# ECE 2400 Computer Systems Programming

## Topic 13: Object-Oriented Programming

http://www.csl.cornell.edu/courses/ece2400
School of Electrical and Computer Engineering
Cornell University

revision: 2021-11-18-20-42

Please do not ask for solutions. Students should compare their solutions to solutions from their fellow students, discuss their solutions with the instructors during lab/office hours, and/or post their solutions on Ed for discussion.

## List of Problems

<table>
<thead>
<tr>
<th>1</th>
<th>Short Answer</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.A</td>
<td>Constructors and Destructors</td>
<td>3</td>
</tr>
<tr>
<td>1.B</td>
<td>Accessor Member Function</td>
<td>4</td>
</tr>
<tr>
<td>1.C</td>
<td>Clock Class (Weight: ×2)</td>
<td>5</td>
</tr>
<tr>
<td>1.D</td>
<td>Integers in Sign Magnitude Form</td>
<td>6</td>
</tr>
<tr>
<td>1.E</td>
<td>Filling a Sequence with Zeros</td>
<td>7</td>
</tr>
<tr>
<td>1.F</td>
<td>Nested Virtual Function Calls</td>
<td>9</td>
</tr>
<tr>
<td>1.G</td>
<td>Dynamic Polymorphic Array of Objects</td>
<td>10</td>
</tr>
<tr>
<td>1.H</td>
<td>Shape Interface Inheritance</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th>Set Data Structures</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.A</td>
<td>SetLL: Set Implemented with a List of Lists</td>
<td>13</td>
</tr>
<tr>
<td>2.B</td>
<td>SetLL::contains Complexity Analysis</td>
<td>17</td>
</tr>
<tr>
<td>2.C</td>
<td>SetAA: Set Implemented with an Array of Arrays</td>
<td>19</td>
</tr>
<tr>
<td>2.D</td>
<td>SetAA::contains Complexity Analysis</td>
<td>21</td>
</tr>
<tr>
<td>2.E</td>
<td>Comparing Set Data Structures</td>
<td>22</td>
</tr>
</tbody>
</table>
Problem 1. Short Answer

Carefully plan your solution before starting to write your response. Please be brief and to the point; if at all possible, limit your answers to the space provided.
Part 1.A  Constructors and Destructors

The following class Foo includes a single dynamically allocated integer. **Draw the state diagram that corresponds to the execution of this C++ program.** You must clearly label all variables in your diagram (including any implicit variables), explicitly show all constructors and destructors, and fully expand out the stack frames for all function calls.

```cpp
class Foo
{
    public:
        Foo() { m_p = nullptr; }
        Foo( int v )
        {
            m_p = new int;
            *m_p = v;
        }
        Foo( const Foo& x )
        {
            if ( x.m_p != nullptr ) {  
                m_p = new int;
                *m_p = *x.m_p;
            } else
                m_p = nullptr;
        }
    ~Foo() { delete m_p; }
};

private:
    int* m_p;
};

void bar()
{
    Foo a(3);
    bar();
    Foo b(a);
    return 0;
}

int main( void )
{
    bar();
    return 0;
}
```
Part 1.B Accessor Member Function

The following class Foo includes a single dynamically allocated integer, and the class Bar includes a single dynamically allocated instances of Foo. **Draw the state diagram that corresponds to the execution of this C++ program.** You must clearly label all variables in your diagram (including any implicit variables), explicitly show all constructors and destructors, and fully expand out the stack frames for all function calls.

```cpp
class Foo
{
    public:
    Foo() { m_i = new int; }
    ~Foo() { delete m_i; }
    void set( int v ) {
        *m_i = v;
    }
    private:
    int* m_i;
};

class Bar
{
    public:
    Bar() { m_f = new Foo; }
    ~Bar() { delete m_f; }
    void set( int v ) {
        m_f->set(v);
    }
    private:
    Foo* m_f;
};

int main( void )
{
    Bar bar;
    bar.set( 12 );
    return 0;
};
```
Part 1.C Clock Class (Weight: ×2)

Consider the following C++ program which defines a new class Clock to represent a clock with both an hour (m_hour) and a minute (m_min) field. **Draw the state diagram that corresponds to the execution of this C++ program.** You must clearly label all variables in your diagram (including any implicit variables), explicitly show all constructors and destructors, and fully expand out the stack frames for all function calls.

```cpp
1 class Clock
2 {
3     public:
4         Clock()
5         {
6             m_hour = 0;
7             m_min = 0;
8         }
9         
10         Clock( int hour, int min ) {
11             if ( (hour < 0) || (hour > 12)
12                 || (min < 0) || (min >= 60) )
13                 throw -1;
14             
15             m_hour = hour;
16             m_min = min;
17         }
18         
19         void tick() {
20             m_min++;
21             if ( m_min == 60 ) {
22                 m_hour = m_hour + 1;
23                 m_min = 0;
24             }
25             if ( m_hour == 12 )
26                 m_hour = 0;
27         }
28         
29         private:
30             int m_hour;
31             int m_min;
32         };
33         
34         int main( void )
35         {
36             Clock c0;
37             Clock c1( 9, 30 );
38             c0 = c1;
39             c0.tick();
40             return 0;
41         }
```
Part 1.D  Integers in Sign Magnitude Form

The following class Integer stores integer whole numbers in sign magnitude form (i.e., the $m_{pos}$ field indicates if the number is positive, and the $m_{mag}$ field is the unsigned magnitude). Draw the state diagram that corresponds to the execution of this C++ program. You must clearly label all variables in your diagram (including any implicit variables) and fully expand out the stack frames for all function calls. You do not need to show any constructors or destructors at all in your state diagram! However, you do need to show the execute of constructors and destructors using the execution boxes.

class Integer
{
  public:
    Integer()
    {
      $m_{pos}$ = true;
      $m_{mag}$ = 0
    }
    Integer( int v )
    {
      $m_{pos}$ = ( v >= 0 );
      $m_{mag}$ = $m_{pos} \land v : -v$;
    }
    int get_value() const
    {
      int sign = $m_{pos} \land 1 : -1$;
      return sign * $m_{mag}$;
    }
  private:
    bool $m_{pos}$;
    unsigned int $m_{mag}$;
};

Integer
operator+( const Integer& w, int x )
{
  int y = w.get_value();
  Integer z( x + y );
  return z;
}

int main( void )
{
  Integer a(-3);
  Integer b = a + 2;
  return 0;
}
Part 1.E Filling a Sequence with Zeros

Recall the C++ singly linked list data structure from lecture. The interface and implementation are shown below with one addition. We have added a new size member function to calculate the number of elements currently stored in the linked list.

```cpp
class SListInt {
public:
    SListInt();
    ~SListInt();

    void push_front(int v);
    int size() const;

private:
    struct Node {
        int value;
        Node* next_p;
    };

    Node* m_head_p;
};

SListInt::SListInt() {
    m_head_p = nullptr;
}

void SListInt::push_front(int v) {
    Node* new_node_p = new Node;
    new_node_p->value = v;
    new_node_p->next_p = m_head_p;
    m_head_p = new_node_p;
}

SListInt::~SListInt() {
    while ( m_head_p != nullptr ) {
        Node* temp_p = m_head_p->next_p;
        delete m_head_p;
        m_head_p = temp_p;
    }
}

int SListInt::size() const {
    int num_elm = 0;
    Node* curr_p = m_head_p;
    while ( curr_p != nullptr ) {
        num_elm += 1;
        curr_p = curr_p->next_p;
    }
    return num_elm;
}
```

Consider the following algorithm to fill a SListInt with zeros. Note the list may or may not be initially empty. An example is shown below illustrating the use of this algorithm on a list which initially contains two elements.

```cpp
void fill_with_zeros(SListInt* lst, int N) {
    while (lst->size() < N)
        lst->push_front(0);
}

int main( void ) {
    SListInt lst;
    lst.push_front(42);
    lst.push_front(13);
    fill_with_zeros(&lst, 10);
    return 0;
}
```
What is the time complexity of the `fill_with_zeros` algorithm when applied to `SListInt` as a function of \( N \)? Use asymptotic big-O notation. Use the six-step process described in lecture for complexity analysis to justify your answer. You must explicitly number and identify all six steps to receive full credit.
Part 1.F Nested Virtual Function Calls

Consider the following three classes: Foo and Bar inherit from IBase. Draw the state diagram that corresponds to the execution of this C++ program. You must clearly label all variables in your diagram (including any implicit variables) and fully expand out the stack frames for all function calls. You do not need to show any constructors or destructors at all in your state diagram! However, you do need to show the execute of constructors and destructors using the execution boxes. You must explicitly indicate which version of baz and go is being called.

```cpp
class IBase {
public:
  virtual ~IBase() { }
  virtual void baz(IBase* x) = 0;
  virtual void go() = 0;
};
class Foo : public IBase {
public:
  Foo() { m_d = 0; }
  void baz(IBase* x) { x->go(); }
  void go() { m_d += 1; }
private:
  int m_d;
};
class Bar : public IBase {
public:
  Bar() { m_d = 10; }
  void baz(IBase* x) { x->go(); }
  void go() { m_d += 2; }
private:
  int m_d;
};
int main( void ) {
  Foo a;
  Bar b;
  a.baz(&b);
  Foo c;
  c.baz(&c);
  return 0;
}
```
Part 1.G Dynamic Polymorphic Array of Objects

The code on the following page illustrates using dynamic polymorphism to create an array of two I0bjects. **Draw the state diagram that corresponds to the execution of this C++ program.** You must clearly label all variables in your diagram (including any implicit variables) and fully expand out the stack frames for all function calls. **However, you do not need to show any constructors or destructors at all on the stack or execution boxes!**
You do not need to show any constructors or destructors at all on the stack or execution boxes!

class IObject
{
public:
  virtual IObject* clone() const = 0;
...
};
class Integer : public IObject
{
public:
  Integer( int i ) { m_i = i; }
  IObject* clone() const {
    return new Integer( *this );
  }
...
private:
  int m_i;
};
class Double : public IObject
{
public:
  Double( double d ) { m_d = d; }
  IObject* clone() const {
    return new Double( *this );
  }
...
private:
  double m_d;
};
void append( IObject** objs, int idx, const IObject& obj )
{
  objs[idx] = obj.clone();
}
int main( void )
{
  IObject* objs[2];
  Integer a(5);
  append( objs, 0, a );
  Double b(2.5);
  append( objs, 1, b );
  return 0;
};
Part 1.H Shape Interface Inheritance

Consider the following C++ program which uses an IShape class hierarchy similar to lecture. **Draw the state diagram that corresponds to the execution of this C++ program.** You must clearly label all variables in your diagram (including any implicit variables) and fully expand out the stack frames for all function calls. **You do not need to show any constructors or destructors at all in your state diagram!**

```cpp
class IShape
{
    public:
        virtual void scale( int s ) = 0;
    ...;
};

class Point : public IShape
{
    public:
        Point() { m_x = 0; m_y = 0; }
        Point( int x, int y ) { m_x = x; m_y = y; }
        void scale( int s ) { m_x = m_x*s; m_y = m_y*s; }
    ...
    private:
        int m_x; int m_y;
};

class Circle : public IShape
{
    public:
        Circle() { m_x = 0; m_y = 0; m_r = 0; }
        Circle( int x, int y, int r ) { m_x = x; m_y = y; m_r = r; }
        void scale( int s ) { m_r = m_r*s; }
    ...
    private:
        int m_x; int m_y; int m_r;
};

void scale_shapes( IShape* s0, IShape* s1, int s )
{
    s0->scale( s );
    s1->scale( s );
}

int main( void )
{
    Point a(1,2);
    Circle b(3,4,2);
    scale_shapes( &a, &b, 2 );
    return 0;
}
```
Problem 2. Set Data Structures

In this problem, you will explore two different approaches to implementing a set abstract data type (ADT) for integers. The first approach will use a list of lists, while the second approach will use an array of arrays. Recall that the set ADT should be able to add items to the set and to also query if the set contains an item.

Part 2.A SetLL: Set Implemented with a List of Lists

The first implementation of the set data structure will use a list of lists. An important aspect of this implementation is that it will rely on the notion of a key associated with each integer added to the set. The key $k$ of an integer $v$ is defined as $k = v \% K$ where $\%$ is the C/C++ modulus operator and $K$ is an implementation-defined constant. Recall that the modulus operator essentially returns the remainder of $v/K$. So for example, if $v = 24$ and $K = 10$, then $k = 4$ (i.e., $24/10$ leaves a remainder of 4) and if $v = 30$ and $K = 8$, then $k = 6$ (i.e., $30/8$ leaves a remainder of 6).

The high-level idea for this implementation is shown below. This example uses $K = 10$, and the set contains $\{ 24, 37, 44, 72, 74, 97 \}$. Each node in the outer list contains a key, an inner list of integers, and a pointer to the next node of the outer list. This implementation should preserve the invariant that all integers with the same key are stored in the same inner list. So we use the following steps when adding integers to the set: (1) check to see if the integer is already in the set; (2) if the integer is not in the set, then we iterate through the outer list to see if the integer’s key is already in the set; (3) if the integer’s key is in the set, then we simply push the new integer onto the front of the list corresponding to the integer’s key; (4) if the integer’s key is not in the set, then we add a new node to the outer list and push the new integer onto the front of the list corresponding to this new node.
For the inner list, our implementation will use the standard list data structure for ints which we discussed in lecture. The interface for this data structure is shown below for reference. You should assume the implementation of this list data structure is exactly as described in lecture.

```cpp
class ListInt
{
  public:
    ListInt();
    ~ListInt();
    void push_front(int v);

class Itr
{
  public:
    Itr( Node* node_p );
    void next();
    int& get();
    bool eq( const Itr& itr ) const;

  private:
    Node* m_node_p;

  Itr begin();
  Itr end();

  private:
    struct Node
    {
      int value;
      Node* next_p;
    };  
    Node* m_head_p;

  ListInt::Itr operator++( ListInt::Itr& itr, int ); // postfix (itr++)
  ListInt::Itr& operator++( ListInt::Itr& itr ); // prefix (++itr)
  int& operator*( ListInt::Itr& itr );
  bool operator!=( const ListInt::Itr& itr0, const ListInt::Itr& itr1 );
```
The interface and the private member fields for the set data structure which uses a list of lists is shown below. We have also provided the implementation of the add member function. Study this function closely before continuing with this problem.

```cpp
class SetLL
{
public:
    SetLL();
    ~SetLL();
    void add( int v );
    bool contains( int v ) const;
private:
    // Compile-time constant
    static const int K = 10;
    struct OuterNode
    {
        int key;
        ListInt list;
        OuterNode* next_p;
    };
    OuterNode* m_head_p;
};

void SetLL::add( int v )
{
    // Key is the remainder of v / K
    int key = v % K;
    // Step 1. See if set already contains v
    if ( contains(v) )
        return;
    // Step 2. See if key is in the set
    bool found = false;
    OuterNode* curr_p = m_head_p;
    while ( !found && (curr_p != nullptr) ) {
        // Step 3. Key is in the set, add v to list
        if ( curr_p->key == key ) {
            curr_p->list.push_front( v );
            found = true;
        }
        curr_p = curr_p->next_p;
    }
    // Step 4. Key is not in set so ...
    // ... create new outer node
    // ... add v to the list in new outer node
    if ( !found ) {
        OuterNode* new_node_p = new OuterNode();
        new_node_p->key = key;
        new_node_p->list = ListInt();
        new_node_p->next_p = m_head_p;
        m_head_p = new_node_p;
        // Insert the value into front of new list
        m_head_p->list.push_front( v );
    }
}
```
Implement the `SetLL::contains` C++ member function. Clearly identify any corner cases and choose a reasonable approach to handle those corner cases. While you are welcome to use pseudocode to plan your approach, your final solution must be written using valid C++ syntax.

```cpp
bool SetLL::contains(int v) const {
    // Implementation goes here
}
```
Part 2.B SetLL::contains Complexity Analysis

Assume we have $N$ integers in the set, and that the values of these integers are uniformly distributed across the entire range of integers. Recall that $K$ is a compile-time constant used to calculate the keys. **What is the worst-case execution time and time complexity for the SetLL::contains member function as a function of $N$ and $K$?** Use asymptotic big-O notation for time complexity. Justify your answer.

As before, assume we have $N$ integers in the set, and that the values of these integers are uniformly distributed across the entire range of integers. **What is the space usage and space complexity of this data structure as a function of $N$ and $K$?** Use asymptotic big-O notation for space complexity. Justify your answer. In other words, how much space is required to store $N$ unique integers in the set as a function of $N$ and $K$?
In the previous analysis, we assumed $K$ was fixed at compile time. Consider an adaptive variant of this data structure which attempts to ensure $K$ is always roughly proportional to $\sqrt{N}$. To implement the adaptive variant, we would augment the SetLL::add member function to periodically check to see if $K$ is much smaller or larger than $\sqrt{N}$. When $K$ is far from $\sqrt{N}$, the SetLL::add member function will create a new list of lists and then copy the contents of the current set into the new list of lists using $K = \sqrt{N}$. The adaptive variant will have the property that $K$ will always be roughly proportional to $\sqrt{N}$. What is the worst-case execution time and time complexity for the SetLL::contains member function for the adaptive variant of this set data structure? Use asymptotic big-O notation for the time complexity. Justify your answer. Again, you must assume that the $N$ integers currently in the set are uniformly distributed across the entire range of integers. You can ignore the time it takes to periodically create a new list of lists and copy the integers into this new list of lists.
Part 2.C SetAA: Set Implemented with an Array of Arrays

The second implementation of the set data structure will use an array of arrays. This implementation will also rely on the notion of a key associated with each integer added to the set. The key \( k \) of an integer \( v \) is defined as \( k = v \mod K \), exactly as in the previous parts of this problem. This part will also assume that we only wish to add positive integers to the set, and that we know at compile time the maximum integer \( (M - 1) \) we might ever want to add to the set. So there are three key parameters or variables we will need to take into consideration: \( N \) is the number of integers currently in the set, \( K \) is used to calculate the key for any given integer, and \( M - 1 \) is the maximum integer we might ever want to add to the set.

The high-level idea for this implementation is shown below. This example uses \( K = 10 \) and \( M = 100 \), and the set contains \{ 24, 37, 44, 72, 74, 97 \}. There is one element in the data structure for every possible integer we might ever want to add to the set. The value of the element is true if the corresponding integer is in the set and false if the corresponding integer is not in the set. Each element in the outer array is essentially an inner array. There are \( K \) elements in the outer array and each of the \( K \) inner arrays have \( M/K \) elements. You can assume that \( M \) is evenly divisible by \( K \).

The interface and the private member fields for the set data structure which uses an array of arrays is also shown below.

```cpp
class SetAA {
    public:
        SetAA();
        ~SetAA();
        void add(int v);
        bool contains(int v) const;
    private:
        // Compile-time constants
        static const int M = 100;
        static const int K = 10;
        bool m_data[K][M/K];
};
```
Implement the `SetAA::add` and `SetAA::contains` C++ member functions. Clearly identify any corner cases and choose a reasonable approach to handle those corner cases. While you are welcome to use pseudocode to plan your approach, your final solution must be written using valid C++ syntax.

```cpp
void SetAA::add( int v )
{
    assert( (v >= 0) && ( v < M ) );
}
```

```cpp
bool SetAA::contains( int v )
{
    assert( (v >= 0) && ( v < M ) );
}
```
Part 2.D SetAA::contains Complexity Analysis

Assume we have \( N \) integers in the set, and that the values of these integers are uniformly distributed across the range from 0 to \( M - 1 \). Recall that \( K \) is a compile-time constant used to calculate the keys and that \( M - 1 \) is the largest possible integer we might ever want to add to the set. What is the worst-case execution time and time complexity for the SetAA::contains member function? Use asymptotic big-O notation for time complexity. Justify your answer.

Assume we have \( N \) integers in the set, and that the values of these integers are uniformly distributed across the range from 0 to \( M - 1 \). What is the space usage and space complexity of this data structure? Use asymptotic big-O notation for the space complexity. Justify your answer. In other words, how much space is required to store \( N \) unique integers (in the range 0 to \( M - 1 \)) in the set as a function of \( N \), \( K \), and \( M \)?
Part 2.E Comparing Set Data Structures

In this problem, you will be qualitatively comparing various search algorithms. **Begin by filling in the following table based on your analysis in this problem.** The time complexity should be the worst-case time complexity for the `contains` member function assume we have $N$ integers in the set, and that the values of these integers are uniformly distributed across the range of 0 to $M - 1$. Recall that for the adaptive implementation, you can ignore the time it takes to periodically create a new list of lists and copy the integers into this new list of lists. The space complexity should capture how much heap space is required to store $N$ unique integers in the set. Your answers should be in terms of $N$, $K$, and $M$ as appropriate. For the `SetL` row, assume we implement the set using a single linked list as discussed in lecture.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SetL</code> (one list)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>SetLL</code> (list of lists, not adaptive)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>SetLL</code> (list of lists, adaptive)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>SetAA</code> (array of arrays)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Use these results along with deeper insights to perform a comparative analysis of these set data structures, with the ultimate goal of **making a compelling argument for which data structure will perform better across a large number of usage scenarios.** While you are free to use whatever approach you like, we recommend you structure your response in several paragraphs. The **first paragraph** might discuss the performance of `contains` for all four data structure using time complexity analysis. Remember that time complexity analysis is not the entire story; it is just the starting point for understanding execution time. The **second paragraph** might discuss the heap space usage of all four data structures using space complexity analysis. Remember that space complexity analysis is not the entire story; it is just the starting point for understanding space usage. The **third paragraph** might discuss other qualitative metrics such as generality, maintainability, and design complexity. The **final paragraph** can conclude by making a compelling argument for which data structure will perform better in the general case, or if you cannot strongly argue for a single data structure explain why. Your answer will be assessed on how well you argue your position.