CS1010E Lecture 7

Functional Abstraction and Recursion

Henry Chia (hchia@comp.nus.edu.sg)

Semester 1 2016 / 2017

Lecture Outline

- Functional abstraction
  - Structured programming and top-down design
  - Modular-design principles
- Recursion
  - Recurrence relations and recursive functions
  - Recursive invocation/call
  - General recursive problems

Designing Solutions for "Large" Problems

- Break down large problem into smaller/simpler sub-problems
- Define functions/procedures to handle each sub-problem
- Functional abstraction
  - Know **what** a function does, not **how** it is done
- Structured programming
  - Programs having a logical structure that makes them easy to read, understand and modify
  - Adopt a **top-down** modular design methodology:
    - Start with the "large" task and break into sub-tasks
    - Break these sub-tasks into smaller sub-tasks until the smallest ones have trivial solutions
    - Compose all sub-tasks to solve the original problem

Example: Calculate Taxi Fare (Revisited)

- Calculate the taxi fare based on the following fare structure:

Meter Fare	Normal	Limousine	Chrysler
Flag-Down (inclusive of 1st km or less)	\$3.40	\$3.90	\$5.00
Every 400m thereafter or less up to 10km	\$0.22	\$0.22	\$0.33
Every 350 metres thereafter or less after 10 km	\$0.22	\$0.22	\$0.33

- Input:
  - Type of taxi: Normal(1), Limousine(2), Chrysler(3)
  - Distance travelled: non-negative integer
- Sample Run:

```
1 12500
$10.22
```

- Peak-hour surcharge is ignored; you can extend that later

Top-level program

```c
#include <stdio.h>

int computeFare(int type, int dist);
void printFare(int fare);

int main(void) {
   int type, dist, fare;

   scanf("%d%d", &type, &dist);
   fare = computeFare(type, dist);
   printFare(fare);
   return 0;
}

int computeFare(int type, int dist) {
   return 1022;
}

void printFare(int fare) {
   printf("$%d.%d0\n", fare / 100, (fare % 100) / 10);
   return;
}
```

- Since `computeFare` might require further decomposition, replace with a "stub", so that program can still be tested

Breaking down `computeFare`

```c
int computeFare(int type, int dist) {
   int fare;

   if (dist == 0) {
      fare = 0;
   } else if (dist <= 1000) {
      fare = stage1Fare(type);
   } else if (dist <= 10000) {
      fare = stage1Fare(type) + stage2Fare(type, dist-1000);
   } else {
      fare = stage1Fare(type) + stage2Fare(type, 9000) +
             stage3Fare(type, dist-10000);
   }
   return fare;
}
```

- Breakdown `computeFare` into fare stages via stubs

```c
int stage1Fare(int type) {
   printf("stage1\n");
   return 0;
}
```

```c
int stage2Fare(int type,
               int dist) {
   printf("stage2\n");
   return 0;
}
```

```c
int stage3Fare(int type,
               int dist) {
   printf("stage3\n");
   return 0;
}
```


Calculating Fares for Different Stages

```c
int stage1Fare(int type) {
   if (type == NORMAL)
      return 340;
   else if (type == LIMOUSINE)
      return 390;
   else if (type == CHRYSLER)
      return 500;
   else
      return 0;
}
```

```c
int stage2Fare(int type, int dist) {
   int rate;
   rate = getRate(type);
   return computeStage(dist, rate, 400);
}

int stage3Fare(int type, int dist) {
   int rate;
   rate = getRate(type);
   return computeStage(dist, rate, 350);
}
```

```c
int getRate(int type) {
   if (type == NORMAL)
      return 22;
   else if (type == LIMOUSINE)
      return 22;
   else if (type == CHRYSLER)
      return 33;
   else
      return 0;
}

int computeStage(int dist, int rate, int block) {
   return ( ( (dist - 1) / block ) + 1 ) * rate;
}
```


Principles in Modular Design

- Abstraction
  - Know *what* it does but don't care *how* it is done
- Reusability
  - Identify modules that can be used repeatedly
  - One function definition; many calls to the same function
- High cohesion
  - Do only one thing, and do it well
- Loose coupling
  - Minimize the use of "output parameters" that allows one module to directly change properties (variables) of another module

Recursion

- Recursion can be a powerful tool for solving certain classes of problems in which the solution can be defined in terms of a similar but smaller problem
  - Then this smaller problem is defined in terms of a similar but still smaller problem
    - Then this smaller problem is defined in terms of a similar but still smaller problem
      - Then this smaller problem is defined in terms of a similar but still smaller problem
- Redefinition of the problem into smaller problems continues until the "smallest" problem has a unique solution that is then used to determine the overall solution

Motivating Example: Factorial

- $n!$ (read as n factorial) is defined as:

$$n! = (n)(n-1)(n-2)\cdots(3)(2)(1) \text{ with } n \geq 0,\ 0! = 1$$

- Or as a recurrence relation:

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

- Factorial defined in terms of a product that involves smaller and smaller factorials

Winding and Unwinding

- **Windup** phase:

  factorials are continually re-defined until $0!$

$$\begin{aligned} 5! &= \underline{5 \cdot 4!} \\ &= 5 \cdot \underline{4 \cdot 3!} \\ &= 5 \cdot 4 \cdot \underline{3 \cdot 2!} \\ &= 5 \cdot 4 \cdot 3 \cdot \underline{2 \cdot 1!} \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot \underline{1 \cdot 0!} \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot \underline{0!} \end{aligned}$$

- **Unwind** phase:

  substitute $0! = 1$ and back-track while substituting values for the factorials

$$\begin{aligned} & 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot \underline{1} \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot \underline{1 \cdot (1)} \\ &= 5 \cdot 4 \cdot 3 \cdot \underline{2 \cdot (1)} \\ &= 5 \cdot 4 \cdot \underline{3 \cdot (2)} \\ &= 5 \cdot \underline{4 \cdot (6)} \\ &= \underline{5 \cdot (24)} \\ &= (120) \end{aligned}$$

Factorial Computation

```c
#include <stdio.h>

int fact(int n);

int main(void) {
   int n;

   printf("Enter n: ");
   scanf("%d", &n);
   printf("The factorial of %d is %d\n", n, fact(n));

   return 0;
}
```

- Iterative

```c
int fact(int n) {
   int fac = 1;

   while (n > 1) {
      fac = fac * n;
      n--;
   }
   return fac;
}
```

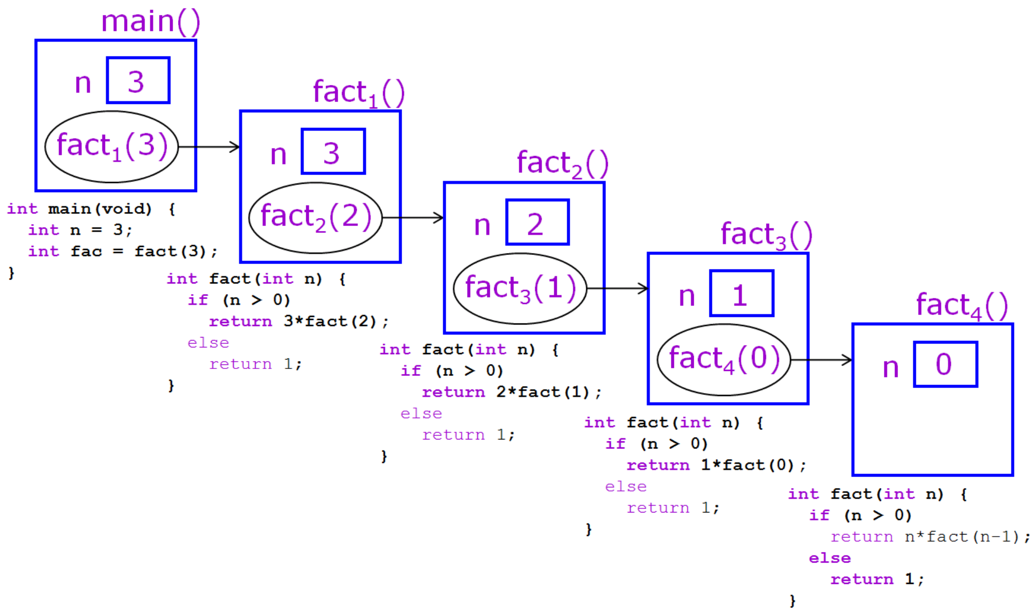
- Recursive

```c
int fact(int n) {
   if (n > 0) {
      return n * fact(n - 1);
   } else {
      return 1;
   }
}
```

Recursive Function

- A function that "calls itself" is a **recursive function**
  - **Recursive case**: allows the function to call "itself" recursively with an argument that gets "smaller" so as to **eventually satisfy the base case**
  - **Base case**: keeps the function from recursing infinitely
- A selection control flow construct (typically `if..else`) is used to decide whether to execute the base or recursive cases
- In place of multiple loops in the iterative computation, the recursive computation makes use of multiple function calls
- As compared with the iterative version, "how it works" is less apparent in the recursive one; in fact, one needs only appreciate "what it does" — **functional abstraction**

Recursive Calls



Fibonacci Sequence

- The **Fibonacci sequence** is a sequence of numbers $(f_0, f_1, f_2, \dots)$ in which $f_0 = 0$ and $f_1 = 1$ with each successive number being the sum of the previous two

$$f_k = \begin{cases} k & \text{if } k \in \{0, 1\} \\ f_{k-1} + f_{k-2} & \text{if } k > 1 \end{cases}$$

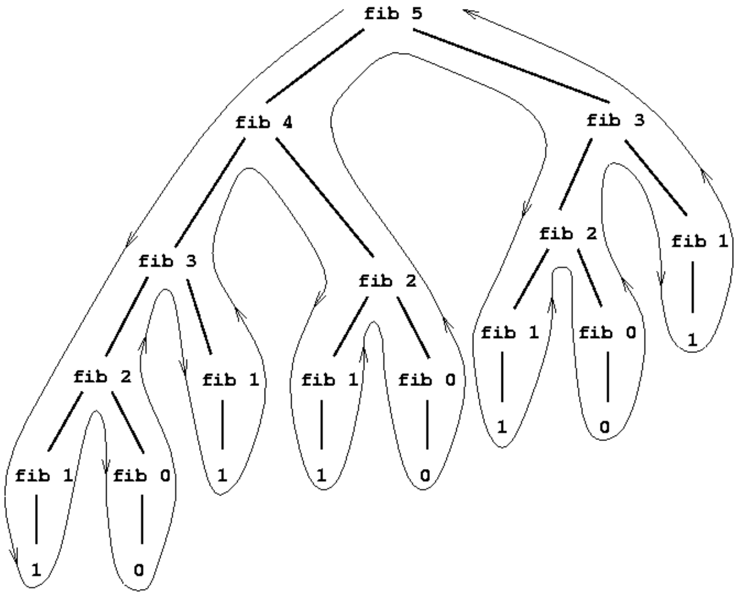
- The first few values of the Fibonacci sequence are:

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad \dots$$

Fibonacci Sequence

```
#include <stdio.h>

int fib(int k);

int main(void) {
   int k;

   printf("Enter k: ");
   scanf("%d", &k);
   printf("F(%d) = %d\n", k, fib(k));
   return 0;
}

int fib(int k)  {
   if (k <= 1)
      return k;
   else
      return fib(k - 1) + fib(k - 2);
}
```

Recursive Tree



Conditional Operator

- C allows a **conditional operator** to be used in place of a simple `if/else` statement
- Conditional operator has three arguments—a condition, an expression to perform if the condition is true, and an expression to perform if the condition is false
- The operation is indicated with a question mark following the condition, and with a colon between the two expressions
- Example: `c = (a < b) ? (b - a) : (b + a);` is equivalent to

```
if (a < b) {
   c = b - a;
} else {
   c = b + a;
}
```


Conditional Operator

- The recursive factorial and Fibonnaci functions can be simplified:

```
int fact(int n) {
   return (n > 0) ? n * fact(n - 1) : 1;
}

int fib(int k)  {
   return (k <= 1) ? k : fib(k - 1) + fib(k - 2);
}
```

- Trace the execution of `fib(5)`

Problem Solving: GCD

- Greatest common divisor (GCD) of two non-negative integers (not both zero)
- Recurrence relation:

$$gcd(m,n) = \begin{cases} gcd(n, m \bmod n) & \text{if } n > 0 \\ m & \text{if } n = 0 \end{cases}$$

- Write the recursive function `int gcd(int m, int n);`
- Trace the function invocations

General Recursive Problems

- Problems need not always be specified with a given recurrence relation or recursive formulation
- Example, pattern printing $(n = 5)$
  - From n to 1 stars
  - From 1 to n stars

```
*****          *
****           **
***            ***
**             ****
*              *****
```


General Recursive Problems

```
void printRowOfStars(int n) {
   if (n == 0) {
      printf("\n");
   } else {
      printf("*");
      printRowOfStars(n - 1);
   }
   return;
}

void printPattern(int n) {
   if (n > 0) {
      printRowOfStars(n);
      printPattern(n - 1);
   }
   return;
}
```

- Note that recursive calls invoked during the wind-up phase

Recursion During Unwinding

- Recursive calls can be invoked during the unwind phase

```
void printPattern(int n) {
   if (n > 0) {
      printPattern(n - 1);
      printRowOfStars(n);
   }
   return;
}
```

- Making recursive calls during both windup and uuwind

```
void printPattern(int n) {
   if (n > 1) {
      printRowOfStars(n);
      printPattern(n - 1);
      printRowOfStars(n);
   } else {
      printRowOfStars(n);
   }
   return;
}
```


Lecture Summary

- Functional abstraction and top-down design
  - Identify that a large problem can be broken down into smaller and simpler sub-problems
  - Apply modular design principles to develop each module incrementally
- Recursion
  - Identify a problem can be broken down into a similar but smaller problems with the smallest one trivially solved
  - Identify the base and recursive case(s)
  - Appreciate recursion as a cascade of functions calls with separate activations of the same function implementation
  - *Don't try to think recursively about a recursive process*