CSE 142
Computer Programming I

Pointer Parameters
or... Mysteries of scanf Revealed!

Overview
Concepts this lecture
Function parameters
Call by value (review)
Pointer parameters - call by reference
Pointer types
& and * operators

Reading
6.1 Output (pointer) 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

What Does This Print?
/* change x and y */
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;
}

Output: 4 7

Function Call Review
Remember how function calls are executed:
Allocate space for parameters and local variables
Initialize parameters by copying argument values
Begin execution of the function body

Trace
/* change x and y */
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;
}

Output: 4 7
Call By Value is Not Enough

Parameters are initialized with copies of the arguments; there is no further connection!

If a function changes its parameters, it affects the local copy only.

To change the arguments in the caller, the function needs access to the locations of the arguments, not just their values.

But... why would we ever want to change the parameters?

New Type: Pointer

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

the variable xp has type pointer to int
(often written: xp has type int*)

Pointer Solution to move_one

The & operator in front of a variable name creates a pointer to that variable

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);
    return 0;
}

Trace

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);
    return 0;
}

Output: 3 8

Trace

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);
    return 0;
}

Output: 3 8
**Aliases**

*x_ptr and *y_ptr act like aliases for the variables a and b in the function call.
When you change *x_ptr and *y_ptr you are changing the values of the caller’s variables.
To create these aliases you need to use &a, &b in the call.

**Pointer Types**

Three new types:
- `int *` “pointer to int”
- `double *”pointer to double”`
- `char *”pointer to char”`

These are all different:
- pointer to a char can’t be used where a pointer to an int is needed.
- pointer to double can’t be used where a double is needed!

**Pointer Operators**

Two new (unary) operators:
- `&”address of”`
- `*”location pointed to by”`

* can be applied only to a pointer

Keep track of the types:
- if x has type double,
  - &x has type “pointer to double” or “double *”

**Vocabulary**

Dereferencing or indirection:
- following a pointer to a memory location

The book calls pointer parameters “output parameters”:
- can be used to provide a value (“input”) as usual, and/or store a changed value (“output”)
- Don’t confuse with printed output (printf)

**Why Use Pointers?**

For parameters:
- in functions that need to change their actual parameters (such as move_one)
- in functions that need multiple “return” values (such as scanf)

These are the only uses in this course

In advanced programming, pointers are used to create dynamic data structures.
Now we can make sense out of the punctuation in `scanf`

```c
int x, y, z;
scanf("%d %d %d", x, y, x+y); // NO!
scanf("%d %d", &x, &y); // YES! Why?
```

---

**Problem:** Find the midpoint of a line segment.

**Algorithm:** Find the average of the coordinates of the endpoints:

\[
\begin{align*}
\text{x}_{\text{mid}} &= \frac{x_1 + x_2}{2.0} \\
\text{y}_{\text{mid}} &= \frac{y_1 + y_2}{2.0}
\end{align*}
\]

**Programming approach:** We'd like to package this in a function.

---

**Function Specification**

Function specification: given endpoints \((x_1,y_1)\) and \((x_2,y_2)\) of a line segment, store the coordinates of the midpoint in \((\text{midx}, \text{midy})\).

Parameters:
- x1, y1, x2, y2, &midx, &my

The \((\text{midx}, \text{my})\) parameters are being altered, so they need to be pointers.

---

**Midpoint Function: Code**

```c
void set_midpoint( double x1, double y1, 
double x2, double y2, 
double *midx_p, double *my_p )
{
    *midx_p = (x1 + x2) / 2.0;  
    *my_p   = (y1 + y2) / 2.0;
}
```

---

**Example: Gameboard Coordinates**

- **Board Coordinates**
  - row, column (used by players)
- **Screen Coordinates**
  - x, y (used by graphics package)

Problem: convert \((x,y)\) to \((\text{row}, \text{col})\)
**Coordinate Conversion: Analysis**

![Coordinate Conversion Diagram]

**Coordinate Conversion: Code**

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

void screen_to_board (int screenx, int screeny, /* coords 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);
```

**Problem: Reorder**

Suppose we want a function to arrange its two parameters in reverse numeric order.

Example:
- -1, 5 need to be reordered as 5, -1
- 12, 3 is already in order (no change needed)

Parameter analysis: since we might change the parameter values, they have to be pointers

This example is a small version of a very important problem in computer science, called “sorting”

**Code for Reorder**

```c
/* ensure *p1 >= *p2, interchanging values if needed */
void reorder(int *p1, int *p2) {
    int tmp;
    if (*p1 < *p2) {
        tmp = *p1;
        *p1 = *p2;
        *p2 = tmp;
    }
}
```

```
 These 3 lines can be said to "swap" two values
```

**swap as a Function**

```c
/* interchange "p" and "q" */
void swap (int *p, int *q) {
    int temp = *p;
    *p = *q;
    *q = temp;
}
```

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

**Reorder Implemented using swap**

```c
/* ensure *p1 >= *p2, interchanging values if needed */
void reorder(int *p1, int *p2) {
    if (*p1 < *p2) {
        swap(____, ____);
    }
}
```

What goes in the blanks?
Pointer Parameters (Wrong!)

Normally, if a pointer is expected, we create one using &:

```c
/* ensure *p1 >= *p2, interchanging values if needed */
void reorder(int *p1, int *p2) {
    if (*p1 < *p2)
        swap( &p1 , &p2 );
}
But that can’t be right - p1 and p2 are already pointers!
What are the types of expressions &p1 and &p2?
```

Pointer Parameters (Right!)

Right answer: if the types match (int *), we use the pointers directly

```c
/* ensure *p1 >= *p2, interchanging values if needed */
void reorder(int *p1, int *p2) {
    if (*p1 < *p2)
        swap(p1,p2);
}
```

Pointers and scanf Once More

Problem: User is supposed to enter ‘y’ or ‘n’, and no other answer is acceptable. Read until user enters ‘y’ or ‘n’ and give character to caller

```c
void Read_y_or_n(char *chp) {
    ...}
int main(void) {
    char ch;
    Read_y_or_n(&ch);
    ...}
```

Wrapping Up

Pointers are needed when the parameter value may be changed & creates a pointer * dereferences the value pointed to

This completes the technical discussion of functions in C for this course

Learning how to design and use functions will be a continuing concern in the course
QOTD: When an Irresistible Force Meets an Immutable Object

Does the function call to swap inside the function below change the value of px?
- If so, to what?
- If not, is it possible for any function call to change px's value? How?

```c
void tryIt(int * px, int * py)
{
    swap(px, py);
}
```

WARNING: This may be harder than it seems!