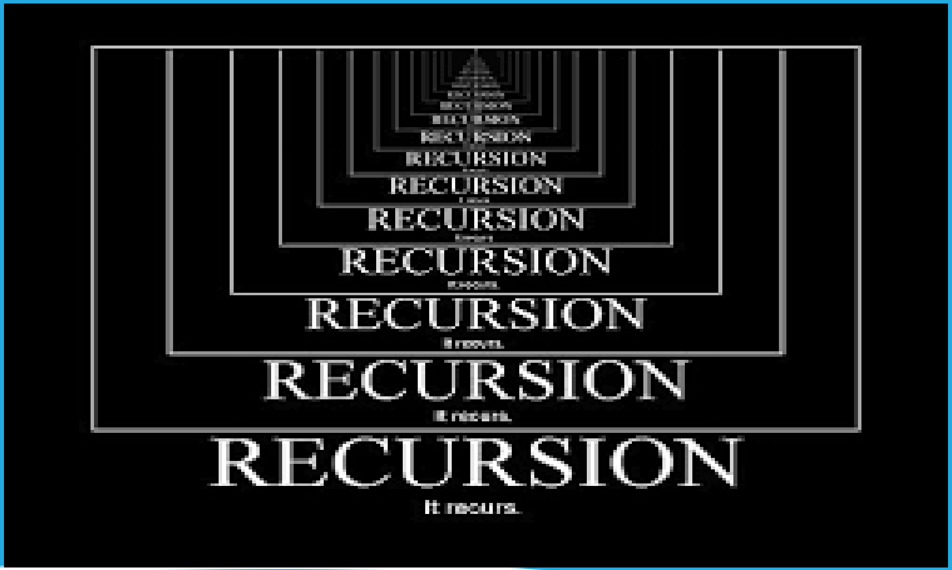
INTRODUCTION TO PROGRAMING

Chapter 7

Recursion



Khoa Công Nghệ Thông Tin
Trường Đại Học Khoa Học Tự Nhiên
ĐHQG-HCM

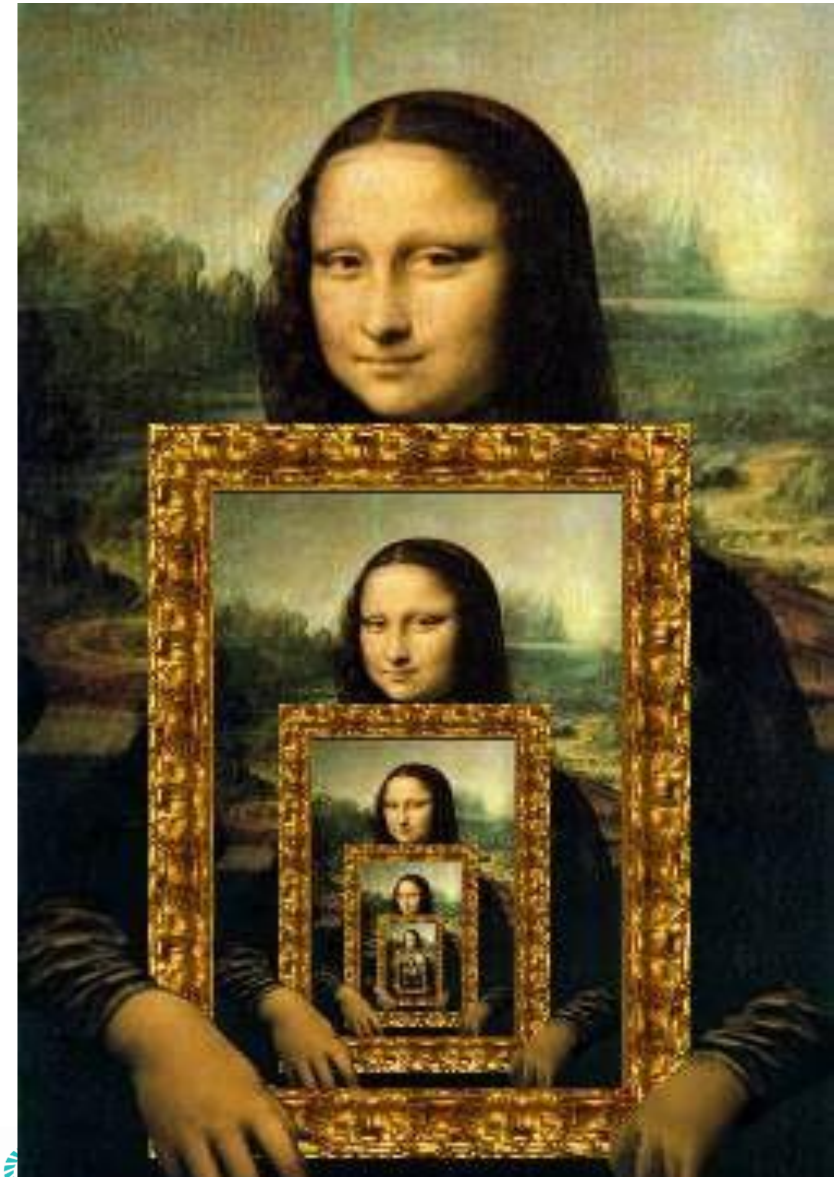
GV: Thái Hùng Văn

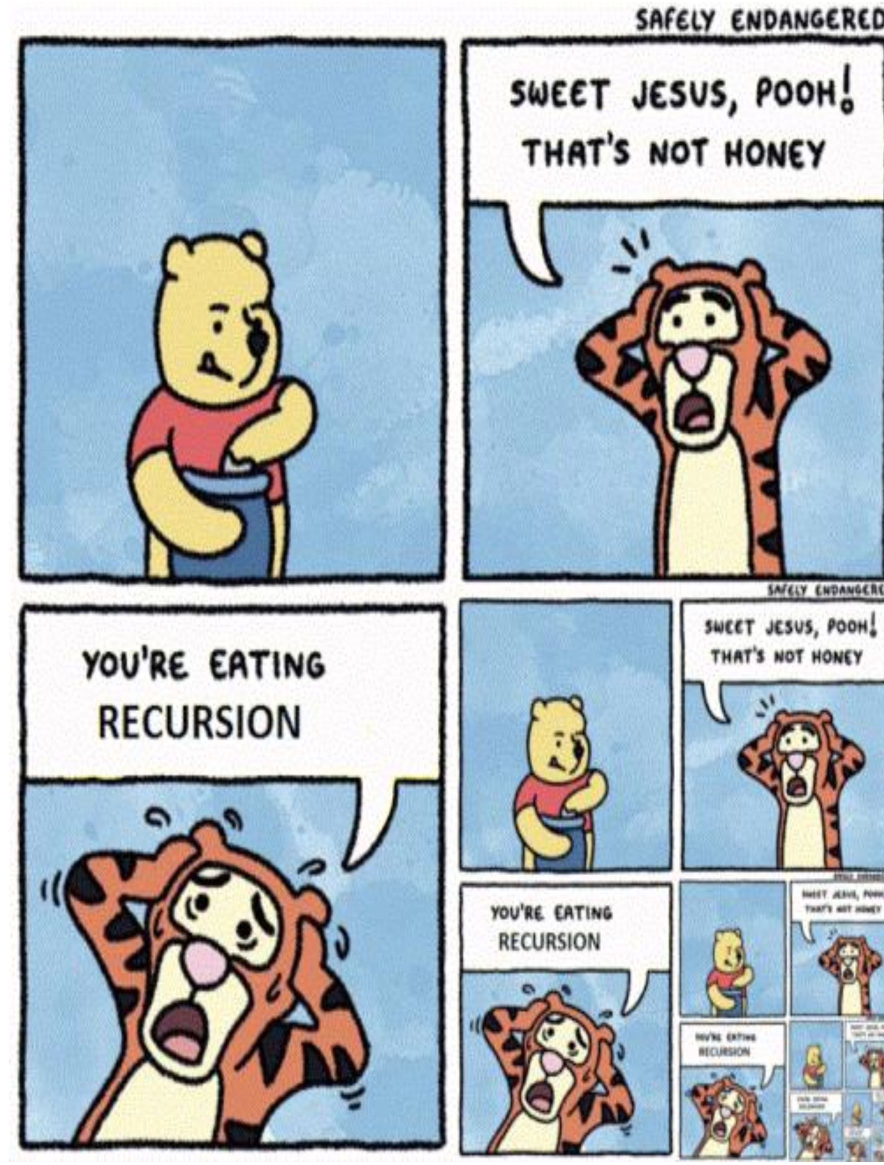
Objectives

In this chapter, you will:

- Understand about Recursion.
- Learn about Recursive Definitions, Sequences and Algorithms
- Learn about Components of a Recursive Implementation
- Be able to decompose a recursive problem into Recursive Steps and Base Cases
- Know the Types of Recursion
- Examine Recursive Function in C ++ Program.
- Understand the Advantages and Disadvantages of Recursion vs. Iteration

Recursive Examples





Recursive Examples

Recursion

- When faced with a difficult problem, a classic technique is to break it down into smaller parts that can be solved more easily. Recursion is one way to do this.
- In some cases, recursion is the best way to solve the problem.
  - Examples: Define a tree data structure and traversal the tree (define Folder and browse files in a folder), ..
- Recursion is a useful tool, but it can increase memory usage and make debugging difficult.

Recursion definitions

- **Define an entity in terms of itself**. There are two parts:
  - **Base case** (basis): the most primitive case(s) of the entity are defined without reference to the entity.
  - **Recursive case** (inductive) : new cases of the entity are defined in terms of simpler cases of the entity.
- Examples:
  - A recursive definition of exponentiation ($\mathbf{a^n}$):
    - $\mathbf{a^0 = 1}$ *// base case*
    - $\mathbf{a^n = (a^{n-1})*a}$, for $n > 0$ *// recursive cases (steps)*
  - Recursively defined data structures:

```
struct Node {
        ... ...
        Node *Next;
};
```

Recursion sequences

- A sequence is an **ordered list** of objects, which is potentially infinite. S(K) denotes Kth object in sequence.
- A sequence is defined recursively by explicitly naming the first value *(or the first few values)* and then define later values in the sequence in terms of earlier values.
- Examples:
  - A recursively defined sequence:
    - S(0) = 1 *// base case*
    - S(n+1) = (n+1) * S(n), for n >= 0 *// recursive case*

  what does the sequence look like?
  - The Fibonacci Sequence is defined by:
    - F(1) = 1, F(2) = 1 *// base case*
    - F(n) = F(n-2) + F(n-1), for n > 2 *// recursive case*

Recursively Defined Algorithms

If a recurrence relation exists for an operation, then the algorithm can be written either iteratively or recursively.

recursive_algorithm(**input**) **{**

if (isSmallEnough (**input**)) *// base-case*

compute the solution and return it

else *// recursive case*

break input into simpler instances input1, input 2,...

solution1 = recursive_algorithm (**input1**)

solution2 = recursive_algorithm (**input2**)

...

figure out solution to this problem from ***solution1, solution2, ..***

return **solution**

}

Components of a Recursive Implementation

- A recursive implementation always has 2 parts:
  - **Base case:** the simplest, smallest instance of the problem, that can't be decomposed any further. Base cases often correspond to emptiness: empty string, empty list, empty set, empty tree, zero,..
  - **Recursive step: decomposes** a larger instance into one or more simpler or smaller instances that can be solved by recursive calls (*to this same function*, *but with the inputs somehow reduced in size or complexity*, *closer to a base case*), and then **recombines** the results of those subproblems to produce the solution to the original problem.

  [*may be more than one base case or more than one recursive step*]

- Helper function: it is a way to reduce complexity within other functions. In some cases, it's useful to require a stronger specification for the recursive steps, to make the recursive decomposition simpler or more elegant

Recursively Defined Algorithms - Example

Factorial $S(n) = n! = 1*2*3*...*(n-1)*n$ can be defined recursively as follows:

- $S(0) = 1$,
- $S(n) = n*S(n-1)$ *for n >= 1*

iteratively	recursively
`long factorial(int n) {` `  long f = 1;` `  for (int i=2; i<n; ++i)` `     f *= i;` `  return f;` `}`	`long factorial(int n) {` `  if (n == 0) // base case` `       return 1;` ` //else   // recursive case` `       return n* factorial(n-1);` `}`

How the Stack grows in Recursive Function

void main() { long x = **factorial** (**3**); }

starts in main	calls factorial(3)	calls factorial(2)	calls factorial(1)	calls factorial(0)	returns to factorial(1)	returns to factorial(2)	returns to factorial(3)	returns to main
				factorial n = 0 **returns 1**				
			factorial n = 1	factorial n = 1	factorial n = 1 **returns 1**			
		factorial n = 2	factorial n = 2	factorial n = 2	factorial n = 2	factorial n = 2 **returns 2**		
	factorial n = 3	factorial n = 3	factorial n = 3	factorial n = 3	factorial n = 3	factorial n = 3	factorial n = 3 **returns 6**	
main	main x	main x	main x	main x	main x	main x	main x	main x = 6

Example - Fibonaci

The Fibonacci Sequence is defined by:

- F(1) = 1, F(2) = 1 *// base case*
- F(n) = F(n-2) + F(n-1), for n > 2 *// recursive case*

Recursive Algorithm	Fibonaci *[Multiple Recursion]*
<pre>recursive_function(input) { if (isSmallEnough (input)) // base-case compute the solution and return it else // recursive case break input into simpler instances input1, input 2,... solution1 = recursive_function (input1) solution2 = recursive_function (input2) ... figure out solution to this problem from solution1, 2,.. return solution }</pre>	<pre>long fibonaci(int n) { if (n == 0 || n == 1) // base case return 1; //else // recursive case long f_1 = fibonaci(n-1); long f_2 = fibonaci(n-2); return f_1 + f_2; }</pre>

Example - Sum

$$S(n) = 1/2 + 2/3 + 3/4 + \ldots + n/(n+1)$$

- $S(0) = 0$ *// base case*
- $S(n) = S(n-1) + n/(n+1)$, *for n > 0* *// recursive case*

Recursive Algorithm	Sum *[Single /Tail Recursion]*
recursive_function(input) { if (isSmallEnough (input)) *// base-case* *compute the solution and return it* else *// recursive case* *break input into simpler instances input1, input 2,...* solution1 = recursive_function (input1) solution2 = recursive_function (input2) ... *figure out solution to this problem from solution1, 2,..* return solution }	long sum(int n) { if (n == 0) *// base case* return 0; //else *// recursive case* long S_1 = sum (n-1); return S_1 + n/(n+1); }

Example – Odd | Even

- **Even**: (n = 0) or (n-1 is Odd)
- **Odd**: (n==1) or (n-1 is Even)

[Indirect Recursion]	[Đệ quy Hỗ tương]
<pre>bool isEven (unsigned n) { if (n == 0) // base case return true; //else // recursive case return isOdd (n-1); }</pre>	<pre>bool isOdd (unsigned n) { if (n == 1) // base case return true; //else // recursive case return isEven (n-1); }</pre>

Example – Square root

S(n)=sqrt(2021+sqrt(2021+....+sqrt(2021+sqrt(2021))))

// n dấu cộng

- S(0) = sqrt (2021) *// base case*
- S(n) = sqrt (2021 + S(n-1)) *// recursive case*

Recursive Algorithm	Square Root *[Nested Recursion / Đệ quy lồng]*
recursive_function(input) { **if (isSmallEnough (input))** *// base-case* ***compute the solution and return it*** **else** *// recursive case* ***break input into simpler instances input1, input 2,...*** **solution1 = recursive_function (input1)** **solution2 = recursive_function (input2)** **...** *figure out solution to this problem from solution1, 2,..* **return solution** }	double **SquareRoot** (int **n**) { if (**n** == 0) *// base case* return **sqrt(2021);** //else *// recursive case* return **sqrt(2021+ SquareRoot(n-1));** }

Example – Print Array


```cpp
#include <iostream>
#include <fstream>
#define FILENAME  "C:\\temp\\Example.bin"
#define MAX 100
using namespace std;

/* write N items to File */
bool WriteFile(const char * FileName, float * Arr, int N)
{
  ofstream fout ( FileName, ios::binary);  // open file for output
  if (fout.fail() ) return false;
  fout.write((char*)&N, sizeof(int)); // write number of elements
  fout.write((char*)Arr, N*sizeof(float)); // write array to file
  fout.close();
  return true;
}

/* read Array from File */
bool ReadFile(const char * FileName, float * &Arr, int &N)
{
  ifstream fin (FileName, ios::binary);  // open file for input
  if (fin.fail() ) return false;
  fin.read((char*)&N, sizeof(int)); // read number of elements
  Arr = new float [N];
  fin.read((char*)Arr, N*sizeof(float)); // read array from file
  fin.close();
  return true;
}
```

```cpp
void PrintArr0( float *Arr, int Num ) { //recursive function
   static int i; // using static variable
    if (i == Num) { // base case
        i = 0;
        cout << endl;
        return;
    }
    cout << Arr[i] << " ";
    i++;
    PrintArr0 ( Arr, Num );
}
void PrintArr1( float *Arr, int Num ) { //recursive function
    if (Num==0) { // base case
        cout << endl;
        return;
    }
    cout << Arr[Num-1] << " ";
    PrintArr1 ( Arr, Num-1 );
}

int main() {
    float a[MAX] = { 1.1, 30.4, 1.5, 2.9, 20.11, 24.12 };
    if (!WriteFile(FILENAME, a, 5)) return -1;
    int n; float *b;
    if (!ReadFile(FILENAME, b, n) ) return -2;
    PrintArr1 ( b, n ); PrintArr0 ( b, n );
    return 0;
}
```

Types of Recursion

- **Single** and **Multiple Recursion**, when while recursion that contains single or multiple self-references
  - **Single Recursion (Linear Recursion)** contains only a single self-reference, includes **Tail Recursion** and **Head Recursion** (If a recursive function calling itself and that recursive call is the last /first statement in the function).
  - **Multiple Recursion (Tree Recursion)** contains multiple self-reference, function calling itself for more than one time.
- **Direct** and **Indirect Recursion**.
  - **Direct Recursion:** function calls itself.
  - **Indirect Recursion:** occurs when a function is called not by itself but by another function that it called
- **Nested Recursion**: a recursive function will pass the parameter as a recursive call. That means "*recursion inside recursion*"