ECE 2400 Computer Systems Programming, Fall 2021
Section 10: C++ Problem-Based Learning

revision: 2021-11-23-13-47

In the following problems, we will explore a data structure suitable for use in managing user information in an operating system. More specifically, we wish to design a data structure to track group membership. Each *user* can be a member of one or more *groups*, and the operating system uses group membership to control access to machines, directories, and files. For example, the following commands will display which groups you are a member of on `ecelinux`:

```
% whoami
cb535
% groups | tr -s ' ' '\n'
pug-cb535
en-cs-linux-users
en-ec-faculty
en-ec-instructional-linux
en-ec-ece2400-administrators
```

Membership in the `en-ec-instructional-linux` group enables access to the `ecelinux` servers. Our data structure should enable quickly checking to see if a given user is in a given group, while also providing efficient storage since there can be many users managed by the operating system. We will take an incremental approach. First, we will review an object-oriented string, and then we will explore a set of strings data structure. Finally, we will leverage the set of strings data structure to implement a user database data structure.

Problem 1. Object-Oriented String

Recall from lecture the object-oriented string class. This class elegantly applies the scope-based resource management pattern to automatically allocate space on the heap for a string, copy/assign strings correctly, and then eventually delete a string. Here is simplified version of the interface:

```
class String
{
 public:

  String();
  String( const char* s );
  String( const String& str );
  ~String();

  String& operator=( const String& str );

  bool operator==( const String& str1 );

 private:
  // implementation defined
};
```

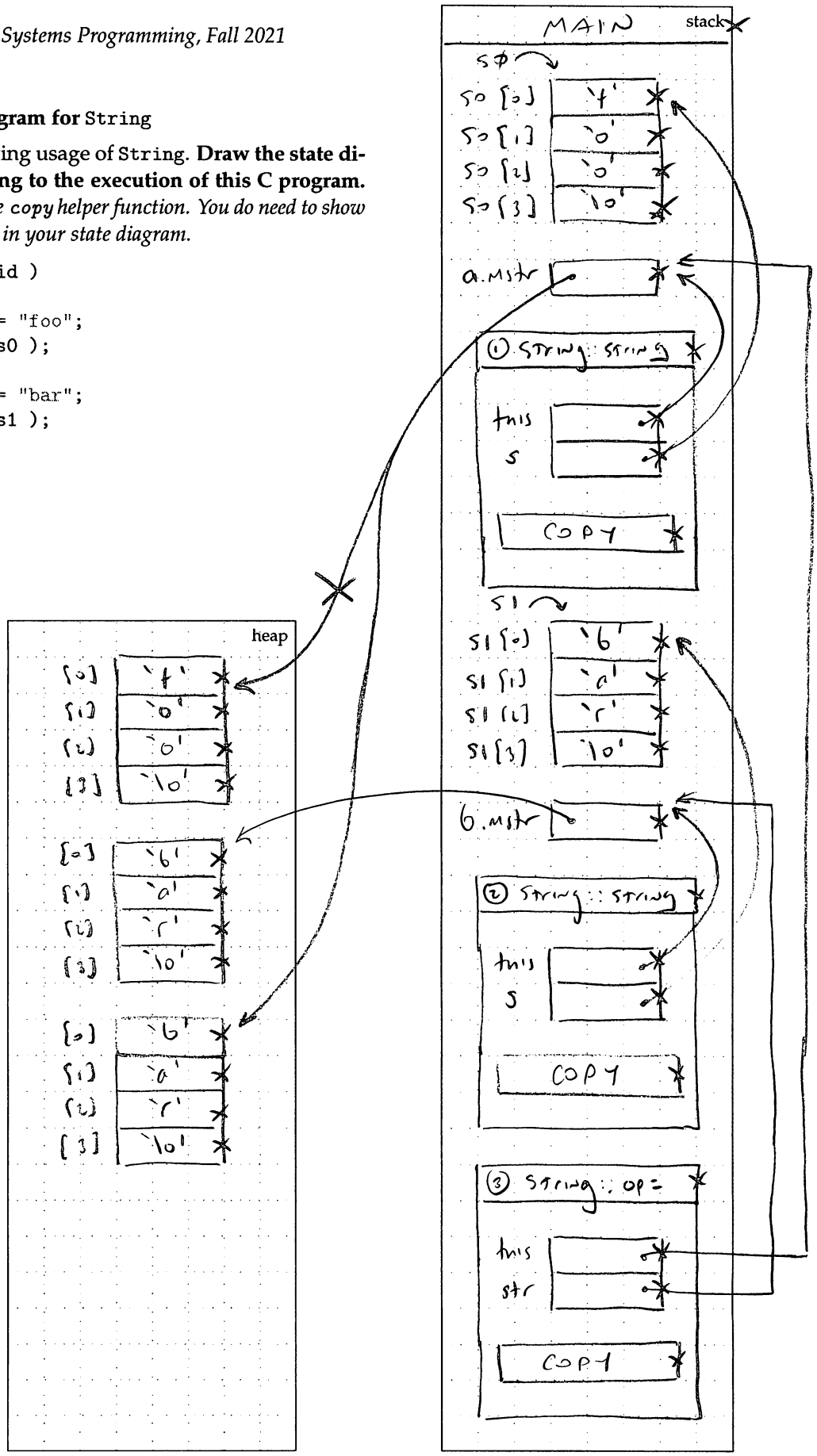
The implementation will have one data member: a pointer to an array of characters allocated on the heap. We will define a default constructor and a non-default constructor that initializes our string from an array of characters. Because we are allocating data on the heap, we will need to use the rule of three. In other words, we need to implement a destructor, copy constructor, and overloaded assignment operator. We also overload the equality operator, which could be implemented in a similar fashion to our work in an earlier discussion section. An implementation for the string class is included below.

```
class String
{
 public:
  String()                    { m_str = new char[1]; m_str[0] = '\0'; }
  String( const char* s )     { copy( s );         }
  String( const String& str ) { copy( str.m_str ); }
  ~String()                   { delete[] m_str;    }

  String& operator=( const String& str )
  {
    if ( this != &str ) {
      delete[] m_str;
      copy( str.m_str );
    }
    return *this;
  }

  bool operator==( const String& str )
  {
    // ... from earlier discussion section ...
  }

 private:

  void copy( const char* s )
  {
    // Determine how many characters are in given String
    int i = 0;
    while ( s[i] != '\0' )
      i++;

    // Size to allocate is one more since we need the null terminator
    int size = i + 1;

    // Dynamically allocate space for copy
    m_str = new char[size];

    // Copy the given String to the newly allocated String
    for ( int i = 0; i < size; i++ )
      m_str[i] = s[i];
  }

  char* m_str;
};
```

Part 1.A State Diagram for `String`

Consider the following usage of `String`. **Draw the state diagram corresponding to the execution of this C program.** *Do not expand out the copy helper function. You do need to show any destructors at all in your state diagram.*

```
int main( void )
{
  char s0[] = "foo";
  String a( s0 );

  char s1[] = "bar";
  String b( s1 );

  a = b;
  return 0;
}
```



Problem 2. Set of Strings Data Structure

Let's use our object-oriented string class to create a new set of strings data structure. The class declaration is as follows:

```
1  class SetStr
2  {
3   public:
4    SetStr();
5
6    void add( const String& str );
7    bool contains( const String& str );
8
9   private:
10   int    m_size;
11   String m_data[16];
12 };
```

The `m_size` variable is used to store how many strings are currently in the set. This simple implementation has a fixed limit of 16 strings, but we could also create a resizable set of strings where the internal array grows dynamically.

Part 2.A Implementing `SetStr`

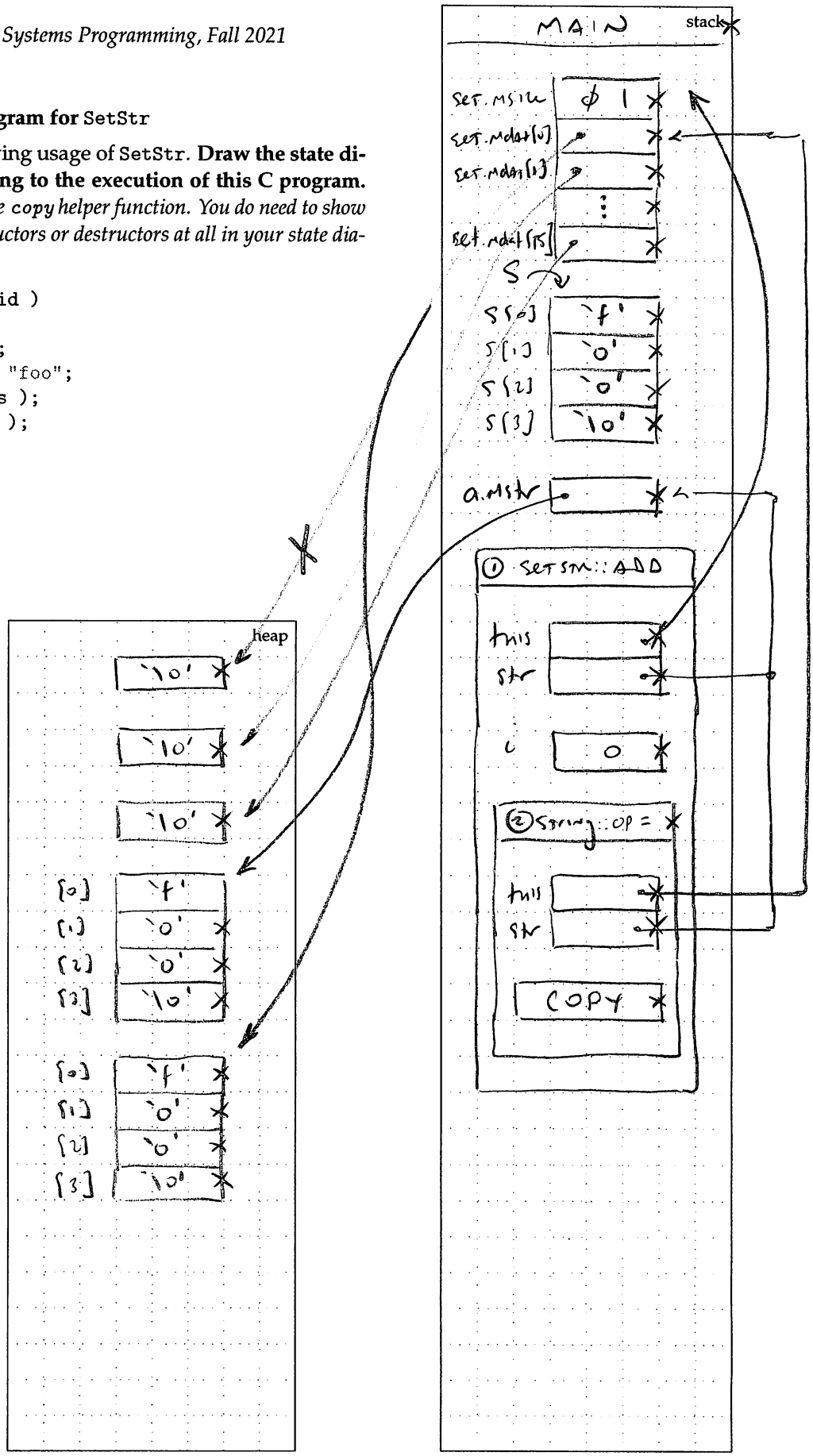
Implement the constructor, `add`, and `contains` member functions in the `SetStr` class. If there is no room in the set then throw the integer 42 as an exception. Your implementation of `add` should only add a new string if that string does not already exist in the set.

```
SetStr::SetStr()
{
  m_size = 0;
}

void SetStr::add( const String& str )
{
  if ( m_size == 16 )
    throw 42;

  for ( int i = 0; i < m_size; i++ ) {
    if ( m_data[i] == str )
      return;
  }

  m_data[m_size] = str;
  m_size++;
}

bool SetStr::contains( const String& str ) const
{
  for ( int i = 0; i < m_size; i++ ) {
    if ( m_data[i] == str )
      return true;
  }
  return false;
}
```

Part 2.B State Diagram for `SetStr`

Consider the following usage of `SetStr`. **Draw the state diagram corresponding to the execution of this C program.** *Do not expand out the `copy` helper function. You do need to show any show any constructors or destructors at all in your state diagram.*

```
int main( void )
{
  SetStr set;
  char s[] = "foo";
  String a( s );
  set.add( a );
  return 0;
}
```



Part 2.B Complexity Analysis for `SetStr`

We can generalize our `SetStr` data structure so it is resizable and can store any number of strings. Assume we wish to store N strings in the set and that each string has K characters. **Fill in the following table with the worst-case execution time and time complexity of the `SetStr::contains` member function and the inherent space usage and space complexity of the `SetStr` data structure. Justify your answer.** Your execution time and space usage should be a function of both N and K.

		`SetStr`
`contains` execution time	$T_K(N)$	KN
`contains` time complexity	big-O	$O(N)$
space usage	$T_K(N)$	KN
space complexity	big-O	$O(N)$

Problem 3. User Database Data Structure

Now consider a new user database data structure with the following interface.

```
1  class UserDB
2  {
3   public:
4
5    void add_user( const String& user );
6    void add_group( const String& group );
7    void add_user_to_group( const String& user, const String& group );
8    bool is_user_in_group( const String& user, const String& group );
9
10  private:
11
12   struct GroupInfo : public IObject
13   {
14     GroupInfo* clone() const;
15     String name;
16     SetStr users;
17   };
18
19   SListIObj m_groups;
20
21 };
```

We want to be able to add users to the data structure, add groups to the data structure, add users to a specific group, and check to see if a user is in a specific group. The above implementation uses a `GroupInfo` struct to store the group name (`String`) along with a user set (`SetStr`) containing the users in that group. The `is_user_in_group` function will iterate through the list of groups, find the correct group, and check to see if the given user is a member of the group using the corresponding user set.

Assume there are G unique groups. Each group has N members. Each user name has K characters and each group name also has K characters. **Fill in the following table with the worst-case execution time and time complexity of the `is_user_in_group` member function and the inherent space usage and space complexity of the `UserDB` data structure. Justify your answer.** Your execution time and space usage should be a function of N, K, and G. Your time and space complexity should be with respect to the number of users per group (N), since we want to understand how the execution time and space usage grow with larger numbers of users.

		`UserDB` (unoptimized)
`is_user_in_group` execution time	$T_{K,G}(N)$	$GK + NK$
`is_user_in_group` time complexity	big-O	$O(N)$
space usage	$T_{K,G}(N)$	$GK + GNK$
space complexity	big-O	$O(N)$

For the execution time, we iterate through the items in the list checking to see if the group name matches the given group name. This requires GK time, since in the worst case we must search through all G items in the list, and in the worst case, each comparison requires comparing K characters. Once we have found the right group, we then must iterate through N user names in the user set, each of length K. This requires NK time in the worst case when the user is the last in the user set. So the total execution time is $GK + NK$. For the space usage, we require GK space to store every group name, and there are G groups each with N members each with K characters requiring GNK space. Thus the total space usage is $GK + GNK$.

Users are usually members of many different groups, meaning there can be significant redundancy in this data structure. The same user name can be stored many times in different user sets. Let R be a number between 0–1 representing a *redundancy factor*. If R is close to one, then there is significant redundancy with very few users in almost every group. If $R = 0$, then there is no redundancy and every user is in exactly one group. Assume we know that we know ahead of time there will be significant redundancy. **Propose a new optimized data structure that can reduce the space usage of `UserDB` by exploiting this redundancy. Fill in the corresponding column in the following table. Justify your answer.** Your execution time and space usage should be a function of N, K, R, and G. Your time and space complexity should be with respect to the number of users per group (N), since we want to understand how the execution time and space usage grow with larger numbers of users.

		`UserDB` (optimized)
`is_user_in_group` execution time	$T_{K,R,G}(N)$	$(1-R)GNK + GK + N$
`is_user_in_group` time complexity	big-O	$O(N)$
space usage	$T_{K,R,G}(N)$	$(1-R)GNK + GK + GN$
space complexity	big-O	$O(N)$

To exploit this redundancy, we can use a map ADT to map user names to an integer user ID, and then we can store just the user ID in each user set. The user IDs will be much more compact than storing the entire user name. The number of entries in the map is $(1-R)GN$ and depends on R. The closer R is to one then there is more redundancy, and thus the map will be smaller compared to the total number of entries across all user sets (GN). The closer R is to zero then there is less redundancy, and thus the map will be larger compared to the total number of entries across all user sets (GN).

For the execution time, we first must iterate through the user names in the user map, which contains $(1-R)GN$ user names each of length K. This requires $(1-R)GNK$ time in the worst case when we are looking for the final user. Then we must iterate through the items in the list checking to see if the group name matches the given group name. This requires GK time, since in the worst case we must search through all G items in the list, and in the worst case, each comparison requires comparing K characters. Once we have found the right group, we then must iterate through N user names in the user set but now checking the user names is constant time (instead of proportional to K) since we are just checking integer user IDs. This requires N time in the worst case when the user is the last in the user set. So the total execution time is $(1-R)GNK + GK + N$.

For the space usage, we require $(1-R)GNK$ space for the map, GK space to store every group name, and there are G groups each with N members each with an integer user ID requiring GN space. Thus the total space usage is $(1-R)GNK + GK + GN$. This makes intuitive sense. The closer R is to one, then the less space we need to allocate to the map (i.e., there are fewer unique user names). The closer R is to zero, then the more space we need to allocate to the map (i.e., there are more unique user names).

Derive a space-usage break-even point for K as a function of R that captures when using the optimized data structure will result in reduced space usage. Interpret this result.

The break-even point will be when the space use of the unoptimized data structure equals the space usage of the optimized data structure.

$$
\begin{aligned}
S_{\text{unopt}}(N) &= S_{\text{opt}}(N) \\
GK + GNK &= (1-R)GNK + GK + GN \\
K + NK &= (1-R)NK + K + N \\
NK &= (1-R)NK + N \\
K &= (1-R)K + 1 \\
K - (1-R)K &= 1 \\
K - K + KR &= 1 \\
KR &= 1 \\
K &= 1/R
\end{aligned}
$$

The inverse relationship between the number of characters in user names (K) and the amount of redundancy (R) makes intuitive sense. If the user and group names are very long, then there does not need much redundancy to see a significant reduction in space usage. Exploiting each instance of a duplicate user name can save storing a significant number of characters. However, if the user and group names are short, then there will need to be more redundancy to see a significant reduction in space usage.

Note how asymptotic complexity analysis does not capture any of these trade-offs. Both the unoptimized and optimized data structures have a time and space complexity of $O(N)$.