Course: ENSF 337 – Fall 2020

Lab #: Lab 7

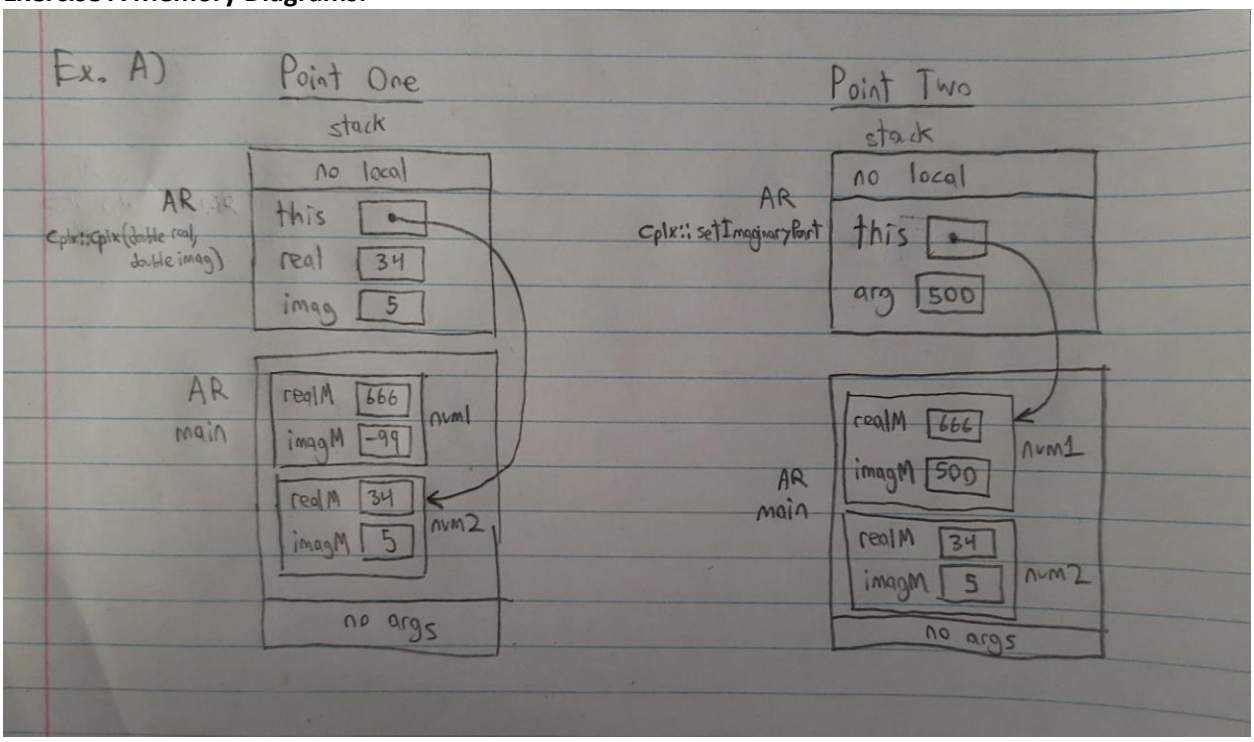
Instructor: M. Moussavi

Student Name: Quentin Jennings

Lab Section: B03

Submission Date: 2020-11-15

Exercise A Memory Diagrams:



Exercise C lab7Clock.h:

```
1   // File lab7Clock.h
2   // ENSF 337  LAB 7 - EXERCISE C
3   // By: Quentin Jennings
4
5   #ifndef LAB7CLOCK
6   #define LAB7CLOCK
7
8   class Clock{
9   private:
10      int hour, minute, second;
11      int hms_to_sec();
12      void sec_to_hms(int s);
13
14
15  public:
16      Clock();
17      Clock(int s);
18      Clock(int h, int m, int s);
19
20      int get_hour()const;
21      int get_minute()const;
22      int get_second()const;
23
24      void set_hour(int h);
25      void set_minute(int m);
26      void set_second(int s);
27
28      void increment();
29      void decrement();
30      void add_seconds(int s);
31  };
32
33  #endif
```

Exercise C lab7Clock.cpp:

```cpp
// File lab7Clock.cpp
// ENSF 337  LAB 7 - EXERCISE C
// By: Quentin Jennings

#include "lab7Clock.h"

//constructors
Clock::Clock(): hour(0), minute(0), second(0) {}
Clock::Clock(int s): hour(0), minute(0), second(0)
{
    if(s > 0)
        sec_to_hms(s);
}
Clock::Clock(int h, int m, int s): hour(h), minute(m), second(s)
{
    if(hour < 0 || hour > 23 || minute < 0 || minute > 59 || second < 0 || second > 59)
    {
        hour = 0;
        minute = 0;
        second = 0;
    }
}

//private functions
void Clock::sec_to_hms(int s)
{
    hour = 0;
    minute = 0;
    second = s;

    while(second > 59)
    {
        second -= 60;
        minute++;
    }
    while(minute > 59)
    {
        minute -= 60;
        hour++;
    }
    while(hour > 23)
    {
        hour -= 24;
    }
}
int Clock::hms_to_sec()
{
    return second + 60*minute + 3600*hour;
}

```

```
51  //public functions
52  int Clock::get_hour()const {return hour;}
53  int Clock::get_minute()const {return minute;}
54  int Clock::get_second()const {return second;}
55
56  void Clock::set_hour(int h)
57  {
58      if(h >= 0 && h <= 23)
59          hour = h;
60  }
61  void Clock::set_minute(int m)
62  {
63      if(m >= 0 && m <= 59)
64          minute = m;
65  }
66  void Clock::set_second(int s)
67  {
68      if(s >= 0 && s <= 59)
69          second = s;
70  }
71
72  void Clock::increment()
73  {
74      second++;
75      sec_to_hms(hms_to_sec());
76  }
77  void Clock::decrement()
78  {
79      if(hms_to_sec() == 0)
80      {
81          hour = 23;
82          minute = 59;
83          second = 59;
84      }
85      else
86      {
87          second--;
88          sec_to_hms(hms_to_sec());
89      }
90  }
91  void Clock::add_seconds(int s)
92  {
93      sec_to_hms(hms_to_sec() + s);
94  }
95
```

Exercise C Output:

```
Object t1 is created. Expected time is: 00:00:00
00:00:00
Object t1 incremented by 86400 seconds. Expected time is: 00:00:00
00:00:00
Object t2 is created. Expected time is: 00:00:05
00:00:05
Object t2 decremented by 6 seconds. Expected time is: 23:59:59
23:59:59
After setting t1's hour to 21. Expected time is: 21:00:00
21:00:00
Setting t1's hour to 60 (invalid value). Expected time is: 21:00:00
21:00:00
Setting t2's minute to 20. Expected time is: 23:20:59
23:20:59
Setting t2's second to 50. Expected time is 23:20:50
23:20:50
Adding 2350 seconds to t2. Expected time is: 00:00:00
00:00:00
Adding 72000 seconds to t2. Expected time is: 20:00:00
20:00:00
Adding 216000 seconds to t2. Expected time is: 08:00:00
08:00:00
Object t3 is created. Expected time is: 00:00:00
00:00:00
Adding 1 second to clock t3. Expected time is: 00:00:01
00:00:01
After calling decrement for t3. Expected time is: 00:00:00
00:00:00
After incrementing t3 by 86400 seconds. Expected time is: 00:00:00
00:00:00
After decrementing t3 by 86401 seconds. Expected time is: 23:59:59
23:59:59
After decrementing t3 by 864010 seconds. Expected time is: 23:59:49
23:59:49
t4 is created with invalid value (25 for hour). Expected to show: 00:00:00
00:00:00
t5 is created with invalid value (-8 for minute). Expected to show: 00:00:00
00:00:00
t6 is created with invalid value (61 for second). Expected to show: 00:00:00
00:00:00
t7 is created with invalid value (negative value). Expected to show: 00:00:00
00:00:00
```

Exercise D simpleVector.cpp:

```cpp
#include "simpleVector.h"
#include <cassert>
using namespace std;

SimpleVector::SimpleVector(const TYPE *arr, int n) {
    storageM = new TYPE[n];
    sizeM = n;
    capacityM = n;
    for(int i =0; i < sizeM; i++)
        storageM[i] = arr[i];
}

TYPE& SimpleVector::at(int i) {
    assert(i >= 0 && i < sizeM);
    return storageM[i];
}

const TYPE& SimpleVector::at(int i)const {
    assert(i >= 0 && i < sizeM);
    return storageM[i];
}


// The followng member function should follow the above-mentioned memory
// management policy to resize the vector, if necessary. More specifically:
//   - If sizeM < capacityM it doesn't need to make any changes to the size of
//     allocated memory for vector
//   - Otherwise it follows the above-mentioned memory policy to create additionl
//     memory space and adds the new value, val, to the end of the current vector
//     and increments the value of sizeM by 1
void SimpleVector::push_back(TYPE arg) {
    if(sizeM == 0 && capacityM == 0)
    {
        capacityM = 2;
        delete [] storageM;
        storageM = new TYPE[2];
    }
    else if(sizeM == capacityM && capacityM > 0)
    {
        capacityM *= 2;
        TYPE* tempMem = new TYPE[capacityM];
        for(int i = 0; i < sizeM; i++)
            tempMem[i] = storageM[i];
        delete [] storageM;
        storageM = tempMem;
    }
    storageM[sizeM] = arg;
    sizeM++;
}

SimpleVector::SimpleVector(const SimpleVector& source): sizeM(source.sizeM), capacityM(source.capacityM) {
    if(!source.storageM)
        storageM = source.storageM;
    else
    {
        storageM = new TYPE[capacityM];
        assert(storageM);
        for(int i = 0; i < sizeM; i++)
            storageM[i] = source.storageM[i];
    }

}
```

```
91  SimpleVector& SimpleVector::operator= (const SimpleVector& rhs ){
92      if(this != &rhs)
93      {
94          sizeM = rhs.sizeM;
95          capacityM = rhs.capacityM;
96
97          delete [] storageM;
98          storageM = new  TYPE[capacityM];
99          assert(storageM);
100         for(int i = 0; i < sizeM; i++)
101             storageM[i] = rhs.storageM[i];
102     }
103
104     return *this;
105 }
106
```

Exercise D Output:

```
Object v1 is expected to display: 45 69 12
45 69 12
Object v2 is expected to diaplay: 3000 6000 7000 8000
3000 6000 7000 8000

After two calls to at v1 is expected to display: 1000 2000 12:
1000 2000 12

v2 expected to display: 3000 6000 7000 8000 21 28
3000 6000 7000 8000 21 28

After copy v2 is expected to display: 1000 2000 12
1000 2000 12

v1 is expected to display: 1000 2000 8000
1000 2000 8000

v3 is expected to diplay: 1000 2000 12
1000 2000 12

v2 is expected to display: -333 2000 12
-333 2000 12

v4 is expected to diplay: 1000 2000 8000
1000 2000 8000

v1 after self-copy is expected to diplay: -1000 2000 8000
-1000 2000 8000

v1 after chain-copy is expected to diplay: 1000 2000 12
1000 2000 12

v2 after chain-copy is expected to diplay: 1000 2000 12
1000 2000 12
```

Exercise E Memory Diagrams: