

# REFERENCES, POINTERS AND STRUCTS

---

Problem Solving with Computers-I

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

C++

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

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



# Pointer assignment

```
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

# Modify the function to swap the values of a and b: use pointers

```
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;  
}
```

Draw the pointer diagram for your code

# Segmentation faults (aka segfault)

- Segfault- your program has crashed!
- What caused the crash?
  - Read or write to a memory location that either doesn't exist or you don't have permission to access
  - Dereferencing a null pointer
- Avoid segfaults in your code by
  - Always initializing a pointer to null upon declaration
  - Performing a null check before dereferencing it
  - Avoid redundant null checks by specifying pre and post conditions for functions that use pointers

```
int *p;  
*p = 5;
```

Q: Which of the following is true about the above code?

A	Compile time error
B	Runtime error
C	Code runs without error

# References in C++

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

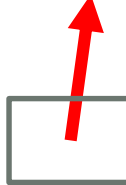
A reference in C++ is an alias for another variable

A. d 

e 

C. d   
e

B. d 

e 

D. This code causes an error

# References in C++

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

How does the diagram change with this code?

A. 

f: 

B. 

e:   
f: 

C.   
d: 

D. Other or error

## 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?



# Call by reference: Modify to correctly swap a and b

```
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;  
}
```

# C++ structures

- A **struct** is a data structure composed of simpler data types.

```
struct Point {  
    double x;  
    double y;  
};
```

# Pointers to structures

The C arrow operator ( $->$ ) dereferences and extracts a structure field with a single operator.

```
struct Point {  
    double x;  
    double y;  
};
```

Demo program using  
points

# References to structures

Draw a diagram to show the state of memory when the function `setPoint` is called

```
void setPoint(Point &q double x, double y)
{
    //Code to set the x and y values of q
}
```

```
int main(){
    Point p;
    setPoint(p, 100.0, 200);
    cout <<p.x <<" " <<p.y<<endl
}
```

## 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

# 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](http://ieng9.ucsd.edu/~cs30x/rt_lt.rule.html)

# Complex declarations in C/C++

## Right-Left Rule

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

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.

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

# 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[ ])[ ] ;
```



## Pointer assignment: Trace the code

```
int x=10, y=20;
```

```
int *p1 = &x, *p2 = &y;
```

```
p2 = p1;
```

```
int **p3;
```

```
p3 = &p2;
```

# Next time

- Arrays and pointers
- Arrays of structs
- Dynamic memory allocation