Outline

- In this presentation, we will:
  - Describe recursively defined functions
  - Overview problems that can be solved recursive
  - Look at mathematical problems defined recursively
    - Factorial
    - Binomial coefficients
    - Fibonacci numbers
    - Integer exponentiation
    - Ackermann function
    - Greatest common divisor
  - Consider the effect on the call stack and describe tail recursion

Recursion

- In the previous topic, we saw how a function could call itself
  - If $\pi < x \leq 2\pi$, you can calculates $\sin(x)$ by calculating $-\sin(x - \pi)$
  - If $\pi/2 < x \leq \pi$, you can calculates $\sin(x)$ by calculating $\sin(\pi/2 - x)$

- When a function calls itself
  - That call is said to be a *recursive* call
  - The process is described as *recursion*

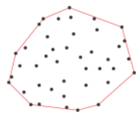
- Etymology:
  - From the Latin verb *recurrere* meaning
    "to run back" or "to run again"
  - From English verb recur meaning
    "to occur again periodically or repeatedly"

Recursion

- Many larger problems can be solved or calculated by solving a similar, but simpler problem through the same means
  - Consider the high-low game:
    - You must guess a number from one to one million
  - Strategy:
    - Guess 500,000 and
      - If you're right, nicely done…
      - If you are told that is low, guess 750,000
      - If you are told that it is high, guess 250,000
  - Guessing one number in one million is reduced to
    guessing one number in 500,000, which is then reduced to
    guessing one number in 250,000, which is then reduced to
    guessing one number in 125,000, etc., etc.
- A maximum of 20 guesses is required

Recursion

- Many algorithms for solving problems can be recursively defined:
  - Searching a list of sorted numbers
  - Sorting a list of unsorted numbers
  - Fast integer multiplication
  - Given random points scattered in the plane, find the pair that are closest to each other
  - The fast Fourier transform
  - Matrix-matrix multiplication
  - Searching for a file within a directory
  - Searching a maze
  - Finding the *convex hull* of a set of points



Recursion

- We will focus on mathematical recursion:
  - Mathematical expressions that are recursively defined
- Examples:
  - Factorial
  - Binomial coefficients
  - Greatest common divisor
- Later in this course, we will see a binary search

Recursion

- A problem that is defined recursively must:
  - For some inputs, the problem must be easily solvable
    - These are called *base cases*
  - For other inputs, the problem can be solved by first solving an easier problem, the solution to which can be used to solve the more current problem
    - These are the *recursive cases*
  - The more complex problems must ultimately lead to the easier problems

Recursion

- Format:

```
typename recursion( parameters… ) {
    // Base cases
    if ( condition-for-base-cases ) {
        // Calculate and return the solution...
        return solution;
    }

    // Recursive cases
    //  - determine simpler cases
    //  - will at some point call
    //        recursion( simpler-case-arguments… );
    return solution;
}
```

Factorial

- The factorial can be defined recursively:

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n \geq 0 \end{cases}$$

- This defines $n!$ in terms of $(n-1)!$
  - To calculate $n!$, you must calculate $(n-1)!$
  - To calculate $(n-1)!$, you must calculate $(n-2)!$
  - At some point, you need a result, and in this case, it is $0! = 1$

Factorial

- For example:

$5! = 5 \times 4!$	but $4! = 4 \times 3!$	so $1! = 1 \times 1 = 1$
	but $3! = 3 \times 2!$	so $2! = 2 \times 1 = 2$
	but $2! = 2 \times 1!$	so $3! = 3 \times 2 = 6$
	but $1! = 1 \times 0!$	so $4! = 4 \times 6 = 24$
	but $0! = \mathbf{1}$	so $5! = 5 \times 24 = 120$

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n \geq 0 \end{cases}$$

Factorial

- Implementing this

```
unsigned int factorial( unsigned int n ) {
    // Base case
    if ( n == 0 ) {
        return 1;
    }

    // Recursive case
    return n*factorial( n - 1 );
}
```

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n \geq 0 \end{cases}$$

Binomial coefficients

- The binomial coefficient $\binom{n}{k}$ is defined as:
  - The number of ways k items can be chosen from n unique items
  - Related to Pascal's triangle
- A recursive definition is:

$$\binom{n}{k} = \begin{cases} 0 & k < 0 \text{ or } k > n \\ 1 & k = 0 \text{ or } k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1} & 0 < k < n \end{cases}$$

Binomial coefficients

- Implementing this

```
unsigned int binomial( unsigned int n, unsigned int k ) {
    // Special cases:
    if ( k > n ) {
        return 0;
    }

    // Base cases:
    if ( (k == 0) || (k == n) ) {
        return 1;
    }

    // Recursive case:
    return binomial( n - 1, k ) + binomial( n - 1, k - 1 );
}
```

$$\binom{n}{k} = \begin{cases} 0 & k < 0 \text{ or } k > n \\ 1 & k = 0 \text{ or } k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1} & 0 < k < n \end{cases}$$

Greatest common divisor

- For $m \ge n \ge 0$, let gcd(m, n) be their greatest common divisor
- Because $m \ge n$, it follows $m = an + r$
  - Consequently, gcd(m, n) must also divide both n and r
  - Thus, $\gcd(m,n) = \begin{cases} m & n = 0 \\ \gcd(n, m \bmod n) & n > 0 \end{cases}$

```
unsigned int gcd( unsigned int m, unsigned int n ) {
    // Special case:
    if ( n > m ) {
        return gcd( n, m );
    }

    if ( n == 0 ) {
        return m;
    } else {
        return gcd( n, m % n );
    }
}
```


Other recursively defined functions

- Fibonacci numbers are defined recursively:

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n \ge 2 \end{cases}$$

- The Ackerman function looks simple, but leads to astronomically large values:

$$A(m,n) = \begin{cases} n+1 & m = 0 \\ A(m-1,1) & m \ge 1 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & m \ge 1 \text{ and } n \ge 1 \end{cases}$$

Other recursively defined functions

- Integer exponentiation may be defined recursive in two ways:

$$x^n = \begin{cases} 1 & n = 0 \\ \dfrac{1}{x^{-n}} & n < 0 \\ x \cdot x^{n-1} & n > 0 \end{cases} \qquad x^n = \begin{cases} 1 & n = 0 \\ \dfrac{1}{x^{-n}} & n < 0 \\ \left(x^m\right)^2 & n = 2m \text{ with } m > 0 \\ x \cdot \left(x^m\right)^2 & n = 2m+1 \text{ with } m \ge 0 \end{cases}$$

  - You should implement both of these versions

The call stack and tail recursion

- When we calculated *n*!, we performed a calculation with what was returned:

```
return n*factorial( n - 1 );
```

- We need to store the current parameter value of `n` to calculate the value after we calculate `factorial( n - 1 )`

1		return value for `factorial(0)`
1	n	**parameter for `factorial(1)`**
2	n	parameter for `factorial(2)`
3	n	parameter for `factorial(3)`
4	n	parameter for `factorial(4)`
5	n	parameter for `factorial(5)`
6	n	parameter for `factorial(6)`
7	n	parameter for `factorial(7)`
…		local variable for `main()`

The call stack and tail recursion

- Notice that when we call the recursive gcd, the return statement is just another function call:

```
return gcd( n, m % n );
```

  – The return value is whatever the called function returns
    - No parameters or local variables of the current call to gcd are required—why keep the memory
    - Some compilers will simply reuse the memory already on the stack
      – This is called *tail recursion*

The call stack and tail recursion

- For example, in calculating gcd(15, 9), when calling gcd(9, 6), it will reuse the memory on the call stack as opposed to adding to it

```
int main() {
    std::cout << gcd( 15, 9 ) << std::endl;
    return 0;
}
```

9	n	parameters for `gcd(15, 9)`
15	m	
…		local variable for `main()`

The call stack and tail recursion

- In calculating gcd(9, 6), when calling gcd(6, 3), it will again reuse the memory on the call stack as opposed to adding to it

6	n	parameters for `gcd(9, 6)`
9	m	
…		local variable for `main()`

The call stack and tail recursion

- In calculating gcd(6, 3), when calling gcd(3, 0), it will again reuse the memory on the call stack as opposed to adding to it

3	n	parameters for `gcd(6, 3)`
6	m	
…		local variable for `main()`

The call stack and tail recursion

- In calculating gcd(3, 0), because the 2nd argument is 0, it returns 3

0	n	parameters for `gcd(3, 0)`
3	m	
…		local variable for `main()`

The call stack and tail recursion

- The return value of 3 is put on the top of the stack, and thus is returned to `main()`

3	return value for `gcd(3, 0)`
…	local variable for `main()`

The call stack and tail recursion

- Tail recursion can be used to minimize memory for recursive calls:

```
typename tail_recursion( parameters… ) {
    // Base cases
    if ( condition-for-base-cases ) {
        // Calculate and return the solution...
        return solution;
    }

    // Recursive cases
    //  - determine simpler arguments
    return tail_recursion( simpler-case-arguments… );
}
```

Summary

- After this lesson, you now
  - Understand the idea of literal data in source code
  - You understand how to encode
    - Integers
    - Characters
    - Strings
    - Floating-point numbers (reals)
    - Boolean

    in your source code
  - Everything else in C++ deals with the storage and manipulation of data

References

[1] Wikipedia
https://en.wikipedia.org/wiki/Recurrence_relation
https://en.wikipedia.org/wiki/Recursion_(computer_science)

Acknowledgments

Proof read by Dr. Thomas McConkey

Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.

Disclaimer