

# Functions in C++

# Acclimating to C++

# C++ Functions

- Functions in C++ are similar to methods in Java:
  - Piece of code that performs some task.
  - Can accept parameters.
  - Can return a value.
- Syntax similar to Java:

```
return-type function-name(parameters) {  
    /* ... function body ... */  
}
```

Note: no  
**public** or  
**private**.

# The main Function

- A C++ program begins execution in a function called `main` with the following signature:

```
int main() {  
    /* ... code to execute ... */  
    return 0;  
}
```

- By convention, `main` should return 0 unless the program encounters an error.

# A Simple C++ Program

What Went Wrong?

# One-Pass Compilation

- Unlike some languages like Java or C#, C++ has a ***one-pass compiler***.
- If a function has not yet been declared when you try to use it, you will get a compiler error.

# Function Prototypes

- A ***function prototype*** is a declaration that tells the C++ compiler about an upcoming function.
- Function prototypes look like this:  
***return-type function-name(parameters);***
- Place function prototypes high up in your program, typically, before any other functions are defined.
- A function can be used if the compiler has seen either the function itself or its prototype.



# Getting Input from the User

- In C++, we use `cout` to display text.
- We can also use `cin` to receive input.
- For technical reasons, we've written some functions for you that do input.
  - Take CS106L to see why!
- The library "`simpio.h`" contains methods for reading input:

```
int getInteger(string prompt = "");
```

```
double getReal(string prompt = "");
```

```
string getLine(string prompt = "");
```

# Getting Input from the User

- In C++, we use `cout` to display text.
- We can also use `cin` to receive input.
- For technical reasons, we've written some functions for you that do input.
  - Take CS106L to see why!
- The library "`simpio.h`" contains methods for reading input:

```
int getInteger(string prompt = "");  
double getReal(string prompt = "");  
string getLine(string prompt = "");
```

These functions have **default arguments**. If you don't specify a prompt, it will use the empty string.

Let's write some functions!

# Factorials

- The number ***n factorial***, denoted ***n!***, is

$$n \times (n - 1) \times \dots \times 3 \times 2 \times 1$$

- For example:
  - $3! = 3 \times 2 \times 1 = 6.$
  - $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
  - $0! = 1$  (by definition)
- Factorials show up everywhere:
  - Determining how quickly computers can sort values (more on that later this quarter).
  - Counting ways to shuffle a deck of cards.
  - Approximating trig functions on a computer.

# Summing Up Digits

- Ever seen that test for divisibility by three?
  - Add up the digits of the number; if that's divisible by three, the original number is divisible by three (and vice-versa).
- Let's write a function

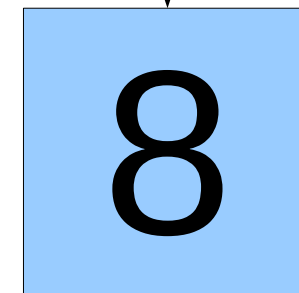
```
int sumOfDigitsOf(int n)
```

that takes in a number and returns the sum of its digits.

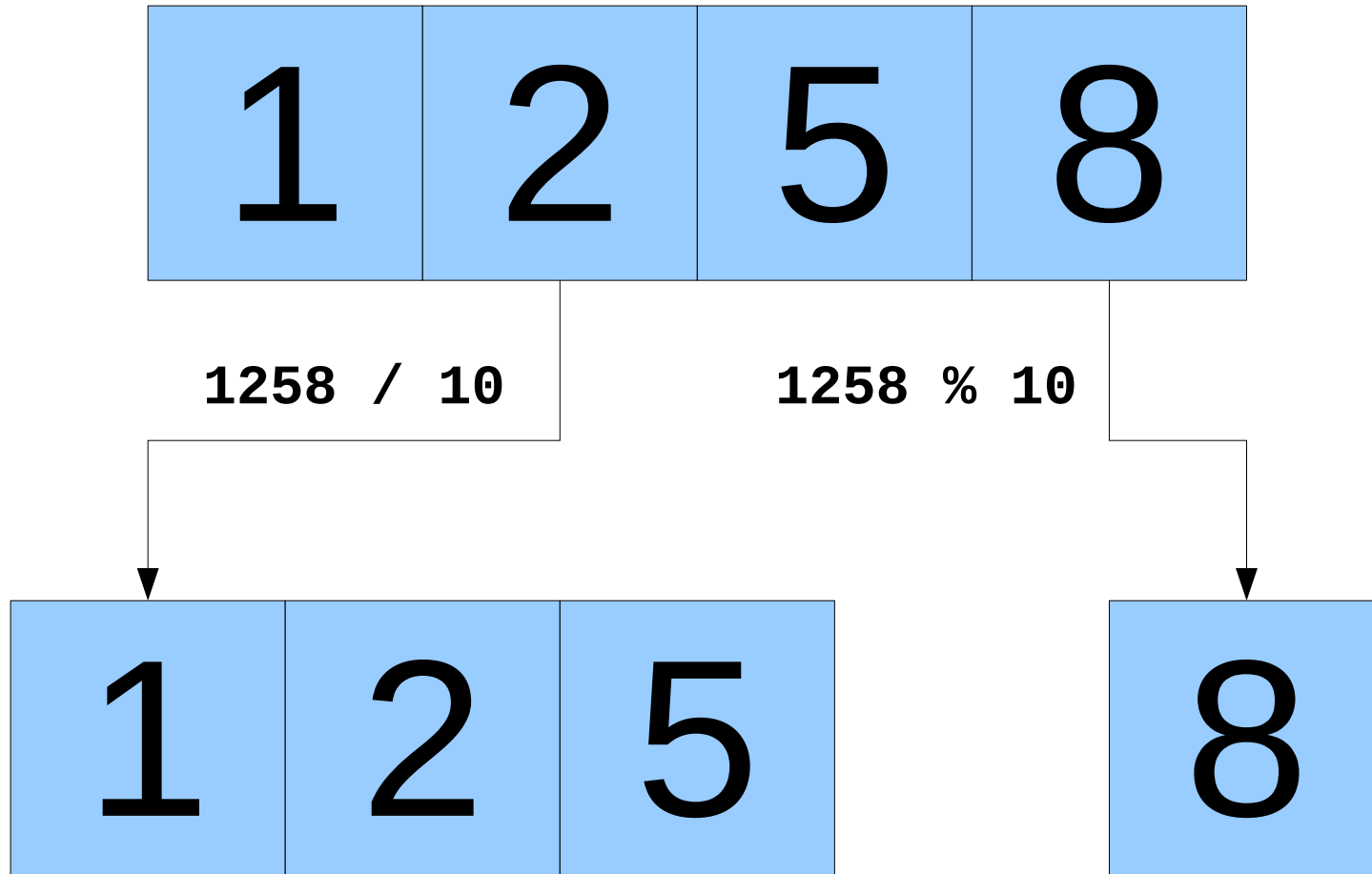
# Working One Digit at a Time



$$1258 \% 10$$



# Working One Digit at a Time



# Digital Roots

- The ***digital root*** is the number you get when you add up the digits of a number and repeat until you get a single digit.
- The digital root of 5 is 5.
- The digital root of 25 is found as follows:

$$2 + 5 = 7$$

So the digital root of 25 is 7.

- The digital root of 137 is found as follows:

$$1 + 3 + 7 = 11$$

$$1 + 1 = 2$$

- So the digital root of 137 is 2.



**Time-Out for Announcements!**

# Section Signups

- Section signups go live tomorrow at 5:00PM and are open until Sunday at 5:00PM.
- Sign up using this link:  
**<http://cs198.stanford.edu/section>**
- You need to sign up here even if you're already enrolled on Axxess; *we don't use Axxess for sections in this class.*

# Qt Creator Help Session

- Having trouble getting Qt Creator set up? We're holding a help session on

***Thursday, 7PM - 9PM***

***Tresidder, First Floor***

- Before you show up, please try troubleshooting using the resources provided on the course website.

# SoE Dean Search Townhall

- The School of Engineering is doing a search for a new dean after Persis Drell's promotion to Provost.
- The SoE is holding a student townhall and are soliciting input.
- Want to have a say in the direction of the School of Engineering? Stop on by!
- Friday, 12:00PM - 1:30PM in the Mackenzie Room (top floor of Huang).
- Can't make it? Send suggestions to

**[deansearch@stanford.edu](mailto:deansearch@stanford.edu)**



# WiCS Frosh Intern Program

> { Curious about CS? Looking for a community on campus? Excited about the WiCS mission? }

> **Apply for the WiCS Frosh Intern program at [bit.ly/frosh-intern](http://bit.ly/frosh-intern)**

> { Frosh interns rotate through different WiCS teams, work on meaningful projects, and join a community of lifelong friends and mentors. }

> **Applications are due Saturday, Jan. 14 at 11:59 PM**



Stanford Women in Computer Science

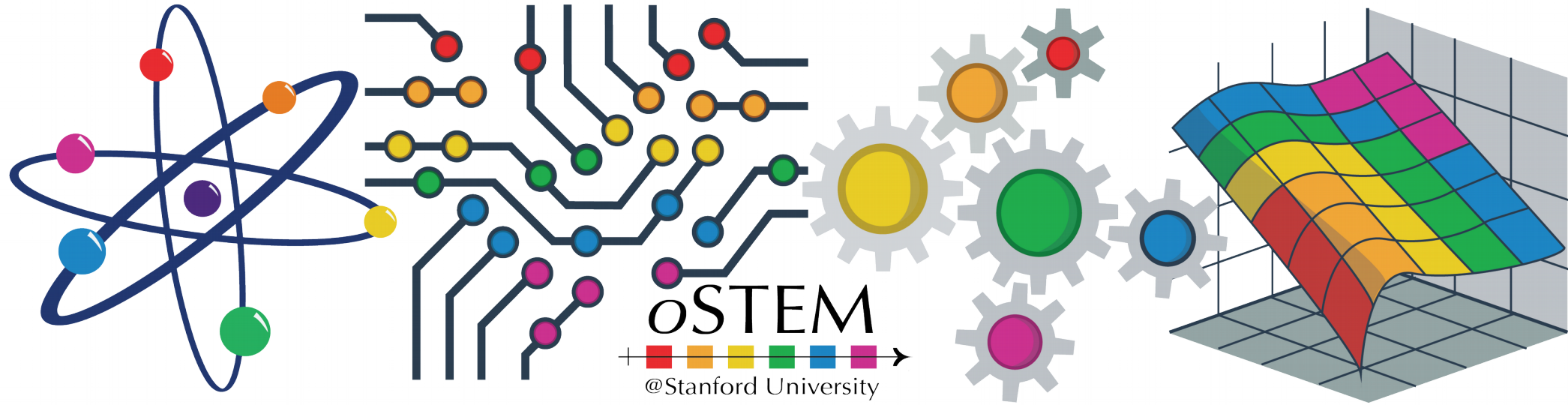


QueeringScience

QueeringTechnology

QueeringEngineering

QueeringMathematics



- Stanford's chapter of oSTEM (Out in STEM) is holding two mixer events:
  - Undergrad mixer: Wednesday, January 18.
  - Graduate mixer: Thursday, January 19.
- Both events are at 6PM at the LGBT-CRC. Dinner is provided!
- Want to get on their mailing list? [Click here!](#)

CS+SOCIAL GOOD PRESENTS:

# SYNTAX & SAGE



*Learn how software, nature,  
art, and urban design collide.*

January 13th, 4-5pm

Old Union, Room 121

Sep Kavmar, a professor at the MIT Media Lab, will be talking about his book, *Syntax and Sage*, which weaves together ideas about software, nature, art, and urban design to show how software shapes the world, how the world shapes people, and how people shape software.

**Back to CS106B!**



# Thinking Recursively

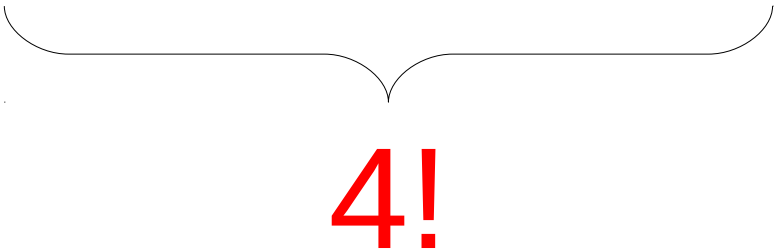
# Factorial Revisited

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

# Factorial Revisited

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

# Factorial Revisited

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$


The diagram illustrates the relationship between 5! and 4!. The numbers 4, 3, 2, and 1 in the product are highlighted in red. A curly brace groups these four numbers, with '4!' written below it, indicating that 4! is the product of 4, 3, 2, and 1.

# Factorial Revisited

$$5! = 5 \times 4!$$

# Factorial Revisited

$$5! = 5 \times 4!$$

# Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3 \times 2 \times 1$$

# Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3 \times 2 \times 1$$



# Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3 \times 2 \times 1$$



$$3!$$

# Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

# Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

# Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2 \times 1$$

# Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

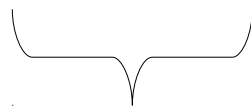
$$3! = 3 \times 2 \times 1$$

# Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2 \times 1$$



**2!**

# Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

# Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$



# Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

# Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1 \times 0!$$

# Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1 \times 0!$$

$$0! = 1$$

# Another View of Factorials

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n - 1)! & \text{otherwise} \end{cases}$$

# Another View of Factorials

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n - 1)! & \text{otherwise} \end{cases}$$

```
int factorial(int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}
```

# Another View of Factorials

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n - 1)! & \text{otherwise} \end{cases}$$

```
int factorial(int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}
```

# Recursion in Action

```
int main() {  
    int n = factorial(5);  
    cout << "5! = " << n << endl;  
}
```

# Recursion in Action

```
int main() {  
    int n = factorial(5);  
    cout << "5! = " << n << endl;  
}
```



# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        if (n == 0) {  
            return 1;  
        } else {  
            return n * factorial(n - 1);  
        }  
    }  
    int n 5  
}
```

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        if (n == 0) {  
            return 1;  
        } else {  
            return n * factorial(n - 1);  
        }  
    }  
}
```

int n 5

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        if (n == 0) {  
            return 1;  
        } else {  
            return n * factorial(n - 1);  
        }  
    }  
    int n 5
```

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        if (n == 0) {  
            return 1;  
        } else {  
            return n * factorial(n - 1);  
        }  
    }  
    int n 5  
}
```

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            if (n == 0) {  
                return 1;  
            } else {  
                return n * factorial(n - 1);  
            }  
        }  
    }  
}
```

int n 4

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            if (n == 0) {  
                return 1;  
            } else {  
                return n * factorial(n - 1);  
            }  
        }  
    }  
}
```

`int n` 4

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            if (n == 0) {  
                return 1;  
            } else {  
                return n * factorial(n - 1);  
            }  
        }  
    }  
}
```

`int n` 4

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            if (n == 0) {  
                return 1;  
            } else {  
                return n * factorial(n - 1);  
            }  
        }  
    }  
}
```

`int n` 4



# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            int factorial(int n) {  
                if (n == 0) {  
                    return 1;  
                } else {  
                    return n * factorial(n - 1);  
                }  
            }  
        }  
    }  
    int n 3
```

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            int factorial(int n) {  
                if (n == 0) {  
                    return 1;  
                } else {  
                    return n * factorial(n - 1);  
                }  
            }  
        }  
    }  
}
```

`int n` 3

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            int factorial(int n) {  
                if (n == 0) {  
                    return 1;  
                } else {  
                    return n * factorial(n - 1);  
                }  
            }  
        }  
    }  
}
```

`int n` 3

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            int factorial(int n) {  
                if (n == 0) {  
                    return 1;  
                } else {  
                    return n * factorial(n - 1);  
                }  
            }  
        }  
    }  
}
```

`int n` 3

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            int factorial(int n) {  
                int factorial(int n) {  
                    if (n == 0) {  
                        return 1;  
                    } else {  
                        return n * factorial(n - 1);  
                    }  
                }  
            }  
        }  
    }  
}
```

`int n` 2

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            int factorial(int n) {  
                int factorial(int n) {  
                    if (n == 0) {  
                        return 1;  
                    } else {  
                        return n * factorial(n - 1);  
                    }  
                }  
            }  
        }  
    }  
}
```

`int n` 2

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            int factorial(int n) {  
                int factorial(int n) {  
                    if (n == 0) {  
                        return 1;  
                    } else {  
                        return n * factorial(n - 1);  
                    }  
                }  
            }  
        }  
    }  
}
```

`int n` 2

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            int factorial(int n) {  
                int factorial(int n) {  
                    if (n == 0) {  
                        return 1;  
                    } else {  
                        return n * factorial(n - 1);  
                    }  
                }  
            }  
        }  
    }  
}
```

`int n` 2



# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            int factorial(int n) {  
                int factorial(int n) {  
                    int factorial(int n) {  
                        if (n == 0) {  
                            return 1;  
                        } else {  
                            return n * factorial(n - 1);  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

`int n` 1

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            int factorial(int n) {  
                int factorial(int n) {  
                    int factorial(int n) {  
                        if (n == 0) {  
                            return 1;  
                        } else {  
                            return n * factorial(n - 1);  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

`int n` 1

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            int factorial(int n) {  
                int factorial(int n) {  
                    int factorial(int n) {  
                        if (n == 0) {  
                            return 1;  
                        } else {  
                            return n * factorial(n - 1);  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

`int n` 1

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            int factorial(int n) {  
                int factorial(int n) {  
                    int factorial(int n) {  
                        if (n == 0) {  
                            return 1;  
                        } else {  
                            return n * factorial(n - 1);  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

`int n` 1

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            int factorial(int n) {  
                int factorial(int n) {  
                    int factorial(int n) {  
                        int factorial(int n) {  
                            if (n == 0) {  
                                return 1;  
                            } else {  
                                return n * factorial(n - 1);  
                            }  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

`int n` 0

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            int factorial(int n) {  
                int factorial(int n) {  
                    int factorial(int n) {  
                        int factorial(int n) {  
                            if (n == 0) {  
                                return 1;  
                            } else {  
                                return n * factorial(n - 1);  
                            }  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

`int n` 0

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            int factorial(int n) {  
                int factorial(int n) {  
                    int factorial(int n) {  
                        int factorial(int n) {  
                            if (n == 0) {  
                                return 1;  
                            } else {  
                                return n * factorial(n - 1);  
                            }  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

`int n` 0

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            int factorial(int n) {  
                int factorial(int n) {  
                    int factorial(int n) {  
                        if (n == 0) {  
                            return 1;  
                        } else {  
                            return n * factorial(n - 1);  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

1

int n 1



# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            int factorial(int n) {  
                int factorial(int n) {  
                    if (n == 0) {  
                        return 1;  
                    } else {  
                        return n * factorial(n - 1);  
                    }  
                }  
            }  
        }  
    }  
}
```

The diagram illustrates the execution of the recursive factorial function. The code is shown in nested boxes representing function calls. A yellow box with the number 1 is connected by a line to the return statement `return n * factorial(n - 1);` in the innermost call. A blue box with the number 2 is next to the `int n` declaration in the same call.

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            int factorial(int n) {  
                if (n == 0) {  
                    return 1;  
                } else {  
                    return n * factorial(n - 1);  
                }  
            }  
        }  
    }  
}
```

The diagram illustrates the execution of a recursive factorial function. The code is shown in four overlapping boxes, representing the call stack. The innermost box shows the function call for `factorial(3)`, where the variable `n` is 3. A red box highlights the recursive call `return n * factorial(n - 1);`. A yellow box contains the number 2, representing the result of the recursive call `factorial(2)`. A blue box contains the number 3, representing the current value of `n` in the outermost frame.

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        int factorial(int n) {  
            if (n == 0) {  
                return 1;  
            } else {  
                return n * factorial(n - 1);  
            }  
        }  
    }  
}
```

The diagram illustrates the recursive call for `factorial(4)`. A yellow box containing the number `6` is connected by a line to the recursive call `factorial(n - 1)` in the code. A blue box containing the number `4` is positioned next to the parameter `n` in the same code line.

# Recursion in Action

```
int main() {  
    int factorial(int n) {  
        if (n == 0) {  
            return 1;  
        } else {  
            return n * factorial(n - 1);  
        }  
    }  
}
```

The diagram illustrates a recursive call in a factorial function. A yellow box containing the number 24 is connected by a line to the recursive call `factorial(n - 1)` in the code. A blue box containing the number 5 is positioned next to the variable `n` in the function signature, representing the current value of `n`.

# Recursion in Action

```
int main() {  
    int n = factorial(5);  
    cout << "5! = " << n << endl;  
}
```

int n 120

# Recursion in Action

```
int main() {  
    int n = factorial(5);  
    cout << "5! = " << n << endl;  
}
```

int n 120

# Thinking Recursively

- Solving a problem with recursion requires two steps.
- First, determine how to solve the problem for simple cases.
  - This is called the ***base case***.
- Second, determine how to break down larger cases into smaller instances.
  - This is called the ***recursive step***.

# Summing Up Digits

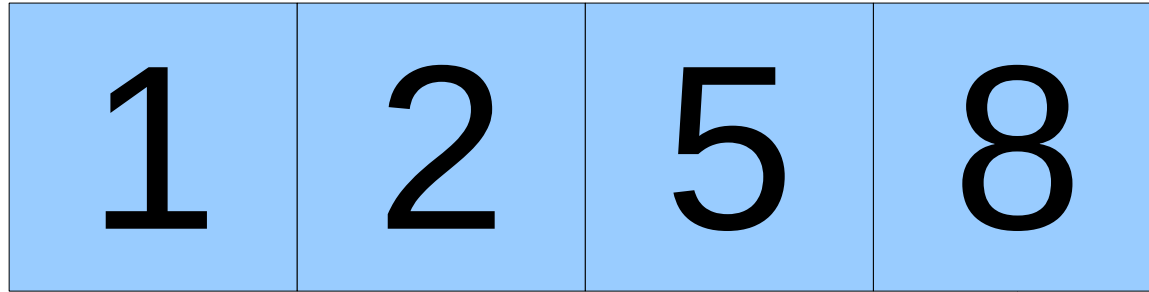
- One way to compute the sum of the digits of a number is shown here:

```
int sumOfDigitsOf(int n) {  
    int result = 0;  
    while (n != 0) {  
        result += n % 10;  
        n /= 10;  
    }  
    return result;  
}
```

- How would we rewrite this function recursively?



# Summing Up Digits



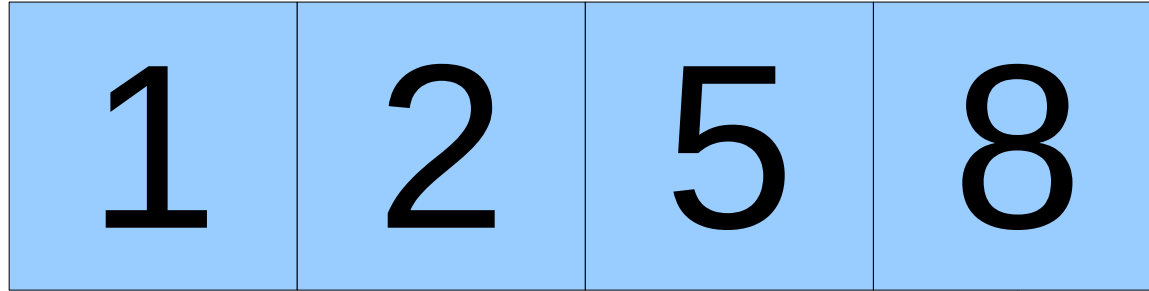
The sum of the digits of  
this number is equal to...

the sum of the digits of  
this number...

plus this number.



# Summing Up Digits



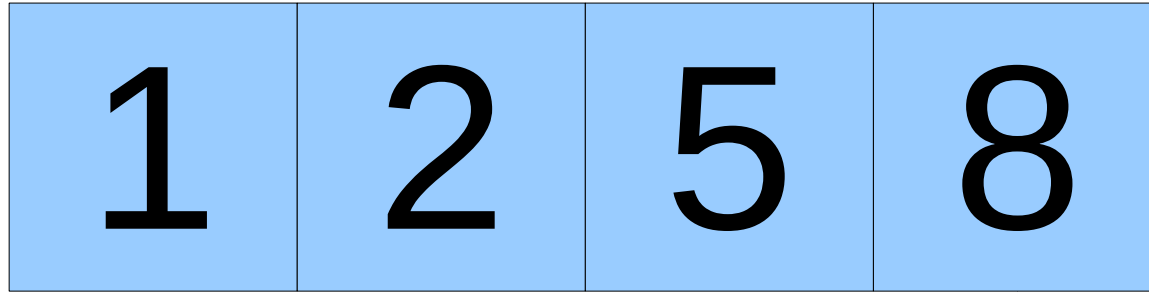
sumofDigitsof(n)  
is equal to...

the sum of the digits of  
this number...

plus this number.



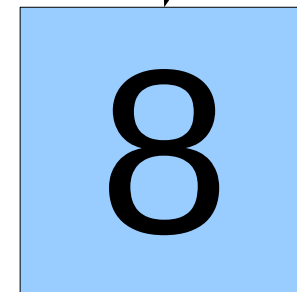
# Summing Up Digits



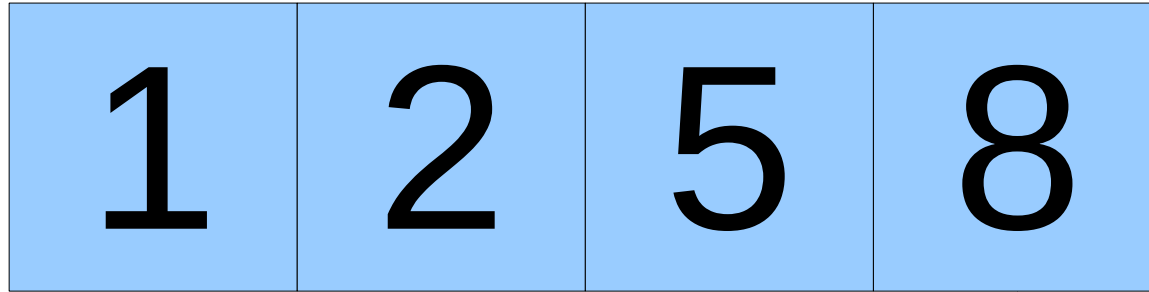
sumofDigitsof(n)  
is equal to...

sumofDigitsof(n / 10)

plus this number.



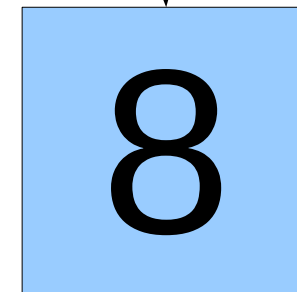
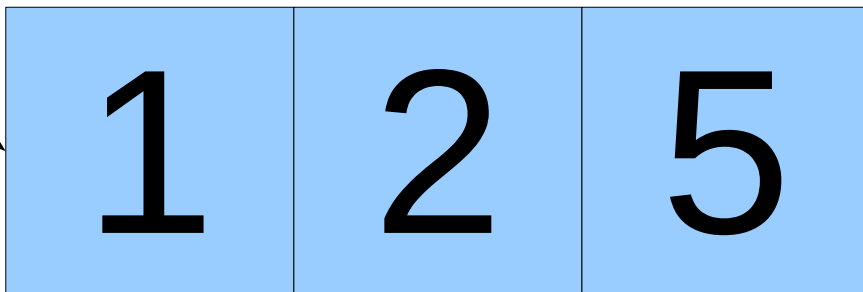
# Summing Up Digits



sumofDigitsof(n)  
is equal to...

sumofDigitsof(n / 10)

+ (n % 10)



# Summing Up Digits

- Here's the recursive rewrite we just came up with:

```
int sumOfDigitsOf(int n) {  
    if (n < 10) {  
        return n;  
    } else {  
        return sumOfDigitsOf(n / 10) + (n % 10);  
    }  
}
```

- Notice the structure:
  - If the problem is sufficiently simple, solve it directly.
  - Otherwise, reduce it to a smaller instance and solve that one.

# Tracing the Recursion

```
int main() {  
    int sum = sumOfDigitsOf(137);  
    cout << "Sum is " << sum << endl;  
}
```

# Tracing the Recursion

```
int main() {  
    int sum = sumOfDigitsOf(137);  
    cout << "Sum is " << sum << endl;  
}
```

# Tracing the Recursion

```
int main() {  
    int sumOfDigitsOf(int n) {  
        if (n < 10) {  
            return n;  
        } else {  
            return sumOfDigitsOf(n / 10) + (n % 10);  
        }  
    }  
}
```

int n 137



# Tracing the Recursion

```
int main() {  
    int sumOfDigitsOf(int n) {  
        if (n < 10) {  
            return n;  
        } else {  
            return sumOfDigitsOf(n / 10) + (n % 10);  
        }  
    }  
}
```

int n 137

# Tracing the Recursion

```
int main() {  
    int sumOfDigitsOf(int n) {  
        if (n < 10) {  
            return n;  
        } else {  
            return sumOfDigitsOf(n / 10) + (n % 10);  
        }  
    }  
}
```

int n 137

# Tracing the Recursion

```
int main() {
```

```
    int sumOfDigitsOf(int n) {
```

```
        if (n < 10) {
```

```
            return n;
```

```
        } else {
```

```
            return sumOfDigitsOf(n / 10) + (n % 10);
```

```
        }
```

```
    }
```

```
int n 137
```

# Tracing the Recursion

```
int main() {
```

```
    int sumOfDigitsOf(int n) {
```

```
        if (n < 10) {
```

```
            return n;
```

```
        } else {
```

```
            return sumOfDigitsOf(n / 10) + (n % 10);
```

```
        }
```

```
    }
```

```
int n 137
```

# Tracing the Recursion

```
int main() {  
    int sumOfDigitsOf(int n) {  
        int sumOfDigitsOf(int n) {  
            if (n < 10) {  
                return n;  
            } else {  
                return sumOfDigitsOf(n / 10) + (n % 10);  
            }  
        }  
    }  
}
```

int n 13

# Tracing the Recursion

```
int main() {
```

```
    int sumOfDigitsOf(int n) {
```

```
        int sumOfDigitsOf(int n) {
```

```
            if (n < 10) {
```

```
                return n;
```

```
            } else {
```

```
                return sumOfDigitsOf(n / 10) + (n % 10);
```

```
            }
```

```
        }
```

```
int n 13
```

```
107
```

# Tracing the Recursion

```
int main() {
```

```
    int sumOfDigitsOf(int n) {
```

```
        int sumOfDigitsOf(int n) {
```

```
            if (n < 10) {
```

```
                return n;
```

```
            } else {
```

```
                return sumOfDigitsOf(n / 10) + (n % 10);
```

```
            }
```

```
        }
```

```
int n 13
```

```
107
```

# Tracing the Recursion

```
int main() {
```

```
    int sumOfDigitsOf(int n) {
```

```
        int sumOfDigitsOf(int n) {
```

```
            if (n < 10) {
```

```
                return n;
```

```
            } else {
```

```
                return sumOfDigitsOf(n / 10) + (n % 10);
```

```
            }
```

```
        }
```

```
int n 13
```



# Tracing the Recursion

```
int main() {
```

```
    int sumOfDigitsOf(int n) {
```

```
        int sumOfDigitsOf(int n) {
```

```
            if (n < 10) {
```

```
                return n;
```

```
            } else {
```

```
                return sumOfDigitsOf(n / 10) + (n % 10);
```

```
            }
```

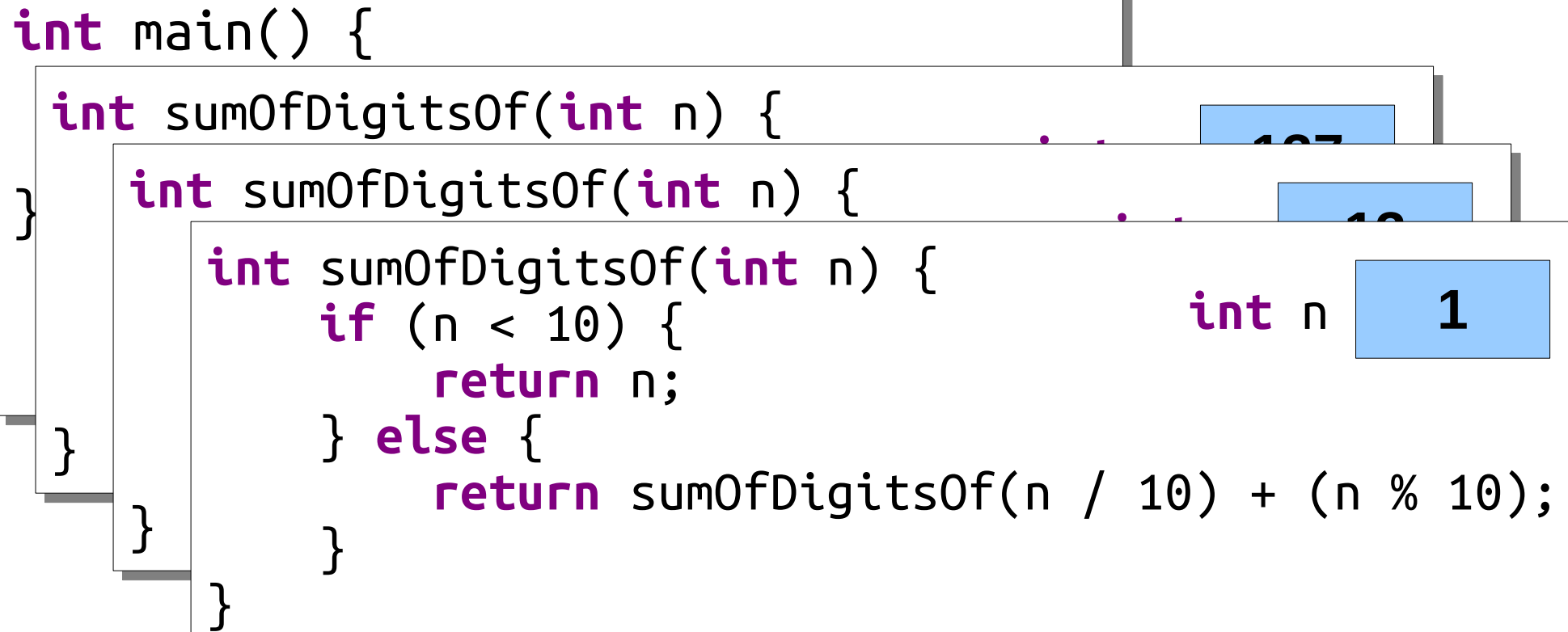
```
        }
```

```
int n 13
```

```
107
```

# Tracing the Recursion

```
int main() {  
    int sumOfDigitsOf(int n) {  
        int sumOfDigitsOf(int n) {  
            int sumOfDigitsOf(int n) {  
                if (n < 10) {  
                    return n;  
                } else {  
                    return sumOfDigitsOf(n / 10) + (n % 10);  
                }  
            }  
        }  
    }  
}
```



The diagram illustrates the recursive calls for the function `sumOfDigitsOf` with the input `107`. The calls are shown as overlapping boxes, representing the call stack. The top-most box shows the initial call with `n = 107`. The next box shows the call with `n = 10`. The next box shows the call with `n = 1`. The bottom-most box shows the call with `n = 1`, where the value `1` is highlighted in a blue box, indicating the base case of the recursion.

# Tracing the Recursion

```
int main() {  
    int sumOfDigitsOf(int n) {  
        int sumOfDigitsOf(int n) {  
            int sumOfDigitsOf(int n) {  
                if (n < 10) {  
                    return n;  
                } else {  
                    return sumOfDigitsOf(n / 10) + (n % 10);  
                }  
            }  
        }  
    }  
}
```

int n 1

# Tracing the Recursion

```
int main() {  
    int sumOfDigitsOf(int n) {  
        int sumOfDigitsOf(int n) {  
            int sumOfDigitsOf(int n) {  
                if (n < 10) {  
                    return n;  
                } else {  
                    return sumOfDigitsOf(n / 10) + (n % 10);  
                }  
            }  
        }  
    }  
}
```

The diagram illustrates the recursive calls for the function `sumOfDigitsOf` with the input `107`. The stack frames are shown as overlapping boxes, representing the call stack. The top frame shows the initial call with `n = 107`. The middle frame shows the recursive call with `n = 10`. The bottom frame shows the base case with `n = 1`. The `return n;` statement in the bottom frame is highlighted with a blue box, indicating the return value for the base case.

# Tracing the Recursion

```
int main() {
```

```
    int sumOfDigitsOf(int n) {
```

```
        int sumOfDigitsOf(int n) {
```

```
            if (n < 10) {
```

```
                return n;
```

```
            } else {
```

```
                return sumOfDigitsOf(n / 10) + (n % 10);
```

**1**

int n **13**

# Tracing the Recursion

```
int main() {
```

```
    int sumOfDigitsOf(int n) {
```

```
        int sumOfDigitsOf(int n) {
```

```
            if (n < 10) {
```

```
                return n;
```

```
            } else {
```

```
                return sumOfDigitsOf(n / 10) + (n % 10);
```

**1**

int n 13

# Tracing the Recursion

```
int main() {
```

```
    int sumOfDigitsOf(int n) {
```

```
        int sumOfDigitsOf(int n) {
```

```
            if (n < 10) {
```

```
                return n;
```

```
            } else {
```

```
                return sumOfDigitsOf(n / 10) + (n % 10);
```

**1**

**+**

**3**

int n **13**

# Tracing the Recursion

```
int main() {
```

```
    int sumOfDigitsOf(int n) {
```

```
        if (n < 10) {
```

```
            return n;
```

```
        } else {
```

```
            return sumOfDigitsOf(n / 10) + (n % 10);
```

```
        }
```

```
    }
```

```
int n 137
```

4



# Tracing the Recursion

```
int main() {
```

```
    int sumOfDigitsOf(int n) {
```

```
        if (n < 10) {
```

```
            return n;
```

```
        } else {
```

```
            return sumOfDigitsOf(n / 10) + (n % 10);
```

```
        }
```

```
    }
```

```
int n 137
```

4

# Tracing the Recursion

```
int main() {
```

```
    int sumOfDigitsOf(int n) {
```

```
        if (n < 10) {
```

```
            return n;
```

```
        } else {
```

```
            return sumOfDigitsOf(n / 10) + (n % 10);
```

```
        }
```

```
    }
```

int n 137

4

+

7

# Tracing the Recursion

```
int main() {  
    int sum = sumOfDigitsOf(137);  
    cout << "Sum is " << sum << endl;  
}
```

**11**

# Recap from Today

- In C++, to call a function, that function must either be defined above the call site or **prototyped** before the call site.
- Execution in C++ programs begins in a function called `main`.
- You can split a number into “everything but the last digit” and “the last digit” by dividing and modding by 10.
- A **recursive function** is one that calls itself. It has a **base case** to handle easy cases and a **recursive step** to turn bigger versions of the problem into smaller ones.
- Functions can be written both iteratively and recursively.

# Your Action Items

- Read Chapter 1 and Chapter 2 of the textbook for more background on C++ basics and writing functions.
- Read Chapter 7 of the textbook for more exposition on recursion.
- Keep an eye out for the section signup link, which will get sent out tomorrow afternoon.
- Aim to complete Assignment 0 by Friday.
  - Just under a third of you are already done! Exciting!

# Next Time

- ***Strings and Streams***
  - Representing and manipulating text.
  - File I/O in C++.