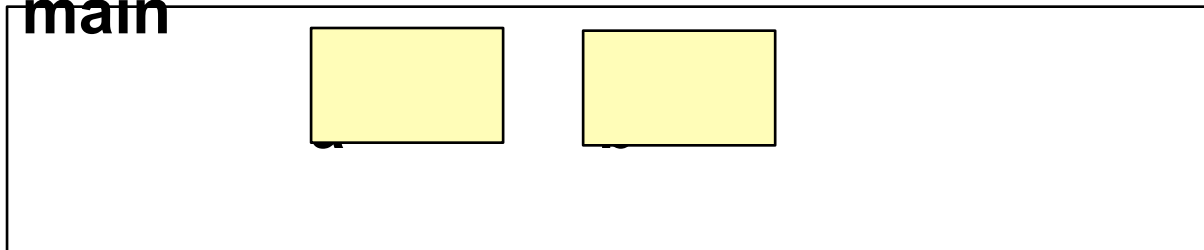
...

```
swap (&a, &b) ;
```

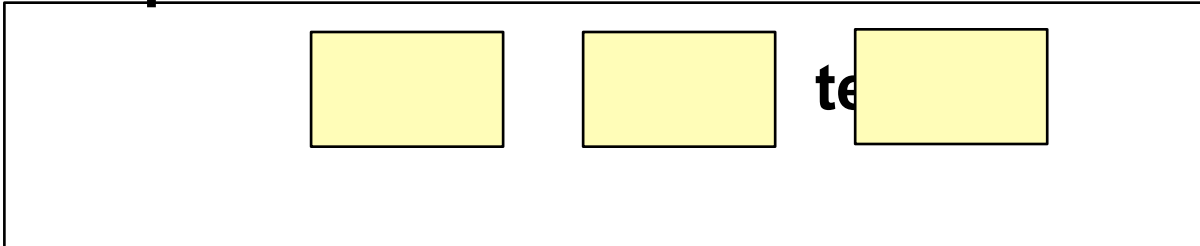
Trace

using debugger

main



swap



Aliases

A way to think about pointer parameters:

***p1** and ***p2** act like an **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.

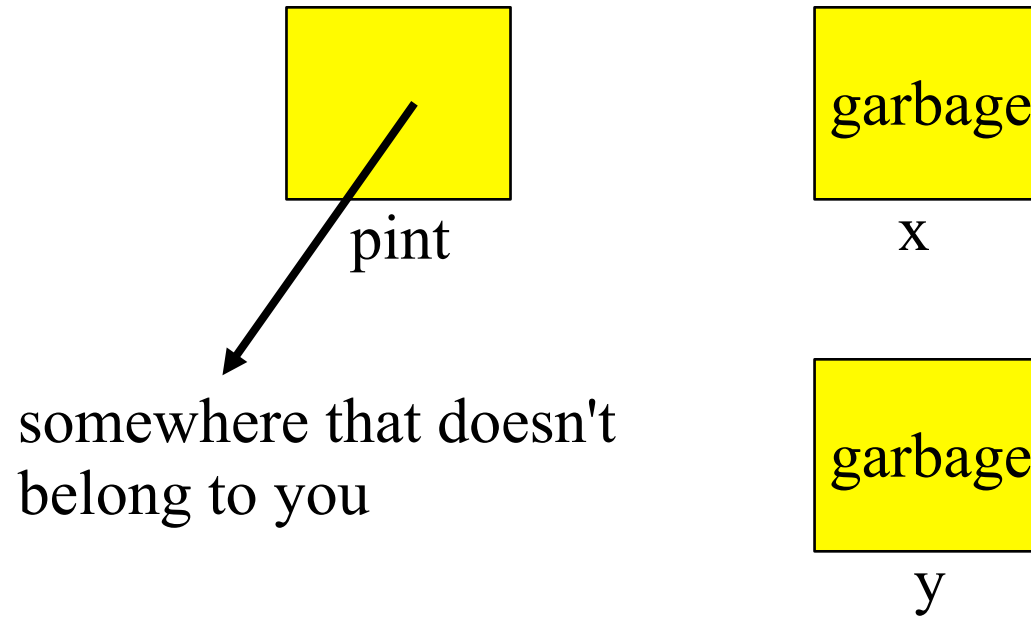
Otherwise, calls are like Xerox copies

(except for arrays which also use aliases)

What's in a Pointer?

- like any other variable, garbage until you initialize it.

```
int * pint;  
int x,y;
```



What's in a Pointer?

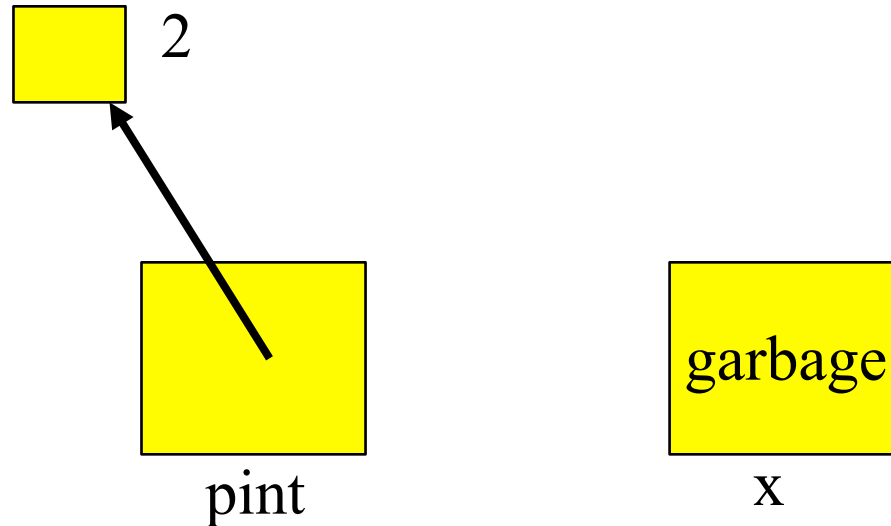
- Don't try to assign to pointers directly

```
int * pint;
```

```
pint = 2; // bad – points to memory location '2';
```

What's bad?

Memory address
2 doesn't belong
to your program.

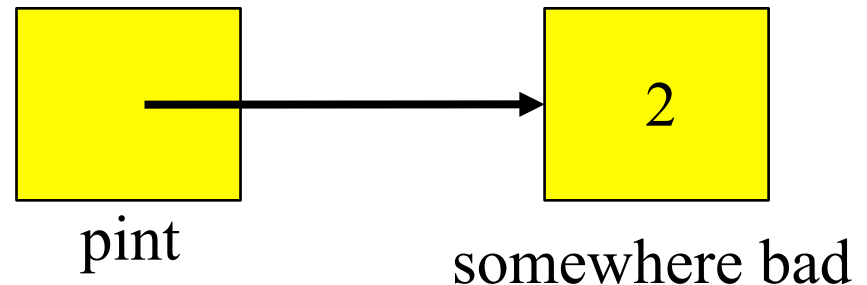


What's in a Pointer?

- Don't assign into what a pointer points to, unless you've first initialized the pointer.

```
int * pint;
```

```
*pint = 2; // bad – tries to put '2' in somewhere –  
probably a memory location that doesn't  
belong to you
```



What's in a Pointer?

- **Assign addresses to pointers**

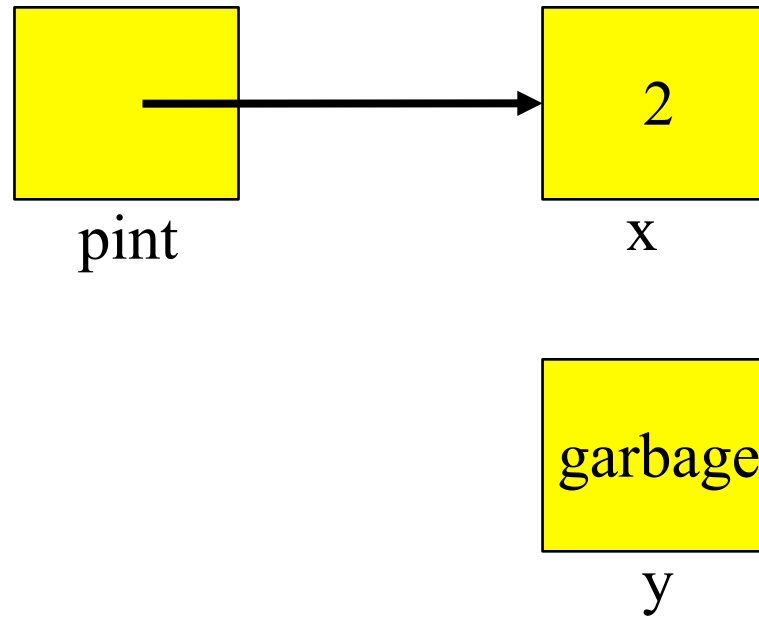
```
int * pint;
```

```
int x,y;
```

```
x= 2;
```

```
pint = &x;
```

```
*pint = 2;
```



Using Output Parameters

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 – return all three:***
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) ;  
...  
}
```

sort3 as a Function

```
void sort3 (int *xp, int *yp, int *zp) {  
    if ( *xp > *yp ) swap(xp, yp) ;  
    if ( *xp > *zp ) swap(xp, zp) ;  
    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

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 */
```

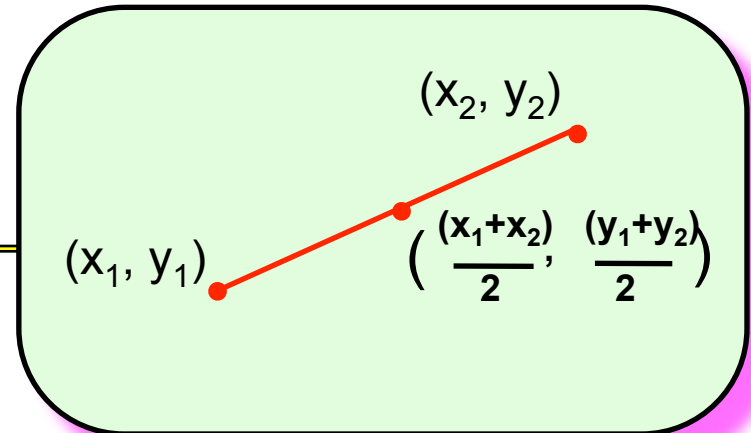
```
ip = &x;        /* forget it! */
```

```
xp = ip;       /* bad */
```

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);
```

& 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;
```

chp

temp

ch_ab

Moral:

Wrong rule: “always use &’s in scanf”

Right rule: “always use addresses in
scanf”

& in scanf again, II

```
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

Const and Pointers

- **yes, pointers can be const too**

```
int x;  
const int * ptr_to_x = &x;
```

- **...and so can what they point to!**

```
const double PRICE=34.99;  
const double * const ptr_to_price = &double;
```

- **we'll see this again when we do strings...**

Const and Pointers

- If the const is on the left, the pointer is const – it can only point in one place.

```
int x;  
const int * ptr_to_x = &x;
```

- If the const is between the * and the variable name, what's pointed to is const.

```
const int x =3;  
int * const ptr_to_x = &x;
```