

Multidimensional Arrays I

- declaration:

```
#define N_ROW 3
#define N_COL 3

double darr[N_ROW], daarr[N_ROW][N_COL];
```

- use:

```
for (ii = 0; ii < N_ROW; ++ii)
    darr[ii] = ii;

for (ii = 0; ii < N_ROW; ++ii) {
    for (jj = 0; jj < N_COL; ++jj)
        daarr[ii][jj] = ii + jj;
}
```

Multidimensional Pointers I

- declaration:

```
double *dptr = NULL, **dpptr = NULL;
```

- usage:

```
dptr = mem_heap_double_ptr(N_ROW);
```

```
for (ii = 0; ii < N_ROW; ++ii)  
    dptr[ii] = ii;
```

```
dpptr = mem_heap_double_ptr_ptr(N_ROW);
```

```
for (ii = 0; ii < N_ROW; ++ii)  
    dpptr[ii] = mem_heap_double_ptr(N_COL);
```

```
for (ii = 0; ii < N_ROW; ++ii) {  
    for (jj = 0; jj < N_COL; ++jj)  
        dpptr[ii][jj] = ii + jj;  
}
```

Multidimensional Pointers II

- the `double_ptr` memory allocation function:

```
double *
mem_heap_double_ptr (int count)
{
    double *dval_ptr;

    dval_ptr = (double *) malloc(count * sizeof(double));
    if (dval_ptr == NULL) {
        printf("malloc failed to acquire memory: aborting...\n");
        abort( );
    }
    return dval_ptr;
}
```

Multidimensional Pointers III

- the `double_ptr_ptr` memory allocation function:

```
double **
mem_heap_double_ptr_ptr (int count)
{
    double **dval_pptr;

    dval_pptr = (double **) malloc(count * sizeof(double *));
    if (dval_pptr == NULL) {
        printf("malloc failed to acquire memory: aborting...\n");
        abort( );
    }
    return dval_pptr;
}
```

Aside I

- what is the difference between the following variables?

```
double daarr[N_ROW][N_COL], *darr_ptr[N_ROW], **dpptr;
```

- `daarr` is a “matrix” of dimension $N_ROW \times N_COL$: both the dimensions are fixed
- `darr_ptr` is an “array” of N_ROW many pointers where each of `darr_ptr[i]` potentially could be of different length, i.e. a “ragged array” of fixed length N_ROW , if you like: only one dimension fixed
- `dpptr` is potentially a “ragged array” of variable length: none of the two dimensions are fixed

Aside II

- does the following work?

```
double darr[N_ROW], *dptr_for_darr = NULL;

dptr_for_darr = darr;
printf("dptr_for_darr:\n");
for (ii = 0; ii < N_ROW; ++ii)
    printf("%g ", dptr_for_darr[ii]);
```

- yes it does, here we are accessing the elements of darr via the pointer (arithmetic) dptr_for_darr, things are fine

Aside III

- why *doesn't* the following work? here the pointer arithmetic fails, you get warnings and Bus Error, its pretty deep

```
double daarr[N_ROW][N_COL], **dpptr_for_daarr = NULL;

dpptr_for_daarr = daarr;
printf("dpptr_for_daarr:\n");
for (ii = 0; ii < N_ROW; ++ii) {
    for (jj = 0; jj < N_COL; ++jj)
        printf("%g ", dpptr_for_daarr[ii][jj]);
    printf("\n");
}
```

- if you can understand the pointer arithmetic output in the next slide its great, if not we'll have to come back to this when we have time or come ask me, if you are really curious, for the time being remember *not to do (something like) this!*
- the code for generating the output which is in file prog14.c is even more deep, come ask me if you want to really see through everything

Aside IV

- the promised esoteric output:

dpptr_for_daarr and daarr comparison:

```
dpptr_for_daarr = 0xbffff7f8, daarr = 0xbffff7f8
```

```
dpptr_for_daarr + 0 = 0xbffff7f8, daarr + 0 = 0xbffff7f8
```

```
0xbffff7f8 + 0 * sizeof(double *) = 0xbffff7f8
```

```
0xbffff7f8 + 0 * 3 * sizeof(double) = 0xbffff7f8
```

```
dpptr_for_daarr + 1 = 0xbffff7fc, daarr + 1 = 0xbffff810
```

```
0xbffff7f8 + 1 * sizeof(double *) = 0xbffff7fc
```

```
0xbffff7f8 + 1 * 3 * sizeof(double) = 0xbffff810
```

```
dpptr_for_daarr + 2 = 0xbffff800, daarr + 2 = 0xbffff828
```

```
0xbffff7f8 + 2 * sizeof(double *) = 0xbffff800
```

```
0xbffff7f8 + 2 * 3 * sizeof(double) = 0xbffff828
```

- summary: a two-dimensional array is stored as a one-dimensional array internally, see that?

Multidimensional Pointers IV

- so what is a double ** variable anyway? consider the following:

```
double dd = 1.11, *ddp = &dd, **ddpp = &ddp;
```

dd

```
name: ``dd``
address: 0xbffff848
value: 1.11
type: double
```

ddp

```
name: ``ddp``
address: 0xbffff850
value: 0xbffff848
type: double *
```

ddp

```
name: ``ddp``
address: 0xbffff850
value: 0xbffff848
type: double *
```

ddpp

```
name: ``ddpp``
address: 0xbffff854
value: 0xbffff850
type: double **
```

Multidimensional Pointers V

- so let us observe:
 - `&dd` is same as `ddp`
 - `*ddp` is same as `dd`
 - `&ddp` is same as `ddpp`
 - `*ddpp` is same as `ddp`
 - `**ddpp` is same as `*ddp` which is same as `dd`
- does `&&dd` make sense?
 - no, because `&dd` is `0xbffff848` i.e. an address but what is an address of an address?
- but `&ddp` does make sense
 - `ddp` (which has value `&dd` i.e. `0xbffff848`) is a variable and it indeed has an address!

Pass by Value/Reference I

- lets consider this:

```
void
foo_dd (double dd)
{
    printf("in foo_dd: val = %g, add = %p\n", dd, &dd);
}

/* some code */

double dd = 1.11;
printf("dd: val = %g, add = %p\n", dd, &dd);
foo_dd(dd);
```

- the output of the above:

```
dd: val = 1.11, add = 0xbffff848
in foo_dd: val = 1.11, add = 0xbffff778
```

- why are the addresses &dd different in and outside foo_dd()? think about pass by value, and remember names don't matter!

Pass by Value/Reference II

- lets consider this:

```
void
foo_ddp (double *ddp_arg)
{
    printf("in foo_ddp: val = %p, add = %p, val_at_val = %g\n",
           ddp_arg, &ddp_arg, *ddp_arg);
}

/* some code */

double dd = 1.11, *ddp = NULL;
ddp = &dd;
foo_ddp(&dd);
foo_ddp(ddp);
```

- the output from above:

```
ddp: val = 0xbffff848, add = 0xbffff850, val_at_val = 1.11
in foo_ddp: val = 0xbffff848, add = 0xbffff7b8, val_at_val = 1.11
in foo_ddp: val = 0xbffff848, add = 0xbffff7b8, val_at_val = 1.11
```

Pass by Value/Reference III

- things to note:
 - we can call `foo_ddp` both as `foo_ddp(&dd)` and `foo_ddp(ddp)`
 - * in the call `foo_ddp(&dd)` we are passing `dd` by reference to `foo_ddp()`
 - * in the call `foo_ddp(ddp)` we are passing `ddp` by value to `foo_ddp()`
 - in either of the above cases the argument `double *ddp_arg` which is a local variable gets the value `&dd`
 - note `ddp_arg` and `ddp` has different addresses, precisely because `ddp` is passed by value i.e. both `ddp_arg` and `ddp` has the same value `&dd` but are two different variables

Pass by Value/Reference IV

- now consider is the code:

```
void
foo_ddpp (double **ddpp)
{
    printf("in foo_ddpp: val = %p, add = %p, val_at_val = %p, " \
          "val_at_val_val = %g\n", ddpp, &ddpp, *ddpp, **ddpp);
}

/* some code */

double dd = 1.11, *ddp = NULL, **ddpp = NULL;
ddp = &dd;
ddpp = &ddp;
printf("foo_ddpp: val = %p, add = %p, val_at_val = %p, " \
      "val_at_val_val = %g\n", ddpp, &ddpp, *ddpp, **ddpp);
foo_ddpp(&ddp);
```

- note deliberately the name of the argument of `foo_ddpp()` is chosen to be `ddpp` not to confuse you but just remind you names don't matter and you may not need to invent names all the time while writing functions!

Pass by Value/Reference V

- can we explain the output, remember the difference between call by reference and the concept of local variables:

```
foo_ddpp: val = 0xbffff850, add = 0xbffff854,  
val_at_val = 0xbffff848, val_at_val_val = 1.11
```

```
in foo_ddpp: val = 0xbffff850, add = 0xbffff7b8,  
val_at_val = 0xbffff848, val_at_val_val = 1.11
```

Pass by Value/Reference VI

- so here is what we have in summary, suppose we have the following:

```
double bar, *bar_ptr = &bar;
```

```
void  
foo_ptr (double *ddp);
```

```
void  
foo_ptr_ptr (double **ddpp);
```

- in `foo_ptr(&bar)`, we pass `bar` by reference and hence `foo_ptr` is allowed to change `bar`
 - in `foo_ptr(bar_ptr)`, we pass `bar_ptr` by value and hence `foo_ptr` is allowed to change whatever `bar_ptr` points to and not `bar_ptr` itself
 - in `foo_ptr_ptr(&bar_ptr)`, we pass `bar_ptr` by reference and hence `foo_ptr_ptr` is allowed to change `bar_ptr`
- so if you want a function to change the value of “something” pass it by reference otherwise pass it by value

Code Files

prog14.c