Arrays of pointers

ESC101: Fundamentals of Computing

Nisheeth

The Golden Rules of Pointers

RULE 1: All pointers store addresses, take 8 bytes to store

Does not matter whether pointer to variable, to array, to another pointer etc

RULE 2 (Reference): &a gives address of variable a

Does not matter whether variable a is char, long, or even a pointer variable

Special case for static array variables – next slide

RULE 3: (Dereference): Whenever expression expr generates an address, *(expr) gives value stored at that address

RULE 4: (Arithmetic): Pointer arithmetic is w.r.t datatype

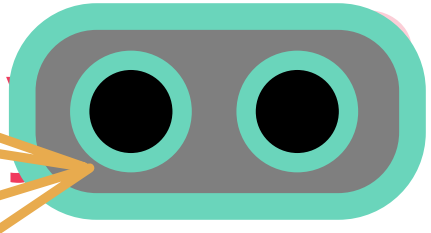
char* arithmetic w.r.t. 1 byte blocks, int* w.r.t. 4 byte blocks, double* 8 bytes

RULE 5: Name of array points to first element of array

Does not matter whether dynamic (e.g., malloc-ed) array or static array

The Cur

a, b, c all point to their respective first elements ☺



I will hide the location of the pointer a and b from you since I store these pointers secretly in a location called the **symbol table** – no access!!

Three types

Static arra

You can modify the pointer c by saying c++ but I will not allow you to say things like a++, b++. I also dont allow you to free/realloc a and b ☺

od",&n); int b[n];

s int *c = (int*)malloc(n * sizeof(int));

For static arrays (fixed/variable length) sizeof() gives total size of array. However, for malloc-ed arrays, sizeof(c) just gives 8, the space required to store the pointer c ☹

&c gives us the address where the pointer c is stored ☺

&a just gives us the address of first element a[0] again ☹

&b just gives us the address of first element b[0] again ☹

The getline function: Revisited

Read a single line of text from input (i.e. till '\n')

Uses realloc-like methods to expand the char

For char array, we need a malloc ed array for this reason

int len = 11; // I only expe ered

char *str =

getline(&st

If user input doesn't fit inside original array, str will contain pointer to expanded array, len will be length of new array

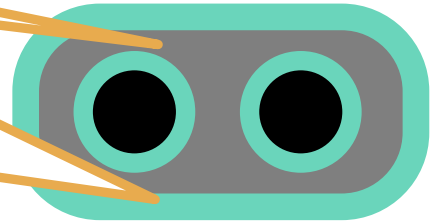
char **ptrstr = &str;

getline(ptrstr, &len, st

Inception? ☺

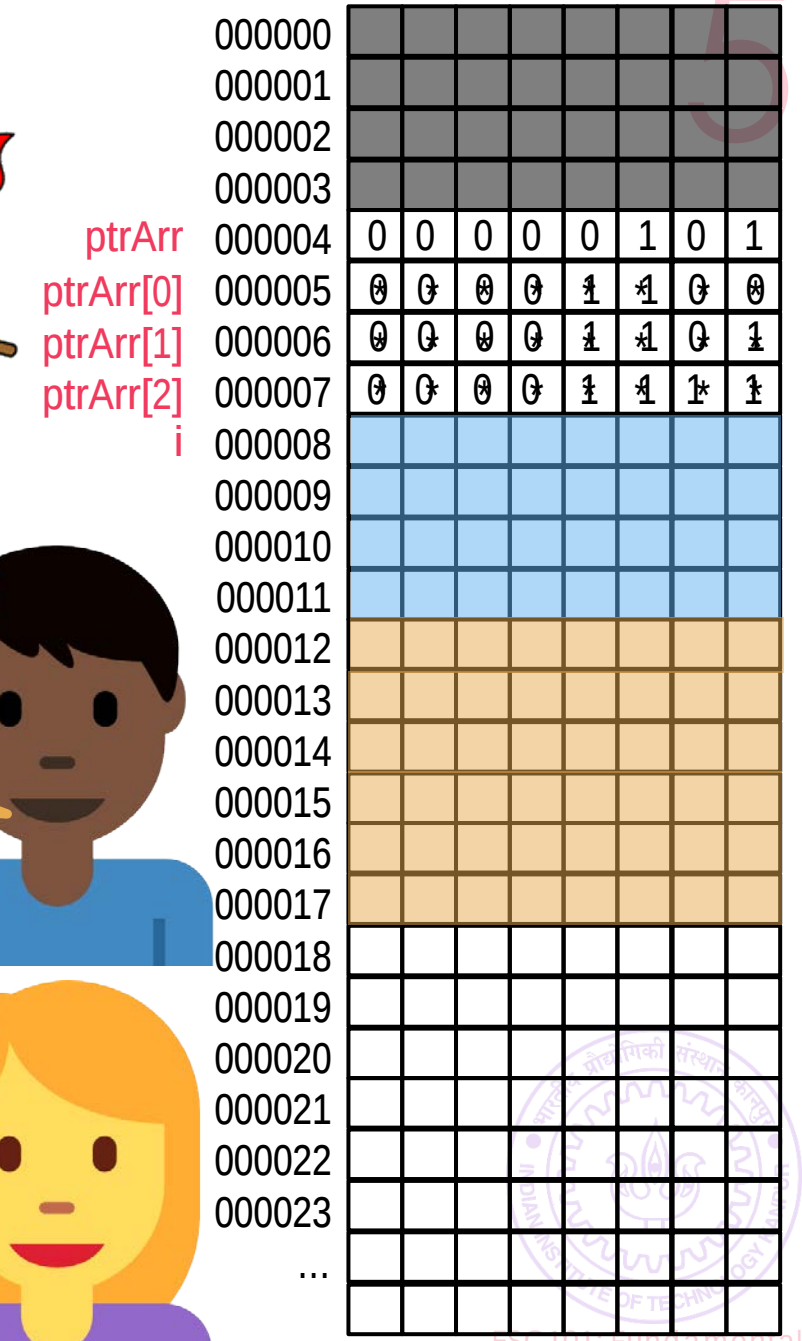
Pointer to a pointer simply stores the address of a pointer variable

printf("%ld",*ptrstr) will print address of first char in str
printf("%c",**ptrstr) will print the first char in str
printf("%s",*ptrstr) will print entire string str



WARNING: len may be larger than length of input + 1
Get actual length of input using strlen() from string.h

Note: ptrArr, ptrArr[0], ptrArr[1], ptrArr[2], each a pointer, will take 8 bytes to store – figure addressing is not accurate (shows one byte for each)



Note: Array of pointers is equivalent to pointer of pointers

Alternate, dynamic way to declare array of pointers

```
char **ptrArr = (char**)malloc(3*sizeof(char*));
```


Very useful in programs where we have to create a known number of arrays of unknown length

However, in this case we should also free pointer array by writing

```
free(ptrArr);
```

An Example: Printing all substrings

Read a string and create an array containing all its substrings (i.e. contiguous).

Display the substrings (note: non-unique substrings allowed, i.e., a substring may appear multiple times).

Input: ESC

Output:
E
ES
ESC
S
SC
C

An Example: Printing all substrings

What are the possible substrings for a string having length len?

For $0 \leq i < len$ and for every $i \leq j < len$, consider the substring between the i^{th} and j^{th} index.

An idea: Allocate a 2D char array having $\frac{len \times (len+1)}{2}$ rows

(why? And how many columns to use?)

Now copy the substrings into different rows of this array

Let is use array of pointers or pointer of pointers to do the above

Solution: Version 1

```
int len, i, j, k=0, nsubstr;
char st[100], **substrs;
scanf("%s",st);
len = strlen(st);
nsubstr = len*(len+1)/2;
substrs = (char**)malloc(sizeof(char*) * nsubstr);
for (i=0; i<nsubstr; i++)
    substrs[i] = (char*)malloc(sizeof(char) * (len+1));

for (i=0; i<len; i++){
    for (j=i; j<len; j++){
        strncpy(substrs[k], st+i, j-i+1);
        k++;
    }
}
for (i=0; i<k; i++)
    printf("%s\n",substrs[i]);
```

```
for (i=0; i<k; i++)
    free(substrs[i]);
free(substrs);
```

Wasted too much space..

E	'\0'		
E	S	'\0'	
E	S	C	'\0'
S	'\0'		
S	C	'\0'	
C	'\0'		

Solution: Version 2

```
int len, i, j, k=0,nsubstr; char st[100], **substrs;
scanf("%s",st);
len = strlen(st);
nsubstr = len*(len+1)/2;
substrs = (char**)malloc(sizeof(char*) * nsubstr);

for (i=0; i<len; i++)
   for (j=i; j<len; j++){
      substrs[k] = (char*)malloc(sizeof(char) * (j-i+2));
      strncpy(substrs[k], st+i, j-i+1);
      k++;
    }
for (i=0; i<k; i++)
    printf("%s\n",substrs[i]);
```

```
for (i=0; i<k; i++)
    free(substrs[i]);
free(substrs);
```

This version uses much less memory compared to version 1