CSE 142
Computer Programming I

Complex Conditions

From Homework Descriptions to Programs

Negating Conditions

Suppose we want a while loop to terminate as soon as *either x is 17 or x is 42*

Which is it?

```
while (x!=17 || x!=42) …
while (x!=17 && x!=42) …
```

Either way? Something else?

Truth tables and DeMorgan's Law give us tools for answering such questions

Truth Tables for && and ||

A "truth table" lists all possible combinations of values, and the result of each combination

P Q	P && Q	P || Q
T T	T	T
T F	F	T
F T	F	T
F F	F	F

P and Q stand for any conditional expressions ("boolean value")

Truth Table for not (!)

P	!P
T	F
F	T

Equivalence of Complex Expressions

```
if ( ! (age < 25  &&  sex  ==  'M' ) )
        printf ( "Cheap rates. \n") ;
```

is equivalent to

```
if ( age >= 25  ||  sex  !=  'M' ) )
        printf ( "Cheap rates. \n") ;
```

Or is it?

DeMorgan's Laws

DeMorgan's laws help determine when two complex conditions are equivalent

They state:

! (P && Q)   is equivalent to (!P || !Q)

! (P || Q)   is equivalent to (!P && !Q)

This applies for any Boolean expressions P and Q, which might themselves be complex expressions

Proof of DeMorgan

Is it really true that !(P&&Q) == (!P || !Q) ?

P	Q	(P&&Q)	!(P&&Q)	!P	!Q	(! P || !Q)
T	T	T	F	F	F	F
T	F	F	T	F	T	T
F	T	F	T	T	F	T
F	F	F	T	T	T	T

Exercise: Prove the other law

Solution To a Previous Question

We wanted a while loop to terminate as soon as either x is 17 or x is 42.

So the loop condition is

```
          while ( ! (x==17 || x==42)) …
```

Using DeMorgan's Law we can rewrite as

```
          while (x != 17 && x != 42) …
```

A truth table would show that

```
          while (x != 17 || x != 42)
```

is wrong! (It's always true, for one thing…)

CSE 142
Computer Programming I

Pointer Parameters

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

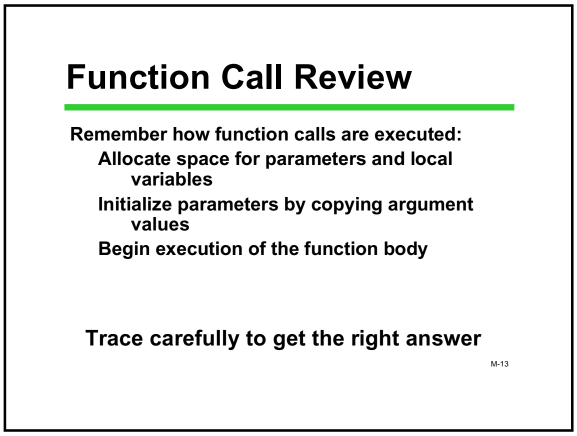
6.6, 6.7 Debugging and common programming errors

What Does This Print?

```
/* change x and y */
void MoveOne ( int x, int y ) {
    x = x - 1;
    y = y + 1;
}

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

Output:

```
3   8   ?
4   7   ?
```



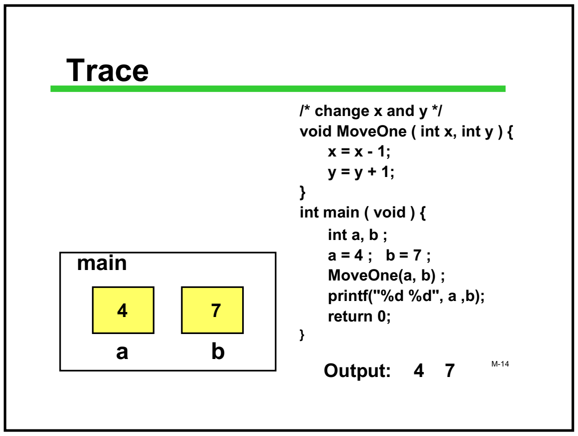
Function Call Review

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 carefully to get the right answer

M-13

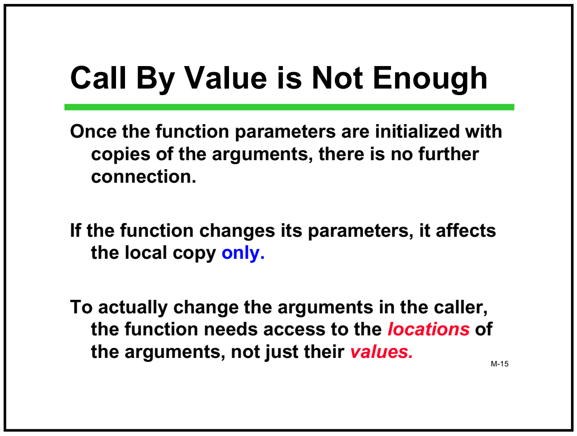


Trace

```
/* change x and y */
void MoveOne ( int x, int y ) {
     x = x - 1;
     y = y + 1;
}
int main ( void ) {
     int a, b ;
     a = 4 ;   b = 7 ;
     MoveOne(a, b) ;
     printf("%d %d", a ,b);
     return 0;
}
```

Output: 4   7

M-14



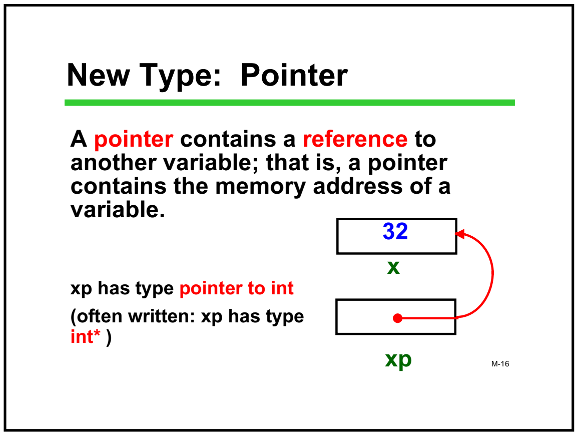
Call By Value is Not Enough

Once the function parameters are initialized with copies of the arguments, there is no further connection.

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

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

M-15



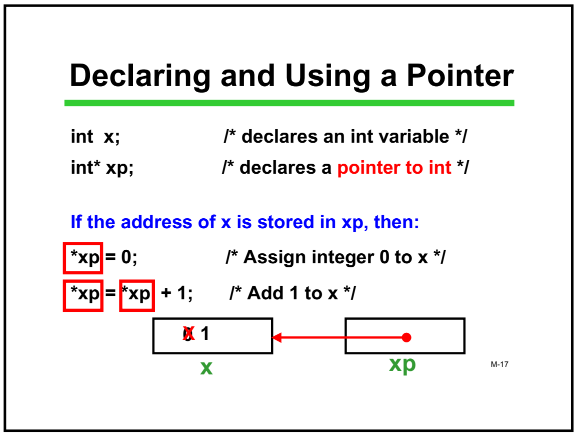
New Type: Pointer

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

xp has type pointer to int

(often written: xp has type int*)

M-16



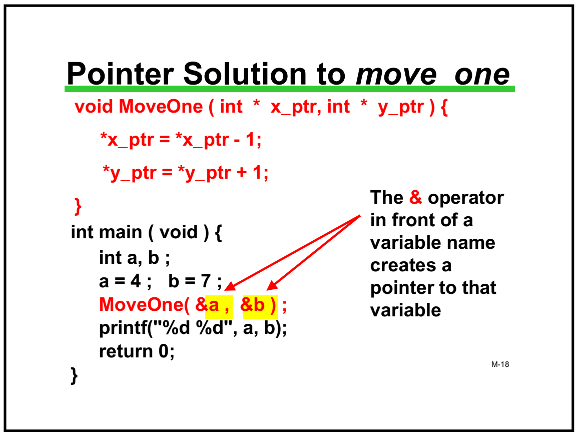
Declaring and Using a Pointer

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

If the address of x is stored in xp, then:

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

M-17



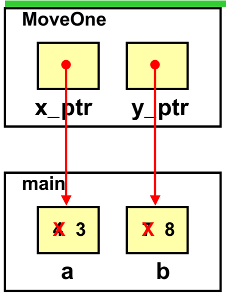
Pointer Solution to *move_one*

```
void MoveOne ( 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 ;
    MoveOne( &a ,  &b ) ;
    printf("%d %d", a, b);
    return 0;
}
```

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

M-18

Trace



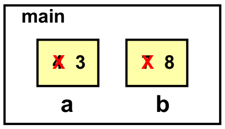
```
void MoveOne (
      int  * x_ptr,
      int  * y_ptr ) {
   *x_ptr = *x_ptr - 1;
   *y_ptr = *y_ptr + 1;
}
```

```
a = 4 ;   b = 7 ;
MoveOne( &a ,  &b ) ;
```

Output:

Trace

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



```
a = 4 ;   b = 7 ;
MoveOne( &a ,  &b ) ;
```

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 - a pointer to a char can't be used if the function parameter is supposed to be a pointer to an int, for example.

Pointer Operators

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

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.

scanf Revisited

Now we can make sense out of the punctuation in *scanf*

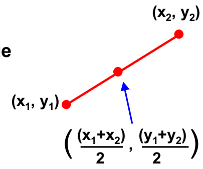
```
int x,y,z;

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

Example: Midpoint Of A Line

Problem: Find the midpoint of a line segment.

Algorithm: find the average of the coordinates of the endpoints:

```
xmid = (x1+x2)/2.0;
ymid = (y1+y2)/2.0;
```



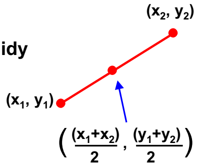
Programming approach: We’d like to package this in a function

Function Specification

Function specification: given endpoints (x1,y1) and (x2,y2) of a line segment, store the coordinates of the midpoint in (midx, midy)

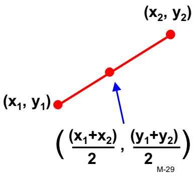
Parameters:
x1, y1, x2, y2, midx, and midy

The (midx,midy) parameters are being altered, so they need to be pointers



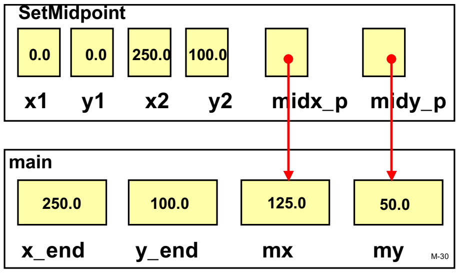
Midpoint Function: Code

```
void SetMidpoint( double x1, double y1,
                  double x2, double y2,
                  double * midx_p, double * midy_p )
{
      *midx_p = (x1 + x2) / 2.0;
      *midy_p = (y1 + y2) / 2.0;
}

double x_end, y_end, mx, my;
x_end = 250.0; y_end = 100.0;
SetMidpoint(0.0, 0.0,
            x_end, y_end,
            &mx, &my);
```



Trace

```
SetMidpoint(0.0, 0.0,
            x_end, y_end,
            &mx, &my);
```



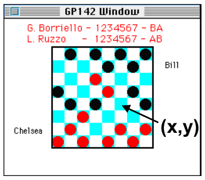
Example: Gameboard Coordinates

Board Coordinates

row, column (used by players)

Screen Coordinates

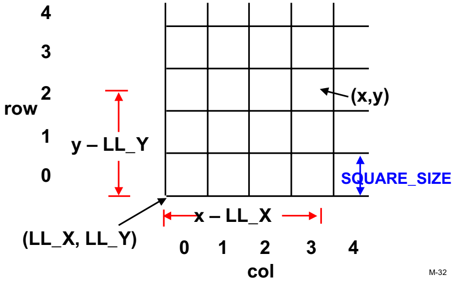
x, y (used by graphics package)



Problem: convert (x,y) to (row,col)

M-31

Coordinate Conversion: Analysis



M-32

Coordinate Conversion: Code

```
int LL_X = 40;
int LL_Y = 20;
int 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);
```

M-33

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"

M-34

Code for Reorder

```
/* 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

M-35

swap as a Function

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

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

M-36

Reorder Implemented using *swap*

```
/* 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 &:

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

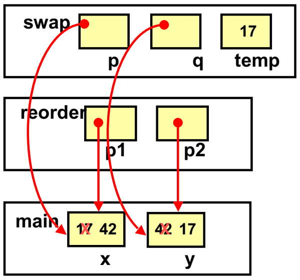
But that can't be right - p1 and p2 are already pointers!
What are the types of expressions &p1 and &p2?

Pointer Parameters (Right!)

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

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

Trace



```
void swap(int *p,
               int *q){
   …
}

void reorder(int*p1,
               int*p2) {
   if (*p1 < *p2)
      swap(p1,p2);
}

int x, y;
x = 17; y = 42;
reorder(&x,&y);
```

Pointers and scanf Once More

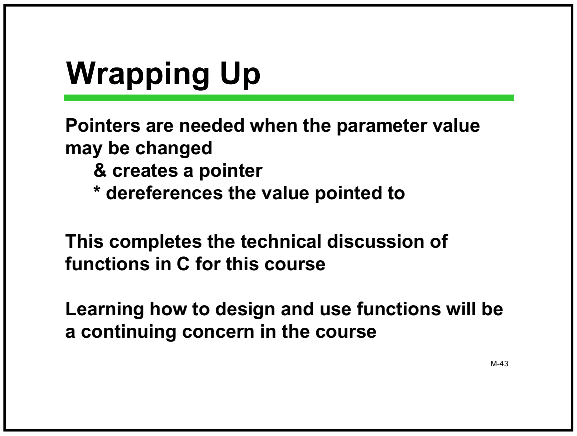
Problem: User is supposed to enter 'y' or 'n', and no other answer is acceptable. Read until user enters 'y' or 'n' and return input

```
void Read_y_or_n(char *chp) {
    ...
}

int main(void) {
    char ch;
    Read_y_or_n(&ch);
    ...
```

Pointers and scanf Once More

```
/* read until user enters 'y' or 'n' and return input */

void Read_y_or_n(char *chp) {
    printf("Enter an 'y' or a 'n'.\n");
    scanf("%c", chp);
    while ( *chp != 'y' && *chp != 'n') {
       printf ("\nSorry, try again\n");
       scanf("%c", chp);
    }
}
int main(void) {
    char ch;
    Read_y_or_n(&ch);
    ...
```

No '&' !



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

M-43