CS1010E: Programming Methodology

Tutorial 05: Function

27 Feb 2017 - 04 Mar 2017

1. Discussion Questions

(a) [Simple Function Reasoning] What is/are the output of code fragments below?

i.
```
void foo(int x) {
  x = x + 1; printf("%d ", x); return;
}
int main() {
  int x = 1; printf("%d ", x);
  foo(x);    printf("%d ", x);
  return 0;
}
```
i. _____

ii.
```
void foo(int a, int b) { printf("%d ", a-b); return }
int main() {
  int a = 1, b = 2; printf("%d ", a-b);
  foo(b, a);        printf("%d ", a-b);
  return 0;
}
```
ii. _____

iii.
```
int next(int n) { return n+2; }
int main() {
  int x = 5; printf("%d", next(x));
}
```
iii. _____

iv.
```
int n = 1010;
int next() { return n+2; }
int main() {
  printf("%d ", next());
  printf("%d ", next());
  printf("%d ", next());
  return 0;
}
```
iv. _____

v.
```
int next() {
  static int n = 0; return ++n;
}
int main)() {
  printf("%d ", next());
  printf("%d ", next());
  printf("%d ", next());
  return 0;
}
```

v. _____

(b) [Random Number] What is/are the possible values generated by the code fragments on random number (**rand**) below? Use the following notation:

- Lower bound *inclusive*: `[`
- Lower bound *exclusive*: `(`
- Upper bound *inclusive*: `]`
- Upper bound *exclusive*: `)`

Exclusive means that the value will **never** be equal to the stated value but close enough. For instance, the range `(0.00, 11.00]` means that the lower bound value will never be `0.00` but close to it and the upper bound value is at most exactly `11.00`.

i. `printf("%d", rand()%10);`

i. _____

ii. `printf("%d", rand()%56+25);`

ii. _____

iii. `printf("%.2f", rand()/(RAND_MAX+1)*9.0);`

iii. `[0.00, 0.00]`

iv. `printf("%.2f", rand()*9.0/RAND_MAX);`

iv. _____

v. `printf("%.2f", rand()*9.0/RAND_MAX+1);`

v. _____

2. Program Analysis

(a) [Complex Function Reasoning] What is/are the output of code fragments below?

i.
```c
double foo(int n) {
  return n/2;
}
int main() {
  printf("%.1f", foo(3.5));
  return 0;
}
```

i. 1.0 (type coercion)

ii.
```c
int c = 0;
void find_min(int a, int b, int c) {
  c = (a < b) ? a : b;
}
int main() {
  int a = 5, b = 9;
  find_min(a, b, c);
  printf("%d", c);
  return 0;
}
```

ii. 0 (scope override)

iii.
```c
int foo(int n) {
  return n/2;
}
int bar(int n) {
  return foo(n*3+1);
}
int main() {
  printf("%d ", bar(3));
  printf("%d ", bar(bar(3)));
  return 0;
}
```

iii. 5 8 (nested function)

iv.
```c
#define TWICE(x) x+x
int twice(int x) { return x+x; }
int main() {
  printf("%d ", TWICE(5)*TWICE(5));
  printf("%d ", twice(5)*twice(5));
  return 0;
}
```

iv. 35 100 (order of operations)

(b) [Random Number Reasoning] What is/are the output of code fragments below?

i.
```c
int n = 100, m = rand(), ans = 0;
while(n*m%2 == 0) {
  n -= 2;
  m  = rand();
  if(n == 0) break;
  ans++;
} printf("%d", ans);
```

i. 49 (n is always even)

ii.
```
int n = rand()*9+81, m = 0;
while(n > 0) {
  m += n%10;
  n /= 10;
} printf("%d", m%3);
```

ii. 0 (n is multiple of 3)

iii.
```
double n = rand()/(RAND_MAX+1)*9.0;
int ans = 0;
while(n > 0) {
  ans += rand();
  n   -= 1.0;
} printf("%d", ans);
```

iii. 0 (integer division)

3. Designing a Solution

(a) [Modularity; Mathematics] Prime triples are **three (3)** consecutive primes such that the difference between first of the triples and the last of the triples differs by exactly **six (6)**. For instance, the triples $(5, 7, 11)$, $(7, 11, 13)$, and $(11, 13, 17)$ are all prime triples. Given two numbers `lower` and `upper` finds all prime triples within the range [`lower`, `upper`].

Write your program below (note: *the function prototypes are meant to help you, utilize them properly*):

```
bool isPrime(int n) {
  /* Check if n is a prime number or not */
  int i;                // a somewhat more optimal algorithm is shown below:
  for(i=2; i<n; i++) { // if(n%2 == 0) return false;
    if(n%i == 0)       // for(i=3; i*i<=n; i+=2)
      return false;    //   if(n%i == 0)
  }                    //     return false;
  return true;
}
void printTriple(int n) {
  /* Find prime triples starting from n and print if exist */
  int i;
  if(isPrime(n) && isPrime(n+6))
    for(i=n+1; i<n+6; i++)
      if(isPrime(i))
        printf("(%d,%d,%d)", n, i, n+6);
  /* Simpler Alternative: with knowledge of prime number theory */
  //if(isPrime(n) && isPrime(n+2) && isPrime(n+4)) printf("(%d,%d,%d)",n,n+2,n+4);
  //if(isPrime(n) && isPrime(n+2) && isPrime(n+6)) printf("(%d,%d,%d)",n,n+2,n+6);
  //if(isPrime(n) && isPrime(n+4) && isPrime(n+6)) printf("(%d,%d,%d)",n,n+4,n+6);
}
int main() {
  int lower, upper, i; scanf("%d %d", &lower, &upper);
  /* Your Solution Here */
  for(i=lower; i<=upper-6; i++) {
    printTriple(i);
  }
```

```
  return 0;
}
```

4. Challenge

(a) [Simulation; Random Number] Monty Hall problem is a probability puzzle based on American television game "*Let's Make a Deal*". The game setup consists of **three (3)** doors (named Door 0, Door 1, and Door 2). Behind two of the doors are goats and behind the last one is the prize. The objective is to guess which of the door contains the prize. The steps in the game is described below:

1. Player pick a door.
2. Host opens a door that is not picked by the player and contains a goat.
3. Player is given a *choice* to change the door before Player's final option is opened.

The question is, is it better to stay with your original selection or switch your selection. To answer that, we will simulate the **two (2)** tactics:

(A) Never change door
(B) Always change door

i. Write a code for Tactic (A). Use the following template:

```
int never(int chosen, int opened) {
  /* chosen: initial player choice, opened: opened by host */
  return chosen;
}
```

ii. Write a code for Tactic (B). Use the following template:

```
int always(int chosen, int opened) {
  /* chosen: initial player choice, opened: opened by host */
  if(chosen == 0 && opened == 1) { return 2; }
  if(chosen == 0 && opened == 2) { return 1; }
  if(chosen == 1 && opened == 0) { return 2; }
  if(chosen == 1 && opened == 2) { return 0; }
  if(chosen == 2 && opened == 0) { return 1; }
  if(chosen == 2 && opened == 1) { return 0; }


}
```

iii. Write a code to select tactics based on user input: 1) Tactic (A) and 2) Tactic (B). Use the following template:

```
int tactic(int C, int O, int T) {
  /* C: initial player choice, O: opened by host, T: user tactic */
  return (T==1) ? never(C, O) : always(C, O);


}
```

iv. Simulate the three basic tactics for 10000 runs with uniform probability of players choosing any door and uniform probability of prize being in any door. Which is the best tactic?

```
/* Select the door to be opened: NOT chosen AND NOT prize (always tactic) */
int open(int chosen, int prize) {
```

```
  if(chosen == prize) return (chosen+(rand()%2)+1)%3;
  else                return always(chosen, prize);   // Same algo as always
}
#define M 10000
int main() {
  int T, n, winMax = -1, winCurr, prize, chosen, opened, best;
  srand(time(NULL));
  for(T=1; T<=3; T++) {                    // Enumerate tactics
    winCurr = 0;
    for(n=0; n<M; n++) {
      prize  = rand()%3;                   // Initialization
      chosen = rand()%3;                   // Step 1
      opened = open(chosen, prize);        // Step 2
      chosen = tactic(chosen, opened, T); // Step 3
      if(chosen == prize) winCurr++;
    }
    printf("Tactic: %d, Win probability: %.2f%%\n", T, winCurr*100.0/M);
    if(winMax < winCurr) {
       winMax = winCurr;
       best   = T;
    }
  }
  printf("Best tactic is: %d, Win probability: %.2f%%", best, winMax*100.0/M);
  // Best Tactic is to "Always Change" with probability 2/3 of winning
}
```

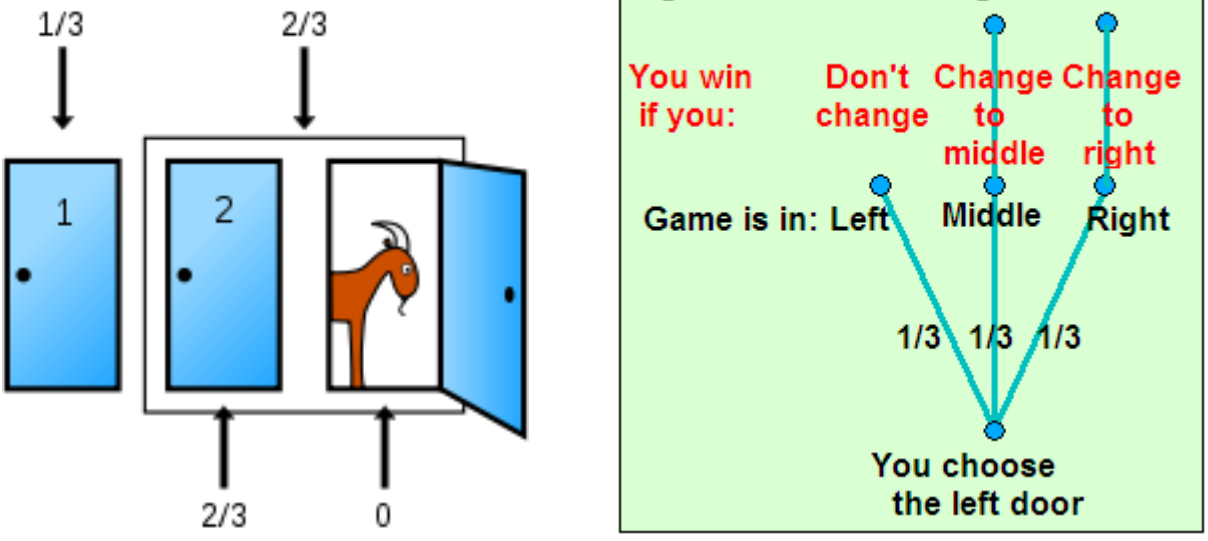

Diagram 1: Explanation of Monty Hall problem in probability theory.