Pointers to Pointers

Computer Science Department

Eastern Washington University

Yun Tian (Tony) Ph.D.

Outline for Today

- pointers to pointers
- demo code

2D arrays

- Now we access values in 2D array using:
  - *(*(multi + 3) + 1) returns the value at row 3 and column 1, which is char ‘8’.
  - We get the same value as multi[3][1].

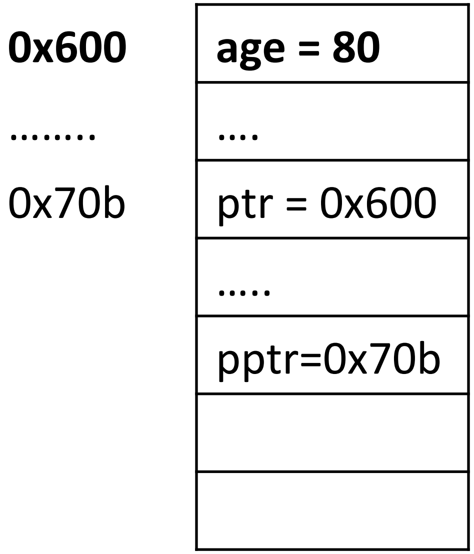
```
for (row = 0; row < ROWS; row++)
 {
      for (col = 0; col < COLS; col++)
      {
         printf("\n%d  ",multi[row][col]);
         printf("%d ",  *( *(multi + row) + col) );
      }
  }
```

Pointers to Pointers

- **int **pptr;**
- Make sense that
  - * has right to left associativity.
  - int ** pptr actually is int* (*pptr)
  - Does int* (*pptr) looks like int* Q ?
    - Q is (*pptr) in expression above.
    - Q could be considered as pointer variable, hold the address to an integer value.

Pointers to Pointers

- **int **pptr;**
- **int age = 80;**
- **int *ptr = &age;**
- **pptr = &ptr;**
- **……**
- **To access the value of age,**
  - **We use **pptr**

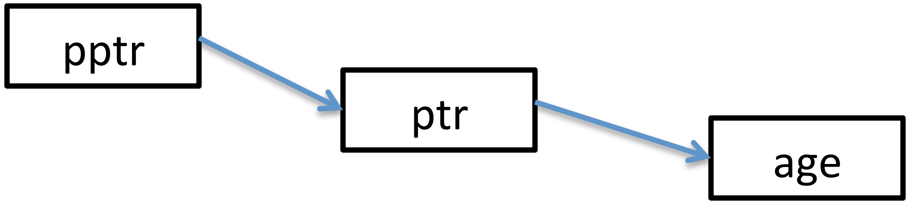


0x600	**age = 80**
……..	….
0x70b	ptr = 0x600
	…..
	pptr=0x70b

Pointers to Pointers

- **int **pptr;**
- **int age = 80;**
- **int *ptr = &age;**
- **pptr = &ptr;**
- **To access the value of age,**
  - **We use **pptr**

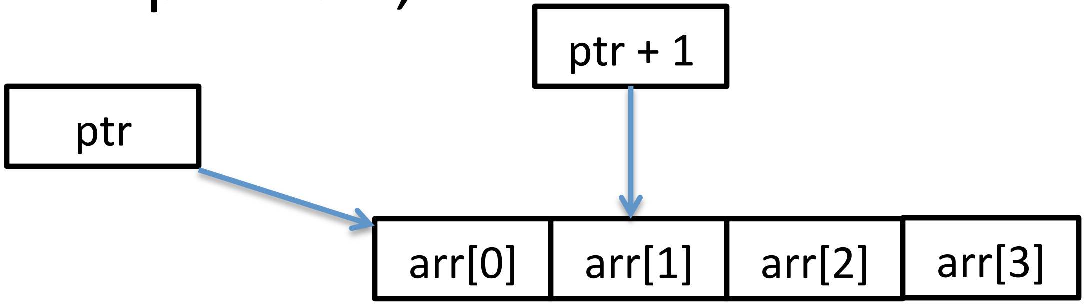
0x600	**age = 80**
……..	….
0x70b	ptr = 0x600
	…..
	pptr=0x70b



Pointers

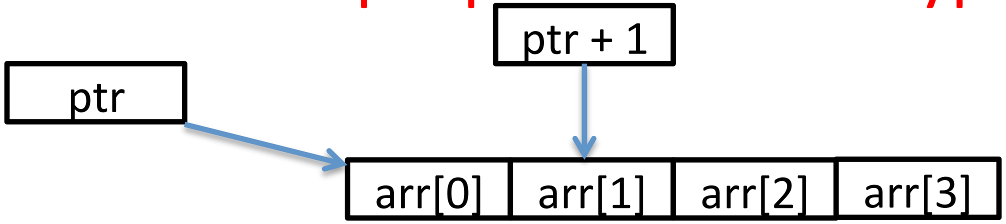
- Pointer could points to an array of contiguous elements.
- What we have learned about pointers?

```
int arr[4] = { 7, 8, 9, 10};
int *ptr = arr;
```



Pointer

- int * ptr ➔ int (* ptr) → after dereference ptr, we get integer type,
  - Means ptr points to an array of integer numbers.
    - *(ptr + 0) is type of int.
    - ptr[1], *(ptr + 1) is type of int.
    - “points to” also called “holds the address of”
  - Each element ptr points to is of type of int.

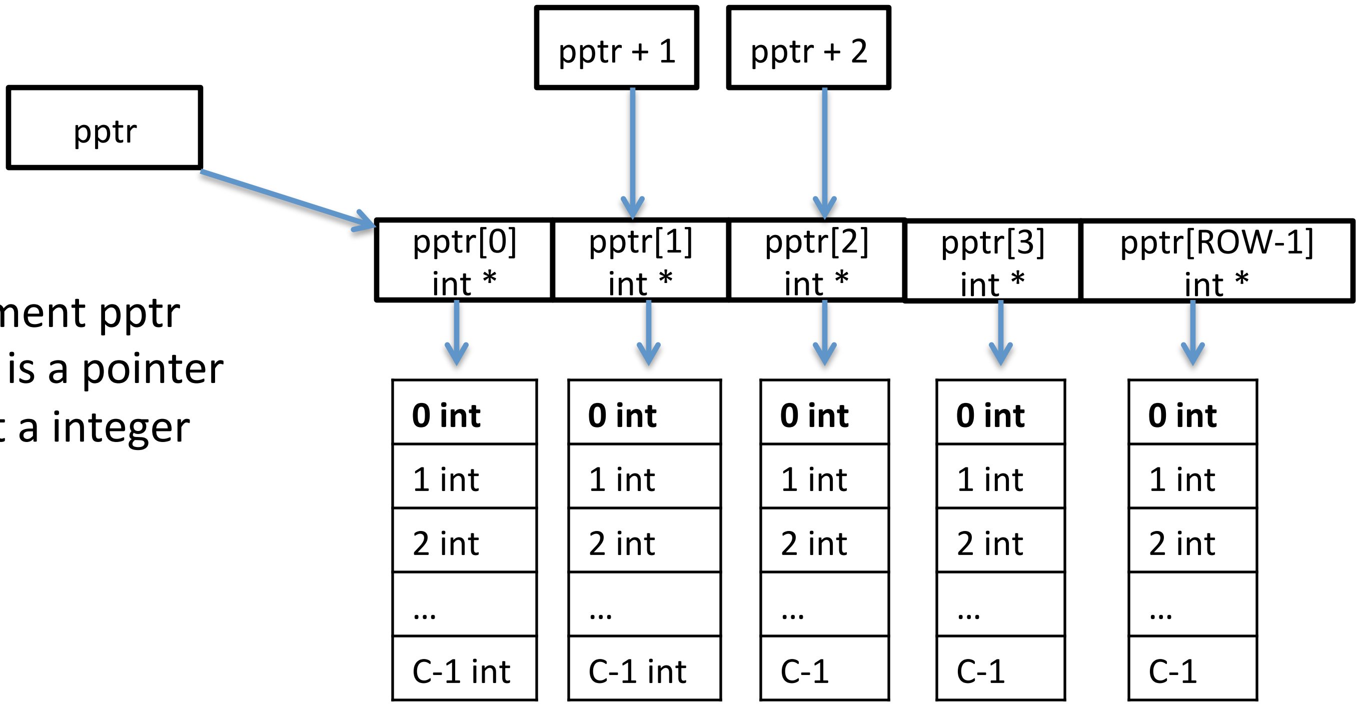


Pointer to Pointer

- int **pptr ➔ (int *) (*pptr)
  - If we think (int *) as type Q, then we rewrite the original expression to Q *pptr.
  - What does ‘Q *pptr’ mean?
    - *(pptr + 0) is type of Q.
    - pptr[1], *(pptr + 1) is type of Q.
  - Each element pptr points to is of type of Q.
  - Type of Q is (int *), i.e. pointer to an array of integers.

Pointer to Pointer

- int **pptr ➔ (int *) (*pptr)
  - – Each element pptr points to is of type of (int *).



pptr

pptr + 1

pptr + 2

pptr[0] int *	pptr[1] int *	pptr[2] int *	pptr[3] int *	pptr[ROW-1] int *
0 int	**0 int**	**0 int**	**0 int**	**0 int**
1 int	1 int	1 int	1 int	1 int
2 int	2 int	2 int	2 int	2 int
…	…	…	…	…
C-1 int	C-1 int	C-1	C-1	C-1

Each element pptr points to is a pointer itself, not a integer yet.

Pointer to Pointer

- **initialization and memory allocation**
  - Same as in Java, you have to explicitly allocate space for cells that pptr points to. (Rows)
  - And space for pptr[i] points to. (Columns)
  - Otherwise you will get segmentation fault, like the null pointer exception in Java.
- See the attached Demo Code
- Can you see the difference between a static 2D array and a dynamic 2D array represented with pointer to pointers?

Pointers to Pointers

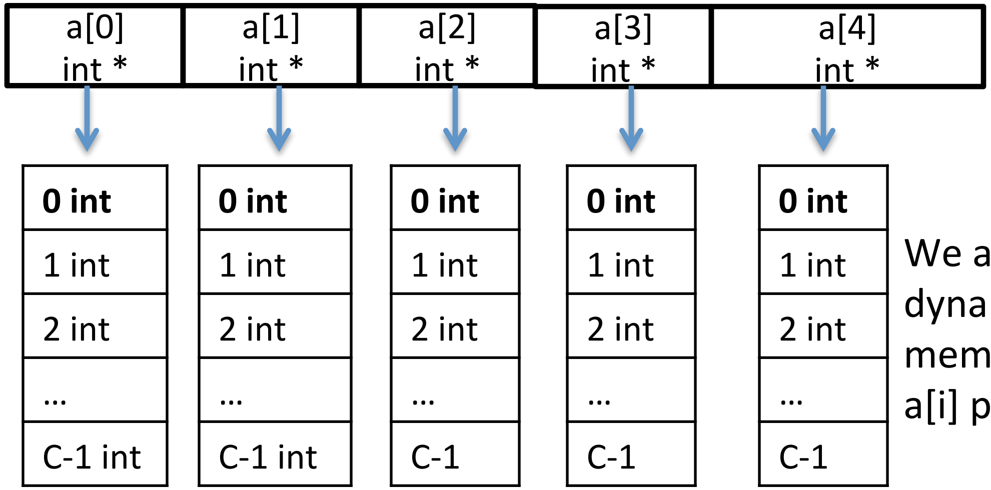
- **int **pptr;**
- pptr[10][3] returns the value at row 10 column 3.
  - We can use pptr as a 2D array name.
  - Same as * (*(pptr + 10) + 3)
  - Same as * (pptr[10] + 3) // this only works for double pointers, not work for 2D array names.

Arrays of Pointers

- char * names[100] → (char *) names[100]
  - Each element **names[i]** is a char pointer or a string.
  - E.g. Useful when we know the maximum number of students we have.
    - **names[i]** holds the name for each student.

Arrays of Pointers

- **int * a[5]; ➔ (int *) a[5]**
  - Each element **a[i]** is a pointer that points to one integer or an array of integers.



Number of rows are statically allocated.

We are still required to dynamically allocate memory for the space that a[i] points to. (Columns)

Arrays of Pointers

```
#define MaxNameLen 50
…..
//
void readNames( char * names[],  int numStu )
{
        int i;
        for( i = 0; i < numStu; i ++ )
        {
                if ( names[i] == NULL )
                        names[i] = (char *)malloc( MaxNameLen * sizeof ( char ));
                fgets( name[i], MaxNameLen, stdin );
        }
 }
```

Summary

- pointer to pointer (double pointers)
- we use double pointers as dynamic 2D arrays.
- different from or similar to static 2D array,
  - double points **pptr, inside the row that pptr[i] points to, cells are contiguous, same as static 2D array.
  - Unlike 2D static array, address of two adjacent rows might not be contiguous.
    - The end of row that pptr[i] points to might not be the beginning of pptr[i+1] points to.
- Arrays of pointers.

Summary

- Extremely cautious about, ((int *p; int **pptr)
  - Programmer has to allocate memory that a pointer points to, before you use the memory that the pointer points to.
    - You have had the pointer point to a meaningful location.
    - For pptr, you have to allocate two pieces of them.
  - Where you allocate the memory?
    - You allocate inside a function, then return that piece of memory that can be used in main(),
    - You allocate the memory in main(), then you pass the initialized pptr or ptr into another function.

Summary

- Good practices, with ((int *p; int **pptr)
  - When defining pointers, initialize it to NULL immediately, before you allocate memory for them.
    - int *p = NULL; int ** pptr = NULL;
  - Then, in each function that takes a pointer as a parameter, ask yourself:
    - Has the pointer already pointed to a meaningful memory chunk?
    - Or am I supposed to allocate memory inside the function for the pointer, and return it as a dynamic array?
  - Also, you can always check before you dereference,
    - if(p != NULL)   printf(“%d”, *p);