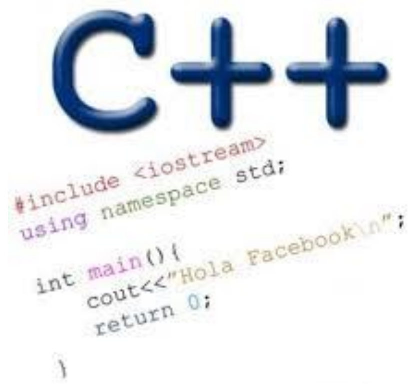
POINTER ARITHMETIC
ARRAYS, POINTERS AND STRUCTS

Problem Solving with Computers-I

https://ucsb-cs16-sp17.github.io/



```
#include <iostream>
using namespace std;

int main(){
    cout<<"Hola Facebook\n";
    return 0;
}
```

Announcements

- Midterm next week Wed (05/12)
- All material covered in labs00-lab05 (including lab05). Key topics: file IO (only those aspects covered in lab03), Pointers, arrays, pointers and structs, function call mechanics (pass by value, reference and address), arrays of structs
- All lecture material until Monday’s lectures

Review: Pointer assignment: Trace the code

```
int x=10, y=20;
int *p1 = &x, *p2 =&y;
p2 = p1;
int **p3;
p3 = &p2;
```

Two important facts about Pointers

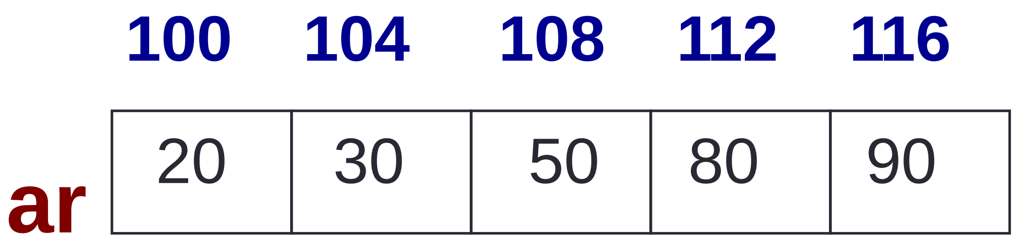
1) A pointer can only point to one type –(basic or derived) such as `int`, `char`, a `struct`, another pointer, etc

2) After declaring a pointer: `int *ptr;`

   `ptr` doesn’t actually point to anything yet. We can either:

   - make it point to something that already exists, or
   - allocate room in memory for something new that it will point to
   - Null check before dereferencing

Arrays and pointers



- `ar` is a pointer to the first element
- `ar[0]` is the same as `*ar`
- `ar[2]` is the same as `*(ar+2)`

- Use pointers to pass arrays in functions
- Use *pointer arithmetic* to access arrays more conveniently

Pointer Arithmetic

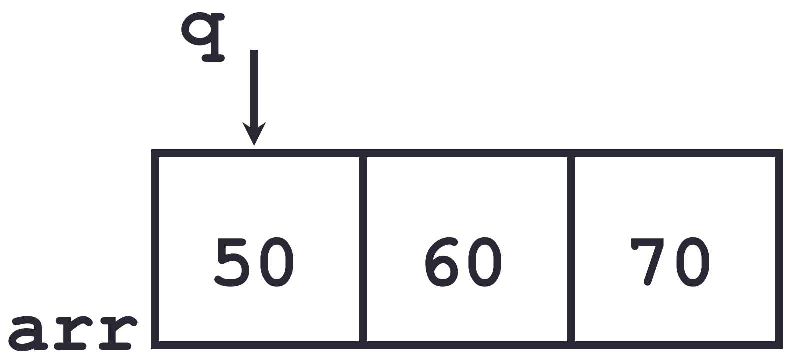
```
int arr[]={50, 60, 70};
int *p;
p = arr;
p = p + 1;
*p = *p + 1;
```

Passing arrays to functions

```
int main(){
  int arr[]={50, 60, 70};




}
int sum(int b[], int len){




}
```

Code to demonstrate how arrays are passed to functions

```
void IncrementPtr(int *p){
    p++;
}

int arr[3] = {50, 60, 70};
int *q = arr;
IncrementPtr(q);
```



Which of the following is true after **`IncrementPtr(q)`** is called in the above code:

A. **‘q’** points to the next element in the array with value 60

B. **‘q’** points to the first element in the array with value 50

How should we implement `IncrementPtr()`, so that ‘q’ points to 60 when the following code executes?

```
void IncrementPtr(int **p){
    p++;
}
```

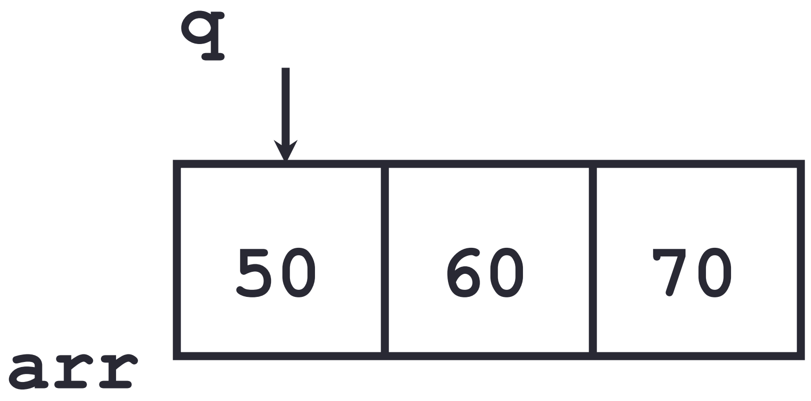
```
int arr[3] = {50, 60, 70};
int *q = arr;
IncrementPtr(&q);
```

A. `p =  p + 1;`

B. `&p = &p + 1;`

C. `*p= *p + 1;`

D. `p= &p+1;`



Demo

- In class demo to show how you would create an array of structs, initialize them and pass the array to a function (this relates to the last problem on hw 10)

Pointer Arithmetic Question

How many of the following are invalid?

I. pointer + integer (ptr+1)
II. integer + pointer (1+ptr)
III. pointer + pointer (ptr + ptr)
IV. pointer – integer (ptr – 1)
V. integer – pointer (1 – ptr)
VI. pointer – pointer (ptr – ptr)
VII. compare pointer to pointer (ptr == ptr)
VIII. compare pointer to integer (1 == ptr)
IX. compare pointer to 0 (ptr == 0)
X. compare pointer to NULL (ptr == NULL)

```
#invalid
    A: 1
    B: 2
    C: 3
    D: 4
    E: 5
```

Pointer Arithmetic

- What if we have an array of large structs (objects)?
  - C++ takes care of it: In reality, `ptr+1` doesn't add `1` to the memory address, but rather adds the size of the array element.
  - C++ knows the size of the thing a pointer points to – every addition or subtraction moves that many bytes: 1 byte for a char, 4 bytes for an int, etc.

Complex declarations in C/C++

How do we decipher declarations of this sort?

```
int **arr[];
```

Read

*	as "pointer to" (always on the left of identifier)
[]	as "array of" (always to the right of identifier)
( )	as "function returning" (always to the right …)

For more info see:
http://ieng9.ucsd.edu/~cs30x/rt_lt.rule.html

Complex declarations in C/C++

Right-Left Rule

```
int **arr [];
```

Illegal combinations include:

[]() - cannot have an array of functions
()() - cannot have a function that returns a function
()[] - cannot have a function that returns an array

Step 1: Find the identifier

Step 2: Look at the symbols to the right of the identifier. Continue right until you run out of symbols *OR* hit a *right* parenthesis ")"

Step 3: Look at the symbol to the left of the identifier. If it is not one of the symbols ‘*’, ‘(), ‘[]’ just say it. Otherwise, translate it into English using the table in the previous slide. Keep going left until you run out of symbols *OR* hit a *left* parenthesis "(".

Repeat steps 2 and 3 until you've formed your declaration.

Complex declarations in C/C++

```
int i;
int *i;
int  a[10];
int f( );
int **p;
int (*p)[];
int (*fp) ( );
int *p[];
int af[]( );
int *f();
int fa()[];
int ff()();
int (**ppa)[];
int (*apa[ ])[ ] ;
```

Next time

- Dynamic memory allocation