Functions

Prototype (header files.h)

return type: int

name: main

Arguments: (int ,)

```
main( int , );
printSquares( int numSquares );
```

return type → void

but doesn't return anything

Function Signature

Function Code (.c)

```
int main() {
    // stuff
    return(0);
}
```

Function → abstraction, reuse, encapsulation

① solve problems by breaking down into smaller

ex) rocket → ends up the cost to determine exactly how much fuel to pump into boosters

Reuse allows more efficient

Separate our code into capsules or buckets that don't effect other pieces

① side effects: if it modifies something outside of its code

② no-side effect (pure)

ex) Python `a=[1,2]` `a.sort()`

↳ Racket, Haskell

Function Call;

```
y = sqrt(7);
```

sqrt: Function, 7: Argument


```
y = sqrt(7);
x = sqrt(y);
x = sqrt(sqrt(sqrt(x)));
```

nesting

Printf, scanf where

	where	arg	return
abs(x)	stdlib.h	int	int
ceil(x)	math.h	double	double
46.7 → 47.0			
cos(x)		double	double
fabs(x)		double	double

pow double, double double

ex) $ax^2 + bx + c = 0 \rightarrow \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

```
double root1(double a, double, double c) {
    disc = sqrt( pow(b, 2) - 4 * a * c );
    root1 = ( -b + disc ) / ( 2 * a );
    return (root1);
}
```

→ may be absorbed

Problem

You want draw some Stick Figure

Fig 2.

- Triangles
- circle
- parallel lines
- intersecting lines
- straight line

Algorithm

1. Draw circle
2. Draw Triangle → intersecting base
3. Draw intersecting line

Prototypes

```
void draw_circle();
void draw_triangle();
void draw_base();
void draw_intersect();
```

main.c

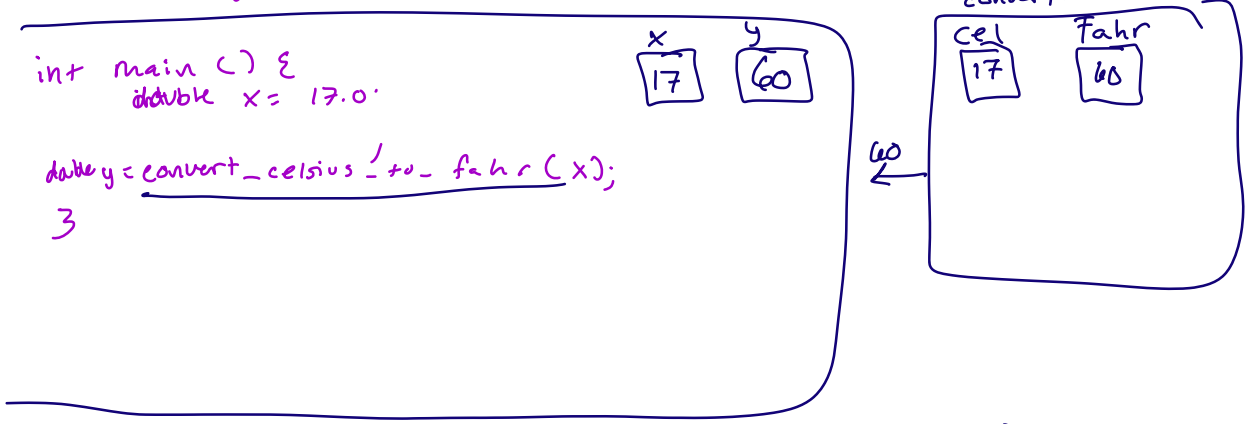
```
int main() {
    draw_circle();
    draw_triangle();
    draw_intersect();
}
```

```
void draw_circle() {
    printf("  *  \n");
    printf(" * * \n");
    printf("*   *\n");
}
```

```
void draw_h() {
    Printf("**   **\n");
    printf("**   **\n");
    // Put a bunch of stars until its an H
}
```

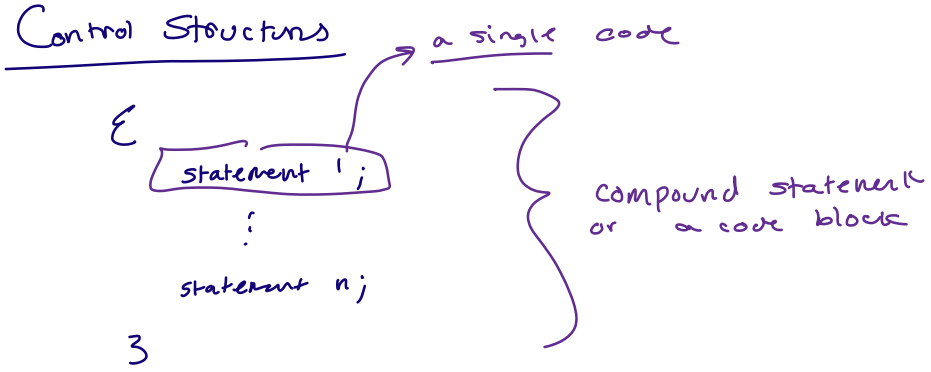
Function that arguments:

Primitive datatypes (int, char, bool ...) are passed by value when arguments or returns.



(Also, check modules canvas textbook chapters)

Control Structures



A condition is an expression which evaluates to a boolean.

→ boolean variable

→ Variable operator variable

example a == c

<	less than	infix binary operators
>	greater than	
<=	less than eq	
>=	gteq	
==	equals	
!=	not equals	

a + b ← infix

add(a, b) ← not infix

→ ! ← not unary

!(a < b) ⟷ (a >= b)

examples

```
int a = 7;
int b = 8;
condition = a<b;  ⇝ True
```

Conjunction ← this "and" symbol above remember ?

&&	and operator	condition && condition
||	or operator	condition || condition

[& → just by itself in front of a variable gets its address]

Truth Tables

a	b	&&
0	0	0
0	1	0
1	0	0
1	1	1

a	b	||
0	0	0
0	1	1
1	0	1
1	1	1

Fun Fact: NAND → can make every other boolean function

Not = NAND(a, a)

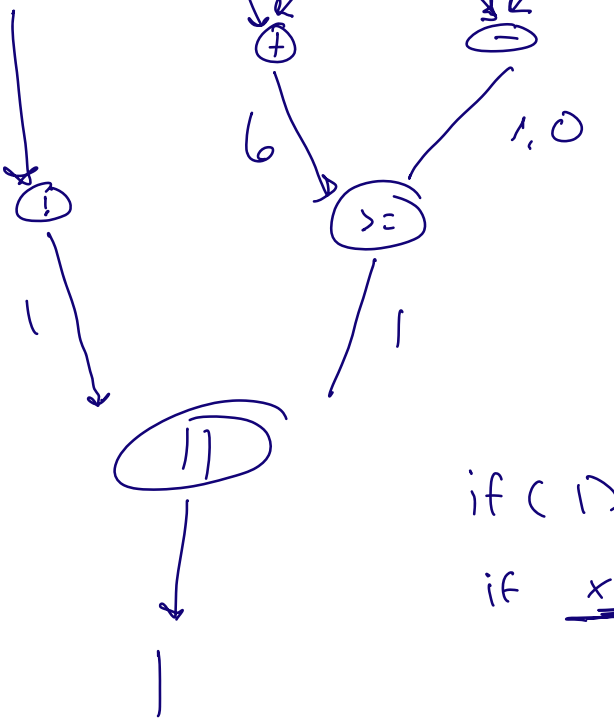
AND = NOT(NAND(·, ·))

ex) flag, x, y, 7 — 0, 3.0, 4.0, 2.0

First → last

- (negation), ! (not)
*, /, %
+ - (binary)
< <= => >
== !=
&&
||
=

!flag || (y + z >= x - z)



if (1); → if (true);

if x == 1

DeMorg's Law

$$(X \cap Y)^c = X^c \cup Y^c$$

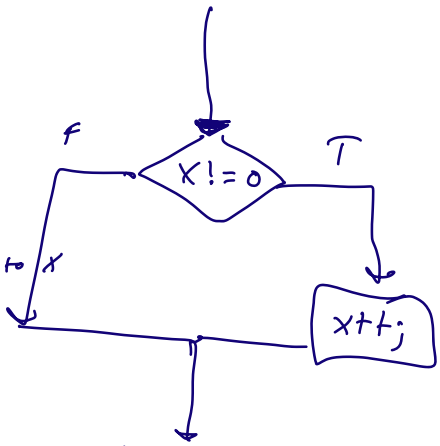
↳ $!(X \&\& Y) = (!X) \,||\, (!Y)$

First Structure

IF statements;

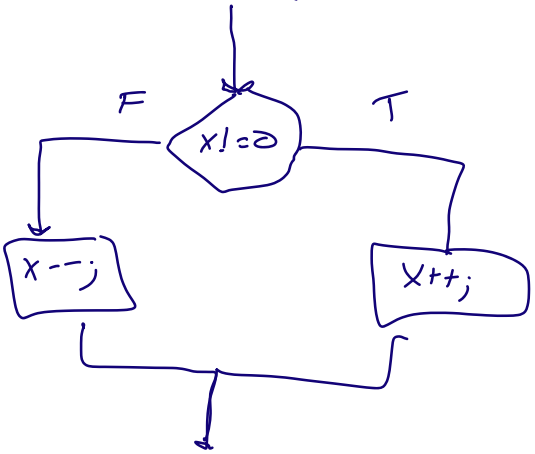
```
if (condition) {
    // do this if true
}
```

```
if (x!=0) {
    x++; //adds one to x
}
```




```
if (condition) {
    // do 1
} else {
    // otherwise
}
```

```
if(x!=0){
    x++;
} else {
    x--;
}
```



→ code blocks with {} have scope only of themselves and greater

↳
```
int b = 6
if (...) {
    int a = 2
    printf("%d", b);
}
printf("%d", a); ← error
```

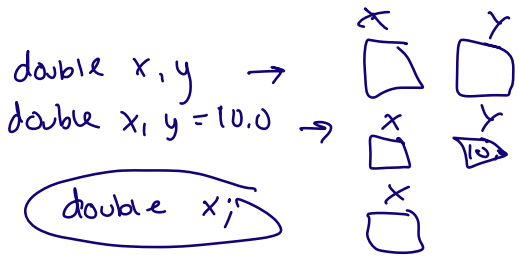
```
int func1(int x){
    if (x != 0) {
        return x;
    }
    return -1;
}
```

```
if (x!=0) {
    return x;
} else {
    return -1;
}
```

If statements can be nested

ex)
```
double x, y = 10.0
if (y < 15.00) {
    if (y >= 0.0) {
        x = 5 * y;
    } else {
        x = 2 * y;
    }
} else {
    x = 3 * y;
}
```

→ at the end x will equal 50



ex) If - else if - else

```
if (condition) {
    // do something
} else if (condition) {
    // do something
} else {
    // do something
}
```

```
void printReason (int month) {
    if (month == 12) || (month <= 3) {
        printf("winter");
    } else if (month <= 6) {
        printf("spring");
    } else if (month <= 9) {
        printf("summer");
    } else {
        printf("Fall");
    }
}
```

[order of blocks and conditionals]

Month <= 6 && month > 3

Short circuit → if (x && y)

↓ F

↳ if you're in an and statement, and the first expression is false, the second thing never checked.

ex)
```
double x, y
x = 6;
y = 0;
if (y != 0 && y/x == 7.0) {

}
```

if true || ~~0/0~~ {

}

Switch

expression chr or boolean involving discrete cases

examples → Days of month

Syntax

```
char var = 'A';
Switch (var) {
    case 'M':
    case 'm': printf("Monday\n");
              break;
    case 'T';
    case 't': printf("Tuesday");
              break;
}
```

```
switch (expression) {
    case '_': | statement
              break;
    case '_';
    case '_':
              break;

    default :
}
```

```
int main() {
    char class;
    scanf("%c", &class);
    switch (class) {
        case 'B':
        case 'b': printf("battleship");
                  break;
        case 'C':
        case 'c': printf("cruiser");

        defaut : printf("char dec is not valid game piece", class);
    }
```

```
if class == 'B' ||
   class == 'b' {
}
else if class ==
}
```

machine

jump if condit code location

jump if 'a' go here →

int x;

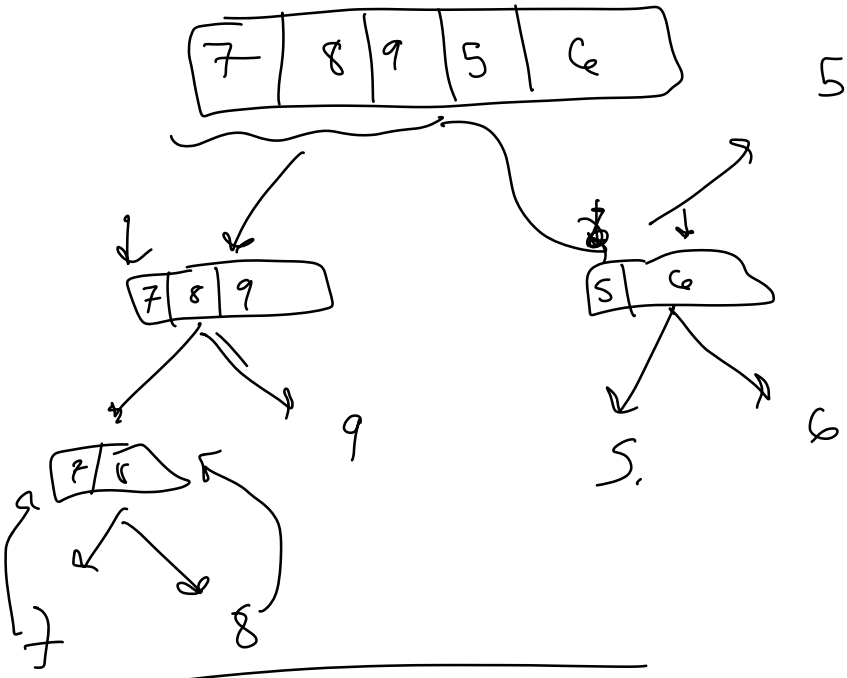
Swith

case 1:

Recursion

Of repeating process on smaller and chunks to solve a problem.

↳ Mergesort



Factorials $= X! = X \cdot (X-1) \cdot (X-2) \cdots (1)$

```
int factorial(int n) {
    if n==1 {
        return 1;
    }
    return n * factorial(n-1);
}
```

factorial(5); 32

return 5 · factorial(4)

↳return 4 · factorial(3) 6

↳return 3 · factorial(2) 2

↳return 2 · factorial(1) 1

↳return 1;

(1) Make sure base case is smallest case

(2) you don't get infinite recursion

$x_0 = 1$

$x_1 = 1$

$x_n = x_{n-1} + x_{n-2}$

```
int fib(int n) {
    if n == 0 || n == 1 {
        return 1;
    }
    return fib(n-1) + fib(n-2);
}
```

(3) you may need helper fun :

```
//print a triangle to screen
void myFunction(int h) {
    if (h == 0) {
        return;
    }
    //loop to print h stars
    myFunction(h-1);
}
```

4

* * * *

**

*

```
void myFunction(int h) {
    if (h == 0) {
        return;
    }
    myFunction(h-1);
    //print my *
}
```

*

**

*

** **

Arrays

```
int arr[7]; // from 0, ..., size-1
```
(int: datatype; arr: name)

```
arr[6] = 8; // assign to spot 7 the number 8
printf("%d", arr[3]); // this accesses the fourth and prints it.
```