

ECE 2400 Computer Systems Programming, Fall 2021

Section 10: C++ Problem-Based Learning

revision: 2021-11-19-19-31

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:

```
1 % whoami
2 cb535
3 % groups | tr -s ' ' '\n'
4 pug-cb535
5 en-cs-linux-users
6 en-ec-faculty
7 en-ec-instructional-linux
8 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:

```
1 class String
2 {
3     public:
4
5         String();
6         String( const char* s );
7         String( const String& str );
8         ~String();
9
10        String& operator=( const String& str );
11
12        bool operator==( const String& str1 );
13
14    private:
15        // implementation defined
16 };
```

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.

```

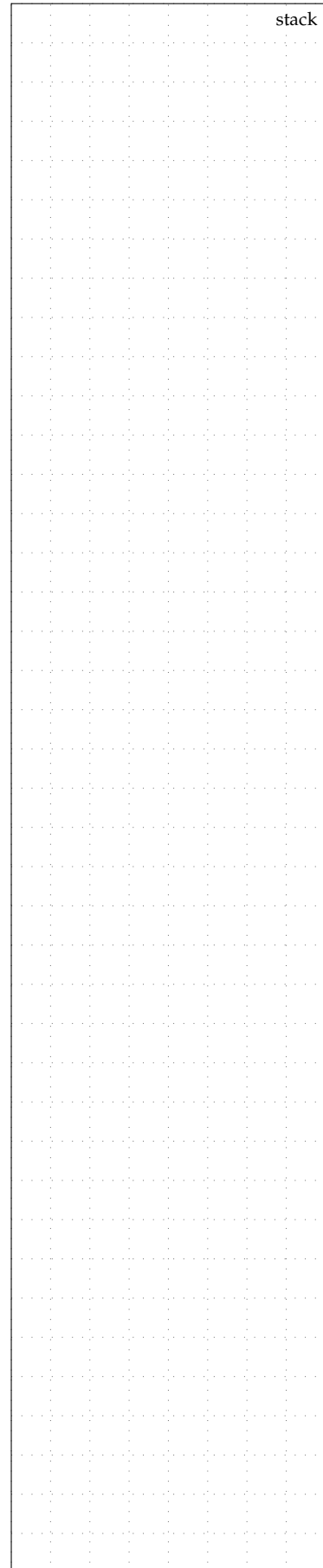
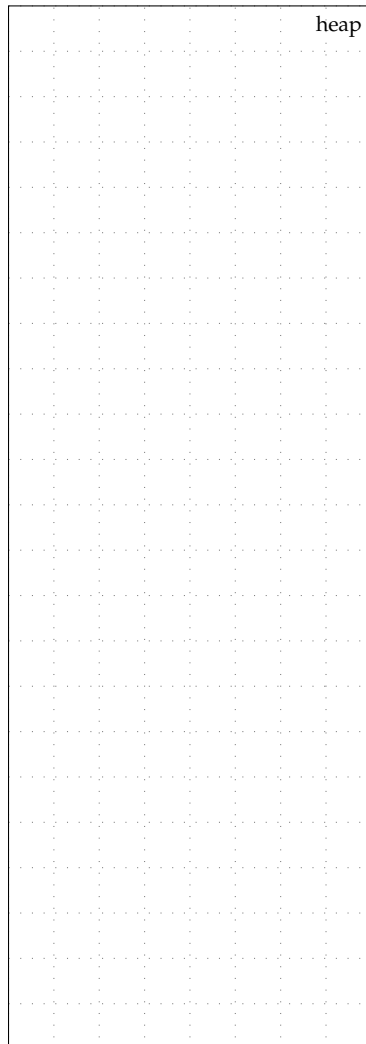
1 class String
2 {
3 public:
4     String()                { m_str = new char[1]; m_str[0] = '\0'; }
5     String( const char* s ) { copy( s ); }
6     String( const String& str ) { copy( str.m_str ); }
7     ~String()               { delete[] m_str; }
8
9     String& operator=( const String& str )
10    {
11        if ( this != &str ) {
12            delete[] m_str;
13            copy( str.m_str );
14        }
15        return *this;
16    }
17
18    bool operator==( const String& str )
19    {
20        // ... from earlier discussion section ...
21    }
22
23 private:
24
25     void copy( const char* s )
26     {
27         // Determine how many characters are in given String
28         int i = 0;
29         while ( s[i] != '\0' )
30             i++;
31
32         // Size to allocate is one more since we need the null terminator
33         int size = i + 1;
34
35         // Dynamically allocate space for copy
36         m_str = new char[size];
37
38         // Copy the given String to the newly allocated String
39         for ( int i = 0; i < size; i++ )
40             m_str[i] = s[i];
41     }
42
43     char* m_str;
44 };

```

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.

```
01 int main( void )
02 {
03     char s0[] = "foo";
04     String a( s0 );
05
06     char s1[] = "bar";
07     String b( s1 );
08
09     a = b;
10     return 0;
11 }
```



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:

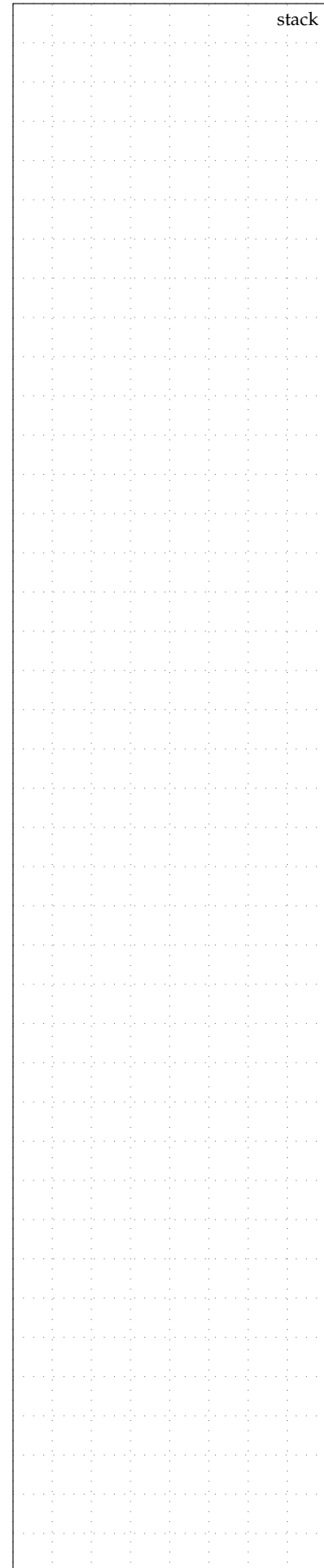
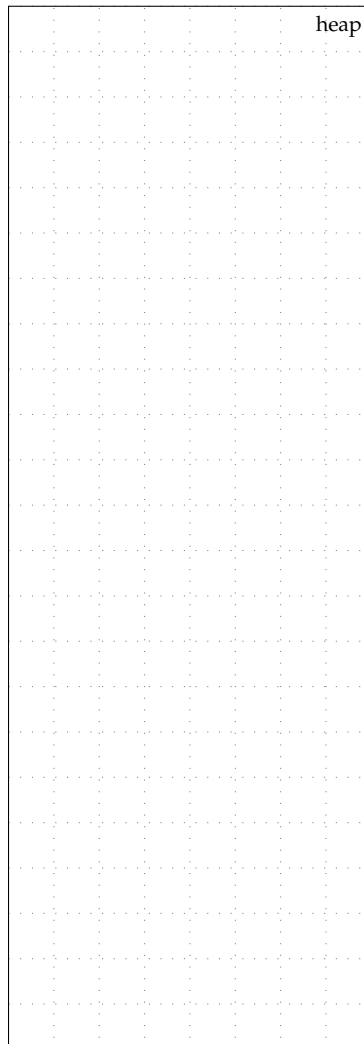
```
1 class SetStr
2 {
3     public:
4         SetStr();
5
6         void add( const String& str );
7         bool contains( const String& str );
8
9     private:
10        int    m_size;
11        String m_data[16];
12    };
```

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.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.

```
01 int main( void )
02 {
03     SetStr set;
04     char s[] = "foo";
05     String a( s );
06     set.add( a );
07     return 0;
08 }
```



Part 2.C 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	$T_K(N)$
contains time complexity	big-O
space usage	$T_K(N)$
space complexity	big-O

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)	
<code>is_user_in_group</code> execution time	$T_{K,G}(N)$
<code>is_user_in_group</code> time complexity	big-O
space usage	$T_{K,G}(N)$
space complexity	big-O

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	$T_{K,R,G}(N)$
is_user_in_group time complexity	big-O
space usage	$T_{K,R,G}(N)$
space complexity	big-O

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.