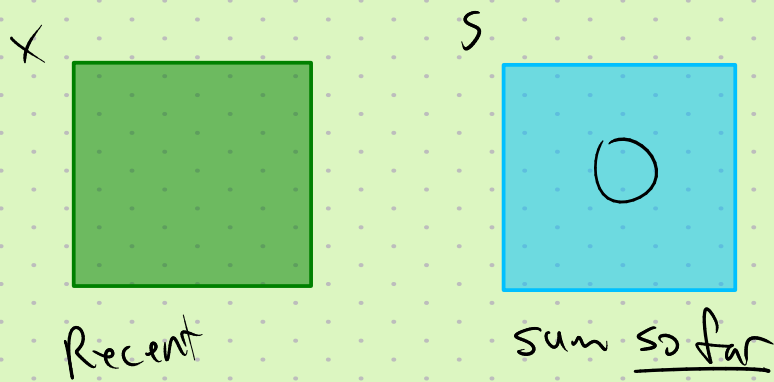
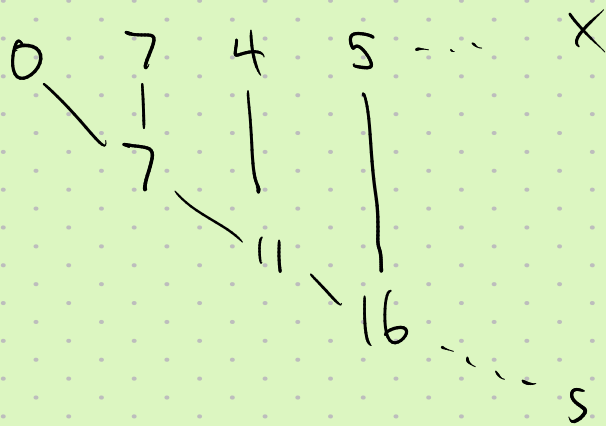


Last time: computing sum of integers from stdin.

Idea: 2 variables:



Then we kind of fold each new value into a running total



Note: This is a special case of the "fold" pattern.

Say you have a binary operator $\boxed{?}$ and you

want to compute

$$x_1 \boxed{?} x_2 \boxed{?} x_3 \boxed{?} \dots \boxed{?} x_n$$

($\boxed{?}$ could be $+$, $*$, ...)

Suppose also, you have a "neutral element" e :

for all x , $x \boxed{?} e = e \boxed{?} x = x$.

E.g. $e = 0$ for $+$, $e = 1$ for $*$.

Then you have the following meta solution:

```
s = e;  
while (cin >> x)  
    s = s  $\boxed{?}$  x;
```

Note: our largest ~~the~~ program follows this exactly!

$a \boxed{?} b \stackrel{\text{def}}{=} \max(a, b)$ ($\stackrel{\text{def}}{=}$ signals a new definition)

$e = -\infty$ ($\approx \text{INT_MIN}$)

Similarly, if $\boxed{?} = \min$, $e = \infty$ ($\approx \text{INT_MAX}$),
meta program would compute smallest integer.

There are tons of examples... Maybe read about
functional programming.

Review of basic ingredients / tools.

- allocate storage / memory, e.g.

```
int x;  
float y;  
char c;  
string s;  
bool b;
```

- Overwrite stuff, e.g.

```
x = 7;  
y = 2.0;  
c = 'A';  
s = "abcdef";  
b = (x == 99);
```

- Do something maybe (≤ 1 times), e.g.

```
if (x < 100) {  
    // thing to do if x < 100  
    ...  
}
```

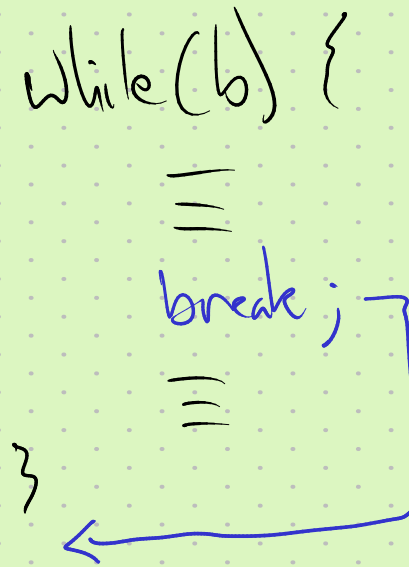
See also else, else if ...

- Do something repeatedly (until a condition is broken)

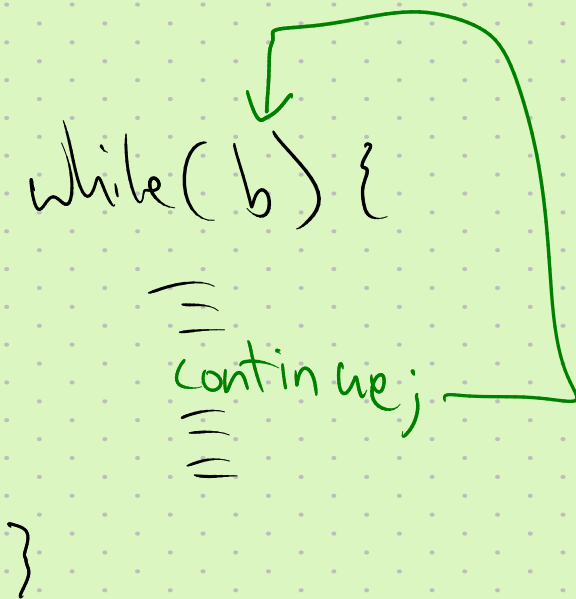
```
while(b) {  
    ==  
    ==  
    ==  
}
```

A black arrow starts from the opening curly brace of the while loop and points down to the closing curly brace, indicating the loop body.

```
while(b) {  
    ==  
    ==  
    break;  
    ==  
}
```

A blue arrow starts from the opening curly brace of the while loop and points down to the closing curly brace, indicating the loop body. A blue arrow also points from the 'break;' statement to the closing curly brace, indicating the exit from the loop.

```
while(b) {  
    ==  
    continue;  
    ==  
}
```

A green arrow starts from the opening curly brace of the while loop and points down to the closing curly brace, indicating the loop body. A green arrow also points from the 'continue;' statement to the opening curly brace, indicating the jump back to the start of the loop.