```
/* Dynamic memory allocation (DMA) example of an Airplane structure.
   It shows a lot of ways to work with structs, arrays, addresses, and
   pointers.

   Here is the output from the program:

sizeof(struct Airplane) = 68

Using a pointer:      Flight 101 goes from Vancouver to Calgary
Not using a pointer:  Flight 101 goes from Vancouver to Calgary

Using a pointer:      Flight 201 goes from Vancouver to Edmonton
Not using a pointer:  Flight 201 goes from Vancouver to Edmonton

Using a pointer:      Flight 202 goes from Vancouver to Winnipeg
Not using a pointer:  Flight 202 goes from Vancouver to Winnipeg

Using a pointer:      Flight 301 goes from Montreal to Toronto
Not using a pointer:  Flight 301 goes from Montreal to Toronto

Using a pointer:      Flight 401 goes from Toronto to Vancouver
Not using a pointer:  Flight 401 goes from Toronto to Vancouver

Using a pointer:      Flight 402 goes from Toronto to San Francisco
Not using a pointer:  Flight 402 goes from Toronto to San Francisco

Using a pointer:      Flight 403 goes from Toronto to Los Angeles
Not using a pointer:  Flight 403 goes from Toronto to Los Angeles

Using a pointer:      Flight 503 goes from New York to Calgary
Not using a pointer:  Flight 503 goes from New York to Calgary

Using a pointer:      Flight 504 goes from Calgary to Fort McMurray
Not using a pointer:  Flight 504 goes from Calgary to Fort McMurray
Press any key to continue . . .

*/
```

Airplane structs
Arrays
Addresses
Pointers
DMA

Output

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_CITY_LENGTH 32
```

An Airplane structure has 3 fields:

flight-number	source	destin.

```
/* Define an Airplane data type.  It holds an Airplane struct (record) of
   related information about one plane. */
struct Airplane {
   int      flight_number;
   char     source[MAX_CITY_LENGTH];       /* array of characters */
   char     destination[MAX_CITY_LENGTH];  /* probably better to use DMA */
};


/* Function prototypes */
void print_airplane(struct Airplane   plane);      /* copy the struct */
void print_airplane_PTR(struct Airplane * plane);  /* use a pointer instead */
```

We'll use a pointer for one of the functions

```
int main(void)
{
                                           /* AC = Air Canada;  WJ = WestJet */
1    struct Airplane     AC;               /* allocate one plane */
2    struct Airplane     WJ[10];           /* allocate an array of 10 planes */
3    struct Airplane *   dynamic_AC;       /* point to one (or more) planes */
4    struct Airplane *   dynamic_AC2;      /* point to one (or more) planes */
5    struct Airplane *   dynamic_WJ[10];   /* allocate an array of 10 POINTERS;
                                              EACH pointer will point to
                                              one (or more) planes */
     struct Airplane *   temp_show_me[10];
```

We'll show examples using different variables.

```
    printf("sizeof(struct Airplane) = %d\n", sizeof(struct Airplane));
```

Example ① AC

```
    /* Example 1:  A stand-alone, statically (automatically), allocated record. */
    AC.flight_number = 101;
    strcpy(AC.source, "Vancouver");       /* strcpy_s would be more secure */
    strcpy(AC.destination, "Calgary");
    print_airplane_PTR(&AC);              /* pass an address (pointer) across */
    print_airplane(AC);                   /* copy the whole struct across */
```

AC

101	Vancouver	Calgary

② WJ[10] array of structs

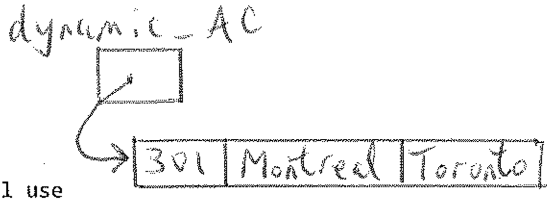
```
    /* Example 2:   A statically (automatically) allocated array.
                    We'll use 2 of the 10 structs in the array. */
    /* Example 2a:  We'll use the familiar, APSC 160 way [ ]. */
    WJ[5].flight_number = 201;
    strcpy(WJ[5].source, "Vancouver");
    strcpy(WJ[5].destination, "Edmonton");
    print_airplane_PTR(&WJ[5]);     /* pass an address/pointer */
    print_airplane(WJ[5]);          /* pass the whole element (a struct) */

    /* Example 2b:  We'll use another popular way, using pointer arithmetic. */
    (*(WJ + 6)).flight_number = 202;              /* same as WJ[6] */
    strcpy( (*(WJ + 6)).source, "Vancouver");
    strcpy( (WJ + 6)->destination, "Winnipeg");  /* also works */

    print_airplane_PTR( WJ + 6 );  /* pass an address */
    print_airplane( *(WJ + 6) );   /* pass the whole element */
```

WJ

[0]	?	?	?
[1]	?	?	?
[2]	?	?	?
[3]	?	?	?
[4]	?	?	?
[5]	201	Vancouver	Edmonton
[6]	202	Vancouver	Winnipeg
[7]	?	?	?
[8]	?	?	?
[9]	?	?	?

③ dynamic_AC

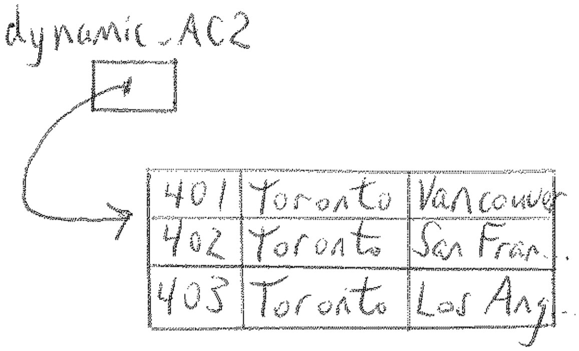
```
    /* Example 3:   Dynamically allocate exactly one plane. */
    dynamic_AC = (struct Airplane *) malloc( 1 * sizeof(struct Airplane) );
    dynamic_AC->flight_number = 301;
    strcpy(dynamic_AC->source, "Montreal");
    strcpy(dynamic_AC->destination, "Toronto");
    print_airplane_PTR(dynamic_AC);     /* pass an address */
    print_airplane(*dynamic_AC);        /* pass the actual struct */
```



301	Montreal	Toronto

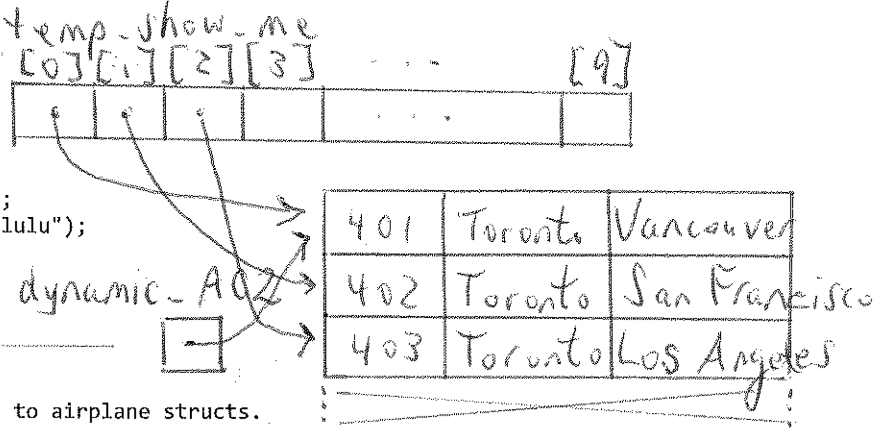
④ dynamic_AC2

```
    /* Example 4:   Dynamically allocate more than one plane, but still use
                    only one pointer. */
    dynamic_AC2 = (struct Airplane *) malloc( 3 * sizeof(struct Airplane) );

    dynamic_AC2[0].flight_number = 401;
    // Above line is same as:   (*(dynamic_AC2 + 0)).flight_number = 401; */
    strcpy(dynamic_AC2[0].source, "Toronto");
    strcpy(dynamic_AC2[0].destination, "Vancouver");
    print_airplane_PTR(&dynamic_AC2[0]);   /* pass an address */
    print_airplane(dynamic_AC2[0]);        /* pass the struct */

    dynamic_AC2[1].flight_number = 402;
    strcpy(dynamic_AC2[1].source, "Toronto");
    strcpy(dynamic_AC2[1].destination, "San Francisco");
    print_airplane_PTR(&dynamic_AC2[1]);
    print_airplane(dynamic_AC2[1]);
```



401	Toronto	Vancouver
402	Toronto	San Fran.
403	Toronto	Los Ang.

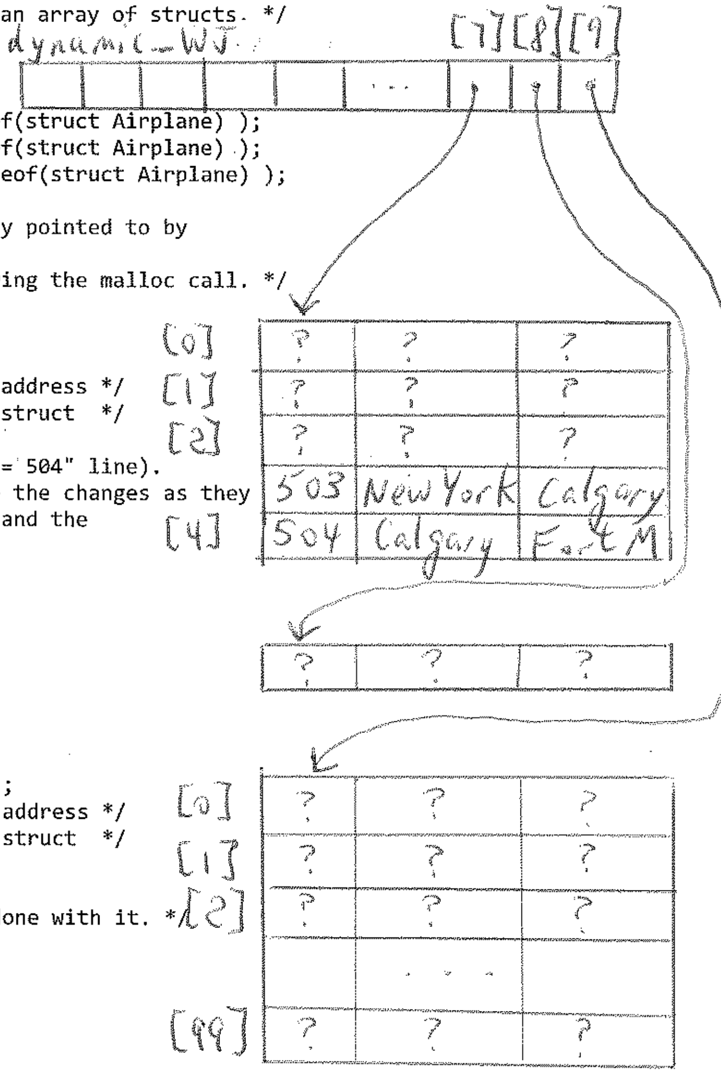
```
    dynamic_AC2[2].flight_number = 403;
    strcpy(dynamic_AC2[2].source, "Toronto");
    strcpy(dynamic_AC2[2].destination, "Los Angeles");
    print_airplane_PTR(&dynamic_AC2[2]);
    print_airplane(dynamic_AC2[2]);

    /* The following dynamic_AC2 code, if uncommented, would likely crash the
       program.  Why?
       Uncomment that set of lines, and set a breakpoint on the " = 404" line
       below to see what the variables look like in memory.  Expand the
       memory locations in the debugger, and click on "Step Over".
       Note that temp_show_me[] is simply used to help visualize memory
       (if dynamic_AC2 doesn't show enough information). */

    temp_show_me[0] = &dynamic_AC2[0];
    temp_show_me[1] = &dynamic_AC2[1];
    temp_show_me[2] = &dynamic_AC2[2];

    /*
    dynamic_AC2[3].flight_number = 404;
    strcpy(dynamic_AC2[3].source, "Toronto");
    strcpy(dynamic_AC2[3].destination, "Honolulu");
    print_airplane_PTR(&dynamic_AC2[3]);
    print_airplane(dynamic_AC2[3]);
    */
```



5

dynamic_WJ[10]

```
    /* Example 5:   Use an array of pointers to airplane structs.
                    EACH pointer can point to zero, one, or more dynamically allocated
                    airplane structs.  By "more", I mean an array of structs. */

    dynamic_WJ[0] = NULL;   /* zero planes */
    dynamic_WJ[1] = NULL;
    dynamic_WJ[7] = (struct Airplane *) malloc( 5 * sizeof(struct Airplane) );
    dynamic_WJ[8] = (struct Airplane *) malloc( 1 * sizeof(struct Airplane) );
    dynamic_WJ[9] = (struct Airplane *) malloc( 100 * sizeof(struct Airplane) );

    /* Let's populate the 5th plane in the 5-element array pointed to by
       dynamic_WJ[7].
       P.S.  Normally, you'd check the return code following the malloc call. */
    dynamic_WJ[7][3].flight_number = 503;
    strcpy(dynamic_WJ[7][3].source, "New York");
    strcpy(dynamic_WJ[7][3].destination, "Calgary");
    print_airplane_PTR(&dynamic_WJ[7][3]);   /* pass the address */
    print_airplane(dynamic_WJ[7][3]);        /* pass the struct  */

    /* Useful if setting a breakpoint below (say on the "= 504" line).
       Then, use the debugger's "Step Over" button to see the changes as they
       happen, in the (expanded) temp_show_me[] variable and the
       dynamic_WJ variable. */
    temp_show_me[0] = &dynamic_WJ[7][0];
    temp_show_me[1] = &dynamic_WJ[7][1];
    temp_show_me[2] = &dynamic_WJ[7][2];
    temp_show_me[3] = &dynamic_WJ[7][3];
    temp_show_me[4] = &dynamic_WJ[7][4];

    dynamic_WJ[7][4].flight_number = 504;
    strcpy(dynamic_WJ[7][4].source, "Calgary");
    strcpy(dynamic_WJ[7][4].destination, "Fort McMurray");
    print_airplane_PTR(&dynamic_WJ[7][4]);   /* pass the address */
    print_airplane(dynamic_WJ[7][4]);        /* pass the struct  */


    /*  It's a good idea to free the memory when you're done with it. */
```



```
    /*  ... */

    system("pause");
    return 0;
}


/* This function prints the airplane's info using call by value,
   but without using a pointer in the parameter list.  Thus, we
   copy the whole struct into the function. */
void print_airplane(struct Airplane  plane)
{
    printf("Not using a pointer:  Flight %d ", plane.flight_number);
    /* The following line also works:                                    */
    /* printf("Not using a pointer:  Flight %d ", (&plane)->flight_number); */
    printf("goes from %s to %s\n", plane.source, plane.destination);
}
```

function gets passed a struct

```
/* This function prints an airplane's info, using a pointer in the
   parameter list.  Thus, the airplane's struct isn't copied;  only its
   address is passed to the function.  Informally, we sometimes refer to
   this as "call by reference", but technically it's "call by value" using
   pointers. */
void print_airplane_PTR(struct Airplane * plane)
{
    printf("\n");
    printf("Using a pointer:      Flight %d ", plane->flight_number);
    /* printf("Flight %d ", (*plane).flight_number); */  /* also works */
    printf("goes from %s to %s\n", plane->source, plane->destination);
}
```

function gets passed a pointer