Programming Design, Spring 2014
Suggested Solution for Homework 06

Solution provider: 謝佳吟

Problem 1:

(a) Four local variables are created in the memory before the first statement of the function is executed. `diceCount`, `trialCount` and `seed` are `int` variables using call by value and `avg[]` is an array pointer using call by pointer to store double values.

(b) The calculation of double variables may easily introduce errors. Besides, among all the calculations, variable `sum` should be a double type variable only when doing the averaging. Therefore, an explicit casting way uses less memory.

(c)

```
int main()
{
    double avg[1000]={0};
    diceAvg(5, 1000, avg, 0);
    int count[10]={0};
    int index=0;

    for(int i=0; i<1000; i++)
    {
        index=static_cast<int>((avg[i] - 1.0)/ 0.5);
        count[index]++;
    }

    for(int i=0; i<10; i++)
    {
        cout << count[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Result:

```
4 11 90 117 270 204 216 61 24 3
```

(d) The reason `avg` must be a pointer is that before getting the input `trialCount`, we do not know the size of memory we should assign to `avg`. What is missing is that we do not release the space dynamically allocated to `avg`.

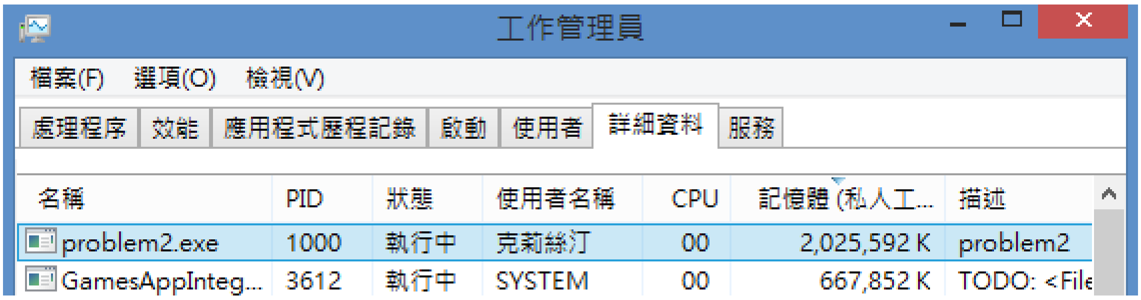
(e) With the increase of `diceCount`, the frequency distributions become more centralized as shown in the following picture.

IntervalsUpperBounds (start from 0.0)	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0	5.5
diceCount= 1	158	0	156	0	161	0	175	0	165	0
diceCount= 3	13	31	123	86	253	125	204	66	79	14
diceCount= 5	4	11	90	117	270	204	216	61	24	3
diceCount= 7	2	4	59	150	296	223	208	44	14	0
diceCount= 9	0	1	45	129	323	279	188	32	3	0
diceCount=11	0	1	24	122	339	327	162	23	2	0
diceCount=13	0	0	19	115	352	341	155	18	0	0
diceCount=15	0	0	13	97	389	343	149	9	0	0
diceCount=17	0	0	8	90	413	353	130	6	0	0
diceCount=19	0	0	5	88	414	372	112	9	0	0
diceCount=21	0	0	1	79	426	381	109	4	0	0
diceCount=23	0	0	4	74	419	415	85	3	0	0
diceCount=25	0	0	1	61	444	420	72	2	0	0
diceCount=27	0	0	1	68	415	452	64	0	0	0
diceCount=29	0	0	0	44	446	453	56	1	0	0
diceCount=31	0	0	0	43	466	432	58	1	0	0
diceCount=33	0	0	1	44	454	451	50	0	0	0
diceCount=35	0	0	0	32	477	456	35	0	0	0
diceCount=37	0	0	1	43	456	458	42	0	0	0
diceCount=39	0	0	0	32	476	456	36	0	0	0

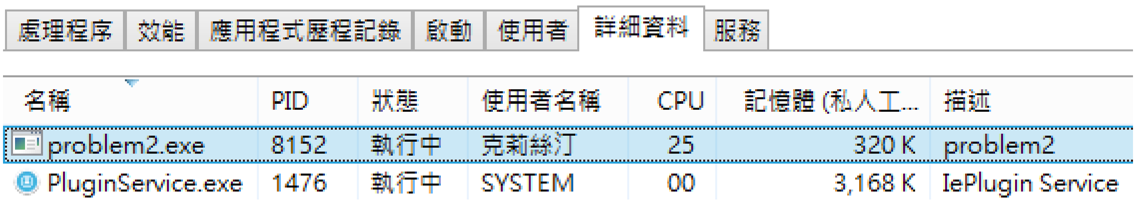
Problem 2:

(a) 200000000*8 bytes (nearly 1.5GB) are wasted. Each round in the for loop created an 8 bytes space for a double variable which is pointed by pointer d. Then, the program set pointer d to a new created 8 bytes space in the next round without deleting the former created space, which cause the problem called "memory leak".

(b)



(c) If the memory is released by deleting it before running the next round in the for loop, only few memory will be occupied when executing the program.



Problem 3:

(a) Both implementations are intuitive. However, if the initial process will only be used by the member array `m[]`, it is better to choose to implement a member-function in order to make the program clearer. Otherwise, you may choose to implement a global-function which can be reused to initialize all the arrays in the program.

(b) a.n is create to store the size of array m of a. If a.n is modified after initializing a.m, it might cause error in the latter part of the program once using a.n to represent the size of a.m, such as index of a.m is out of range.

(c) a.m should be deleted before the second initializing action, if not (such as the problem's statement), it will cause memory leak.

(d) If the program ends without releasing the memory, it will cause memory leak.

(e)

```
bool MyVector::noNegative(){
    for(int i=0; i<dimention; i++){
        if(a.m[i]<0)
            return false;
    }
    return true;
}
```

Problem 4:

See the cpp files: **z1.cpp** and **z2.cpp**

- **z1** is the makespan of the following naive schedule:

  Assigning job i to machine mod$(i-1, m)$+1, where mod(a, b) is the remainder of a divided by b.

- **z2** is the makespan of the following better designed schedule:

  First, sort the jobs by its loading in descending order, and then assign each job i to machine which has the minimum loading at that moment.