Day: Date: / /

Pointers

Pointer variables, which are often only called pointers, is a special variable that holds ~~an address~~ memory address.

Declaring a pointer:

dereferencing operator.

int * ptr ← name of pointer.

datatype.

int* ptr

read it like this: ptr is a pointer to an integer.

- pointer has a size depending upon the architecture.
  e.g 64-bit architecture has 8B for a pointer.

Operators:

- & = referencing operator
- * = dereferencing operator.

Yousaf

CamScanner

Day: Date: / /

Example:

```
int x = 5;
int *ptr = &x;
cout << *ptr << endl;
cout << ptr << endl;
```

This requires an address

& gives an address

Terminal
5
0x100 # some address

Initializing a pointer:

- we should always initialize a pointer.
- If we don't have a value to initialize with then with null.
  - There are 3 ways to initialize a pointer with null:
    1) `int *ptr = NULL;`
    2) `int * ptr = nullptr;`
    3) `int* ptr = 0;`

Double pointers:

e.g `int **pptr = NULL;`

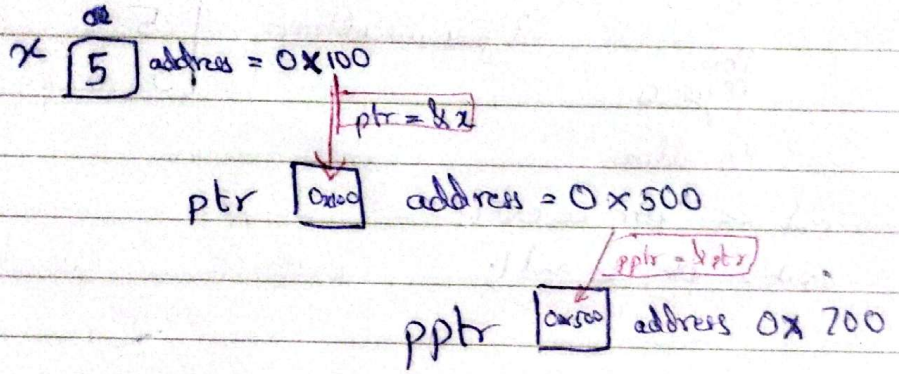
pptr is a pointer to an integer pointer

- A double pointer will store an address of a pointer so, pptr = &ptr.

Yousaf

Day: Date: / /

- if we want to get to the number then we will de reference the pointer by equal number of asterisk used to make the pointer.



[redacted]

Arrays and pointer:

- pointer ~~can~~ ~~also~~ ~~to~~ point to certain address.
- Arrays, the name of Array also points to the base address or first address. So arrays can also be said to be a pointer.

pointers can be used to access arrays:

```
int arr[3] = {1,2,3}
int* ptr = arr; you don't need &arr as arr is already an address
cout << *ptr << "\n" << *(ptr+1) << "\n" << ptr[1] << endl;
```

Out put: 1
1 2
2

Day: Date: / /

(ptr + 1) this is pointer arithmetic, will be talked about later. This gives the next address of the next element in the array.

we can also use the ptr as ptr[1] as this is also allowed and means the same thing.

Likewise:

we can also use the arr like *(arr+1), *arr, *(arr+2). And this is ALLOWED.

Important points:

- int * ptr = &array ✗ NOT Allowed.
  This is already an address.

  but

  cout << array << endl; ✓
  cout << &array << endl; ✓ allowed. gives same result

≡ An array is a certain type of pointer (constant pointer)

= You have to write "*" with every pointer you declare:

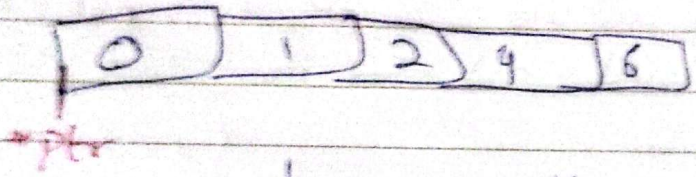
int * ptr, p, *abc;
ptr: pointer
p: not a pointer, simply an int.
abc: pointer

Yousaf

Date: / /

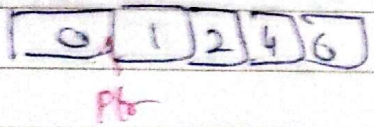
Pointer Arithmetic:

- You can add or subtract to a pointer.
- Increment or decrement is also allowed.



- ptr is currently at 0 index.

ptr++ :



- ptr now at 1st index.

But how? we only added one but each index is 4Bytes apart?

- This depends on the datatype of ptr. If ptr is an int type +1 will mean addition of 4B. Short int will addition of 2B. Char will mean +1B.

- Same for -1.

Yousaf

Day: Date: / /

Pointer aliasing:

Aliasing is linking two or more variables in such a way that if one changes then both change.

Weak aliasing:

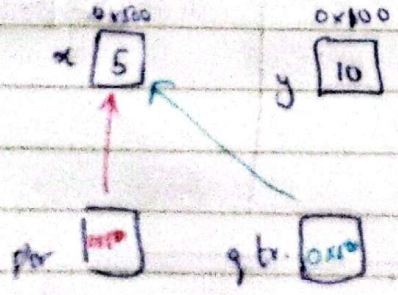
```
1  int main(){
2      int x = 5, y = 10;
3      int *ptr = &x;
4      int *qtr = ptr;
5
6      *ptr = 7;
7      cout << *qtr;
8
9  return 0;
10 }
```

Terminal:

7

So as we changed the value in ptr qtr also changed. but this is weak aliasing.

Memory (RAM)



Why weak?

As you can see in the memory. when we do qtr = ptr; qtr store the same address as ptr. But now if we change ptr to 8y then qtr doesn't change and still points to x. So qtr and ptr are no longer alias

Yousaf

Day: Date: / /

Stronger aliasing:

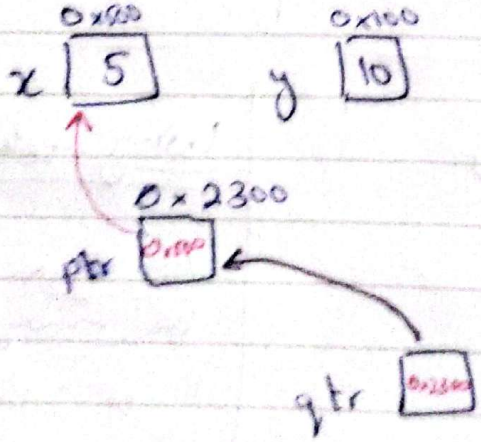
```
1  int main(){
2      int x = 5, y = 10;
3      int * ptr = &x;
4      int ** qtr = &ptr;
5
6      *ptr = 7;
7      cout << ** qtr << endl;
8
9      ptr = &y;
10     cout << ** qtr << endl;
11 return 0;
12 }
```

Terminal:

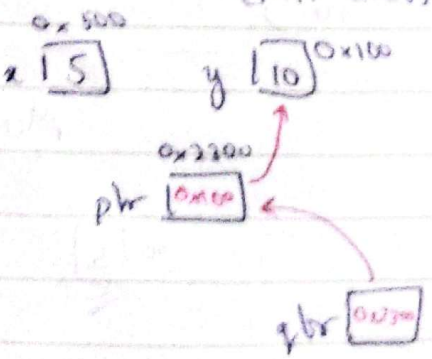
```
7
10
```

Now even if we change ~~value~~ of the address that ptr points to, qtr will change.

Memory: (Till line 8)



Memory (After line 8)



- So now we can see that even if ptr points to something else qtr. will also change as it points to ptr.
- But we can only change ptr to change qtr and cannot do vice versa.

Day: Date: / /

Strongest aliasing (True aliasing):

```
1  int main(){
2      int x=5, y=10, z=15;
3      int *ptr = &x;
4
5      int *&qtr = ptr;
6
7      cout << *qtr <<endl;
8
9      *ptr = &y;
10     cout << *qtr <<endl;
11
12     qtr = &z;
13     cout<< *ptr <<endl;
14 return 0;
15 }
```

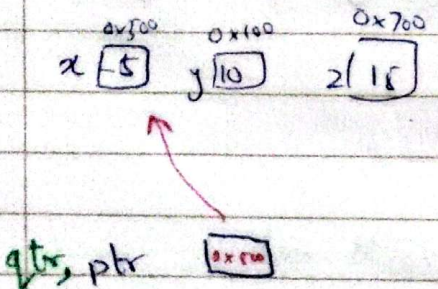
Terminal

```
5
10
15
```

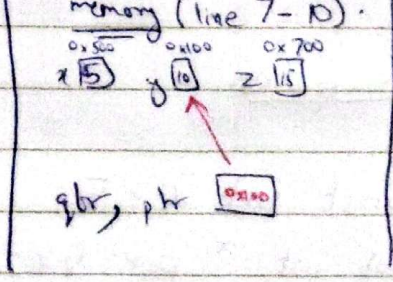
Explanation

- qtr is no new variable.
- qtr is now another name for ptr.
- qtr is same as ptr and qtr has no memory of its own and qtr and ptr are both same.
- qtr is now an Alias of ptr.

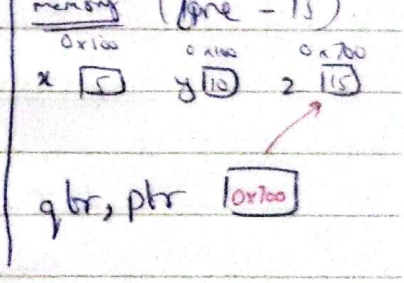
Memory (till line 7):



memory (line 7-10):



memory (line 10-15):



- Now qtr and ptr are same.
- It doesn't matter whichever we change, both will always change.
- True aliasing.

Yousaf

Day: Date: / /

Constant pointers:

3 types:

1) pointer to a constant integer;

```
int x = 5, y = 10;
const int *p = &x;   OR   int const *p = &x;
```

read it like this:

p is a pointer to a constant integer variable

- so now p points to a constant integer so now you cannot change value by *p. but you can still change value by x = 6 or x = x+1 etc.
- You can still change address that p points to eg p = &y is allowed

Important points:

```
const int x = 5;
const int *p = &x;   ✓ allowed
int *p = &x          ✗ not allowed as p points to integer not constant integer.
```

Yousuf

CamScanner

Day: Date: / /

2) constant pointer to an integer.

```
int x=5, y=10;
int * const p = &x;
```

constant pointer to an integer variable.

- cannot change address
- can change value of x
- need to ~~declare~~ ~~it~~ initialize.

- Arrays are this type of pointers.

3) constant pointer to a constant integer

```
const int * const p = &x
```

OR.

```
int const * const p = &x;
```

- Cannot change address or value.

Yousaf

CamScanner

Day: Date: / /

Why are arrays constant pointers (2nd type):

```
int arr[5];
int * const ptr;
```
} same.

This is made in stack and if we change it, it cannot be deleted as compiler deletes what it creates and if we change it compiler doesn't delete it.

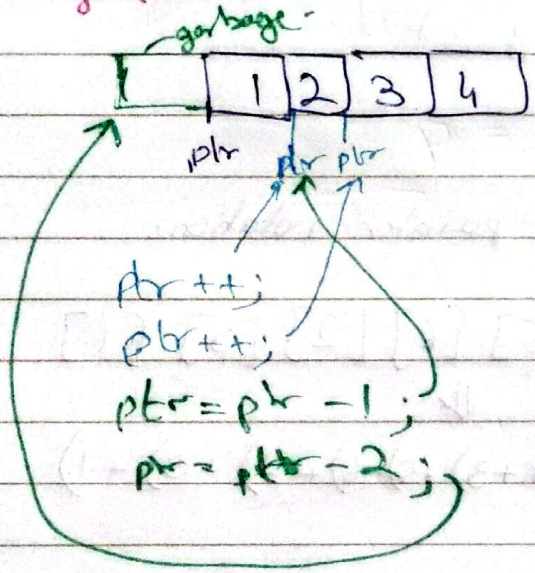
Yousuf

Day: Date: / /

Arrays: (An important point):

int arr [ ] = {1, 2, 3, 4}

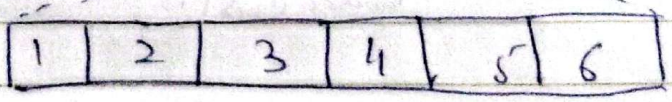
e.g ptr = arr



2D Arrays:

- It is just an illusion that 2D Arrays are made like this ▦ instead they are a single line:

1	2
3	4
5	6



int arr [3] [4] = {1, 2, 3 ..... 11, 12};

cout << arr
cout << arr[0]
cout << &arr[0][0]

} These 3 will give same output.

Yousaf

Day: Date: / / /

arr[1][3]
*(arr+1)[3]
((arr+1)+3)
} same things.

- array through pointer notation:

arr[3][5][6][2][0][1]

⇓

((*(*(*(*(arr+3)+5)+6)+2)+0)+1)

An efficient code for traversing in a 2D array:

Normal code:

```
for(int i=0; i<3; i++){
    for(int j=0; j<4; j++){
        cout << arr[i][j];
    }
    cout<<endl;
}
```

- This a slow code because everytime or all 12 times in this code we first de reference *(arr+i) and then de refer *(*(arr+i)+j).

Total amount of de referencing = 12+12 = 24.

Yousaf

Day: Date: / /

Faster code:

```
for(int i=0; i<3; i++) {

    int *p = arr[i]; //or *(arr+i)
    for(int j=0; j<4; j++){
        cout p[j] ;//or *(p+j)
    }
}
```

- Now this is a faster code because now *(arr+i) is dereferenced 3 times which is 9 less than 12 times.

Total amount of dereferencing = 3+12 = 15.

- This speed might not be visible in small program but for huge array it is very time consuming.

Yousaf

Day: Date: / /

Dynamic memory Allocation (DMA):

We can dynamically make an array or variable in the heap.

Making a variable:

```
int *p = new int;
```

data Type needed.

This keyword makes the variable in the heap.

- We created this in the heap but now remember that heap needs to deleted by ourselves and compiler won't do it. HOW?

For deletion we use: delete p;

```
delete p;
```

- delete doesn't really delete p but deallocate it. This means that now that memory is not in control of p and can be given to any other pointer.

- But even after p delete, p is a dangling pointer so we need to write:

```
p = NULL;
```

Yousuf

Day: Date: / /

code:

```
int *p;        → This a garbage or wild pointer.
p = new int;   → a pointer.

delete p;      → a dangling pointer.
p = nullptr;
```

Dangling pointers:

These are pointers which have a valid address but using them can cause segmentation fault.

What happens if we don't delete:

- Memory leak.

e.g

```
int *p = new int;
p = nullptr;
```

we made a pointer and gave it 4B as it is an int. but when we don nullptr p's address changes and now we have no way to get control of the integer we had created.

Yousaf

Day: Date: / /

2)

```
int fun(){
    int x = 5;
    return &x;
}

int main() {
    int * x = fun();
return 0;
}
```

This above code will give segemen tation error or may run free as x has become a dangling pointer as it points to some address which is already destroyed.

(This will work correct if you made x in fun() as STATIC Integer).

Day

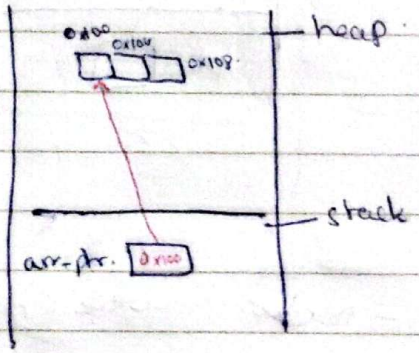
Date: / /

Dynamic 1D Arrays:

To create an array:

```
int * arr_ptr = new int [3];
```

This will create an array of size 3 and type integer in heap.



Dynamic 2D arrays:

- To create a dynamic 2D array we have to make an array of pointers which will each point to an 1D array.

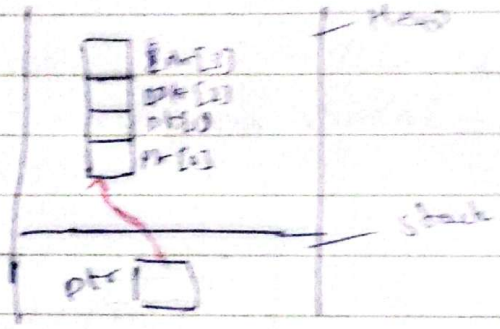
```
1   int **ptr = new int *[rows]
2   for(int i=0; i< rows; i++) {
3       *(ptr+i) = new int [colums]
4   }
```

what is happening?

Page: Date: /

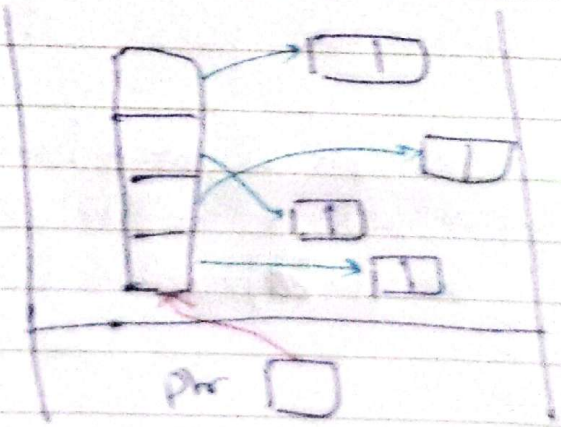
line1:

ptr makes an array of pointers.



line 3:

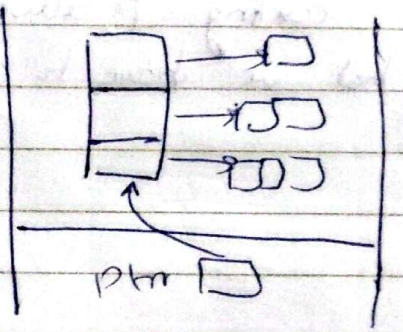
- Now we have an array of pointers.
- Now each pointer is pointing to another array.



Day: Date: / /

Now we have a 2D array which is accessible of 4 Rows and 2 colums.

- like this we can make any type of array even triangular.



code:

```
int **ptr = new int *[3]
*(ptr+0) = new int [1];
*(ptr+1) = new int [2];
*(ptr+2) = new int [3];
```

deletion:

First we will have to delete all inner 1D arrays, and then delete the array of pointers.

```
for(int i=0; i<rows; i++){
    delete [](ptr+i);
}
delete [] ptr;
ptr = NULL;
```

Day: Date:

Pointer casting.

casting is a function (any type implicit or explicit).

- They are built-in.

- pointer casting is always explicit casting meaning that we have to do it ourselves.

Example.

~~char~~

short int ptr[2] = {6961, 87}

short int * ptr = ptr; // No casting as both are

char * ctr = ptr; * error

char * ctr = (char *) ptr; ✓ allowed

casting

Yousaf

Day: Date: / /

- But pointer casting is very dangerous. because if pointer points to int it decodes 4B, if pointer points to char it decodes 1B. It doesn't matter whatever the data is. So if we do pointer casting and change short int to char we just start decoding 1B instead of 2B.

example:

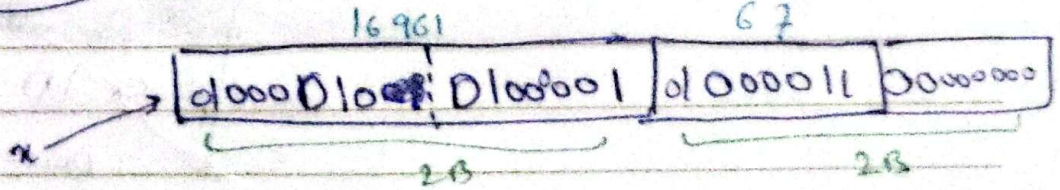
```
short int x[2] = {16961, 67};
char * ctr = (char*) x;

cout << *ctr << endl;
cout << *(ctr+1) << endl;
cout << *(ctr+2) << endl;
```

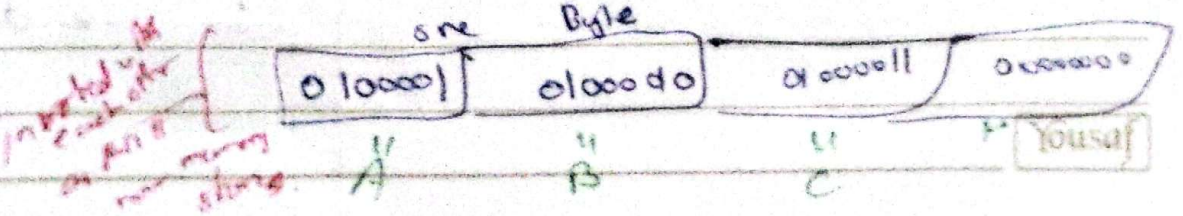
Terminal:
A
B
C

Why:

let's look at the memory.



when we convert to char it Beam one Byte



Day: Date:

Changing dimensions using casting:

```
int arr[][4] = {1,2,3,4,5,6,7,8,9,10,11,12};

int (*ptr)[4] = arr;
```

we made ptr like int(*ptr)[4] not like int *ptr[4] as this is error as ptr

To change dimensions:

```
int (*ptr)[2] = (int(*)[2])arr;
```

casting

Now ptr becomes:

1 2
3 4
5 6
7 8
9 10
11 12

writing we write int (*ptr)[4]:

ptr is retrieved and then we give 4 to each retrieve. This means a 2D array.

while int *ptr[4]: mean ptr[4] will be execute first and it will be a a 1D array of 4 integer pointer.

same we did in int(*)[2]

Yousaf