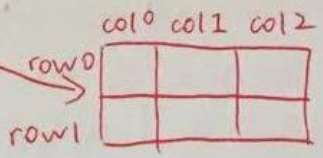
Multidimensional Arrays

Oregon State University
College of Engineering

2D Static array

Beginning address of where array is in memory

- data_type array_name[rows][cols];
  - int array[2][3];
  - int array[4][2][3];
  - int array[2][4][2][3];



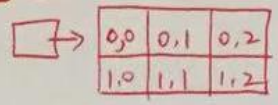
You can have arrays of more than 2 dimensions

- What are examples of these?
  - 2-D – Matrices, Spreadsheet, Minesweeper, Battleship, etc.
  - 3-D – Multiple Spreadsheets, (x, y, z) system
  - 4-D – (x, y, z, time) system

*2D static array has contiguous memory on the stack

Initializing 2-D Arrays

(static)



- **Declaration:** int array[2][3] = {{0,0,0},{0,0,0}};
- **Individual elements:** array[0][0]=0;
  array[0][1]=0; array[0][2]=0;
  array[1][0]=0; array[1][1]=0;
  array[1][2]=0;

this is not good if your array is large!

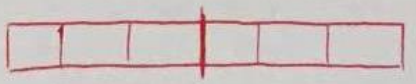
- **Loop:** Better choice

C/C++ is row major, the memory is laid out by row

This is how it looks like in memory:

rows for outer loop
cols for inner loop

```
for(i = 0; i < 2; i++)
    for(j = 0; j < 3; j++)
        array[i][j]=0;
```



*The number of cols is called "stride", in this case, it is 3.

- Why do we need multiple brackets?

To specify each element in the 2D array. To interpret array[i][j], it is:

from starting point of the array + (i * stride) + j

Reading/Printing 2-D Arrays

Oregon State University
College of Engineering

- Reading Array Values

```
for(i = 0; i < 2; i++)
  for(j = 0; j < 3; j++) {
      cout << "Enter a value for " << i << ", " << j << ":
  ";
          cin >> array[i][j];
      }
```

specify rows

specify cols

Both start at 0

Use double brackets to access content

- Printing Array Values

```
for(i = 0; i < 2; i++)
  for(j = 0; j < 3; j++)
        cout << "Array: " << array[i][j] <<
endl;
```

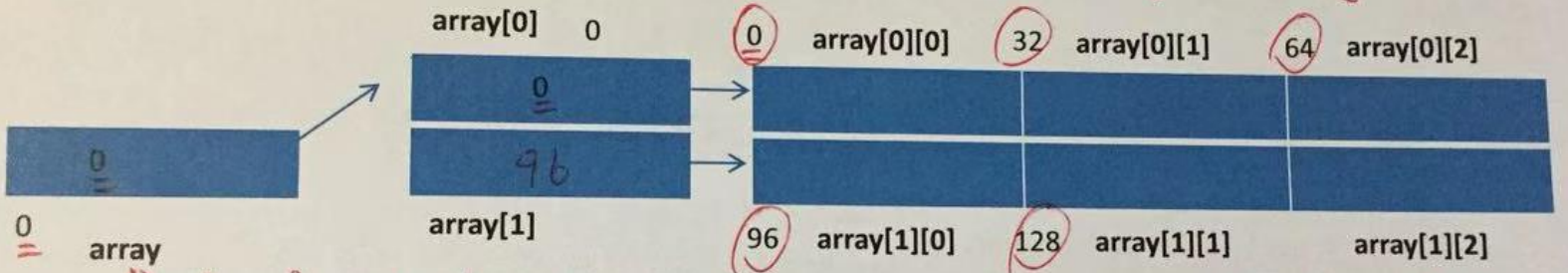
stack heap double pointers

Static vs. Dynamic 2-D arrays...

Oregon State University College of Engineering

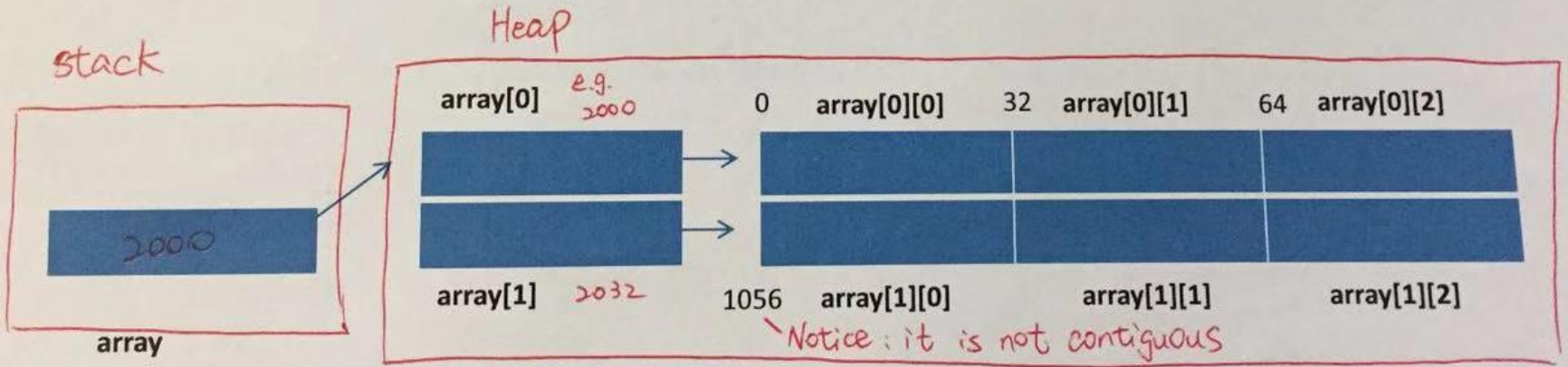
stack e.g. int array [2][3]

This is done at compile time



↑ self-ref constant pointer, which means array, &array, array[0], &array[0][0] are the same

In this case, 0.



e.g. int ** array;

array = new int*[2];

array[0] = new int[3];
array[1] = new int[3];

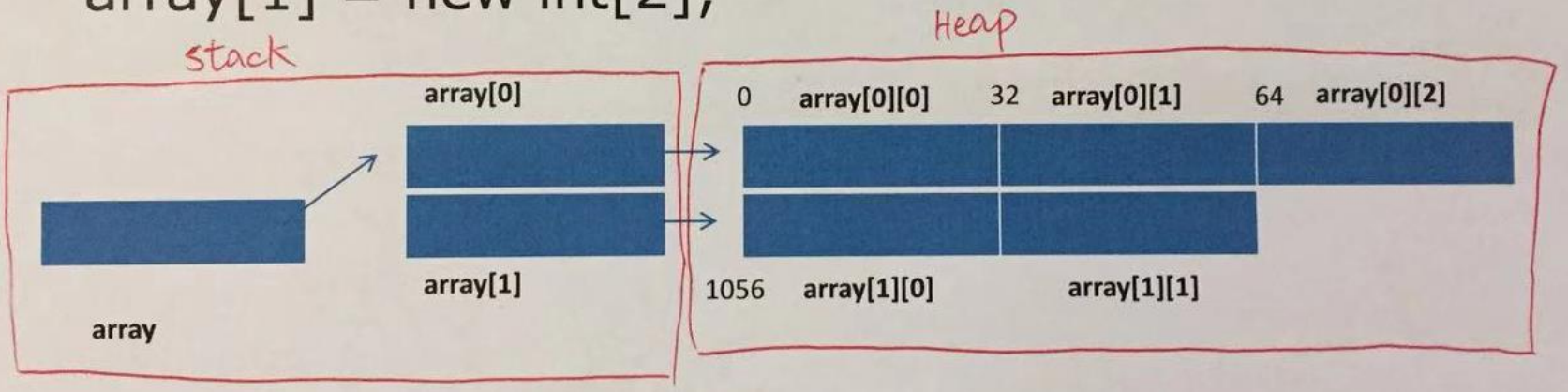
Jagged Arrays

Oregon State University
College of Engineering

at compile time, you know the # of rows

num of rows

```
int *array[2];
array[0] = new int[3];
array[1] = new int[2];
```



Passing a 2-D Array (Static)

Oregon State University
College of Engineering

```
int main() {
   int array[5][5];
   ...
   pass_2darray(array);
   ...
}
void pass_2darray(int a[5][5]) {
   cout << "Array at zero: " << a[0][0] << endl;
}
```

stride (# of cols)

You can pass both, but not necessary

OR

```
void pass_2darray(int a[][5]) {
   cout << "Array at zero: " << a[0][0] << endl;
}
```

stride (# of cols) is necessary

Passing a 2-D Array (Dynamic)

Oregon State University
College of Engineering

```
int main() {
   int **array;
   …
   pass_2darray(array);
   …
}
void pass_2darray(int *a[]) {
    cout << "Array at zero: " << a[0][0] << endl;
}
```

OR

```
void pass_2darray(int **a) {
   cout << "Array at zero: " << a[0][0] << endl;
}
```

Passing a 2-D Array (Dynamic)

Oregon State University
College of Engineering

```
int main() {
   int *array[2];
   ...
   pass_2darray(array);
   ...
}
void pass_2darray(int *a[]) {
    cout << "Array at zero: " << a[0][0] << endl;
}
```

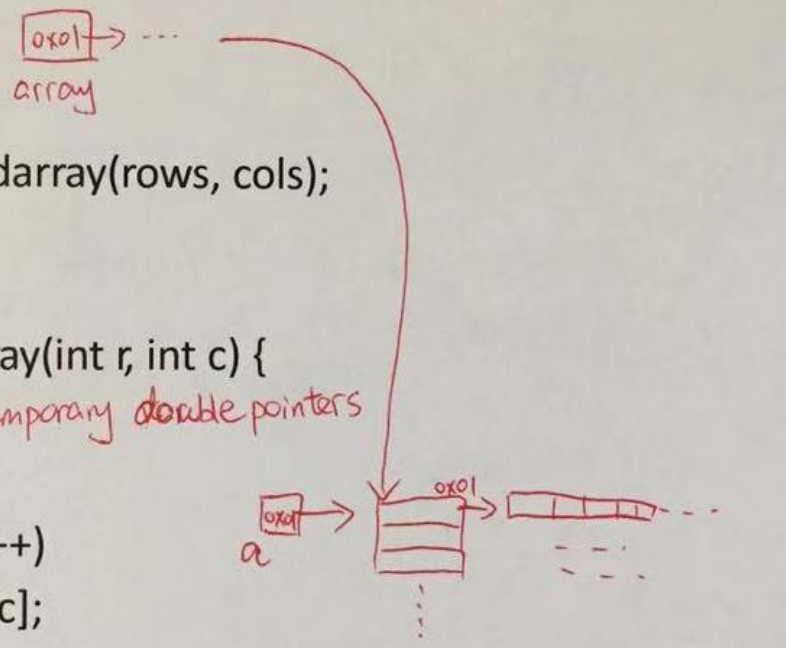
OR

```
void pass_2darray(int **a) {
   cout << "Array at zero: " << a[0][0] << endl;
}
```

Create 2-D Array in Functions

Oregon State University
College of Engineering

```
int main() {
  int **array;
  ...
  array = create_2darray(rows, cols);
  ...
}
int **create_2darray(int r, int c) {
  int **a;
  a = new int*[r];
  for(int i=0; i<r; i++)
    a[i] = new int[c];
  return a;
}
```



Can't put these onto stack since once you leave the function, the stack memory here will be popped off

return the memory address a is pointing to

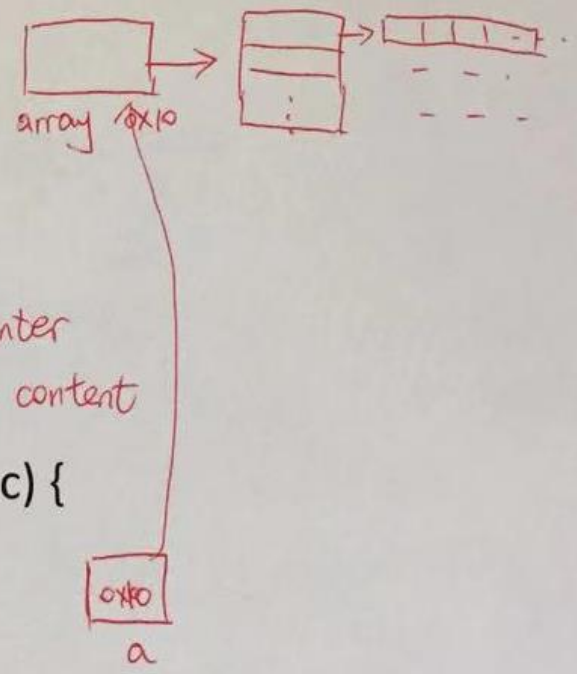
Create 2-D Array in Functions


```
int main() {
  int **array;
  ...
  create_2darray(&array, rows, cols);
  ...
}
void create_2darray(int ***a, int r, int c) {
  *a = new int*[r];
  for(int i=0; i<r; i++)
    (*a)[i] = new int[c];
}
```

pass the address of the double pointer since we are trying to change its content

dereference

1st



Create 2-D Array in Functions

Oregon State University
College of Engineering

```
int main() {
   int **array;
   ...
create_2darray(array, rows, cols);
   ...
}
void create_2darray(int **&a, int r, int c) {
   a = new int[r];
   for(int i=0; i<r; i++)
      a[i] = new int[c];
}
```

pass the double pointer as reference

Only free memory on the heap

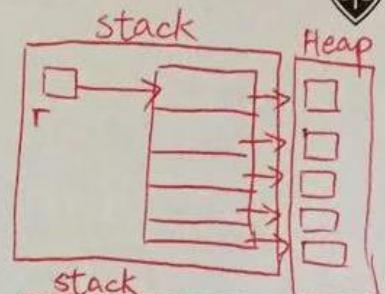
How does freeing memory work?

Oregon State University
College of Engineering

```
int *r[5], **s;

for(int i=0; i < 5; i++)
   r[i]=new int;
for(int i=0; i < 5; i++)
   delete r[i];
```

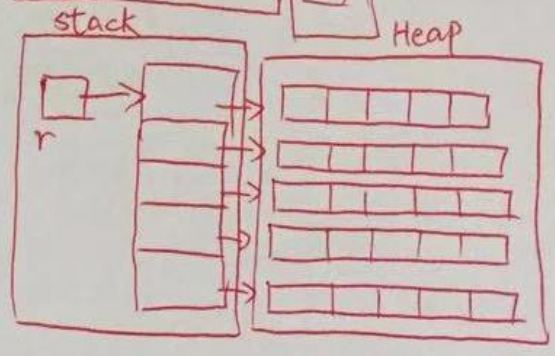
← one int per array



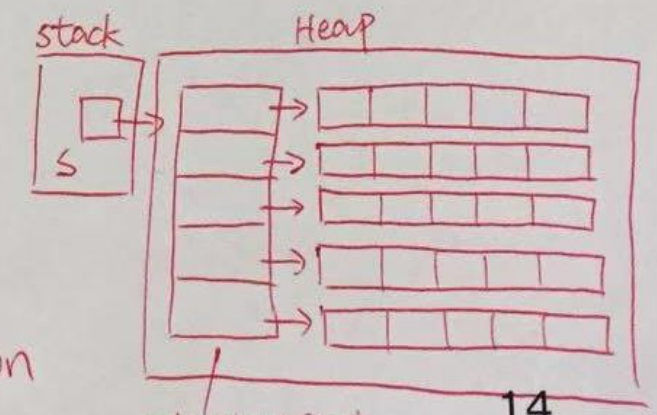

```
for(int i=0; i < 5; i++)
   r[i]=new int[5];
for(int i=0; i < 5; i++)
   delete [] r[i];
```

← 5 int per array

need this to delete whole array




```
s=new int*[5];
for(int i=0; i < 5; i++)
   s[i]=new int[5];
for(int i=0; i < 5; i++)
   delete [] s[i];
delete [] s;
```



create these first, delete these last

☆ Deletion has the reverse order of creation