

601.220 Intermediate Programming

Dynamic 2D arrays & const w/ pointers

Outline

- Dynamic allocation of 2D arrays
- Pointer types & `const`

Two dimensional arrays - static allocation review

- `int a[5][3];` creates array with 5 rows, 3 columns each
- `a[2][1] = 17;` stores value in 3rd row, 2nd column
- array is stored sequentially in memory, in row order
- `*(a + 10)` is the same as `a[3][1]`

Dynamically-allocated two dimensional arrays - use a 1D array of items and “fake” two dimensions

- `int *a = malloc(sizeof(int) * num_rows * num_cols);`
- Use a single array with one dimension
 - Convert `[row][col]` indexing to `[row * num_cols + col]`, and back
 - `a[7] = 17;`
 `// a[7] means a[2][1] for num_cols==3,`
 `// since 7 == 2*3 + 1`
- `free(a);`

Dynamically-allocated two dimensional arrays - double (**) memory allocation

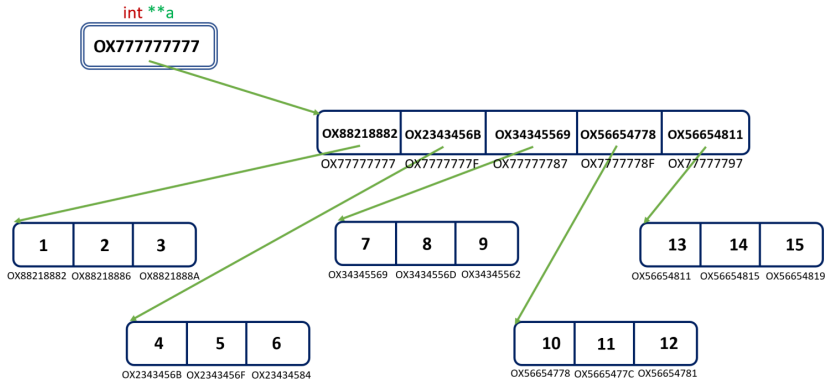
- Use a 1D array of pointers to item arrays ““ `int **a = malloc(sizeof(int) * num_rows);`

```
for (int i = 0; i < num_rows; i++) { a[i] = malloc(sizeof(int) *  
num_cols); }
```

```
a[2][1] = 17; // this works!
```

```
for (int i = 0; i < num_rows; i++) { free(a[i]); } free(a); // note  
this one last free! ““
```

5 by 3 2D Array using 1D array of pointers



Decomposing a dynamically-allocated 2D array

- given `int **a` has been fully allocated as in prior slides
- `a[i]` is of type `int *`, for valid values of `i`
- represents one row in the 2D array
- can be used in the same ways a 1D array variable can be used

Rows of a 2D array as 1D arrays

```
// rowProcessing.c:
#include <stdio.h>

void printFloats(float fray[], int count) {
    for (int i = 0; i < count; i++)
        printf("%.1f ", fray[i]);
}

int main(void) {
    float fra[5][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}, {13, 14, 15}};
    for (int r = 0; r < 5; r++) {
        printFloats(fra[r], 3);
        printf("\n");
    }
    return 0;
}
```

```
$ gcc -std=c99 -Wall -Wextra -pedantic rowProcessing.c
$ ./a.out
1.0 2.0 3.0
4.0 5.0 6.0
7.0 8.0 9.0
10.0 11.0 12.0
13.0 14.0 15.0
```


Non-uniform (jagged) 2D arrays - how to

It is possible for rows in a dynamically allocated 2D array to be different sizes

```
// create 10 pointers to rows
int **ra2d = malloc(sizeof(int*) * 10);

// create rows with sizes 1 to 10
for (int i = 0; i < 10; i++) {
    ra2d[i] = malloc(sizeof(int) * (i + 1));
}
```

Non-uniform (jagged) 2D arrays - pitfalls

- must remember to free the memory for each row, and then the ra2d itself
- need to be careful when using since rows are different sizes!
- might want a parallel array to hold the length of each row

Using const

- `const` means “constant” and prevents modification of the element to which it is applied
- Recall: to make a variable non-modifiable: `const int num`
- if local variable, it must be initialized when declared
`const int num = 10;`
- if parameter variable, it cannot be changed within the function
- can pass a non-const variable to a const parameter (more restrictive)
- cannot pass a const variable to a non-const parameter
- `const` can be used at different points in pointer type declarations, each with different meanings

Pointers, arrays and const - protect the data pointed to

- To make a (mutable) pointer to const (non-modifiable) data:
`const int * iptr`
- prevents changing contents of the pointed to memory
`*iptr = 10; // not allowed`
`iptr = # // allowed for int variable num`
- similar to `const int iray[]` as a function parameter
`iray[0] = 10; // not allowed`
`iray = malloc(sizeof(int)); // allowed`
- only copy of the calling variable is affected, not the original

Pointers, arrays and const - protect the pointer

- To make a const (non-modifiable) pointer:
`int * const iptr`
- similar to `int iray[10]`; as a local variable
- if not a parameter, must set when declaring:
`int * const iptr = #`
- prevents assignments to change (the address stored in) `iptr` or `iray`
`iptr = &other; // not allowed`
`iray = b; // not allowed`

Pointers, arrays and const - double const

- To make a const ptr to const data:
`const int * const iptr`
- doesn't allow changes to pointer variable itself, or the memory it points to

`*iptr = 10; // not allowed`

`iptr = # // not allowed`

- similar to `const int iray[] = { 1, 2, 3 }; as local variable`

`iray[0] = 10; // not allowed`

`iray = malloc(sizeof(int)); // not allowed`

Read declarations from right to left to get them correct!