



REFERENCES, POINTERS PASSING PARAMETERS TO FUNCTIONS

Problem Solving with Computers-I

C++

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

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

GitHub



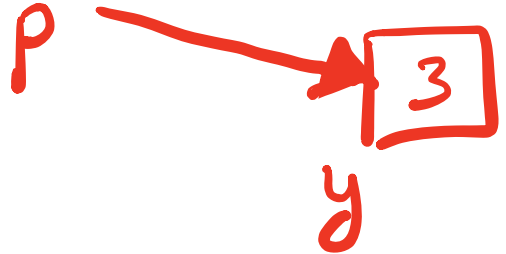
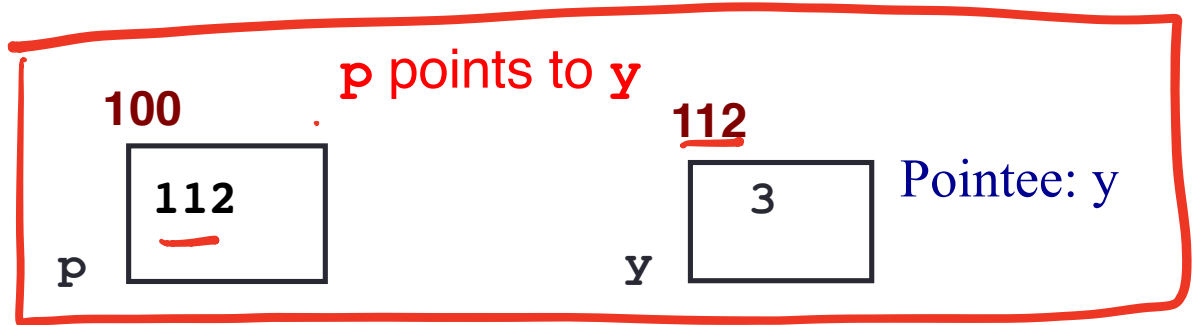
Announcements

- H05 and H06 are released. (pdf versions available on website)
- Please submit a pdf version of your answers to the assignment on gradescope before the due date
 - Print, write by hand, scan, upload
 - Download, annotate, upload
 - Use Word (or some other text editor to write the answers only), convert to pdf and upload.

Pointer Diagrams:

Diagrams that show the relationship between pointers and pointees

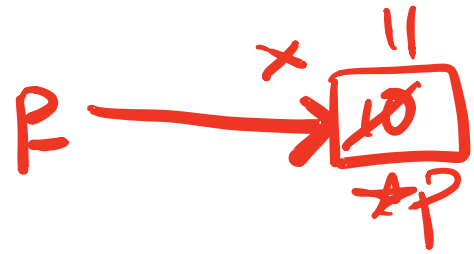
Pointer: p



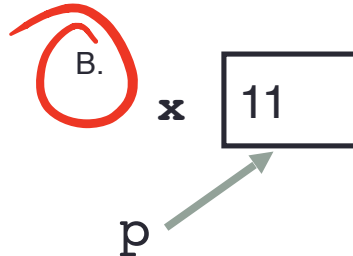
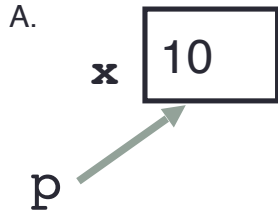
Tracing code involving pointers

```
int *p;  
int x = 10;  
p = &x; 10
```

```
*p = *p + 1;
```



Q: Which of the following pointer diagrams best represents the outcome of the above code?



C. Neither, the code is incorrect

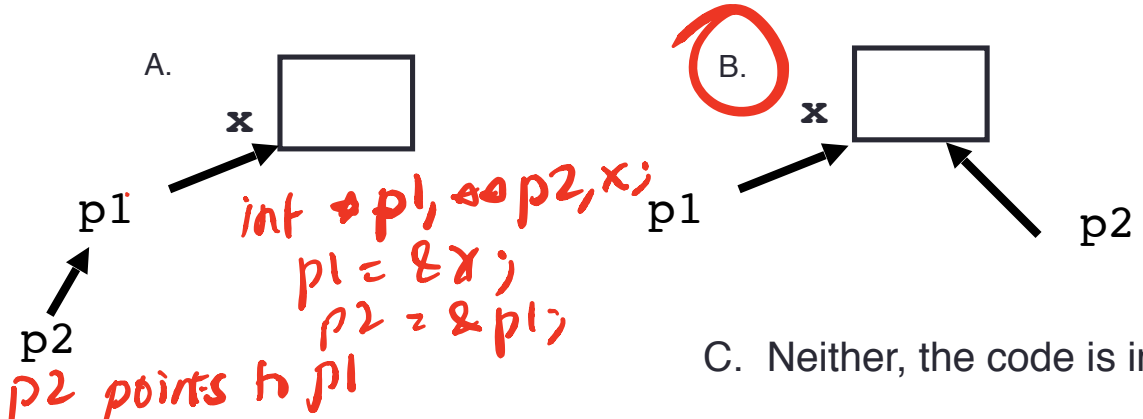
Pointer assignment

$\left. \begin{array}{l} \text{int } *p1; \\ \text{int } *p2; \\ \text{int } x; \end{array} \right\} \Rightarrow \text{int } *p1, *p2, x;$

```
int *p1, *p2, x;
```

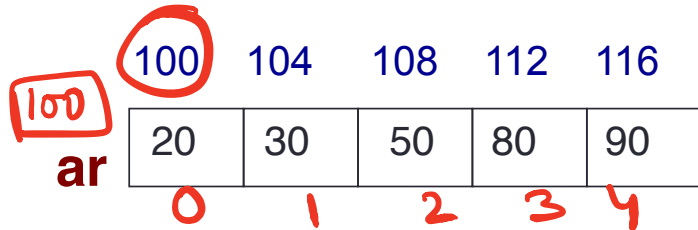
```
p1 = &x;  
p2 = p1;
```

Q: Which of the following pointer diagrams best represents the outcome of the above code?



C. Neither, the code is incorrect

Arrays and pointers



- `ar` is like 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

`ar[0]`
↓
`*ar`

`ar+2`
↓
Add the size of 2
ints to `ar`

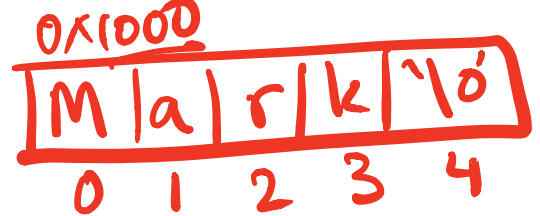
Pointer arithmetic

Address of element at index
2

What is the output of the code?

```
char s1[] = "Mark";  
char s2[] = "Jill";  
for (int i = 0; i <= 4; i++)  
    s2[i] = s1[i];  
if (s1 == s2) s1 = "Art";  
cout << s1 << " " << s2 << endl;
```

S1



S2



checks if base address of arrays are equal.

Compiler error! cannot change the base address of the array

- A. Mark Jill
- B. Mark Mark
- C. Art Mark
- D. Compiler error
- E. Run-time error

Pass by value

```
void swapValue(int x, int y){
    int tmp = x;
    x = y;
    y = tmp;
}
int main() {
    int a=30, b=40;
    cout<<a<<" "<<b<<endl;
    swapValue( a, b);
    cout<<a<<" "<<b<<endl;
}
```

40 30

What is printed by
this code?

A.

30 40
30 40

B.

30 40
40 30

C. Something else

swap Value
x [30] *y* [40]
x, y are copies of a & b

a [30] *b* [40]
Local Var.

of main


```

void swapValue (int &x, int &y) {
    int tmp = x;
    x = y;
    y = tmp;
}

```

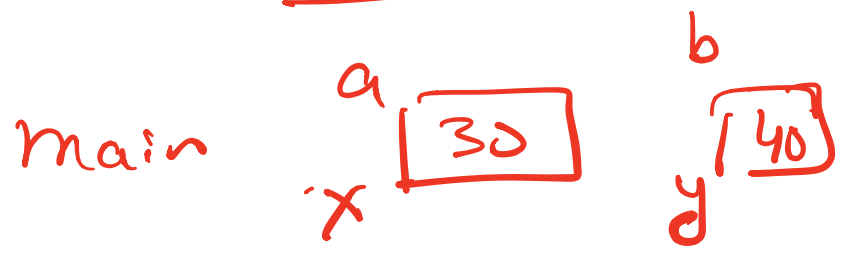
↓ ↓
x & y are
REFERENCES

```

int main () {
    int a = 30, b = 40;
    swapValue(a, b);
    // Pass by reference.
}

```

↓
nice names
for other variables



Notice the difference between the memory diagram on this page compared to the previous page (when variables were passed by values)

References in C++

```
int main() {  
    int d = 5;  
    int &e = d;  
}
```

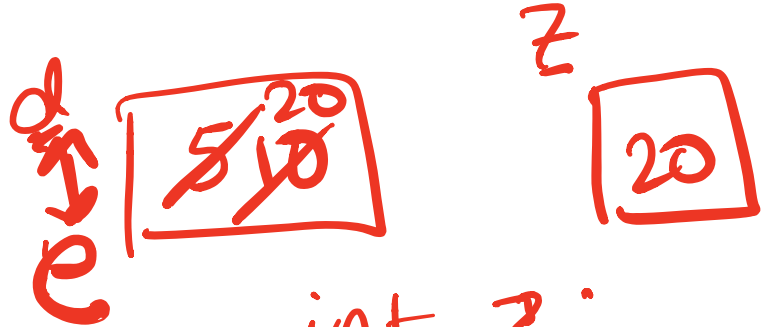
e is a reference to d

d = 10;

e = 20;

cout << d;

A reference in C++ is an alias for another variable



int z;

z = e;

e = x;

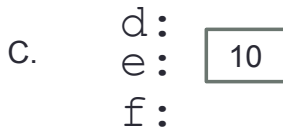
References in C++

```
int main() {  
    int d = 5;  
    int &e = d;  
    int f = 10;  
    e = f;  
}
```

How does the diagram change with this code?



f: 10



e: 10
f: 10

D. Other or error

Passing parameters by reference

```
void swapValue(int&x, int&y){  
    int tmp = x;  
    x = y;  
    y = tmp;  
}
```

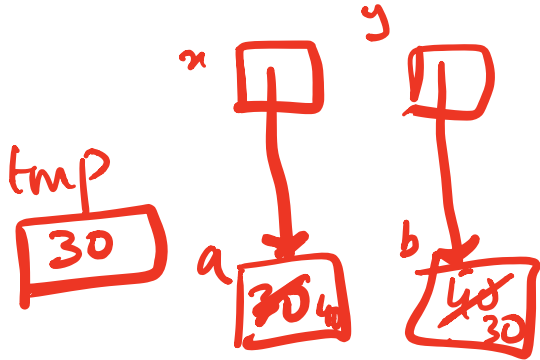
```
int main() {  
    int a=30, b=40;  
    swapValue( a, b);  
    cout<<a<<" "<<b<<endl;  
}
```

40 30

Passing parameters by address

```
void swapValue(int* x, int* y){  
    int tmp = *x;  
    *x = *y;  
    *y = tmp;  
}
```

```
int main() {  
    int a=30, b=40;  
    swapValue(&a, &b);  
    cout<<a<<" "<<b<<endl;  
}
```



Pass by address : The address of a & b are copied into pointers x and y

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.

Pointer Arithmetic

```
int ar[]={20, 30, 50, 80, 90};  
int *p;  
p = arr;  
p = p + 1;  
*p = *p + 1;
```

Draw the array ar after the above code is executed

100 104 108 . . .

Pointer Arithmetic

0 1 2 3 4

```
int ar[]={20, 30, 50, 80, 90};
```

How many of the following are invalid?

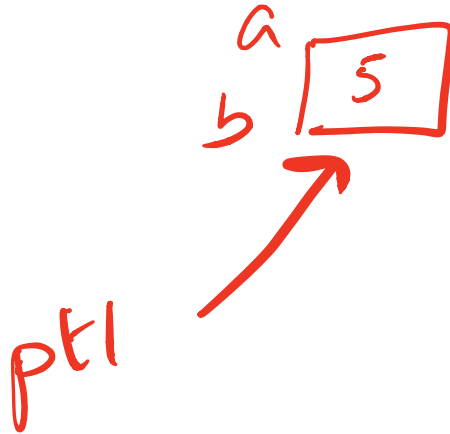
ptr = ptr - 1;

- 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) ✓

| <u>#invalid</u> | |
|-----------------|---|
| A: | 1 |
| B: | 2 |
| C: | 3 |
| D: | 4 |
| E: | 5 |

Pointers and references: Draw the diagram for this code

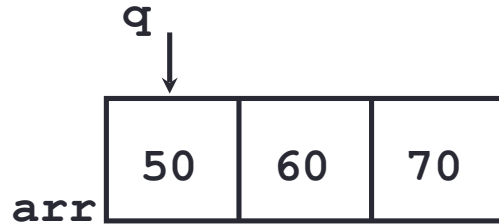
```
int a = 5;  
int &b = a;  
int *pt1 = &a;
```



What are three ways
to change the value of
'a' to 42?

```
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:

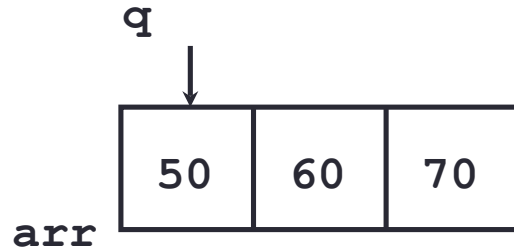
- A. 'q' points to the next element in the array with value 60
- B. 'q' points to the first element in the array with value 50

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;`



Pointer pitfalls

- Dereferencing a pointer that does not point to anything results in undefined behavior.
- On most occasions your program will crash
- Segmentation faults: Program crashes because code tried to access memory location that either doesn't exist or you don't have access to

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

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

Next time

- Structs
- Arrays of structs