Learning Objectives: Recursion

- **Define recursion**
- **Identify the base case**
- **Identify the recursive pattern**

What is Recursion

What is Recursion?

Solving a coding problem with functions involves breaking down the problem into smaller problems. When these smaller problems are variations of the larger problem (also know as self-similar), then recursion can be used. For example, the mathematical function factorial is self-similar. Five factorial (`5!`) is calculated as `5 * 4 * 3 * 2 * 1`. Mouse over the image below to see that `5!` is really just `5 * 4!`, and `4!` is really just `4 * 3!` and so on.

Because `5!` is self-similar, recursion can be used to calculate the answer. Recursive functions are functions that call themselves. Copy the following code into the text editor on your left and click `TRY IT` to test the code.

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

//add function definitions below this line

/**
 * Calculates factorial using recursion
 *
 * @param n, integer
 * @return factorial of n, integer
 */
int Factorial(int n) {
  if (n == 1) {
    return 1;
  }
  else {
    return n * Factorial(n - 1);
  }
}

//add function definitions above this line

int main() {

  //add code below this line

  cout << Factorial(5) << endl;
  return 0;

  //add code above this line

}
```

Optional: Click the Code Visualizer link below to see how C++ handles this recursive function behind-the-scenes.

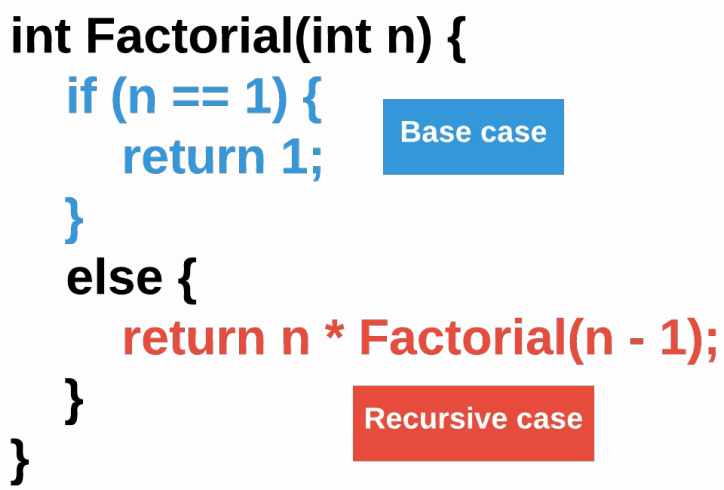
Code Visualizer

Recursion is an abstract and difficult topic, so it might be a bit hard to follow what is going on here. When `n` is 5, C++ starts a multiplication problem of `5 * Factorial(4)`. The function runs again and the multiplication problem becomes `5 * 4 * Factorial(3)`. This continues until `n` is 1. C++ returns the value `1`, and C++ solves the multiplication problem `5 * 4 * 3 * 2 * 1`. The video below should help explain how `5!` is calculated recursively.

Error

The Base Case

Each recursive function has two parts: the recursive case (where the function calls itself with a different parameter) and the base case (where the function stops calling itself and returns a value).



.guides/img/CppRecursion

The base case is the most important part of a recursive function. Without it, the function will never stop calling itself. Like an infinite loop, C++ will stop the program with an error. Replace the function in your code with the one below and see what happens.

```
/**
 * This recursive function returns an error
 *
 * @param n, integer
 * @return factorial of n, integer
 */
int Factorial(int n) {
  return n * Factorial(n - 1);
}
```

Code Visualizer

Always start with the base case when creating a recursive function. Each time the function is called recursively, the program should get one step closer to the base case.

challenge

What happens if you:

- Change the `Factorial()` function to look like:

```
int Factorial(int n) {
  if (n == 1) { //base case
    return 1;
  }
  else { //recursive case
    return n * Factorial(n - 1);
  }
}
```

- Change the print statement to `cout << Factorial(0) << endl;`?
- Try to modify the base case so that `Factorial(0)` does not result in an error.
- Test your new base case with a negative number.

▼

Solution

The factorial operation only works with positive integers. So the base case should be:

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

Code Visualizer

Fibonacci Sequence

Fibonacci Number

A Fibonacci number is a number in which the current number is the sum of the previous two Fibonacci numbers.

The Fibonacci sequence is defined as $F_n = F_{n-1} + F_{n-2}$ where "F" is the function and "n" is the parameter

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	...
0	1	1	2	3	5	8	13	21	...

Fibonacci Sequence

Calculating a Fibonacci number is self-similar, which means it can be defined with recursion. Setting the base case is important to avoid infinite recursion. When the number `n` is 0 the Fibonacci number is 0, and when `n` is 1 the Fibonacci number is 1. So if `n` is less than or equal to 1, then return `n`. That is the base case.

```
/**
* @param n, integer
* @return Fibonacci number of n, integer
*/
int Fibonacci(int n) {
  if (n <= 1) {
    return n;
  }
  else {
    return(Fibonacci(n-1) + Fibonacci(n-2));
  }
}

int main() {
  cout << Fibonacci(3) << endl;
  return 0;
}
```

Code Visualizer

challenge

What happens if you:

- Change the print statement to `cout << Fibonacci(0) << endl;`?
- Change the print statement to `cout << Fibonacci(8) << endl;`?
- Change the print statement to `cout << Fibonacci(30) << endl;`?

▼ **Where is the code visualizer?**

The code visualizer will only step through your code 1,000 times. These recursive functions exceed this limit and generate an error message. Because of this, the code visualizer was removed.

Fibonacci Sequence

Fibonacci numbers are most often talked about as a sequence. The `main()` function below adds the functionality of printing a Fibonacci sequence of a predetermined length. Replace your current `main()` function with the one below and `TRY IT`.

```
int main() {
  int fibonacci_length = 4;
  for (int i = 0; i < fibonacci_length; i++) {
    cout << Fibonacci(i) << endl;
  }
  return 0;
}
```

Code Visualizer

challenge

What happens if you:

- Change `fibonacci_length` to `10`?
- Change `fibonacci_length` to `30`?
- Change `fibonacci_length` to `50`?

▼ **Why is C++ timing out?**

The code written above is terribly inefficient. Each time through the loop, C++ is calculating the same Fibonacci numbers again and again. When `i` is 1, C++ calculates the Fibonacci numbers for 0 and 1. When `i` is 2, C++ is calculating the Fibonacci numbers for 0, 1, and 2. Once `i` becomes large enough, it becomes too much work for C++ to have to recalculate these large numbers over and over again. There is a more efficient way to do this by using vector. The idea is to store previously calculated Fibonacci numbers in the vector. So instead of recalculating the same numbers again and again, you can get these numbers from the vector. If a Fibonacci number is not in the vector, then calculate it and add it to the vector. Copy and paste the code below into the IDE if you want to run it.

NOTE: `long` is a data type that can hold much larger values than `int` can. Thus, for larger numbers, `long` is necessary.

```
long Fibonacci(long n) {
  static vector<long> v = {0, 1};

  if (n < v.size())
    return v.at(n);
  else {
    v.push_back(Fibonacci(n - 1) + Fibonacci(n - 2));
    return v.at(n);
  }
}

int main() {
  int fib_length = 50;
    for (int i = 0; i < fib_length; i++) {
    cout << Fibonacci(i) << endl;
  }
  return 0;
}
```